VHDL Code Generation in the Ptolemy II Environment

Man-Kit Leung
Terry Esther Filiba
Vinayak Nagpal

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2008-140
http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-140.html

October 28, 2008

VHDL Code Generation in the Ptolemy II Environment

Terry Filiba, Man-Kit Leung, Vinayak Nagpal

December 18, 2006

Abstract

It is becoming increasingly popular to describe real time signal proessing systems targetted for FPGA or ASIC implementation using structural signal flow graphs. We have implemented support for generation of synthesizeable as well as testbench VHDL code from Ptolemy II models. A helper based approach borrowing heavily from the existing Ptolemy II C *code generation framework is used. This work demonstrates the extensibility of the helper based code-generation approach and sets the stage for future research in synthesis of efficient hardware descriptions from heterogenous visual models.*

1 Introduction

The use of Field Programmable Gate Arrays (FPGAs) is becoming increasingly common in modern embedded systems, bridging the design space between software and hardware by presenting a layer of configurable logic which is often termed *gateware*. There is increasing demand for design environments which allow modeling, simulation and synthesis of software, gateware as well as hardware components simultaneously in a unified framework. Commercial tools that synthesize gateware from high level actor oriented structural descriptions have been popular among system designers for some time. For example, the user community of the BEE2 (Berkeley Emulation Engine) FPGA platform liberally harnesses the Xilinx System Generator toolkit integrated with Mathworks Simulink for designing digital signal processing subsystems.

We believe that the Ptolemy II heterogeneous systems modeling framework can provide a richer environment for hardware-gateware-software co-design. Ptolemy II supports multiple well defined models of computation, it has a structured language for higher order components and the software framework is available in the public domain. Ptolemy II already supports software code generation in the *C* programming language. In this project we have complemented that with support for VHDL code generation.

With this work we aim to eventually provide an improved visual modeling and design environment for the FPGA based real time signal processing community.

2 Related work

It has long been realized that DSP applications are more intuitively described using visual signal flow graphs. Commercial tools like Xilinx System Generator (XSG) [3] provide a library of Simulink blocks which can be used as primitives for building DSP hardware. The library consists of multiplexors, adders, registers, memories etc and each can be parameterized for implementation on Xilinx FPGA's. It allows designers to perform bit and cycle accurate simulations of their system and later translate the design to Xilinx compatible gateware. For translation XSG internally calls Xilinx Core-Gen, a software tool that allows generating high level components like multipliers and memories with desired specification to directly map onto lower level Xilinx primitives. This approach limits its use to Xilinx FPGAs. It is non-trivial to re-target a XSG translated design to another FPGA family or an Application Specific In-

tegrated Circuit (ASIC). Other commercial tools like Synplify DSP also based on Simulink provide similar features but can translate designs into platform independent specifications. All such systems use execution semantics closer to the Synchronous Reactive (SR) domain in Ptolemy II. This is a natural outcome of the fact that synchronous logic is best described by SR semantics when all clock domains in a design differ by integer multiples of a base frequency. Truly asynchronous boundaries can be modelled using Discrete Event semantics which underly hardware description languages like Verilog and VHDL. As designs get larger and span many clock boundaries the designer may need more flexibility with the expressiveness and abstraction of modeling semantics. For example, parts of the system which are synchronous and are timing or resource critical may need to be modelled using SR semantics, while asynchronous interfaces may need DE semantics. At the top level of the system and for components that do not need a lot of optimization, higher level semantics like that of Synchronous Data Flow (SDF) may be preferable. SDF semantics implicitly infer queues between actors to take care of differences in firing rates (clock speeds) between blocks. Translation from SDF to gateware is non trivial and there has been previous work in attempting this using Ptolemy Classic [5].

The introduction of VHDL code-generation in Ptolemy II will set the stage for further research in the area of improved heterogeneous design environments for gateware and hardware designers. In this work we claim generation of behavorial Register Transfer Level (RTL) VHDL which is independent of implementation platforms and can be targeted to ASICs as well. Our code-generation framework borrows heavily from the existing *C* code-generation infrastructure in Ptolemy II.

3 Ptolemy II

Ptolemy II (PTII) is a software framework developed to study modeling, simulation, and design of concurrent, real-time systems [4]. It supports compositional and hierarchical modeling. It is based on an actor-oriented language, in which actors are block components that communicate with each other through port connections. While the placement of actors and their interconnects gives structural syntax to a model, the execution semantics of a model is defined by its director. A director represents the model of computation (MoC) that governs the communication between actors.

3.1 Domain (Model of Computation)

A Ptolemy II domain is the implementation of a specific model of computation. The terms *domain* and MoC will be used interchangeably throughout the paper. The $Dataflow$ (DF) domain is a common communication semantics, where actors are fired whenever input tokens are present. Input buffers are used to prevent overflow. DF is generally considered as a timeless domain. The concept of time is abstracted to simply describe the partial order of token arrival to buffers. On the other hand, the $Synchronous/Reactive$ (SR) domain makes use of a very different abstraction of time, which is represented by discrete intervals separated with clock ticks. All actors are required to fire and produce one output token every tick of the clock. Their outputs need to follow a fixed-point semantics, meaning that outputs may not change their values until the next clock tick. For further details of the PTII domains, please refer to the Ptolemy II documentation[4].

Given that every domain offers a particular kind and degree of abstraction, part of a designer's job is to choose the appropriate domain for a model. A properly chosen domain simplifies design complexity. Furthermore, composing multiple domains in the same model gives designers a very powerful tool in specifying designs. This is called, heterogeneous modeling[2], and has been a main focus of the Ptolemy II framework.

3.2 Code Generation

Building code generation support in Ptolemy II aims to automate the process of targeting simulation models to actual application platforms[5]. The PTII code generation framework takes a helper-based approach,

in which a set of helper components are used to generate code for the PTII actors. Each helper has a one-to-one correspondence to a actor. As a logical separation between their domains of relevance, helpers are code generation components while actors are simulation components. The code generation process is an interaction between the code-Gen kernel and a set of helpers. The kernel stitches together code blocks harvested by the helpers. Each helper has a code template that contains code blocks written in the target language and macros. The macro language allows code block parameterization. A detailed explanation of the code generation framework and the macro language can be found in the Ptolemy II documentation[4].

The logic behind a helper-based framework is based on separation of concerns to reduce generality. This means having a light-weight kernel and a library of helper classes that spread out complexity. The architecture allows incremental and rapid development because helpers classes contains simple logic and code blocks that are mutually exclusive. As another theory of the helper framework, a different set of helpers implies code generation of a different target language. Currently, the code generation framework only supports C code generation; thus, adding VHDL code generation would prove this hypothesis.

4 System Implementation

Our goals are to, first, generate VHDL code from Ptolemy II models and, second, ensure bit-and-cycle consistency between Ptolemy II simulation and VHDL implementation. Along with that, our VHDL code generator primarily targets DSP applications, which operate on specific set of primitive components, data structure and communication semantics. Our implementation requires us to create these primitive components as a subset of actors in the Ptolemy framework. In order to map signals onto DSP hardware, we require our actors to operate on strictly fixed-point data type. Our implementation currently supports the SR domain. However, the proposed code generation framework, by no means, is limited to a single domain. Support for code generation of heterogeneous models will be future work.

4.1 Ptolemy II Actors

To support hardware generation, the Ptolemy II actors need to convey information to the code generator that must be provided by the user. In order to obtain the information we have created a new set of actors which have hardware specific parameters.

Ptolemy II implements polymorphic actors that accept and output data of varying types. For example, the AddSubtractor actor is capable of operating on types such as double, fixed point, integer, and even strings. Because VHDL is a strongly typed language, the input and output ports in actors that generate VHDL must be constrained to a single type. The fixed point type is typically used by DSP hardware designers. This type allows for the representation of fractional values, in contrast to the integer type, without requiring complex hardware which is necessary for computation on double precision data. For this reason, our current subset of actors enforce that all input and output data be of the fixed point type. The code generation is not restricted to this type. Developing actors that operate on other types would only require a new Java actor and a helper for code generation (described in Section 4.3) to be written.

The bit widths of the ports in any component must be fixed in hardware. To simulate this in the Ptolemy II actors there is a precision parameter that is used to define the data size and the location of the binary point. The overflow parameter and rounding parameters specify how to handle output results that cannot be represented by the output precision. The overflow and rounding strategies can be translated into VHDL to ensure the simulated behavior is consistent with the hardware behavior.

Adding multiple registers at the end of a component in VHDL causes the synthesized component to be pipelined. This register re-timing provides better performance since the clock need not be limited by the timing of the entire component. The addition of a latency parameter in Ptolemy II actors is used to indicate the number of pipeline stages to be generated in the VHDL component.

4.2 VHDL Library

The Code Generation framework maps a Ptolemy actor to a primitive block in a library of VHDL signal processing primitives. The mapping from Ptolemy to the library is one to many. The primitives include blocks like a signed fixed point two input adder, an unsigned subtracter etc. These blocks accept parameters like bit widths, fixed point precision, choice of truncation or rounding on overflows etc. as VHDL generics. The fixed point arithmetic is implemented using backward compatible packages released by IEEE for the proposed fixed point support in VHDL-2006 standard [1]. Depending on parameters supplied by the user to the Ptolemy actor, the Code Generator chooses a primitive block to instantiate and stitch into the VHDL output. These primitives accept latency values and instantiate register chains to allow timing optimizations by register retiming. This has been tested using Synplify synthesis tools. In addition to providing RTL primitives, the library also supports special primitives which do not synthesize into hardware but can emulate the behavior of corresponding Ptolemy actors and provide a testing framework to run in any VHDL simulator.

4.3 Code-Gen Kernel and Helpers

The design of the code generator is a mapping of VHDL code generation on the general code-gen framework. This forseeably requires two changes: a new implementation of the kernel and a set of VHDL helper classes (and code templates) for the newly created simulation actors. As mentioned before, the code-gen kernel interface is light-weight and contains only a small set of functions. The VHDL code-gen kernel implements these functions to arrange the order of code blocks and write the code to a list of files with proper headers and footers.

The code generator output is a *structural* VHDL implementation. (i.e. the model is represented as a structured signal flow graph) consisting of VHDL library elements (described in Section 4.2) as atomic building blocks. As compared to sequential code, this is easier to generate. The generated VHDL top entity block consists of the component declarations and instantiation blocks of sub entities. Each code template file contains blocks of code for the declaration and instantiation of a library primitive. These code blocks are parameterized to specify which library primitive to instantiate and what parameters to pass to it. A typical VHDL code template file (IntegerCounter.vhdl) is illustrated below:

```
/*** componentBlock ***/
component ptcounter is
    GENERIC (
        WIDTH  : INTEGER := 32;
        USE_ENABLE : boolean := TRUE;
        RESET_ACTIVE_VALUE : std_logic := '0';
        ENABLE_ACTIVE_VALUE : std_logic := '0';
        WRAP : boolean := TRUE
    );
    PORT (
        clk  : IN std_logic ;
        reset : IN std_logic ;
        enable : IN std_logic ;
        output : OUT std_logic_vector (WIDTH-1 DOWNTO 0)
    );
end component ptcounter;
/**/

/*** instantiationBlock ($width, $useEnable, $wrap, $enableSi
$actorSymbol(instance): ptcounter
    GENERIC MAP (
        WIDTH => $width,
        USE_ENABLE => $useEnable;
        WRAP => $wrap
    )

    PORT MAP (
        clk => clk,
        reset => reset,
        enable => $enableSignal,
        output => $ref(output)
    );
/**/
```

A helper is responsible for harvesting the proper code block by supplying the right parameters. The parameters are specified in a simple macro language that the code generator implements. The macros are used to provide unique labels and instance-specific values. The kernel generates compliant VHDL output by collecting and arranging the harvested code blocks from every helper in the model.

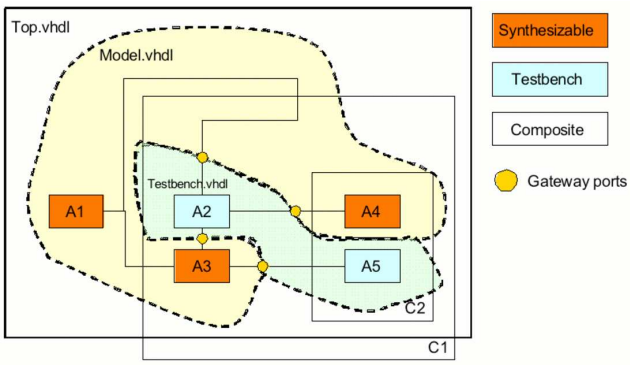

Figure 1: Synthesizable vs. Testbench Files

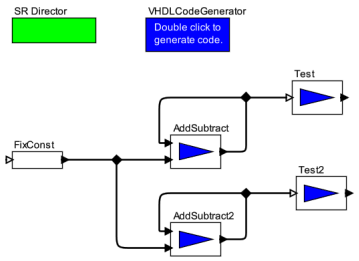

Figure 2: This models the accumulation of a fixed-point value.

4.4 Testing Methodology

Ptolemy II provides a Test actor which uses regression testing to check that the output of a simulation is consistent with the output observed by the actor in previous simulations on the same system. We utilize this actor to test the VHDL being generated by the system. When generating VHDL the Test actor will create two signals. One outputs the values that were observed by the Ptolemy II Test actor and the second outputs the values that are being generated by the hardware. If the two signals (which can be viewed with a VHDL simulator) are equal, then the generated VHDL is consistent with finite simulation of the Ptolemy II design at the bit and cycle level.

It is useful to distinguish parts of the design that are for testing from parts that should be synthesized into hardware. The Ptolemy II actors include a *synthesizable* parameter to facilitate this. During code generation, any actors marked as synthesizable will be put into separate (RTL) file from the other non-synthesizable or testbench (TB) actors. The code generator also outputs a file which instantiates and stitches together the RTL and TB. The Ptolemy II hierarchy poses problems when dividing the actors because both non-synthesizable and synthesizable actors may be in the same composite actor (refer to the composites in Figure 1). The hierarchy is flattened in order to partition the synthesizable and non-synthesizable actors. Retaining the heirarchy of composites in the model to the generated hardware is not seen as particularly useful as we do not expect users to hand tweak the VHDL generated by the code generator. Each actor is placed in its appropriate file and any connection that spans the partition creates a VHDL port in both files.

Automatically partitioning the generated VHDL can provide many features to the user. The generated code is generic so other VHDL test bench files can be swapped in so long as they have the same ports. Also, actors that are placed in the test bench file can take advantage of non-synthesizable VHDL as only the RTL output is passed to synthesis tools which translate VHDL into gateware/hardware implementations. For example, the Test actor and other testing only actors will never be included in the RTL output. Such testing only actors can use all the testing features provided by the VHDL language. One can also use otherwise synthesizeable actors for building a testbench and mark them as non-synthesizable to keep them out of the RTL output. This simplifies the design of test cases and actors that can simulate complex behavior.

5 The Add/Sub Accumulator Example

To show how VHDL code is actually generated, let's go through the code generation process for the following model:

In this model, the FixConst and AddSubtract actors are synthesizable components, while the Test

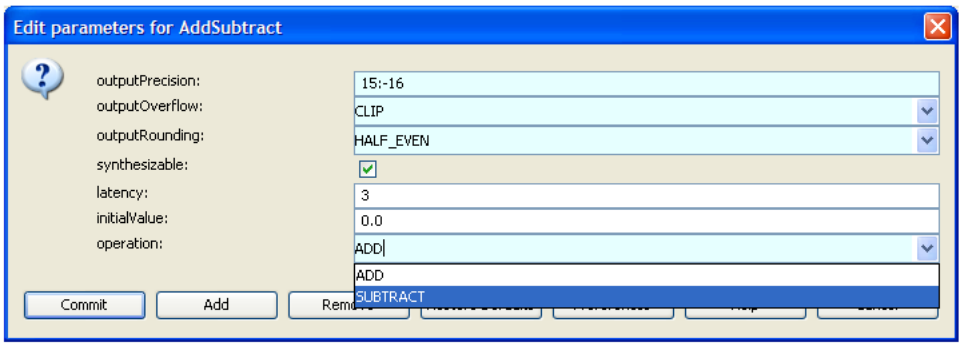

Figure 3: The parameters for the AddSubtract actor.

instances are testbench (non-synthesizable) components. A actor can be parameterized at the GUI level, and its helper uses the information to choose and give arguments for code blocks. As shown in Figure 3, the AddSubtract actor has a parameter called *operation* which tells the code-gen helper whether an adder or subtracter circuit needs to be instantiated. Other parameters have similar effects in influencing the resulting circuit. For example, the value of the *latency* parameter decides whether combinational or pipelined logic is produced.

By invoking the VHDL code generator (the box in blue), a synthesizable entity, a testbench entity and a top entity are generated. Generating the synthesizable entity requires code blocks harvesting from the AddSubtract and FixConst helpers, while the testbench entity requires code blocks from the two instances of the Test helper. The two entities are connected by gateway ports (See Section 4.4). The top entity is the instantiation of the RTL and TB entities.

6 Future work

As the next step, we want to extend the code generation framework to support other domains (i.e. *Dataflow, Discrete Event, FSM (Finite State Machnes)*, etc.) and heterogeneous models. We would also like to add hardware benchmarking features, against other comparable tools such as XSG, in area and timing comparison.

7 Acknowledgments

This project receives a joint support from the Ptolemy, BEE, and CASPER groups. We want to thank Professor Edward A. Lee, Dr. Chen Chang, Dr. Dan Werthimer, Christopher Brooks, and Pierre Droz for their mentoring and advice.

References

[1] David Bishop. Fixed point package user's guide.

[2] C. Brooks, E. A. Lee, S. Neuendorffer X. Liu, Y. Zhao, and H. Zhang. Heterogeneous concurrent modeling and design in java, July 2005.

[3] J. Hwang, B. Milne, N. Shirazi, and J. Stroomer. System level tools for DSP in FPGA's, 2001.

[4] J. Davis II, M. Goel, C. Hylands, B. Kienhuis, E. A. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Reekie, N. Smyth, J. Tsay, and Y. Xiong. Overview of the ptolemy project, July 1999.

[5] M. C. Williamson. Synthesis of parallel hardware implementations from synchronous dataflow graph specifications. Technical Report UCB/ERL M98/45, University of California, Berkeley, June 1998.