Using Verilog HDL to Teach Computer Architecture Concepts

Dr. Daniel C. Hyde
Computer Science Department
Bucknell University
Lewisburg, PA 17837, USA
hyde@bucknell.edu

Paper presented at

Workshop on Computer Architecture Education

June 27th, 1998
Barcelona, Spain

Held in conjunction with the

25th International Symposium on Computer Architecture

June 27 - July 1, 1998
Using Verilog HDL to Teach Computer Architecture Concepts

Dr. Daniel C. Hyde
Computer Science Department
Bucknell University
Lewisburg, PA 17837, USA
hyde@bucknell.edu

1. Introduction

Students in computer architecture courses, especially undergraduates, need to *design* computer components in order to gain an in-depth understanding of architectural concepts. For maximum benefit, students must be active learners, engage the material and design, i. e., produce components to meet a specific need. Unfortunately, computers have become so sophisticated that designing architectural components, e. g., a cache memory, in hardware is not feasible in a one semester course. This paper describes an approach where students use a hardware description language (HDL), Verilog HDL and an associated simulator, to design components of computer systems and explore architectural concepts. To support this approach, the author has developed web-based course materials which include a manual on Verilog HDL, a paper on how to realize his Verilog-based computational model in digital circuits and twelve structured laboratory exercises.

Other engineering educators have used hardware description languages in their courses before, but at a lower level, e. g., digital circuit design or VLSI design. What is distinctive about the author's approach is the use of an industrial standard HDL in a computer architecture course. Students, especially the ones who consider themselves software-types, are able to design and test hardware, for example, a CPU with instruction look ahead or a floating point adder. Further, they gain valuable insight into the power of computer-aided-design tools used by hardware designers in industry.

In the course, we stress that our goal is to formulate a computational model in the Verilog notation so we no longer need to think in digital circuits. That is, we develop a higher level of abstraction to think about digital systems that is much more concise than digital circuits, e. g., sequential machines. A few lines of Verilog code may translate into hundreds of flip flops, **AND**, **OR** and **NOT** gates. This Verilog model is precise and concise -- the Verilog notation supplies the information for *both* the data unit and the control unit associated with the control sequence. This approach is basically the one used in industry to design digital integrated circuits, such as microprocessor chips. The students are told that with automated tools, the Verilog code could be translated to the integrated circuit masks, say, for CMOS.

2. Institutional Context

The author has course tested this approach at Bucknell University for three semesters. The approach is used in an undergraduate computer architecture course taken primarily by computer science seniors. The prerequisite for the course is a traditional computer organization course which the students take in their sophomore year. The architecture course is organized as three one-hour lectures and a two-hour structured laboratory session per week. During the laboratory, the students access the web-based materials and the Verilog HDL simulator on Sparc workstations. The course uses the popular text *Computer Architecture: A Quantitative Approach*, second edition, by John L. Hennessy and David A. Patterson. Whereas the text focuses on *analyzing* a design, the author's approach complements the text by having the student *learn* and *do* designs as well. This design aspect is especially important to educational programs in the United States seeking accreditation by the Accreditation Board of Engineering and Technology (ABET). (Most engineering programs in the United States are accredited by ABET which serves as a stamp of approval on the quality of the program.)

3. Hardware Description Languages

Digital systems are highly complex. At their most detailed level, they may consist of millions of elements, e. g., transistors or logic gates. For many decades, logic schematics served as the *lingua franca* of logic design, but not any more. Today, hardware complexity has grown to such a degree that a schematic with logic gates is almost useless, as it shows only a web of connectivity and not the functionality of design. Since the 1970s, computer engineers and electrical engineers have moved toward hardware description languages (HDLs). The most prominent HDLs in industry are Verilog and VHDL. Verilog is the top HDL used by over 10,000 designers at such hardware vendors as Sun Microsystems, Apple Computer and Motorola. Industrial designers prefer Verilog. The syntax of Verilog is based on the C language, while the syntax of VHDL is based on Ada. Since the author's students know C or C++, Verilog was the obvious choice. A free Verilog simulator is available from SynaptiCAD, Inc. For Windows 95/NT, Windows 3.1, Macintosh, SunOS and Linux platforms, they offer FREE versions of their VeriWell product, which is available from http://www.syncad.com/ver_down.htm. The free versions are the same as the industrial versions except they are restricted to a maximum of 1000 lines of HDL code.

4. Verilog HDL

The Verilog language provides the digital designer with a means of describing a digital system at a wide range of levels of abstraction, and, at the same time, provides access to computer-aided design tools to aid in the design process at these levels.

Verilog allows hardware designers to express their design with behavioral constructs, deferring the details of implementation to a later stage of design in the design. An abstract representation helps the designer explore architectural alternatives through simulations and detect design bottlenecks before detailed design begins.

Though the behavioral level of Verilog is a high level description of a digital system, it is still a precise notation. Computer aided design tools exist which will “compile” the Verilog notation to the level of circuits consisting of logic gates and flip flops. Verilog also allows the designer to specify designs at the logical gate level using gate constructs and at the transistor level using switch constructs. A primary use of HDLs in industry is the simulation of designs before the designer must commit to fabrication.

5. Use of Verilog in Course

Our goal in the computer architecture course is not to create VLSI chips but to use Verilog to precisely describe the *functionality* of *any* digital system, for example, a computer.

In the course, a small subset of the Verilog language is used to describe and develop computer architectural concepts using a register transfer level model of computation. The course also uses the structural and gate levels of Verilog to design such things as registers from D-flip flops and adders from gates. To illustrate, we describe a simple computer in the Verilog-based model.

Assume we have a very simple computer, with 1024 words of memory (**MEM**) each 32-bits wide, a memory address register (**MA**), a memory data register (**MD**), an accumulator (**AC**), an instruction register (**IR**) and a program counter (**PC**). Since we have 1024 words of memory, the **PC** and **MA** need to be 10-bits wide.

We can declare the registers and memory in Verilog as the following:

```
reg [0:31] AC, MD, IR;
reg [0:9]  MA, PC;
reg [0:31] MEM [0:1023];
```

We assume a digital system can be viewed as moving vectors of bits between registers. This level of abstraction is called the *register transfer level*. For example, we may describe in Verilog an *instruction fetch* by four register transfers:

```
// instruction fetch
#1 MA <= PC;
#1 MD <= MEM[MA]; // memory read
#1 IR <= MD;
#1 PC <= PC + 1;
```

The meaning of the first line is to transfer the 10 bits of the **PC** into the **MA** register *after* waiting one clock period (the **#1**). Note that it is important that we use Verilog's *blocking* assignment operator (<=) rather than the *non-blocking* assignment (=). In our computational model, we assume that trailing edge triggered D-flip flops are used for the registers. Therefore, our Verilog notation models the situation because the blocking assignment operator (<=) means to block the assignment until the *end* of the current unit in simulation time.

Assuming the operation code for a LOAD instruction is 0000 in binary and is in the first four bits of **IR**, we could design the *decode* and *execute* part of a LOAD instruction as follows:

```
// decode and execute code for a LOAD
#1 if (IR[0:3] == 4'b0000) begin
      #1 MA <= IR[22:31]; // last 10 bits are address
      #1 MD <= MEM[MA];   // memory read
      #1 AC <= MD;
end
```

The students use Verilog to describe their digital systems but also to test their designs by a *simulator* running on UNIX workstations. See the Appendix for the complete Verilog program of this simple computer. The above design is very slow! A LOAD instruction would take eight clock periods. Later in the course, the students learn to carefully analyze the Verilog code as well as introduce concurrency to improve the speed. Thereby, the students learn the importance of fine tuning the hardware for maximum performance.

6. Computational Model of Course

Register transfers are general. In fact, we can view a computation as a specific class of register transfer.

Definition: A **computation** consists of placing some *Boolean* function of the contents of argument registers into a destination register.

That is, in the course, a computation is a register transfer. In computer design, register transfers are very important. They are everywhere -- in arithmetic logic units (ALU), control units (CU), memory subsystems, I/O devices and interconnection networks.

Students learn that we need only Boolean functions. We don't need arithmetic or higher order functions. To realize the Boolean functions, we design combinational logic circuits, for example, an adder, from **AND**, **OR** and **NOT** gates.

However, Boolean functions have no concept of time. To incorporate the passage of time in our model, we define computing as a sequence of several register transfers where each transfer takes one or more clock periods.

Definition: Computing is a sequence of computations.

Therefore, the above Verilog code for the LOAD instruction has seven computations or register transfers. We would say that performing a LOAD instruction is computing because we do seven computations one after the other, in sequential order.

A major part of the description of a computer is a plan defining each register transfer, or computation, and specifying the order and timing in which these register transfers will take place. In the course, students learn that the Verilog-based model describes both the *data unit* which contains the digital circuits for each register transfer and the *control unit* which sequences these register transfers at the proper times.

One of the clever parts of the Verilog language design was making register transfers look like assignment statements in other programming languages. Since many designers are comfortable with a language like C or Pascal, Verilog has had a large degree of success.

7. Realization of Verilog Code in Digital Logic

In the course we demonstrate that our subset of Verilog code can be realized in digital logic circuits. Verilog is a structured language like C++ with **sequence**, **if-then-else**, **case**, **while, repeat** and **for** constructs. In the course, we show how each control flow construct can be easily "compiled" to a digital circuit as part of the control unit (a finite state machine). Also, we show that this translation from Verilog code to digital circuit can be automated.

8. Why Verilog is Better than C or C++ for Modeling Hardware

Verilog has features used to model digital circuits that are not available in traditional procedural languages like C or C++. One feature is the **continuous assignment** statement which is active for the lifetime of the program. Whenever the arguments of the expression on the right-hand side change, the outputs change, possibly after a specified time delay. This statement is used to model combinational circuits such as an adder.

Another feature not found in C or C++ is the modeling of concurrency. For example, several register transfers can be performed in the same clock period, or concurrently. Given the following register transfers without any data dependencies:

```
#1 A <= B;
#1 C <= D;
```

we can remove the second **#1** and have the transfers done in the same clock period.

```
#1 A <= B; C <= D;
```

The semantics of the Verilog *blocking assignment* says to evaluate all the right-hand sides and block the assignments to left-hand sides until the *end* of the current unit of simulation time. This models our assumption that the registers are composed of trailing edge D-flip flops where the information is clocked into the flip flops at the end of the clock period.

Verilog has other language features for handling concurrency. For example, a digital system with its own control unit is modeled by the **initial** (and **always**) construct. Several **initial** constructs are executed concurrently. Within an **initial** construct, a structured **fork** and **join** allows multiple threads of control within a control unit.

Also, another feature not found in C or C++, Verilog allows the execution of a procedural statement when triggered by a value change on a wire or a register or the occurrence of a named event. For example, this is useful to model interrupts as follows:

```
@(posedge I) Intr = line&mask;  // controlled by positive edge of I
```

9. Laboratory Exercises

The students finish a series of twelve laboratory exercises that build on a simple four instruction computer by adding addressing modes, integer multiply, instruction lookahead, cache and floating point add. Along the way, the students explore Verilog's structural modeling to construct a carry ripple adder from **AND**, **OR** and **EXCLUSIVE OR** gates, and explore concurrency with **fork** and **join** constructs and multiple digital systems and signaling.

The students find this approach using the Verilog notation easy to relate to the Hennessy and Patterson text and their previous course work. The author finds using a major industrial HDL is highly motivating to the students. The hardware-types see learning Verilog as an important mark on their resumes for job opportunities. The software-types are also motivated, as they see Verilog as another programming language to learn.

10. Conclusions

The author has developed an approach which allows students to design hardware components using Verilog HDL, a hardware description language. The author has developed a 32-page Verilog Manual and a 12-page paper on his Verilog-based computational model and how to realize the model in digital circuits. Along with the free Verilog simulator supplied by SynaptiCAD Inc., these web-based materials supply to computer architecture instructors the ability to teach design of architectural concepts as well the flexibility and freedom to modify the approach to integrate with their current instructional needs. Over a dozen universities have requested permission to copy the materials and hand them out to their students. All the materials are available on the author's web site.

11. URLs for Web Pages

1. URL for the computer architecture course including the twelve laboratory exercises:
   http://www.eg.bucknell.edu/~cs320/fall-1997/index.html

2. URL for use of Verilog in course:
   http://www.eg.bucknell.edu/~cs320/fall-1997/verilog.html

3. URL for Verilog Manual:
   http://www.eg.bucknell.edu/~cs320/1995-fall/verilog-manual.html

12. Appendix - Complete Verilog Program of a Simple Computer

```
// Ultra Simple Computer in Verilog HDL, by Dan Hyde; June 1, 1998

module ultra;
  // simple computer with 4-bit op codes in first four bits
  // and 10 bit address in last ten bits of 32-bit instructions
  // 0000 load  M   Load contents at address M into AC
  // 0001 store M   Store contents of AC into address M
  // 0010 add   M   Add contents at address M to AC
  // 0011 jump  M   Jump to instruction at address M

parameter clock = 1;

// declare registers and flip flops
reg [0:31] AC, IR, MD;
reg [0:9] PC, MA;
reg [0:31] MEM[0:1023];  // 1024 words of 32-bit memory
```

```
// The two "initial" and the "always" constructs run concurrently

// Will stop the execution after 100 simulation units.
initial begin: stop_at
   #(100*clock) $stop;
end

// Initialize the PC register and memory MEM with test program
initial begin: init
    PC = 10;                // start of machine language program
    MEM[3] =  32'b00000000000000000000000000000010;   // Data 2
    MEM[4] =  32'b00000000000000000000000000000001;   // Data 1
    MEM[10] = 32'b00000000000000000000000000000011;   // Load 3
    MEM[11] = 32'b00100000000000000000000000000100;   // Add 4
    MEM[12] = 32'b00010000000000000000000000000101;   // Store 5
    MEM[13] = 32'b00110000000000000000000000001011;   // Jump 11

    $display("Time  PC     IR      MA     MD        AC      MEM[5]");
// monitor following registers and memory location and print when any change
$monitor("  %0d  %h  %h  %h  %h  %h  %h", $time, PC, IR, MA, MD, AC, MEM[5]);

end

// main_process will loop until simulation is over
always begin: main_process

      // Instruction Fetch
      #clock MA <= PC;
      #clock MD <= MEM[MA];              // memory read
      #clock IR <= MD; MA <= MD[22:31];  // last ten bits are address
      #clock PC <= PC + 1;

      //decode and execute instruction
      if(IR[0:3] == 4'b0000) begin           // load
          #clock MD <= MEM[MA];
          #clock AC <= MD;
      end
      if(IR[0:3] == 4'b0001) begin           // store
          #clock MD <= AC;
          #clock MEM[MA] <= MD;
      end
      if(IR[0:3] == 4'b0010) begin           // add
          #clock MD <= MEM[MA];
          #clock AC <= AC + MD;
      end
      if(IR[0:3] == 4'b0011) begin           // jump
          #clock PC <= MA;
      end
end

endmodule
```

```
altair{128}% veriwell ultra.v
VeriWell for SPARC HDL     <Version 2.0.1>    Tue Jul  7 15:23:54 1998

  This is a free version of the VeriWell for SPARC Simulator
  Distribute this freely; call 1-800-VERIWELL for ordering information
  See the file "!readme.1st" for more information

  Copyright (c) 1994 Wellspring Solutions, Inc.
         All rights reserved

Entering Phase I...
Compiling source file : ultra.v
The size of this model is [3%, 3%] of the capacity of the free version

Entering Phase II...
Entering Phase III...
No errors in compilation
Top-level modules:
   ultra

Time  PC     IR      MA     MD       AC      MEM[5]
  0  00a  xxxxxxxx  xxx  xxxxxxxx  xxxxxxxx  xxxxxxxx
  1  00a  xxxxxxxx  00a  xxxxxxxx  xxxxxxxx  xxxxxxxx
  2  00a  xxxxxxxx  00a  00000003  xxxxxxxx  xxxxxxxx
  3  00a  00000003  003  00000003  xxxxxxxx  xxxxxxxx
  4  00b  00000003  003  00000003  xxxxxxxx  xxxxxxxx
  5  00b  00000003  003  00000002  xxxxxxxx  xxxxxxxx
  6  00b  00000003  003  00000002  00000002  xxxxxxxx
  7  00b  00000003  00b  00000002  00000002  xxxxxxxx
  8  00b  00000003  00b  20000004  00000002  xxxxxxxx
  9  00b  20000004  004  20000004  00000002  xxxxxxxx
 10  00c  20000004  004  20000004  00000002  xxxxxxxx
 11  00c  20000004  004  00000001  00000002  xxxxxxxx
 12  00c  20000004  004  00000001  00000003  xxxxxxxx
 13  00c  20000004  00c  00000001  00000003  xxxxxxxx
 14  00c  20000004  00c  10000005  00000003  xxxxxxxx
 15  00c  10000005  005  10000005  00000003  xxxxxxxx
 16  00d  10000005  005  10000005  00000003  xxxxxxxx
 17  00d  10000005  005  00000003  00000003  xxxxxxxx
 18  00d  10000005  005  00000003  00000003  00000003
 19  00d  10000005  00d  00000003  00000003  00000003
 20  00d  10000005  00d  3000000b  00000003  00000003
 21  00d  3000000b  00b  3000000b  00000003  00000003
 22  00e  3000000b  00b  3000000b  00000003  00000003
 23  00b  3000000b  00b  3000000b  00000003  00000003
 25  00b  3000000b  00b  20000004  00000003  00000003
 26  00b  20000004  004  20000004  00000003  00000003
 27  00c  20000004  004  20000004  00000003  00000003
 28  00c  20000004  004  00000001  00000003  00000003
 29  00c  20000004  004  00000001  00000004  00000003
 30  00c  20000004  00c  00000001  00000004  00000003
 31  00c  20000004  00c  10000005  00000004  00000003
 32  00c  10000005  005  10000005  00000004  00000003
 33  00d  10000005  005  10000005  00000004  00000003
 34  00d  10000005  005  00000004  00000004  00000003
 35  00d  10000005  005  00000004  00000004  00000004
```

36 00d 10000005 00d 00000004 00000004 00000004
37 00d 10000005 00d 3000000b 00000004 00000004
38 00d 3000000b 00b 3000000b 00000004 00000004
39 00e 3000000b 00b 3000000b 00000004 00000004
40 00b 3000000b 00b 3000000b 00000004 00000004
42 00b 3000000b 00b 20000004 00000004 00000004
43 00b 20000004 004 20000004 00000004 00000004
44 00c 20000004 004 20000004 00000004 00000004
45 00c 20000004 004 00000001 00000004 00000004
46 00c 20000004 004 00000001 00000005 00000004
47 00c 20000004 00c 00000001 00000005 00000004
48 00c 20000004 00c 10000005 00000005 00000004
49 00c 10000005 005 10000005 00000005 00000004
50 00d 10000005 005 10000005 00000005 00000004
51 00d 10000005 005 00000005 00000005 00000004
52 00d 10000005 005 00000005 00000005 00000005
53 00d 10000005 00d 00000005 00000005 00000005
54 00d 10000005 00d 3000000b 00000005 00000005
55 00d 3000000b 00b 3000000b 00000005 00000005
56 00e 3000000b 00b 3000000b 00000005 00000005
57 00b 3000000b 00b 3000000b 00000005 00000005
59 00b 3000000b 00b 20000004 00000005 00000005
60 00b 20000004 004 20000004 00000005 00000005
61 00c 20000004 004 20000004 00000005 00000005
62 00c 20000004 004 00000001 00000005 00000005
63 00c 20000004 004 00000001 00000006 00000005
64 00c 20000004 00c 00000001 00000006 00000005
65 00c 20000004 00c 10000005 00000006 00000005
66 00c 10000005 005 10000005 00000006 00000005
67 00d 10000005 005 10000005 00000006 00000005
68 00d 10000005 005 00000006 00000006 00000005
69 00d 10000005 005 00000006 00000006 00000006
70 00d 10000005 00d 00000006 00000006 00000006
71 00d 10000005 00d 3000000b 00000006 00000006
72 00d 3000000b 00b 3000000b 00000006 00000006
73 00e 3000000b 00b 3000000b 00000006 00000006
74 00b 3000000b 00b 3000000b 00000006 00000006
76 00b 3000000b 00b 20000004 00000006 00000006
77 00b 20000004 004 20000004 00000006 00000006
78 00c 20000004 004 20000004 00000006 00000006
79 00c 20000004 004 00000001 00000006 00000006
80 00c 20000004 004 00000001 00000007 00000006
81 00c 20000004 00c 00000001 00000007 00000006
82 00c 20000004 00c 10000005 00000007 00000006
83 00c 10000005 005 10000005 00000007 00000006
84 00d 10000005 005 10000005 00000007 00000006
85 00d 10000005 005 00000007 00000007 00000006
86 00d 10000005 005 00000007 00000007 00000007
87 00d 10000005 00d 00000007 00000007 00000007
88 00d 10000005 00d 3000000b 00000007 00000007
89 00d 3000000b 00b 3000000b 00000007 00000007
90 00e 3000000b 00b 3000000b 00000007 00000007
91 00b 3000000b 00b 3000000b 00000007 00000007
93 00b 3000000b 00b 20000004 00000007 00000007
94 00b 20000004 004 20000004 00000007 00000007
95 00c 20000004 004 20000004 00000007 00000007

```
 96  00c  20000004  004  00000001  00000007  00000007
 97  00c  20000004  004  00000001  00000008  00000007
 98  00c  20000004  00c  00000001  00000008  00000007
 99  00c  20000004  00c  10000005  00000008  00000007
Stop at simulation time 100
C1>
charcoal{47}%
```