Hardware Description Languages

Gaurav Mehta, Sridhar Kintali

University of California, Santa Barbara

{g_mehta, skintali}@cs.ucsb.edu

Abstract: The past few years have seen a lot of work being done in Hardware Design methodologies, ranging from the addition of behavioral synthesis to working in mainstream software programming languages for describing the hardware. This report looks at the traditional hardware design ways and the new approaches that have been adopted over the years for increasing the designer's productivity and reducing the error rate in hardware design descriptions.

1. INTRODUCTION

In electronics, a Hardware Description Language or HDL is any language from a class of computer languages for formal description of digital electronic circuits. It describes the behavior of an electronic circuit or system from which the physical circuit or system can then be attained. The principal feature of a HDL is that it contains the capability to describe the function of hardware independent of implementation. It can describe the circuit's operation, its design and organization, and tests to verify its operation by means of simulation. In contrast to a software programming language, an HDL's syntax and semantics include explicit notations for expressing time and concurrency, which are the primary attributes of hardware.

Descriptions can be based on structure or behavior. Structural description is a textual replacement for a graphical schematic. The design approach follows a hierarchical model. The lowest level in the hierarchy is composed of primitives provided by the specific language being used. Behavior or functional approach describes what a module does. It does not care about the underlying implementation. The circuit for a module is generated in the synthesis step. Languages whose only characteristic is to express circuit connectivity between hierarchies of blocks are classified as 'netlist' languages.

A 'netlist' describes the connectivity of an electronic design. They usually convey connectivity information and provide nothing more than instances, nets, and perhaps some attributes. Most netlists either contain or refer to descriptions of the parts or devices used. Each time a part is used in a netlist, this is called an 'instance'. Thus, each instance has a 'master', or 'definition'. These definitions will usually list the connections that can be made to that kind of device, and some basic properties of that device. Nets are the 'wires' that connect things together in the circuit. Netlists can be either *physical* or *logical;* either *instance-based* or *net-based;* and *flat* or *hierarchical.*

HDLs have two purposes. First, they are used to write a model for the expected behavior of a circuit before that circuit is designed and built. The model is fed into a computer program, called a simulator, which allows the designer to verify that his solution behaves correctly. Second, they are used to write a detailed description of a circuit that is fed into another computer program called a logic compiler. The

output of the compiler is used to configure a programmable logic device that has the desired function. Often, the HDL code that has been simulated in the first step is re-used and compiled in the second step.

Figure 1, shows a general design flow using HDL. We start the design by writing the VHDL code. The first step in the synthesis process is compilation. Compilation is the conversion of high-level HDL language, which describes the circuit at the Register Transfer Level (RTL), into a netlist at the gate level. The second step is optimization which is performed on the gate-level netlist, for speed or for area. At this stage, the design can be simulated. Finally, a place-and-route (fitter) software will generate the physical layout for a PLD/FPGA chip, or will generate the masks for an ASIC.

The rest of the paper is organized as follows. Section 2 describes the synthesis step in the design cycle which includes the compilation of a HDL RTL code into a netlist and the optimization of that netlist. Several intermediate steps in the synthesis process that are not depicted in the figure are also discussed. The place-and-route step is briefly described in Section 3 followed by a overview of programming language based HDLs in Section 4 and Section5. Section 6 concludes.

2. SYNTHESIS

Synthesis is a process by which an abstract form of desired circuit behavior (typically register transfer level (RTL)) is turned into a design implementation in terms of logic gates. Common examples of this process include synthesis of HDLs, including VHDL and Verilog. Some tools can generate bitstreams for programmable logic devices such as PALs or FPGAs, while others target the creation of ASICs

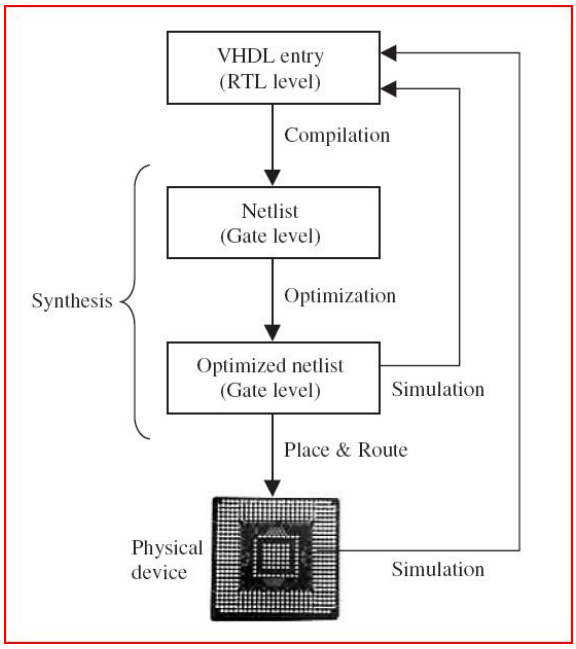

Figure 1 HDL Design Flow (Volnei A. Pedroni, 'Circuit Design with VHDL')

2.1 Behavioral Synthesis

If the description provided is behavioral then the first step is to convert that behavioral description to synthesizable structural description, to be more specific; RTL level description. This is done by behavioral synthesis. Behavioral synthesis is an automated design process that takes in the behavioral or algorithmic level description and creates the corresponding RTL implementation out of it. It is employed as a part of the behavioral design flow and as can be seen, it tends to increase the level of abstraction of the whole hardware design process. It was brought into the field of hardware design so as to increase the productivity of the designer and reduce the scope of error in the process.

The way the behavioral synthesis tools work is that they take in the behavioral or algorithmic description and then do a cycle by cycle analysis fetching and creating details needed for the hardware implementation. Once the RTL (Register Transfer Level) description is created, the further process leverages from the already existing logic synthesis tools.

The RTL description is then fed to the conventional logic synthesis flow to get the gate level implementation for the hardware design. The tools used by the behavioral synthesis process perform the task of transforming untimed and partially timed behavioral code into the fully timed RTL description.

Behavioral design flow scores over the traditional design flow by providing a layer of abstraction which allows the designer to focus more on the functional part of the circuit rather than the nitty-gritty of the interconnections and other low level stuff, thereby increasing the designer's productivity as well as the accuracy of the circuit design. The creation of the fully timed RTL implementation is taken care by the behavioral synthesis toolset.

A lot of research has gone into the behavioral design flow and the area of behavioral synthesis. A lot of research papers are out comparing the quality of code and the time it takes to develop a circuit by the traditional design flow and the behavioral design flow. It can be inferred from this literature that there is substantial reduction in the overall time of the hardware design development with increase in the quality level of the code using the behavioral synthesis way. An overall reduction of about 50% or more has been reported by the users.

Stages of the behavioral synthesis process

Number of activities takes place during the behavioral synthesis process. Different behavioral synthesis tools perform these activities in different orders and using different algorithms. Some behavioral synthesis tools combine some of these activities or perform them iteratively to converge on the desired solution. The activities are listed below following a general order of execution.

Lexical processing: Behavioral synthesis process takes in an algorithmic description of the design and this piece of code, written in high-level language undergoes lexical processing wherein it is parsed and converted to an internal representation. Lexical processing in case of behavioral synthesis is similar to that used in conventional high-level language compilation.

Algorithm optimization: Optimizations that can be performed on the algorithm itself include common subexpression elimination and constant folding. Many of these optimizations are commonly used in high-level language compilers or parallelizing compilers.

Control/Dataflow analysis: The result of this process is usually a Control/Dataflow Graph (CDFG) which determines which values are needed prior to computation of other values. No concept of time exists in the CDFG.

Library processing: The RTL implementation produced by behavioral synthesis will depend on the capabilities and characteristics of the library parts available for the specific implementation technology to be used. Library processing reads the available libraries and determines the functional, timing, and area characteristics of the available parts.

Resource allocation: Resource allocation establishes a set of functional units that will be adequate to implement the design. In many behavioral synthesis systems, an initial resource allocation is performed and subsequently modified during scheduling and/or binding.

Scheduling: Scheduling introduces parallelism and the concept of time. It transforms the algorithm into an FSM representation. Using the data dependencies of the algorithm and the latencies of the functional units in the library, the operations of the algorithm are assigned to specific clock cycles. There are often many

possible schedules. Directives that constrain the result with respect to latency, pipelining, and resource utilization will affect the schedule that is chosen.

Functional unit binding: Binding assigns the operations of the algorithm to specific instances of functional units from the library.

Register binding: In cases where values are produced in one clock cycle and consumed in another, these values must be stored in registers or memory. The register binding process allocates registers as needed and assigns each value to a physical register. Analysis of the lifetime of each data value can identify opportunities to use the same physical register to store different values at different times. This is done to reduce the size of the resulting design.

Output processing: The datapath and finite state machine resulting from all of the previous steps are written out as RTL source code in the target language. This code can be structured in a number of ways to optimize the downstream logic synthesis process or to enhance the readability of the code.

2.2 Logic Optimization

Logic Optimization is a step of the VLSI design cycle where the synthesizer performs modifications on the design of the circuit. These modifications manipulate the netlist to obtain an equivalent but better circuit according to the optimization goals. Most common optimization goals include:

- minimize the area
- reduce power consumption
- satisfy timing constraints
- reduce switching noise
- improve the testability of the final circuit

Most of the time the target logic is a single wire that violates some optimization constraints. One way to do automatic optimization is via redundancy addition and removal. The method is to target a wire that violates certain specification constraints and attempt to remove it by adding redundant logic and/or blocking all possible paths that exist from the target wire to the primary outputs.

ATPG (acronym for both Automatic Test Pattern Generation and Automatic Test Pattern Generator) is an electronic design automation method/technology used to find an input (or test) sequence that, when applied to a digital circuit, distinguishes between correct circuit behavior and faulty circuit behavior. DEDC(acronym for Design Error Diagnosis & Correction) is a way to fix problems found by ATPG. DEDC takes the erroneous design and the output of ATPG as its input and outputs a list of all applicable corrections. This list can be sorted by optimality of the fix, so we can pick the best fix to the problem. Although ATPG, in itself, does not do any optimizations, when combined with DEDC it becomes a very powerful optimization tool. In this optimization technique, the synthesizer first introduces an intentional design error by removing some part of the target logic. It then uses ATPG to derive the test vectors for this design error. In the third step, the synthesizer uses a DEDC algorithm to search for the correction. It chooses the most optimal correction from the output of DEDC and applies the correction. In the final step, it verifies the correctness of the new design. However, the downside to this methodology is that, the DEDC is not a runtime efficient algorithm.

2.3 Technology Independent Mapping

Technology Independent Mapping is the phase of logic synthesis when the optimized logic network is converted to a generic network using simple NAND gates. This generic network is in a technology-independent form

2.4 Technology Mapping

Technology Mapping is the phase of logic synthesis when gates are selected from a technology library to implement the circuit. We need technology mapping because straight implementation may not be good or efficient, whereas gates in the library are pre-designed and are usually optimized in terms of area, delay, power, etc. Generally faster gates are put along the critical path and area efficient gates off the critical path. Several technology mapping algorithms exist. However, all of the them have similar basic requirements:

- provide high quality solutions
- adapt to different libraries with minimal effort
- support different cost functions
- be runtime efficient

The two most used approaches for technology mapping are Rule-based and Graph-based. Rule-based algorithms are faster due to preset rules, whereas Graph-based algorithms output more efficient circuits.

2.5 Static Timing Analysis

Static Timing Analysis is a method of computing the expected timing of a digital circuit without requiring simulation. High-performance integrated circuits have traditionally been characterized by the clock frequency at which they operate. Gauging the ability of a circuit to operate at the specified speed requires the ability to measure, during the design process, its delay at numerous steps. Moreover, delay calculation must be incorporated into the inner loop of timing optimizers at various phases of design, such as logic synthesis, layout (placement and routing), and in in-place optimizations performed late in the design cycle. While such timing measurements can theoretically be performed using a rigorous circuit simulation, such an approach is liable to be too slow to be practical. Static timing analysis plays a vital role in facilitating the fast and reasonably accurate measurement of circuit timing. The speedup appears due to the use of simplified delay models, and on account of the fact that its ability to consider the effects of logical interactions between signals is limited.

3. PLACE & ROUTE

Place & Route generally comes after the generation of a netlist. As its name implies, it is composed of two steps, placement and routing. The first step, placement, involves deciding where to place all electronic components, circuitry, and logic elements in a generally limited amount of space. This is followed by routing, which decides the exact design of all the wires needed to connect the placed components. This step must implement all the desired connections while following the rules and limitations of the manufacturing process. Although placement occurs before routing, it is not absolute. It might be tweaked by rotating or moving components later on in the process.

4. HDL's AND SOFTWARE PROGRAMMING LANGUAGE

A HDL is analogous to a software programming language, but with major differences. Programming languages are inherently procedural (single-threaded), with limited syntactical and semantic support to

handle concurrency. HDLs, on the other hand, can model multiple parallel processes (such as flipflops, adders, etc.) that automatically execute independently of one another. Any change to the process's input automatically triggers an update in the simulator's process stack. Both programming languages and HDLs are processed by a compiler (usually called a synthesizer in the HDL case), but with different goals. For HDLs, 'compiler' refers to synthesis, a process of transforming the HDL code listing into a physically realizable gate netlist. The netlist output can take any of many forms: a "simulation" netlist with gate-delay information, a "handoff" netlist for post-synthesis place and route, or a generic industry-standard EDIF format (for subsequent conversion to a JEDEC-format file).On the other hand, a software compiler converts the source-code listing into a microprocessor-specific object-code, for execution on the target microprocessor.

5. EMERGING TRENDS IN HARDWARE DESIGN

As electronic systems grow increasingly complex, and reconfigurable systems become increasingly mainstream, there is growing desire in the industry for a single language that can perform some tasks of both hardware design and software programming.

5.1 System C

SystemC is an example of one such attempt. SystemC is a C++ class library and methodology that can effectively be used to create a cycle-accurate model of a system consisting of software, hardware and their interfaces. It has a rich set of data types for modeling the system. It allows multiple abstraction levels, from high level design down to cycle-accurate RTL level. SystemC is being promoted as a way to combine high level languages with concurrency models to allow faster design cycles for FPGAs than is possible using traditional HDLs.

- Modules are the basic building blocks to partition a design
- Modules allow to partition complex systems in smaller components
- Modules hide internal data representation, use interfaces
- Modules are classes in C++
- Modules are similar to “entity” in VHDL

5.2 JHDL

JHDL is an acronym for Just another Hardware Description Language. Behavioral synthesis is still to be added in this but it is already being touted as the HDL for runtime reconfigurable systems. It is a structurally based HDL implemented in JAVA. It focuses primarily on building circuits via object oriented approach treating collection of gates as Java Objects.

When circuitry for run –time reconfiguration is designed, the designer has to perform two tasks. The first task is to define the circuitry that implements the functionality of the application and this can be done using traditional CAD (Computer Aided Design) tools such as VHDL synthesis. The second is to develop a supervisory software program that controls the configurable computing paradigm during the running of the application. JHDL allows both these tasks to be done simultaneously using Java as the language

JHDL allows the user to design the structure and layout of a circuit, debug the circuit in simulation, netlist and interface for bit-stream synthesis, and so forth, all with a single Java description. Java is used to implement a simulation kernel that models hardware execution with a set of classes such as "Wire",

"Synchronous", "Combinational Logic", and so forth. The dynamic creation/destruction of these objects is exploited to model run-time hardware reconfiguration.

5.3 MyHDL

MyHDL is based on Python Language. It is an attempt to make Python available to the hardware designers as a hardware description and verification language. The greatest power lies in the ability to convert a MyHDL design to Verilog or VHDL. This provides a path into a traditional design flow.

Modeling

The clarity and power of Python makes MyHDL a powerful tool for high level modeling. Python has always been famous for the elegant solutions it offers for complex modeling problems and this can be leveraged in the hardware design description as well. Moreover, Python is outstanding for rapid application development and experimentation.

MyHDL uses Python generators for modeling hardware concurrency. Generators can be described as resumable functions. Semantically, the python generators are similar to always blocks of Verilog and processes in VHDL. A hardware module is modeled as a function that returns generators.

Simulation and Verification

The built-in simulator runs on top of the Python interpreter. It supports waveform viewing by tracing signal changes in a VCD file.

MyHDL brings the notion of unit testing in the area of hardware design. Python's unit test framework can be used on hardware designs.

MyHDL can also be used as hardware verification language for Verilog designs, by co-simulation with traditional HDL simulators.

Conversion to Verilog and VHDL

MyHDL makes it possible to convert the MyHDL design code to the conventional Verilog and VHDL and hence allowing leveraging from the pre existing logic synthesis toolset. Providing the path from the behavioral design flow to the traditional design flow has been the biggest power of these software programming language based HDLs. There are limitations on what can and what cannot be translated to conventional Verilog and VHDL , but the convertible subset is way bigger than the synthesis subset of the traditional design flow and includes features that can be used for high level modeling and test benches hence lending MyHDL its power.

The converter works on an instantiated design that has been fully elaborated. Consequently, the original design structure can be arbitrarily complex. Moreover, the conversion limitations apply only to code inside generators. Outside generators, Python's full power can be used without compromising convertibility.

Finally, the converter automates a number of tasks that are hard in Verilog or VHDL directly. A notable feature is the automated handling of signed arithmetic issues.

Other Software Programming Language based HDLs that are doing the rounds are RHDL (based on Ruby), Lava (based on functional programming language Haskell) ,etc.

6. CONCLUSION

Hardware design started with RTL (synthesizable structural description) coding in traditional HDL. Then as the need for improvements in design quality and designer's productivity was felt, people looked at the more structured design methodologies for an extensive reuse of existing components and subsystems. Then the next level of improvement was to add abstraction to the whole design process so that the designers can focus more on the functionality rather than the interconnections and other low level stuff. This gave rise to the notion of behavioral description and behavioral synthesis. Developing hardware design with this raised level of abstraction definitely increased the designer's productivity but nothing could beat the level of efficiency and accuracy of designing hardware structurally in traditional hardware languages for ASICs like circuits which were one time programmable circuits. With the advent of reconfigurable computing (made possible by FPGA's) , which offered the scope for changing the design if something wrong happens really made people ask for one language for both software and hardware design and there came System C and JHDL. And if the trend continues, traditional HDLs such as VHDL and Verilog may just find themselves on the verge of being obsolete.

7 REFERENCES

[1] Hardware Description Language. http://en.wikipedia.org/wiki/Hardware_description_language

[2] Volnei A. Pedroni, 'Circuit Design with VHDL', 2004,MIT PRESS

[3] Application-Specific Integrated Circuits ,1997,Addison Wesley Longman, Inc

[4] Behavioral Synthesis. http://www.eetimes.com/news/design/showArticle.jhtml?articleID=18900783

[5] Behavioral to RTL Design Flow in SystemC. http://www.design-reuse.com/articles/7354/from-behavioral-to-rtl-design-flow-in-systemc.html

[6] JHDL. http://www.jhdl.org/intro.html

[7] JHDL. http://en.wikipedia.org/wiki/JHDL

[8] P. Bellows, B. Hutchings, "JHDL - An HDL for Reconfigurable Systems", fccm,pp.175, IEEE Symposium on FPGAs for Custom Computing Machines, 1998

[9] Reconfigurable Computing http://en.wikipedia.org/wiki/Reconfigurable_computing

[10] MyHDL. http://www.myhdl.org/doku.php/overview

[11] I. Alston and B. Madahar, "From C to netlists: hardware engineering for software engineers?",

Electronics & Communication Engineering Journal, August 2002