Abstract Verilog: A Hardware Description Language for Novice Students

Carl Ebeling and Brian French
University of Washington
Department of Computer Science and Engineering
Seattle, WA
ebeling@cs.washington.edu, bfrench@andrew.cmu.edu

Abstract

All modern hardware design makes use of hardware description languages like Verilog and VHDL. Thus these languages are presented very early in the curriculum, often in the first design course. Unfortunately, these languages often cause substantial confusion for students learning hardware design, particularly students that have experience writing programs in sequential languages like C and Java.

To address this problem, we have defined a language called Abstract Verilog which is similar to Verilog, but which has well-defined, clean parallel execution semantics. Abstract Verilog has a somewhat restricted syntax in order to reduce the cognitive load for new students. However, almost any program that can be written in Verilog can be written in Abstract Verilog, where it is shorter and easier to understand. We have used Abstract Verilog successfully in both introductory and advanced design classes.

1. Introduction

Virtually every computer science, computer engineering and electrical engineering curriculum includes a required course on digital design where students learn the basics of combinational and sequential logic. Hardware description languages like Verilog and VHDL are typically covered in these courses because HDLs and the design tools that compile them into hardware are central to all modern digital design[1].

Unfortunately, students find these languages confusing, particularly students who have programmed in languages like C and Java. HDL’s are inherently parallel programming language and require a new way of thinking about programming. In fact, we find it best to postpone introducing HDL’s until students have at least several weeks of hardware design experience using schematics and block diagrams, and understand the inherent parallel execution semantics of hardware.

There are many sources of confusion and error in Verilog[2]1 for novice students:

- The differences between the Verilog wire and reg declarations have nothing to do with the difference between wires and registers.
- Registers are created implicitly when signals are assigned in always blocks triggered by (posedge clk). Incomplete trigger lists and incomplete combinational assignments lead to inadvertent latch creation.
- Blocking and non-blocking assignments are needed because of the implicit register semantics of Verilog, but these are hard for students to understand and are confusing to interpret.
- Implementing Mealy finite state machines requires the use of two always blocks. More generally, the logic for a particular component may be spread out in several places making it hard to understand and easy to make mistakes.

Most experienced Verilog designers follow stringent guidelines that steer them clear of many of these pitfalls. By contrast, our approach was to avoid these problems by defining a variant of Verilog with clear and straightforward execution semantics. This allows students to focus on design and not the idiosyncrasies of Verilog. We call this variant “Abstract Verilog” because it provides abstractions that hide the idiosyncrasies of Verilog. We have written a translator that translates Abstract Verilog programs into standard Verilog-2001 programs that can be simulated and synthesized using standard Verilog simulation and synthesis tools.

2. Definition of Abstract Verilog

One of the fundamental problems with Verilog is that it is used both for test fixtures, which requires system-level
features, and for synthesizeable hardware components. By contrast, Abstract Verilog is intended for describing synthesizeable hardware components, which results in a deceptively simple and transparent language. The single-use definition of Abstract Verilog also makes it easier for students to come up to speed.

There are two types of signals in Abstract Verilog, combinational and registered, declared using the COMB and REGISTER declarations. That is, registers are not created implicitly by posedge clk events, but are declared explicitly. Since signals are declared explicitly, no latches can be created implicitly. Combinational signals are assigned values using the = operator, while registered signals are assigned using the <-- operator. Although operators are redundant since all signals are declared, it is well-known that such redundancy reduces the number of programming errors.

Signals are assigned values in an ALWAYS block. This ALWAYS block is executed “continuously” just like an always @(*) block in Verilog. This idea of continuous execution is the key concept to understanding the semantics of an Abstract Verilog program: the program acts as if the ALWAYS block is always being executed, which is exactly how hardware behaves. The statements in the ALWAYS block are repeatedly executed in sequential order in zero time - delays have no meaning in Abstract Verilog. If there is more than one ALWAYS block, then they are executed in parallel, again just the way hardware does.

Combinational assignments take place immediately, while register assignments take effect only at the next rising edge of the clock. That is, registers retain their value during the continuous execution of the ALWAYS block, only changing when the clock “ticks”, when all register assignments are executed simultaneously using the standard Verilog non-blocking assignment. Thus there can be no errors of execution ordering, and thus any discrepancy between simulation and synthesized hardware.

Abstract Verilog also includes declarations for both asynchronous and synchronous dual-ported memories. Writes to both types of memories have the same semantics as assignments to register signals: a memory appears to be a vector of registers. Reads are done using combinational assignment for asynchronous memories and registered assignment for synchronous memories. These memories supported by Abstract Verilog correspond closely to the distributed and block memories provided by modern FPGAs.

Abstract Verilog otherwise uses the same syntax for signal declarations, data values, and expressions, and supports Verilog statements like if, case, for and while as well as functions and tasks.

The program below implements a simple enabled counter with a limit value. The register is declared with an initial value. This is the value assigned to the register when reset is asserted. Note that tc can be assigned before vcount is assigned because vcount is a registered value.

```verilog
module countv (clk, reset, cen, vcount, tc);
parameter limit = 666;
input clk;
input reset;
input cen;
output [9:0] vcount;
output tc; // Terminal count
REGISTER [9:0] vcount = 0;
COMB tc;

ALWAYS begin
  tc = (vcount == limit);
  if (cen) begin
    if (vcount < limit) begin
      vcount <-- vcount + 1;
    end else begin
      vcount <-- 0;
    end
  end
end
endmodule
```

3. Conclusion

We have used Abstract Verilog in several undergraduate and graduate hardware design classes. It takes much less time to get students up to speed writing and simulating programs, and there are far fewer errors caused by confusion with the semantics of the language.

There are, however, a few disadvantages. Students feel that they are not learning “real Verilog” and that they will be at a disadvantage when interviewing for a job. However, the style of Abstract Verilog programs is very close to that used by good hardware designers. We have found that students who understand how to write Abstract Verilog programs quickly pick up on the difference with standard Verilog, and in fact, use a much better style than typical Verilog programmers.

Acknowledgment This material is based upon work supported by the National Science Foundation under Grant No. 0220095.

References
