The Utility of a Structured Hardware Language as a Pedagogical Tool

Mehran Massoumi

Department of Mathematics & Computer Science
California State University, Hayward
Hayward, CA 94542
massoumi@sbcglobal.net

Abstract: Senior/Graduate courses in Computer Organization and Design are most effective if they employ a Hardware Description Language (HDL) capable of modeling Register Transfer (RT) behavior. Such a language should allow description of design details and evaluation of architectural alternatives without sidetracking the students with esoteric constructs. Due to the momentum gained by Verilog and VHDL, both IEEE standards, one tends to rule out any non-standard and to-the-point languages, even though their benefits are readily observed. Moreover, what is really needed is a structured language that does not overwhelm the students with complex semantics and yet is powerful enough to allow functional capture and concise manipulation of the design. In this paper, a structured HDL is introduced. Experiences in using this language in a graduate course as well as the student's reactions will be discussed.

Introduction:

Throughout the decades of digital computer history, various symbols and notations have been developed for capturing the logical behavior of digital circuits while avoiding their electronic and fabrication details. The primary motivation has been to facilitate documentation, design, and analysis of complex systems. A few examples of such notations are schematics, Boolean expressions, timing charts, state transition tables, block diagrams, and hardware description languages. Although most of these notations are being used in practice today, hardware description languages have received remarkable acceptance from the design community since early1990s.

A hardware language can serve as a principal means of communication between members of a design team. The conciseness and readability of HDLs minimize the need for any natural language, and more error prone, discussion of the design. Furthermore, HDLs are commonly used for design entry into a variety of analysis and synthesis software tools, which greatly facilitate the verification and realization of any given design. It should be noted that a HDL is a language and only a language and has no apparent algebraic structure in terms of guiding the user to a minimal implementation. However, with practice, one will readily arrive at modeling techniques that yield more efficient hardware realizations.

Designers today routinely use VHDL[4,6] and Verilog[2,3,5], both IEEE standard HDLs, in their design flows. However, the role of these languages in teaching Computer Organization and Design is not well defined or convincingly effective. Both Verilog and VHDL are complex and

broad languages; they require a relatively long time to learn, and do not suggest any particular structure for already well-established design characteristics such as the control and data partitions. Due to their catchall premise, some aspects of these languages are foreign to even the mainstream designers. In this paper, it is suggested that a focused and structured HDL can be used more effectively for a design course than the prevailing standard languages. To that end, a new language, SHDL (Structured Hardware Description Language), will be introduced. Experiences in using this language in a graduate Computer Organization course at the CSU as well as the student's feedback will be discussed. It will be apparent that a carefully designed language will not only facilitate understanding of the design details but will pave the way for mastering the more complex standard languages.

SHDL, A Structured Language:

SHDL is a language designed for the purpose of concise functional description and manipulation of hardware algorithms at the RT level of abstraction. This language has been designed based on the following criteria:

- Should not take more than one week of lecture to teach, assuming prior knowledge of logic design, introductory Computer Architecture, and a programming language.
- Provide a rich set of constructs, which will facilitate description of any digital design at the Register-Transfer-Level[1].
- Where possible, use the same syntax and semantics as Verilog[3].
- Support a close and clear correspondence between the language constructs and the intended hardware[1].
- Guide the user in thinking in terms of hardware partitions[1] such as State Machines, Bus structures, and hardware resources such as arithmetic, logic, and relational functions.
- Use the cycle-based model of time.

Similar to Verilog, a basic unit in SHDL is a module, which is syntactically defined by the following rule, where keywords are shown in boldface letters:

module <*name*>**;**
    <*port declarations*>
    <*wire, tri, and register, declarations*>
    <*statements*>
endmodule

Even though large digital systems are partitioned into many interconnected modules, more often than not design problems in a classroom environment are limited to a single module. Hence, in the interest of simplicity, those language features pertaining to the management of design hierarchy can be safely omitted. The body of a SHDL description consist of one or more statements or sequences, where a sequence models the flow of control as well as the functions that must take place in each state. As will be evident in the pursuing example, a sequence will guide the user in maintaining the partition between the control and the data sections of a design. The control section in a sequence will cause register transfers to take place in the data section by

sending signals on a set of control lines. In some systems, the sequencing of control will be influenced by branching information fed back from the data section.

In the following example, a module is described to check if a character on the input lines *char_in* has already been received within the past 16 data receptions. A pulse on the input line *ready* marks the presence of a new character on *char_in* and a *busy* output is asserted while the checking is in progress. If a duplicate is detected, the output *duplicate* is asserted and will remain asserted until the next *ready* input is set.

```
Module duplicate_checker;
    input clock, reset, ready, char_in[7:0];
    output duplicate, busy;
    register duplicate, mem<0:15> [7:0], buffer[7:0], idx[3:0], next[3:0];
    wire  condition;
    sequence (clock, reset(1))
        1: begin
            ready ? duplicate <= 1'b0;
            ready ? buffer <= char_in;
            ready ? idx <= next – 4'b1;
            goto(1 if !ready);  goto (2 if ready);
        end
        2: begin
            busy = 1'b1;
            (buffer == mem<idx>) ? duplicate <= 1'b1;
            idx <= idx – 4'b1;
            condition = ((idx == next) || duplicate);
            goto(2 if !condition); goto(3 if condition);
        end
        3: begin
            busy = 1'b1;
            mem<next> <= buffer;
            next <= next + 4'b1;
            goto(1);
        end
    endsequence
endmodule
```

This design consists of a state machine with three states, marked by states 1, 2, and 3. The goto statements in each control step cause a conditional or unconditional branch to take place. Additionally, each step consists of a number of concurrent statements some of which are connections and some are register transfers. The connection statements are symbolized by '=' and the transfer statements by '<='. The dichotomy of control and data is maintained by the syntax of the sequence statement where states are represented by integers and the state assignment is abstracted out.

State 1 is the initial state where the machine waits for the *ready* pulse, after which the input character is clocked into the *buffer* register, *duplicate* bit is reset to zero, and the *idx* register is initialized. The transfer statements in state 1 are preceded by the "ready ?", which serves as the clock enable condition on the corresponding register. In other words, the following two transfer statements are functionally equivalent, but the first implies a register with a clock enable expression and the second is a register with a mux input and no clock enable.

Ready ? duplicate <= 1'b0;
duplicate <= ready ? 1'b0 : duplicate;

State 2 is where the actual search for a duplicate character takes place where every row of the *mem* array is checked for a match. The branch out of state 2 and into state 3 will occur if only if a match is detected or if all rows of the *mem* array have been traversed, whichever occurs first. State 3 is where the current buffer is stored in the *mem* array and the control is transferred back to state 1. The statement "mem<next> <= buffer;" implies a decoding logic on the register *next*, which is used as a variable index into the array.

It should be noted that the expressions in SHDL follow the same syntax and semantics as Verilog. Moreover, other similarities with Verilog are apparent in the declarations and the port definitions.

Generating logic from a SHDL description can be readily achieved automatically or manually. The following simplified procedure outlines the steps for logic realization of the data registers in a SHDL description. Other components such as buses follow similar steps.

Foreach register bit 'r' **do**
    identify the set of states S in which 'r' is driven;
    foreach state 's' in S **do**
        use the driver expression 'd' of 'r' to form "d && s" to be or-ed with the data input of 'r';
        if 'r' has a clock enable condition 'e' **then**
            Form the expression "e && s" to be or-ed with clock enable input of 'r';
        else
            logical-or 's' with clock enable input of 'r';

As an example, the register *duplicate* is driven in both states 1 and 2. In state 1, the clock enable condition is the signal *ready* and in state 2 the enable condition is the expression "(buffer == mem<idx>)". Both enable conditions are and-ed with their corresponding state signal and the resulting expressions are or-ed to form the expression for the clock enable input of the *duplicate* register. The data input of *duplicate* is simply the control signal for state 2.

Using SHDL in a Computer Organization and Design Course:

SHDL has been used successfully in a 10-week graduate course in Computer Organization and Design at the CSU, Hayward. The second week of the quarter was devoted to presenting the SHDL details during which several sample designs were described, covering all aspects of the

language. Moreover, students were familiarized with taking a word specification of a design and composing a SHDL description for that design. Also discussed, was how to arrive at a circuit realization from a given SHDL description. Based on performances in home works and exams, students became quickly proficient in reading and understanding a given SHDL description. They easily managed to trace the register values in a given description as well as arrive at a circuit realization. However, proficiency in taking a word specification and its corresponding hardware algorithm and arrive at an acceptable SHDL description did not occur until later in the quarter.

Starting from the third week, SHDL was used extensively to describe various CPU architectures, memory controllers, bus arbiters, and arithmetic algorithms. It served as an effective tool to illustrate the timing details for various operations and how they can be changed for better performance. Although students were improving their SHDL skills throughout the course, the language was never a hindrance in the flow of the course and did not overwhelm the main focus. Overall, the students had a very positive experience with using the language, as reflected in an anonymous survey that was taken on the 9th week. Basically, the students were asked (1) if SHDL was readily learned, and (2) if SHDL helped them in learning the material. A total of 36 students, who were the total enrollment for two sessions taught in two quarters, participated in the survey. None of the students in the survey had any previous knowledge of Verilog or VHDL, although a few had limited exposure to an HDL in their previous courses. The results are illustrated below:

	Yes	Somewhat	No
Was SHDL readily learned?	27	6	3
Was SHDL helpful?	29	5	2

Students often commented unfavorably on the fact that SHDL is not a standard language. A clear response is that the time invested in learning SHDL is very minimal compared to that of effectively using the available standard languages and the benefits of using SHDL is already established. Moreover, learning SHDL serves as a stepping-stone for mastering other more complex languages. It can be safely claimed that the time invested in learning SHDL will shorten the time required to become proficient in Verilog or VHDL. Also, general concepts such as RTL design and concurrency are already covered in a structured paradigm once students become familiar with SHDL.

Conclusion:

In this paper a simple and yet expressive hardware language, SHDL, was introduced for the purpose of teaching Computer Organization and Design. Positive experiences in using this language coupled with feedback from students suggest that SHDL has the promise of being an effective tool in modeling and studying design alternatives. Moreover, it is argued that the usefulness of standard languages, such as Verilog and VHDL, as a pedagogical tool is not convincingly established. This is evident in the fact that prevalent textbooks in Computer Organization today hardly rely on either language for conveying CPU and interface design concepts. Both Verilog and VHDL are complex languages and require several weeks to learn. Furthermore, knowing these languages a priori is helpful but does not necessarily make them a

preferred candidate because of their unstructured, verbose, and catchall nature. However, it should be noted that learning Verilog and VHDL in general is essential for students aspiring to be designers. To that end, having learned SHDL will solidify a better understanding of those languages.

Bibliography:

[1] F. J. Hill, G. R. Peterson, Digital Systems: Hardware Organization and Design, 3rd edition, John Wiley, New York, 1987.

[2] S. Palnitkar, Verilog HDL: A Guide to Digital Design and Synthesis, Prentice Hall, 1996.

[3] IEEE Standard Verilog Language Reference Manual, IEEE Standard Number 1364-2001, 2001.

[4] IEEE Standard VHDL Language Reference Manual, IEEE Standard Number 1076-2002, 2002.

[5] Z. Navabi, Verilog Digital System Design, McGraw Hill, 1999.

[6] Z. Navabi, VHDL: Analysis and Modeling of Digital Systems, 2nd edition, McGraw Hill, 1998.

Biography:

MEHRAN MASSOUMI received his Ph.D. in Electrical and Computer Engineering from the University of Arizona in 1994. He has worked with a number of Design Automation companies, where he was responsible for HDL synthesis products. Currently, he is with Averant Inc, a verification company he co-founded in 1998, and has also been lecturing at the CSU, Hayward.