A Domain-Specific Modeling Approach to the Auto-Generation of VHDL Core Wrappers

by

Erica Jones

A Thesis Submitted to the Faculty of the
Department of Electrical and Computer Engineering
In Partial Fulfillment of the Requirements
For the Degree of
Master of Science
In the Graduate College
The University of Arizona

August 12, 2011
STATEMENT BY AUTHOR

This thesis has been submitted in partial fulfillment of requirements for an advanced degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the head of the major department or the Dean of the Graduate College when in his or her judgment the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

Signed: __________________________

APPROVAL BY THESIS DIRECTOR

This thesis has been approved on the date shown below:

________________________________  _______________________
Dr. Jonathan Sprinkle                Date
Assistant Professor
ACKNOWLEDGMENTS

This research is dedicated to my husband and our new born son.

A special thank you to my thesis professor, Jonathan Sprinkle, and my committee members, Ali Akoglu and Roman Lysecky, for being so accommodating to our growing family’s needs. We appreciate it more than you know.
Abstract

Time to market and the “first time right” solution has become one of the most significant challenges that design teams face today. To help curb this problem, many companies rely on VHDL reuse to help expedite their FPGA design process. While an attractive option, it can be difficult to reuse code that has been developed in an ad hoc fashion. To make design reuse even harder, many companies are leaning towards Commercial Off the Shelf (COTS) products which allow users to provide custom code on commercial hardware. While this may save development dollars when it comes to board design, it can make reuse more difficult as each COTS design usually has its own custom bus interface that it uses to pass data back and forth between software and the FPGA code.

The work in this thesis mitigates the difficult transition between COTS boards by giving developers portability of existing VHDL code between different bus platforms. This allows the end user to be able to reuse already existing VHDL engines and cores (either in house or purchased) by allowing them to be rehosted from hardware platform to platform.

The portability of the interfaces is accomplished through a novel domain specific modeling approach, developed for this thesis. The approach permits developers to model a bus to which their engine core will be attached. This includes routing the ports on the engine to external FPGA pins, connecting them to be driven by bus module registers, and which should be connected to a data FIFO. Once a model is created, a model interpreter synthesizes the necessary VHDL files. Model driven synthesis will only create the raw HDL files and will not do any type of logic synthesis or device targeting. This gives designers complete portability, as output files can be used with any industry standard tool.

Using this modeling and model based synthesis tool offers many advantages to the end user including faster portability between COTS platforms, reduced design time, reduced design costs, and most importantly more flexibility.
# Table of Contents

List of Figures ......................................................... 7

List of Tables .......................................................... 10

Chapter 1. Introduction ............................................... 11

Chapter 2. Background ................................................ 17
  2.1. Very High Speed Integrated Circuit Hardware Description Language (VHDL) ......................................................... 17
  2.2. Automated Firmware Generation .................................. 18
  2.3. Modeling and GME .................................................. 22
  2.4. Wrapper Design .................................................... 27
  2.5. Summary ........................................................... 31

Chapter 3. Approach .................................................. 33
  3.1. Introduction and Objectives ..................................... 33
  3.2. A Sample Domain Example ....................................... 36
  3.3. Metamodel ........................................................ 39
    3.3.1. Atoms of a Core .............................................. 40
    3.3.2. ATOMS OF A BUS MODULE ................................... 45
    3.3.3. Atoms of a FIFO ............................................. 53
    3.3.4. Constraints .................................................. 62

Chapter 4. Implementation ......................................... 63
  4.1. Bus Module ....................................................... 64
  4.2. Bus Package ..................................................... 67
  4.3. Top Module ...................................................... 69

Chapter 5. Results .................................................. 71
  5.1. Simple Example Using Bus Module and Core Models .......... 71
    5.1.1. Simple Bus Model Example ................................ 71
    5.1.2. Core Example Using Xilinx 8b10b Encoder ............... 77
    5.1.3. Putting it all Together .................................... 86
  5.2. Example Using FIFOs ........................................... 87
    5.2.1. Bus Module ................................................ 87
    5.2.2. Core ....................................................... 93
    5.2.3. FIFO ..................................................... 98
  5.3. Output Behaviors ............................................... 107
**Table of Contents—Continued**

5.3.1. Simple Example Using Bus Module and Core Models 107
5.3.2. Example Using FIFOs 108
5.3.3. ModelSim Simulations 109
5.3.4. Synthesis 110
5.4. Application Constraints 111

**Chapter 6. Conclusion**

6.1. Summary 115
6.2. Future Work 115

**References** 117

**Appendix A. Generated VHDL for Bus Module (Example 1)** 121
**Appendix B. Generated VHDL for Bus Package (Example 1)** 133
**Appendix C. Generated VHDL for Top Module (Example 1)** 135
**Appendix D. Generated VHDL for Bus Module (Example 2)** 138
**Appendix E. Generated VHDL for Bus Package (Example 2)** 150
**Appendix F. Generated VHDL for Top Module (Example 2)** 152
### List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 3.1</td>
<td>Example Design Block Diagram Without Data FIFOs</td>
<td>36</td>
</tr>
<tr>
<td>Figure 3.2</td>
<td>Example Bus Module Timing Diagram</td>
<td>37</td>
</tr>
<tr>
<td>Figure 3.3</td>
<td>Abridged metamodel, contained atoms removed for brevity.</td>
<td>39</td>
</tr>
<tr>
<td>Figure 3.4</td>
<td>Constant Atom with Attributes</td>
<td>41</td>
</tr>
<tr>
<td>Figure 3.5</td>
<td>FIFO Signal Atom with Attributes</td>
<td>42</td>
</tr>
<tr>
<td>Figure 3.6</td>
<td>Open Atom with Attributes</td>
<td>43</td>
</tr>
<tr>
<td>Figure 3.7</td>
<td>Register Bits Atom with Attributes</td>
<td>43</td>
</tr>
<tr>
<td>Figure 3.8</td>
<td>Top Level Port Atom with Attributes</td>
<td>44</td>
</tr>
<tr>
<td>Figure 3.9</td>
<td>Address and Data Bus Atom with Attributes</td>
<td>46</td>
</tr>
<tr>
<td>Figure 3.10</td>
<td>Address Bus Atom with Attributes</td>
<td>47</td>
</tr>
<tr>
<td>Figure 3.11</td>
<td>Address Select Atom with Attributes</td>
<td>48</td>
</tr>
<tr>
<td>Figure 3.12</td>
<td>Bus Clock Atom with Attributes</td>
<td>48</td>
</tr>
<tr>
<td>Figure 3.13</td>
<td>Data Bus Atom with Attributes</td>
<td>49</td>
</tr>
<tr>
<td>Figure 3.14</td>
<td>Data Select Atom with Attributes</td>
<td>49</td>
</tr>
<tr>
<td>Figure 3.15</td>
<td>Read and Write Atom with Attributes</td>
<td>50</td>
</tr>
<tr>
<td>Figure 3.16</td>
<td>Read Select Atom with Attributes</td>
<td>51</td>
</tr>
<tr>
<td>Figure 3.17</td>
<td>Start Atom with Attributes</td>
<td>51</td>
</tr>
<tr>
<td>Figure 3.18</td>
<td>Stop Atom with Attributes</td>
<td>52</td>
</tr>
<tr>
<td>Figure 3.19</td>
<td>Write Select Atom with Attributes</td>
<td>52</td>
</tr>
<tr>
<td>Figure 3.20</td>
<td>Asynchronous Clear Atom with Attributes</td>
<td>54</td>
</tr>
<tr>
<td>Figure 3.21</td>
<td>Clock Atom with Attributes</td>
<td>54</td>
</tr>
<tr>
<td>Figure 3.22</td>
<td>Data Atom with Attributes</td>
<td>55</td>
</tr>
<tr>
<td>Figure 3.23</td>
<td>Empty Atom with Attributes</td>
<td>55</td>
</tr>
<tr>
<td>Figure 3.24</td>
<td>Full Atom with Attributes</td>
<td>56</td>
</tr>
<tr>
<td>Figure 3.25</td>
<td>Q Atom with Attributes</td>
<td>56</td>
</tr>
<tr>
<td>Figure 3.26</td>
<td>Read Clock Atom with Attributes</td>
<td>57</td>
</tr>
<tr>
<td>Figure 3.27</td>
<td>Read Empty Atom with Attributes</td>
<td>57</td>
</tr>
<tr>
<td>Figure 3.28</td>
<td>Read Request Atom with Attributes</td>
<td>58</td>
</tr>
<tr>
<td>Figure 3.29</td>
<td>Read Words Used Atom with Attributes</td>
<td>58</td>
</tr>
<tr>
<td>Figure 3.30</td>
<td>Write Clock Atom with Attributes</td>
<td>59</td>
</tr>
<tr>
<td>Figure 3.31</td>
<td>Write Full Atom with Attributes</td>
<td>59</td>
</tr>
<tr>
<td>Figure 3.32</td>
<td>Write Request Atom with Attributes</td>
<td>60</td>
</tr>
<tr>
<td>Figure 3.33</td>
<td>Words Used Atom with Attributes</td>
<td>61</td>
</tr>
<tr>
<td>Figure 3.34</td>
<td>Write Words Used Atom with Attributes</td>
<td>61</td>
</tr>
<tr>
<td>Figure 5.1</td>
<td>Bus Module Example - Bus Clock with Attributes</td>
<td>72</td>
</tr>
<tr>
<td>Figure 5.2</td>
<td>Bus Module Example - Start with Attributes</td>
<td>73</td>
</tr>
<tr>
<td>Figure 5.3</td>
<td>Bus Module Example - Address Bus with Attributes</td>
<td>74</td>
</tr>
</tbody>
</table>
LIST OF FIGURES—Continued

Figure 5.4. Bus Module Example - Write and Read with Attributes ........................................ 75
Figure 5.5. Bus Module Example - Data Bus with Attributes .................................................. 75
Figure 5.6. Bus Module Example - Stop with Attributes ......................................................... 76
Figure 5.7. Example Bus Module Top Level Diagram .............................................................. 77
Figure 5.8. Example Core Model Instantiation – Xilinx 8b10b Encoder ................................ 78
Figure 5.9. Example Core Atom Instantiation – “din” of the 8b10b Encoder ............................ 79
Figure 5.10. Example Core Atom Instantiation – “clk” of the 8b10b Encoder ......................... 80
Figure 5.11. Example Core Atom Instantiation – “ce” of the 8b10b Encoder ......................... 81
Figure 5.12. Example Core Atom Instantiation – “kin” of the 8b10b Encoder ......................... 83
Figure 5.13. Example Core Atom Instantiation – “dout” of the 8b10b Encoder ....................... 83
Figure 5.14. Example Core Atom Instantiation – “kerr” of the 8b10b Encoder ....................... 84
Figure 5.15. Example Core Atom Instantiation – “disp_out” of the 8b10b Encoder .................. 85
Figure 5.16. Complete Core Model of the 8b10b Encoder ................................................... 86
Figure 5.17. High Level Block Diagram of an Example Using FIFOs ..................................... 87
Figure 5.18. Bus Module Timing Diagram of an Example Using FIFOs ................................... 88
Figure 5.19. Bus Module Example – Shared Address and Data .............................................. 89
Figure 5.20. Bus Module Example – Address Select .............................................................. 90
Figure 5.21. Bus Module Example – Data Select .................................................................. 91
Figure 5.22. Bus Module Example – Write Select .................................................................. 91
Figure 5.23. Bus Module Example – Read Select .................................................................. 92
Figure 5.24. Complete Bus Module Model ............................................................................ 93
Figure 5.25. Core Example – Tx Data Register ....................................................................... 95
Figure 5.26. Core Example – Tx FIFO Read Clock ................................................................. 95
Figure 5.27. Core Example – Tx FIFO Read Request ............................................................... 96
Figure 5.28. Core Example – Mode Select ............................................................................ 97
Figure 5.29. Complete Core Model of Data Serialization Module .......................................... 98
Figure 5.30. FIFO Example – Data ..................................................................................... 99
Figure 5.31. FIFO Example – Write Clock ........................................................................... 100
Figure 5.32. FIFO Example – Write Request ........................................................................ 101
Figure 5.33. FIFO Example – Read Clock ........................................................................... 101
Figure 5.34. FIFO Example – Read Request ........................................................................ 102
Figure 5.35. FIFO Example – Q ....................................................................................... 103
Figure 5.36. FIFO Example – Write Empty ......................................................................... 104
Figure 5.37. FIFO Example – Write Full ............................................................................... 104
Figure 5.38. FIFO Example – Read Empty ........................................................................... 105
Figure 5.39. FIFO Example – Read Full ............................................................................... 106
Figure 5.40. FIFO Example – Complete Model .................................................................... 106
Figure 5.41. Simple Example Bus Module Simulation .......................................................... 110
List of Figures—Continued

Figure 5.42. Simple Example – Synplify Timing Performance Summary . . 111
Figure 5.43. Simple Example – Xilinx Device Utilization Summary . . . . 112
LIST OF TABLES
Chapter 1

INTRODUCTION

In this busy world, businesses and engineers are always being pushed to produce quality products better, faster, and cheaper. Not only do bosses and shareholders want their new technology better, faster, and cheaper, they also want it right the first time. Of course the reason for this push is a no brainer: the faster companies can come out with new technology and better products than their competitors, the more money they can make. The more money a company makes, the happier (and wealthier) their shareholders are. However, releasing a sub-quality product with multiple revisions after public release can also create problems not only to a products reputation, but also to a company’s reputation. Therefore, it is imperative that quality products not only come out the door quickly, but they also must be designed in a way that they can be manufactured cheaply so that even the average consumer can afford to have the next latest and greatest high tech advancements.

To achieve these constraints, companies are frequently looking to other companies for help. The age in which each company makes its own designs out of completely proprietary components has passed. For example, cell phone connectors each had their own proprietary charging connectors even five years ago. Companies like Nokia had their own charging cable and Motorola had their own version of the charging cable. However, at the end of the day, the cable still accomplished the exact same thing; it charged a cell phone. However, in the short time to market, cell phone manufacturers have turned to more commercial means such as the mini-USB connector. This not only allows power to the cell phone to recharge the battery but also allows the cell phone to connect to a computer in a standard format. This adaptation saves the company significant time and money because it can take something that is already
standard and proven. This gives them added capability without all the risk of a new, custom design. To put the icing on the cake, customers are happy too, as they have fewer cell phone connectors cluttering up their counter space.

Simple design changes like these are being used by companies every day to help save time and money. It goes without saying that changes like these can also help cut down on redesign costs. Redesign can cost companies serious money both in the design phase and the loss of customer base once the product is on retailer shelves. Just like cell phone, more emphasis is being placed on using standards. These standards are becoming quickly incorporated in “Commercial Off the Shelf” (COTS) products. These products are items that are readily available from manufacturers that are 100% ready to be used as a component in either a system or a design. These are used quite frequently in test stations as most products need to be thoroughly tested before release. For example, in the defense industry, a missile manufacturing facility will create a station to trick the missile hardware into thinking its flying so it can remove the design bugs before performing a live test fire. This is done by stimulating the outputs from a very high level, usually at the interface. Companies purchase these COTS cards that they can write custom code on to emulate their interface protocol to get pertinent data and messages to the missile for testing. These can come in a variety of form factors that include PCI, PCI Express, VME, and PXI. Each card can come with a variety of bells and whistles depending on the output that is needed for a particular unit, test, or design. This greatly reduces costs, because it is much less expensive than spinning a custom hardware card.

The only problem with using standards is that there are so many to choose from. Therefore, from design to design on common products, engineers find themselves working on things that are similar but different. Perhaps, in one design an engineer is trying to put an SDLC interface into COTS PCI card FPGA. Once that is complete, the engineer moves onto the next assignment to find that she is putting an SDLC interface but this time onto a VME card. Three months later, her new task is a
large design that uses a PCI card with a large FPGA but now incorporates both an SDLC, fiber optic link, and a UART. However, a smaller internal bus is used to transport data from interface to interface inside the FPGA. Most likely the same SDLC engine core would be used for each design. However, now there has to be added logic developed to actually link the engine to the bus module that would be responsible for transporting data from the bus to the engine core. This would include the development of data storage devices such as a RAM or FIFO as well as control and status registers to configure and/or control the interface.

When taking a closer look at these buses it is even more apparent how similar, yet different they are. Most data buses contain the same elements. For example, buses come with a clock that all other bus control signals are synchronized. Some buses run fast such as PCI-E that can run 166 MHz. Other buses such as the Industry Pack (IP) Bus which runs either at 8 or 32 MHz. Either way, speed is not important for comparison. Data and processes most likely still run at the clock frequency, one just transfers more data in a shorter period of time. In addition, a data bus must have a way to transport the actual data. Some parallel data buses are 16 bits wide and others are 32 bits wide. However, data buses can really be any size or shape imaginable.

Now, that a way to transport data has been decided, there needs to be a way to signal where that data is going. This is done with an address bus. Again, it can vary from bus to bus, but most buses have an address bus in which the bus master can tell the slave where the data should be stored or which data the master would like to be provided. Some buses even share an address and data bus and have separate discrete lines that signal when the data on the data bus is valid data or a valid address. Furthermore, most bus masters need to have a way to start and end a bus cycle. Starting a bus transaction could be as simple as a master asserting a discrete signal for one clock cycle. In most cases, a cycle is stopped by the bus slave using a discrete stop or acknowledge signal. With that being said, buses can come in all
different kinds of combinations and flavors. However, one thing that they all have in common is a way to transfer data from a source to a destination. Some data buses have the ability to transfer data more quickly and efficiently, but that may not matter in some design applications. Due to this fact, many engineers spend a lot of design time tweaking code slightly so a solution can be found.

In addition to COTS and standardization, most FPGA designers have an engine core library that they rely on to help expedite their designs. In the example used previously, it would be assumed that the same SDLC engine would be used on all three designs. It just so happens that each time the engine needs to be connected to a different type of data bus. Other third company parties have also tried to capitalize on this fact. Major FPGA manufacturers such as Altera and Xilinx have their own reuse library that they offer to their customers. These libraries can provide standard interfaces such as 8b10b, UART, SDLC, etc. Furthermore, other third party manufactures that don’t specialize in making FPGAs are also selling rights to their cores. This is advantageous to designers as these cores are usually proven and well documented. For example, an engineer can instantiate a UART core into their design and read in the vendor documentation how to stimulate it from a top level. This saves a great amount of time as an engineer can have a UART solution that works reliably in a matter of hours versus weeks.

On top of trying to use common parts and reuse, many companies are starting to rely on modeling to help save money. Most people are familiar with the mechanical or civil concept of modeling where engineers and architects will build a small replica of the finished product. However, coding through modeling is being increasingly popular. This allows engineers to model their system instead of writing the actual code. Therefore, when small tweaks and changes need to be made the system changes are easier. This is due to the fact that code is built out of smaller modules that are governed by parameters that build the overall design. One big advantage to model based design is that developers can create their own code from their model using a
technique called model based synthesis. This allows the parser to traverse the model and output code accordingly. This allows multiple versions and/or revisions to be generated extremely quickly with only a few clicks of the mouse.

Another time and money saving technique that many FPGA and VHDL designers have been using is wrapping. This allows designers to take files that already exist and easily make them compatible with files and components that they originally weren’t designed to. For example, let’s say that an engine design has a top level component that is supposed to receive input data serially through three input lines. However, the design that will utilize this component will only take the data in parallel through 15 different signals and buses. The original engine can still be used with a slight modification. A bus wrapper can be written to transform data from how the system will be providing it to how the engine component is expecting it. Some engineers have taken this one step further and automated this process to expedite the time it takes to transform a file into something their system can actually use.

The research presented in this thesis combines the ideas of wrapping and modeling. It will be assumed that the designer already has a working engine core, which can be acquired from any legitimate source. Custom designs and intellectual property (IP) purchased from a third party vendor will be equally supported by the tool developed as a direct result of this research. Using a modeling environment, the end user will be able to take their pre-defined engine and hook it up to a bus module of their unique specifications. They will use the modeling environment to fully describe the inter-workings of their specific bus module. The designer will also be fully responsible for describing to the tool how the bus and engine module will interact with one another. After the interactions have been fully specified, the user will be able to push the “create VHDL” button and get fully synthesizable VHDL as an output of this tool.
Thesis Statement

Through a domain-specific modeling approach, existing VHDL cores can be rehosted efficiently and accurately between hardware platforms.

The approach is driven by bus functionality and system/core interactions, and model based synthesis can generate the necessary module files to program any FPGA device. The result is more consistent than a cloning approach, and reduces the specification time for complex designs.
Chapter 2

BACKGROUND

2.1 Very High Speed Integrated Circuit Hardware Description Language (VHDL)

Back in the early 1980’s there was a need to transmit hardware design data from one entity to another. This allows both vendors and engineers to have a fully defined way of communication between sub-entities within a design. In addition, vendors can sell products that are fully specified to engineers working at different levels of abstraction.

VHDL is a language used to represent a hardware circuit in code, but before it could become mainstream it needed to be formalized. This was accomplished over a period of two years and was championed by the Institute for Defense Analyses.

The language was modeled after hardware circuits that could be reduced to smaller components. Therefore, just as you could buy an IC that performed a particular function, the creators of formalized VHDL created a way that designers could contain that same functionality within a block of code called a component. Each component is comprised of two main sections. The first is a design entity that creates a list of inputs and outputs (I/O) that communicate with logic or devices outside the component. The second is an architecture entity that describes the internal operation or organization of that particular component. These components can then be structurally connected to create networks of components that work together to perform an overall design objective. The functionality of the digital circuit is primarily event driven, usually by a clock [1].

VHDL offers many advantages to its users. One is that it can be synthesized from code into gates on programmed onto Field Programmable Gate Arrays (FPGAs.) Synthesis is the act of taking a hardware description and turning it into optimized
gate level solution. Due to the complexity of large FPGAs this is usually accomplished through Computer Aided Design (CAD) tools. [2] FPGAs have become popular in recent years as they offer a high execution speed that is generally obtained in hardware, and also, are very efficient in incorporating changes as they are reconfigurable like software. FPGAs offer an attractive solution over traditional Integrated Circuits (ICs) because of their multifunction and reconfigurable use [3].

In recent years, companies and engineers have been battling time to market constraints in order to maximize profit. It is well known that reducing design time can significantly increase company’s bottom line. While this may sound like an insignificant problem, it has been quite challenging for engineers over the years to keep up with performance and complexity requirements of new technology and still meet their deadlines. Garry Hughes from IBM was quoted saying that design time must be minimized without compromising efficiency and “first-time-right” requirements [4]. VHDL and the design synthesis tools can help overcome time to market challenges due to its wide variety of use and re-configurability in targeting FPGAs as a design solution. However, as with anything else, there is always room for improvement in this area. Much time and energy has been spent in both universities and private industry to try to reduce the design cycle time for both VHDL and FPGAs.

2.2 Automated Firmware Generation

As transistors have become small, the problems that engineers must solve become more complex. This creates systems that are very large and intricate which take significant of time and money to develop. Non-recurring engineering costs can greatly reduce a company’s bottom line. Therefore, much effort has been made in the recent years to reduce these costs. One of the key elements to development cost reduction is to reduce the amount of design time that it takes for a company to get a product out the door. The easiest way to do this is to reuse code that has already been
developed. Often, companies release multiple versions of the same product that they will continually improve upon from release to release. It only makes sense to use what was previously developed as a path forward for creating a new product when applicable.

However, this isn’t always as easy as it may seem. Frequently VHDL designers do not design for reuse. In the paper “Design for Reusability in VHDL” the authors point out the difficulty in reusing code that hasn’t been developed with reuse in mind. It is their hypothesis that while reuse offers higher productivity of System on a Chip (SOC), reusability for the current design and reusability by others can be two very different things. It is easy to reuse a piece of code on the same project or one that is very similar. However, better reuse is achieved when designers can reuse a logic block on a project that is completely unrelated.

This type of design reuse can only happen when four rules are followed. First, the code needs to be “parameterizable.” In other words, it needs to be able to be configured in multiple ways. Second, the code must also be adequately documented. Without proper documentation it can be difficult for an engineer that isn’t the designer to learn about the code and easily use it. Third, the code must also provide simulation and verification scripts to support integration. A waveform can be key to understanding the interworking of a module. It is also helpful to prove that the code that was received is performing the correct operation and behaving as advertised. This can also help by having simulation files that can be used to integrate the reuse code in with the new code. Lastly, to be reusable the code must provide a way for future updates and maintenance. While it is ideal to only reuse code from a component level, this is not always feasible. Sometimes only a few minor tweaks and changes need to be made in order for an existing piece of code to make its way onto a new system. However, if the code wasn’t structured in a way that it can be easily modifiable this can make it near impossible for a module or component to evolve throughout the years. The four rules of reusability must be followed or else it may be
very difficult or even impossible to have portable VHDL that can be used in different systems that perform different tasks. Unfortunately, the ad hoc design methods that are commonly used make reuse very difficult or almost impossible [5].

Agun and Chang outlined requirements for reusability in 2002, but even almost a decade ago, design reuse is by no means a new concept. In the 90’s auto-generated VHDL became the focal point of many research projects. The goal of all of these projects was to have generated VHDL that could be used from system to system. The main goal in [6] was to take a timing diagram and turn it into a usable piece of VHDL. Timing diagrams were the perfect candidate in this paper, because they already provided a method of exclusively defining the time and data domains. This was considered a great starting point, as most hardware systems today can be thought of as communicating subsystems. This approach was advertised as being useful with telecommunication circuits.

However, this solution never offered a fully automated process. The user needs to guide the tool through “guided bottom up synthesis” in which the user helps build sub-modules one by one. The user must provide a timing graph that describes a pre-defined module description that the tool will verify matches the desired system behavior. In addition to the timing graph, the user must also provide extended computation diagrams are used to specify data computation. In a nutshell, the timing diagrams are used to describe data flow requirements from clock cycle to clock cycle and the computation diagrams describe how the data will be manipulated during each individual clock cycle. Input ports and output ports can also be defined using this tool.

However, the main problem and set back with this implementation is that it is extremely difficult to specify internal module events. These would be things that did not occur in the specification graph. Once these diagrams are established, the tool will then perform the auto-generation of VHDL. The tool uses a language based on process algebra called LOTOS. However, for this application they use LOTOS with
time. From here, they will output VHDL that can be then hand tweaked to arrive at the end product. Although this is a step in the right direction, this method of auto-generating VHDL isn’t completely hands free. After defining the overall timing functionality and port map, there still needs to be some human interaction to close the loop for a useable VHDL component. The main hitch in the design is determining how to code internal functionality such as addition, subtracting, shifting, etc that isn’t defined in the main timing diagram [6].

In another approach, the paper [7] looks to exploit the commonality of the design functionality found in most microprocessors. This is done by creating VHDL by a dedicated description language. The authors created their own high level language that would fully describe the microprocessor interface of a particular design that they then would parse using YACC. It would then go through a synthesizer that was developed as part of the research to generate VHDL. In this new language, the users would define components and registers in the design and then associate them with classes such as “Read/Write” or “Write Only.” They can also associate a specific functionality with each register. Their tool will then go back through and make all the implicit connections between registers and port maps thus expediting a solution. In their analysis, this code generation tool expedites development time, reduces error, quick modification and regeneration ability, and easy generation of large VHDL code [7]. While this could have been a time saving approach, learning a language to code in another language did not catch on.

More graphical approaches have also been used to take an idea and transform it into VHDL. The authors of [8] attempt to model VHDL at a hierarchical level that consists of a tree of instances that are properly connected. Applicable constraints can also be added. Using these specifications along with a class hierarchy, a final instance tree is created. The final instance tree is the last of the diagrams that are created using this synthesis process. From here, the diagrams are transformed into

\[^1\text{YACC a widely used tool for software compiler design.}\]
a list of class and instance notations known as CLINT. These CLINT notations can be transformed directly into VHDL. CLINT class descriptions are turned into VHDL entity declarations and architecture bodies. These files can now be fed into a VHDL synthesis tool such as synopsis. The drawback to this approach is that there is no formal specification that is used to create the VHDL output. While the authors found that this led to speedy code generation, this can be a drawback to most users because informal specification can lead to ambiguities and grey areas \cite{8}.

There have been many other approaches that have been attempted that haven’t had a big impact on the way we do things today. One thing that gained popularity was schematic capture but never really had the popularity that coding in VHDL has. Schematic capture is the art of pictorially representing HDL code and taking that picture and synthesizing it. There was a time that it gained a lot of popularity with hardware designers as it allowed them to produce FPGA code without having to learn the VHDL programming language. Ashish Mathur and Prakash Parikh wrote that an approach where they would make functional blocks comprised of schematic capture elements would enhance reuse and portability \cite{9}. However, there are definite drawbacks to using this code, such as maintainability and portability. Many of these schematic drawings are not portable between commercial tools, hindering reuse, as the output solution can never be used across platforms.

Many automation approaches have been tried since the early 90s. Two decades later, these approaches have not caught on because a “one size fits all” approach doesn’t work. It is extremely difficult to create a tool that will do everything that the VHDL language will do, but better and more efficient.

### 2.3 Modeling and GME

Companies and their employees have been using modeling techniques for decades to streamline their processes. For example, architects routinely make small scale models
of buildings and bridges to help facilitate robust design as well as having something tangible that they can pitch to their clients. Once they have a miniature model, they then can make tweaks to account for problems with preliminary structural testing, incorporate new ideas, or adjust the existing design to better suit their customers’ needs. Car manufacturers also use modeling on a routine basis to aid in their design process. Once a model is created, engineers can perform aerodynamic tests to get a better design or take it to their test group to get public feedback on their new concepts. Companies routinely rely on modeling to come up with a better product design before the product is even built, which can add up to big savings on redesign.

Modeling has also become popular in the coding world. Back in the early 1990’s a group of engineers at Vanderbilt University needed an efficient way to build systems that continuously interacted with their environment. They categorized this need as high performance reactive systems that were complex, concurrent, and non-deterministic thus making these system very difficult to build. To solve this problem, they came built their own modeling environment called “Caddmas” that was comprised of 100 parallel processors that gave them a model based approach to software synthesis [8]. This provided the maneuverability needed to adapt to an environment where software and hardware specifications and requirements were constantly changing. For this engineering team, the model based synthesis tool gave them many advantages such as having multiple aspects therefore hiding the complexity through hierarchy. The domain specific synthesizer allowed them to rebuild their code from any level and allowed them to build different kinds of subsystems in a time efficient manor.

Another advantage to this design process is the ability to rebuild a running system without having to touch anything but relevant components. Once a model was implemented in the Caddmas paradigm, it is then synthesized to produce an executable set of communicating programs across a network of processors. Throughout their experience, synthesizable models greatly reduced design time, but it did not come
without a cost. They had additional overhead caused by the tools that cost them time when they executed as well as additional memory use in addition the extra two minutes that it took the synthesizer to generate a solution. Another setback is that this modeling environment and synthesizing tools were specifically designed and built for their custom application. Therefore, at time of publication, only three Caddmas systems were fully built; while all three worked, necessary support tools were still being developed to support the first station. The engineers of this system would have benefited from having access to generic modeling tools that were not widely popular at the time this system was developed [8].

Other engineers have also pulled the "model based synthesis" thread throughout the last few decades. To bridge the gap between a high level of abstraction functional model from the actual efficient hardware implementation, authors Sander and Jantsch used a generic skeleton computation models for typical design patterns that they then could configure for specific designs. A synthesis application would then infer all the details of the design [10]. Another example [11] used synthesis on models from Unified Modeling Language (UML) to create functional test cases for software under test. They found using this synthesis automation technique that they could achieve significant productivity gain, coherent development process, and version/product independence. Furthermore, Bombieri, Fummi, and Guarnieri came up with a tool that would automate the process of transitioning from transaction level modeling (TLM) using SystemC to register transfer level (RTL) that can actually be turned into logic cells by current commercial tools [12].

A advantage of modeling is that it reduces the complexity of coding an entire system. In the paper by Karsai [13], the problem of programming is complex because it is both algorithmic and structural. The structural side is much more difficult to solve as it involves a lot of complexity management of how the different components of a system interact with each other. This problem of complexity management can be overcome by domain specific visual programming. While many programs support
visual programming, not all support domain specific programming and modeling [13].
In the examples above, all the researchers created an environment domain that was
specific for their problem. One advantage to this approach is that you have a tool
that can significantly decrease your man hours. Unfortunately, this can be very time
consuming developing, debugging, and integrating a tool that can really only be used
for one application.

Therefore, it was important to develop a tool that allowed users to have a generic
and configurable programming environment that allows for domain and paradigm
customization. This is essential for complexity management as much of the minute
details about a system are hidden to the end user. For this to work, the system
needs to be represented by a model or diagram and then can be parsed to create
code that can run on real systems. To gain an executable, the model created by
the end user must undergo model based synthesis. How the model is transformed
into an executable depends on its model interpreter. An interpreter will build a
run time solution based on the objects present in the model and the way they are
interconnected. To accomplish this, the modeling environment should be generic,
easily configurable, and independent from application-domain concepts.

There are six main principles such an environment should follow. These include:
connectivity, hierarchical modeling, multiple aspects, references and associations,
conditionals, and replicators. The solution that was created for a visual program-
ing tool that Vanderbilt University created is called General Modeling Environment
(GME) [14]. In GME, modeling is based on part-whole hierarchies with objects that
can be predefined or user defined. This allows the user to construct a model based
on preexisting models and ones that the user decides to create. The end user is also
able to set attributes that are associated with both kinds of models. GME also gives
the user input and output capability through the use of ports. This allows differ-
ent levels of hierarchy to communicate with one another. In [13], Karsai uses the
example of using model based programming on Intelligent Process-Control System.
GME allowed the author to solve control system problems related to monitoring, control, simulation, and diagnostic problems throughout multiple domains due to GME’s flexible environment. This has the potential to be put to use in such plants as oil refineries and chemical plants [13].

The advantage to using GME is that it is so generic that it can be used in a wide variety of applications. GME has even been used in the auto industry to model hybrid motors and their interactions with other subsystems in an automobile [15]. It was found that using the GME modeling environment the engineers could potentially enhance cost savings, design time, and performance enhancement. In another hybrid example, GME was used with a bond graph approach to represent complex physical systems that allowed them to do complex analysis after the translator was ran [16]. Although they were not able to create visual feedback causality for the end user, they were able to find constraint violations in their designs. In another example, GME was used to deploy reconfiguration of an existing codebase that provides mesh radio capability. One of the biggest advantages of this modeling environment and model based synthesis design was the ability for end users to make quick changes to the code without having to consult the original design engineers [17]. The tool has also proven useful in the petroleum industry by helping provide efficient management of assets. This has improved forecasting, reliability estimation, and real time data flow mapping [18]. GME was also able to support the modeling of a safety critical real time feedback control systems that contained multiple redundancies in its components to reduced design time to prototype an execution model [19]. GME has even been used in non-technical applications such as business. The authors Mohan, Choi, and Min of [20] used GME to make a business metamodel that would attempt to close the gap between business needs and IT. This is only a small fraction of examples of the kinds of meta-models that can be created using GME. These examples just go to prove that due to the flexible meta-modeling environment of GME, that a single modeling environment can fulfill the needs of extremely unrelated domains.
One domain in which modeling is less popular, is in firmware design with languages such as VHDL. Up until recently, most companies rely on their firmware engineers to write application specific code for each design that must be completed. However, more emphasis has been placed on reduced cost and design times. This has turned most companies and engineers towards Commercial Off the Shelf (COTS) products, as well as design reuse either from their own company’s intellectual property (IP) or purchasing code from a third party. The main issue with these techniques is that usually the reused code or COTS products do not completely satisfy all the design requirements and additional modifications must be made to the code. A modeling environment would ease the development time cycle of firmware, as it would allow the designer to configure the code into specifically what he or she needed.

2.4 Wrapper Design

VHDL wrappers have been implemented in the past, none have taken a domain specific approach. A wrapper is a way of encapsulating existing code to either use it in an application that it was not written for, or to optimize it in some fashion. Wrappers and more importantly, VHDL wrappers, have become an increasingly popular research area. For example, test wrappers have been involved in many other different kinds of applications with VHDL and FPGAs. In [21] VHDL wrappers are used to create test access mechanisms to interface with functional cores so that optimizes test time. This helps them save time by reducing scan-in and scan-out time of the wrapped cores. In addition, wrappers were used in [22] to generate a way to simplify communication between two different modules that do not share the same clock domains. The authors were able to use the bus wrapper coupled with a 500 MHz generic clock to make a single time domain so that the different modules could easily communicate. In another example, an open core protocol (OCP) is presented as a way to bridge the communication gap between two IP cores or buses. They found by generated an
OCP VHDL wrapper that they could reduce the power consumption of a system while increasing the speed [23]. This in fact was based on a standard for wrappers. Wrappers have been a key element in taking code and making it work in multiple applications.

To take things one step further, there is much research that is geared more towards automating the VHDL wrapper. In [24] Java is used to create a test tool that will help expedite testing of deeply embedded logic cores located on a system on a chip (SoC). Each test has a test pattern source and a sink that generates and stores responses from the core along with a mechanism for transporting the necessary data to the source and from the sync. It also comes with a core test wrapper that allows the user to select between inputs and outputs of the core for visibility purposes. The authors created an application module that allows users to generate core module test wrappers. In this paper, it was necessary to overcome the growing problem of large test data patterns coupled with extended testing times. The user has the option of selecting a default configuration, serial extraction configuration, and parallel extraction configuration that can be configured to execute internally or externally. The wrapper application module allows the user to specify all the user defined parameters of the test wrapper that will first cut and paste the core code into the output file. Next, internal scan registers will be instantiated that include input, output, and length of registers. After this is complete, the tool will set the order of the boundary scan paths. When it is all said and done, there are four VHDL files (that are fully synthesizable) including a definitions file, multiplexer block, control block, and top level wrapper that can be used along with the original core in an FPGA fabric. It was discovered that this tool greatly reduced the time of scanning techniques. This can lead to significant cost and integration time savings due to the reconfigurability and flexibility of the tool. It also helps by reducing the amount of time that an engineer actually has to touch the code as well as potentially eliminating coding errors by having this testing system automated. This tool is very flexible along with being very simple as it was
originally designed to teach students about wrapper core design and is used at the Slovack University of Technology.

There has also been significant focus placed on intellectual property (IP) core generation in recent years. These cores can be provided by FPGA vendors such as Altera or Xilinx. They can also be purchased from other third party vendors. These IP cores are self contained modules that perform a specific function such as a UART, FIFO, and 8b10b encryptor/decryptor. One of the reasons that IP cores have gained such popularity is that they greatly reduce design time and for most cases have been thoroughly tested and integrated. The authors of [25] use these IP cores and turn them into C function calls that can also be run-time configurable on FPGAs. This is achieved by providing a common wrapper to multiple cores whose main responsibility is to hide timing details to the high level language compiler. The goal of this research is to automate the wrapping process (based on C) through timing information. This system is named Riverside Optimizing Compiler for Configurable Computing or ROCCC. The input to ROCCC is a high level C wrapper where clock cycle delays are represented as function calls. ROCCC will then convert these control flow graphs to data flow graphs that are transformed into scheduled pipeline instructions. This tool may also be used to support run time reconfiguration. Results show that this tool has reasonable execution overhead as well as reasonable execution time [25]. While this is a good start to the automation process, C level programming that synthesizes into HDLs can be very inefficient and problematic. Although it turns IP cores into function calls, this does not take any of the burdens off of the user to interface the core with a bus module or other code that is in the upper hierarchy of the design.

In [26], the authors reduce design time and overall cost of developing embedded systems that are cheap, rich in features, and have a short time to market through high level codesign tools. The paper attacks these issues as well as power consumption and electromagnetic emissions using self timed systems. Unlike most VHDL timing
schemes, there are no global clocks used in this design. Rather, the authors chose to use a hand shaking scheme to achieve all their timing and data exchange requirements. Self timed circuits are becoming more and more attractive to industry due to the fact that they’re modular, have low energy consumption, generate lower electromagnetic emissions, and possess more tolerance to environmental variations. Simulink was used as the high level design tool to get an original model which will then be used to synthesize into VHDL code through their own tool called CodeSimulink. The synchronous data flow feature is used in Simulink to let their tool understand the correct execution order per data dependencies. The author’s tool, CodeSimulink, has the capability to generate both VHDL firmware and C-code for software. This is accomplished by having blocks of data elements in libraries that are configurable for each specific design. This research project takes a known modeling environment, Simulink, and allows users to specify data flow between different logic block elements. Using their own custom tools, they are able to turn that model into VHDL code that can be ran asynchronously on an FPGA.

The paper “Bridge Over Troubled Wrappers” outlines a technique where the authors automate the process of generating HDL descriptions between mismatched IP protocols. They were able to use their tool to create many bridges and wrappers to function inside a SOC. The authors were able to exploit the fact that many SOCs use complex protocols in which to communicate with one another. One problem that the authors faced during their design process was a lack of internal and external standard bus architecture across the cores. For their algorithms to work, the end user must provide a finite state machine (FSM) of the protocol descriptions and the mapping between the protocols for the data buses. The FSM is intended to help the tool capture all the correct interfaces between the two protocols.

The tool uses four main algorithms to accomplish this task. The first algorithm is responsible for computing all the possible actions that the interface can perform. Next, an algorithm is used to find all the valid actions to transition from one interface
to another. After this is complete, an algorithm is used to compute counters that will make sure that no data will get lost inside the wrapper. For example, a counter is incremented each time a data token enters the wrapper and then is decremented when it leaves. To finish this process, all redundant paths are removed from the solution. These algorithms have proven to be useful as they address issues of data width mismatch, data type mismatches, pipelined operations, complex branching, and different clock speeds [27].

2.5 Summary

Many different aspects to VHDL automation have been generated in this section. It is important to note that there have been quite a few approaches that have been taken by researches over the past few decades. In the early 90’s emphasis was placed on techniques that were designed to almost replace VHDL. These research projects attempted to come up with a “one size fits all” approach that is all inclusive. They tried to make tools that would satisfy every need in every application. Some research projects even proposed new languages that would better represent the VHDL language than simply using VHDL itself. Obviously this was not a good technique, because most of the tools that were developed during this time period were left back in 90’s. The main issue that these research projects had was they tried to be have an all inclusive solution that represented VHDL better than just writing the code.

While this was going on, modeling environments also picked up steam. Countless hours were spent developing application specific modeling tools. While the benefits were great as once an environment was developed, code could be quickly generated and regenerated using the tool to fix problems and efficiently move between application platforms. However, this technique was not overly beneficial due to the fact that hours saved during development time were spent actually making the modeling environment and tool. To overcome this obstacle, generic modeling environments increased in
popularity. These modeling environments let users define what is referred to as a metamodel that users could then create their own models from. This metamodel approach was a successful, because it allowed the modeling tool to be generic while giving engineers the capability to create an application specific modeling environment. One such tool that has been heavily used for this approach is Generic Modeling Environment (GME) that was created by researchers at Vanderbilt University.

In more recent years, the VHDL wrapper approach has quickly gained popularity. Wrappers allow engineers to take existing code and optimize it for speed or allow it to be used for applications in which it wasn’t originally written. To take this idea one step further, researches even developed systems in which they could automate the wrapper generation process so that way they could expedite code portability between unrelated systems. The research in this paper contains a hybrid of all these ideas and puts them together to make a system that can generate VHDL from a model that was developed in a generic modeling environment. Instead of having automated VHDL that is all inclusive, this research focuses on being able to rehost VHDL code that already exists to decrease design and integration time.
Chapter 3

APPROACH

3.1 Introduction and Objectives

In recent years, an overwhelming push has most in the tech industry going towards more common platforms with reuse. This is to directly combat the high costs of developing a product before selling it to the public in attempt to make a profit. Significant pressure has been placed on engineers to make things better, faster, and cheaper. Reuse is a great idea in theory, but more managers and engineers alike are realizing that this is an element that has to be taken into account from the beginning of the design phase. It cannot wait until its predecessor is out the door to conclude that and hope there will be 90% reuse for the next biggest and best technological device. There is much time and effort spent planning on reuse and modularizing code such that way it can be used as building blocks in future designs.

That said, designers need a way to expedite their development time. There are many third party companies that are trying to solve this problem. In hardware, for example, engineers may choose to buy COTS technology or integrated circuits (ICs) for a specific element in their design that is already available to consumers on the market. While the average person wouldn’t have much of a use for these devices, other companies can use these products in their designs to save both time and money. For example, it would be highly unlikely that a cell phone manufacturer would make its own SRAM chips to place on its board. Instead, it would be more likely that they would purchase them from another vendor and then place them on the board. The same goes for communication ports. Most likely they would buy a USB port of some type that they could purchase rather cheaply and then integrate to their overall design.
The same logic applies to VHDL and firmware applications. There are thousands of lines of code that run on almost any hi-tech device available these days. These products can range from a microwave to a laptop computer. That said, it does not make sense to start over with every design. Not only would you lose key development time, it would be crippling during integration time where designers would be forced to bring together the components of a system together for the very first time...again. Therefore, strategic code reuse where and when possible makes the most sense.

However, accomplishing this task can be a lot easier said than done. The optimal approach to code reuse is to split the VHDL code up into modules. This is already common industry practice. These modules can then become portable from system to system. However, many times these portable modules are not always a perfect fit. They still need additional padding code to make them work in their new environment. The biggest differences between systems may not be computation or bit manipulation, but rather how data is passed in and out of the device. The purpose of this research is an attempt to make this shift between communication platforms easier and more manageable.

The approach of this research is to use model integrated computing to help manage this issue. It is assumed that the designer already has a working engine core completely developed. This code can be something that has been previously developed by the designer, reuse from a different project in the same company, functions that are generated by a tool vendor such as Xilinx or Altera, or even purchased from a third party vendor. The goal is to use modeling to describe how this engine or core component should be connected to its communication counterpart. The user will even have the option of putting data storage devices (aka FIFOs) in between the core and engine to deal with clock domain crossings. The end user will be responsible for depicting the detailed workings of the data bus. They will also be responsible for describing in the model how the engine is to interact with the bus module registers.

Once the model has been completed, the user will then synthesize the model.
This is done using a graph traversal method that is supported in the chosen modeling tool. The modeling software will be responsible for going through and finding all the different elements of the model as well as their corresponding attributes. This information will be used to generate three pure VHDL source code files. The first file that will be generated will be a package. The package will define generics in the design such as the width of the address and data bus. It will also define other critical elements of the bus. This package will be used by the bus module file that is generated. To save time and effort on the tool development end, this bus module will mostly stay the same from design to design. It will depend on the bus package as previously described to configure this file to work as specified by the model. Lastly, the tool will output a top file. The top file will specify how the bus module and the engine module will communicate with each other. It will also be responsible for hooking up the engine and bus module ports to the top level ports of the design.

The files generated by the modeling environment will be in pure VHDL form. Pure VHDL is the only thing that can be almost guaranteed to be portable from platform to platform. For example, let’s consider using the popular Xilinx CoreGen wizard to generate an 8b10b encryptor. While most likely the developer will get a working encryptor, this code is not portable to any other device targeting tool other than Xilinx. This is due to the fact that the output of this wizard is not the complete VHDL needed to synthesize the design. Instead, it is a generic user interface that will be instantiated in the custom code, a list of generics that configure that module to behave as the user specified, and pointers back to their own proprietary source code. This is to prohibit engineers from using IP purchased through the Xilinx tool to be used on any other FPGA than Xilinx. This may seem devious. However, the Xilinx Corporation is not alone in this practice. Most other vendors have the exact same practice. To get around these issues and to make this tool fully portable, the code that is generated is pure source code, and can be used on virtually any platform.

This brings about the bottom line. Everything that we do today has to be bigger,
faster, better, and right the first time. One way that this can be accomplished for reconfigurable embedded systems is to increase portability between communication interfaces so that engine cores can be reused more easily. In the long run, this is beneficial to both the engineer and the company. Coding simple bus interfaces and connecting them to cores can sometimes be more tedious and tasks. However, using this tool to expedite these tasks, leaves them to do more interesting things that cannot be automated as easily. The bottom line is that companies that use techniques like the one presented in this thesis will be able to develop faster, have bug free code faster, and get their products out the door faster. This will lead them to high reliability products, decreased time to market, and reduce expenses.

3.2 A Sample Domain Example

Many times, in simple designs there is a very generic template that most designs will follow. This simple design will include an engine core and a bus module. It will
not include a data FIFO at this time. Consider an example where an engineer would like to use a Xilinx CoreGen component and hook it directly up to the bus. A high level block diagram can be found in Figure 3.1. This figure has three main modules. The first is the bus module. This module will be responsible for containing all the data registers that will be used in the bus module. It will also be responsible for sustaining all data bus activities. It is important to note that this tool will only create a slave bus interface. This design also contains an engine module that is an 8b10b encoder. Code for this encoder already exists and was obtained using the CoreGen Wizard in the Xilinx ISE tool. Last, this design also contains a top module. This contains all the lower level components of the design (i.e. the engine module and bus module) as well as all the high level ports that will be external to the FPGA.

![Simple Bus Timing Diagram](image)

**Figure 3.2. Example Bus Module Timing Diagram**

For this example, a data bus timing diagram was generated, in Figure 3.2. This is a simple example of a synchronous data bus. This bus contains a clock that all the other control and status signals are synchronized to. This particular bus has a start signal that is exactly one clock cycle pulse that will kick off an active bus cycle. This
bus has a combined read/write select line. The bus master will designate a write cycle when this line is low and a read cycle when this line is high. This bus has separate data and address buses. The bus master will be responsible for placing the desired address on the data bus during an active cycle for the slave to reference. The data bus is bidirectional, but the slave will only place data back onto the bus during a read cycle. The last signal will be a status signal back to the master signaling that the bus cycle been completed. If this a read cycle, the requested data will be placed on the bus at this time.

As stated previously, this tool is designed to take an existing core module and attach it quickly to a new bus module. The Xilinx CoreGen 8b10b Encoder was selected for this example. This module will take an 8 bit unencoded value and change it into an encoded 10 bit value. This standard was original created by IBM so that way there would be an equal number of high bits and low bits in order to counteract the bias in a receiver that can be caused by too many of the same bits in a row. Even though the encoder is generated by Xilinx, it was customized to have the minimum number of inputs and outputs needed for this application. This is a synchronous engine module that is driven by a clock external to the FPGA. It also takes an 8 bit data input that will be specified by the lower 8 bits of a bus module register. It also contains two control signals that will be driven by the lower two data bits of the control register located in the bus module. The “ce” signal is an enable signal and used to gate the output of this module. The “kin” signal is a special character control signal. The “ce” signal will be controlled by bit 1 of the control register. The “kin” signal will be controlled by bit 0 of the control register. There are three output signals that are used by the core module. The first is the 10 bit encoded, parallel data output. There are two status signals that are controlled by this module. The “kerr” is used to indicate a special character error. The “disp_out” signal is used to indicate whether the current disparity of the system is high or low. All three of these outputs will be connected to the top module.
This is a simple example that will be used to demonstrate the capabilities of the auto-generation of VHDL tool. This example will use the autoVHDL paradigm to create a model that will represent this system in the conclusion chapter.

### 3.3 Metamodel

This research uses the General Modeling Environment (GME) developed by a group of engineers at Vanderbilt University. This design choice was made because of the flexibility of GME to support virtually any modeling domain. In GME, users create what is referred to as a metamodel. This metamodel is a model of the modeling language. This metamodel is used to create all the options and rules of the model. It will describe what kinds of things can exist, what attributes they possess, how they can be connected together, and basic rules of what can or cannot exist in the model. Once the metamodel is created, users can then use the metamodel to create a model of their system. This is beneficial as users are not modeling from the ground up but rather from a predetermined set of building blocks. GME is used to create a basic metamodel for this research.

![Abridged metamodel](image)

**Figure 3.3.** Abridged metamodel, contained atoms removed for brevity.
At a top level there are three models that exist within the basic paradigm. These models are FIFO, BusModule, and Core. Each model possesses its own atoms and attributes that can be used to fully describe the individual models. However, it is first important to understand these models from a high level. The BusModule model is used to create a model of a bus. This model should be instantiated when a user wants to define the interworking of a data bus. The modeling environment will use the model to determine exactly how the bus should interact with the outside world. Users will be able to define specific signals, widths of buses, and key elements of the bus cycle. Another model available for instantiation is the FIFO. This FIFO instantiation simply lets the environment know that the user has a FIFO that he or she will be using in the final design and does not create VHDL for a FIFO. Rather, it just instructs the model how the FIFO will be utilized inside the overall design. Lastly, the metamodel includes a model of the core engine. Just like the FIFO, the core engine module does not create a VHDL file for engine functionality. It defines how the engine interacts with other design elements. Using the three elements, a full system can modeled within GME. While there is room for expansion, the current design will only support one core engine and one bus module.

3.3.1 Atoms of a Core

As described above, the user has the option of modeling a core that will be used in the system. It was previously described from a very high level. However, there are many options and features that must be described to make a model complete. The purpose of the individual atoms inside the core model is so that the user can define every signal on the port map of the VHDL module. This is important as if a signal is missing, the code will not be able to compile when the time comes. To complete a model, the user must represent every signal on the engine model with one of the atoms described below. Multiple instantiations of specific atoms are completely
allowed. For example, let’s say that a core has a reset signal that is driven by a top level port. The user would select a `topLevelPort` atom and place it on the screen. The user would then be responsible for defining the specific attributes associated with the `topLevelPort` atom. In this case there are only two. These attributes are `bitWidth` and `signalName`. The bit width is defined to be a non-zero integer number and will correspond to the width of the vector. The signal name is the name as it appears on the port. This is extremely important as the model interpreter will use this name as the port map definition on the core component. If it does not match exactly to what is specified in the design file, there will be problems at compile time and it will not work. The other atoms that are available to the user along with their attributes are described in more detail below.

<table>
<thead>
<tr>
<th>Constant Atom Description</th>
<th>Constant Atom Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>This atom should be used to describe a port on the core component that will be connected to a constant.</td>
<td><code>bitWidth</code>, <code>signalName</code>, <code>binaryValue</code></td>
</tr>
</tbody>
</table>

Attributes: `bitWidth` – Bit width is used to describe how big the vector is. This should be a non-zero integer value.

`signalName` – This should be the name that matches the port on the core component.

`binaryValue` – This should be the value of the constant and correspond to the bit width.

Warnings: This atom should only be used for static input values to the port. Do not
use this atom to represent an output on the port map. In addition, the signal name must match exactly to the one found on the port map of the component. Failure to do so will result in errors at compile time.

(Figure Reference: Figure 3.4)

Figure 3.5. FIFO Signal Atom with Attributes

_FIFOSignal Atom_ Description: This atom should be used to describe a port on the core component that will be connected to a port on the FIFO component.

Attributes:
- `bitWidth` – Bit width is used to describe how big the vector is. This should be a non-zero integer value.
- `signalName` – This should be the name that matches the port on the core component.
- `FIFOSignal` – This should be an enumerated list that the user can select from. The user must select the signal that corresponds to the functionality on the FIFO port map.

Warnings: This atom should only be used where there is a FIFO model instantiated in the top level model. Otherwise, the user will get an error during compile time. In addition, the signal name must match exactly to the one found on the port map of the component. Failure to do so will result in errors at compile time.

(Figure Reference: Figure 3.5)

_Open Atom_ Description: This atom should be used to describe an output port on the core module that is not connected.
Figure 3.6. Open Atom with Attributes

Attributes: **signalName** – This should be the name that matches the port on the core component.

direction – This is the direction of the port. The user may select either “in,” “out,” or “inout” from the drop down menu.

bitWidth – Bit width is used to describe how big the vector is. This should be a non-zero integer value.

Warnings: This designation can only be used with an output port. It violates the rules of VHDL to use this designation with an input port. In addition, the signal name must match exactly to the one found on the port map of the component. Failure to do so will result in errors at compile time.

*(Figure Reference: Figure 3.6)*

Figure 3.7. Register Bits Atom with Attributes
RegisterBits Atom Description: This atom should be used to describe a port on the core module that is connected to a status/control register that is part of the bus module. This can be used with either an input or output port.

Attributes:

bitPosition – This attribute should be used to tell which bits of the register that this signal needs to use.

signalName – This should be the name that matches the port on the core component.

bitWidth – Bit width is used to describe how big the vector is. This should be a non-zero integer value.

registerAddress – Register address is used to describe in the model which address this port receives its data from.

registerType – This tells the modeling environment how to treat the register. This can be defined as a write only, read only, or read/write register.

direction – This is the direction of the port. The user may select either “in,” “out,” or “inout” from the drop down menu.

Warnings: The signal name must match exactly to the one found on the port map of the component. Failure to do so will result in errors at compile time.

(Figure Reference: Figure 3.7)

![Top Level Port Atom with Attributes]

Figure 3.8. Top Level Port Atom with Attributes

TopLevelPort Atom Description: This atom should be used to describe a port on the core component that is connected to a top level port.
Attributes: **signalName** – This should be the name that matches the port on the core component.

**bitWidth** – Bit width is used to describe how big the vector is. This should be a non-zero integer value.

**direction** – This is the direction of the port. The user may select either “in,” “out,” or “inout” from the drop down menu.

Warnings: The signal name must match exactly to the one found on the port map of the component. Failure to do so will result in errors at compile time.

*(Figure Reference: Figure 3.8)*

### 3.3.2 ATOMS OF A BUS MODULE

Just like the core module, the end users can instantiate a model of the bus module. This should be done when uses which to create a bus module VHDL file. Just like the **Core**, the complete interworking of the **BusModule** need to be defined in the modeling environment. However this model is unique from the other two in the sense that it actually does not have code that already exists. In the **FIFO** and **Core** modules, the user is only responsible for defining high level ports of the component and how they must interact with the system. In the **BusModule**, the user is responsible for modeling the functionality of the bus. GME will then take this model and create a VHDL file that represents the modeled behavior.

It is important to note that the **BusModule** model only handles simple bus cases. In no way, shape, or form does this model handle all possible cases that a bus can be. To begin, the **BusModule** must have an address bus and data bus or a combined address/data bus. The user can define these vectors to be any size they need to fit their design. If the address bus and data bus are separate, it will be assumed that the data and address are valid at the same time on the buses. The data will be assumed valid when the start signal is asserted. However, if the address and data buses are
combined, there will have to be two separate signals to tell the bus logic when the data on the combined bus is of type address and another signal to tell the bus logic when the data on the combined bus is of type data.

In addition to the address and data buses, the user must also model how the cycles are specified. In this model there can only be a read cycle or a write cycle. High data density transactions such as Direct Memory Access (DMAs) or burst cycles are not supported in the current design. The user will have two options to model as far as read and write signals are concerned. The first is a combined read or write line. The user will have to select that atom inside the model and then specify which polarity signifies a write cycle and which signifies a read cycle. The second option inside the modeling paradigm is to have a separate read and write signals.

Each atom inside the bus module should be used to describe the overall interworkings of the bus. Each atom that is associated with the BusModule is described in more detail below.

<table>
<thead>
<tr>
<th>AddrAndDataBus Atom</th>
<th>Description : This atom should be used to model a combined address and data bus.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active</td>
<td>Attributes : Active – This should be used to tell whether the signal is an active high or an active low signal.</td>
</tr>
<tr>
<td>SignalName</td>
<td>SignalName – This should be the name that matches the port on the core component.</td>
</tr>
</tbody>
</table>

![Figure 3.9. Address and Data Bus Atom with Attributes](image-url)
Warnings: To use this atom, the address and data bus must be the same data with. This atom should be used in conjunction with the `addrSel` and `dataSel` atoms for properly bus functionality.

(Figure Reference: Figure 3.9)

![AddressBus Atom with Attributes](image)

**Figure 3.10. Address Bus Atom with Attributes**

*AddressBus Atom* Description: This atom should be used to describe an exclusive address bus.

Attributes: `bitWidth` – Bit width is used to describe how big the vector is. This should be a non-zero integer value.

`signalName` – This should be the name that matches the port on the core component.

Warnings: This should *not* be used to describe combined address and data bus. This should be used in conjunction with the `DataBus` atom. It will be assumed that data is valid on the address and data bus at the same time when the cycle is initiated. This atom does not have to have the same bit width as the corresponding data bus.

(Figure Reference: Figure 3.10)

*AddrSel Atom* Description: This atom should be used to determine when the data on the combined address/data bus is of type address.

Attributes: `Active` – This should be used to tell whether the signal is an active high or an active low signal.

`signalName` – This should be the name that matches the port on the core component.
Figure 3.11. Address Select Atom with Attributes

Warnings: This atom should be used in combination with the atoms AddrAndDataBus and DataSel to represent proper bus functionality.

(Figure Reference: Figure 3.11)

Figure 3.12. Bus Clock Atom with Attributes

BusClk Atom Description: This atom should be used to describe the bus clock.

Attributes: signalName – This should be the name that matches the port on the core component.

Warnings: This atom has to be included for proper bus functionality. All other bus module control and status signals will be synced to this clock.

(Figure Reference: Figure 3.12)

DataBus Atom Description: This atom should be used to model an exclusive data bus.
Attributes: `bitWidth` – Bit width is used to describe how big the vector is. This should be a non-zero integer value.

`signalName` – This should be the name that matches the port on the core component.

**Warnings:** This should *not* be used to describe combined address and data bus. This should be used in conjunction with the `AddrBus` atom. It will be assumed that data is valid on the address and data bus at the same time when the cycle is initiated. This atom does not have to have the same bit width as the corresponding address bus.

(*Figure Reference: Figure 3.13*)

**DataSel Atom Description:** This atom should be used to determine when the data on the combined address/data bus is of type data.
Attributes: **Active** – This should be used to tell whether the signal is an active high or an active low signal.

**SignalName** – This should be the name that matches the port on the core component.

Warnings: This atom should be used in combination with the atoms `AddrAndDataBus` and `AddrSel` to represent proper bus functionality. *(Figure Reference: Figure 3.14)*

![ReadAndWrite Atom with Attributes](image)

**Figure 3.15. Read and Write Atom with Attributes**

*ReadAndWrite Atom Description:* This atom is used to describe an instance where the read/write select line is combined.

**Attributes:** **SignalName** – This should be the name that matches the port on the core component.

**WritePolarity** – This should be used to describe whether the write cycle designation is active high or active low.

Warnings: This atom should be the only thing needed to describe a read/write select line that uses one line to describe both states. Do *not* use this atom with `ReadSel` or `WriteSel`.

*(Figure Reference: Figure 3.15)*

*ReadSel Atom Description:* This atom should be used to describe a bus that has separate read and write lines. This is used to represent the read select line.

**Attributes:** **Active** – This should be used to tell whether the signal is an active high or an active low signal.

**SignalName** – This should be the name that matches the port on the core component.
Figure 3.16. Read Select Atom with Attributes

Warnings: To ensure proper bus functionality, this atom should be used in conjunction with the `WriteSel` atom. It should not, however, be used with the `ReadAndWrite` atom.

(Figure Reference: Figure 3.16)

![Figure 3.16. Read Select Atom with Attributes](image)

Figure 3.17. Start Atom with Attributes

*Start Atom* Description: This atom should be used to describe the functionality of the signal that kicks off a bus cycle.

Attributes: `Active` – This should be used to tell whether the signal is an active high or an active low signal.

`SignalName` – This should be the name that matches the port on the core component.

Warnings: This atom must be used in the design!

(Figure Reference: Figure 3.17)
Stop Atom Description: This atom should be used to describe the functionality of the signal that stops the bus cycle. During this time, the data is placed back on the data bus during a read cycle.

Attributes:
- **Active**: This should be used to tell whether the signal is an active high or an active low signal.
- **SignalName**: This should be the name that matches the port on the core component.
- **clkCycleStartToStop**: This attribute is used to know how many cycles to stop the bus cycle after. This is including the clock cycle it takes to assert the clock cycle. Data will be placed on the bus according to this attribute.

Warnings: This atom must be used in the model and the **clkCycleStartToStop** attribute should be greater than one.

(Figure Reference: Figure 3.18)

WriteSel Atom Description: This atom should be used to describe the functionality of the signal that writes to the bus cycle. During this time, the data is placed back on the data bus during a write cycle.

Attributes:
- **Active**: This should be used to tell whether the signal is an active high or an active low signal.
- **SignalName**: This should be the name that matches the port on the core component.

(Figure Reference: Figure 3.19)
**WriteSel Atom** Description: This atom should be used to describe a bus that has separate read and write lines. This is used to represent the write select line.

Attributes: **Active** – This should be used to tell whether the signal is an active high or an active low signal.

**SignalName** – This should be the name that matches the port on the core component.

Warnings: To ensure proper bus functionality, this atom should be used in conjunction with the **ReadSel** atom. It should not, however, be used with the **ReadAndWrite** atom. *(Figure Reference: Figure 3.19)*

### 3.3.3 Atoms of a FIFO

Last, the user has the option of modeling a **FIFO** in the design. Unlike the **Core** and **BusModule** models, the **FIFO** model is not necessary for a complete design. The **FIFO** should be used to buffer data between the engine component and the bus module. Many times this is used for data that is going to be sent out or captured by the engine core. An asynchronous \(^1\) **FIFO** can be used to safely jump clock domains between the engine and the bus module \(^2\). Only the current **FIFO** supports both synchronous and asynchronous **FIFOs**. Just like the **Core**, this model does not generate a **FIFO** in VHDL. This file should already exist. This can be a **FIFO** component that was created by the user or one that is was generated by a commercial tool such as Xilinx or Altera. The different atoms of the model are described in more detail below.

**Aclr Atom** Description: This atom is used to represent an asynchronous clear or a reset signal. This atom can be used with both synchronous or asynchronous **FIFOs**.

Attributes: **SignalName** – This should be the name that matches the port on the core component.

---

\(^1\) Asynchronous FIFOs are those that have separate write and read clock domains

\(^2\) An asynchronous FIFO is one where the write and read logic run at their own separate clock rates.
Figure 3.20. Asynchronous Clear Atom with Attributes

Warnings: None

(Figure Reference: Figure 3.20)

Figure 3.21. Clock Atom with Attributes

**Clk Atom** Description: This atom is used to describe the clock used in this design. This atom should be used with synchronous FIFOs.

Attributes: SignalName – This should be the name that matches the port on the core component.

Warnings: This atom should only be used with a synchronous FIFO.

(Figure Reference: Figure 3.21)

**Data Atom** Description: This atom represents the input data to the FIFO. This atom can be used with both synchronous and asynchronous FIFOs.

Attributes: SignalName – This should be the name that matches the port on the core component.

bitWidth – Bit width is used to describe how big the vector is. This should be a
Figure 3.22. Data Atom with Attributes

<table>
<thead>
<tr>
<th>data &lt;&lt;Atom&gt;&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>bitWidth : field</td>
</tr>
<tr>
<td>SignalName : field</td>
</tr>
</tbody>
</table>

non-zero integer value.

Warnings: None

(Figure Reference: Figure 3.22)

Figure 3.23. Empty Atom with Attributes

<table>
<thead>
<tr>
<th>empty &lt;&lt;Atom&gt;&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>SignalName : field</td>
</tr>
</tbody>
</table>

Empty Atom Description: This atom represents the FIFO empty flag. This atom can only be used with a synchronous FIFO.

Attributes: SignalName – This should be the name that matches the port on the core component.

Warnings: This atom should only be used with a synchronous FIFO.

(Figure Reference: Figure 3.23)

Full Atom Description: This atom represents the FIFO full flag. This atom can only be used with a synchronous FIFO.
Figure 3.24. Full Atom with Attributes

Attributes: SignalName – This should be the name that matches the port on the core component.

Warnings: This atom should only be used with a synchronous FIFO.

(Figure Reference: Figure 3.24)

Figure 3.25. Q Atom with Attributes

Q Atom Description: This atom represents the output of the data FIFO. This atom can be used with both a synchronous or an asynchronous FIFO.

Attributes: SignalName – This should be the name that matches the port on the core component.

bitWidth – Bit width is used to describe how big the vector is. This should be a non-zero integer value.

Warnings: None (Figure Reference: Figure 3.25)
**Figure 3.26.** Read Clock Atom with Attributes

*RdClk Atom* Description: This atom represents the read clock of an asynchronous FIFO.  
Attributes: *SignalName* – This should be the name that matches the port on the core component.  
Warnings: This atom should *only* be used with an asynchronous FIFO.  
(*Figure Reference: Figure 3.26*)

**Figure 3.27.** Read Empty Atom with Attributes

*RdEmpty Atom* Description: This atom represents the empty flag according to the read clock of an asynchronous FIFO.  
Attributes: *SignalName* – This should be the name that matches the port on the core component.  
Warnings: This atom should *only* be used with an asynchronous FIFO.  
(*Figure Reference: Figure 3.27*)
Figure 3.28. Read Request Atom with Attributes

*rdReq Atom* Description: This atom represents the read request for a FIFO. This signal will be used to read data out of the FIFO. This atom can be used with both a synchronous and an asynchronous FIFO.

Attributes: *SignalName* – This should be the name that matches the port on the core component.

Warnings: None

(Figure Reference: Figure 3.28)

Figure 3.29. Read Words Used Atom with Attributes

*rdWrdsUsed Atom* Description: This atom should be used to model the number of words that are currently in the FIFO according to the read clock. This atom is intended for use with an asynchronous FIFO.

Attributes: *SignalName* – This should be the name that matches the port on the
core component.

**bitWidth** – Bit width is used to describe how big the vector is. This should be a non-zero integer value.

**Warnings** : This atom should only be used with an asynchronous FIFO.

(Figure Reference : Figure 3.29)

![Figure 3.30. Write Clock Atom with Attributes](image)

*WrClk Atom* Description : This atom is used to describe the write clock that will be used to drive the write logic of an asynchronous FIFO.

Attributes : **SignalName** – This should be the name that matches the port on the core component.

**Warnings** : This atom should only be used with an asynchronous FIFO.

(Figure Reference : Figure 3.30)

![Figure 3.31. Write Full Atom with Attributes](image)
**WrFull Atom** Description: This atom should be used to represent the FIFO full flag according to the write clock in an asynchronous FIFO.

Attributes: **SignalName** – This should be the name that matches the port on the core component.

Warnings: This atom should *only* be used with an asynchronous FIFO.

*(Figure Reference: Figure 3.31)*

\[
\text{wrReq} \\
\text{<<Atom>>} \\
\text{SignalName : field}
\]

**Figure 3.32.** Write Request Atom with Attributes

**WrReq Atom** Description: This atom is used to describe the signal that will write the data into the FIFO. This can be used with either a synchronous or an asynchronous FIFO.

Attributes: **SignalName** – This should be the name that matches the port on the core component.

Warnings: None

*(Figure Reference: Figure 3.32)*

**WdsUsed Atom** Description: This atom is used to describe the status register that tells the exact number of words inside a synchronous FIFO at any given time.

Attributes: **SignalName** – This should be the name that matches the port on the core component.

**bitWidth** – Bit width is used to describe how big the vector is. This should be a non-zero integer value.
Figure 3.33. Words Used Atom with Attributes

Warnings: This atom should only be used with a synchronous FIFO.
(Figure Reference: Figure 3.33)

Figure 3.34. Write Words Used Atom with Attributes

WrWrdsUsed Atom Description: This atom is used to describe the status register of an asynchronous FIFO that represents the number of words in the FIFO according to the write clock.

Attributes: SignalName – This should be the name that matches the port on the core component.

bitWidth – Bit width is used to describe how big the vector is. This should be a non-zero integer value.

Warnings: This atom should only be used with an asynchronous FIFO.
(Figure Reference: Figure 3.34)
3.3.4 Constraints

The metamodel designer can impose constraints on the modeling environment using GME. These constraints dictate how the models can be created. These constraints can govern which models or atoms must exist in the design and how they interact with each other. For example, when using a metamodel of a car, there most likely would be a constraint that the car needed to have exactly four tires. This constraint is used to make sure the model follows the rules of what elements are necessary to make up a car.

However, in this design there are no constraints that are placed upon the user. This is due to the fact that every design is different and perhaps there is a corner case that the metamodeling environment does not account for. In addition, the code that is generated after model based synthesis is pure VHDL code. These files can then be tweaked afterwards to achieve the full desired effects. There are basic elements of the model that the synthesizer will check for. If these basic elements are missing, there will be a warning message that will be sent to the console to let the user know that these basic elements that the modeling environment expects are missing.

Various warnings that can be generated by not having a consistent model can be encoded as constraints as part of future work.
Chapter 4

IMPLEMENTATION

After the model is fully defined, it is now time for model based synthesis to take place. For this application, utilities and files that are already available with the GME tool were used to help expedite this development process. GME offers a feature called the Bon Extender and Bon Visitor. The tool traverses the models and atoms associated with the metamodel, and generates two .cpp and two .h files that allow the model synthesis developer to easily traverse the models using a series of predefined function calls. This saves a lot of time as effort as Microsoft Visual Studios can pull up a Bon Extender project that is already set up to work with the GME metamodel. To add customized code, there is a predefined area in which to include all the custom code required for that particular metamodel.

To begin, the synthesis code will use the Bon Visitor functions to go out and get the root folder of the project. This root folder will contain all the models and atoms that were generated for a specific model using the predefined metamodel. Once this is complete, there is code that will go through and find all the child models in the design. This will collect all the bus module, FIFO, and core models that the user has defined. From there, GME has provided a set of function calls that can find all the atoms that have been created within a specific Core, FIFO, or Bus Module model. Once this is complete, the specific attributes that have been set for each model can be extracted and used to create the VHDL output files. The specifics of how each model is interpreted is discussed in this chapter.

The output of the interpreter will be comprised of three separate VHDL files. The first is the bus module. This is a generic bus module that will be used for all bus module designs as it is designed to use any combination of bus module atoms. In
order for the bus module to work properly, the bus package is required. The bus package defines the specifics of the bus module. It tells the bus module which atoms were used (or not) in the model so that proper bus function can be replicated. The final VHDL output files is the top file. This file is responsible for pulling together the newly created bus module, core file, and FIFO files (if applicable). The model synthesizer will be responsible for traversing through the remaining models and their associated atoms and determining how everything must be connected together. Each VHDL file is described in more detail in its own individual section of this chapter.

4.1 Bus Module

The bus module file is the only generated file as a result of this tool that is called the AutoVHDL tool that will be the same from design to design. Instead of trying to make a different bus module for every design, it was much easier to make one generic file who’s function is governed by a package. This package will be built from the model and allow the actual bus module code not to change. This code is comprised of all the bus elements necessary for proper functionality. This section of the C++ code prints out a standard bus module file. The code contains a large number of print to file statements that will create the bus module. The rest of the section will discuss how the bus module was implemented.

The top of the bus module file is responsible for defining the comments at the top of the file as well as the different library and package definitions. The comments that are generated at the top of the file are very simple in the fact that it tells the users that this code was generated by the AutoVHDL paradigm using GME and that it should not be modified. After this is complete, the file will then define the generic ieee library calls. This includes the “std_logic_1164.all” and the “numeric_std” sections of the library that common for most all VHDL designs. In accordance with good programming practices and standards, the numeric standard package will be used
instead of the “unsigned” or “signed” packages that have been used in the past.

After the comments and library definitions, an entity declaration is needed to define all the ports that will be associated with the bus module. To ensure this file is generic, all the atoms that are available in the metamodel are included in the top level ports for this module. All these atoms are included, but not all will be used in every design file. To accommodate this fact, some ports will be held at constants or left open in the top module instantiation of the bus module. Each bus module must have a reset that is *not* included in the metamodel. This signal must be driven at the input to the FPGA and assumed to be active high. From there, the bus clock, address bus, start, data select, read/write, read select, write select, address select, stop, address/data bus, and data bus atoms will each have a port in the bus module. All ports except for the data bus, address bus, and address/data bus will be defined as a standard logic signal. The other three signals will be defined as vectors and their width is defined by a constant that will be defined by the package. Last, the internal register addresses will be defined on the port map as well. This contains data that will become control/status data to the rest of the module.

After the entity is complete, the architecture must be defined. To begin, all incoming signals are registered on the rising edge of the bus clock. These signals include the start, data select, address select, read select, write select, read/write, address bus, data bus, and address/data bus. To avoid an active state during a reset, the polarity of each of these signals will be evaluated so that the inactive state will be output when reset is applied.

The next two processes in the bus module deals with the latching the incoming address bus and data bus data. This data can come from either a combined address/data bus or a separate data bus. This is governed by a constant set in the bus package. If it is a separate data bus, data will be captured when the start signal is active. However, if the model defines a combined address/data bus, the data bus will be sampled when the “data select” signal is asserted. In addition to latching the
data bus data, the address contained on the address bus must also be latched. The address bus can be contained in either a separate address bus or a combined address bus. This is also governed by a constant in the bus package. If the address bus is a separate address bus, the address will be captured when the start signal is asserted. Otherwise, the address will be on a combined address/data bus and captured when the “address select” signal has been asserted.

After this is complete, the read/write line must also be captured. The read/write select data can be acquired using any of the three different read/write signals. These signals are used to translate the read/write signal a common internal combined read/write signal. This internal signal designates a read cycle when this signal is high and a write cycle when this signal is low. The three bus module ports are used to drive this internal read/write signal. If the bus package defines a combined read/write signal, it will make sure that this signal is translated so that a write cycle occurs when this signal is low and a read cycle when this signal is high. If the bus package defines separate read and write select lines, the logic will combine them into a single signal that defines a write cycle when low and a read cycle when high.

Next, the start and stop signals are used to know when to kick off and complete a bus module cycle. Using the start signal polarity in the bus package, it will assert an internal bus module signal called “cycle active” when the bus master has kicked off a cycle. Once the bus module has ended a cycle, this internal signal will return to an inactive state. This signal will drive a counter that will allow the module to count the number of clock cycles in an active cycle. Once the amount has been reached, the bus module will assert the stop cycle so that the transaction is terminated. When the stop signal is asserted, requested data will be placed back on the data bus during a read cycle.

Last, the bus module is responsible for generating all the internal registers. These registers will take data off the bus and store it for internal use. These registers will also be used to populate the data bus during a read cycle as well.
Again, this bus module will fit all applications that can be created by the bus module model atoms. It is a generic file that is defined by a set of constants that are located in the bus package file. This allows the bus module to be the same from design to design. This bus module output file is shown in Appendix A.

4.2 Bus Package

The bus module package is the most important part of proper bus module functionality. Without a fully defined bus package, the bus module will basically be useless. To begin, this file starts with a generic set of ieee library calls. After this is complete, each individual constant is created in accordance with the model. The first part of the C++ code is responsible for traversing through the model and finding all the atoms that exist. If the atom exists, the different attributes that are associated with each atom are extract from the model. It is completely acceptable if all possible atoms do not exist in the model. Decisions on what should be included in the bus package will be made based both on what is included and what is not included in the model. Please note that all these constants must exist inside the bus package or else the bus module will not compile. If an atom does not exist, a generic value will be inserted in the package for compilation reasons.

As stated previous, the data bus, address bus, and combined address/data bus are defined to be of type standard logic vector. However, until now, there has been no reference to the actual width of these buses. If the separate address bus exists, the bit width attribute will be used to define this constants. If the separate data bus exists, the bit width attribute will also be used to define the width of the data bus. If the separate address or data buses do not exist, a generic value of 31 will be entered. This value is just included for compilation purposes and will not be used during bus module transactions. If the combined address/data bus is used, the bit width attribute will be used to define the width of the address/data bus. The same
rules apply if the combined address/data bus is not used. A generic width of 31 will be used but it will not be used in active bus transactions.

Next the polarity of the start, stop, read select, write select, data select, address select, and read/write line will need to be defined. Each one of these modules has an attribute that can be set that defines if these signals are active high or active low. If they exist, the active attribute will be used to define these constants inside the bus package. However, if these atoms do not exist in the model, they will be set to active high. If the models do not exist, they are included for bus module compilation purposes only and will not be used in active bus cycles.

After this is complete, the package will define if the address/data buses are combined or separate. If the address and data bus atoms both exist, the package will be configured to use a separate bus architecture. However, if only the combined address/data bus atom exists, the bus package will be configured to use a combined bus architecture. If for some reason, there is some hybrid of the models that has been used, the user will receive an warning message in the GME console. Likewise, the same thing will be done with the write/read select lines. If both the read select and write select atoms exist, the bus package will be configured to operate with separate write and read lines. However, if only the read/write atom exists, the bus package will be configured to operate with a combined read/write line architecture. If for some reason there is a combination of these atoms, the user will get a warning in the GME console and urged to revise their design.

There are three more constants that need to be defined for proper bus functionality. The first is the cycle count that is derived from the stop atom. The cycle count attribute will be used to determine how many cycles the each bus cycle will be. However, the minimum cycle count for each active cycle is four clock cycles. If the integer defined in the stop atom is less than four, the constant will be set in the bus package to four and the user will receive a warning in the GME console. The constant “generate end” will be used to make all the different registers located in
the bus module. This will be determined by using the address bus width. Last, the read/write access of the registers will need to be defined. This will be done by using the registers defined in the core model.

These few constants make up the entire bus package. This package will be able to fully govern the bus functionality of the bus module.

4.3 Top Module

The top module file is the only file in the entire design that is created using all the models. This module is responsible for connecting the bus module with the core and FIFO components if applicable. To begin, this module contains generic comments at the top of the file saying this file was generated using the AutoVHDL tool. It also defines the generic IEEE libraries that are necessary in almost every VHDL design. In addition, it also points to the bus package that is located in the work directory.

To begin, the top module must first define all the top level ports. It is assumed that only the core module and bus module will have signals that enter or leave the FPGA. This section of the C++ code will traverse through the core model. It will find every instance of the open, register bits, constant, FIFO signal, and top level port atoms and store all the associated attributes. Only the atoms that are of type top level port will be included in the top module port map. The code will go through each atom that was defined in the model and define a name, direction, and signal/vector width on the port map. Once this is complete, the code will then go through the bus module atoms again. If a bus module atom exists, it will be included in the port map. If the bus module atom does not exist, it will be excluded and either defined as a constant or left open in the instantiation. After this is complete, the entity finished being defined.

The next part of the code to be created is the architecture. It is assumed that the bus module will be included in every design. Recall that the bus module is the same
from design to design and the correct functionality of this module is defined by the bus package. Therefore, the bus module component will be defined with a series of print to file statements. No extra logic is necessary at this time. After this is complete, the code will go through the core and FIFO models to create the component definitions. Each atom in the core will be evaluated to define the signal name, direction, and bit width. After the core is complete, the same thing will be done with the FIFO model.

Before the architecture can actually start being defined, the internal module signals must be defined. The only internal signals that are used will be the register data and those connecting the core to the FIFO instantiation (if applicable). The C++ code will go through the FIFO Signal toms of the core model and create these internal signals. After this is complete, all the component and internal signal definitions are complete.

Next, the component instantiation will need to be created. The individual atoms of the bus module will be checked to see whether or not they exist. If they do exist, they will be connected to the top module ports. If it does not exist and it is an input to the bus module, it will be defined as zero. However, if it does not exist and is defined as an output or bidirectional port, it will be left open. The core module will instantiation will depend on the individual atoms found in the model. Each atom type will be evaluated and connected appropriately. The same applies for the FIFO instantiations.

After the code transverses through the remainder of the FIFO and Core modules, it should fully define a top module output. The next section describes in great detail some examples using the tool and what should be expecting in the output.
5.1 Simple Example Using Bus Module and Core Models

5.1.1 Simple Bus Model Example

Previously in the approach chapter, a simple FPGA was described as a bus module that will be connected to an 8b10b encoder that was taken from the Xilinx Core Generator. The full explanation of the intended design can be found in the approach chapter. For a high level block diagram of this system, refer to Figure 3.1. The timing diagram for the bus that will be modeled is shown in Figure 3.2. This simple bus consists of a clock, data bus, start, address bus, read/write select, and stop acknowledge signal. All the different aspects of this bus can be modeled using the autoVHDL paradigm. To begin, the user must first create a bus model. This is done by selecting BusModule icon from the parts browser and dragging it into the model. There is only one attribute for the bus model. This is the componentName. In this case, the component was named simpleBus. Now that the BusModule model has been implementing in the model, the user must now double click on the image to further define the aspects of the data bus.

Bus Clock: The first element in this bus is a clock. This clock synchronizes all the other bus control and status signals. This clock will drive all the logic inside the bus module. To instantiate a busClk atom, select the Bus Clock atom icon from the part browser and drag it into the model. The bus clock atom only has one parameter to set. This is the SignalName that will be associated with the clock. In this case, the bus clock was named clk. For more information, refer to Figure 5.1.
Start The next signal in the bus is the start signal. In the timing diagram, the start signal kicks off the beginning of an active bus cycle. This signal will be used to count clock cycles to know when the end the bus cycle with the stop signal. This cycle has two attributes that can be set. The first is the SignalName. In this case, the signal was defined to be start. In addition to the signal name, the Polarity of this signal must also be defined. As shown in the bus timing diagram, this signal is active high. To instantiate a Start atom, select the Start atom icon from the part browser and drag it into the model. After this is complete, the attributes to this atom need to be set and previously described. For more information, refer to Figure 5.2.

Address Bus The address signal will be used to know which address the bus master is trying to communicate with. This is represented in the model by instantiating an address atom. As shown in the timing diagram, this signal is 8 bits wide. To instantiate an address atom, select the address atom icon from the parts browser and drag it into the model. There are two attributes associated with the address atom that must be defined for proper bus implementation. The SignalName was chosen to
Write and Read  This bus has a combined write and read line as shown in the timing diagram. This means that when this signal is one polarity it signifies a bus write cycle. When the signal is held at the opposite polarity, it signifies a bus read cycle. In this case, a bus write cycle is being requested when this signal is low. Oppositely, a bus read cycle is being requested when this signal is high. It is important to remember that this modeling environment supports both a separate and combined read and write line(s). In this case, the atom that needs to be instantiated inside the model is the WriteAndRead atom. The WriteSel and ReadSel atoms are NOT to be used for this design. These atoms should only be used when modeling separate read and write signals. To instantiate a WriteAndRead atom, select the WriteAndRead atom icon from the parts browser and drag it into the model. There are two attributes that need to be set that are associated with this atom. The first is the SignalName. In this case, the signal was named “rdWrLow.” The user must also tell the modeling environment which Polarity signifies a write cycle. In this case, WriteActiveLow

Figure 5.2. Bus Module Example - Start with Attributes
was selected to model the fact that this line is asserted low during a requested write cycle. For more information, refer to Figure 5.4.

**Data Bus** The bus also has a bidirectional data bus that is shown in the timing diagram. This is modeled in the autoVHDL paradigm by instantiating a data bus atom. To instantiate a `DataBus` atom, select the data bus atom from the parts browser and drag it into the model. There are two attributes that must be set so the bus can be generated properly. The first is the `SignalName`. In this case, the signal was named `dataBusBi`. Second, the data `BusWidth` must be defined. In this case, the data bus width was defined to be 16. For more information, refer to Figure 5.5.

**Stop** To end the bus cycles, the bus module will use the stop signal. If the bus master is specifying a read cycle, the requested data will be placed on the bus during this time. To instantiate a `Stop` atom, select the data from the parts browser and drag it into the model. There are three attributes that must be set so the bus can be generate properly. The first is `SignalName`. In this case, the signal name is “ackLow.” The second attribute that must be set is the `Active`. This tells the model interpreter if
Figure 5.4. Bus Module Example - Write and Read with Attributes

Figure 5.5. Bus Module Example - Data Bus with Attributes
the stop signal is active high or active low. In this case “low” was selected from the drop down menu to signal the model interpreter that this signal is indeed active low. The last thing that must be defined is the number of clock cycles from the beginning of the bus cycle to the end. This is done using the attribute \texttt{clkCycleStartToStop}. This is defined to be three clock cycles. For more information, refer to Figure 5.6.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{bus_module_example_stop_with_attributes.png}
\caption{Bus Module Example - Stop with Attributes}
\end{figure}

\textit{Complete Bus Module Model} Now that all the individual atoms have been used to fully define the bus module, the model should resemble the one above. It should contain one atom each of the \texttt{start}, \texttt{readAndWrite}, \texttt{stop}, \texttt{busClk}, \texttt{addressBus}, and \texttt{dataBus}. These atoms are used to define EVERY bus signal that is shown on the timing diagram. Once all the parameters are set, the model interpreter will be able to generate a VHDL file that will properly support this bus functionality. It is important to remember that exactly one atom per signal must be used. If there is not enough or too many, the functionality of the bus module VHDL output file will not be correct. See Figure 5.7.
5.1.2 Core Example Using Xilinx 8b10b Encoder

Just like the bus module example, a simple FPGA was described as a bus module that will be connected to an 8b10b encoder that was taken from the Xilinx Core Generator. The full explanation of the intended design can be found in the approach chapter. For a high level block diagram of this system, refer to Figure 3.1. In this portion of the example, a Core Model must be instantiated into the system to represent this Xilinx 8b10b Encoder. This is done by selecting Core icon from the parts browser and dragging it into the model. There is only one attribute for the Core Model that needs to be set. This is the componentName attribute. In this example, the componentName attribute is set to “encoder8b10b.” This is shown in Figure 5.8.

Once this is instantiated, the user must define the different atoms of the Core model. Once inside the model, each signal on the port map must have their own atom representation. The way this 8b10b encoder was implemented, there are seven...
unique ports. Each port must be represented by either a constant, FIFO signal, open, register bit(s), or a top level port. Each port on the 8b10b encoder component is described below.

*Data In* This signal of the encoder is the 8 bit data input that will be encoded. According to the design in the block diagram, this input will be driven by the bus master through the data bus. Therefore, this port will need to be connected to the bus module. To instantiate a `RegisterBits` atom, select it from the parts browser and drag it into the model. There are six attributes that must be set so the bus can be generate properly. The first attribute that must be set is the `signalName`. This signal name must exactly what’s on the port map in order for everything to be connected properly. In this case, the name is “din”. Next, the `registerType` must be set. This is done from a drop down menu. In this case, the register will be both readable and writable by the bus master so “read/write” will be selected from the
menu. The user will also have to define the registerAddress attribute. In this case, the hexadecimal address will be set at 0x02. After this is complete, the user will have to also set the bitWidth and bitPosition. This will tell the model synthesizer where in the register these bits will live. The lower 8 bits will host the 8b10b data to be encoded. Therefore, the bitWidth and the bitPosition will be set to “7 downto 0”. Last, the direction attribute must be set. In this case, “in” should be selected from the drop down menu. For more details, refer to Figure 5.9.

![Object Inspector for register bits](image)

**Figure 5.9. Example Core Atom Instantiation – “din” of the 8b10b Encoder**

*Clock* This signal of the encoder is the clock that will drive this component. According to the high level block diagram, this signal comes from a top level port. Therefore, a topLevelPort atom must be used to represent this signal. To instantiate a topLevelPort atom, select it from the parts browser and drag it into the model. There are only three attributes that need to be set to accomplish this. The first is the signalName attribute. Again, this attribute must match the name on the port
map of the component in order to ensure correct functionality after model synthesis. In this case, the name of this signal is “clk”. The other attribute that must be set is the **bitWidth** attribute. The last attribute that needs to be set is the **direction** attribute. In this case, “in” should be selected from the drop down menu. This will tell the synthesizer how wide the signal is. In this case, the attribute should be set to 1. For more details, refer to Figure 5.10.

**Figure 5.10.** Example Core Atom Instantiation – “clk” of the 8b10b Encoder

*Chip Enable* This signal of the encoder is the “on/off” switch that will be used to enable and disable the encoder. According to the high level block diagram, this signal should be driven by the bus master. Therefore, it must be connected to the bus module using the **registerBits** atom. To instantiate a **registerBits** atom, select it from the parts browser and drag it into the model. Again, there are six attributes that must be set to fully describe the functionality of this port map signal. The first is the **signalName** attribute that should be set to “ce” per the port map
on the component. Second, the register type must be set by using the `registerType` attribute. This register should be both write and read accessible to the bus master. Therefore, the “read/write” option must be selected from the drop down menu. Next, the register address will need to be defined using the `registerAddress` attribute. In this case, the address will be defined to be hexadecimal 0x04. The `direction` attribute also needs to be set. In this case, “in” should be selected from the drop down menu. To finish, the `bitWidth` and `bitPosition` must be defined. Per the high level block diagram, this bit lives at bit 1 of this register. Therefore, the `bitWidth` should be set to “1” and the `bitPosition` should also be set to “1”. For more details, refer to Figure 5.11.

![Object Inspector](image)

**Figure 5.11.** Example Core Atom Instantiation – “ce” of the 8b10b Encoder

*Special Character* This signal of the encoder is used to describe if the input data is regular data to be encoded or a special character. When this signal is high, the ten bit output data of this module will represent a special character. This signal is
controlled by the bus master and therefore needs to be connected to the bus module. A `registerBits` atom will need to be instantiated to represent this signal on the port map. To instantiate a `registerBits` atom, select it from the parts browser and drag it into the model. Again, there are six attributes that must be set to fully describe the functionality of this port map signal. The first is the `signalName` attribute that should be set the same as the signal name on the component port map. In this case, it should be set to “kin”. Next, the `registerPolarity` attribute should be set to “read/write” using the drop down menu. This will allow the bus master to both read and write this particular address. The `registerAddress` should be set to hexadecimal value 0x04. It will share an address location with the “ce” signal. The `bitPosition` and `bitWidth` must also be set. In this case, the `bitWidth` will set to be “1”. According to the block diagram, this should be at bit 0 of the register. Therefore, `bitPosition` should be set to “0”. It is important to note that this bit will share an address location with the “ce” bit. It is up to the user to make sure that the `registerPolarity` is configured the same and the bits are not conflicting. Finally, the `direction` attribute should be set to “in” by selecting it from the drop down menu. See Figure 5.12.

**Data Out** This signal of the encoder is the 10 bit encoded output of this component. According to the high level block diagram, this signal is routed to the pins of the FPGA. This signal will need to be instantiated as a `topLevelPort` atom. To instantiate a `topLevelPort` atom, select it from the parts browser and drag it into the model. Again, there are three attributes that will need to be set on this atom. The first is `signalName`. In this case, it will need to be named “dout” to exactly match the port name on the component. This is a ten bit signal so the attribute `bitWidth` will need to be set to “9 downto 0”. The `direction` must also be set to “out” by selecting it from the drop down menu. See Figure 5.13.
Figure 5.12. Example Core Atom Instantiation – “kin” of the 8b10b Encoder

Figure 5.13. Example Core Atom Instantiation – “dout” of the 8b10b Encoder
Special Character Error This signal of the component is used to signal when a special character error has occurred. According to the block diagram, this signal is connected to the pins of the FPGA. To accomplish this, this signal on the port map must be represented as `topLevelPort`. To instantiate a `topLevelPort` atom, select it from the parts browser and drag it into the model. This atom has three attributes that should be set to ensure proper functionality. The first is the `signalName`. In this case, it should be named “kerr” to exactly match the name that is located on the port map. The `bitWidth` must also be set. In this case, the signal is only one bit wide so it should be set to “1”. The `direction` must also be set to “out” by selecting it from the drop down menu. See Figure 5.14.

![Figure 5.14. Example Core Atom Instantiation – “kerr” of the 8b10b Encoder](image)

Disparity Out This signal is used to keep track of the current disparity of the encoder. If this signal is high, the next data value will be encoded as a positive value. If this signal is low, the next data value will be encoded as a negative value. According
to the high level block diagram, this port should be connected to the pins of the FPGA. To accomplish this, this signal must be represented by a `topLevelPort` atom. To instantiate a `topLevelPort` atom, select it from the parts browser and drag it into the model. This atom has three attributes that should be set to ensure proper functionality. The first is the `signalName`. In this case, it should be named “disp_out” to exactly match the name that is located on the port map. The `bitWidth` must also be set. In this case, the signal is only one bit wide so it should be set to “1”. The `direction` must also be set to “out” by selecting it from the drop down menu. See Figure 5.15.

![Figure 5.15. Example Core Atom Instantiation – “disp_out” of the 8b10b Encoder](image-url)

*Complete Core Model* Now that all the individual atoms have been used to fully define the 8b10b encoder, the model should resemble the one above. It should contain one atom each of the `din`, `kin`, `ce`, `clk`, `dout`, `kerr` and `disp_out`. These atoms are used to define EVERY signal that is shown in the high level block diagram. Once all the
parameters are set, the model interpreter will use this model to know how to construct
the top level module. In this case, the top level module will be constructed to connect
the bus module and 8b10b encoder modules together. It is important to remember
that no code will be generated from the Core model instantiation. Rather, it will
assume that this module already exists and provide provisions for it to communicate
with the previously defined bus module. After it this is complete, there should be
seven atoms instantiated within the model to represent each signal on the 8b10b
encoder’s port map. See Figure 5.16.

5.1.3 Putting it all Together

Once both the Bus Module and Core are complete, this simple FPGA is now com-
pletely modeled using autoVHDL. Once this is complete, the code may now be syn-
thesized. The output of this model synthesis will be three individual VHDL files.
The first file will be the package which will tell the bus module how to function.
The second file will be the bus module file that will be fully configured to interface
with the model defined bus master. Last, a top file will be generated. This file is responsible for linking the bus module and 8b10b encoder VHDL files.

### 5.2 Example Using FIFOs

This example uses the model to represent a system that is a bit more complex than the previous model. In this case, the model still contains both a bus module and an engine/core components. However, these modules both interface with a data FIFOs. There are a few more differences between the first example and this one. To begin, a different bus module will need to be created as it has a different timing diagram. The core is now not generated from a commercial tool. Instead, it was custom created by a design engineer. Its specifics are not important and will only be evaluated from a high level port. The high level block diagram of this system can be found in Figure 5.17.

![Figure 5.17. High Level Block Diagram of an Example Using FIFOs](image)

### 5.2.1 Bus Module

This bus module has some similarities and some differences from the bus module used in the previous example. Like the previous example, this bus module contains a clock, start, and acknowledge signals. The clock signal will synchronize the rest of the bus signals. The start signal will be used to being a bus cycle and the acknowledge
signal will be used to end the cycle. During a master requested read cycle, the data will be placed back on the data bus while the acknowledge signal is active. For brevity, these signals will not be included in detail. The first difference between the first and second models is the address and data buses. In this example, the address and data buses share a combined 32 bit address bus. Therefore, there are two additional signals called address and data select. Address select will assert when the data on the bus is valid address data. Data select will assert when the data on the bus is valid data. There also is a difference between the write/read select lines. In this example, there are separate write and read select lines. This is shown in more detail in the bus timing diagram in Figure 5.18. Each of the new atoms necessary for modeling this system will be described in detail below.

![Bus Timing Diagram](image)

**Figure 5.18.** Bus Module Timing Diagram of an Example Using FIFOs

*Address/Data Bus* This is the combined address and data bus. This signal will need to be represented using the `addrAndDataBus`. To instantiate a `addrAndDataBus` atom, select it from the parts browser and drag it into the model. There are 2 attributes that are associated with this atom. The first is the `signalName`. In this case, this signal will be named “addrData”. The other attribute that must be set is `bitWidth`. This is should be set to “32” as this bus is 32 bits wide. See Figure 5.19.
Address Select  This is a control signal that is used by the bus master using a shared address/data bus. When this signal is active, there is valid address data on the bus. This signal will need to be represented by the addrSel atom. To instantiate a addrSel atom, select it from the parts browser and drag it into the model. There are two attributes that are associated with this atom. The first is signalName. In this case, the signal name is “addrSel”. The second attribute that will need to be specified is the active attribute. According to the bus module timing diagram, this signal is active high. Therefore, “high” should be selected from the drop down menu. See Figure 5.20.

Data Select  This is another control signal that is used by the bus master using a shared address/data bus. When this signal is active, there is valid data on the bus. This signal will need to be represented by the dataSel atom. To instantiate a dataSel atom, select it from the parts browser and drag it into the model. There are two
attributes that are associated with this atom. The first is the `signalName` attribute that should be “dataSel”. The second attribute is `active`. This attribute will tell the model synthesizer if this is an active high or active low signal. According to the bus module timing diagram, this signal should be set to “high.” See Figure 5.21.

**Write Select** This signal is the write select signal of the data bus. When this signal is asserted, the bus master is designating a write cycle. This signal will need to be represented by the `writeSel` atom. To instantiate a `writeSel` atom, select it from the parts browser and drag it into the model. There are two attributes that are associated with this atom. The first is the `signalName` attribute that should be “writeSel”. The second attribute is `active`. This attribute will tell the model synthesizer if this is an active high or active low signal. According to the bus module timing diagram, this signal should be set to “high.” See Figure 5.22.

---

**Figure 5.20. Bus Module Example – Address Select**

![Image of Address Select](image)

Address Sel

`addrSel`
Figure 5.21. Bus Module Example – Data Select

Figure 5.22. Bus Module Example – Write Select
*Read Select* This signal is the read select signal of the data bus. When this signal is asserted, the bus master is designated a read cycle. This signal will need to be represented by the `readSel` atom. To instantiate a `readSel` atom, select it from the parts browser and drag it into the model. There are two attributes that are associated with this atom. The first is the `signalName` attribute that should be “readSel”. The second attribute is `active`. This attribute will tell the model synthesizer if this is an active high or active low signal. According to the bus module timing diagram, this signal should be set to “high.” See Figure 5.23.

![Object Inspector](image)

**Figure 5.23.** Bus Module Example – Read Select

*Complete Bus Module Model* Now that all the individual atoms have been used to fully define the bus module, the model should resemble the one above. It should contain one atom each of the `busClock`, `addrAndDataBus`, `readSel`, `start`, `writeSel`, `addrSel`, `stop`, and `dataSel`. These atoms are used to define EVERY bus signal that is shown on the timing diagram. Once all the parameters are set, the model interpreter will
be able to generate a VHDL file that will properly support this bus functionality. It is important to remember that exactly one atom per signal must be used. If there is not enough or too many, the functionality of the bus module VHDL output file will not be correct. See Figure 5.24.

**Figure 5.24. Complete Bus Module Model**

### 5.2.2 Core

This core is slightly different from the previous example. For starters, this module is a custom design versus something that was generated from a commercial tool. However, this should not matter when modeling this example. This model also interfaces directly with the data FIFO. There are some similarities between this example core and the previous example core. There are the signals “txDataOut” and “txClkOut” are all connected to the top level module. These signals will enter and exit by FPGA pins and should be represented with the `topLevelPort` atom as well. There are also
three control signals that are driven by register bits. These are the “enable”, “errorInject,” and “bitRateSel” signals that also should be represented by the registerBits atom. For brevity purposes, these models will not be describe in detail as they are so similar to the previous example. The new signals will be described in detail below.

**Tx Data Register** This signal is the 32 bit data register that is coming from the output of the FIFO. This data will be used in the serial output of this core module. This signal should be represented by a FIFOSignal. To instantiate a FIFOSignal atom, select it from the parts browser and drag it into the model. There are three attributes that are associated with this atom. The first attribute is signalName. According to the high level block diagram, this attribute should be set to “txDataReg.” The next attribute that must be set is the bitWidth. This should be set to “32” as this is a 32 bit register. Last, the fifoOptions should be set. This attribute tells which FIFO signal to connect the port to. In this case, this signal is connected to the output of the data FIFO. Therefore, this attribute should be set to “q” using the drop down menu. See Figure 5.25.

**Tx FIFO Read Clock** This signal the clock that is used to drive the read logic of the asynchronous FIFO. This signal will need to be represented using the FIFOSignal. To instantiate a FIFOSignal atom, select it from the parts browser and drag it into the model. There are three attributes that are associated with this atom. The first attribute is signalName. According to the high level block diagram, this attribute should be set to “txFifoRdClk.” The next attribute that must be set is the bitWidth. This should be set to “1” as this is a single clock signal. Last, the fifoOptions should be set. This attribute tells which FIFO signal to connect the port to. In this case, this signal is connected to the read clock port of the FIFO. Therefore, this attribute should be set to “rdClk” using the drop down menu. See Figure 5.26.
Figure 5.25. Core Example – Tx Data Register

Figure 5.26. Core Example – Tx FIFO Read Clock
**Tx FIFO Read Request** This signal is the request that is sent to the FIFO to release the next data word. This signal will need to be represented using the **FIFOSignal**. To instantiate a **FIFOSignal** atom, select it from the parts browser and drag it into the model. There are three attributes that are associated with this atom. The first attribute is **signalName**. According to the high level block diagram, this attribute should be set to “txFifoRdReq.” The next attribute that must be set is the **bitWidth**. This should be set to “1” as this is a single signal. Last, the **fifoOptions** should be set. This attribute tells which FIFO signal to connect the port to. In this case, this signal is connected to the read request port of the FIFO. Therefore, this attribute should be set to “rdReq” using the drop down menu. See Figure 5.27.

![FIFO Signal](image)

**Figure 5.27. Core Example – Tx FIFO Read Request**

**Mode Select** According to the high level block diagram, this signal is a constant that is being driven high. This signal will need to be represented using the **constant**. To instantiate a **constant** atom, select it from the parts browser and drag it into
the model. There are three attributes that are associated with this atom. The first attribute is *signalName*. According to the high level block diagram, this attribute should be set to “modeSel.” The next attribute that must be set is the *bitWidth*. This should be set to “1” as this is a single signal. Last, the *binaryValue* needs to be set as well. In this case, this signal is always being driven high. Therefore, this signal should be set to a ‘1’. See Figure 5.28.

![Figure 5.28. Core Example – Mode Select](image)

*Complete Core Model* Now that all the individual atoms have been used to fully define the serial core, the model should be complete and ready to be synthesized. It should contain all the model atoms described above. These atoms are used to define *every* signal that is shown in the high level block diagram for the core module. Once all the parameters are set, the model interpreter will use this model to know how to construct the top level module. In this case, the top level module will be constructed to connect the bus module and FIFOs to the core engine. It is important to remember that no
code will be generated from the Core model instantiation. Rather, it will assume that this module already exists and provide provisions for it to communicate with the previously defined bus module. See Figure 5.29.

![Figure 5.29. Complete Core Model of Data Serialization Module](image)

### 5.2.3 FIFO

Next, the model must include a FIFO to complete the design. Just like the core model, this will not actually generate any code. This file is already assumed to exist. Each signal that exists on the bus module port map will need to be included in this model. Each signal can be derived from the block diagram.

*Data* This signal is the input data to the FIFO. This signal will be represented using a *data* atom. To instantiate a *data* atom, select it from the parts browser and drag it into the model. There are two attributes that are associated with this atom. The first is the *signalName*. This should be set to the name of the signal located on the
port map. In this case, it should be set to “data.” The bitWidth must also be set to 32 as the vector is 32 bits wide. See Figure 5.30.

![Figure 5.30. FIFO Example – Data](image)

**Write Clock** This signal is what all the write logic is synchronized to. Since this is an asynchronous FIFO, the read logic will be synchronized to the read clock. This signal will be represented using a wrClk atom. To instantiate a wrClk atom, select it from the parts browser and drag it into the model. There is only one attribute that is associated with this atom. This is the signalName. This should be set to the name of the signal located on the port map. In this case, the attribute should be set to “wrClk.” See Figure 5.31.

**Write Request** This FIFO will store the data located on the “data” port when this signal asserts for one clock cycle. Since this is an asynchronous FIFO, this signal should be synced with the write clock. This signal will be represented by the wrReq.
To instantiate a `wrReq` atom, select it from the parts browser and drag it into the model. There is only one attribute that is associated with this atom. This is the `signalName`. This should be set to the name of the signal located on the port map. In this case, the attribute should be set to “wrReq.” See Figure 5.32.

*Read Clock* This signal is the clock that all the read logic will be synchronized to. Since this is an asynchronous FIFO, the write logic will be synced to the write clock. This signal will be represented by the `rdClk`. To instantiate a `rdClk` atom, select it from the parts browser and drag it into the model. There is only one attribute that is associated with this atom. This is the `signalName`. This should be set to the name of the signal located on the port map. In this case, the attribute should be set to “rdClk.” See Figure 5.33.

*Read Request* This signal will pull data out of the FIFO when this signal is asserted for one clock cycle. This signal will be synchronous to the read clock. This signal will
Figure 5.32. FIFO Example – Write Request

Figure 5.33. FIFO Example – Read Clock
be represented by the \texttt{rdReq}. To instantiate a \texttt{rdReq} atom, select it from the parts browser and drag it into the model. There is only one attribute that is associated with this atom. This is the \texttt{signalName}. This should be set to the name of the signal located on the port map. In this case, the attribute should be set to “rdReq.” See Figure 5.34.

![Image of rdReq atom](image)

**Figure 5.34. FIFO Example – Read Request**

\[ Q \] This is the output of the data FIFO. This signal will be represented by the \texttt{q} atom. To instantiate a \texttt{q} atom, select it from the parts browser and drag it into the model. There are two attributes that are associated with this atom. This is the \texttt{signalName}. This should be set to the name of the signal located on the port map. In this case, the attribute should be set to “q.” The second attribute that must be set is the \texttt{bitWidth}. This should be set to 32 as this is a 32 bit wide FIFO. See Figure 5.35.

**Write Empty** This signal is a FIFO status signal. It will assert high when the FIFO is empty according to the write clock. This signal will be represented by the \texttt{wrEmpty}
atom. To instantiate a `wrEmpty` atom, select it from the parts browser and drag it into the model. There is one attribute that is associated with this atom. This is the `signalName`. This should be set to the name of the signal located on the port map. In this case, the attribute should be set to “wrEmpty.” See Figure 5.36.

Write Full This signal is a FIFO status signal. It will assert high when the FIFO is full according to the write clock. The signal will be represented by the `wrFull`. To instantiate a `wrFull` atom, select it from the parts browser and drag it into the model. There is one attribute that is associated with this atom. This is the `signalName`. This should be set to the name of the signal located on the port map. In this case, the attribute should be set to “wrFull.” See Figure 5.37.

Read Empty This signal is a FIFO status signal. It will assert high when the FIFO is empty according to the read clock. The signal will be represented by the `rdEmpty` atom. To instantiate a `rdEmpty` atom, select it from the parts browser and drag it
Figure 5.36. FIFO Example – Write Empty

Figure 5.37. FIFO Example – Write Full
into the model. There is one attribute that is associated with this atom. This is the signalName. This should be set to the name of the signal located on the port map. In this case, the attribute should be set to “rdEmpty.” See Figure 5.38.

![Image of Object Inspector with rdEmpty selected](image.png)

**Figure 5.38.** FIFO Example – Read Empty

*Read Full* This signal is a FIFO status signal. It will assert high when the FIFO is full according to the read clock. The signal will be represented by the rdFull atom. To instantiate a rdFull atom, select it from the parts browser and drag it into the model. There is one attribute that is associated with this atom. This is the signalName. This should be set to the name of the signal located on the port map. In this case, the attribute should be set to “rdFull.” See Figure 5.39.

*FIFO Model* The user needs to ensure that there is one FIFO atom for every signal on the port map of the FIFO. Once this is complete, it should look like Figure 5.40.
Figure 5.39. FIFO Example – Read Full

Figure 5.40. FIFO Example – Complete Model
5.3 Output Behaviors

5.3.1 Simple Example Using Bus Module and Core Models

As previously stated, after model based synthesis is complete there will be three generated VHDL output files. These files are “bus_module.vhd,” “bus_package.vhd” and “top_module.vhd.” These files can be found in Appendices A, B, and C. Each output file is described in more detail below.

**Bus Module** The bus module output code can be found in Appendix A. This code will be identical to any other bus module that is generated by the autoVHDL tool. This file is responsible for all interaction with the bus master. This is the only module that is permitted to put data back on the data bus during read cycles.

**Bus Package** The bus package is responsible for defining proper bus module functionality. This package was created from the atoms and their attributes that were defined in the bus module model. At the beginning of the file, all the buses are defined. In this case, the data bus was set to be 31 and the address bus was set to be 15. Note that all the vectors start at bit 0. The address/data bus width is defaulted to 31 bits wide. This is only required for compilation and will not be used for bus cycles.

This file has also set the polarity values of the start, stop, read select, write select, read/write, data select, and address select lines. In this case, all these will be set active high except for the stop signal. The stop signal will be an active low signal. Again, the read select, write select, address select, and data select signals will not be used, but need to be in the definition file.

Next, the type of address/data and read/write lines is defined. In this case, the bus module is set up to have a combined read/write line and separate address/data buses.

Last, the number of cycles in a bus cycle and the read/write access privileges of the registers need to be defined. Since the number of cycles is less than four, the tool
defaulted to four clock cycles. In addition, the registers have all been defined to be both read and write accessible by the bus master.

The output can be found in Appendix B.

*Top Module* The top module can be found by referencing Appendix C. The top module includes a port map of all the bus module elements that were defined in the module. All the bus module elements that were not defined were left as constants of open in the bus module instantiation. In addition, the 8b10b core is also defined in this file. Each port is connected as defined by the core model.

### 5.3.2 Example Using FIFOs

*Bus Module* The bus module output code can be found in Appendix D. This code will be identical to any other bus module that is generated by the autoVHDL tool. This file is responsible for all interaction with the bus master. This is the only module that is permitted to put data back on the data bus during read cycles. Please note that this output file is exactly the same as the one that was generated as part of the first example located in Appendix A.

*Bus Package* The bus package is responsible for defining proper bus module functionality. It is this file that makes the bus function different than the example used in the first example even though the “bus_module.vhd” files are identical. At the beginning of the file, all the buses are defined. In this case, it is a shared address/data bus that is 32 bits wide. Therefore the address/data bus width is set to 31. The separate address and data bus widths are defaulted to 31. The separate address/data buses will not be used, but need to be included for bus module compilation.

The file also set the polarity of the start, stop, read select, write select, read/write, data select, and address select lines. In this case, all these will be set active high except for the stop signal. The stop signal will be an active low signal. In this instance, the
read/write signal will not be used. However the constant still needs to be set in the package file for compilation purposes.

Next, the type of address/data and read/write lines is defined. In this case, the bus module is set up to have separate read write lines and a shared address/data bus.

Last, the number of cycles in a bus cycle and the read/write active privileges of the registers need to be defined. Since the number of cycles is less than four, the tool defaulted to four clock cycles. In addition, the registers have all been defined to be both read and write accessible by the user.

The output can be found in Appendix E.

Top Module The top module can be found by referencing Appendix F. The top module includes a port map of all the bus module elements that were defined in the module. All the bus module elements that were not defined were left as constants in the bus module instantiation. In addition, the transmission core and FIFO components and their instantiations are also included in this file. Each port of the core component is defined by the core model. Likewise, each port of the FIFO component is defined by the FIFO model.

5.3.3 ModelSim Simulations

Simple Example with Core and Bus Module After the code was generated using the autoVHDL tool, it was both compiled and synthesized using ModelSim. A simple test bench was written to perform a write cycle and then a read cycle to address 0x02. Please note that this simulation only includes the bus module and bus package files. Since every core module will vary for each design, it was not included in the simulation. This is shown in Figure 5.41.

Example with FIFOs After these files were generated, they were compiled using ModelSim. Unfortunately, due to the large number of addresses ModelSim could not han-
Figure 5.41. Simple Example Bus Module Simulation
dle simulating the second example. The large amount of memory needed to actually run the simulation caused ModelSim to generate a fault and stop the simulation. This is something that is discussed further in this chapter under the application constraints section.

5.3.4 Synthesis

Due to the fact that this design supports many core modules that can vary in size, the only synthesis that was performed on the files was that of the bus module. This can give designers a clue of how much extra room will be needed to support the bus module. Synthesis was performed with both Synplicity’s Synplify and Xilinx. The maximum estimated speed that the code will run was estimated by synplify. This EDIF output file was then taken to Xilinx for device target. The Xilinx design summary, including LUT usage and total utilization, will be provided for both examples. The device targeted for these examples was the Xilinx Virtex 5 XC5VLX30FF324.

Please note that the estimated speed and utilization is simply an estimate. Depending on the fabric of the FPGA and its attributes, the bus module has the potential to run faster or slower. Obviously, if the part is older does not have as many resources, it will not be able to run as fast as a newer FPGA with more resources.
Simple Example Using Bus Module and Core Models Using Synplify, synthesis of the bus module and bus package was performed. Maximum clock speed is estimated to be 320 MHz. The Figure 5.42 shows the excerpt from the output log provided by the Synplify tool.

![Synplify Timing Report](image)

**Figure 5.42. Simple Example – Synplify Timing Performance Summary**

After this was complete, the output EDIF file was used in Xilinx to complete the device targeting. For the Virtex 5 in the current bus module configuration, it utilized 6 percent of the overall device. The output summary can be found in Figure 5.43.

Example using FIFO Unfortunately, synthesis and device targeting did not go well for the second example. One of the application constraints of this tool is that the size of the solution quickly spirals out of control as the address bus width gets larger. This is due to the fact that a register is generated for each possible register address. Due to the fact that the address registers took up so much room, it could not create a feasible solution for this model. This is discussed in more detail in the Application Constraints section of this chapter.

5.4 Application Constraints

As with any tool, there are design limitations and constraints that are associated with the autoVHDL tool. To begin, there is only one model of each the Core, Bus Module,
**Xilinx Design Summary**

<table>
<thead>
<tr>
<th>Slice Logic Utilization</th>
<th>Used</th>
<th>Available</th>
<th>Utilization</th>
<th>Notes(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Slice Registers</td>
<td>1,295</td>
<td>14,200</td>
<td>6%</td>
<td></td>
</tr>
<tr>
<td>Number used as Flip Flops</td>
<td>1,254</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number used as Latches</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of Slice LUTs</td>
<td>478</td>
<td>14,200</td>
<td>2%</td>
<td></td>
</tr>
<tr>
<td>Number used as Logic</td>
<td>478</td>
<td>14,200</td>
<td>2%</td>
<td></td>
</tr>
<tr>
<td>Number using O6 output only</td>
<td>446</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number using O5 and O6</td>
<td>32</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Slice Logic Distribution**

| Number of occupied Slice | 423 | 4,800 | 8% |         |
| Number of LUT Flip Flop pairs used | 1,347 | | | |
| Number with an unused Flip Flop | 51 | 1,347 | 3% |         |
| Number with an unused LUT | 969 | 1,347 | 64% |         |
| Number of fully used LUT-FF pairs | 427 | 1,347 | 31% |         |
| Number of unique control sets | 0 | | | |

**IO Utilization**

| Number of bonded IOBs | 22 | 220 | 12% |         |

**Specific Feature Utilization**

| Number of BUFG/BUFSSL cells | 2 | | 6% |         |
| Number used as BUFGs | 2 | | | |

**FIGURE 5.43. Simple Example — Xilinx Device Utilization Summary**
and FIFO. This was done to help manage complexity. However, this isn’t necessarily
the most realistic approach for most designs. FPGAs available in today’s market are
increasingly large and are able to support a multitude of files. However, if this tool
is taken as a proof of concept, it can be easily expanded to include more cores and
FIFOs.

Another constraint is that bus module must be able to be represented by the
model in order to generate a functional top level module. The model exploits the
commonalities of many buses but is by no means all inclusive. If the bus module
cannot be represented in the paradigm, this tool can not be used to generate a feasible
solution for a particular design. Much emphasis has been placed on common buses
such as the PCI bus. It is possible that this model could be configured to include
standard buses that does not require user definition.

In many systems, data throughput over data buses can become an issue. Many
times, this throughput issue can be resolved by having efficient bus cycles. The
minimum number of clock cycles per bus cycle is four. This is not overly slow but
is by no means fast. This is due to the number of registers in the design. It takes a
larger number of clock cycles, because there isn’t a great deal of combination logic in
the bus implementation. Having the minimum number of clock cycles at four could
mean that this bus module implementation could not sustain a system with high bus
bandwidth requirements. The bus module could use more combination logic to make
the minimum number of clock cycles per active clock cycles lower. However, that
only comes with a trade off. This would mean that the bus could not be ran at as
high of a frequency, because a high level of combination logic between registers.

Last, there is an issue with the size of the address bus. One register the width
of the data bus is created for each valid address. This can be quite consuming
is only 8 bits wide, there will be 256 registers generated. In the second example,
there are 31 address bits making 2,147,483,647 registers. Obviously not all of these
registers will be used in the design. However, they are still being created and taking
up space in the fabric. To make matters worse, the range on an integer is only \(-2,147,483,648\) to \(2,147,483,647\). Therefore, the current interpreter cannot support an address but greater than 32 bits wide. This creates a problem not only for address width constraints, but also for space inside the FPGA. This bus module will not be able to be used on smaller FPGAs the way that it is currently constructed. This can be resolved by changing the implementation so that an address register is only implemented if it exists. This would significantly cut down on the amount of fabric that is utilized in the FPGA.

At the time of publication, the FIFO model will create a component declaration but not an instantiation. All the framework is included in the paradigm and model interpreter to handle this model. However, current implantation does not generate code that is totally feasible. It is possible in its current state to use this model. Hand tweaking after code generation is necessary to get the top module to compile.

All these issues can be addressed in future work.
Chapter 6

CONCLUSION

6.1 Summary

Time to market combined with emphasis on heavy reuse has made an engineering job even more difficult. With profits frequently defining the bottom line, company policy often pushes engineers to utilize code that already exists. This can be problematic if the code was not constructed in a modular fashion. This could lead to a solution that follows the old cliche of putting a square peg in a round hole.

To counteract this problem, autoVHDL was developed to give VHDL developers more portability with their code. Using a domain specific modeling environment, the users can take cores (purchased or proprietary) and rehost them quickly on different platforms by simply providing a model of a data bus and how it interacts with the core. The tool will provide output files of the bus module and a top file that will define how all the component level interacts take place. Using this tool saves significant time and money, because it will reduce the amount of time developers have to spend coding.

6.2 Future Work

This tool has created a foundation on which future improvements can be made. The existing tool is a great tool for proof of concept, but to make it more usable for more VHDL applications, updates should be made.

To begin with, the current paradigm does not contain any constraints. Instead, it prints warning statements to the console to warn the user that optimal output functionality will not be achieved. It would be beneficial for users who are just
starting to work with the paradigm and/or GME to have actual errors are generated. This could minimize later frustration with the tool’s output files.

In addition, all atoms are unpopulated when they are added to the models. It would be beneficial to have these values populated automatically. This could be taken one step further so that the models could be instantiated to represent mainstream commercial buses without any user intervention. Such example would be the PCI bus.

The interpreter could also be expanded so that way more than one model of each Core, Bus Module, and FIFO can be used at one time. This would allow this tool to be used for bigger projects that utilize large FPGAs with an abundance of resources.

One other issue that could be improved upon that has been previously discussed is the trade off between cycle time and clock speed. It would be beneficial to the end user if he or she could pick whether or not the bus module should be implemented with more emphasis on running faster or having a minimum number of clock cycles. If the user would like a faster bus, it would result in greater cycle times. Oppositely, if the user would like less cycle time, they may have to compromise their bus speed.
REFERENCES


Appendix A

GENERATED VHDL FOR BUS MODULE (EXAMPLE 1)

The following code shows the generated output of the bus model, based on the model created for the example discussed in Figure 3.1.

ex1_bus_module.vhd

```vhdl
-- Bus Module file generated by AutoVHDL paradigm using GME

-- DO NOT EDIT THIS FILE !!!!!!!

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

library work;
use work.bus_package.all;

entity bus_module is
  port(
    clkIn : in std_logic;
    -- Bus clock; All bus clock logic will be synchronized to this clock
    rstIn : in std_logic;
    -- Active high reset
    addrBusIn : in std_logic_vector(ADDR_BUS_WIDTH downto 0);
    -- Separate address bus
    startIn : in std_logic;
    -- Kicks off active bus cycle
    dataSelIn : in std_logic;
    -- Used with combined address/data bus to say data is valid data bus data
    rdWrIn : in std_logic;
    -- Combined read/write select line. Polarity determined by constants in package
    readSelIn : in std_logic;
    -- Separate read select line; Asserts during an active read cycle;
    writeSelIn : in std_logic;
    -- Separate write select line; Asserts during an active write cycle;
    addrSelIn : in std_logic;
    -- Used with combined address/data bus. Asserts when data on bus is active address data;

```


stopOut : out std_logic;
   -- Ends an active bus cycle.
addrDataBus : inout std_logic_vector(DATA_ADDR_WIDTH downto 0);
   -- Combined address/data bus
dataBus : inout std_logic_vector(DATA_BUS_WIDTH downto 0);
   -- Separate data bus
regDataBi : inout addressData
   -- Data registers for all possible register addresses
);
end entity bus_module;

architecture rtl of bus_module is
begin
   -- This process is responsible for registering all the signals that are coming in
   from the bus master. The signals will be reset to a safe state by looking
at the polarity found in the package during reset. After reset is relinquished, they will update as specified by the bus master on the bus clock rising edge.

RegProc : process(clkIn, rstIn)
begin
if (rstIn = '1') then
  if (START_POLARITY = '1') then
    startR <= '0';
  else
    startR <= '1';
  end if;
  if (DATA_SEL_POLARITY = '1') then
    dataSelR <= '0';
  else
    dataSelR <= '1';
  end if;
  if (ADDR_SEL_POLARITY = '1') then
    addrSelR <= '0';
  else
    addrSelR <= '1';
  end if;
  if (RD_SEL_POLARITY = '1') then
    readSelR <= '0';
  else
    readSelR <= '1';
  end if;
  if (WR_SEL_POLARITY = '1') then
    writeSelR <= '0';
  else
    writeSelR <= '1';
  end if;
  if (RD_WR_POLARITY = '1') then
    rdWrR <= '0';
  else
    rdWrR <= '1';
  end if;
  addrDataBusR <= (others => '0');
dataBusR <= (others => '0');
addrBusR <= (others => '0');
elsif (clkIn 'event and clkIn = '1') then
    startR <= startIn;
    dataSelR <= dataSelIn;
    addrSelR <= addrSelIn;
    readSelR <= readSelIn;
    writeSelR <= writeSelIn;
    rdWrR <= rdWrIn;
    addrDataBusR <= addrDataBus;
    dataBusR <= dataBus;
    addrBusR <= addrBusIn;
end if;
end process RegProc;

−−
This process latches the address. The constants will tell the
−−
code whether or not the address comes from the combined address
−−
data bus or the address only bus. If it is an address only bus,
−−
it will use the polarity constant to know if it should capture
−−
when the address select line is high or low.
−−

AddrLatchProc : process(clkIn, rstIn)
begin
if (rstIn = '1') then
    addrLatchR <= (others => '0');
elsif (clkIn 'event and clkIn = '1') then
    if (ADDR_DATA_SEL = '0') then — Separate addr/data bus
        if (START_POLARITY = '1') then — Active high start signal
            if (startR = '1') then — Active high start signal
                addrLatchR <= addrBusR;
            end if;
        else
            if (startR = '0') then — Active low start signal
                addrLatchR <= addrBusR;
            end if;
        end if;
    else — Shared addr/data bus
        if (ADDR_SEL_POLARITY = '1') then — Active high addr sel signal
            if (addrSelR = '1') then
                addrLatchR <= dataAddrBusR;
            end if;
        else
            else
if (addrSelR = '0') then — Active low addr sel signal
    addrLatchR <= dataAddrBusR;
end if;
end if;
end if;
end if;
end if;
end process AddrLatchProc;

This process latches the data. It will evaluate whether or not it's a shared or separate data bus. Then it will look at the start signal or the dataSel or the start signal to see when the data should be captured on the data bus.

DataLatchProc : process(clkIn, rstIn)
begin
if (rstIn = '1') then
    dataLatchR <= (others => '0');
elsf (clkIn'event and clkIn = '1') then
    if (DATA_VALID_SEL = '0') then — Uses separate addr and data buses
        if (START_POLARITY = '1') then
            if (startR = '1') then — Active high start signal
                dataLatchR <= dataBusR;
            end if;
        else
            if (startR = '0') then — Active low start signal
                dataLatchR <= dataBusR;
            end if;
        end if;
    else — Uses combined addr and data bus
        if (DATA_SEL_POLARITY = '1') then — active high data select signal
            if (dataSelR = '1') then
                dataLatchR <= dataAddrBusR;
            end if;
        else — active low data select signal
            if (dataSelR = '0') then
                dataLatchR <= dataBusR;
            end if;
        end if;
    end if;
end if;
end if;
end if;
end if;
end if;
end if;
end if;
end if;
end process AddrLatchProc;
dataLatchR <= dataAddrBusR;
end if;
end if;
end if;
end if;
end process DataLatchProc;

— This process is responsible for figuring out the value of the read/write line.
— Internal to the bus module, everything gets combined into a signal write/read line
— that is low during a write cycle and high during a read cycle. The atoms found in
— the models are used to figured out when this bus should go high or low.

ReadWriteProc : process(clkIn, rstIn)
begin
if (rstIn = '1') then
rdWrLowR <= '1';
elseif (clkIn'event and clkIn = '1') then
if ((WR_RD_SEL = '0') and (START_POLARITY = '1')) then — separate read/write lines w/ start polarity active high
if (startR = '1') then
if (WR_SEL_POLARITY = '1') then — write select active high
if (wrSelR = '1') then
rdWrLowR <= '0';
else
rdWrLowR <= '1';
end if;
else — write select active low
if (wrSelR = '0') then
rdWrLowR <= '0';
else
rdWrLowR <= '1';
end if;
end if;
elsif ((WR_RD_SEL = '0') and (START_POLARITY = '0')) then — separate read/write lines w/ start polarity active low
if (startR = '0') then
if (WR_SEL_POLARITY = '1') then — write select active high
if (wrSelR = '1') then
    rdWrLowR <= '0';
else
    rdWrLowR <= '1';
end if;
else — write select active low
if (wrSelR = '0') then
    rdWrLowR <= '0';
else
    rdWrLowR <= '1';
end if;
end if;

elsif ((WR_RD_SEL = '1') and (START_POLARITY = '1')) then — combined read/write line w/ start polarity active high
if (startR = '1') then
    if (RD_WR_POLARITY = '1') then — read is active low
        if (rdWrR = '0') then
            rdWrLowR <= '1';
        else
            rdWrLowR <= '0';
        end if;
    else — write is active low
        if (rdWr = '0') then
            rdWrLowR <= '0';
        else
            rdWrLowR <= '1';
        end if;
    end if;
else — write is active low
    if (rdWrR = '0') then
        rdWrLowR <= '0';
    else
        rdWrLowR <= '1';
    end if;
end if;

elsif ((WR_RD_SEL = '1') and (START_POLARITY = '0')) then — combined read/write line w/ start polarity active low
if (startR = '0') then
    if (RD_WR_POLARITY = '1') then — read is active low
        if (rdWrR = '0') then
            rdWrLowR <= '1';
        else
            rdWrLowR <= '0';
        end if;
    else — write is active low
        if (rdWr = '0') then
            rdWrLowR <= '0';
        else — write is active low
            if (rdWr = '0') then
                rdWrLowR <= '0';
            else
                rdWrLowR <= '1';
            end if;
        end if;
    end if;
else — write is active low
    if (rdWrR = '0') then
        rdWrLowR <= '0';
    else — write is active low
        if (rdWr = '0') then
            rdWrLowR <= '0';
        else
            rdWrLowR <= '1';
        end if;
    end if;
else — write is active low
    if (rdWrR = '0') then
        rdWrLowR <= '0';
    else — write is active low
        if (rdWr = '0') then
            rdWrLowR <= '0';
        else
            rdWrLowR <= '1';
        end if;
    end if;
else — write is active low
    if (rdWrR = '0') then
        rdWrLowR <= '0';
    else — write is active low
        if (rdWr = '0') then
            rdWrLowR <= '0';
        else
            rdWrLowR <= '1';
        end if;
    end if;
else — write is active low
    if (rdWrR = '0') then
        rdWrLowR <= '0';
    else — write is active low
        if (rdWr = '0') then
            rdWrLowR <= '0';
        else
            rdWrLowR <= '1';
        end if;
    end if;
else — write is active low
    if (rdWrR = '0') then
        rdWrLowR <= '0';
    else — write is active low
        if (rdWr = '0') then
            rdWrLowR <= '0';
        else
            rdWrLowR <= '1';
        end if;
    end if;
else — write is active low
    if (rdWrR = '0') then
        rdWrLowR <= '0';
    else — write is active low
        if (rdWr = '0') then
            rdWrLowR <= '0';
        else
            rdWrLowR <= '1';
        end if;
    end if;
else — write is active low
    if (rdWrR = '0') then
        rdWrLowR <= '0';
    else — write is active low
        if (rdWr = '0') then
            rdWrLowR <= '0';
        else
            rdWrLowR <= '1';
        end if;
    end if;
else — write is active low
    if (rdWrR = '0') then
        rdWrLowR <= '0';
    else — write is active low
        if (rdWr = '0') then
            rdWrLowR <= '0';
        else
            rdWrLowR <= '1';
        end if;
    end if;
else — write is active low
    if (rdWrR = '0') then
        rdWrLowR <= '0';
    else — write is active low
        if (rdWr = '0') then
            rdWrLowR <= '0';
        else
            rdWrLowR <= '1';
        end if;
    end if;
else — write is active low
    if (rdWrR = '0') then
        rdWrLowR <= '0';
    else — write is active low
        if (rdWr = '0') then
            rdWrLowR <= '0';
        else
            rdWrLowR <= '1';
        end if;
    end if;
else — write is active low
    if (rdWrR = '0') then
        rdWrLowR <= '0';
    else — write is active low
        if (rdWr = '0') then
            rdWrLowR <= '0';
        else
            rdWrLowR <= '1';
        end if;
    end if;
else — write is active low
    if (rdWrR = '0') then
        rdWrLowR <= '0';
    else — write is active low
        if (rdWr = '0') then
            rdWrLowR <= '0';
        else
            rdWrLowR <= '1';
        end if;
    end if;
else — write is active low
    if (rdWrR = '0') then
        rdWrLowR <= '0';
    else — write is active low
        if (rdWr = '0') then
            rdWrLowR <= '0';
        else
            rdWrLowR <= '1';
        end if;
    end if;
else — write is active low
    if (rdWrR = '0') then
        rdWrLowR <= '0';
    else — write is active low
        if (rdWr = '0') then
            rdWrLowR <= '0';
        else
            rdWrLowR <= '1';
        end if;
    end if;
else — write is active low
    if (rdWrR = '0') then
        rdWrLowR <= '0';
    else — write is active low
        if (rdWr = '0') then
            rdWrLowR <= '0';
        else
            rdWrLowR <= '1';
        end if;
    end if;
else — write is active low
    if (rdWrR = '0') then
        rdWrLowR <= '0';
    else — write is active low
        if (rdWr = '0') then
            rdWrLowR <= '0';
        else
            rdWrLowR <= '1';
        end if;
    end if;
else — write is active low
    if (rdWrR = '0') then
        rdWrLowR <= '0';
    else — write is active low
        if (rdWr = '0') then
            rdWrLowR <= '0';
        else
            rdWrLowR <= '1';
        end if;
    end if;
else — write is active low
    if (rdWrR = '0') then
        rdWrLowR <= '0';
    else — write is active low
        if (rdWr = '0') then
            rdWrLowR <= '0';
        else
            rdWrLowR <= '1';
        end if;
    end if;
else — write is active low
    if (rdWrR = '0') then
        rdWrLowR <= '0';
    else — write is active low
        if (rdWr = '0') then
            rdWrLowR <= '0';
        else
            rdWrLowR <= '1';
        end if;
    end if;
else — write is active low
    if (rdWrR = '0') then
        rdWrLowR <= '0';
    else — write is active low
        if (rdWr = '0') then
            rdWrLowR <= '0';
        else
            rdWrLowR <= '1';
        end if;
    end if;
else — write is active low
    if (rdWrR = '0') then
        rdWrLowR <= '0';
    else — write is active low
        if (rdWr = '0') then
            rdWrLowR <= '0';
        else
            rdWrLowR <= '1';
        end if;
    end if;
else — write is active low
    if (rdWrR = '0') then
        rdWrLowR <= '0';
    else — write is active low
        if (rdWr = '0') then
            rdWrLowR <= '0';
        else
            rdWrLowR <= '1';
        end if;
    end if;
else — write is active low
    if (rdWrR = '0') then
        rdWrLowR <= '0';
    else — write is active low
        if (rdWr = '0') then
            rdWrLowR <= '0';
        else
            rdWrLowR <= '1';
        end if;
    end if;
else — write is active low
    if (rdWrR = '0') then
        rdWrLowR <= '0';
    else — write is active low
        if (rdWr = '0') then
            rdWrLowR <= '0';
        else
            rdWrLowR <= '1';
        end if;
    end if;
else — write is active low
    if (rdWrR = '0') then
        rdWrLowR <= '0';
    else — write is active low
        if (rdWr = '0') then
            rdWrLowR <= '0';
        else
            rdWrLowR <= '1';
        end if;
    end if;
else — write is active low
    if (rdWrR = '0') then
        rdWrLowR <= '0';
    else — write is active low
        if (rdWr = '0') then
            rdWrLowR <= '0';
        else
            rdWrLowR <= '1';
        end if;
    end if;
rdWrLowR <= '0';
else
    rdWrLowR <= '1';
end if;
end if;

end if;

end if;
end if;
end if;
end if;
end if;

end process ReadWriteProc;

StartProc : process(rstIn, stopR, startR)
begin

if (rstIn = '1') then
    cycleActiveR <= '0';
else
    stop2R <= stopR;
if ((STOP_POLARITY = '1') and (stop2R = '1')) then
    cycleActiveR <= '0';
elsif ((STOP_POLARITY = '0') and (stop2R = '0')) then
    cycleActiveR <= '0';
else
    if ((START_POLARITY = '1') and (startR = '1')) then
        cycleActiveR <= '1';
elsif ((START_POLARITY = '0') and (startR = '0')) then
        cycleActiveR <= '0';
end if;
end if;
end if;
end process StartProc;

This process is responsible for keeping track of an active cycle. This is kicked off by the start signal and will end when it receives a valid stop signal.

This process counts the number of cycles that pass in a single bus.
It uses the signal in the previous process to know when to start and stop counting.

CycleCountProc : process(rstIn, clkIn)
begin
if (rstIn = '1') then
cycleCntR <= 0;
elif (clkIn'event and clkIn = '1') then
if (cycleActiveR = '1') then
cycleCntR <= cycleCntR + 1;
else
cycleCntR <= 0;
end if;
end if;
end process;

This process uses the counter to know when to stop the bus cycle. When all the clock cycles have lapsed, it will assert the internal stop signal.

StopProc : process(rstIn, clkIn)
begin
if (rstIn = '1') then
if (STOP_POLARITY = '1') then
stopR <= '0';
else
stopR <= '1';
end if;
elif (clkIn'event and clkIn = '1') then
if ((cycleCntR = CYCLE_CNT) and (STOP_POLARITY = '1')) then
stopR <= '1';
elif ((cycleCntR = CYCLE_CNT) and (STOP_POLARITY = '0')) then
stopR <= '0';
elif (STOP_POLARITY = '0') then
stopR <= '1';
elif (STOP_POLARITY = '1') then
...
stopR <= '0';
end if;
end if;
end process StopProc;
stopt <= stopR;
--
-- This process is responsible for generating data to the registers
--
regDataBi <= (others => (others => 'Z'));
--
-- Converting the current address to an integer
--
addrUnsigned <= unsigned(addrLatchR);
addrInt <= to_integer(addrUnsigned);
--
-- Generates one data latch for every possible address location provided
-- that the register type is Write Only or Read/Write
--
RegDataGen : for i in 0 to GENERATEEND generate
RegDataProc : process(clkIn, rstIn)
begin
if (rstIn = '1') then
regDataBi(i) <= (others => 'Z');
elsif (clkIn'event and clkIn = '1') then
if (REG_RW(i) = '1') then
if ((cycleActiveR = '1') and (rdWrLowR = '0') and (i = addrInt))
then
regDataBi(i) <= dataLatchR;
end if;
elsif (WO_RO(i) = '1') then
if ((cycleActiveR = '1') and (rdWrLowR = '0') and (i = addrInt))
then
regDataBi(i) <= dataLatchR;
end if;
end if;
end if;
end if;
This process is responsible for determining which data should go back on the bus. This will be determined by the address bus provided that the register type is Read Only or Read/Write. If it is a write only register, all ones will be put back on the data bus.

ReturnDataProc : process(clkIn, rstIn) begin

if (rstIn = '1') then
  returnDataR <= (others => '0');
elsif (clkIn'event and clkIn = '1') then
  if (REG_RW(addrInt) = '1') then
    returnDataR <= regDataBi(addrInt);
  elsif (WO_RO(addrInt) = '0') then
    returnDataR <= regDataBi(addrInt);
  else
    returnDataR <= (others => '1');
  end if;
end if;
end process ReturnDataProc;

This process uses the stop signal to know when to put the data back on the bus. It takes into account if the stop signal is active high or active low.

PutDataProc : process(rstIn, stopR, rdWrLowR) begin

if (rstIn = '1') then
  putDataOnBusR <= '0';
else
  if (((stopR = '1') and (STOP_POLARITY = '1')) and (rdWrLowR = '1')) then
    putDataOnBusR <= '1';
  end if;
end if;
end process PutDataProc;
elsif ((stopR = '0') and (STOP_Polarity = '0')) and (rdWrLowR = '1')) then
  putDataOnBusR <= '1';
else
  putDataOnBusR <= '0';
end if;
end process PutDataProc;
dataBus <= (others => 'Z');
-- Statement puts data back on the data bus.
--
dataBus <= returnDataR when (putDataOnBusR = '1') else (others => 'Z');
end architecture rtl;
Appendix B

Generated VHDL for Bus Package (Example 1)

The following code shows the generated output of the bus package, based on the model created for the example discussed in Figure 3.1.

ex1_bus_package.vhd

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

package bus_package is

constant DATA_BUS_WIDTH : integer := 15;
constant DATA_ADDR_WIDTH : integer := 31;
constant ADDR_BUS_WIDTH : integer := 7;

constant START_POLARITY : std_logic := '1'; -- 0 = Negative Polarity
    1 = Positive Polarity;
class constant STOP_POLARITY : std_logic := '0'; -- 0 = Negative Polarity
    1 = Positive Polarity;
class constant RD_SEL_POLARITY : std_logic := '1'; -- 0 = Negative Polarity
    1 = Positive Polarity;
class constant WR_SEL_POLARITY : std_logic := '1'; -- 0 = Negative Polarity
    1 = Positive Polarity;
class constant RD_WR_POLARITY : std_logic := '0'; -- 0 = Write Active Low;
        1 = Read Active Low;
class constant DATA_SEL_POLARITY : std_logic := '1'; -- 0 = Write Active Low;
        1 = Read Active Low;
class constant ADDR_SEL_POLARITY : std_logic := '1'; -- 0 = Write Active Low;
        1 = Read Active Low;
class constant ADDR_DATA_SEL : std_logic := '0'; -- 0 = Separate Addr/Data Bus
    1 = Shared Addr/Data Bus;
class constant DATA_VALID_SEL : std_logic := '0'; -- 0 = Separate Addr/Data Bus
    1 = Shared Addr/Data Bus;
class constant WR_RD_SEL : std_logic := '0'; -- 0 = Separate wr/rd lines
        1 = Shared wr/rd lines;
class constant CYCLE_CNT : integer := 1;
class constant GENERATE_END : integer := 255;
```
constant REG_RW : std_logic_vector (GENERATE_END downto 0) :=
    (others => '1');
constant WO_RO : std_logic_vector (GENERATE_END downto 0) :=
    (others => '0');
type addressData is array (GENERATE_END downto 0) of std_logic_vector(
    DATA_BUS_WIDTH downto 0);
end;
Appendix C

**GENERATED VHDL FOR TOP MODULE (EXAMPLE 1)**

The following code shows the generated output of the top module, based on the model created for the example discussed in Figure 3.1.

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
library work;
use work.bus_package.all;

entity top_module is
  port(
    rstIn : in std_logic;
    busClkIn : in std_logic;
    addrBusIn : in std_logic_vector(ADDR_BUS_WIDTH downto 0);
    dataBus : inout std_logic_vector(DATA_BUS_WIDTH downto 0);
    startIn : in std_logic;
    stopOut : out std_logic;
    rdWrIn : in std_logic;
    dout : out std_logic_vector(7 downto 0);
    kerr : out std_logic;
    disp_out : out std_logic;
    clk : in std_logic);
end entity top_module;

architecture struct of top_module is

component bus_module
  port(
    clkIn : in std_logic;
    -- Bus clock; All bus clock logic will be synchronized to this clock
```
rstIn : in std_logic;
  — Active high reset
addrBusIn : in std_logic_vector(ADDR_BUS_WIDTH downto 0);
  — Separate address bus
startIn : in std_logic;
  — Kicks off active bus cycle
dataSelIn : in std_logic;
  — Used with combined address/data bus to say data is valid
dataBus data
rdWrIn : in std_logic;
  — Combined read/write select line. Polarity determined by
  constants in package
readSelIn : in std_logic;
  — Separate read select line; Asserts during an active read
cycle;
writeSelIn : in std_logic;
  — Separate write select line; Asserts during an active write
cycle;
addrSelIn : in std_logic;
  — Used with combined address/data bus. Asserts when data on
  bus is active address data;
stopOut : out std_logic;
  — Ends an active bus cycle.
addrDataBus : inout std_logic_vector(DATA_ADDR_WIDTH downto 0);
  — Combined address/data bus
dataBus : inout std_logic_vector(DATA_BUS_WIDTH downto 0);
  — Separate data bus
regDataBi : inout addressData
):
end component;

component encoder8b10b
port(
dout : out std_logic_vector(7 downto 0);
kerr : out std_logic;
disp_out : out std_logic;
clk : in std_logic;
din : in std_logic_vector(7 downto 0);
kin : in std_logic;
ce : in std_logic);
end component;
signal regData : addressData;
begin
bus_module_inst : bus_module
port map

66 | rstIn         => rstIn,  
67 | clkIn         => busClkIn,  
68 | addrDataBus   => open,  
69 | addrBusIn     => addrBusIn,  
70 | dataBus       => dataBus,  
71 | dataSelIn     => '0',  
72 | startIn       => startIn,  
73 | stopOut       => stopOut,  
74 | readSelIn     => '0',  
75 | writeSelIn    => '0',  
76 | rdWrIn        => rdWrIn,  
77 | addrSelIn     => '0',  
78 | regDataBi     => regData  
79 | );  
80 |  
81 | encoder8b10b_inst : encoder8b10b  
82 | port map(  
83 | dout => dout,  
84 | kerr => kerr,  
85 | disp_out => disp_out,  
86 | clk => clk,  
87 | din => regData(02)(7 downto 0),  
88 | kin => regData(04)(0),  
89 | ce => regData(04)(1)); end architecture;  
  
. output/ex1_top_module.vhd
Appendix D

GENERATED VHDL FOR BUS MODULE (EXAMPLE 2)

The following code shows the generated output of the bus model, based on the model created a complex example.

```
ex2_bus_module.vhd

library ieee; use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

library work; use work.bus_package.all;

entity bus_module is
  port(
    clkIn : in std_logic; -- Bus clock; All bus clock logic will be synchronized to this clock
    rstIn : in std_logic; -- Active high reset
    addrBusIn : in std_logic-vector(ADDR_BUS WIDTH downto 0); -- Separate address bus
    startIn : in std_logic; -- Kicks off active bus cycle
    dataSelIn : in std_logic; -- Used with combined address/data bus to say data is valid data bus data
    rdWrIn : in std_logic; -- Combined read/write select line. Polarity determined by
     constants in package
    readSelIn : in std_logic; -- Separate read select line; Asserts during an active read cycle;
    writeSelIn : in std_logic; -- Separate write select line; Asserts during an active write cycle;
    addrSelIn : in std_logic; -- Used with combined address/data bus. Asserts when data on bus is
       active address data;

```
stopOut : out std_logic;
  -- Ends an active bus cycle.
addrDataBus : inout std_logic_vector(DATA_ADDR_WIDTH downto 0);
  -- Combined address/data bus
dataBus : inout std_logic_vector(DATA_BUS_WIDTH downto 0);
  -- Separate data bus
regDataBi : inout addressData
  -- Data registers for all possible register addresses
end entity bus_module;

architecture rtl of bus_module is

signal dataBusR : std_logic_vector(DATA_BUS_WIDTH downto 0);
signal addrBusR : std_logic_vector(ADDR_BUS_WIDTH downto 0);
signal dataAddrBusR : std_logic_vector(DATA_ADDR_WIDTH downto 0);
signal returnDataR : std_logic_vector(DATA_BUS_WIDTH downto 0);
signal addrSelR : std_logic;
signal startR : std_logic;
signal rdWrR : std_logic;
signal readSelR : std_logic;
signal writeSelR : std_logic;
signal stopR : std_logic;
signal dataSelR : std_logic;
signal interruptR : std_logic;
signal addrLatchR : std_logic_vector(ADDR_BUS_WIDTH downto 0);
signal dataLatchR : std_logic_vector(DATA_ADDR_WIDTH downto 0);
signal rdWrLowR : std_logic;
signal wrSelR : std_logic;
signal rdSelR : std_logic;
signal wrRdSelR : std_logic;
signal cycleActiveR : std_logic;
signal cycleCntR : integer;
signal stop2R : std_logic;
signal putDataOnBusR : std_logic;
signal addrDataBusR : std_logic_vector(DATA_ADDR_WIDTH downto 0);
signal addrUnsigned : unsigned(ADDR_BUS_WIDTH downto 0);
signal addrInt : integer;
signal stopCnt : integer;

begin
  -- This process is responsible for registering all the signals that are coming in
  -- from the bus master. The signals will be reset to a safe state by looking
at the polarity found in the package during reset. After reset is relinquished, they will update as specified by the bus master on the bus clock rising edge.

RegProc : process(clkIn, rstIn)
begin
if (rstIn = '1') then
  if (START_POLARITY = '1') then
    startR <= '0';
  else
    startR <= '1';
  end if;
if (DATA_SEL_POLARITY = '1') then
  dataSelR <= '0';
else
  dataSelR <= '1';
end if;
if (ADDR_SEL_POLARITY = '1') then
  addrSelR <= '0';
else
  addrSelR <= '1';
end if;
if (RD_SEL_POLARITY = '1') then
  readSelR <= '0';
else
  readSelR <= '1';
end if;
  if (WR_SEL_POLARITY = '1') then
    writeSelR <= '0';
  else
    writeSelR <= '1';
  end if;
if (RD_WR_POLARITY = '1') then
  rdWrR <= '0';
else
  rdWrR <= '1';
end if;
addrDataBusR <= (others => '0');
dataBusR <= (others => '0');
addrBusR <= (others => '0');
elif (clkIn 'event and clkIn = '1') then
  startR <= startIn;
dataSelR  <= dataSelIn;
addrSelR  <= addrSelIn;
readSelR  <= readSelIn;
writeSelR <= writeSelIn;
rdWrR     <= rdWrIn;
addrDataBusR <= addrDataBus;
dataBusR   <= dataBus;
addrBusR  <= addrBusIn;
end if;
end process RegProc;

−−
This process latches the address. The constants will tell the
−−
code whether or not the address comes from the combined address
−−
data bus or the address only bus. If it is an address only bus,
−−
it will use the polarity constant to know if it should capture
−−
when the address select line is high or low.
−−

AddrLatchProc : process(clkIn, rstIn)
begin
if (rstIn = '1') then
  addrLatchR <= (others => '0');
elif (clkIn 'event and clkIn = '1') then
  if (ADDR_DATA_SEL = '0') then — Separate addr/data bus
    if (START_POLARITY = '1') then — Active high start signal
      addrLatchR <= addrBusR;
    end if;
    else
      if (startR = '0') then — Active low start signal
        addrLatchR <= addrBusR;
      end if;
    end if;
  else — Shared addr/data bus
    if (ADDR_SEL_POLARITY = '1') then — Active high addr sel signal
      if (addrSelR = '1') then
        addrLatchR <= dataAddrBusR;
      end if;
    else
      end if;
  end if;
end process AddrLatchProc;
if (addrSelR = '0') then — Active low addr sel signal
  addrLatchR <= dataAddrBusR;
  end if;
end if;
end if;
end if;
end if;
end process AddrLatchProc;

DataLatchProc : process(clkIn, rstIn)
begin
  if (rstIn = '1') then
    dataLatchR <= (others => '0');
  elsif (clkIn'event and clkIn = '1') then
    if (DATA_VALID_SEL = '0') then — Uses separate addr and data buses
      if (START_POLARITY = '1') then
        if (startR = '1') then — Active high start signal
          dataLatchR <= dataBusR;
        end if;
      else
        if (startR = '0') then — Active low start signal
          dataLatchR <= dataBusR;
        end if;
      end if;
    else — Uses combined addr and data bus
      if (DATA_SEL_POLARITY = '1') then — active high data select signal
        if (dataSelR = '1') then
          dataLatchR <= dataAddrBusR;
        end if;
      else — active low data select signal
        if (dataSelR = '0') then
          dataLatchR <= dataBusR;
        end if;
      end if;
    end if;
  end if;
end process DataLatchProc;
dataLatchR <= dataAddrBusR;
end if;
end if;
end if;
end if;
end process DataLatchProc;

This process is responsible for figuring out the value of the read/write line.

Internal to the bus module, everything gets combined into a signal write/read line that is low during a write cycle and high during a read cycle. The atoms found in the models are used to figured out when this bus should go high or low.

ReadWriteProc : process(clkIn, rstIn)
begin
if (rstIn = '1') then
rdWrLowR <= '1';
elsif (clkIn'event and clkIn = '1') then
if ((WR_RD_SEL = '0') and (START_POLARITY = '1')) then — separate read/write lines w/ start polarity active high
if (startR = '1') then
if (WR_SEL_POLARITY = '1') then — write select active high
if (wrSelR = '1') then
rdWrLowR <= '0';
else
rdWrLowR <= '1';
end if;
else — write select active low
if (wrSelR = '0') then
rdWrLowR <= '0';
else
rdWrLowR <= '1';
end if;
end if;
elsif ((WR_RD_SEL = '0') and (START_POLARITY = '0')) then — separate read/write lines w/ start polarity active low
if (startR = '0') then
if (WR_SEL_POLARITY = '1') then — write select active high
if (wrSelR = '1') then
    rdWrLowR <= '0';
else
    rdWrLowR <= '1';
end if;

else — write select active low
    if (wrSelR = '0') then
        rdWrLowR <= '0';
    else
        rdWrLowR <= '1';
    end if;
end if;
else if ((WR_RD_SEL = '1') and (START_POLARITY = '1')) then —
combined read/write line w/ start polarity active high
    if (startR = '1') then
        if (RD_WR_POLARITY = '1') then — read is active low
            if (rdWrR = '0') then
                rdWrLowR <= '1';
            else
                rdWrLowR <= '0';
            end if;
        else — write is active low
            if (rdWrR = '0') then
                rdWrLowR <= '0';
            else
                rdWrLowR <= '1';
            end if;
        end if;
    else
        — write is active low
        if (rdWrR = '0') then
            rdWrLowR <= '0';
        else
            rdWrLowR <= '1';
        end if;
    end if;
else if ((WR_RD_SEL = '1') and (START_POLARITY = '0')) then —
combined read/write line w/ start polarity active low
    if (startR = '0') then
        if (RD_WR_POLARITY = '1') then — read is active low
            if (rdWrR = '0') then
                rdWrLowR <= '1';
            else
                rdWrLowR <= '0';
            end if;
        else — write is active low
            if (rdWrR = '0') then
                rdWrLowR <= '0';
            end if;
        end if;
    else
        — write is active low
        if (rdWrR = '0') then
            rdWrLowR <= '0';
        end if;
    end if;
end if;
rdWrLowR <= '0';
else
rdWrLowR <= '1';
end if;
end if;
end if;
end if;
end if;
end if;
end process ReadWriteProc;

This process is responsible for keeping track of an active cycle. This is kicked off by the start signal and will end when it receives a valid stop signal.

StartProc : process(rstIn, stopR, startR)
begin
if (rstIn = '1') then
cycleActiveR <= '0';
else
stop2R <= stopR;
if ((STOP_POLARITY = '1') and (stop2R = '1')) then
cycleActiveR <= '0';
elsif ((STOP_POLARITY = '0') and (stop2R = '0')) then
cycleActiveR <= '0';
else
if ((START_POLARITY = '1') and (startR = '1')) then
cycleActiveR <= '1';
elsif ((START_POLARITY = '0') and (startR = '0')) then
cycleActiveR <= '0';
end if;
end if;
end if;
end process StartProc;

This process counts the number of cycles that pass in a single bus
— cycle. It uses the signal in the previous process to know when to start and stop counting.

```
CycleCountProc : process(rstIn, clkIn)
begin
  if (rstIn = '1') then
    cycleCntR <= 0;
  elsif (clkIn'event and clkIn = '1') then
    if (cycleActiveR = '1') then
      cycleCntR <= cycleCntR + 1;
    else
      cycleCntR <= 0;
    end if;
  end if;
end process;
```

— This process uses the counter to know when to stop the bus cycle.
— When all the clock cycles have lapsed, it will assert the internal stop signal.

```
StopProc : process(rstIn, clkIn)
begin
  if (rstIn = '1') then
    if (STOP_POLARITY = '1') then
      stopR <= '0';
    else
      stopR <= '1';
    end if;
  elsif (clkIn'event and clkIn = '1') then
    if ((cycleCntR = CYCLE_CNT) and (STOP_POLARITY = '1')) then
      stopR <= '1';
    elsif ((cycleCntR = CYCLE_CNT) and (STOP_POLARITY = '0')) then
      stopR <= '0';
    elsif (STOP_POLARITY = '0') then
      stopR <= '1';
    elsif (STOP_POLARITY = '1') then
      stopR <= '1';
    end if;
  end if;
end process;
```
stopR <= '0';
end if;

end if;

end process StopProc;

stopOut <= stopR;

-- This process is responsible for generating data to the registers

regDataBi <= (others => (others => 'Z'));

-- Converting the current address to an integer

addrUnsigned <= unsigned(addrLatchR);
addrInt <= to_integer(addrUnsigned);

-- Generates one data latch for every possible address location provided that the register type is Write Only or Read/Write

RegDataGen : for i in 0 to GENERATE generate
RegDataProc : process(clkIn, rstIn)
begin

if (rstIn = '1') then
  regDataBi(i) <= (others => 'Z');
elsif (clkIn'event and clkIn = '1') then
  if (REG_RW(i) = '1') then
    if ((cycleActiveR = '1') and (rdWrLowR = '0') and (i = addrInt)) then
      regDataBi(i) <= dataLatchR;
    end if;
  elsif (WO_RO(i) = '1') then
    if ((cycleActiveR = '1') and (rdWrLowR = '0') and (i = addrInt)) then
      regDataBi(i) <= dataLatchR;
    end if;
  end if;
end if;
end if;
end process RegDataProc;
end generate;

--
-- This process is responsible for determining which data should go back
-- on the bus. This will be determined by the address bus provided that
-- the register type is Read Only or Read/Write. If it is a write
-- only register, all ones will be put back on the data bus.
--
ReturnDataProc : process(clkIn, rstIn)
begin
  if (rstIn = '1') then
    returnDataR <= (others => '0');
  elsif (clkIn'event and clkIn = '1') then
    if (REG_RW(addrInt) = '1') then
      returnDataR <= regDataBi(addrInt);
    elsif (WO_RATE(addrInt) = '0') then
      returnDataR <= regDataBi(addrInt);
    else
      returnDataR <= (others => '1');
    end if;
  end if;
end process ReturnDataProc;

--
-- This process uses the stop signal to know when to put the
-- data back on the bus. It takes into account if the stop
-- signal is active high or active low.
--
PutDataProc : process(rstIn, stopR, rdWrLowR)
begin
  if (rstIn = '1') then
    putDataOnBusR <= '0';
  else
    if (((stopR = '1') and (STOP_POLARITY = '1')) and (rdWrLowR = '1'))
      then
      putDataOnBusR <= '1';
  end if;
end process PutDataProc;
elsif (((stopR = '0') and (STOP_POLARITY = '0')) and (rdWrLowR = '1')) then
    putDataOnBusR <= '1';
else
    putDataOnBusR <= '0';
end if;
end process PutDataProc;
dataBus <= (others => 'Z');
-- Statement puts data back on the data bus.
dataBus <= returnDataR when (putDataOnBusR = '1') else (others => 'Z');
end architecture rtl;
Appendix E

**Generated VHDL for Bus Package (Example 2)**

The following code shows the generated output of the bus package, based on the model created a complex example.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

package bus_package is

constant DATA_BUS_WIDTH : integer := 31;
constant DATA_ADDR_WIDTH : integer := 31;
constant ADDR Bus_WIDTH : integer := 31;
constant START_POLARITY : std_logic := '1'; -- 0 = Negative Polarity
 ; 1 = Positive Polarity;
constant STOP_POLARITY : std_logic := '0'; -- 0 = Negative Polarity
 ; 1 = Positive Polarity;
constant RD_SEL_POLARITY : std_logic := '1'; -- 0 = Negative Polarity
 ; 1 = Positive Polarity;
constant WR_SEL_POLARITY : std_logic := '1'; -- 0 = Negative Polarity
 ; 1 = Positive Polarity;
constant RD_WR_POLARITY : std_logic := '0'; -- 0 = Write Active Low;
 ; 1 = Read Active Low;
constant DATA_SEL_POLARITY : std_logic := '1'; -- 0 = Write Active Low;
 ; 1 = Read Active Low;
constant ADDR_SEL_POLARITY : std_logic := '1'; -- 0 = Write Active Low;
 ; 1 = Read Active Low;
constant ADDR_DATA_SEL : std_logic := '1'; -- 0 = Separate Addr/Data Bus
 ; 1 = Shared Addr/Data Bus
constant DATA_VALID_SEL : std_logic := '1'; -- 0 = Separate Addr/Data Bus
 ; 1 = Shared Addr/Data Bus
constant WR_RD_SEL : std_logic := '1'; -- 0 = Separate wr/rd lines; 1 = Shared wr/rd lines
constant CYCLE_CNT : integer := 1 ;
constant GENERATE_END : integer := 2147483648;
```
constant REG_RW : std_logic_vector (GENERATE_END downto 0) :=
(others => '1');
constant WO_RW : std_logic_vector (GENERATE_END downto 0) :=
(others => '0');
type addressData is array (GENERATE_END downto 0) of std_logic_vector(
DATA_BUS_WIDTH downto 0);
end;
Appendix F

GENERATED VHDL FOR TOP MODULE (EXAMPLE 2)

The following code shows the generated output of the top module, based on the model created a complex example.

ex2_top_module.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
library work;
use work.bus_package.all;

entity top_module is
  port(
    rstIn : in std_logic;
    busClkIn : in std_logic;
    addrDataBus : inout std_logic_vector(DATA_ADDR_WIDTH downto 0);
    dataSelIn : in std_logic;
    startIn : in std_logic;
    stopOut : out std_logic;
    readSelIn : in std_logic;
    writeSelIn : in std_logic;
    addrSelIn : in std_logic;
    txClkOut : in std_logic;
    txDataOut : in std_logic);
end entity top_module;

architecture struct of top_module is

  component bus_module

    port

      ( clkIn : in std_logic;

      — Bus clock; All bus clock logic will be synchronized to this clock
```
rstIn : in std_logic;
    -- Active high reset
addrBusIn : in std_logic_vector(ADDR_BUS_WIDTH downto 0);
    -- Separate address bus
startIn : in std_logic;
    -- Kicks off active bus cycle
dataSelIn : in std_logic;
    -- Used with combined address/data bus to say data is valid
data bus data
rdWrIn : in std_logic;
    -- Combined read/write select line. Polarity determined by
    -- constants in package
readSelIn : in std_logic;
    -- Separate read select line; Asserts during an active read
cycle;
writeSelIn : in std_logic;
    -- Separate write select line; Asserts during an active write
cycle;
addrSelIn : in std_logic;
    -- Used with combined address/data bus. Asserts when data on
bus is active address data;
stopOut : out std_logic;
    -- Ends an active bus cycle.
addrDataBus : inout std_logic_vector(DATA_ADDR_WIDTH downto 0);
    -- Combined address/data bus
dataBus : inout std_logic_vector(DATA_BUS_WIDTH downto 0);
    -- Separate data bus
regDataBi : inout addressData
end component;

component customSerial
port(
    txClkOut : in std_logic;
    txDataOut : in std_logic;
    modeSel : in std_logic;
    txFifoRdClk : out std_logic;
    txFifoRdReq : out std_logic;
    txDataReg : out std_logic;
    bitRateSel : in std_logic_vector(3 downto 0);
    enable : in std_logic;
    errorInject : in std_logic);
end component;
signal regData : addressData;
signal rdReq : std_logic;
signal rdClk : std_logic;
signal q : std_logic;
component myFifo
  port(
    data : in std_logic_vector(32 downto 0);
    q : out std_logic_vector(31 downto 0);
    rdClk : in std_logic;
    rdEmpty : out std_logic;
    rdFull : out std_logic;
    rdReq : in std_logic;
    wrClk : in std_logic;
    wrEmpty : out std_logic;
    wrFull : out std_logic;
    wrReq : in std_logic);
end component;

begin
  bus_module_inst : bus_module
  port map
  ( rstIn => rstIn,
    clkIn => busClkIn,
    addrDataBus => addrDataBus,
    addrBusIn => (others => '0'),
    dataBus => open,
    dataSelIn => dataSelIn,
    startIn => startIn,
    stopOut => stopOut,
    readSelIn => readSelIn,
    writeSelIn => writeSelIn,
    rdWrIn => '1',
    addrSelIn => addrSelIn,
    regDataBi => regData
  );

customSerial_inst : customSerial
  port map(
    txClkOut => txClkOut,
    txDataOut => txDataOut,
    modeSel => '0',
    txFifoRdReq => rdReq,
    txFifoRdClk => rdClk,
    txDataReg => q,
    bitRateSel => regData(00000004)(5 downto 2),
    enable => regData(00000004)(0),
    errorInject => regData(00000004)(1)); end architecture;