autoVHDL: A Domain-Specific Modeling Language for the Auto-Generation of VHDL Core Wrappers

Erica Jones
Raytheon Missile Systems
ADDR 1
Tucson, AZ ZIPCODE
ejones@email.arizona.edu

Jonathan Sprinkle
University of Arizona, ECE
1230 E Speedway Blvd, Bldg #104
Tucson, AZ 85721-0104
sprinkle@ECE.Arizona.Edu

ABSTRACT

Reconfigurable embedded hardware is a staple of many applications in defense technology and applied engineering. The integration of various embedded hardware "cores" (i.e., the computing units) is complicated by the unintended complexities inherent in the consistent and correct construction of communication pathways—specified using VHDL. This paper presents a domain-specific modeling approach to reducing this complexity. The results include demonstration of the tool, where generated VHDL code with complex data and processing requirements is simulated.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*Very high-level languages*; I.6.3 [**Simulation and Modeling**]: Applications; I.6.5 [**Simulation and Modeling**]: Model Development

Keywords

Domain-specific modeling, embedded systems, reconfigurable computing, code synthesis

1. INTRODUCTION

Today's competitive world has put a large strain on embedded systems engineers' design efforts, and time to market and the "first time right" solution have become one of the most significant challenges that design teams face today [8]. To help curb this problem, many companies in the domain of reconfigurable embedded software and hardware rely on the reuse of VHDL to help expedite their FPGA design process. Although code reuse clearly has its advantages, the reuse of code that has been created and designed using *ad hoc* methods is nontrivial.

To make design reuse even harder, many companies are leaning towards Commercial Off the Shelf (COTS) products that allow users to provide custom code on commercial hardware. While such an approach saves development dollars when it comes to board design, it makes reuse more difficult as each COTS design usually has its own custom interface. The result is the distribution of integration code throughout a design, and at inconsistent locations based on *ad hoc* design methods.

Reconfigurable embedded processors are frequently used to perform hardware-in-the-loop and other validation schemes is widespread in industry. This application requires the repurposing of existing core engines across various bus architectures, or different speed and space constraints in order to validate a design's function behavior prior to prototype construction. An *ad hoc* construction of the necessary software to integrate these engines is frequently carried out through cloning or similar methods.

This paper mitigates the complex transition between FPGA boards through domain-specific modeling. The inherent restrictions in a DSM solution give developers a platform in which to utilize the compositional portability of different hardware (bus) platforms *independently* of how those bus platforms are specified in code. This allows the end user to be able to reuse already existing VHDL engines and cores developed using in-house or purchased designs, through logical composition directly through the bus interfaces.

The implementation of interface portability is accomplished through software synthesis from the DSM model. Developers model the characteristics of the bus to which their engine core will be attached, including routing the ports on the engine to external FPGA pins, connecting them to be driven by bus module registers, etc. Once a model is created, a model interpreter synthesizes the necessary VHDL files, which are available to any industry-standard tool.

2. BACKGROUND

Although this domain seems likely to have been impacted by domain-specific modeling, no such impact has occurred to date. The following literature review and background provide a foundation for the novelty of the work.

2.1 Reconfigurable Hardware

VHDL is designed to provide a software specification of an integrated circuit for systems named components. 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 driven by clock events [12], resulting in a time-triggered reactive system behavior.

VHDL offers many advantages to its users. One is that it can be synthesized from code into gates, or 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 [11]. FPGAs have become

popular in recent years as they offer a high execution speed that is generally obtained in hardware, but are very efficient in incorporating changes as they are reconfigurable like software [10].

2.2 Automated Firmware Generation

As is the case in traditional software, VHDL coders do not design for reuse. In [3] the authors discuss the difficulty in reusing code that hasn't been developed with reuse in mind. While reuse offers higher productivity of SOC, reusability for the current design and reusability *by others* are quite different. Better reuse is achieved when designers can reuse a logic block on a project that is completely unrelated. The authors of [3] point out that design reuse can be streamlined when happen when several rules are obeyed by all developers (including parameterizing the code, providing standalone scripts to synthesize it, etc.). However, the integration of code from various authors makes it difficult to assume those authors obeyed these rules; on the contrary, it seems that treating the component as a logic block makes more sense.

In the 1990s, auto-generated VHDL became the focal point of many research projects. The end goal of all of these projects was to have a piece of VHDL that could be generated, in order to be used from system to system. In [6] the contribution is 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. However, the main problem and set back with this implementation is that it is extremely difficult to specify event-based behaviors, even from an internal module. Such a limitation can be likened to the need to generate software from only a finite set of sequence diagrams. Also, it is not possible to determine functionality such as addition, subtracting, shifting, etc., that is not defined in the main timing diagram.

In [5] the goal is to exploit the commonality of the design functionality found in most microprocessors, by creating VHDL by a dedicated description language—essentially, a DSL. 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. Another synthesizer then generates the final VHDL. In the DSL, the users would define components and registers in the design and then associate them with classes such as "Read/Write" or "Write Only." Users can also associate a specific functionality with each register. Analysis showed that this code generation tool expedites development time, reduces error, has quick modification and regeneration ability, and easy generation of large libraries of VHDL code. This differs from the contribution of this paper in that the microprocessor interface (rather than an FPGA core) is the primary focus of that work.

Other papers take consider graphical approaches to transform a specification into VHDL. In [1] the authors model VHDL at a hierarchical level that consists of a tree of instances that are properly connected with applicable constraints. 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. The drawback to this approach is that there are no formal specifications that are used to create the VHDL output. While the authors found that this led to speedy code generation, this can be a drawback to some users because informal specification of behavior.

Recently there has been significant focus on intellectual property (IP) core generation. These cores can be provided by FPGA venders such as Altera or Xilinx or purchased from other third party vendors. These IP cores are self contained modules that perform a specific function. 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. In [7] the work focuses on these IP cores and turning them into C function calls that can also be run-time configurable on FPGAs. Results show this tool has reasonable execution overhead as well as reasonable execution time. While this is a good start to the automation process, C level programming that synthesizes into HDLs can be very inefficient and problematic. Although the result 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 part of the high-level design.

Nearly two decades after automated firmware generation research began, the approaches haven't caught on because a "one size fits all" approach does not work. It is extremely difficult to create a tool that will do everything that the VHDL language will do but better and more efficient—this validates the approach taken in this paper, to attack a specific domain that is consistently used by practitioners.

2.3 Wrapper Design

VHDL wrappers have been implemented in the past, but most have avoided a domain-specific approach. In [2], the authors use Java 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.

Turning from testing to time to market, [13] looks at the 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. 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 that tool understand the correct execution order per data dependencies. The tool can generate both VHDL firmware and C-code for software. A key limitation when compared to the approach in this paper is the lack of integration of various cores from a high-level specification.

Finally, in [4] a technique is outlined to automate the process of generating HDL descriptions between mismatched IP protocols. The authors were able to use their tool to create many bridges and wrappers to function inside a SOC, and to exploit the fact that many SOC use complex protocols in which to compete with one another. It is important to note that one problem the authors faced was a lack of internal and external standard bus architecture. 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 approach addresses issues of data width mismatch, data type mismatches, pipelined operations, complex branching, and different clock speeds. However, that work is a more domain-independent approach, and thus the integration specification requires significant steps that are consistently the same in the domain approached in this work.

3. THE PROBLEM: RAPID INTEGRATION, POTENTIALLY COMPLEX SIGNALS

The reuse of VHDL core modules is common in practice, but largely through ad hoc methods. The problem, therefore, is to reduce the amount of specification needed to reuse VHDL core modules, without sacrificing the necessary configurability options. Additional details of the domain are provided in this section in order to demonstrate the depth of the problem.

3.1 Domain-Specific Concepts

In many designs of this domain, a generic template may be used to abstract the various design axes. The simple design includes an engine core and a bus module, and a data FIFO (first-in, first-out) buffer.

Consider an example where an engineer would like to use a proprietary component (e.g., the Xilinx CoreGen) and hook it directly up to the bus. A high level block diagram of this design can be seen in Figure 1. The figure has three main modules.

3.1.1 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.

3.1.2 Engine Module

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.

3.1.3 Top Module

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.

3.1.4 Execution Semantics

For this example, a data bus timing diagram was generated. This timing diagram can be referenced in Figure 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.

3.2 Scope of Cores and I/O Constraints

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 the case study; this module takes an 8 bit unencoded value and changes it into an encoded 10 bit value. However, in order to generically address the problem in this work, generically compatible cores should be able to be reused.

Thus, the solution should consider synchronous engine modules driven by a clock external to the FPGA. Such cores should be configured to take various inputs and outputs (in the form of either data signals/buses, control signals, etc.).

Notably, the solution should permit multi-input, multi-output models, as many core modules have more than one output signal or bus. Outputs are (in the domain) connected to the top module, and may be data signals, status bits/signals, etc.

4. APPROACH AND IMPLEMENTATION

The domain-specific modeling approach is implemented through the use of the Generic Modeling Environment (GME). In this tool, a domain-specific language named autoVHDL is modeled through a metamodel. That domain-specific language is then mapped to guarantee the necessary execution semantics through a model interpreter that synthesizes the necessary VHDL code to integrate the FPGA core to the various components of the design.

4.1 Metamodel

An abridged metamodel for the autoVHDL domain-specific language is given in Figure 3. At the 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 that is 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 too will just define to the tool how the engine interacts with other design elements. Using this 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.

Due to space constraints, a full summary of all contained objects inside these three key models is not possible. A full

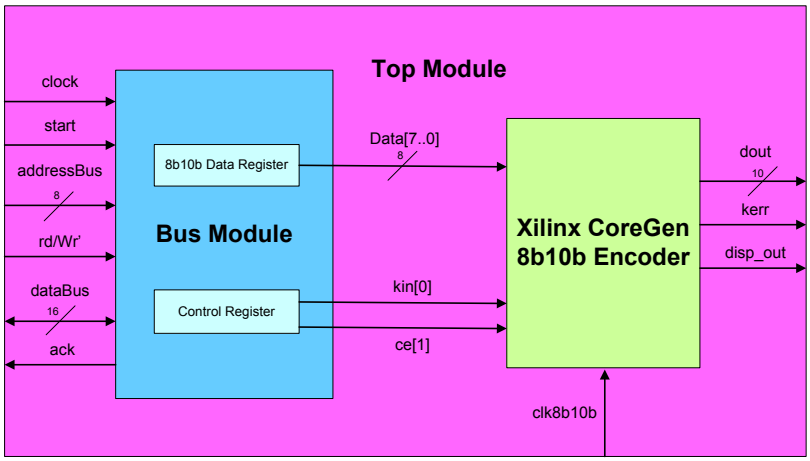

Figure 1: Typical design block diagram (without data FIFOs). The main modules are the bus, an engine module (the CoreGen Encoder), and the container (Top Module).

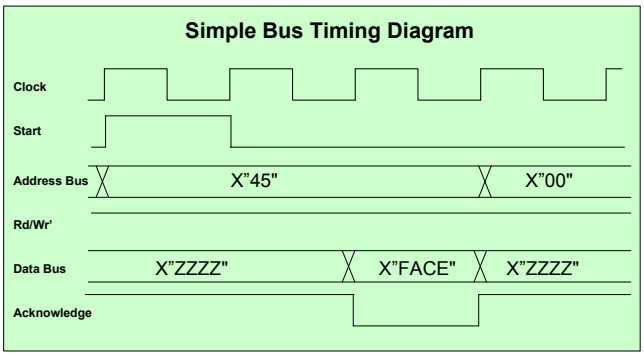

Figure 2: Example Bus Module Timing Diagram

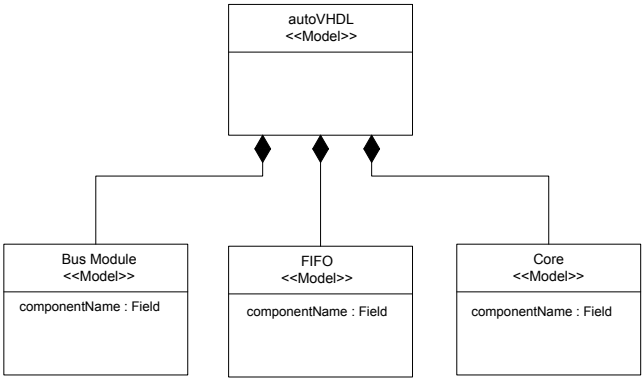

Figure 3: autoVHDL language metamodel (abridged for brevity).

description of all available blocks and types is available to the interested reader in a full version of this paper, found in the lead author's thesis [9].

The types of the contained objects correspond to the domain-specific attributes of a bus timing diagram. For example, clock, start, read, write, and read/write bus signals (among many others) are found in the domain, but not all signal types are found in each bus. The benefit of the domain-specific approach in enumerating these types is that the user has a concise specification of the bus, and can compare this specification (in a single location) to the example timing diagram—which usually drives the specification of the integration models when the various VHDL core wrappers are written by hand.

4.2 Modeling Language Semantics

Model based program synthesis is performed on a completed model. The implementation utilized the various technologies available with the GME tool. The synthesis code utilizes the Visitor design pattern to traverse the models and atoms defined by the modeler. The traversal first collects all the bus module, FIFO, and core models that the user has defined, and then traverses the contained objects inside those models. Once this is complete, the specific attributes that have been set for each model can be extracted and used to create the three VHDL output files.

4.2.1 Bus module VHDL file

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. This file defines the characteristics of the data bus, including the width of the data and address buses, classification of signals, registration of all incoming signals (including the best practices of resetting all signals to a safe state), generating data to the registers, conversion of data types, etc. While this code is straightforward, its menial nature lends itself to cloning—and the various problems associated with that software practice when performing system integration.

4.2.2 Bus package VHDL file

In order for the bus module to work properly, the bus

Slice Logic Utilization	Used	Available	Utilization
Number of Slice Registers	1,296	19,200	6%
Number used as Flip Flops	1,294		
Number used as Latches	2		
Number of Slice LUTs	478	19,200	2%
Number used as logic	478	19,200	2%
Number using O6 output only	446		
Number using O6 and O6	32		
Slice Logic Distribution			
Number of Occupied Slices	423	4,800	8%
Num. of LUT FF pairs used	1,347		
Number with an unused FF	51	1,347	3%
Number with an unused LUT	869	1,347	64%
Num. fully used LUT-FF pairs	427	1,347	31%
Number of unique control sets	68		
IO Utilization			
Number of bonded IOBs	27	220	12%
Specific Feature Utilization			
Number of BUFG/BUFGCTRLs	2	32	6%
Number used as BUFGs	2		

Table 1: Device Utilization Summary for design in Figure 1.

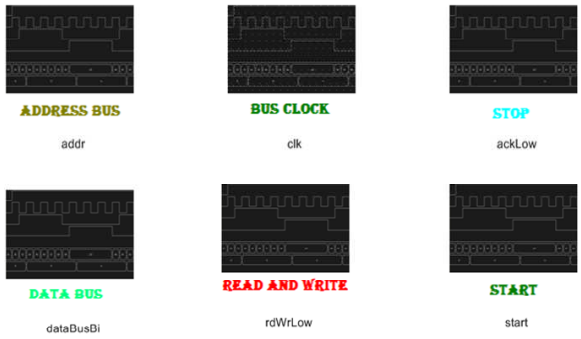

Figure 4: Internal components of bus module from the case study. These atomic types represent the kinds of signals for firmware integration (screenshot from the autoVHDL modeling language.)

package is required. The bus package defines the specifics of the bus module. It tells the bus module which atoms were used or not used in the model so that proper bus function can be replicated. This file defines assumed polarities of the read and write signals, whether or not address and data buses are shared, etc.

4.2.3 *Top module VHDL file*

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). In short, this file is the integration file that connects all the communication pathways. The model synthesizer will be responsible for traversing through the remaining models and their associated atoms and determining how everything must be connected together.

5. RESULTS

To validate the code generator, a model representing the design shown in Figure 1 was created and the three output VHDL files were generated. As mentioned, 3 VHDL files were generated, comprised of 649 lines of VHDL code (not including the sizes of various included libraries). The generated code was then used by industry standard tools and hardware for validation. Although the number of generated lines is not large, the generated code is consistent with best practices, and avoids errors in cloning (which are prominent in this domain) that are discovered only after costly loss of time during simulation and debugging.

Validation was carried out using a Xilinx Virtex 5 device target. Timing reports from Synplify[1] and device utilization (using Xilinx's proprietary FPGA synthesis tools) were obtained from the generated output. The Synplify performance summary indicates the worst slack in the design is $-468ms$, and that the design will run at $320.8MHz$. The design summary (indicating device utilization of the design) is summarized in Table 1.

It is important to note that depending on the selected device (in this case, a Xilinx Virtex 5), speed will vary as different parts have different resources. Newer devices will be able to push the bus module faster than older, slower parts. Additionally, this design may not fit on small devices due to the register size. This is addressed in the next section.

[1]Synplify, a product of Synopsis, performs various optimizations of a design prior to synthesizing FPGA logic.

Finally, these results are valid for the bus module, as the device utilization for alternative cores will vary based on the selected core. A simulation of the generated VHDL code can be seen in Figure 5.

6. CONCLUSION AND FUTURE WORK

The use of reconfigurable embedded processors to perform hardware-in-the-loop and other validation schemes is widespread in industry. This application requires the frequent repurposing of existing core engines across various bus architectures, or different speed and space constraints in order to validate a design's function behavior prior to prototype construction. The state of the art is unfortunately ad hoc construction of the necessary software to integrate these engines, frequently through cloning or similar methods.

This paper summarized the autoVHDL tool, which permits the integration of VHDL core engines through a domain-specific specification language that permits the structure of the integrated system to be rapidly given. The tool permits a variety of engine cores to be integrated based on the characteristics of the communication pathways, and the domain-specific approach is motivated based on the fixed number (and type) of communication pathways that are available in a particular design. The semantics of the tool permit the generation of VHDL code that serves as a wrapper to the designated VHDL core.

Future work includes optimization of the structure of the generated code, independent of the structure of the model. For instance, at this time a register is generated for every address space regardless if the design actually uses it. A given design could fit on smaller devices if the bus module were able to prune unused address spaces from the generated code. Additional (domain-specific) enhancements include default population of model attributes, based on anticipated usage. For example, models could be instantiated to represent mainstream commercial buses without any user intervention (e.g., the PCI bus). These kinds of automation further reduce the possibility of improper specification—a key to this domain-specific approach.

Finally, the modeling language does not utilize any sophisticated constraints; rather, it prints warning statements to the console to warn the user that optimal output functional-

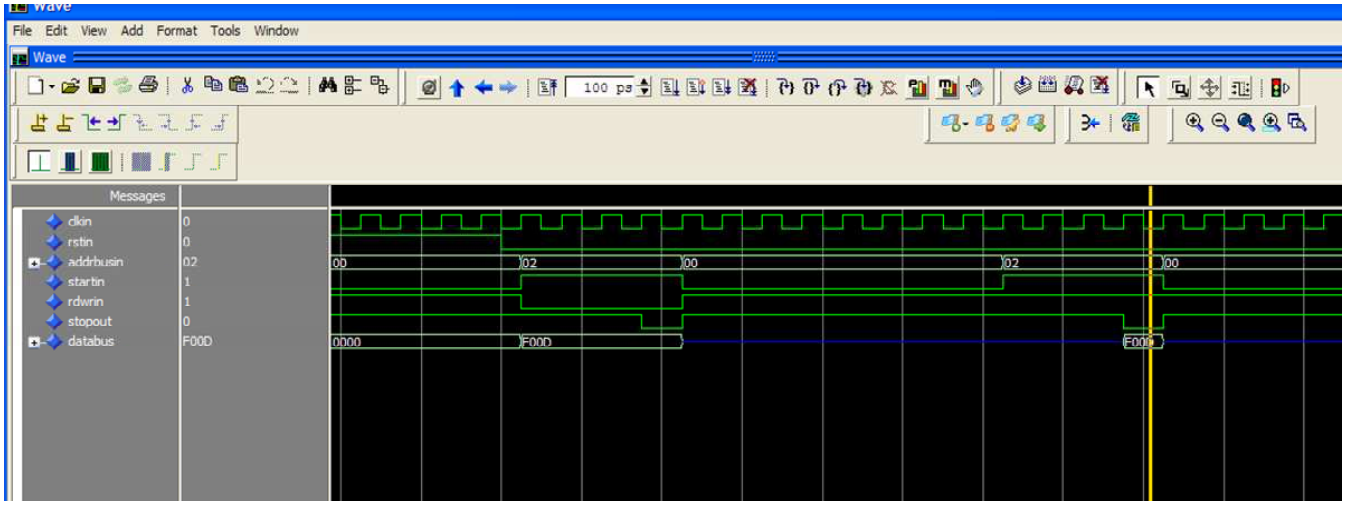

Figure 5: Simulation result of generated VHDL code.

ity 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 generated. This would minimize later frustration with the tool's output files.

7. ACKNOWLEDGMENTS

This work is supported by the National Science Foundation under awards CNS-0915010, CNS-0930919.

8. REFERENCES

[1] B. Abbott, T. Bapty, C. Biegl, G. Karsai, and J. Sztipanovits. Model-based software synthesis. *Software, IEEE*, 10(3):42–52, May 1993.

[2] M. Baláz, J. Beckova, and E. Gramatová. Wrapper tool–learning and application of digital system testability to SoC cores. In *Computational Technologies in Electrical and Electronics Engineering (SIBIRCON), 2010 IEEE Region 8 International Conference on*, pages 384–389, 2010.

[3] J. Chang and S. Agun. Design-for-reusability in VHDL. *Computing Control Engineering Journal*, 12(5):231–239, Oct. 2001.

[4] V. D'silva, S. Ramesh, and A. Sowmya. Bridge over troubled wrappers: automated interface synthesis. In *VLSI Design, 2004. Proceedings. 17th International Conference on*, pages 189–194, 2004.

[5] S. Gastaldello, A. Lometti, and G. Traverso. VHDL code automatic generation for repetitive designs. In *ASIC Conference and Exhibit, 1995., Proceedings of the Eighth Annual IEEE International*, pages 97–100, Sept. 1995.

[6] W. Grass, C. Grobe, S. Lenk, W.-D. Tiedemann, C. Kloos, A. Marin, and T. Robles. Transformation of timing diagram specifications into VHDL code. In *Design Automation Conference, 1995. Proceedings of the ASP-DAC '95/CHDL '95/VLSI '95., IFIP International Conference on Hardware Description Languages; IFIP International Conference on Very Large Scale Integration., Asian and South Pacific*, pages 659–668, Aug. 1995.

[7] Z. Guo, A. Mitra, and W. Najjar. Automation of IP core interface generation for reconfigurable computing. In *Field Programmable Logic and Applications, 2006. FPL '06. International Conference on*, pages 1–6, Aug. 2006.

[8] G. Hughes. Striking a new balance in the nanometer era: first-time-right and time-to-market demands versus technology challenges. In *Design, Automation and Test in Europe, 2005. Proceedings*, volume 1, page 3, 2005.

[9] E. Jones. Auto-generation of VHDL core wrappers using modeling. Master's thesis, University of Arizona, August 2011.

[10] A. Kostadinov and G. Mihov. Reconfigurability–a new feature of the hardware. In *Electronics Technology: Meeting the Challenges of Electronics Technology Progress, 2004. 27th International Spring Seminar on*, volume 3, pages 457–460, May 2004.

[11] J. Pick. VHDL synthesis techniques and recommendations. In *ASIC Conference and Exhibit, 1995., Proceedings of the Eighth Annual IEEE International*, pages 389–394, Sept. 1995.

[12] M. Shahdad. An overview of VHDL language and technology. In *Design Automation, 1986. 23rd Conference on*, pages 320–326, 1986.

[13] M. Tranchero and L. Reyneri. Automatic generation of VHDL code for self-timed circuits from simulink specifications. In *Electronics, Circuits and Systems, 2007. ICECS 2007. 14th IEEE International Conference on*, pages 287–290, Dec. 2007.