Use of HDLs in Teaching of Computer Hardware Courses

Zvonko Vranesic and Stephen Brown
University of Toronto
{zvonko@eecg.toronto.edu}

Abstract

A modern treatment of an introductory course on the design of logic circuits should include an early introduction of a hardware description language (HDL). This can done without sacrificing the emphasis on fundamental concepts of logic circuit design. An example of how this may be achieved is given.

1 Introduction

This presentation focuses on the use of hardware description languages and design automation tools in the teaching of courses on logic circuits and computer architecture. It is based on the experience gained at the University of Toronto, which involved courses in Computer Engineering, Electrical Engineering, and Computer Science programs.

While there has been considerable debate about the optimal way of structuring the courses that teach the concepts of computer hardware, a traditional sequence based on an introductory course in the design of logic circuits and a subsequent course in computer organization (architecture) is still a very attractive option. It is even better if these courses are followed by a more advanced course in computer architecture. This is the structure at the University of Toronto. Each course is accompanied by a laboratory in which students develop a real understanding of the key concepts and the various ways in which they may be implemented in practice. For the purposes of this discussion, we will assume just the basic two-course sequence.

2 Logic Circuits

A course in logic circuits can be taught effectively as soon as the students have acquired an understanding of some high-level programming language and have learned the fundamentals of good programming practices. The course should emphasize the important concepts which include the notions of implementability, cost, optimization, timing, stability, and performance. The amount of material that can be covered depends on the length of the course, the ability of the students, and the quality of the supporting facilities comprising laboratories and CAD tools.

While everybody agrees that the students must learn about logic functions and their implementation, arithmetic circuits, multiplexers, decoders, flip-flops, counters, finite-state machines, and other standard circuits, there is less agreement about the means used to expose students to this material. In particular, when and how should the students discover CAD tools, and what laboratory exercises provide the best learning experience?

Today it is highly advisable to introduce a hardware description language (HDL) as soon as possible. Without an HDL it is impossible to exploit properly the capabilities of CAD tools and FPGA-based laboratory equipment. A prudent choice of HDL is either Verilog or VHDL. It should be noted that Verilog is winning the battle in the industrial environment of North America, so it is likely that it will gain greater favor with academics in the near future.

Our experience shows that the HDL can be introduced surprisingly early. Moreover, the instructor need not spend an inordinate amount of time teaching the intricacies of the language. Students are keen to learn and use the HDL because of its obvious practical value, hence they are willing to learn on their own many details that are illustrated in examples given in the textbook. During lectures, the instructor has to focus on explaining the important differences between the HDL and computer programming languages that students are familiar with. For example, explaining the key differences between Verilog and C can lead to fascinating lectures. Since the HDL will be used in laboratory exercises, which requires a certain amount of homework preparation, the material that should be taught in the classroom may be covered in as little as three to four lectures. This approach is particularly viable if the textbook integrates efficiently the discussion of logic circuit concepts and their possible HDL descriptions.

3 Introducing HDL - A Practical Approach

A good understanding of computer hardware must be based on a good understanding of underlying logic circuits. An HDL, particularly when used at the behavioral level, can mask many important aspects of logic circuits. Therefore, it is important to find a good balance between teaching the students the essence of

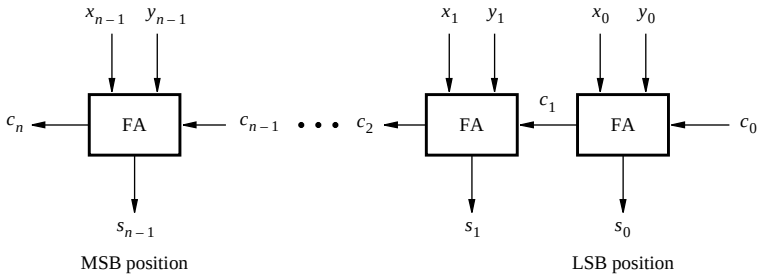

Figure 1: An n-bit ripple-carry adder.

circuits and the efficiency of design using the HDL and CAD tools. It is particularly important that using the HDL does not obscure the existence of fundamental logic blocks such as gates and flip-flops. To illustrate this notion, we will consider an example based on a ripple-carry adder, using Verilog as the HDL [1].

Figure 1 shows the general structure of an n-bit ripple-carry adder, comprising a cascade of full-adder circuits. Knowing that the full-adder is defined by the logic expressions

$$\begin{aligned} s &= x \oplus y \oplus Cin \\ Cout &= x \cdot y + x \cdot Cin + y \cdot Cin \end{aligned}$$

it is easy to visualize the functionality of the cascaded circuit, as well as the propagation delay due to the rippling of the carry. Using these expressions, the full-adder can be defined in Verilog as shown in Figure 2.

```
module  fulladd (Cin, x, y, s, Cout);
   input  Cin, x, y;
   output  s, Cout;

   assign  s = x ^ y ^ Cin;
   assign  Cout = (x & y) | (x & Cin) | (y & Cin);

endmodule
```

Figure 2: Verilog code for the full-adder.

Now, we can specify a ripple-carry adder structurally as indicated in Figure 3. To keep the example simple, this specification defines the inputs X and Y, as well as the sum S, as four-bit vectors. The internal carries are defined as a three-bit vector C. The structure of the resulting circuit is the same as in Figure 1.

This would be an awkward way of describing a larger n-bit adder, so we can use a generic specification instead. The ripple-carry adder in Figure 1 can be described using the expressions

$$\begin{aligned} s_k &= x_k \oplus y_k \oplus c_k \\ c_{k+1} &= x_k y_k + x_k c_k + y_k c_k \end{aligned}$$

```
module  adder4 (carryin, X, Y, S, carryout);
   input  carryin;
   input  [3:0] X, Y;
   output  [3:0] S;
   output  carryout;
   wire  [3:1] C;

   fulladd  stage0  (carryin, X[0], Y[0], S[0], C[1]);
   fulladd  stage1  (C[1], X[1], Y[1], S[1], C[2]);
   fulladd  stage2  (C[2], X[2], Y[2], S[2], C[3]);
   fulladd  stage3  (C[3], X[3], Y[3], S[3], carryout);

endmodule
```

Figure 3: A four-bit adder.

for $k = 0, 1, \ldots, n-1$. A Verilog specification of the adder based on these expressions is given in Figure 4. It is apparent that this approach also implements the cascaded adder structure. Next, the students should learn that Verilog includes higher-level constructs for specification of commonly used circuits. One such construct uses the arithmetic assignment statement, which allows the adder to be specified as shown in Figure 5. We can add to this circuit the capability to produce the carry-out and arithmetic overflow signals as presented in Figure 6. The expressions for these two signals are

$$\begin{aligned} carryout &= x_{n-1}y_{n-1} + x_{n-1}\overline{s}_{n-1} + y_{n-1}\overline{s}_{n-1} \\ overflow &= carryout \oplus x_{n-1} \oplus y_{n-1} \oplus s_{n-1} \end{aligned}$$

They can be derived as a useful exercise.

A more elegant way of specifying the same circuit is given in Figure 7. It uses an $(n+1)$-bit vector named *Sum*. The extra bit, *Sum*$[n]$, becomes the carry-out from bit position $n-1$ in the adder. To make the addends $n+1$ bits long, the vectors X and Y have a zero concatenated on the left side. This conveniently introduces the Verilog concatenate operator. It also ensures that the students see X and Y as bits in a circuit rather than just numbers.

Having introduced the idea of concatenation in Verilog, our circuit can be defined more compactly as shown in Figure 8. Finally, at this point the students can see that even the full-adder circuit can be defined behaviorally as depicted in Figure 9.

The progressive sequence of showing the students different ways in which Verilog code can specify an n-bit adder teaches a number of important aspects of Verilog. It shows the difference between structural and behavioral approaches in defining circuits. It illustrates how a **for** loop generates a cascade by replicating a subcircuit n times. It indicates that powerful statements, such as the arithmetic assignment statement, exist which lead to simple and easily understandable code. It also shows how clever ideas, exploiting the notion of concatenation in our example, can be used to good effect.

It is important to understand that a behavioral specification will not necessarily lead to a circuit structured in the form that the designer may envisage, perhaps as learned from a textbook. When the Verilog compiler of a given CAD tool encounters a construct used to define a commonly used circuit, it will attempt to use a predefined module from a library of parameterized modules provided with the CAD tool. Moreover, if the designed circuit is to be implemented in a technology such as an FPGA, then the final implementation will be in the form of logic elements used in a particular FPGA device. Thus, the implementation may not involve the basic gates that one saw in the lectures!

4 Computer Organization

A good laboratory is essential for conveying the essence of the various structures found in computer systems. The students should learn what a computer looks like through the eyes of a programmer interested primarily in using the machine and a designer intent on developing the hardware needed to build systems. At the University of Toronto we use the Ultragizmo board, which is a custom board containing a Motorola 68000-based microcontroller device, an FPGA device and a variety of interfaces that allow the board to be connected to our main laboratory system which includes a full networking capability. We also use Altera's UP-1 boards, which include programmable logic devices, to provide additional capability.

Typical experiments include investigation of simple I/O using parallel and serial ports, interrupts, adding SRAM chips, DMA controllers, design of arithmetic circuits, A/D and D/A interfaces, and various applications such as processing of sound and controlling simple Lego-implemented robots. At the end of our courses, we tend to have a three-week project for which students may implement anything that the instructor deems interesting. The project may entail even designing a simple processor, or building an interesting system based on an existing soft-core processor that may be instantiated in the FPGA.

The FPGA device makes it possible to quickly implement relatively complex circuits needed in specific experiments. Of course, these circuits have to be specified in the HDL. This raises an interesting question about the necessary competence of students in terms of the HDL use. More specifically, is the knowledge gained in the introductory logic course sufficient to deal successfully with more ambitious designs needed in this laboratory.

Instructors habitually complain that the students do not know as much as they should. Specifically, given a rather limited exposure to the HDL in the logic course, they cannot immediately tackle more demanding tasks in the computer organization laboratory. Indeed, this may be true, particularly in the case of weaker students. However, with just a single refresher tutorial on DOs and DON'Ts of the HDL the students can be brought to a level where they can handle the requirements of the laboratory. As with most subjects, an early exposure to the HDL, followed by a short review and subsequent intensive use in the follow-on course, will leave the students with reasonable competence and a great deal of self-satisfaction.

The described approach has been successful in our practice. It has led to the development of three books [1-3]. The feedback from our students has been very positive.

5 References

1. S. Brown and Z. Vranesic, *Fundamentals of Digital Logic with Verilog Design*, McGraw-Hill, 2002.

2. S. Brown and Z. Vranesic, *Fundamentals of Digital Logic with VHDL Design*, McGraw-Hill, 2000.

3. V.C. Hamacher, Z. Vranesic and S. Zaky, *Computer Organization, ed. 5*, McGraw-Hill, 2002.

```
module  addern (carryin, X, Y, S, carryout);
   parameter  n = 32;
   input  carryin;
   input  [n−1:0] X, Y;
   output  [n−1:0] S;
   output  carryout;
   reg  [n−1:0] S;
   reg  carryout;
   reg  [n:0] C;
   integer  k;

   always @(X or Y or carryin)
   begin
      C[0] = carryin;
      for (k = 0; k < n; k = k+1)
      begin
         S[k] = X[k] ^ Y[k] ^ C[k];
         C[k+1] = (X[k] & Y[k]) | (X[k] & C[k]) | (Y[k] & C[k]);
      end
      carryout = C[n];
   end

endmodule
```

Figure 4: A generic specification of a ripple-carry adder.

```
module  addern (carryin, X, Y, S);
   parameter  n = 32;
   input  carryin;
   input  [n−1:0] X, Y;
   output  [n−1:0] S;
   reg  [n−1:0] S;

   always @(X or Y or carryin)
      S = X + Y + carryin;

endmodule
```

Figure 5: Specification of an n-bit adder using arithmetic assignment.

```
module addern (carryin, X, Y, S, carryout, overflow);
   parameter n = 32;
   input carryin;
   input [n−1:0] X, Y;
   output [n−1:0] S;
   output carryout, overflow;
   reg [n−1:0] S;
   reg carryout, overflow;

   always @(X or Y or carryin)
   begin
      S = X + Y + carryin;
      carryout = (X[n−1] & Y[n−1]) | (X[n−1] & ∼S[n−1]) | (Y[n−1] & ∼S[n−1]);
      overflow = carryout ^ X[n−1] ^ Y[n−1] ^ S[n−1];
   end

endmodule
```

Figure 6: An n-bit adder with carry-out and overflow signals.

```
module addern (carryin, X, Y, S, carryout, overflow);
   parameter n = 32;
   input carryin;
   input [n−1:0] X, Y;
   output [n−1:0] S;
   output carryout, overflow;
   reg [n−1:0] S;
   reg carryout, overflow;
   reg [n:0] Sum;

   always @(X or Y or carryin)
   begin
      Sum = {1'b0, X} + {1'b0, Y} + carryin;
      S = Sum[n−1:0];
      carryout = Sum[n];
      overflow = carryout ^ X[n−1] ^ Y[n−1] ^ S[n−1];
   end

endmodule
```

Figure 7: A different specification of n-bit adder.

```
module  addern (carryin, X, Y, S, carryout, overflow);
   parameter  n = 32;
   input  carryin;
   input  [n−1:0] X, Y;
   output  [n−1:0] S;
   output  carryout, overflow;
   reg  [n−1:0] S;
   reg  carryout, overflow;

   always @(X or Y or carryin)
   begin
      {carryout, S} = X + Y + carryin;
      overflow = carryout ^ X[n−1] ^ Y[n−1] ^ S[n−1];
   end

endmodule
```

Figure 8: Simplified complete specification of an n-bit adder.

```
module  fulladd (Cin, x, y, s, Cout);
   input  Cin, x, y;
   output  s, Cout;
   reg  s, Cout;

   always @(x or y or Cin)
      {Cout, s} = x + y + Cin;

endmodule
```

Figure 9: Behavioral specification of a full-adder.