A key benefit of HDL design methodology is the ability to thoroughly simulate logic before committing a netlist to a real chip. Because HDL is a programming methodology, it can be arbitrarily manipulated in a software simulation environment. The simulator allows a *test bench* to be written in either the HDL or another language (e.g., C/C++) that is responsible for creating stimulus to be applied to the logic modules. Widely used simulators include Cadence's NC-Sim, Model Technology's ModelSim, and Synopsys' VCS and Scirocco. A distinction is made between synthesizable and nonsynthesizable code when writing RTL and test benches. Synthesizable code is that which represents the logic to be implemented in some type of chip. Nonsynthesizable code is used to implement the test bench and usually contains constructs specifically designed for simulation that cannot be converted into real logic through synthesis.

An example of a test bench for the preceding Verilog module might consist of three number generators that apply pseudo-random test stimulus to the three input ports. Automatic verification of the logic would be possible by having the test bench independently compute the function $Y = A\&B + \overline{A}\&C$ and then check the result against the module's output. Such simulation, or verification, techniques can be used to root out the great majority of bugs in a complex design. This is a tremendous feature, because fixing bugs after an ASIC has been fabricated is costly and time consuming. Even in cases in which a PLD is used, it is usually faster to isolate and fix a bug in simulation than in the laboratory. In simulation, there is immediate access to all internal nodes of the design. In the lab, such access may prove quite difficult to achieve.

Verilog and VHDL both support simulation constructs that facilitate writing effective test benches. It is important to realize that these constructs are usually nonsynthesizable (e.g., a random number generator) and that they should be used only for writing test code rather than actual logic.

Gate/instance-level coding is quite useful and is used to varying degrees in almost every design, but the real power of HDL lies at the RTL and behavioral levels. Except in rare circumstances where absolute control over gates is required, instance-level coding is used mainly to connect different modules together. Most logic is written in RTL and behavioral constructs which are often treated together, hence the reason that synthesizable HDL code is often called RTL. Expressing logic in RTL frees the engineer from having to break everything down into individual gates and transfers this responsibility onto the synthesis software. The result is a dramatic increase in productivity and maintainability, because logical representations become concise. The example in Fig. 10.1 can be rewritten in Verilog RTL in multiple styles as shown in Fig. 10.2.

Each of these three styles has its advantages, each is substantially more concise and readable than the gate/instance-level version, and the styles can be freely mixed within the same module according to the engineer's preference. Style number 1 is a *continuous assignment* and makes use of the default wire data type for the output port. A wire is applicable here, because it is implicitly connecting two entities: the logic function and the output port. Continuous assignments are useful in certain cases, because they are concise, but they cannot get too complex without becoming unwieldy.

Style number 2 uses the always block, a keyword that tells the synthesis and simulation tools to perform the specified operations whenever a variable in its *sensitivity list* changes. The sensitivity list defines the variables that are relevant to the always block. If not all relevant variables are included in this list, incorrect results may occur. Always blocks are one of Verilog's fundamental constructs. A design may contain numerous always blocks, each of which contains logic functions that are activated when a variable in the sensitivity list changes state. A combinatorial always block should normally include all of its input variables in the sensitivity list. Failure to do so can lead to unexpected simulation results, because the always block will not be activated if a variable changes state and is not in the sensitivity list.

Style number 3 also uses the always block, but it uses a logical if…else construct in place of Boolean expression. Such logical representations are often preferable so that an engineer can concentrate on the functionality of the logic rather than deriving and simplifying Boolean algebra.

```
module my_logic (
  A, B, C, Y
);

input A, B, C;
output Y;

// Style #1: continuous assignment

assign Y = (A && B) || (!A && C);

// Style #2: behavioral assignment

reg Y;

always @(A or B or C)
begin
  Y = (A && B) || (!A && C);
end

// Style #3: if...then construct

reg Y;

always @(A or B or C)
begin
  if (A)
    Y = B;
  else
    Y = C;
end

endmodule
```

FIGURE 10.2 Verilog RTL-level design.

The two examples shown thus far illustrate basic Verilog HDL syntax with combinatorial logic. Clearly, synchronous logic is critical to digital systems, and it is fully supported by HDLs. D-type flip-flops are most commonly used in digital logic design, and they are directly inferred by using the correct RTL syntax. As always, gate-level instances of flops can be invoked, but this is discouraged for the reasons already discussed. Figure 10.3 shows the Verilog RTL representation of two flops, one with a synchronous reset and the other with an asynchronous reset.

The first syntactical difference to notice is the Verilog keyword *posedge*. Posedge and its complement, *negedge*, modify a sensitivity list variable to activate the always block only when it transitions. Synthesis tools are smart enough to recognize these keywords and infer a clocked flop. Clocked always blocks should not include normal regs or wires in the sensitivity list, because it is only desired to activate the block on the active clock edge or when an optional asynchronous reset transitions.

At reset, a default 0 value is assigned to Q. Constants in Verilog can be explicitly sized and referenced to a particular radix. Preceding a constant with `b denotes it as binary (`h is hex, and `d is decimal). Preceding the radix identifier with a number indicates the number of bits occupied by that constant.

Another syntactical difference to note is the use of a different type of assignment operator: <= instead of =. This is known as a non-blocking (<=) assignment as compared to a blocking (=) assignment. It is considered good practice to use non-blocking assignments when inferring flops, because