7400 devices such as the 74LS138. As digital systems grow more complex, the chances increase that suitable off-the-shelf logic will be either unavailable or impractical to use. The answer is to design and implement custom logic rather than relying solely on a third party to deliver a solution that does exactly what is needed.

Logic design techniques differ according to the scale of logic being implemented. If only a few gates are needed to implement a custom address decoder or timer, the most practical solution may be to write down truth tables, extract Boolean equations with Karnaugh maps, select appropriate 7400 devices, and draw a schematic diagram. This used to be the predominant means of designing logic for many applications, especially where the cost and time of building a custom IC was prohibitive. The original Apple and IBM desktop computers were designed this way, as witnessed by their rows of 7400 ICs.

When functions grow more complex, it becomes awkward and often simply impossible to implement the necessary logic using discrete 7400 devices. Reasons vary from simple density constraints—how much physical area would be consumed by dozens of 7400 ICs—to propagation delay constraints—how fast a signal can pass through multiple discrete logic gates. The answer to many of these problems is custom and semicustom logic ICs. The exact implementation technology differs according to the cost, speed, and time constraints of the application, but the underlying concept is to pack many arbitrary logic functions and flip-flops into one or more large ICs. An *application specific integrated circuit* (ASIC) is a chip that is designed with logic specific to a particular task and manufactured in a fixed configuration. A *programmable logic device* (PLD) is a chip that is manufactured with a programmable configuration, enabling it to serve in many arbitrary applications.

Once the decision is made to implement logic within custom or semicustom logic ICs, a design methodology is necessary to move ahead and solve the problem at hand. It is possible to use the same design techniques in these cases as used for discrete 7400 logic implementations. The trouble with graphical logic representations is that they are bulky and prone to human error. Hardware description languages were developed to ease the implementation of large digital designs by representing logic as Boolean equations as well as through the use of higher-level semantic constructs found in mainstream computer programming languages.

Aside from several proprietary HDLs, the major industry standard languages for logic design are *Verilog* and *VHDL* (Very high speed integrated circuits HDL). Verilog began as a proprietary product that was eventually transformed into an open standard. VHDL was developed as an open standard from the beginning. The two languages have roughly equal market presence and claim religious devotees on both sides. This book does not seek to justify one HDL over the other, nor does it seek to provide a definitive presentation of either. For the sake of practicality, Verilog is chosen to explain HDL concepts and to serve in examples of how HDLs are used in logic design and implementation.

HDLs provide logical representations that are abstracted to varying degrees. According to the engineer's choice or contextual requirements, logic can be represented at the *gate/instance* level, the *register transfer level* (RTL), or the *behavioral* level. Gate/instance-level representations involve the manual instantiation of each physical design element. An element can be an AND gate, a flop, a multiplexer, or an entire microprocessor. These decisions are left to the engineer. In a purely gate/instance-level HDL design, the HDL source code is nothing more than a glorified list of instances and connections between the input/output ports of each instance. The Verilog instance representation of $Y = A\&B + \overline{A}\&C$ is shown in Fig. 10.1. It is somewhat cumbersome but provides full control over the final implementation.

The brief listing in Fig. 10.1 incorporates many basic pieces of a generic Verilog module. First, the module is named and declared with its list of ports. Following the port list, the ports are defined as being inputs or outputs. In this case, the ports are all single net vectors, so no indices are supplied. Next is the main body that defines the function of the module. Verilog recognizes two major variable types: *wires* and *regs*. Wires simply connect two or more entities together. Regs can be assigned val-

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

input A, B, C;
output Y;

wire and1_out, and2_out, notA;

and_gate u_and1 (
  .in1 (A),
  .in2 (B),
  .out (and1_out)
);

not_gate u_not (
  .in  (A),
  .out (notA)
);

and_gate u_and2 (
  .in1 (notA),
  .in2 (C),
  .out (and2_out)
);

or_gate u_or (
  .in1 (and1_out),
  .in2 (and2_out),
  .out (Y)
);

endmodule
```

FIGURE 10.1 Verilog gate/instance level design.

ues at discrete events, as will be soon discussed. When ports are defined, they are assumed to be wires unless declared otherwise. An output port can be declared as a type other than wire.

Being that this example is a gate/instance-level design, all logic is represented by instantiating other modules that have been defined elsewhere. A module is instantiated by invoking its name and following it with an instance name. Here, the common convention of preceding the name with "u_" is used, and multiple instances of the same module type are differentiated by following the name with a number. Individual ports for each module instance are explicitly connected by referencing the port name prefixed with a period and then placing the connecting variable in parentheses. Ports can be implicitly connected by listing only connecting variables in the order in which a module's ports are defined. This is generally considered poor practice, because it is prone to mistakes and is difficult to read.

HDL's textual representation of logic is converted into actual gates through a process called *logic synthesis*. A synthesis program parses the HDL code and generates a *netlist* that contains a detailed list of low-level logic gates and their interconnecting nets, or wires. Synthesis is usually done with a specific implementation target in mind, because each implementation technology differs in the logic primitives that it provides as basic building blocks. The primitive library for an ASIC will differ from that of a PLD, for example. Once synthesis is performed, the netlist can be transformed into a working chip and, hence, a working product.