

FIGURE 9.1 IEEE-recommended dimensions and their construction for logic-gate symbols. (a) NAND gate (b) exclusive-OR gate (an OR gate is a subset).

Figure 9.2 shows some pictorial definitions of objects you can use in a simple schematic. We shall discuss the different types of objects that might appear in an ASIC schematic first and then discuss the different types of connections.

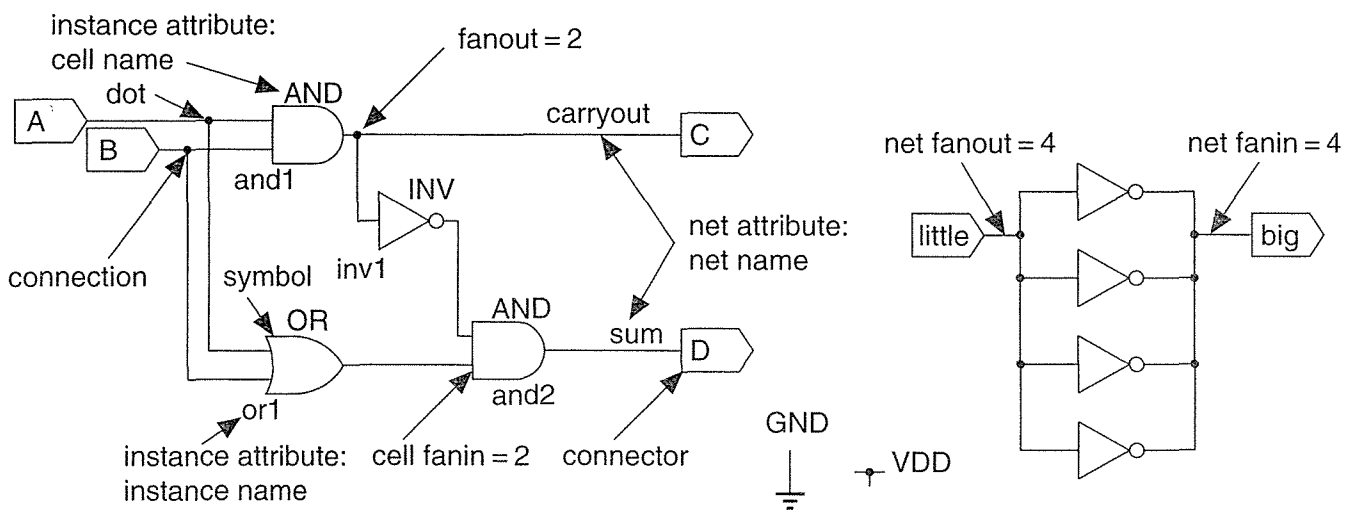


FIGURE 9.2 Terms used in circuit schematics.

Schematic-entry tools for ASIC design are similar to those for printed-circuit board (PCB) design. The basic object on a PCB schematic is a **component** or **device**—a TTL IC or resistor, for example. There may be several hundred components on a typical PCB. If we think of a logic gate on an ASIC as being equivalent to a component on a PCB, then a large ASIC contains hundreds of thousands of components. We can normally draw every component on a few schematic sheets for a PCB, but drawing every component on an ASIC schematic is impractical.

9.1.1 Hierarchical Design

Hierarchy reduces the size and complexity of a schematic. Suppose a building has 10 floors and contains several hundred offices but only three different basic office plans. Furthermore, suppose each of the floors above the ground floor that contains the lobby is identical. Then the plans for the whole building need only show detailed plans for the ground floor and one of the upper floors. The plans for the upper floor need only show the locations of each office and the office type. We can then use a separate set of three detailed plans for each of the different office types. All these different plans together form a nested structure that is a **hierarchical design**. The plan for the whole building is the top-level plan. The plans for the individual offices are the lowest level. To clarify the relationship between different levels of hierarchy we say that a **subschematic** (an office) is a **child** of the **parent** schematic (the floor containing offices). An electrical schematic can contain subschematics. The subschematic, in turn, may contain other subschematics. Figure 9.3 illustrates the principles of schematic hierarchical design.

The alternative to hierarchical design is to draw all of the ASIC components on one giant schematic, with no hierarchy, in a **flat design**. For a modern ASIC containing thousands or more logic gates using a flat design or a flat schematic would be hopelessly impractical. Sometimes we do use **flat netlists** though.

9.1.2 The Cell Library

Components in an ASIC schematic are chosen from a library of cells. Library elements for all types of ASICs are sometimes also known as **modules**. Unfortunately the term *module* will have a very specific meaning when we come to discuss hardware description languages. To avoid any chance of confusion I use the term *cell* to mean either a cell, a module, a macro, or a book from an ASIC library. Library cells are equivalent to the offices in our office building.

Most ASIC companies provide a **schematic library** of primitive gates to be used for schematic entry. The first problem with ASIC schematic libraries is that there are no naming conventions. For example, a primitive two-input NAND gate in a Xilinx FPGA library does not have the same name as the two-input NAND gate in an LSI Logic gate-array library. This means that you cannot take a schematic that you used to create a prototype product using a Xilinx FPGA and use that schematic to create an LSI Logic gate array for production (something you might very likely want to do). As soon as you start entering a schematic using a library from an ASIC

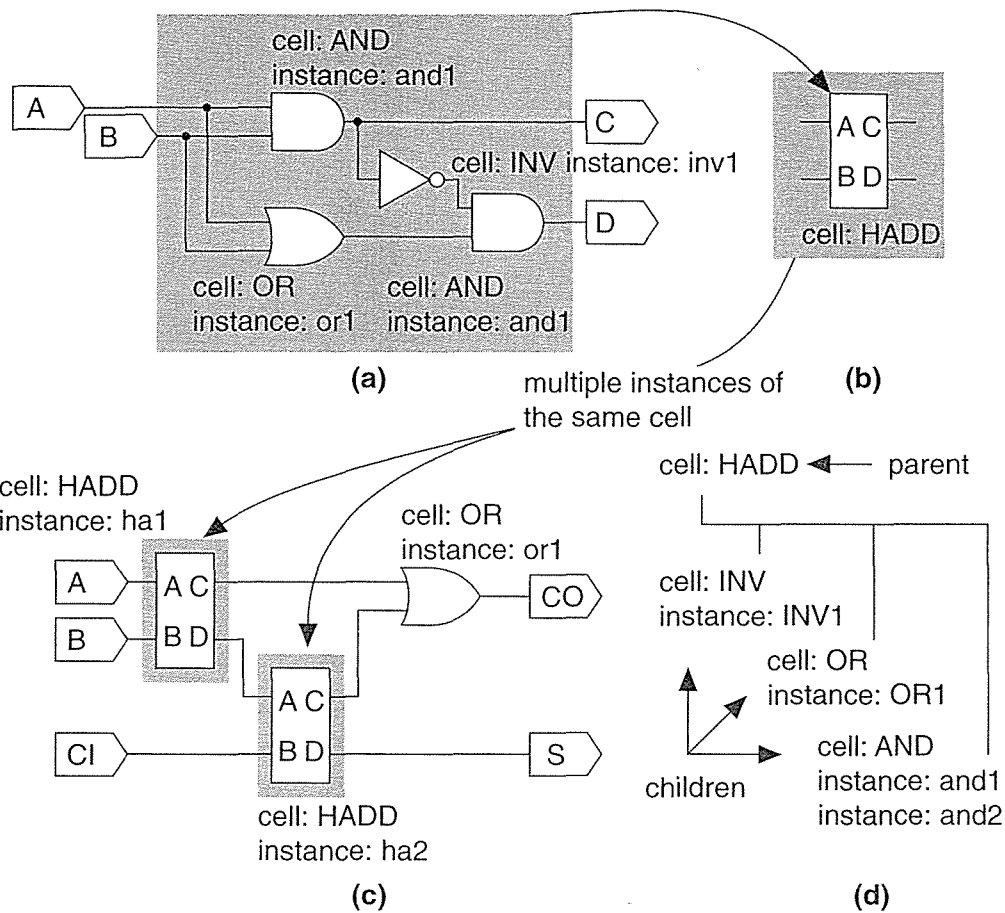


FIGURE 9.3 Schematic example showing hierarchical design. (a) The schematic of a half-adder, the subschematic of cell HADD. (b) A schematic symbol for the half adder. (c) A schematic that uses the half-adder cell. (d) The hierarchy of cell HADD.

vendor, you are, to some extent, making a commitment to use that vendor's ASIC. Most ASIC designers are much happier maintaining a large degree of vendor independence.

A second problem with ASIC schematic libraries is that there are no standards for cell behavior. For example, a two-input MUX in an Actel library operates so that the input labeled A is selected when the MUX select input S = '0'. A two-input MUX in a VLSI Technology library operates in the reverse fashion, so that the input labeled B is selected when S = '0'. These types of differences can cause hard-to-find problems when trying to convert a schematic from one vendor to another by hand. These problems make changing or **retargeting** schematics from one vendor to another difficult. This process is sometimes known as **porting** a design.

Library cells that represent basic logic gates, such as a NAND gate, are known as **primitive cells**, usually referred to just as *cells*. In a hierarchical ASIC design a cell may be a NAND gate, a flip-flop, a multiplier, or even a microprocessor, for example. To use the office building analogy again, each of the three basic office types is a primitive cell. However, the plan for the second floor is also a cell. The second-floor cell is a subschematic of the schematic for the whole building. Now we see why the commonly accepted use of the term *cell* in schematic entry can be so confusing. The term *cell* is used to represent both primitive cells and subschematics. These are two different, but closely related, things.

There are two types of macros for MGAs and programmable ASICs. The most common type of macro is a **hard macro** that includes placement information. A hard macro can change in position and orientation, but the relative location of the transistors, other layout, and wiring inside the macro is fixed. A **soft macro** contains only connection information (between transistors for a gate array or between logic cells for a programmable ASIC). Thus the placement and wiring for a soft macro can vary. This means that the timing parameters for a soft macro can only be determined after you complete the place-and-route step. For this reason the basic library elements for MGAs and programmable ASICs, such as NAND gates, flip-flops, and so on, are hard macros.

A standard cell contains layout information on all mask levels. An MGA hard macro contains layout information on just the metal, contact, and via layers. An MGA soft macro or programmable ASIC macro does not contain any layout information at all, just the details of connections to be made inside the macro.

We can stretch the office building analogy to explain the difference between hard and soft macros. A hard macro would be an office with fixed walls in which you are not allowed to move the furniture. A soft macro would be an office with partitions in which you can move the furniture around and you can also change the shape of your office by moving the partitions.

9.1.3 Names

Each of the cells, primitive or not, that you place on an ASIC schematic has a **cell name**. Each use of a cell is a different **instance** of that cell, and we give each instance a unique **instance name**. A cell instance is somewhere between a copy and a reference to a cell in a library. An analogy would be the pictures of hamburgers on the wall in a fast-food restaurant. The pictures are somewhere between a copy and a reference to a real hamburger.

We represent each cell instance by a picture or **icon**, also known as a **symbol**. We can represent primitive cells, such as NAND and NOR gates, with familiar icons that look like spades and shovels. Some schematic editors offer the option of switching between these familiar icons and using the rectangular IEEE standard symbols for logic gates. Unfortunately the term *icon* is also often used to refer to any of the pictures on a schematic, including those that represent subschematics. There is no accepted way to differentiate between an icon that represents a primitive cell and

one that represents a subschematic that may be in turn a collection of primitive cells. In fact, there is usually no easy way to tell by looking at a schematic which icons represent primitive cells and which represent subschematics.

We will have three different icons for each of the three different primitive offices in the imaginary office building example of Section 9.1.1. We also will have icons to represent the ground floor and the plan for the other floors. We shall call the common plan for the second through tenth floors, `Floor`. Then we say that the second floor is an instance of the cell name `Floor`. The third through tenth floors are also instances of the cell name `Floor`. The same icon will be used to represent the second through tenth floors, but each will have a unique instance name. We shall give them instance names: `FloorTwo`, `FloorThree`, . . . , `FloorTen`. We say that `FloorTwo` through `FloorTen` are unique instance names of the cell name `Floor`.

At the risk of further confusion I should point out that, strictly speaking, the definition of a primitive cell depends on the type of library being used. Schematic-entry libraries for the ASIC designer stop at the level of NAND gates and other similar low-level logic gates. Then, as far as the ASIC designer is concerned, the primitive cells are these logic gates. However, from the view of the library designer there is another level of hierarchy below the level of logic gates. The library designer needs to work with libraries that contain schematics of the gates themselves, and so at this level the primitive cells are transistors.

Let us look at the building analogy again to understand the subtleties of primitive cells. A building contractor need only concern himself with the plans for our office building down to the level of the offices. To the building contractor the primitive cells are the offices. Suppose that the first of the three different office types is a corner office, the second office type has a window, and a third office type is without a window. We shall call these office cells: `CornerOffice`, `WindowOffice`, and `NoWindowOffice`. These cells are primitive cells as far as the contractor is concerned. However, when discussing the plans with a client, the architect of our building will also need to see how each office is furnished. The architect needs to see a level of detail of each office that is more complicated than needed by the building contractor. The architect needs to see the cells that represent the tables, chairs, and desks that make up each type of office. To the architect the primitive cells are a library containing cells such as `chair`, `table`, and `desk`.

9.1.4 Schematic Icons and Symbols

Most schematic-entry programs allow the designer to draw special or custom icons. In addition, the schematic-entry tool will also usually create an icon automatically for a subschematic that is used in a higher-level schematic. This is a **derived icon**, or **derived symbol**. The external connections of the subschematic are automatically attached to the icon, usually a rectangle.

Figure 9.4(c) shows what a derived icon for a cell, `DLAT`, might look like (we could also have drawn this by hand). The subschematic for `DLAT` is shown in Figure 9.4(b). We say that the inverter with the instance name `inv1` in the subsche-

matic is a **subcell** (or submodule) of the cell DLAT. Alternatively we say that cell instance `inv1` is a child of the cell DLAT, and cell DLAT is a parent of cell instance `inv1`.

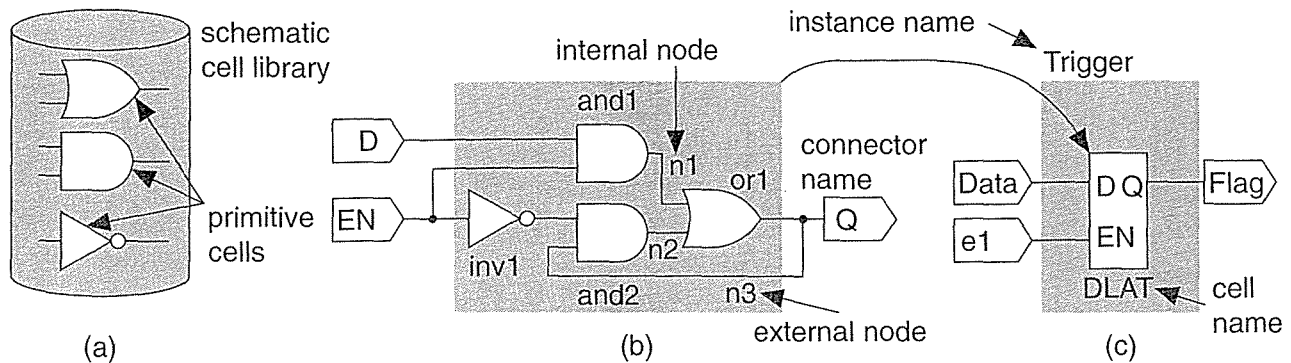


FIGURE 9.4 A cell and its subschematic. (a) A schematic library containing icons for the primitive cells. (b) A subschematic for a cell, DLAT, showing the instance names for the primitive cells. (c) A symbol for cell DLAT.

Figure 9.5(a) shows a more complex subschematic for a 4-bit latch. Each primitive cell instance in this schematic must have a unique name. This can get very tiresome for large circuits. Instead of creating complex, but repetitive, subschematics for complex cells we can use hierarchy.

Figure 9.5(b) shows a hierarchical subschematic for a cell `FourBit`, which in turn uses four instances of the cell DLAT. The four instances of DLAT in Figure 9.5(b) have different instance names: `L1`, `L2`, `L3`, and `L4`. Notice that we cannot use just one name for the four instances of DLAT to indicate that they are all the same cell. If we did, we could not differentiate between `L1` and `L2`, for example.

The vertical row of instances in Figure 9.5(b) looks like a vector of elements. Figure 9.5(c) shows a **vectored instance** representing four copies of the DLAT cell. We say the **cardinality** of this instance is 4. Tools normally use bold lines or some other distinguishing feature to represent a vectored instance. The cardinality information is often shown as a vector. Thus `L[1:4]` represents four instances: `L[1]`, `L[2]`, `L[3]`, `L[4]`. This is convenient because now we can see that all subcells are identical copies of `L`, but we have a unique name for each.

Finally, as shown in Figure 9.5(d) we can create a new symbol for the 4-bit latch, `FourBit`. The symbol for `FourBit` has a 4-bit-wide input bus for the four D inputs, and a 4-bit wide output bus for the four Q outputs. The subschematic for `FourBit` could be either Figure 9.5(a), (b), or (c) (though the exact naming of the inputs and outputs and their attachment to the buses may be different in each case).

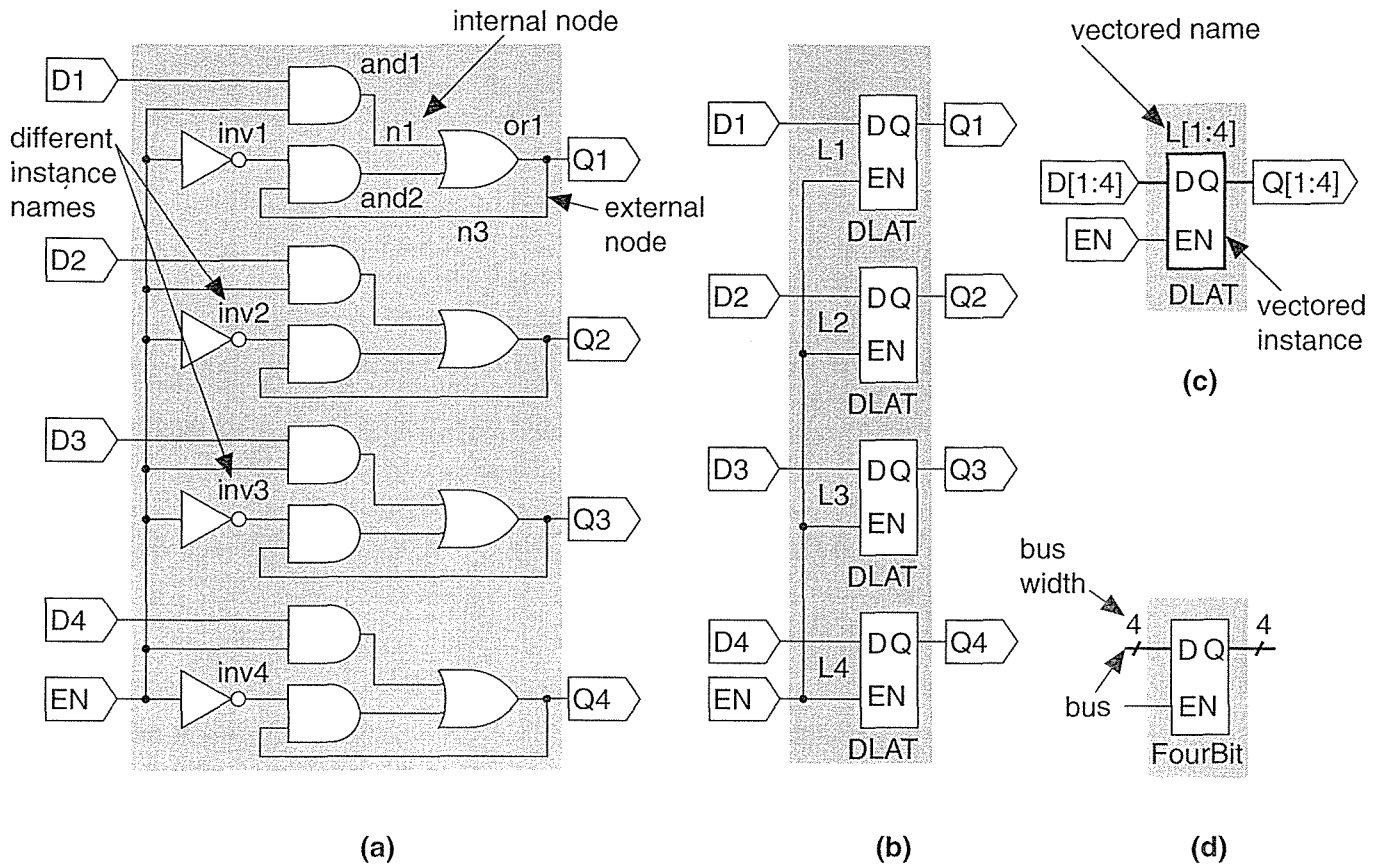


FIGURE 9.5 A 4-bit latch: (a) drawn as a flat schematic from gate-level primitives, (b) drawn as four instances of the cell symbol $DLAT$, (c) drawn using a vectored instance of the $DLAT$ cell symbol with cardinality of 4, (d) drawn using a new cell symbol with cell name $FourBit$.

We need a convention to distinguish, for example, between the inverter subcells, $inv1$, which are children of the cell $DLAT$, which are in turn children of the cell $FourBit$. Most schematic-entry tools do this by combining the instance names of the subcells in a hierarchical manner using a special character as a delimiter. For example, if we drew the subschematic as in Figure 9.5(b), the four inverters in $FourBit$ might be named $L1.inv1$, $L2.inv1$, $L3.inv1$, and $L4.inv1$. Once again this makes it clear that the inverters, $inv1$, are identical in all four subcells.

In our office building example, the offices are subcells of the cell $Floor$. Suppose you and I both have corner offices. Mine is on the second floor and yours is above mine on the third floor. My office is 211 and your office is 311. Another way to name our offices on a building plan might be $FloorTwo.11$ for my office and $FloorThree.11$ for your office. This shows that $FloorTwo.11$ is a subcell of

FloorTwo and also makes it clear that, apart from being on different floors, your office and mine are identical. Both our offices have instance names 11 and are instances of cell name Corner.

9.1.5 Nets

The schematics shown in Figure 9.4 contain both **local nets** and **external nets**. An example of a local net in Figure 9.4(b) is n1, the connection between the output terminal of the AND cell and1 to the OR cell or1. When the four copies of this circuit are placed in the parent cell FourBit in Figure 9.5(d), four copies of net n1 are created. Since the four nets named n1 are not actually electrically connected, even though they have the same name at the lowest hierarchical level, we must somehow find a way to uniquely identify each net.

The usual convention for naming nets in a hierarchical schematic uses the parent cell instance name as a prefix to the local net name. A special character (':' '/' '\$' '#' for example) that is not allowed to appear in names is used as a **delimiter** to separate the net name from the cell instance name. Supposing that we drew the subschematic for cell FourBit as shown in Figure 9.5(b), the four different nets labeled n1 might then become:

```
FourBit.L1:n1   FourBit.L2:n1   FourBit.L3:n1   FourBit.L4:n1
```

This naming is usually done automatically by the schematic-entry tool.

The schematic DLAT also contains three external nets: D, EN, and Q. The terminals on the symbol DLAT connect these nets to other nets in the hierarchical level above. For example, the signal Trigger:flag in Figure 9.4(c) is also Trigger.DLAT:Q. Each schematic tool handles this situation differently, and life becomes especially difficult when we need to refer to these nodes from a simulator outside the schematic tool, for example. HDLs such as VHDL and Verilog have a very precise and well-defined standard for naming nets in hierarchical structures.

9.1.6 Schematic Entry for ASICs and PCBs

A symbol on a schematic may represent a **component**, which may contain **component parts**. You are more likely to come across the use of components in a PCB schematic. A component is slightly different from an ASIC library cell. A simple example of a component would be a TTL gate, an SN74LS00N, that contains four 2-input NAND gates. We call an SN74LS00N a component and each of the individual NAND gates inside is a component part. Another common example of a component would be a resistor pack—a single package that contains several identical resistors.

In PCB design language a component label or name is a **reference designator**. A reference designator is a unique name attribute, such as R99, attached to each component. A reference designator, such as R99, has two pieces: an alpha prefix R and a numerical suffix 99. To understand the difference between reference designators and instance names, we need to look at the special requirements of PCB design.

PCBs usually contain packaged ASICs and other ICs that have pins that are soldered to a board. For rectangular, dual-in-line (DIP) packages the pins are numbered counterclockwise from the upper-left corner looking down on the package.

IC symbols have a **pin number** for each part in the package. For example, the TTL 74174 hex D flip-flop with clear, contains six parts: six identical D flip-flops. The IC symbol representing this device has six `PinNumber` attribute entries for the D input corresponding to the six possible input pins. They are pins 3, 4, 6, 11, 13, and 14.

When we need a flip-flop in our design, we use a symbol for a 74174 from a schematic library, suppose the symbol name is `dffClr`. We shall assign a unique instance name to the symbol, `CarryFF`. Now suppose we need another, identical, flip-flop and we call this `BitFF`. We do not mind which of the six flip-flop parts in a 74174 we use for `CarryFF` and `BitFF`. In fact they do not even have to be in the same package. We shall delay the choice of assigning `CarryFF` and `BitFF` to specific packages until we get to the PCB routing step. So at this point on our schematic we do not even know the pin numbers for `CarryFF` and `BitFF`. For example the D input to `CarryFF` could be pin 3, 4, 6, 11, 13, or 14.

The number of wire crossings on a PCB is minimized by careful assignment of components to packages and choice of parts within a package. So the placement-and-routing software may decide which part of which package to use for `CarryFF` and `BitFF` depending on which is easier to route. Then, only after the placement and routing is complete, are unique reference designators assigned to the component parts. Only at this point do we know where `CarryFF` is actually located on the PCB by referring to the reference designator, which points to a specific part in a specific package. Thus `CarryFF` might be located in `IC4` on our PCB. At this point we also know which pins are used for each symbol. So we now know, for example, that the D-input to `CarryFF` is pin 3 of `IC4`.

There is no process in ASIC design directly equivalent to the process of **part assignment** described above and thus no need to use reference designators. The reference-designator naming convention quickly becomes unwieldy if there are a large number of components in a design. For example, how will we find a NAND gate named `x3146` in an ASIC schematic with 100 pages? Instead, for ASICs, we use a naming scheme based on hierarchy.

In large hierarchical ASIC designs it is difficult to provide a unique reference designator to each element. For this reason ASIC designs use instance names to identify the individual components. Meaningful names can be assigned to low-level components and also the symbols that represent hierarchy. We derive the component names by joining all of the higher level cell names together. A special character is used as a delimiter and separates each level.

Examples of hierarchical instance names are:

```
cpu.alu.adder.and01
MotherBoard:Cache:RAM4:ReadBit4:Inverter2
```

9.1.7 Connections

Cell instances have **terminals** that are the inputs and outputs of the cell. Terminals are also known as **pins**, **connectors**, or **signals**. The term *pin* is widely used, but we shall try to use *terminal*, and reserve the term *pin* for the metal leads on an ASIC package. The term *pin* is used in schematic entry and routing programs that are primarily intended for PCB design.

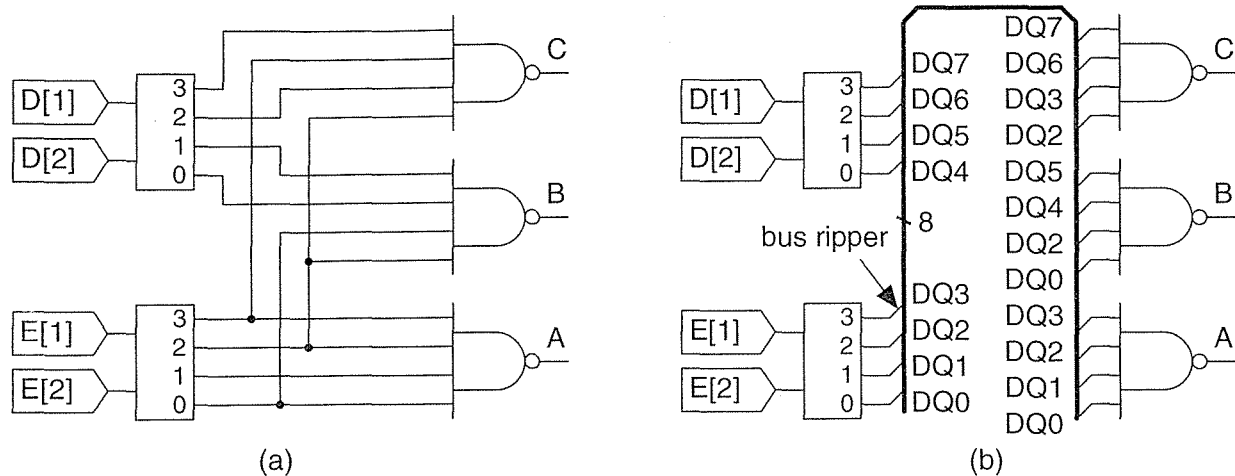


FIGURE 9.6 An example of the use of a bus to simplify a schematic. (a) An address decoder without using a bus. (b) A bus with bus rippers simplifies the schematic and reduces the possibility of making a mistake in creating and reading the schematic.

Electrical connections between cell instances use **wire segments** or **nets**. We can group closely related nets, such as the 32 bits of a 32-bit digital word, together into a **bus** or into **buses** (not busses). If signals on a bus are not closely related, we usually use the term **bundle** or **array** instead of bus. An example of a bundle might be a bus for a SCSI disk system, containing not only data bits but handshake and control signals too. Figure 9.6 shows an example of a bus in a schematic. If we need to access individual nets in a bus or a bundle, we use a **breakout** (also known as a **ripper**, an EDIF term, or **extractor**). For example, a breakout is used to access bits 0–7 of a 32-bit bus. If we need to rearrange bits on a bus, some schematic editors offer something called a **swizzle**. For example, we might use a swizzle to reorder the bits on an 8-bit bus so that the MSB becomes the LSB and so on down to the LSB, which now becomes the MSB. Swizzles can be useful. For example, we can multiply or divide a number by 2 by swizzling all the bits up or down one place on a bus.

9.1.8 Vectored Instances and Buses

So far the naming conventions are fairly standard and easy to follow. However, when we start to use vectored instances and buses (as is now common in large

ASICs), there are potential areas of difficulty and confusion. Figure 9.7(a) shows a schematic for a 16-bit latch that uses multiple copies of the cell `FourBit`. The buses are labeled with the appropriate bits. Figure 9.7(b) shows a new cell symbol for the 16-bit latch with 16-bit wide buses for the inputs, D, and outputs, Q.

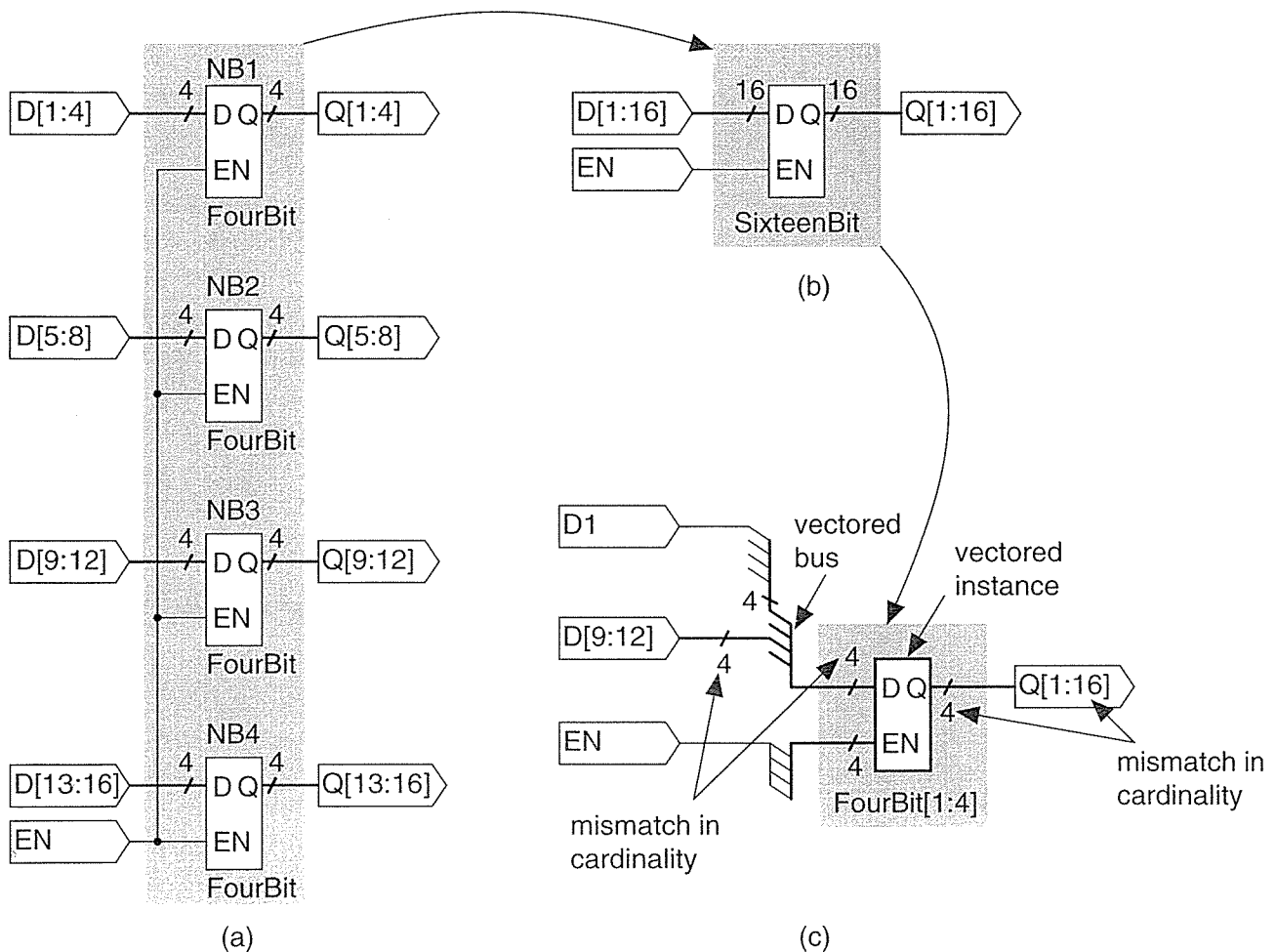


FIGURE 9.7 A 16-bit latch: (a) drawn as four instances of cell `FourBit`; (b) drawn as a cell named `SixteenBit`; (c) drawn as four multiple instances of cell `FourBit`.

Figure 9.7(c) shows an alternative representation of the 16-bit latch using a vectored instance of `FourBit` with cardinality 4. Suppose we wish to make a connection to expressly one bit, D1 (we have used D1 as the first bit rather than the more conventional D0 so that numbering is easier to follow). We also wish to make a connection to bits D9–D12, represented as D[9:12]. We do this using a bus ripper. Now

we have the rather awkward situation of bus naming shown in Figure 9.7(c). Problems arise when we have “buses of buses” because the numbers for the bus widths do not match on either side of a ripper. For this reason it is best to use the single-bus approach shown in Figure 9.7(b) rather than the vectored-bus approach of Figure 9.7(c).

9.1.9 Edit-in-Place

Figure 9.7(b) shows a symbol `SixteenBit`, which uses the subschematic shown in Figure 9.7(a) containing four copies of `FourBit`, named `NB1`, `NB2`, `NB3`, and `NB4` (the `NB` stands for nibble, which is half of a word; a nibble is 4 bits for 8-bit words). Suppose we use the schematic-entry program to edit the subcell `NB1.L1`, which is an instance of `DLAT` inside `NB1`. Perhaps we wish to change the D latch to a D latch with a reset, for example. If the schematic editor supports **edit-in-place**, we can edit a cell instance directly. After we edit the cell, the program will update all the `DLAT` subcells in the cell that is currently loaded to reflect the changes that have been made.

To see how edit-in-place works, consider our office building again. Suppose we wish to change some of the offices on each floor from offices without windows to offices with windows. We select the cell instance `FloorTwo`—that is, an instance of cell `Floor`. Now we choose the edit mode in the schematic-entry program. But wait! Do we want to edit the cell `Floor`, or do we want to edit the cell *instance* `FloorTwo`? If we edit the cell `Floor`, we will be making changes to all of the floors that use cell name `Floor`—that is, instances `FloorTwo` through `FloorTen`. If we edit the cell instance `FloorTwo`, then the second floor will become different from all the other floors. It will no longer be an instance of cell name `Floor` and we will have to create another cell name for the cell used by instance `FloorTwo`. This is like the difference between ordering just one hamburger without pickles and changing the picture on the wall that will change all future hamburgers.

Using edit-in-place we can edit the cell `Floor`. Suppose we change some of the cell instances of cell name `NoWindowOffice` to instances of cell name `WindowOffice`. When we finish editing and save the cell `Floor`, we have effectively changed all of the floors that contain instances of this cell.

Instead of editing a cell in place, you may really want to edit just one instance of a cell and leave any other instances unchanged. In this case you must create a new cell with a new symbol and new, unique cell name. It might also be wise to change the instance name of the new cell to avoid any confusion.

For example, we might change the third-floor plan of our office to be different from the other upper floors. Suppose the third floor is now an instance of cell name `FloorVIP` instead of `Floor`. We could continue to call the third floor cell instance `FloorThree`, but it would be better to rename the instance differently, `FloorSpecial` for example, to make it clear that it is different from all the other floors.

Some tools have the ability to **alias** nets. Aliasing creates a net name from the highest level in the design. Local names are net names at the lowest level such as D, and Q in a flip-flop cell. These local names are automatically replaced by the appropriate top-level names such as Clock1, or Data2, using a **dictionary**. This greatly speeds tracing of signals through a design containing many levels of hierarchy.

9.1.10 Attributes

You can attach a **name**, also known as an **identifier** or **label**, to a component, cell instance, net, terminal, or connector. You can also attach an **attribute**, or **property**, which describes some aspect of the component, cell instance, net, or connector. Each attribute has a name, and some attributes also have values. The most common problems in working with schematics and netlists, especially when you try to exchange schematic information between different tools, are problems in naming.

Since cells and their contents have to be stored in a database, a cell name frequently corresponds (or is mapped to) a filename. This then raises the problems of naming conventions including: case sensitivity, name-collision resolution, dictionaries, handling of “common” special characters (such as embedded blanks or underscores), other special characters (such as characters in foreign alphabets), first-character restrictions, name-length problems (only 28 characters are permitted on an NFS compatible filename), and so on.

9.1.11 Netlist Screener

A surprising number of problems can be found by checking a schematic for obviously fatal errors. A program that analyzes a schematic netlist for simple errors is sometimes called a **schematic screener** or **netlist screener**. Errors that can be found by a netlist screener include:

- unconnected cell inputs,
- unconnected cell outputs,
- nets not driven by any cells,
- too many nets driven by one cell,
- nets driven by more than one cell.

The screener can work continuously as the designer is creating the schematic or can be run as a separate program independently from schematic entry. Usually the designer provides attributes that give the screener the information necessary to perform the checks. A few of the typical attributes that schematic-entry programs use are described next.

A screener usually generates a list of errors together with the locations of the problem on the schematic where appropriate. Some editors associate an identifier, or **handle**, to every piece of a schematic, including comments and every net. Normally there is some convention to the assigned names such as a grid on a schematic. This works like the locator codes on a map, so that a net with A1 as part of the name is in

the upper-left-hand corner, for example. This allows you to quickly and uniquely find any problems found by a screener. The term *handle* is a computer programming term that is used in referring to a location in memory. Each piece of information on a schematic is stored in lists in memory. This technique breaks down completely when we move to HDLs.

Most schematic-entry programs work on a grid. The designer can control the size of the grid and whether it is visible or not. When you place components or wires you can instruct the editor to force your drawing to **snap to grid**. This means that drawing a schematic is like drawing on graph paper. You can only locate symbols, wires, and connections on grid points. This simplifies the internal mechanics of the schematic-entry program. It also makes the transfer of schematics between different EDA systems more manageable. Finally, it allows the designer to produce schematic diagrams that are cleaner in appearance and thus easier to read.

Most schematic-entry programs allow you to find components by instance name or cell name. The editor may either jump to the component location and center the graphic window on the component or highlight the component. More sophisticated options allow more complex searches, perhaps using **wildcard** matching. For example, to find all three-input NAND gates (primitive cell name ND3) or three-input NOR gates (primitive cell name NO3), you could search for cell name N*3, where * is a wildcard symbol standing for any character. The editor may generate a list of components, perhaps with page number and coordinate locations. Extensive find features are useful for large schematics where it quickly becomes impossible to find individual components.

Some schematic editors can complete **automatic naming** of reference designators or instance names to the schematic symbols either as the editor is running or as a postprocessing step. A component attribute, called a prefix, defines the prefix for the name for each type of component. For example, the prefix for all resistor component types may be R. Each time a prefix is found or a new instance is placed, the number in the reference designator or name is automatically incremented. Thus if the last resistor component type you placed was R99, the next time you place a resistor it would automatically be named R100.

For large schematics it is useful to be able to generate a report on the used and unused reference designators. An example would be:

```
Reference designator prefix: R
Unused reference designator numbers: 153, 154
Last used reference designator number: 180
```

If you need this feature, you probably are not using enough hierarchy to simplify your design.

During schematic entry of an ASIC design you will frequently need multiple copies of components. This often occurs during **datapath** design, where operations are carried out across multiple signals on a bus. A common example would be multiple copies of a latch, one for each signal on a bus. It is tedious and inefficient to have to draw and label the same cell many times on a schematic. To simplify this task, most

editors allow you to place a special **vectored cell instance** of a cell. A vectored cell instance, or **vectored instance** for short, uses the same icon for a single instance but with a special attribute, the **cell cardinality**, that denotes the number of copies of the cell. Connections between signals on a bus and vectored instances should be handled automatically. The width or **cardinality** of the bus and the cell cardinality must match, and the design-entry tool should issue a warning if this is not the case.

A schematic-entry program can use a terminal attribute to determine which cell terminals are output terminals and which terminals are input terminals. This attribute is usually called **terminal polarity** or **terminal direction**. Possible values for terminal polarity might be: `input`, `output`, and `bidirectional`. Checking the terminal polarity of the terminals on a net can help find problems such as a net with all input terminals or all output terminals.

The **fanout** of a cell measures the driving capability of an output terminal. The **fanin** of a cell measures the number of input terminals. Fanout is normally measured using a standard load. A **standard load** is the load presented by one input of a primitive cell, usually a two-input NAND. For example, a library cell `Counter` may have an input terminal, `clock`, that is connected to the input terminals of five primitive cells. The loading at this terminal is then five standard loads. We say that the fanout of `clock` is five. In a similar fashion, we say that if a cell `Buffer` is capable of driving the inputs of three primitive cells, the fanout of `Buffer` is three. Using the fanin and fanout attributes a netlist screener can check to see if the fanout driving a net is greater than the sum of all loads on that net. (See Figure 9.2 on page 329.)

9.1.12 Schematic-Entry Tools

Some editors offer **icon edit-in-place** in a similar fashion as schematic edit-in-place for cells. Often you have to toggle editing modes in the schematic-entry program to switch between editing cells and editing cell icons. A schematic-entry program must keep track of when cells are edited. Normally this is done by using a **timestamp** or **datestamp** for each cell. This is a text field within the data file for each cell that holds the date and time that the cell was last modified. When a new schematic or cell is loaded, the program needs to compare its timestamp with the timestamps of any subcells. If any of the subcell timestamps are more recent, then the designer needs to be alerted. Usually a message appears to inform you that changes have been made to subcells since the last time the cell currently loaded was saved. This may be what you expect or it may be a warning that somehow a subcell has been changed inadvertently (perhaps someone else changed it) since you last loaded that cell.

Normally the primitive cells in a library are locked and cannot be edited. If you can edit a primitive cell, you have to make a copy, edit the copy, and rename it. Normally the ASIC designer cannot do this and does not want to. For example, to edit a primitive NAND gate stored in an ASIC schematic library would require that the

subschematic of the primitive cell be available (usually not the case) and also that the next lower level primitives (symbols for the transistors making up the NAND gate) also be available to the designer (also usually not the case).

What do you do if somehow changes were made to a cell by mistake, perhaps by someone else, and you don't want the new cell, you want the old version? Most schematic-entry and other EDA tools keep old versions of files as a back-up in case this kind of problem occurs. Most EDA software automatically keeps track of the different **versions** of a file by appending a **version number** to each file. Usually this is transparent to the designer. Thus when you edit a cell named `Floor`, the file on disk might be called `Floor.6`. When you save the changes, the software will not overwrite `Floor.6`, but write out a new file and automatically name it `Floor.7`.

Some design-entry tools are more sophisticated and allow users to create their own libraries as they complete an ASIC design. Designers can then control access to libraries and the cells that they build during a design. This normally requires that a schematic editor, for example, be part of a larger EDA system or framework rather than work as a stand-alone tool. Sometimes the process of library control operates as a separate tool, as a **design manager** or **library manager**. Often there is a program similar to the UNIX `make` command that keeps track of all files, their dependencies, and the tools that are necessary to create and update each file.

You can normally set the number of back-up versions of files that EDA software keeps. The **version history** controls the number of files the software will keep. If you accidentally update, overwrite, or delete a file, there is usually an option to select and revert to an earlier version. More advanced systems have **check-out** services (which work just as in source control systems in computer programming databases) that prevent these kinds of problems when many people are working on the same design. Whenever possible, the management of design files and different versions should be left under software control because the process can become very complicated. Reverting to an earlier version of a cell can have drastic consequences for other cells that reference the cell you are working with. Attempts to manually edit files by changing version numbers and timestamps can quickly lead to chaos.

Most schematic-entry programs allow you to **undo** commands. This feature may be restricted to simply undoing the last command that you entered, or may be an unlimited undo and redo, allowing you to back up as many commands as you want in the current editing session.

You can spend a lot of time in a schematic editor placing components and drawing the connections between them. Features that simplify initial entry and allow modifications to be made easily can make an enormous difference to the efficiency of the schematic-entry process.

Most schematic editors allow you to make connections by dragging the cursor with the wire following behind, in a process known as **rubber banding**. The connection snaps to a right angle when the connection is completed. For wire connections that require more than two line segments, an automatic wiring feature is useful. This allows you to define the wire path roughly using mouse clicks and have the editor complete the connection.

It is exceedingly painful to move components if you have to rewire connections each time. Most schematic editors allow you to move the components and drag any wires along with them.

One of the most annoying problems that can arise in schematic entry is to think that you have joined two wires on a schematic but find that in reality they do not quite meet. This error can be almost impossible to find. A good editing program will have a way of avoiding this problem. Some editors provide a visual (flash) or audible (beep) feedback when the designer draws a wire that makes an electrical connection with another. Some editors will also automatically insert a dot at a “T” connection to show that an electrical connection is present. Other editors refuse to allow four-way connections to be made, so there can be no ambiguity when wires cross each other if an electrical connection is present or not.

A cell library or a collection of libraries is a key part of the schematic-entry process. The ability to handle and control these libraries is an important feature of any schematic editor. It should be easy to select components from the library to be placed on a schematic.

In large schematics it is necessary to continue large nets and signals across several pages of schematics. Signals such as power and ground, VDD and GND, can be connected using **global nets** or special **connectors**. Global nets allow the designer to label a net with the same name at different places on a schematic page or on different pages without having to draw a connection explicitly. The schematic editor treats these nets as though they were electrically connected. Special connector symbols can be used for connections that cross schematic pages. An **off-page connector** or **multipage connector** is a special symbol that will show and label a connection to different schematic pages. More sophisticated editors can automatically label these connectors with the page numbers of the destination connectors.

9.1.13 Back-Annotation

After you enter a schematic you simulate the design to make sure it works as expected. This completes the logical design. Next you move to ASIC physical design and complete the layout. Only after you complete the layout do you know the parasitic capacitance and therefore the delay associated with the interconnect. This postroute delay information must be returned to the schematic in a process known as **back-annotation**. Then you can complete a final, postlayout simulation to make sure that the specifications for the ASIC are met. Chapter 13 covers simulation, and the physical design steps are covered in Chapters 15 to 17.

9.2 Low-Level Design Languages

Schematics can be a very effective way to convey design information because pictures are such a powerful medium. There are two major problems with schematic entry, however. The first problem is that making changes to a schematic can be diffi-

cult. When you need to include an extra few gates in the middle of a schematic sheet, you may have to redraw the whole sheet. The second problem is that for many years there were no standards on how symbols should be drawn or how the schematic information should be stored in a netlist. These problems led to the development of design-entry tools based on text rather than graphics. As TTL gave way to PLDs, these text-based design tools became increasingly popular as de facto standards began to emerge for the format of the design files.

PLDs are closely related to FPGAs. The major advantage of PLD tools is their low cost, their ease of use, and the tremendous amount of knowledge and number of designs, application notes, textbooks, and examples that have been built up over years of their use. It is natural then that designers would want to use PLD development systems and languages to design FPGAs and other ASICs. For example, there is a tremendous amount of PLD design expertise and working designs that can be reused.

In the case of ASIC design it is important to use the right tool for the job. This may mean that you need to convert from a low-level design medium you have used for PLD design to one more appropriate for ASIC design. Often this is because you are merging several PLDs into a single, much larger, ASIC. The reason for covering the PLD design languages here is not to try and teach you how to use them, but to allow you to read and understand a PLD language and, if necessary, convert it to a form that you can use in another ASIC design system.

9.2.1 ABEL

ABEL is a PLD programming language from Data I/O. Table 9.2 shows some examples of the ABEL statements. The following example code describes a 4:1 MUX (equivalent to the LS153 TTL part):

```

module MUX4
title '4:1 MUX'
MyDevice device 'P16L8' ;
@ALTERNATE
"inputs
A, B, /P1G1, /P1G2 pin 17,18,1,6 "LS153 pins 14,2,1,15
P1C0, P1C1, P1C2, P1C3 pin 2,3,4,5 "LS153 pins 6,5,4,3
P2C0, P2C1, P2C2, P2C3 pin 7,8,9,11 "LS153 pins 10,11,12,13
"outputs
P1Y, P2Y pin 19, 12 "LS153 pins 7,9
equations
    P1Y = P1G*(/B*/A*P1C0 + /B*A*P1C1 + B*/A*P1C2 + B*A*P1C3);
    P1Y = P1G*(/B*/A*P1C0 + /B*A*P1C1 + B*/A*P1C2 + B*A*P1C3);
end MUX4

```

TABLE 9.2 ABEL.

Statement	Example	Comment
Module	<code>module MyModule</code>	You can have multiple modules.
Title	<code>title 'Title in a String'</code>	A string is a character series between quotes.
Device	<code>MYDEV device '22V10' ;</code>	MYDEV is Device ID for documentation. 22V10 is checked by the compiler.
Comment	<code>"comments go between double quotes"</code> <code>"end of line is end of comment"</code>	The end of a line signifies the end of a comment; there is no need for an end quote.
@ALTERNATE	<code>@ALTERNATE "use alternate symbols"</code>	operator alternate default
		AND * &
		OR + #
		NOT / !
		XOR :+ \$
		XNOR *: !&
Pin declaration	<code>MYINPUT pin 2; I3, I4 pin 3, 4 ;</code> <code>/MYOUTPUT pin 22; IO3,IO4 pin 21,20 ;</code>	Pin 22 is the IO for input on pin 2 for a 22V10. MYOUTPUT is active-low at the chip pin. Signal names must start with a letter.
Equations	<code>equations</code> <code>IO4 = HELPER ; HELPER = /I4 ;</code>	Defines combinational logic. Two-pass logic
Assignments	<code>MYOUTPUT = /MYINPUT ;</code> <code>IO3 := I4 ;</code>	Equals '=' is unlocked assignment. Clocked assignment operator (registered IO)
Signal sets	<code>D = [D0, D1, D2, D3] ;</code> <code>Q = [Q0, Q1, Q2, Q3];</code>	A signal set, an ABEL bus
	<code>Q := D ;</code>	4-bit-wide register
Suffix	<code>MYOUTPUT.RE = CLR ;</code> <code>MYOUTPUT.PR = PRE ;</code>	Register reset Register preset
Addition	<code>COUNT = [D0, D1, D2];</code> <code>COUNT := COUNT + 1;</code>	Can't use @ALTERNATE if you use '+' to add.
Enable	<code>ENABLE IO3 = IO2;</code> <code>IO3 = MYINPUT;</code>	Three-state enable (ENABLE is a keyword). IO3 must be a three-state pin.
Constants	<code>K = [1, 0, 1] ;</code>	K is 5.
Relational	<code>IO# = D == K5 ;</code>	Operators: == != < > <= >=
End	<code>end MyModule</code>	Last statement in module

9.2.2 CUPL

CUPL is a PLD design language from Logical Devices. We shall review the CUPL 4.0 language here. The following code is a simple CUPL example describing sequential logic:

```
SEQUENCE BayBridgeTollPlaza {
  PRESENT red
    IF car NEXT green OUT go;      /* conditional synchronous output */
    DEFAULT NEXT red;              /* default next state */
  PRESENT green
    NEXT red; }                   /* unconditional next state */
```

This code describes a state machine with two states. Table 9.3 shows the different state machine assignment statements.

TABLE 9.3 CUPL statements for state-machine entry.

Statement			Description
IF	NEXT		Conditional next state transition
IF	NEXT	OUT	Conditional next state transition with synchronous output
	NEXT		Unconditional next state transition
	NEXT	OUT	Unconditional next state transition with asynchronous output
		OUT	Unconditional asynchronous output
IF		OUT	Conditional asynchronous output
DEFAULT	NEXT		Default next state transition
DEFAULT		OUT	Default asynchronous output
DEFAULT	NEXT	OUT	Default next state transition with synchronous output

You may also encode state machines as truth tables in CUPL. Here is another simple example:

```
FIELD input = [in1..0];
FIELD output = [out3..0];
TABLE input => output {00 => 01; 01 => 02; 10 => 04; 11 => 08; }
```

The advantage of the CUPL language, and text-based PLD languages in general, is now apparent. First, we do not have to enter the detailed logic for the state decoding ourselves—the software does it for us. Second, to make changes only requires simple text editing—fast and convenient.

Table 9.4 shows some examples of CUPL statements. In CUPL Boolean equations may use variables that contain a suffix, or an **extension**, as in the following example:

```
output.ext = (Boolean expression);
```

TABLE 9.4 CUPL.

Statement	Example	Comment
Boolean expression	A = !B;	Logical negation
	A = B & C;	Logical AND
	A = B # C;	Logical OR
	A = B \$ C;	Logical exclusive-OR
Comment	A = B & C /* comment */	
Pin declaration	PIN 1 = CLK;	Device dependent
	PIN = CLK;	Device independent
Node declaration	NODE A;	Number automatically assigned
	NODE [B0..7];	Array of buried nodes
Pinnode declaration	PINNODE 99 = A;	Node assigned by designer
	PINNODE [10..17] = [B0..7];	Array of pinnodes
Bit-field declaration	FIELD Address = [B0..7];	8-bit address field
Bit-field operations	add_one = Address:FF;	True if Address = 0xFF
	add_zero = !(Address:&);	True if Address = 0x00
	add_range = Address:[0F..FF];	True if 0F.LE.Address.LE.FF

The extensions steer the software, known as a **fitter**, in assigning the logic. For example, a signal-name suffix of .OE marks that signal as an output enable.

Here is an example of a CUPL file for a 4-bit counter placed in an ATMEL PLD part that illustrates the use of some common extensions:

```
Name 4BIT; Device V2500B;
/* inputs */
pin 1 = CLK; pin 3 = LD_; pin 17 = RST_;
pin [18,19,20,21] = [I0,I1,I2,I3];
/* outputs */
pin [4,5,6,7] = [Q0,Q1,Q2,Q3];
field CNT = [Q3,Q2,Q1,Q0];
/* equations */
Q3.T = (!Q2 & !Q1 & !Q0) & LD_ & RST_ /* count down */
```

```

# Q3 & !RST_ /* ReSeT */
# (Q3 $ I3) & !LD_ /* Load*/
Q2.T = (!Q1 & !Q0) & LD_ & RST_ # Q2 & !RST_ # (Q2 $ I2) & !LD_ ;
Q1.T = !Q0 & LD_ & RST_ # Q1 & !RST_ # (Q1 $ I1) & !LD_ ;
Q0.T = LD_ & RST_ # Q0 & !RST_ # (Q0 $ I0) & !LD_ ;
CNT.CK = CLK; CNT.OE = 'h'F; CNT.AR = 'h'0; CNT.SP = 'h'0;

```

In this example the suffix extensions have the following effects: `.CK` marks the clock; `.T` configures sequential logic as T flip-flops; `.OE` (wired high) is the output enable; `.AR` (wired low) is the asynchronous reset; and `.SP` (wired low) is the synchronous preset. Table 9.5 shows the different CUPL extensions.

The 4-bit counter is a very simple example of the use of the Atmel ATV2500B. This PLD is quite complex and has many extra “buried” features. In order to use these features in CUPL (and ABEL) you need to refer to special pin numbers and node numbers that are given in tables in the manufacturer’s data sheets. You may need the pin-number tables to reverse engineer or convert a complicated CUPL (or ABEL) design from one format to another.

Atmel also gives skeleton headers and pin declarations for their parts in their data sheets. Table 9.6 shows the headers and pin declarations in ABEL and CUPL format for the ATMEL ATV2500B.

9.2.3 PALASM

PALASM is a PLD design language from AMD/MMI. Table 9.7 shows the format of PALASM statements. The following simple example (a video shift register) shows the most basic features of the PALASM 2 language:

```

TITLE video ; shift register
CHIP video PAL20X8
CK /LD D0 D1 D2 D3 D4 D5 D6 D7 CURS GND NC REV Q7 Q6 Q5 Q4 Q3 Q2 Q1 Q0
/RST VCC
STRING Load 'LD*/REV*/CURS*RST' ; load data
STRING LoadInv 'LD*REV*/CURS*RST' ; load inverted of data
STRING Shift '/LD*/CURS*/RST' ; shift data from MSB to LSB
EQUATIONS
/Q0 := /D0*Load+D0*LoadInv:+/Q1*Shift+RST
/Q1 := /D1*Load+D1*LoadInv:+/Q2*Shift+RST
/Q2 := /D2*Load+D2*LoadInv:+/Q3*Shift+RST
/Q3 := /D3*Load+D3*LoadInv:+/Q4*Shift+RST
/Q4 := /D4*Load+D4*LoadInv:+/Q5*Shift+RST
/Q5 := /D5*Load+D5*LoadInv:+/Q6*Shift+RST
/Q6 := /D6*Load+D6*LoadInv:+/Q7*Shift+RST
/Q7 := /D7*Load+D7*LoadInv:+:Shift+RST;

```

TABLE 9.5 CUPL 4.0 extensions.

Extension ¹	Explanation	Extension	Explanation
D	L D input to a D register	DFB	R D register feedback of combinational output
L	L L input to a latch	LFB	R Latched feedback of combinational output
J, K	L J-K-input to a J-K register	TFB	R T register feedback of combinational output
S, R	L S-R input to an S-R register	INT	R Internal feedback
T	L T input to a T register	IO	R Pin feedback of registered output
DQ	R D output of an input D register	IOD/T	R D/T register on pin feedback path selection
LQ	R Q output of an input latch	IOL	R Latch on pin feedback path selection
AP, AR	L Asynchronous preset/reset	IOAP, IOAR	L Asynchronous preset/reset of register on feedback path
SP, SR	L Synchronous preset/reset	IOSP, IOSR	L Synchronous preset/reset of register on feedback path
CK	L Product clock term (async.)	IOCK	L Clock for pin feedback register
OE	L Product-term output enable	APMUX, ARMUX	L Asynchronous preset/reset multiplexor selection
CA	L Complement array	CKMUX	L Clock multiplexor selector
PR	L Programmable preload	LEMUX	L Latch enable multiplexor selector
CE	L CE input of a D-CE register	OEMUX	L Output enable multiplexor selector
LE	L Product-term latch enable	IMUX	L Input multiplexor selector of two pins
OBS	L Programmable observability of buried nodes	TEC	L Technology-dependent fuse selection
BYP	L Programmable register bypass	T1	L T1 input of 2-T register

¹ L means that the extension is used only on the LHS of an equation; R means that the extension is used only on the RHS of an equation.

TABLE 9.6 ABEL and CUPL pin declarations for an ATMEL ATV2500B.

ABEL	CUPL
<pre> device_id device 'P2500B'; "device_id used for JEDEC filename I1,I2,I3,I17,I18 pin 1,2,3,17,18; O4,O5 pin 4,5 istype 'reg_d,buffer'; O6,O7 pin 6,7 istype 'com'; O4Q2,O7Q2 node 41,44 istype 'reg_d'; O6F2 node 43 istype 'com'; O7Q1 node 220 istype 'reg_d'; </pre>	<pre> device V2500B; pin [1,2,3,17,18] = [I1,I2,I3,I17,I18]; pin [7,6,5,4] = [O7,O6,O5,O4]; pinnode [41,65,44] = [O4Q2,O4Q1,O7Q2]; pinnode [43,68] = [O6Q2,O7Q1]; </pre>

TABLE 9.7 PALASM 2.

Statement	Example	Comment
Chip	<pre> CHIP abc 22V10 CHIP xyz USER </pre>	<p>Specific PAL type</p> <p>Free-form equation entry</p>
Pinlist	<pre> CLK /LD D0 D1 D2 D3 D4 GND NC Q4 Q3 Q2 Q1 Q0 /RST VCC </pre>	Part of CHIP statement; PAL pins in numerical order starting with pin 1
String	<pre> STRING string_name 'text' </pre>	Before EQUATIONS statement
Equations	<pre> EQUATIONS A = /B A = B * C A = B + C A = B :+: C A = B **: C </pre>	<p>After CHIP statement</p> <p>Logical negation</p> <p>Logical AND</p> <p>Logical OR</p> <p>Logical exclusive-OR</p> <p>Logical exclusive-NOR</p>
Polarity inversion	<pre> /A = /(B + C) </pre>	Same as $A = B + C$
Assignment	<pre> A = B + C A := B + C </pre>	<p>Combinational assignment</p> <p>Registered assignment</p>
Comment	<pre> A = B + C ; comment </pre>	Comment
Functional equation	<pre> name.TRST name.CLKF name.RSTF name.SETF </pre>	<p>Output enable control</p> <p>Register clock control</p> <p>Register reset control</p> <p>Register set control</p>

The order of the pin numbers in the previous example is important; the order must correspond to the order of pins for the DEVICE. This means that you probably need the device data sheet in order to be able to translate a design from PALASM to another format by hand. The alternative is to use utilities that many PLD and FPGA companies offer that automatically translate from PALASM to their own formats.

9.3 PLA Tools

We shall use the Berkeley PLA tools to illustrate logic minimization using an example to minimize the logic required to implement the following three logic functions:

$$F1 = A|B|!C; \quad F2 = !B\&C; \quad F3 = A\&B|C;$$

These equations are in eqntott input format. The eqntott (for “equation to truth table”) program converts the input equations into a tabular format. Table 9.8 shows the truth table and eqntott output for functions F1, F2, and F3 that use the six minterms: A, B, !C, !B&C, A&B, C.

TABLE 9.8 A PLA tools example.

Input (6 minterms): F1 = A|B|!C; F2 = !B&C; F3 = A&B|C;

A	B	C	F1	F2	F3	eqntott output	espresso output
0	0	0	1	0	0	.i 3	.i 3
0	0	1	0	1	1	.o 3	.o 3
0	1	0	1	0	0	.p 6	.p 6
0	1	1	1	0	1	--0 100	1-- 100
1	0	0	1	0	0	--1 001	11- 001
1	0	1	1	1	1	-01 010	--0 100
1	1	0	1	0	1	-1- 100	-01 011
1	1	1	1	1	1	1-- 100	-11 101
1	1	0	1	0	1	11- 001	.e
1	1	1	1	0	1	.e	

Output (5 minterms): F1 = A|!C|(B&C); F2 = !B&C; F3 = A&B|(!B&C)|(B&C);

This `eqntott` output is not really a truth table since each line corresponds to a min-term. The output forms the input to the `espresso` logic-minimization program. Table 9.9 shows the format for `espresso` input and output files. Table 9.10 explains the format of the input and output planes of the `espresso` input and output files. The `espresso` output in Table 9.8 corresponds to the `eqntott` logic equations on the next page.

TABLE 9.9 The format of the input and output files used by the PLA design tool `espresso`.

Expression	Explanation
# comment	# must be first character on a line.
[d]	Decimal number
[s]	Character string
.i [d]	Number of input variables
.o [d]	Number of output variables
.p [d]	Number of product terms
.ilb [s1] [s2]... [sn]	Names of the binary-valued variables must be after .i and .o.
.ob [s1] [s2]... [sn]	Names of the output functions must be after .i and .o.
.type f	Following table describes the ON set; DC set is empty.
.type fd	Following table describes the ON set and DC set.
.type fr	Following table describes the ON set and OFF set.
.type fdr	Following table describes the ON set, OFF set, and DC set.
.e	Optional, marks the end of the PLA description.

TABLE 9.10 The format of the plane part of the input and output files for `espresso`.

Plane	Character	Explanation
I	1	The input literal appears in the product term.
I	0	The input literal appears complemented in the product term.
I	-	The input literal does not appear in the product term.
O	1 or 4	This product term appears in the ON set.
O	0	This product term appears in the OFF set.
O	2 or -	This product term appears in the don't care set.
O	3 or ~	No meaning for the value of this function.

$F1 = A|!C|(B\&C); \quad F2 = !B\&C; \quad F3 = A\&B|(!B\&C)|(B\&C);$

We see that espresso reduced the original six minterms to these five: A, A&B, !C, !B&C, B&C.

The Berkeley PLA tools were widely used in the 1980s. They were important stepping stones to modern logic synthesis tools. There are so many testbenches, examples, and old designs that used these tools that we occasionally need to convert files in the Berkeley PLA format to formats used in new tools.

9.4 EDIF

An ASIC designer spends an increasing amount of time forcing different tools to communicate. One standard for exchanging information between EDA tools is the **electronic design interchange format (EDIF)**. We will describe EDIF version 2 0 0. The most important features added in EDIF 3 0 0 were to handle buses, bus rippers, and buses across schematic pages. EDIF 4 0 0 includes new extensions for PCB and multichip module (MCM) data. The **Library of Parameterized Modules (LPM)** standard is also based on EDIF. The newer versions of EDIF have a richer feature set, but the ASIC industry seems to have standardized on EDIF 2 0 0. Most EDA companies now support EDIF. The FPGA companies Altera and Actel use EDIF as their netlist format, and Xilinx has announced its intention to switch from its own XNF format to EDIF. We only have room for a brief description of the EDIF format here. A complete description of the EDIF standard is contained in the **Electronic Industries Association (EIA)** publication, *Electronic Design Interchange Format Version 2 0 0 (ANSI/EIA Standard 548-1988)* [EDIF, 1988].

9.4.1 EDIF Syntax

The structure of EDIF is similar to the Lisp programming language or the Postscript printer language. This makes EDIF a very hard language to read and almost impossible to write by hand. EDIF is intended as an exchange format between tools, not as a design-entry language. Since EDIF is so flexible each company reads and writes different “flavors” of EDIF. Inevitably EDIF from one company does not quite work when we try and use it with a tool from another company, though this situation is improving with the gradual adoption of EDIF 3 0 0. We need to know just enough about EDIF to be able to fix these problems.

Figure 9.8 illustrates the hierarchy of the EDIF file. Within an EDIF file are one or more libraries of cell descriptions. Each library contains technology information that is used in describing the characteristics of the cells it contains. Each cell description contains one or more user-named views of the cell. Each view is defined as a particular `viewType` and contains an `interface` description that identifies where the cell may be connected to and, possibly, a `contents` description that identifies the components and related interconnections that make up the cell.

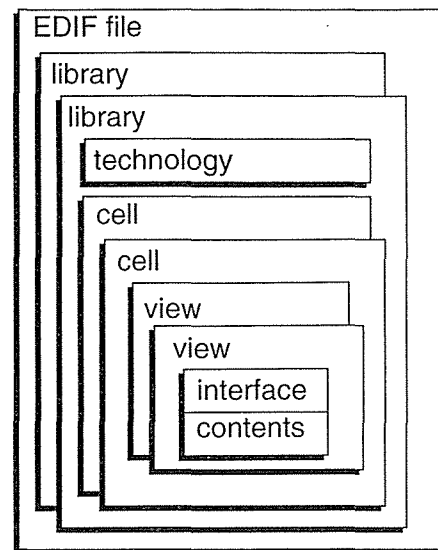


FIGURE 9.8 The hierarchical nature of an EDIF file.

The EDIF syntax consists of a series of statements in the following format:

```
(keywordName {form})
```

A left parenthesis (round bracket) is always followed by a **keyword name**, followed by one or more **EDIF forms** (a form is a sequence of identifiers, primitive data, symbolic constants, or EDIF statements), ending with a right parenthesis. If you have programmed in Lisp or Postscript, you may understand that EDIF uses a “define it before you use it” approach and why there are so many parentheses in an EDIF file.

The semantics of EDIF are defined by the **EDIF keywords**. Keywords are the only types of name that can immediately follow a left parenthesis. Case is not significant in keywords.

An **EDIF identifier** represents the name of an object or group of data. Identifiers are used for name definition, name reference, keywords, and symbolic constants. Valid EDIF identifiers consist of alphanumeric or underscore characters and must be preceded by an ampersand (&) if the first character is not alphabetic. The ampersand is not considered part of the name. The length of an identifier is from 1 to 255 characters and case is not significant. Thus &clock, Clock, and clock all represent the same EDIF name (very confusing).

Numbers in EDIF are 32-bit signed integers. Real numbers use a special EDIF format. For example, the real number 1.4 is represented as (e 14 -1). The e form requires a mantissa (14) and an exponent (-1). Reals are restricted to the range $\pm 1 \times 10^{\pm 35}$. Numbers in EDIF are dimensionless and the units are determined according to where the number occurs in the file. Coordinates and line widths are units of distance and must be related to meters. Each coordinate value is converted

to meters by applying a **scale factor**. Each EDIF library has a **technology** section that contains a required **numberDefinition**. The **scale** keyword is used with the **numberDefinition** to relate EDIF numbers to physical units.

Valid EDIF strings consist of sequences of ASCII characters enclosed in double quotes. Any alphanumeric character is allowed as well as any of the following characters: ! # \$ % & ' () * + , - . / : ; < = > ? @ [\] ^ _ ` { | } ~. Special characters, such as " and % are entered as escape sequences: %number%, where number is the integer value of the ASCII character. For example, "A quote is % 34 %" is a string with an embedded double-quote character. Blank, tab, line feed, and carriage-return characters (white space) are used as delimiters in EDIF. Blank and tab characters are also significant when they appear in strings.

The **rename** keyword can be used to create a new EDIF identifier as follows:

```
(cell (rename TEST_1 "test$1") ...
```

In this example the EDIF string contains the original name, `test$1`, and a new name, `TEST_1`, is created as an EDIF identifier.

9.4.2 An EDIF Netlist Example

Table 9.11 shows an EDIF netlist. This EDIF description corresponds to the halfgate example in Chapter 8 and describes an inverter. We shall explain the functions of the EDIF in Table 9.11 by showing a piece of the code at a time followed by an explanation.

```
(edif halfgate_p
  (edifVersion 2 0 0) (edifLevel 0) (keywordMap (keywordLevel 0))
  (status (written (timeStamp 1996 7 10 22 5 10)
  (program "COMPASS Design Automation -- EDIF Interface"
  (version "v9r1.2 last updated 26-Mar-96")) (author "mikes")))
```

Every EDIF file must have an **edif** form. The **edif** form must have a name, an **edifVersion**, an **edifLevel**, and a **keywordMap**. The **edifVersion** consists of three integers describing the **major** (first number) and **minor version** of EDIF. The **keywordMap** must have a **keywordLevel**. The optional **status** can contain a written form that must have a **timeStamp** and, optionally, **author** or **program** forms.

```
(library xc4000d (edifLevel 0) (technology
```

(The unbalanced parentheses are deliberate since we are showing segments of the EDIF code.) The **library** form must have a name, **edifLevel** and **technology**. The **edifLevel** is normally 0. The `xc4000d` library contains the cells we are using in our schematic.

TABLE 9.11 EDIF file for the halfgate netlist from Chapter 8.

```

(edif halfgate_p                (viewType NETLIST)                (contents
(edifVersion 2 0 0)            (interface                (instance B1_i1
(edifLevel 0)                  (port I                (viewRef
(keywordMap                    (direction INPUT))    COMPASS_mde_view
(keywordLevel 0))              (port O                (cellRef INV
(status                        (direction OUTPUT))    (libraryRef
(written                       (designator "@@Label")))) xc4000d)))
(timeStamp 1996 7 10 22      (library working      (net myInput
5 10)                          (edifLevel 0)        (joined
(program "COMPASS Design      (technology          (portRef myInput)
Automation -- EDIF Interface" (numberDefinition ) (portRef I
(version "v9r1.2 last        (simulationInfo      (instanceRef
updated 26-Mar-96"))         (logicValue H)       B1_i1))))
(author "mikes"))            (logicValue L))      (net myOutput
(library xc4000d              (cell                (joined
(edifLevel 0)                 (rename HALFGATE_P   (portRef myOutput)
(technology                    "halfgate_p")        (portRef O
(numberDefinition )           (cellType GENERIC)  (instanceRef
(simulationInfo               (view COMPASS_nls_view B1_i1))))
(logicValue H)                (viewType NETLIST)  (net VDD
(logicValue L)))              (interface            (joined ))
(cell                          (port myInput       (net VSS
(rename INV "inv")            (direction INPUT))  (joined ))))
(cellType GENERIC)           (port myOutput      (design HALFGATE_P
(view COMPASS_mde_view        (direction OUTPUT)) (cellRef HALFGATE_P
                                (designator "@@Label")) (libraryRef working)))

```

```
(numberDefinition ) (simulationInfo (logicValue H) (logicValue L))
```

The simulationInfo form is used by simulation tools; we do not need that information for netlist purposes for this cell. We shall discuss numberDefinition in the next example. It is not needed in a netlist.

```
(cell (rename INV "inv") (cellType GENERIC)
```

This cell form defines the name and type of a cell inv that we are going to use in the schematic.

```
(view COMPASS_mde_view (viewType NETLIST)
(interface (port I (direction INPUT)) (port O (direction OUTPUT))
(designator "@@Label"))))
```

The NETLIST view of this inverter cell has an input port I and an output port O. There is also a place holder "@@Label" for the instance name of the cell.

```
(library working...
```

This begins the description of our schematic that is in our library working. The lines that follow this library form are similar to the preamble for the cell library xc4000d that we just explained.

```
(cell (rename HALFGATE_P "halfgate_p")(cellType GENERIC)
  (view COMPASS_nls_view (viewType NETLIST)
```

This cell form is for our schematic named halfgate_p.

```
(interface (port myInput (direction INPUT))
  (port myOutput (direction OUTPUT))
```

The interface form defines the names of the ports that were used in our schematic, myInput and myOutput. At this point we have not associated these ports with the ports of the cell INV in the cell library.

```
(designator "@@Label")) (contents (instance B1_i1
```

This gives an instance name B1_i1 to the cell in our schematic.

```
(viewRef COMPASS_mde_view (cellRef INV (libraryRef xc4000d))))
```

The cellRef form links the cell instance name B1_i1 in our schematic to the cell INV in the library xc4000d.

```
(net myInput (joined (portRef myInput)
  (portRef I (instanceRef B1_i1))))
```

The net form for myInput (and the one that follows it for myOutput) ties the net names in our schematic to the ports I and O of the library cell INV.

```
(net VDD (joined )) (net VSS (joined ))))))
```

These forms for the global VDD and VSS nets are often handled differently by different tools (one company might call the negative supply GND instead of VSS, for example). This section is where you most often have to edit the EDIF.

```
(design HALFGATE_P (cellRef HALFGATE_P (libraryRef working))))
```

The design form names and places our design in library working, and completes the EDIF description.

9.4.3 An EDIF Schematic Icon

EDIF is capable of handling many different representations. The next EDIF example is another view of an inverter that describes how to draw the icon (the picture that appears on the printed schematic or on the screen) shown in Figure 9.9. We shall

examine the EDIF created by the CAD/CAM Group's Engineering Capture System (ECS) schematic editor.

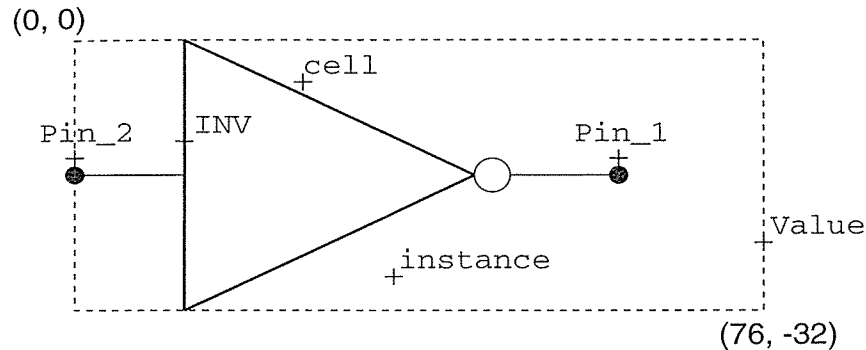


FIGURE 9.9 An EDIF view of an inverter icon. The coordinates shown are in EDIF units. The crosses that show the text location origins and the dotted bounding box do not print as part of the icon.

This time we shall give more detailed explanations after each piece of EDIF code. We shall also maintain balanced parentheses to make the structure easier to follow. To shorten the often lengthy EDIF code, we shall use an ellipsis (...) to indicate any code that has been left out.

```
(edif ECS
  (edifVersion 2 0 0)
  (edifLevel 0)
  (keywordMap (keywordLevel 0))
  (status
    (written
      (timeStamp 1987 8 20 0 50 23)
      (program "CAD/CAM Group, Inc. ECS" (Version "1"))))
  (library USER ...)
  )
  ...
)
```

This preamble is virtually identical to the previous netlist example (and demonstrates that EDIF is useful to store design information as software tools come and go over many years). The first line of the file defines the name of the file. This is followed by lines that identify the version of EDIF being used and the highest EDIF level used in the file (each library may use its own level up to this maximum). EDIF level 0 supports only literal constants and basic constructs. Higher EDIF levels support parameters, expressions, and flow control constructs. EDIF keywords may be

mapped to aliases, and keyword macros may be defined within the keywordMap form. These features are not often used in ASIC design because of a lack of standardization. The keywordLevel 0 indicates these capabilities are not used here. The status construct is used for administration: when the file was created, the software used to create the file, and so on. Following this preamble is the main section of the file, which contains design information.

```
(library USER (edifLevel 0)
  (technology
    (numberDefinition
      (scale 4 (e 254 -5) (unit distance)))
    (figureGroup NORMAL
      (pathWidth 0) (borderWidth 0)
      (textHeight 5))
    (figureGroup WIDE
      (pathWidth 1) (borderWidth 1)
      (textHeight 5)))
  (cell 7404 ...
  )
)
```

The technology form has a numberDefinition that defines the scaling information (we did not use this form for a netlist, but the form must be present). The first numberValue after scale represents EDIF numbers and the second numberValue represents the units specified by the unit form. The EDIF unit for distance is the meter. The numberValue can be an integer or an exponential number. The e form has a mantissa and an exponent. In this example, within the USER library, a distance of 4 EDIF units equals 254×10^{-5} meters (or 4 EDIF units equals 0.1 inch).

After the numberDefinition in the technology form there are one or more figureGroup definitions. A figureGroup defines drawing information such as pathWidth, borderWidth, color, fillPattern, borderPattern, and textHeight. The figureGroup form must have a name, which will be used later in the library to refer back to these definitions. In this example the USER library has one figureGroup (NORMAL) for lines and paths of zero width (the actual width will be implementation dependent) and another figureGroup (WIDE) that will be used for buses with a wider width (for bold lines). The borderWidth is used for drawing filled areas such as rectangles, circles, and polygons. The pathwidth is used for open figures such as lines (paths) and open arcs.

Following the technology section the cell forms each represent a symbol. The cell form has a name that will appear in the names of any files produced. The cellType form GENERIC type is required by this schematic editor. The property form is used to list properties of the cell.

```
(cell 7404 (cellType GENERIC)
  (property SymbolType (string "GATE"))
  (view PCB_Symbol (viewType SCHEMATIC))
```

```

        (interface ...
        )
    )
)

```

The `SymbolType` property is used to distinguish between purely graphical symbols that do not occur in the parts list (a ground connection, for example), gate or component symbols, and block or cell symbols (for hierarchical schematics). The `SymbolType` property is a string that may be `COMPONENT`, `GATE`, `CELL`, `BLOCK`, or `GRAPHIC`. Each cell may contain view forms and each view must have a name. Following the name of the view must be a `viewType` that is either `GRAPHIC` or `SCHEMATIC`. Following the `viewType` is the interface form, which contains the symbol and terminal information. The interface form contains the actual symbol data.

```

(interface
  (port Pin_1
    (designator "2")
    (direction OUTPUT)
    (dcMaxFanout 50))
  (port Pin_2
    (designator "1")
    (direction INPUT)
    (dcFanoutLoad 8)
    (property Cap
      (string "22")))
  (property Value
    (string "45"))
  (symbol ...
  )
)

```

If the symbol has terminals, they are listed before the symbol form. The port form defines each terminal. The required port name is used later in the symbol form to refer back to the port. Since this example is from a PCB design, the terminals have pin numbers that correspond to the IC package leads. The pin numbers are defined in the `designator` form with the pin number as a string. The polarity of the pin is indicated by the `direction` form, which may be `INPUT`, `OUTPUT`, or `INOUT`. If the pin is an output pin, its Drive can be represented by `dcMaxFanout` and if it is an input pin its Load can be represented by `dcFanoutLoad`. The port form can also contain forms unused, `dcMaxFanin`, `dcFaninLoad`, `acLoad`, and `portDelay`. All other attributes for pins besides `PinNumber`, `Polarity`, `Load`, and `Drive` are contained in the `property` form.

An attribute string follows the name of the property in the `string` form. In this example port `Pin_2` has a property `Cap` whose value is 22. This is the input capacitance of the inverter, but the interpretation and use of this value depends on the tools. In ASIC design pins do not have pin numbers, so `designator` is not used.

Instead, the pin names use the property form. So (property NetName (string "1")) would replace the (designator "1") in this example on Pin_2. The interface form may also contain attributes of the symbol.

Symbol attributes are similar to pin attributes. In this example the property name Value has an attribute string "45". The names occurring in the property form may be referenced later in the interface under the symbol form to refer back to the property.

```
(symbol
  (boundingBox (rectangle (pt 0 0) (pt 76 -32)))
  (portImplementation Pin_1
    (connectLocation (figure NORMAL (dot (pt 60 -16))))
    (keywordDisplay designator
      (display NORMAL
        (justify LOWERCENTER) (origin (pt 60 -14))))
  (portImplementation Pin_2
    (connectLocation (figure NORMAL (dot (pt 0 -16))))
    (keywordDisplay designator
      (display NORMAL
        (justify LOWERCENTER) (origin (pt 0 -14))))
  (keywordDisplay cell
    (display NORMAL (justify CENTERLEFT) (origin (pt 25 -5))))
  (keywordDisplay instance
    (display NORMAL
      (justify CENTERLEFT) (origin (pt 36 -28))))
  (keywordDisplay designator
    (display (figureGroupOverride NORMAL (textHeight 7))
      (justify CENTERLEFT) (origin (pt 13 -16))))
  (propertyDisplay Value
    (display (figureGroupOverride NORMAL (textHeight 9))
      (justify CENTERRIGHT) (origin (pt 76 -24))))
  (figure ... )
)
```

The interface contains a symbol that contains the pin locations and graphical information about the icon. The optional boundingBox form encloses all the graphical data. The x- and y-locations of two opposite corners of the bounding rectangle use the pt form. The scale section of the numberDefinition from the technology section of the library determines the units of these coordinates. The pt construct is used to specify coordinate locations in EDIF. The keyword pt must be followed by the x-location and the y-location. For example: (pt 100 200) is at x=100, y=200.

- Each pin in the symbol is given a location using a portImplementation.
- The portImplementation refers back to the port defined in the interface.
- The connectLocation defines the point to connect to the pin.

- The connectLocation is specified as a figure, a dot with a single pt for its location.

```
(symbol
  ( ...
    (figure WIDE
      (path (pointList (pt 12 0) (pt 12 -32)))
      (path (pointList (pt 12 -32) (pt 44 -16)))
      (path (pointList (pt 12 0) (pt 44 -16))))
    (figure NORMAL
      (path (pointList (pt 48 -16) (pt 60 -16)))
      (circle (pt 44 -16) (pt 48 -16))
      (path (pointList (pt 0 -16) (pt 12 -16))))
    (annotate
      (stringDisplay "INV"
        (display NORMAL
          (justify CENTERLEFT) (origin (pt 12 -12))))))
  )
```

The figure form has either a name, previously defined as a figureGroup in the technology section, or a figureGroupOverride form. The figure has all the attributes (pathWidth, borderWidth, and so on) that were defined in the figureGroup unless they are specifically overridden with a figureGroupOverride.

Other objects that may appear in a figure are: circle, openShape, path, polygon, rectangle, and shape. Most schematic editors use a grid, and the pins are only allowed to occur **on grid**.

A portImplementation can contain a keywordDisplay or a propertyDisplay for the location to display the pin number or pin name. For a GATE or COMPONENT, keywordDisplay will display the designator (pin number), and designator is the only keyword that can be displayed. For a BLOCK or CELL, propertyDisplay will display the NetName. The display form displays text in the same way that the figure displays graphics. The display must have either a name previously defined as a figureGroup in the technology section or a figureGroupOverride form. The display will have all the attributes (textHeight for example) defined in the figureGroup unless they are overridden with a figureGroupOverride.

A **symbolic constant** is an EDIF name with a predefined meaning. For example, LOWERLEFT is used to specify text justification. The display form can contain a justify to override the default LOWERLEFT. The display can also contain an orientation that overrides the default R0 (zero rotation). The choices for orientation are rotations (R0, R90, R180, R270), mirror about axis (MX, MY), and mirror with rotation (MXR90, MYR90). The display can contain an origin to override the default (pt 0 0).

The symbol itself can have either keywordDisplay or propertyDisplay forms such as the ones in the portImplementation. The choices for keywordDisplay are: cell for attribute Type, instance for attribute InstName, and designator for

attribute `RefDes`. In the preceding example an attribute window currently mapped to attribute `Value` is displayed at location (76, -24) using right-justified text, and a font size is set with (`textHeight 9`).

The graphical data in the symbol are contained in figure forms. The path form must contain `pointList` with two or more points. The figure may also contain a `rectangle` or `circle`. Two points in a `rectangle` define the opposite corners. Two points in a `circle` represent opposite ends of the diameter. In this example a figure from `figureGroup WIDE` has three lines representing the triangle of the inverter symbol.

Arcs use the `openShape` form. The `openShape` must contain a curve that contains an arc with three points. The three points in an arc correspond to the starting point, any point on the arc, and the end point. For example, (`openShape (curve (arc (pt - 5 0) (pt 0 5) (pt 5 0)))`) is an arc with a radius of 5, centered at the origin. Arcs and lines use the `pathWidth` from the `figureGroup` or `figureGroupOverride`; circles and rectangles use `borderWidth`.

The fixed text for a symbol uses `annotate` forms. The `stringDisplay` in `annotate` contains the text as a string. The `stringDisplay` contains a display with the `textHeight`, justification, and location. The symbol form can contain multiple figure and `annotate` forms.

9.4.4 An EDIF Example

In this section we shall illustrate the use of EDIF in translating a cell library from one set of tools to another—from a Compass Design Automation cell library to the Cadence schematic-entry tools. The code in Table 9.12 shows the EDIF description of the symbol for a two-input AND gate, `an02d1`, from the Compass cell library.

The Cadence schematic tools do contain a procedure, `EDIFIN`, that reads the Compass EDIF files. This procedure works, but, as we shall see, results in some problems when you use the icons in the Cadence schematic-entry tool. Instead we shall make some changes to the original files before we use `EDIFIN` to transfer the information to the Cadence database, `cdba`.

The original Compass EDIF file contains a `figureGroup` for each of the following four EDIF cell symbols:

```
connector_FG  icon_FG  instance_FG  net_FG bus_FG
```

The `EDIFIN` application translates each `figureGroup` to a Cadence layer-purpose pair definition that must be defined in the Cadence technology file associated with the library. If we use the original EDIF file with `EDIFIN` this results in the automatic modification of the Cadence technology file to define layer names, purposes, and the required properties to enable use of the `figureGroup` names. This results in non-Cadence layer names in the Cadence database.

First then, we need to modify the EDIF file to use the standard Cadence layer names shown in Table 9.13. These layer names and their associated purposes and properties are defined in the default Cadence technology file, `default.tf`. There is one more layer name in the Compass files (`bus_FG` `figureGroup`), but since this is not used in the library we can remove this definition from the EDIF input file.

TABLE 9.13 Compass and corresponding Cadence `figureGroup` names.

Compass name	Cadence name	Compass name	Cadence name
<code>connector_FG</code>	<code>pin</code>	<code>net_FG</code>	<code>wire</code>
<code>icon_FG</code>	<code>device</code>	<code>bus_FG</code>	not used
<code>instance_FG</code>	<code>instance</code>		

Internal scaling differences lead to giant characters in the Cadence tools if we use the `textHeight` of 30 defined in the EDIF file. Reducing the `textHeight` to 5 results in a reasonable text height.

The EDIF `numberDefinition` construct, together with the `scale` construct, defines measurement scaling in an EDIF file. In a Cadence schematic EDIF file the `numberDefinition` and `scale` construct is determined by an entry in the associated library technology file that defines the `edifUnit` to `userUnit` ratio. This ratio affects the printed size of an icon.

For example, the distance defined by the following path construct is 10 EDIF units:

```
(path (pointlist (pt 0 0) (pt 0 10)))
```

What is the length of 10 EDIF units? The `numberDefinition` and `scale` construct associates EDIF units with a physical dimension. The following construct

```
(numberDefinition (scale 100 (e 25400 -6) unit DISTANCE))
```

specifies that 100 EDIF units equal 25400×10^{-6} m or approximately 1 inch. Cadence defines schematic measurements in inches by defining the `userUnit` property of the affected `viewType` or `viewName` as `inch` in the Cadence technology file. The Compass EDIF files do not provide values for the `numberDefinition` and `scale` construct, and the Cadence tools default to a value of 160 EDIF units to 1 user unit. We thus need to add a `numberDefinition` and `scale` construct to the Compass EDIF file to control the printed size of icons.

The EDIF file defines blank label placeholders for each cell using the EDIF `property` construct. Cadence EDIFIN does recognize and translate EDIF properties, but to attach a label property to a `cellview` object it must be defined (not blank) and identified as a property using the EDIF `owner` construct in the EDIF file. Since the intent of a placeholder is to hold an empty spot for later use and since

Cadence Composer (the schematic-entry tool) supports label additions to instantiated icons, we can remove the EDIF `label` property construct in each cell and the associated `propertyDisplay` construct from the Compass file.

There is a problem that we need to resolve with naming. This is a problem that sooner or later everyone must tackle in ASIC design—**case sensitivity**.

In EDIF, input and output pins are called ports and they are identified using `portImplementation` constructs. In order that the ports of a particular cell icon_view are correctly associated with the ports in the related functional, layout, and abstract views, they must all have the same name. The Cadence tools are case sensitive in this respect. The Verilog and CIF files corresponding to each cell in the Compass library use lowercase names for each port of a given cell, whereas the EDIF file uses uppercase. The EDIFIN translator allows the case of cell, view, and port names to be automatically changed on translation. Thus pin names such as 'A1' become 'a1' and the original view name 'Icon_view' becomes 'icon_view'.

The `boundingBox` construct defines a bounding box around a symbol (icon). Schematic-capture tools use this to implement various functions. The Cadence Composer tool, for example, uses the bounding box to control the wiring between cells and as a highlight box when selecting components of a schematic. Compass uses a large `boundingBox` definition for the cells to allow space for long hierarchical names. Figure 9.10(a) shows the original `an02d1` cell bounding box that is larger than the cell icon.

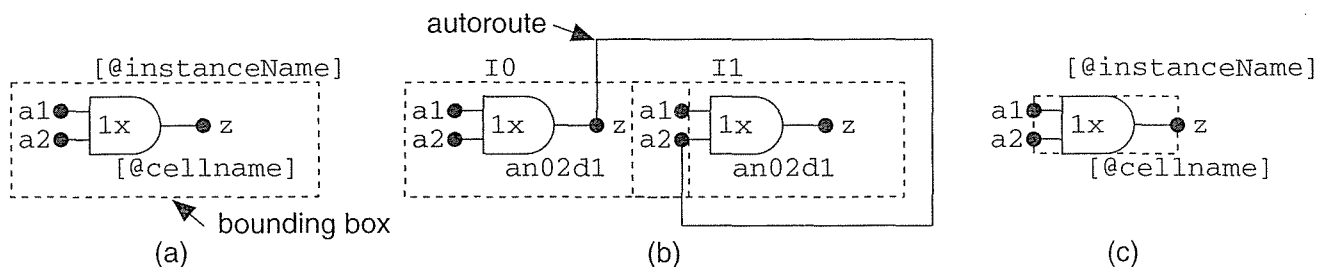


FIGURE 9.10 The bounding box problem. (a) The original bounding box for the `an02d1` icon. (b) Problems in Cadence Composer due to overlapping bounding boxes. (c) A “shrink-wrapped” bounding box created using SKILL.

Icons with large bounding boxes create two problems in Composer. Highlighting all or part of a complex design consisting of many closely spaced cells results in a confusion of overlapped highlight boxes. Also, large boxes force strange wiring patterns between cells that are placed too closely together when Composer's automatic routing algorithm is used. Figure 9.10(b) shows an example of this problem.

There are two solutions to the bounding-box problem. We could modify each `boundingBox` definition in the original EDIF file before translation to conform to the outline of the icon. This involves identifying the outline of each icon in the

EDIF file and is difficult. A simpler approach is to use the Cadence tool programming language, SKILL. SKILL provides direct access to the Cadence database, `cdba`, in order to modify and create objects. Using SKILL you can use a batch file to call functions normally accessed interactively. The solution to the bounding box problem is:

1. Use EDIFIN to create the views in the Cadence database, `cdba`.
2. Use the `schCreateInstBox()` command on each `icon_view` object to eliminate the original bounding box and create a new, minimum-sized, bounding box that is “shrink-wrapped” to each icon.

Figure 9.10(c) shows the results of this process. This modification fixes the problems with highlighting and wiring in Cadence Composer.

This completes the steps required to translate the schematic icons from one set of tools to another. The process can be automated in three ways:

- Write UNIX `sed` and `awk` scripts to make the changes to the EDIF file before using EDIFIN and SKILL.
- Write custom C programs to make the changes to the EDIF file and then proceed as in the first option.
- Perform all the work using SKILL.

The last approach is the most elegant and most easily maintained but is the most difficult to implement (mostly because of the time required to learn SKILL). The whole project took several weeks (including the time it took to learn how to use each of the tools). This is typical of the problems you face when trying to convert data from one system to another.

9.5 CFI Design Representation

The **CAD Framework Initiative (CFI)** is an independent nonprofit organization working on the creation of standards for the electronic CAD industry. One of the areas in which CFI is working is the definition of standards for **design representation (DR)**. The CFI 1.0 standard [CFI, 1992] has tackled the problems of ambiguity in the area of definitions and terms for schematics by defining an **information model (IM)** for electrical connectivity information.

What this means is that a group of engineers got together and proposed a standard way of using the terms and definitions that we have discussed. There are good things and bad things about standards, and one aspect of the CFI 1.0 DR standard illustrates this point. A good thing about the CFI 1.0 DR standard is that it precisely defines what we mean by terms and definitions in schematics, for example. A bad

thing about the CFI DR standard is that in order to be precise it introduces yet more terms that are difficult to understand. A very brief discussion of the CFI 1.0 DR standard is included here, at the end of this chapter, for several reasons:

- It helps to solidify the concepts of the terms and definitions such as cell, net, and instance that we have already discussed. However, there are additional new concepts and terms to define in order to present the standard model, so this is not a good way to introduce schematic terminology.
- The ASIC design engineer is becoming more of a programmer and less of a circuit designer. This trend shows no sign of stopping as ASICs grow larger and systems more complex. A precise understanding of how tools operate and interact is becoming increasingly important.

9.5.1 CFI Connectivity Model

The CFI connectivity model is defined using the **EXPRESS language** and its graphical equivalent **EXPRESS-G**. EXPRESS is an International Standards Organization (ISO) standard [EXPRESS, 1991]. EDIF 3.0.0 and higher also use EXPRESS as the internal formal description of the language. EXPRESS is used to define objects and their relationships. Figure 9.11 shows some simple examples of the EXPRESS-G notation.

The following EXPRESS code (a **schema**) is equivalent to the EXPRESS-G family model shown in Figure 9.11(c):

```

SCHEMA family_model;
  ENTITY person
    ABSTRACT SUPERTYPE OF (ONEOF (man, woman, child));
    name: STRING;
    date of birth: STRING;
  END_ENTITY;

  ENTITY man
    SUBTYPE OF (person);
    wife: SET[0:1] OF woman;
    children: SET[0:?] OF child;
  END_ENTITY;

  ENTITY woman
    SUBTYPE OF (person);
    husband: SET[0:1] OF man;
    children: SET[0:?] OF child;
  END_ENTITY;

  ENTITY child
    SUBTYPE OF (person);
    father: man;
    mother: woman;

```

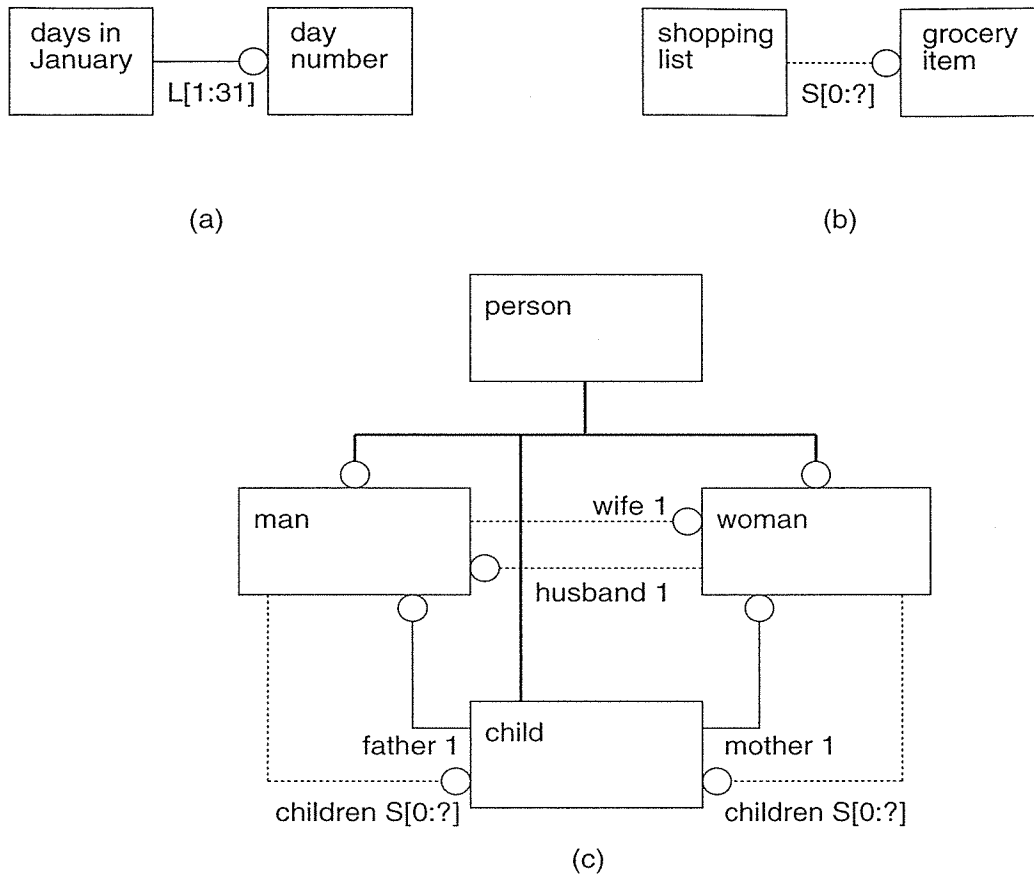


FIGURE 9.11 Examples of EXPRESS-G. (a) Each day in January has a number from 1 to 31. (b) A shopping list may contain a list of items. (c) An EXPRESS-G model for a family.

```
END_ENTITY;
END_SCHEMA;
```

This EXPRESS description is a formal way of saying the following:

- “Men, women, and children are people.”
- “A man can have one woman as a wife, but does not have to.”
- “A wife can have one man as a husband, but does not have to.”
- “A man or a woman can have several children.”
- “A child has one father and one mother.”

Computers can deal more easily with the formal language version of these statements. The formal language and graphical forms are more precise for very complex models.

Figure 9.12 shows the basic structure of the CFI 1.0.0 **Base Connectivity Model (BCM)**. The actual EXPRESS-G diagram for the BCM defined in the CFI 1.0.0 standard is only a little more complicated than Figure 9.12 (containing 21 boxes or types rather than just six). The extra types are used for bundles (a group of nets) and different views of cells (other than the netlist view).

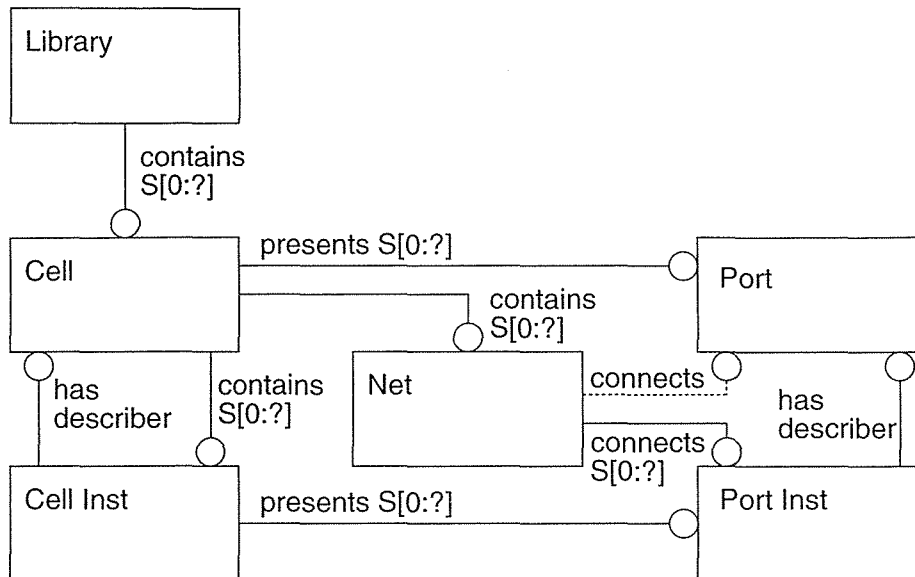


FIGURE 9.12 The original “five-box” model of electrical connectivity. There are actually six boxes or types in this figure; the Library type was added later.

Figure 9.12 says the following (“presents” as used in Figure 9.12 is the Express jargon for “have”):

- “A library contains cells.”
- “Cells have ports, contain nets, and can contain other cells.”
- “Cell instances are copies of a cell and have port instances.”
- “A port instance is a copy of the port in the library cell.”
- “You connect to a port using a net.”
- “Nets connect port instances together.”

Once you understand Figure 9.12 you will see that it replaces the first half of this chapter. Unfortunately you have to read the first half of this chapter to understand Figure 9.12.

9.6 Summary

The important concepts that we covered in this chapter are:

- Schematic entry using a cell library
- Cells and cell instances, nets and ports
- Bus naming, vectored instances in datapath
- Hierarchy
- Editing cells
- PLD languages: ABEL, PALASM, and CUPL
- Logic minimization
- The functions of EDIF
- CFI representation of design information

9.7 Problems

9.1 (EDIF description)

- a. (5 min.) Write an EDIF description for an icon for an inverter (just the input and output wires, a triangle, and a bubble). What problems do you face and what assumptions did you make?
- b. (30 min.+) Try and import your symbol into your schematic-entry tool. If you fail (as you might) explain what the problem is and suggest a direction of attack. *Hint:* If you can, try Problem 9.2 first.

9.2 (EDIF inverter, 15 min.) If you have access to a tool that generates EDIF for the icons, write out the EDIF for an inverter icon. Explain the code.

9.3 (EDIF netlist, 20 min.) Starting with an empty directory and using a schematic editor (such as Viewlogic) draw a schematic with a single inverter (from any cell library).

- a. List the files that are created in the directory.
- b. Print each one (check first to make sure it is ASCII, not binary).
- c. Try and explain the contents.

9.4 (Minitutorial, 60 min.) Write a minitutorial (no more than five pages) that explains how to set up your system (location and nature of any start-up files such as `.ini` files for Viewlogic and so on); how to choose or change a library (for cell icons); how to choose cells, instantiate, label, and connect them; how to select, copy and delete symbols; and how to save a schematic. Use a single inverter connected to an input and output pad as an example.

9.5 (Icons, 30 min.) With an example show how to edit and create a symbol icon. Make a triangular icon (the same size as an inverter in your library but without a bubble) for a series connection of two inverters and call it `myBuffer`.

9.6 (Buses, 30 min.)

a. Create an example of a 16-bit bus: connect 8 inverters to bit zero (the MSB or leftmost bit) and bits 10–16 (as if we were taking the sign bit, bit zero, and the seven least-significant bits from a 16-bit signed number). Name the inverter connected to the sign bit, `SIGN`. Name the other inverters `BIT0` through `BIT7`.

b. Write the netlist as an EDIF file, number the lines, and explain the contents by referencing line numbers.

9.7 (VDD and VSS, 30 min.) Using a simple example of two inverters (one with input connected to VDD, the other with input connected to VSS or GND) explain how your schematic-entry system handles global power and ground nets and their connection to cell pins. Can you connect VDD or VSS to an output pin in your system? If your schematic software has a netlist screener, try it on this example.

9.8 (Hierarchy, 30 min.) Create a very simple hierarchical cell. The lowest level, named `bottom`, contains a single inverter (named `invB`). The highest level, called `top`, contains another inverter, `invT`, whose input is connected to the output of cell `bottom`. Write out the netlist (in internal and EDIF format) and explain how the tool labels a hierarchical cell.

9.9 (Vectored instances, 30 min.) Create a vectored instance of eight inverters, `inv0` through `inv7`. Write the netlist in internal and EDIF form and explain the contents.

9.10 (Dangling wires, 30 min.) Create a cell, `dangle1`, containing two inverters, `inv1` and `inv2`. Connect the input of `inv1` to an external connector, `in1`, and the output of `inv2` to an external connector `out2`. Write the netlist and explain what happens to the unlabeled and unused nets. If you have a netlist screener, run it on this example.

9.11 (PLD languages, 60 min.) Conduct a Web search on ABEL, CUPL, or PALASM (start by searching for “Logical Devices” not “ABEL”). Try and find examples of these files and write an explanation of their function using the descriptions of these languages in this chapter.

9.12 (EDIF 3 0 0, 10 min.) Download the EDIF 3 0 0 example schematic file from <http://www.edif.org/edif/workshop.edf> and see if your EDIF reader will accept it. What is it?

9.13 (EXPRESS-G, 15 min.) Draw an EXPRESS-G diagram for the government of your country. For example, in the United States you would start with the president and the White House and work down through the House and Senate, showing the senators and congressional representatives. In the United Kingdom you would draw the prime minister, the House of Commons, and House of Lords with the various MPs.

9.14 (ABEL PCI Target) (10 min.) Download the Xilinx Application Note, Designing Flexible PCI Interfaces with Xilinx EPLDs, January 1995 ([pci_epld.pdf](http://www.xilinx.com/pci_epld.pdf) at www.xilinx.com). The Appendix of this App. Note contains the ABEL source code for a PCI Bus Interface Target. The code is long but straightforward; most of it describes the next-state transitions for the bus-controller state machine. Extract the ABEL source code using Adobe Acrobat. *Hint:* This is not easy; Acrobat does a poor job of selecting text; you will lose many semicolons at the end of lines that you will have to add by hand. Use Replace... to search for end-of-line, "^p", and replace by " ; ^p" in Word. (60 min.+). Try to convert this code to a system where you can compile it. You may need conversion utilities to do this. For example Altera (www.altera.com) has utilities (EAU018.EXE and EAU019.EXE located at ftp.altera.com/pub) to convert from ABEL 4.0 to AHDL.

9.15 (CUPL, 60 min.) Download and install the CUPL demonstration package from <http://www.protel.com/download.htm>. Write a two-page help sheet on what you did, where the software is installed, and how to run it.

9.16 (PALASM) (30 min.) Download and install PALASM4 v1.5 from the AMD Web site at [ftp://ftp.amd.com/pub/pld/software/palasm](http://ftp.amd.com/pub/pld/software/palasm).

9.17 (CUPL)

- a. (15 min.) Check the equations in the CUPL code for the 4-bit counter in Section 9.2.
- b. (10 min.) Add a count-enable signal to the code.
- c. (30 min.) If you have access to CUPL, compile your answer.

9.18 (EDIF)

- a. (30 min.) Using the syntax definitions below and the example schematic icon shown in Table 9.12 to help you, "stitch" back together the EDIF definition for the 7404 inverter symbol used as an example in Section 9.4.3.
- b. (60 min.+). Try to import the EDIF into your schematic entry system. Comment on any problems and how you attempted to resolve them (including failures).

The **EDIF Reference Manual** [EDIF, 1988] uses the following metasyntax rules:

```
[optional] <at most once> {may be repeated zero or more times}
{this|that} indicates any number of this or that in any order
syntactic names are italic
literal words are bold
SYMBOLIC constants are uppercase
IdentifierNameDef means the name is being defined
IdentifierNameRef means the name is being referenced
```

The syntax definitions of the most common EDIF constructs for schematics are as follows:

```
(edif edifFileNameDef
  edifVersion
```

```

    edifLevel
    keywordMap
    {<status>|external|library|design|comment|userdata} )
(library libraryNameDef
    edifLevel
    technology
    {<status>|cell|comment|userdata} )
(technology numberDefinition
    {figureGroup|fabricate|
    <simulationInfos>|<physicalDesignRule>|comment|userdata} )
(cell cellNameDef
    cellType
    {<status>|view|<viewMap>|property|comment|userdata} )
(view viewNameDef
    viewType
    interface
    {<status>|<contents>|comment|property|userdata} )
(interface
    {port|portBundle|<symbol>|<protectionFrame>|
    <arrayRelatedInfo>|parameter|joined|mustJoin|weakJoined|
    permutable|timing|simulate|<designator>|property|comment|userdata} )
(contents
    {instance|offPageConnector|figure|section|
    net|netBundle|page|commentGraphics|portImplementation|
    timing|simulate|when|follow|logicPort|<boundingBox>|
    comment|userdata} )
(viewMap
    {portMap|portBackAnnotate|instanceMap|instanceBackAnnotate|
    netMap|netBackAnnotate|comment|userdata} )

```

9.8 Bibliography

The data books from AMD, Atmel, and other PLD manufacturers are excellent sources of tutorials, examples, and information on PLD design. The EDIF tutorials produced by the EIA [EDIF, 1988, 1989] are hard to find, but there are few other texts or sources that explain EDIF. EDIF does have a World Wide Web site at <http://www.edif.org>. The EDIF Technical Centre at the University of Manchester (<http://www.cs.man.ac.uk/cad>, I shall refer to this as ~EDIF) serves as a resource center for EDIF, including the formal information models of the EDIF language in EXPRESS format and the BNF definitions of the language syntax. There is a hypertext version of an EDIF 300 schematic file with hypertext links at ~EDIF/EDIFTechnicalCenter/software. CFI has a home page and links to other sites at <http://www.cfi.org>.

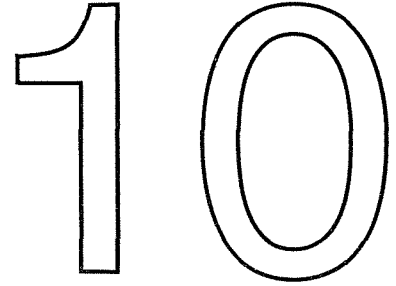
PALASM4 v1.5 is available as “freeware” from AMD at <ftp://ftp.amd.com/pub/pld/software/palasm>. The Data I/O home page at <http://www.data-io.com> is devoted mainly to Synario. The Viewlogic home page is <http://www.viewlogic.com>. Capilano Computing has a Web page at <http://www.capilano.com> with DesignWorks and MacABEL software. Protel (<http://www.protel.com/download.htm>) has Windows-based schematic-entry tools for FPGAs and a CUPL demonstration package. Logical Devices has a site at <http://www.logicaldevices.com>. Atmel has several demonstration and code examples for ABEL and CUPL at <ftp://www.atmel.com/pub/atmel>.

9.9 References

Page numbers in brackets after a reference indicate its location in the chapter body.

- CFI Standards for Electronic Design Automation Release 1.0. 1992. CFI published a four-volume set in 1992, ISBN 1-882750-00-4 (set). The first volume, ISBN 1-882750-01-2, is approximately 300 pages and contains a brief introduction (approximately 10 pages) and the Electrical Connectivity model. Unfortunately two of the volumes were labeled as volume three. The (first) third volume is the Tool Encapsulation Specification, ISBN 1-882750-03-09 (approximately 100 pages). The (second) third volume, ISBN 1-882750-02-0, covers the Inter-Tool Communication Programming Interface (approximately 150 pages). The fourth volume, ISBN 1-882750-04-7, is approximately 100 pages long and covers the Computing Environment Services requirement [p. 369].
- EDIF is maintained by the EIA, EIA Standards Sales Office, 2001 Pennsylvania Ave., N.W., Washington, DC 20006, (202) 457-4966 [p. 355]:
- EDIF Steering Committee. 1988. EDIF Reference Manual Version 2.0.0. Washington, DC: Electronic Industries Association. ISBN 0-7908-0000-4.
- EDIF Steering Committee. 1988. Introduction to EDIF. Washington, DC: Electronic Industries Association. ISBN 0-7908-0001-2.
- EDIF Steering Committee. 1989. EDIF Connectivity. Washington, DC: Electronic Industries Association. ISBN 0-7908-0002-0.
- EDIF Schematic Technical Subcommittee. 1989. Using EDIF 2.0.0 for Schematic Transfer. Washington, DC: Electronic Industries Association.
- EXPRESS Language Reference Manual. ISO TC184/SC4/WG5 Document N14, March 29, 1991 [p. 370].

VHDL



10.1	A Counter	10.11	Operators
10.2	A 4-bit Multiplier	10.12	Arithmetic
10.3	Syntax and Semantics of VHDL	10.13	Concurrent Statements
10.4	Identifiers and Literals	10.14	Execution
10.5	Entities and Architectures	10.15	Configurations and Specifications
10.6	Packages and Libraries	10.16	An Engine Controller
10.7	Interface Declarations	10.17	Summary
10.8	Type Declarations	10.18	Problems
10.9	Other Declarations	10.19	Bibliography
10.10	Sequential Statements	10.20	References

The U.S. Department of Defense (DoD) supported the development of **VHDL** (**VHSIC hardware description language**) as part of the **VHSIC** (**very high-speed IC**) program in the early 1980s. The companies in the VHSIC program found they needed something more than schematic entry to describe large ASICs, and proposed the creation of a hardware description language. VHDL was then handed over to the Institute of Electrical and Electronics Engineers (IEEE) in order to develop and approve the IEEE Standard 1076-1987.¹ As part of its standardization process the DoD has specified the use of VHDL as the documentation, simulation, and verification medium for ASICs (MIL-STD-454). Partly for this reason VHDL has gained

¹Some of the material in this chapter is reprinted with permission from IEEE Std 1076-1993, © 1993 IEEE. All rights reserved.

rapid acceptance, initially for description and documentation, and then for design entry, simulation, and synthesis as well.

The first revision of the 1076 standard was approved in 1993. References to the **VHDL Language Reference Manual (LRM)** in this chapter—[VHDL 87LRM2.1, 93LRM2.2] for example—point to the 1987 and 1993 versions of the LRM [IEEE, 1076-1987 and 1076-1993]. The prefixes 87 and 93 are omitted if the references are the same in both editions. Technically 1076-1987 (known as VHDL-87) is now obsolete and replaced by 1076-1993 (known as VHDL-93). Except for code that is marked 'VHDL-93 only' the examples in this chapter can be **analyzed** (the VHDL word for “compiled”) and simulated using both VHDL-87 and VHDL-93 systems.

10.1 A Counter

The following VHDL model describes an electrical “black box” that contains a 50MHz clock generator and a counter. The counter increments on the negative edge of the clock, counting from zero to seven, and then begins at zero again. The model contains separate *processes* that execute at the same time as each other. Modeling concurrent execution is the major difference between HDLs and computer programming languages such as C.

```
entity Counter_1 is end; -- declare a "black box" called Counter_1
library STD; use STD.TEXTIO.all; -- we need this library to print
architecture Behave_1 of Counter_1 is -- describe the "black box"
-- declare a signal for the clock, type BIT, initial value '0'
    signal Clock : BIT := '0';
-- declare a signal for the count, type INTEGER, initial value 0
    signal Count : INTEGER := 0;
begin
    process begin -- process to generate the clock
        wait for 10 ns; -- a delay of 10 ns is half the clock cycle
        Clock <= not Clock;
        if (now > 340 ns) then wait; end if; -- stop after 340 ns
    end process;
-- process to do the counting, runs concurrently with other processes
    process begin
-- wait here until the clock goes from 1 to 0
        wait until (Clock = '0');
-- now handle the counting
        if (Count = 7) then Count <= 0;
        else Count <= Count + 1;
        end if;
    end process;
    process (Count) variable L: LINE; begin -- process to print
        write(L, now); write(L, STRING(" Count="));
        write(L, Count); writeline(output, L);
    end process;
end;
```

Throughout this book VHDL **keywords** (reserved words that are part of the language) are shown in bold type in code examples (but not in the text). The code examples use the bold keywords to improve readability. VHDL code is often lengthy and the code in this book is always complete wherever possible. In order to save space many of the code examples do not use the conventional spacing and formatting that is normally considered good practice. So “Do as I say and not as I do.”

The steps to simulate the model and the printed results for Counter_1 using the Model Technology V-System/Plus common-kernel simulator are as follows:

```
> vlib work
> vcom Counter_1.vhd
Model Technology VCOM V-System VHDL/Verilog 4.5b
-- Loading package standard
-- Compiling entity counter_1
-- Loading package textio
-- Compiling architecture behave_1 of counter_1
> vsim -c counter_1
# Loading ../std.standard
# Loading ../std.textio(body)
# Loading work.counter_1(behave_1)
VSIM 1> run 500
# 0 ns Count=0
# 20 ns Count=1
(...15 lines omitted...)
# 340 ns Count=1
VSIM 2> quit
>
```

10.2 A 4-bit Multiplier

This section presents a more complex VHDL example to motivate the study of the syntax and semantics of VHDL in the rest of this chapter.

10.2.1 An 8-bit Adder

Table 10.1 shows a VHDL model for the full adder that we described in Section 2.6, “Datapath Logic Cells.” Table 10.2 shows a VHDL model for an 8-bit ripple-carry adder that uses eight instances of the full adder.

10.2.2 A Register Accumulator

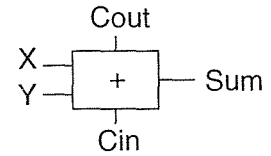
Table 10.3 shows a VHDL model for a positive-edge-triggered D flip-flop with an active-high asynchronous clear. Table 10.4 shows an 8-bit register that uses this D flip-flop model (this model only provides the Q output from the register and leaves the QN flip-flop outputs unconnected).

TABLE 10.1 A full adder.

```

entity Full_Adder is                                --1
  generic (TS : TIME := 0.11 ns; TC : TIME := 0.1 ns); --2
  port (X, Y, Cin: in BIT; Cout, Sum: out BIT);      --3
end Full_Adder;                                    --4

architecture Behave of Full_Adder is                --5
begin                                               --6
  Sum <= X xor Y xor Cin after TS;                  --7
  Cout <= (X and Y) or (X and Cin) or (Y and Cin) after TC; --8
end;                                                --9
    
```



Timing:
 TS (Input to Sum) = 0.11 ns
 TC (Input to Cout) = 0.1 ns

TABLE 10.2 An 8-bit ripple-carry adder.

```

entity Adder8 is                                    --1
  port (A, B: in BIT_VECTOR(7 downto 0);          --2
        Cin: in BIT; Cout: out BIT;              --3
        Sum: out BIT_VECTOR(7 downto 0));        --4
end Adder8;                                        --5

architecture Structure of Adder8 is                --6
  component Full_Adder                             --7
  port (X, Y, Cin: in BIT; Cout, Sum: out BIT);    --8
end component;                                    --9
  signal C: BIT_VECTOR(7 downto 0);               --10
begin                                              --11
  Stages: for i in 7 downto 0 generate             --12
    LowBit: if i = 0 generate                       --13
      FA:Full_Adder port map (A(0),B(0),Cin,C(0),Sum(0)); --14
    end generate;                                  --15
    OtherBits: if i /= 0 generate                  --16
      FA:Full_Adder port map                       --17
        (A(i),B(i),C(i-1),C(i),Sum(i));          --18
    end generate;                                  --19
  end generate;                                    --20
  Cout <= C(7);                                   --21
end;                                               --22
    
```

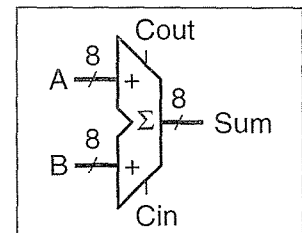
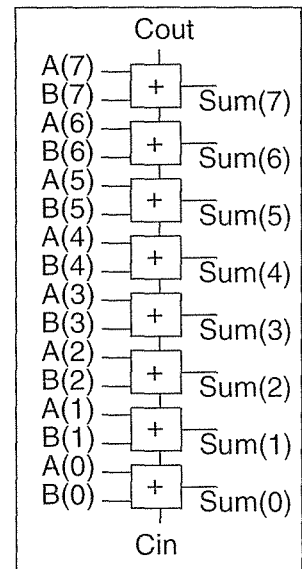


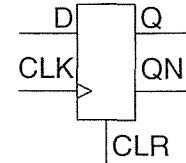
TABLE 10.3 Positive-edge-triggered D flip-flop with asynchronous clear.

```

entity DFFClr is                                --1
  generic(TRQ : TIME := 2 ns; TCQ : TIME := 2 ns); --2
  port (CLR, CLK, D : in BIT; Q, QB : out BIT); --3
end;                                             --4

architecture Behave of DFFClr is                --5
  signal Qi : BIT;                               --6
begin QB <= not Qi; Q <= Qi;                   --7
process (CLR, CLK) begin                        --8
  if CLR = '1' then Qi <= '0' after TRQ;       --9
  elsif CLK'EVENT and CLK = '1'                --10
    then Qi <= D after TCQ;                     --11
  end if;                                       --12
end process;                                    --13
end;                                             --14

```



Timing:
 TRQ (CLR to Q/QN) = 2 ns
 TCQ (CLK to Q/QN) = 2 ns

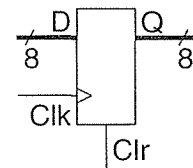
TABLE 10.4 An 8-bit register.

```

entity Register8 is                              --1
  port (D : in BIT_VECTOR(7 downto 0);         --2
        Clk, Clr: in BIT ; Q : out BIT_VECTOR(7 downto 0)); --3
end;                                             --4

architecture Structure of Register8 is          --5
  component DFFClr                               --6
    port (Clr, Clk, D : in BIT; Q, QB : out BIT); --7
  end component;                                --8
begin                                           --9
  STAGES: for i in 7 downto 0 generate          --10
    FF: DFFClr port map (Clr, Clk, D(i), Q(i), open); --11
  end generate;                                  --12
end;                                             --13

```



8-bit register. Uses DFFClr positive edge-triggered flip-flop model.

Table 10.5 shows a model for a datapath multiplexer that consists of eight 2:1 multiplexers with a common select input (this select signal would normally be a control signal in a datapath). The multiplier will use the register and multiplexer components to implement a register accumulator.

10.2.3 Zero Detector

Table 10.6 shows a model for a variable-width zero detector that accepts a bus of any width and will produce a single-bit output of '1' if all input bits are zero.

TABLE 10.5 An 8-bit multiplexer.

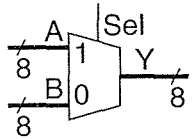
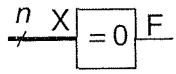
<pre>entity Mux8 is generic (TPD : TIME := 1 ns); port (A, B : in BIT_VECTOR (7 downto 0); Sel : in BIT := '0'; Y : out BIT_VECTOR (7 downto 0)); end; architecture Behave of Mux8 is begin Y <= A after TPD when Sel = '1' else B after TPD; end;</pre>	<pre>--1 --2 --3 --4 --5 --6 --7 --8 --9</pre>	 <p>Eight 2:1 MUXs with single select input. Timing: TPD(input to Y)=1 ns</p>
--	--	--

TABLE 10.6 A zero detector.

<pre>entity AllZero is generic (TPD : TIME := 1 ns); port (X : BIT_VECTOR; F : out BIT); end; architecture Behave of AllZero is begin process (X) begin F <= '1' after TPD; for j in X'RANGE loop if X(j) = '1' then F <= '0' after TPD; end if; end loop; end process; end;</pre>	<pre>--1 --2 --3 --4 --5 --6 --7 --8 --9 --10 --11</pre>	 <p>Variable-width zero detector. Timing: TPD(X to F) =1 ns</p>
--	--	---

10.2.4 A Shift Register

Table 10.7 shows a variable-width shift register that shifts (left or right under input control, DIR) on the positive edge of the clock, CLK, gated by a shift enable, SH. The parallel load, LD, is synchronous and aligns the input LSB to the LSB of the output, filling unused MSBs with zero. Bits vacated during shifts are zero filled. The clear, CLR, is asynchronous.

10.2.5 A State Machine

To multiply two binary numbers A and B, we can use the following algorithm:

1. If the LSB of A is '1', then add B into an accumulator.
2. Shift A one bit to the right and B one bit to the left.
3. Stop when all bits of A are zero.

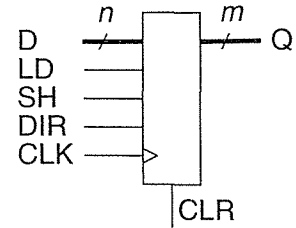
TABLE 10.7 A variable-width shift register.

```

entity ShiftN is --1
  generic (TCQ : TIME := 0.3 ns; TLQ : TIME := 0.5 ns; --2
    TSQ : TIME := 0.7 ns); --3
  port(CLK, CLR, LD, SH, DIR: in BIT; --4
    D: in BIT_VECTOR; Q: out BIT_VECTOR); --5
  begin assert (D'LENGTH <= Q'LENGTH) --6
    report "D wider than output Q" severity Failure; --7
end ShiftN; --8

architecture Behave of ShiftN is --9
  begin Shift: process (CLR, CLK) --10
    subtype InB is NATURAL range D'LENGTH-1 downto 0; --11
    subtype OutB is NATURAL range Q'LENGTH-1 downto 0; --12
    variable St: BIT_VECTOR(OutB); --13
  begin --14
    if CLR = '1' then --15
      St := (others => '0'); Q <= St after TCQ; --16
    elsif CLK'EVENT and CLK='1' then --17
      if LD = '1' then --18
        St := (others => '0'); --19
        St(InB) := D; --20
        Q <= St after TLQ; --21
      elsif SH = '1' then --22
        case DIR is --23
          when '0' => St := '0' & St(St'LEFT downto 1); --24
          when '1' => St := St(St'LEFT-1 downto 0) & '0'; --25
        end case; --26
        Q <= St after TSQ; --27
      end if; --28
    end if; --29
  end process; --30
end; --31

```



- CLK Clock
- CLR Clear, active high
- LD Load, active high
- SH Shift, active high
- DIR Direction, 1 = left
- D Data in
- Q Data out

Variable-width shift register. Input width must be less than output width. Output is left-shifted or right-shifted under control of DIR. Unused MSBs are zero-padded during load. Clear is asynchronous. Load is synchronous.

Timing:
 TCQ (CLR to Q) = 0.3ns
 TLQ (LD to Q) = 0.5ns
 TSQ (SH to Q) = 0.7ns

Table 10.8 shows the VHDL model for a Moore (outputs depend only on the state) finite-state machine for the multiplier, together with its state diagram.

10.2.6 A Multiplier

Table 10.9 shows a schematic and the VHDL code that describes the interconnection of all the components for the multiplier. Notice that the schematic comprises two halves: an 8-bit-wide datapath section (consisting of the registers, adder, multiplexer,

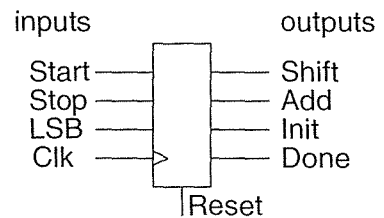
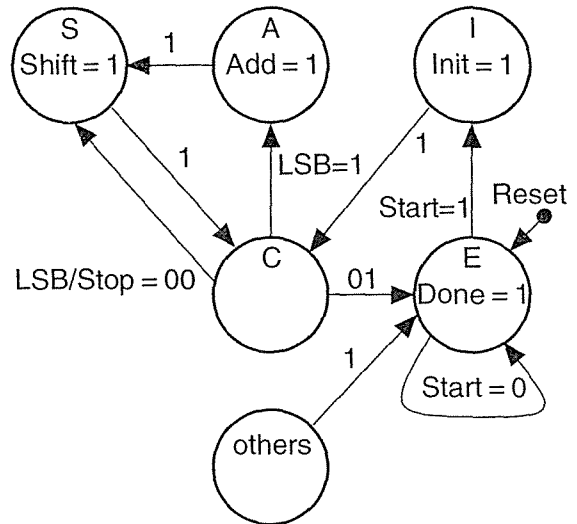
TABLE 10.8 A Moore state machine for the multiplier.

```

entity SM_1 is --1
  generic (TPD : TIME := 1 ns); --2
  port(Start, Clk, LSB, Stop, Reset: in BIT; --3
       Init, Shift, Add, Done : out BIT); --4
end; --5

architecture Moore of SM_1 is --6
  type STATETYPE is (I, C, A, S, E); --7
  signal State: STATETYPE; --8
begin --9
  Init <= '1' after TPD when State = I --10
    else '0' after TPD; --11
  Add <= '1' after TPD when State = A --12
    else '0' after TPD; --13
  Shift <= '1' after TPD when State = S --14
    else '0' after TPD; --15
  Done <= '1' after TPD when State = E --16
    else '0' after TPD; --17
  process (CLK, Reset) begin --18
    if Reset = '1' then State <= E; --19
    elsif CLK'EVENT and CLK = '1' then --20
      case State is --21
        when I => State <= C; --22
        when C => --23
          if LSB = '1' then State <= A; --24
          elsif Stop = '0' then State <= S; --25
          else State <= E; --26
          end if; --27
        when A => State <= S; --28
        when S => State <= C; --29
        when E => --30
          if Start = '1' then State <= I; end if; --31
        end case; --32
      end if; --33
    end process; --34
end; --35

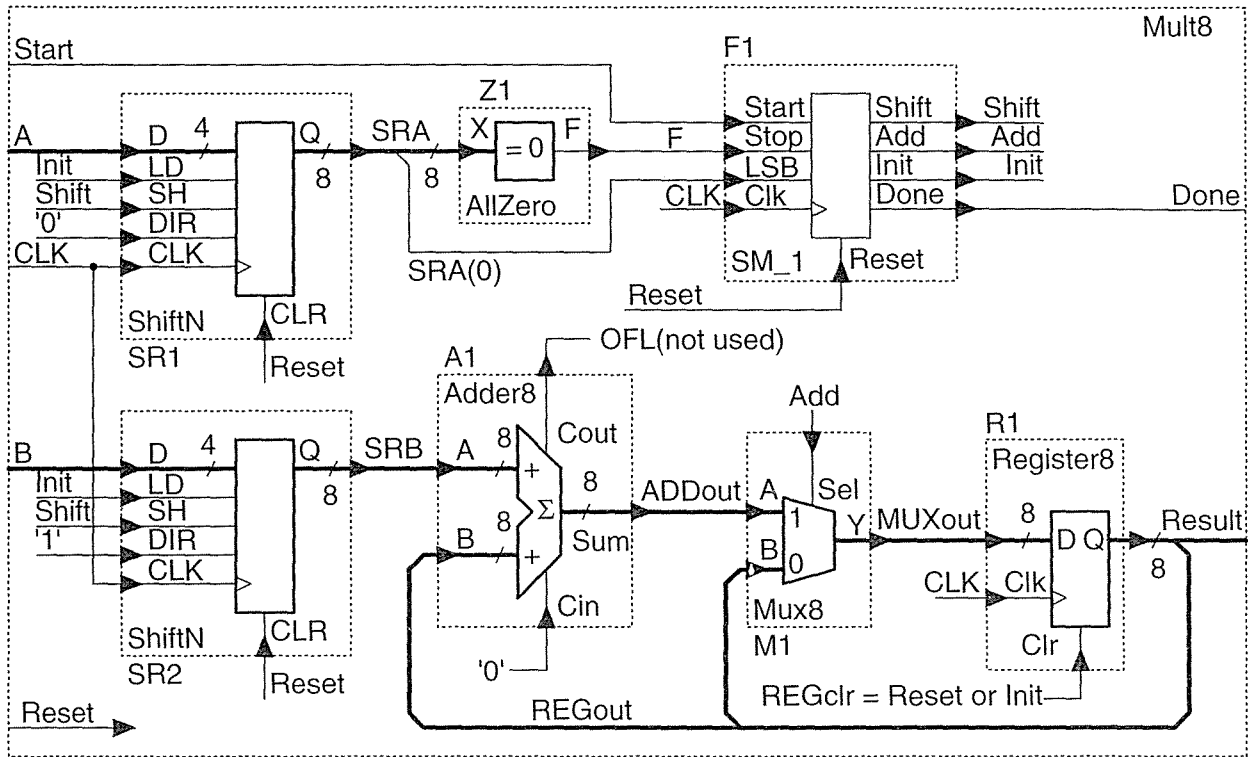
```



State	Function
E	End of multiply cycle.
I	Initialize: clear output register and load input registers.
C	Check if LSB of register A is zero.
A	Add shift register B to accumulator.
S	Shift input register A right and input register B left.

and zero detector) and a control section (the finite-state machine). The arrows in the schematic denote the inputs and outputs of each component. As we shall see in Section 10.7, VHDL has strict rules about the direction of connections.

TABLE 10.9 A 4-bit by 4-bit multiplier.



```

entity Mult8 is --1
port (A, B: in BIT_VECTOR(3 downto 0); Start, CLK, Reset: in BIT; --2
Result: out BIT_VECTOR(7 downto 0); Done: out BIT); end Mult8; --3

architecture Structure of Mult8 is use work.Mult_Components.all; --4
signal SRA, SRB, ADDout, MUXout, REGout: BIT_VECTOR(7 downto 0); --5
signal Zero, Init, Shift, Add, Low: BIT := '0'; signal High: BIT := '1'; --6
signal F, OFL, REGclr: BIT; --7
begin --8
REGclr <= Init or Reset; Result <= REGout; --9
SR1 : ShiftN port map(CLK=>CLK,CLR=>Reset,LD=>Init,SH=>Shift,DIR=>Low ,D=>A,Q=>SRA); --10
SR2 : ShiftN port map(CLK=>CLK,CLR=>Reset,LD=>Init,SH=>Shift,DIR=>High,D=>B,Q=>SRB); --11
Z1 : AllZero port map(X=>SRA,F=>Zero); --12
A1 : Adder8 port map(A=>SRB,B=>REGout,Cin=>Low,Cout=>OFL,Sum=>ADDout); --13
M1 : Mux8 port map(A=>ADDout,B=>REGout,Sel=>Add,Y=>MUXout); --14
R1 : Register8 port map(D=>MUXout,Q=>REGout,Clk=>CLK,Clr=>REGclr); --15
F1 : SM_1 port map(Start,CLK,SRA(0),Zero,Reset,Init,Shift,Add,Done); --16
end; --17

```

10.2.7 Packages and Testbench

To complete and test the multiplier design we need a few more items. First we need the following “components list” for the items in Table 10.9:

```

package Mult_Components is                                --1
component Mux8 port (A,B:BIT_VECTOR(7 downto 0));        --2
  Sel:BIT;Y:out BIT_VECTOR(7 downto 0));end component;    --3
component AllZero port (X : BIT_VECTOR;                  --4
  F:out BIT );end component;                              --5
component Adder8 port (A,B:BIT_VECTOR(7 downto 0);Cin:BIT; --6
  Cout:out BIT;Sum:out BIT_VECTOR(7 downto 0));end component; --7
component Register8 port (D:BIT_VECTOR(7 downto 0);      --8
  Clk,Clr:BIT; Q:out BIT_VECTOR(7 downto 0));end component; --9
component ShiftN port (CLK,CLR,LD,SH,DIR:BIT;D:BIT_VECTOR; --10
  Q:out BIT_VECTOR);end component;                       --11
component SM_1 port (Start,CLK,LSB,Stop,Reset:BIT;       --12
  Init,Shift,Add,Done:out BIT);end component;           --13
end;                                                       --14

```

Next we need some utility code to help test the multiplier. The following VHDL generates a clock with programmable “high” time (HT) and “low” time (LT):

```

package Clock_Utills is                                  --1
procedure Clock (signal C: out Bit; HT, LT:TIME);       --2
end Clock_Utills;                                      --3

package body Clock_Utills is                             --4
procedure Clock (signal C: out Bit; HT, LT:TIME) is     --5
begin                                                    --6
  loop C<='1' after LT, '0' after LT + HT; wait for LT + HT; --7
  end loop;                                             --8
end;                                                    --9
end Clock_Utills;                                       --10

```

Finally, the following code defines two functions that we shall also use for testing—the functions convert an array of bits to a number and vice versa:

```

package Utills is                                       --1
  function Convert (N,L: NATURAL) return BIT_VECTOR;   --2
  function Convert (B: BIT_VECTOR) return NATURAL;     --3
end Utills;                                             --4

package body Utills is                                  --5
  function Convert (N,L: NATURAL) return BIT_VECTOR is --6
    variable T:BIT_VECTOR(L-1 downto 0);              --7
    variable V:NATURAL:= N;                            --8
    begin for i in T'RIGHT to T'LEFT loop              --9
      T(i) := BIT'VAL(V mod 2); V:= V/2;               --10
    end loop; return T;                                --11
  end;                                                  --12

```

```

function Convert (B: BIT_VECTOR) return NATURAL is           --13
  variable T:BIT_VECTOR(B'LENGTH-1 downto 0) := B;         --14
  variable V:NATURAL:= 0;                                   --15
  begin for i in T'RIGHT to T'LEFT loop                     --16
    if T(i) = '1' then V:= V + (2**i); end if;             --17
  end loop; return V;                                       --18
  end;                                                       --19
end Utils;                                                  --20

```

The following code tests the multiplier model. This is a **testbench** (this simple example is not a comprehensive test). First we reset the logic (line 17) and then apply a series of values to the inputs, A and B. The clock generator (line 14) supplies a clock with a 20 ns period. The inputs are changed 1 ns after a positive clock edge, and remain stable for 20 ns through the next positive clock edge.

```

entity Test_Mult8_1 is end; -- runs forever, use break!!    --1
architecture Structure of Test_Mult8_1 is                  --2
use Work.Utils.all; use Work.Clock_Utils.all;             --3
  component Mult8 port                                     --4
    (A, B : BIT_VECTOR(3 downto 0); Start, CLK, Reset : BIT;
    Result : out BIT_VECTOR(7 downto 0); Done : out BIT); --5
  end component;                                         --7
signal A, B : BIT_VECTOR(3 downto 0);                     --8
signal Start, Done : BIT := '0';                          --9
signal CLK, Reset : BIT;                                  --10
signal Result : BIT_VECTOR(7 downto 0);                   --11
signal DA, DB, DR : INTEGER range 0 to 255;              --12
begin                                                     --13
C: Clock(CLK, 10 ns, 10 ns);                              --14
UUT: Mult8 port map (A, B, Start, CLK, Reset, Result, Done); --15
DR <= Convert(Result);                                     --16
Reset <= '1', '0' after 1 ns;                             --17
process begin                                             --18
  for i in 1 to 3 loop for j in 4 to 7 loop               --19
    DA <= i; DB <= j;                                     --20
    A<=Convert(i,A'Length);B<=Convert(j,B'Length);       --21
    wait until CLK'EVENT and CLK='1'; wait for 1 ns;     --22
    Start <= '1', '0' after 20 ns; wait until Done = '1'; --23
    wait until CLK'EVENT and CLK='1';                     --24
  end loop; end loop;                                     --25
  for i in 0 to 1 loop for j in 0 to 15 loop              --26
    DA <= i; DB <= j;                                     --27
    A<=Convert(i,A'Length);B<=Convert(j,B'Length);       --28
    wait until CLK'EVENT and CLK='1'; wait for 1 ns;     --29
    Start <= '1', '0' after 20 ns; wait until Done = '1'; --30
    wait until CLK'EVENT and CLK='1';                     --31
  end loop; end loop;                                     --32
wait;                                                      --33

```



```

end process;                                --34
end;                                         --35

```

Here is the signal trace output from the Compass Scout simulator:

Time(fs) + Cycle	da	db	dr
0+ 0:	0	0	0
0+ 1: *	1 *	4 *	0
...			
92000000+ 3:	1	4 *	4
...			
150000000+ 1: *	1 *	5	4
...			
193000000+ 3:	1	5 *	0
...			
252000000+ 3:	1	5 *	5
...			
310000000+ 1: *	1 *	6	5
...			
353000000+ 3:	1	6 *	0
...			
412000000+ 3:	1	6 *	6

Positive clock edges occur at 10, 30, 50, 70, 90, ... ns. You can see that the output (dr) changes from '0' to '4' at 92 ns, after five clock edges (with a 2 ns delay due to the output register, R1).

10.3 Syntax and Semantics of VHDL

We might define the **syntax** of a very small subset of the English language in **Backus-Naur form (BNF)** using **constructs** as follows:

```

sentence ::= subject verb object.
subject  ::= The|A noun
object   ::= [article] noun {, and article noun}
article  ::= the|a
noun     ::= man|shark|house|food
verb     ::= eats|paints

::= means "can be replaced by"
| means "or"
[] means "contents optional"
{} means "contents can be left out, used once, or repeated"

```

The following two English sentences are correct according to these syntax rules:

A shark eats food.

The house paints the shark, and the house, and a man.

We need **semantic rules** to tell us that the second sentence does not make much sense. Most of the VHDL LRM is dedicated to the definition of the language semantics. Appendix A of the LRM (which is not officially part of the standard) explains the complete VHDL syntax using BNF.

The rules that determine the characters you can use (the “alphabet” of VHDL), where you can put spaces, and so on are **lexical rules** [VHDL LRM13]. Any VHDL description may be written using a subset of the VHDL character set:

```
basic_character ::= upper_case_letter|digit|special_character
                 |space_character|format_effector
```

The two space characters are: space (SP) and the nonbreaking space (NBSP). The five format effectors are: horizontal tabulation (HT), vertical tabulation (VT), carriage return (CR), line feed (LF), and form feed (FF). The characters that are legal in VHDL constructs are defined as the following subsets of the complete character set:

```
graphic_character ::=                                     [10.1]
    upper_case_letter|digit|special_character|space_character
    |lower_case_letter|other_special_character
```

```
special_character ::= " # & ' ( ) * + , - . / : ; < = > [ ] _ | [10.2]
```

The 11 other special characters are: ! \$ % @ ? \ ^ ` { } ~, and (in VHDL-93 only) 34 other characters from the ISO Latin-1 set [ISO, 1987]. If you edit code using a word processor, you either need to turn smart quotes off or override this feature (use Tools... Preferences... General in MS Word; and use CTRL-' and CTRL-" in Frame).

When you learn a language it is difficult to understand how to use a noun without using it in a sentence. Strictly this means that we ought to define a sentence before we define a noun and so on. In this chapter I shall often break the “Define it before you use it” rule and use code examples and BNF definitions that contain VHDL constructs that we have not yet defined. This is often frustrating. You can use the book index and the table of important VHDL constructs at the end of this chapter (Table 10.28) to help find definitions if you need them.

We shall occasionally refer to the VHDL BNF syntax definitions in this chapter using references—BNF [10.1], for example. Only the most important BNF constructs for VHDL are included here in this chapter, but a complete description of the VHDL language syntax is contained in Appendix A.

10.4 Identifiers and Literals

Names (the “nouns” of VHDL) are known as **identifiers** [VHDL LRM13.3]. The correct “spelling” of an identifier is defined in BNF as follows:

```
identifier ::= [10.3]
    letter {[underline] letter_or_digit}
    | \graphic_character{graphic_character}\
```

In this book an underline in VHDL BNF marks items that are new or that have changed in VHDL-93 from VHDL-87. The following are examples of identifiers:

```
s -- A simple name.
S -- A simple name, the same as s. VHDL is not case sensitive.
a_name -- Imbedded underscores are OK.
-- Successive underscores are illegal in names: Ill__egal
-- Names can't start with underscore: _Illegal
-- Names can't end with underscore: Illegal_
Too_Good -- Names must start with a letter.
-- Names can't start with a number: 2_Bad
\74LS00\ -- Extended identifier to break rules (VHDL-93 only).
VHDL \vhdl\ \VHDL\ -- Three different names (VHDL-93 only).
s_array(0) -- A static indexed name (known at analysis time).
s_array(i) -- A non-static indexed name, if i is a variable.
```

You may not use a reserved word as a declared identifier, and it is wise not to use units, special characters, and function names: `ns`, `ms`, `FF`, `read`, `write`, and so on. You may attach qualifiers to names as follows [VHDL LRM6]:

```
CMOS.all -- A selected or expanded name, all units in library CMOS.
Data'LEFT(1) -- An attribute name, LEFT is the attribute designator.
Data(24 downto 1) -- A slice name, part of an array: Data(31 downto 0)
Data(1) -- An indexed name, one element of an array.
```

Comments follow two hyphens ‘--’ and instruct the analyzer to ignore the rest of the line. There are no multiline comments in VHDL. Tabs improve readability, but it is best not to rely on a tab as a space in case the tabs are lost or deleted in conversion. You should thus write code that is still legal if all tabs are deleted.

There are various forms of **literals** (fixed-value items) in VHDL [VHDL LRM13.4–13.7]. The following code shows some examples:

```
entity Literals_1 is end;
architecture Behave of Literals_1 is
begin process
    variable I1 : integer; variable R1 : real;
    variable C1 : CHARACTER; variable S16 : STRING(1 to 16);
    variable BV4 : BIT_VECTOR(0 to 3);
    variable BV12 : BIT_VECTOR(0 to 11);
    variable BV16 : BIT_VECTOR(0 to 15);
```

```

begin
-- Abstract literals are decimal or based literals.
-- Decimal literals are integer or real literals.
-- Integer literal examples (each of these is the same):
    I1 := 120000; I1 := 12e4; I1 := 120_000;
-- Based literal examples (each of these is the same):
    I1 := 2#1111_1111#; I1 := 16#FF#;
-- Base must be an integer from 2 to 16:
    I1 := 16:FF#; -- you may use a : if you don't have #
-- Real literal examples (each of these is the same):
    R1 := 120000.0; R1 := 1.2e5; R1 := 12.0E4;
-- Character literal must be one of the 191 graphic characters.
-- 65 of the 256 ISO Latin-1 set are non-printing control characters
    C1 := 'A'; C1 := 'a'; -- different from each other
-- String literal examples:
    S16 := " string" & " literal";    -- concatenate long strings
    S16 := ""Hello,"" I said!";      -- doubled quotes
    S16 := % string literal%;        -- can use % instead of "
    S16 := %Sale: 50%% off!!!%;      -- doubled %
-- Bit-string literal examples:
    BV4  := B"1100";    -- binary bit-string literal
    BV12 := O"7777";   -- octal  bit-string literal
    BV16 := X"FFFF";   -- hex    bit-string literal
wait; end process; -- the wait prevents an endless loop
end;

```

10.5 Entities and Architectures

The highest-level VHDL construct is the **design file** [VHDL LRM11.1]. A design file contains **design units** that contain one or more **library units**. Library units in turn contain: entity, configuration, and package declarations (**primary units**); and architecture and package bodies (**secondary units**).

```

design_file ::= [10.4]
    {library_clause|use_clause} library_unit
    {{library_clause|use_clause} library_unit}

library_unit ::= primary_unit|secondary_unit

primary_unit ::= [10.5]
    entity_declaration|configuration_declaration|package_declaration

secondary_unit ::= [10.6]
    architecture_body|package_body

```

Using the written language analogy: a VHDL library unit is a “book,” a VHDL design file is a “bookshelf,” and a VHDL library is a collection of bookshelves. A

VHDL primary unit is a little like the chapter title and contents that appear on the first page of each chapter in this book and a VHDL secondary unit is like the chapter contents (though this is stretching our analogy a little far).

I shall describe the very important concepts of entities and architectures in this section and then cover libraries, packages, and package bodies. You define an entity, a black box, using an **entity declaration** [VHDL LRM1.1]. This is the BNF definition:

```
entity_declaration ::= [10.7]
entity identifier is
    [generic (formal_generic_interface_list);]
    [port (formal_port_interface_list);]
    {entity_declarative_item}
    [begin
        {[label:] [postponed] assertion ;
        |[label:] [postponed] passive_procedure_call ;
        |passive_process_statement}]
    end [entity] [entity_identifier] ;
```

The following is an example of an entity declaration for a black box with two inputs and an output:

```
entity Half_Adder is
    port (X, Y : in BIT := '0'; Sum, Cout : out BIT); -- formals
end;
```

Matching the parts of this code with the constructs in BNF [10.7] you can see that the identifier is `Half_Adder` and that `(X, Y: in BIT := '0'; Sum, Cout: out BIT)` corresponds to `(port_interface_list)` in the BNF. The ports `X`, `Y`, `Sum`, and `Cout` are **formal ports** or **formals**. This particular entity `Half_Adder` does not use any of the other optional constructs that are legal in an entity declaration.

The **architecture body** [VHDL LRM1.2] describes what an entity does, or the contents of the black box (it is architecture body and not architecture declaration).

```
architecture_body ::= [10.8]
    architecture identifier of entity_name is
        {block_declarative_item}
        begin
            {concurrent_statement}
        end [architecture] [architecture_identifier] ;
```

For example, the following architecture body (I shall just call it an architecture from now on) describes the contents of the entity `Half_Adder`:

```
architecture Behave of Half_Adder is
    begin Sum <= X xor Y; Cout <= X and Y;
end Behave;
```

We use the same signal names (the formals: `Sum`, `X`, `Y`, and `Cout`) in the architecture as we use in the entity (we say the signals of the “parent” entity are **visible** inside the architecture “child”). An architecture can refer to other entity–architecture pairs—so we can nest black boxes. We shall often refer to an entity–architecture pair as `entity(architecture)`. For example, the architecture `Behave` of the entity `Half_Adder` is `Half_Adder(Behave)`.

Why would we want to describe the outside of a black box (an entity) separately from the description of its contents (its architecture)? Separating the two makes it easier to move between different architectures for an entity (there must be at least one). For example, one architecture may model an entity at a behavioral level, while another architecture may be a structural model.

A structural model that uses an entity in an architecture must declare that entity and its interface using a **component declaration** as follows [VHDL LRM4.5]:

```
component_declaration ::= [10.9]
  component identifier [is]
    [generic (local_generic_interface_list);]
    [port (local_port_interface_list);]
  end component [component_identifier];
```

For example, the following architecture, `Netlist`, is a structural version of the behavioral architecture, `Behave`:

```
architecture Netlist of Half_Adder is
  component MyXor port (A_Xor,B_Xor : in BIT; Z_Xor : out BIT);
end component; -- component with locals
  component MyAnd port (A_And,B_And : in BIT; Z_And : out BIT);
end component; -- component with locals
begin
  Xor1: MyXor port map (X, Y, Sum);    -- instance with actuals
  And1 : MyAnd port map (X, Y, Cout);  -- instance with actuals
end;
```

Notice that:

- We declare the components: `MyAnd`, `MyXor` and their **local ports** (or **locals**): `A_Xor`, `B_Xor`, `Z_Xor`, `A_And`, `B_And`, `Z_And`.
- We instantiate the components with **instance names**: `And1` and `Xor1`.
- We connect instances using **actual ports** (or **actuals**): `X`, `Y`, `Sum`, `Cout`.

Next we define the entities and architectures that we shall use for the components `MyAnd` and `MyXor`. You can think of an entity–architecture pair (and its formal ports) as a data-book specification for a logic cell; the component (and its local ports) corresponds to a software model for the logic cell; and an instance (and its actual ports) is the logic cell.

We do not need to write VHDL code for MyAnd and MyXor; the code is provided as a **technology library** (also called an **ASIC vendor library** because it is often sold or distributed by the ASIC company that will manufacture the chip—the ASIC vendor—and not the software company):

```
-- These definitions are part of a technology library:
entity AndGate is
  port (And_in_1, And_in_2 : in BIT; And_out : out BIT); -- formals
end;

architecture Simple of AndGate is
  begin And_out <= And_in_1 and And_in_2;
end;

entity XorGate is
  port (Xor_in_1, Xor_in_2 : in BIT; Xor_out : out BIT); -- formals
end;

architecture Simple of XorGate is
  begin Xor_out <= Xor_in_1 xor Xor_in_2;
end;
```

If we keep the description of a circuit's interface (the entity) separate from its contents (the architecture), we need a way to link or **bind** them together. A **configuration declaration** [VHDL LRM1.3] binds entities and architectures.

```
configuration_declaration ::= [10.10]
  configuration identifier of entity_name is
    {use_clause|attribute_specification|group_declaration}
    block_configuration
  end [configuration] [configuration_identifier] ;
```

An entity–architecture pair is a **design entity**. The following configuration declaration defines which design entities we wish to use and associates the formal ports (from the entity declaration) with the local ports (from the component declaration):

```
configuration Simplest of Half_Adder is
use work.all;
  for Netlist
    for And1 : MyAnd use entity AndGate(Simple)
      port map -- association: formals => locals
        (And_in_1 => A_And, And_in_2 => B_And, And_out => Z_And);
    end for;
    for Xor1 : MyXor use entity XorGate(Simple)
      port map
        (Xor_in_1 => A_Xor, Xor_in_2 => B_Xor, Xor_out => Z_Xor);
    end for;
  end for;
end;
```

Figure 10.1 diagrams the use of entities, architectures, components, and configurations. This figure seems very complicated, but there are two reasons that VHDL works this way:

- Separating the entity, architecture, component, and configuration makes it easier to reuse code and change libraries. All we have to do is change names in the port maps and configuration declaration.
- We only have to alter and reanalyze the configuration declaration to change which architectures we use in a model—giving us a fast debug cycle.

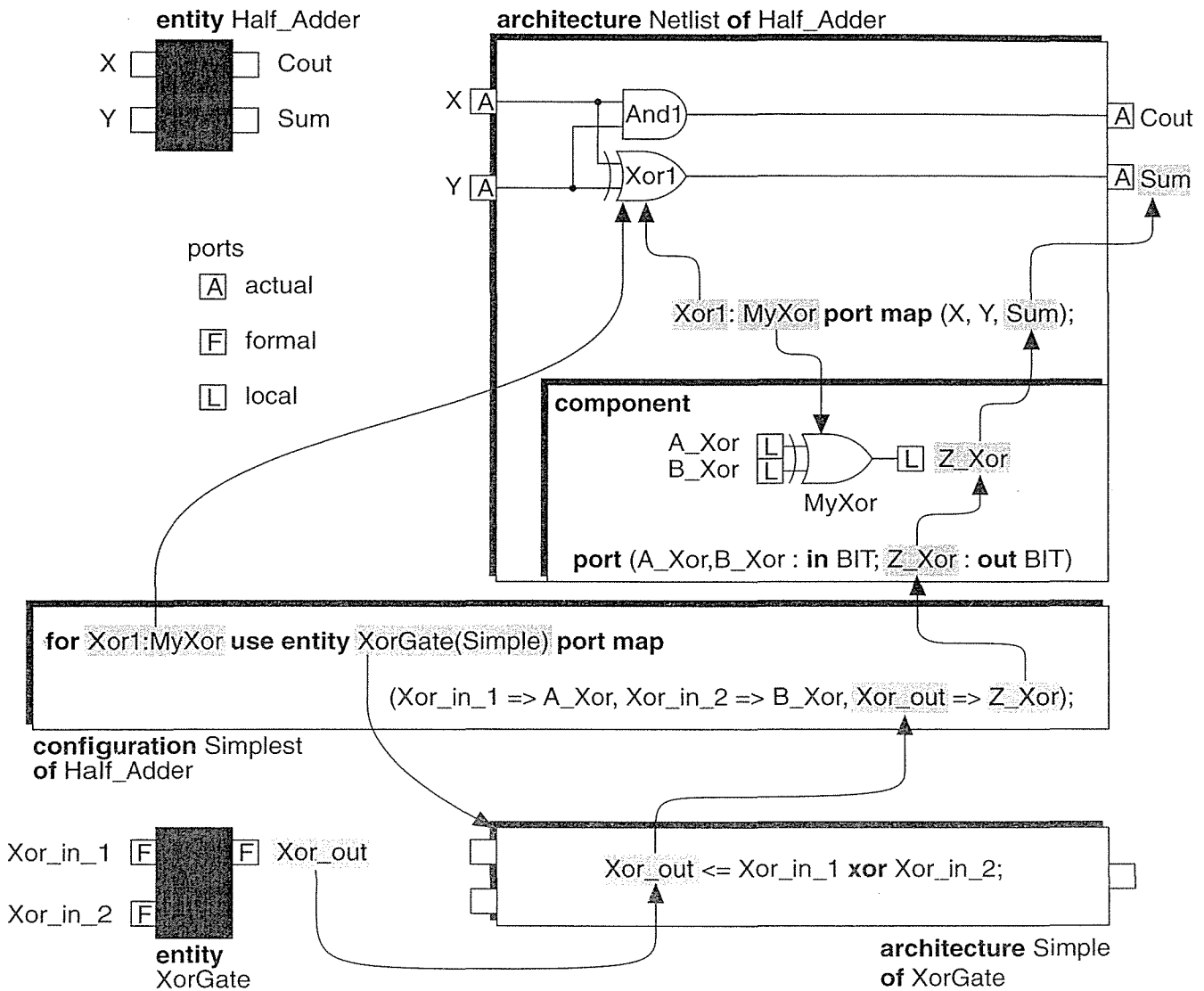


FIGURE 10.1 Entities, architectures, components, ports, port maps, and configurations.

You can think of design units, the analyzed entity–architecture pairs, as compiled object-code modules. The configuration then determines which object-code modules are linked together to form executable binary code.

You may also think of an entity as a block diagram, an architecture for an entity a more detailed circuit schematic for the block diagram, and a configuration as a parts list of the circuit components with their part numbers and manufacturers (also known as a **BOM** for **bill of materials**, rather like a shopping list). Most manufacturers (including the U.S. DoD) use schematics and BOMs as control documents for electronic systems. This is part of the rationale behind the structure of VHDL.

10.6 Packages and Libraries

After the VHDL tool has analyzed entities, architectures, and configurations, it stores the resulting design units in a library. Much of the power of VHDL comes from the use of predefined libraries and packages. A VHDL **design library** [VHDL LRM11.2] is either the current working library (things we are currently analyzing) or a predefined resource library (something we did yesterday, or we bought, or that came with the tool). The **working library** is named `work` and is the place where the code currently being analyzed is stored. Architectures must be in the same library (but they do not have to be in the same physical file on disk) as their parent entities.

You can use a VHDL **package** [VHDL LRM2.5–2.6] to define subprograms (procedures and functions), declare special types, modify the behavior of operators, or to hide complex code. Here is the BNF for a package declaration:

```
package_declaration ::= [10.11]
package identifier is
{subprogram_declaration | type_declaration | subtype_declaration
 | constant_declaration | signal_declaration | file_declaration
 | alias_declaration | component_declaration
 | attribute_declaration | attribute_specification
 | disconnection_specification | use_clause
 | shared_variable_declaration | group_declaration
 | group_template_declaration}
end [package] [package_identifier] ;
```

You need a **package body** if you declare any subprograms in the package declaration (a package declaration and its body do not have to be in the same file):

```
package_body ::=
package body package_identifier is
{subprogram_declaration | subprogram_body
 | type_declaration | subtype_declaration
 | constant_declaration | file_declaration | alias_declaration
 | use_clause
```

```

| shared variable declaration | group declaration
| group template declaration}
end [package body] [package identifier] ;

```

To make a package **visible** [VHDL LRM10.3] (or accessible, so you can see and use the package and its contents), you must include a **library clause** before a design unit and a **use clause** either before a design unit or inside a unit, like this:

```

library MyLib; -- library clause
use MyLib.MyPackage.all; -- use clause
-- design unit (entity + architecture, etc.) follows:

```

The STD and WORK libraries and the STANDARD package are always visible. Things that are visible to an entity are visible to its architecture bodies.

10.6.1 Standard Package

The VHDL **STANDARD package** [VHDL LRM14.2] is defined in the LRM and implicitly declares the following implementation dependent types: TIME, INTEGER, REAL. We shall use uppercase for types defined in an IEEE standard package. Here is part of the STANDARD package showing the explicit type and subtype declarations:

```

package Part_STANDARD is
type BOOLEAN is (FALSE, TRUE); type BIT is ('0', '1');
type SEVERITY_LEVEL is (NOTE, WARNING, ERROR, FAILURE);
subtype NATURAL is INTEGER range 0 to INTEGER'HIGH;
subtype POSITIVE is INTEGER range 1 to INTEGER'HIGH;
type BIT_VECTOR is array (NATURAL range <>) of BIT;
type STRING is array (POSITIVE range <>) of CHARACTER;
-- the following declarations are VHDL-93 only:
attribute FOREIGN: STRING; -- for links to other languages
subtype DELAY_LENGTH is TIME range 0 fs to TIME'HIGH;
type FILE_OPEN_KIND is (READ_MODE,WRITE_MODE,APPEND_MODE);
type FILE_OPEN_STATUS is
(OPEN_OK,STATUS_ERROR,NAME_ERROR,MODE_ERROR);
end Part_STANDARD;

```

Notice that a STRING array must have a positive index. The type TIME is declared in the STANDARD package as follows:

```

type TIME is range implementation_defined -- and varies with software
  units fs; ps = 1000 fs; ns = 1000 ps; us = 1000 ns; ms = 1000 us;
  sec = 1000 ms; min = 60 sec; hr = 60 min; end units;

```

The STANDARD package also declares the function now that returns the current simulation time (with type TIME in VHDL-87 and subtype DELAY_LENGTH in VHDL-93).

In VHDL-93 the CHARACTER type declaration extends the VHDL-87 declaration (the 128 ASCII characters):

```

type Part_CHARACTER is ( -- 128 ASCII characters in VHDL-87
  NUL, SOH, STX, ETX, EOT, ENQ, ACK, BEL, -- 33 control characters
  BS, HT, LF, VT, FF, CR, SO, SI, -- including:
  DLE, DC1, DC2, DC3, DC4, NAK, SYN, ETB, -- format effectors:
  CAN, EM, SUB, ESC, FSP, GSP, RSP, USP, -- horizontal tab = HT
  ' ', '!', '"', '#', '$', '%', '&', '\', -- line feed = LF
  '(', ')', '*', '+', ',', '-', '.', '/', -- vertical tab = VT
  '0', '1', '2', '3', '4', '5', '6', '7', -- form feed = FF
  '8', '9', ':', ';', '<', '=', '>', '?', -- carriage return = CR
  '@', 'A', 'B', 'C', 'D', 'E', 'F', 'G', -- and others:
  'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', -- FSP, GSP, RSP, USP use P
  'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', -- suffix to avoid conflict
  'X', 'Y', 'Z', '[', '\', ']', '^', '_', -- with TIME units
  '`', 'a', 'b', 'c', 'd', 'e', 'f', 'g',
  'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o',
  'p', 'q', 'r', 's', 't', 'u', 'v', 'w',
  'x', 'y', 'z', '{', '|', '}', '~', DEL -- delete = DEL
);
-- VHDL-93 includes 96 more Latin-1 characters, like ¥ (Yen) and
-- 32 more control characters, better not to use any of them.

```

The VHDL-87 character set is the 7-bit coded **ISO 646-1983** standard known as the **ASCII character set**. Each of the printable ASCII graphic **character codes** (there are 33 nonprintable control codes, like DEL for delete) is represented by a **graphic symbol** (the shapes of letters on the keyboard, on the display, and that actually print). VHDL-93 uses the 8-bit coded character set **ISO 8859-1:1987(E)**, known as **ISO Latin-1**. The first 128 characters of the 256 characters in ISO Latin-1 correspond to the 128-character ASCII code. The graphic symbols for the printable ASCII characters are well defined, but not part of the standard (for example, the shape of the graphic symbol that represents 'lowercase a' is recognizable on every keyboard, display, and font). However, the graphic symbols that represent the printable characters from other 128-character codes of the ISO 8-bit character set are different in various fonts, languages, and computer systems. For example, a pound sterling sign in a U.K. character set looks like this—'£', but in some fonts the same character code prints as '#' (known as number sign, hash, or pound). If you use such characters and want to share your models with people in different countries, this can cause problems (you can see all 256 characters in a character set by using Insert... Symbol in MS Word).

10.6.2 Std_logic_1164 Package

VHDL does not have a built-in logic-value system. The STANDARD package pre-defines the type BIT with two logic values, '0' and '1', but we normally need at

least two more values: 'x' (unknown) and 'z' (high-impedance). Unknown is a **metalogical value** because it does not exist in real hardware but is needed for simulation purposes. We could define our own logic-value system with four logic values:

```
type MVL4 is ('X', '0', '1', 'Z'); -- a four-value logic system
```

The proliferation of VHDL logic-value systems prompted the creation of the **Std_logic_1164 package** (defined in IEEE Std 1164-1993) that includes functions to perform logical, shift, resolution, and conversion functions for types defined in the Std_logic_1164 system. To use this package in a design unit, you must include the following library clause (before each design unit) and a use clause (either before or inside the unit):

```
library IEEE; use IEEE.std_logic_1164.all;
```

This Std_Logic_1164 package contains definitions for a nine-value logic system. The following code and comments show the definitions and use of the most important parts of the package²:

```
package Part_STD_LOGIC_1164 is --1
type STD_ULOGIC is --2
( 'U', -- Uninitialized --3
  'X', -- Forcing Unknown --4
  '0', -- Forcing 0 --5
  '1', -- Forcing 1 --6
  'Z', -- High Impedance --7
  'W', -- Weak Unknown --8
  'L', -- Weak 0 --9
  'H', -- Weak 1 --10
  '-' -- Don't Care); --11
type STD_ULOGIC_VECTOR is array (NATURAL range <>) of STD_ULOGIC; --12
function resolved (s : STD_ULOGIC_VECTOR) return STD_ULOGIC; --13
subtype STD_LOGIC is resolved STD_ULOGIC; --14
type STD_LOGIC_VECTOR is array (NATURAL range <>) of STD_LOGIC; --15
subtype X01 is resolved STD_ULOGIC range 'X' to '1'; --16
subtype X01Z is resolved STD_ULOGIC range 'X' to 'Z'; --17
subtype UX01 is resolved STD_ULOGIC range 'U' to '1'; --18
subtype UX01Z is resolved STD_ULOGIC range 'U' to 'Z'; --19
-- Vectorized overloaded logical operators: --20
function "and" (L : STD_ULOGIC; R : STD_ULOGIC) return UX01; --21
-- Logical operators not, and, nand, or, nor, xor, xnor (VHDL-93), --22
-- overloaded for STD_ULOGIC STD_ULOGIC_VECTOR STD_LOGIC_VECTOR. --23
-- Strength strippers and type conversion functions: --24
-- function To_T (X : F) return T; --25
-- defined for types, T and F, where --26
-- F=BIT BIT_VECTOR STD_ULOGIC STD_ULOGIC_VECTOR STD_LOGIC_VECTOR --27
-- T=types F plus types X01 X01Z UX01 (but not type UX01Z) --28
```

²The code in this section is adapted with permission from IEEE Std 1164-1993, © Copyright IEEE. All rights reserved.

```

-- Exclude _'s in T in name: TO_STDULOGIC not TO_STD_ULOGIC           --29
-- To_X01 : L->0, H->1 others->X                                       --30
-- To_X01Z: Z->Z, others as To_X01                                     --31
-- To_UX01: U->U, others as To_X01                                     --32

-- Edge detection functions:                                           --33
function rising_edge (signal s: STD_ULOGIC) return BOOLEAN;          --34
function falling_edge (signal s: STD_ULOGIC) return BOOLEAN;         --35

-- Unknown detection (returns true if s = U, X, Z, W):                 --36
-- function Is_X (s : T) return BOOLEAN;                               --37
-- defined for T = STD_ULOGIC STD_ULOGIC_VECTOR STD_LOGIC_VECTOR.    --38

end Part_STD_LOGIC_1164;                                              --39

```

Notice:

- The type `STD_ULOGIC` has nine logic values. For this reason IEEE Std 1164 is sometimes referred to as MVL9—multivalued logic nine. There are simpler, but nonstandard, MVL4 and MVL7 packages, as well as packages with more than nine logic values, available. Values 'U', 'X', and 'W' are all metalogical values.
- There are weak and forcing logic-value strengths. If more than one logic gate drives a node (there is more than one **driver**) as in wired-OR logic or a three-state bus, for example, the simulator checks the driver strengths to **resolve** the actual logic value of the node using the **resolution function**, `resolved`, defined in the package.
- The subtype `STD_LOGIC` is the **resolved** version of the **unresolved** type `STD_ULOGIC`. Since subtypes are **compatible** with types (you can assign one to the other) you can use either `STD_LOGIC` or `STD_ULOGIC` for a signal with a single driver, but it is generally safer to use `STD_LOGIC`.
- The type `STD_LOGIC_VECTOR` is the resolved version of unresolved type `STD_ULOGIC_VECTOR`. Since these are two different types and are not compatible, you should use `STD_LOGIC_VECTOR`. That way you will not run into a problem when you try to connect a `STD_LOGIC_VECTOR` to a `STD_ULOGIC_VECTOR`.
- The don't care logic value '-' (hyphen), is principally for use by synthesis tools. The value '-' is almost always treated the same as 'X'.
- The 1164 standard defines (or **overloads**) the logical operators for the `STD_LOGIC` types but not the arithmetic operators (see Section 10.12).

10.6.3 TEXTIO Package

You can use the `TEXTIO` package, which is part of the library `STD`, for text input and output [VHDL LRM14.3]. The following code is a part of the `TEXTIO` package

header and, together with the comments, shows the declarations of types, subtypes, and the use of the procedures in the package:

```

package Part_TEXTIO is          -- VHDL-93 version.
type LINE is access STRING;    -- LINE is a pointer to a STRING value.
type TEXT is file of STRING;   -- File of ASCII records.
type SIDE is (RIGHT, LEFT);    -- for justifying output data.
subtype WIDTH is NATURAL;      -- for specifying widths of output fields.
file INPUT : TEXT open READ_MODE is "STD_INPUT"; -- Default input file.
file OUTPUT : TEXT open WRITE_MODE is "STD_OUTPUT"; -- Default output.

-- The following procedures are defined for types, T, where
-- T = BIT BIT_VECTOR BOOLEAN CHARACTER INTEGER REAL TIME STRING
-- procedure READLINE(file F : TEXT; L : out LINE);
-- procedure READ(L : inout LINE; VALUE : out T);
-- procedure READ(L : inout LINE; VALUE : out T; GOOD: out BOOLEAN);
-- procedure WRITELINE(F : out TEXT; L : inout LINE);
-- procedure WRITE(
--     L : inout LINE;
--     VALUE : in T;
--     JUSTIFIED : in SIDE:= RIGHT;
--     FIELD:in WIDTH := 0;
--     DIGITS:in NATURAL := 0; -- for T = REAL only
--     UNIT:in TIME:= ns);    -- for T = TIME only
-- function ENDFILE(F : in TEXT) return BOOLEAN;

end Part_TEXTIO;

```

Here is an example that illustrates how to write to the screen (STD_OUTPUT):

```

library std; use std.textio.all; entity Text is end;
architecture Behave of Text is signal count : INTEGER := 0;
begin count <= 1 after 10 ns, 2 after 20 ns, 3 after 30 ns;
process (count) variable L: LINE; begin
if (count > 0) then
    write(L, now);                -- Write time.
    write(L, STRING'(" count=")); -- STRING' is a type qualification.
    write(L, count); writeline(output, L);
end if; end process; end;

10 ns count=1
20 ns count=2
30 ns count=3

```

10.6.4 Other Packages

VHDL does not predefine arithmetic operators on types that hold bits. Many VHDL simulators provide one or more **arithmetic packages** that allow you to perform arithmetic operations on `std_logic_1164` types. Some companies also provide one

or more **math packages** that contain functions for floating-point algebra, trigonometry, complex algebra, queueing, and statistics (see also [IEEE 1076.2, 1996]).

Synthesis tool companies often provide a special version of an arithmetic package, a **synthesis package**, that allows you to synthesize VHDL that includes arithmetic operators. This type of package may contain special instructions (normally comments that are recognized by the synthesis software) that map common functions (adders, subtractors, multipliers, shift registers, counters, and so on) to ASIC library cells. I shall introduce the IEEE synthesis package in Section 10.12.

Synthesis companies may also provide **component packages** for such cells as power and ground pads, I/O buffers, clock drivers, three-state pads, and bus keepers. These components may be technology-independent (generic) and are mapped to primitives from technology-dependent libraries after synthesis.

10.6.5 Creating Packages

It is often useful to define constants in one central place rather than using literals wherever you need a specific value in your code. One way to do this is by using VHDL **packaged constants** [VHDL LRM4.3.1.1] that you define in a package. Packages that you define are initially part of the working library, *work*. Here are two example packages [VHDL LRM2.5–2.7]:

```
package Adder_Pkg is -- a package declaration
    constant BUSWIDTH : INTEGER := 16;
end Adder_Pkg;

use work.Adder_Pkg.all; -- a use clause
entity Adder is end Adder;
architecture Flexible of Adder is -- work.Adder_Pkg is visible here
    begin process begin
        MyLoop : for j in 0 to BUSWIDTH loop -- adder code goes here
            end loop; wait; -- the wait prevents an endless cycle
        end process;
end Flexible;

package GLOBALS is
    constant HI : BIT := '1'; constant LO: BIT := '0';
end GLOBALS;
```

Here is a package that declares a function and thus requires a package body:

```
package Add_Pkg_Fn is
function add(a, b, c : BIT_VECTOR(3 downto 0)) return BIT_VECTOR;
end Add_Pkg_Fn;

package body Add_Pkg_Fn is
function add(a, b, c : BIT_VECTOR(3 downto 0)) return BIT_VECTOR is
    begin return a xor b xor c; end;
end Add_Pkg_Fn;
```

The following example is similar to the **VITAL (VHDL Initiative Toward ASIC Libraries)** package that provides two alternative methods (procedures or functions) to model primitive gates (I shall describe functions and procedures in more detail in Section 10.9.2):

```
package And_Pkg is
  procedure V_And(a, b : BIT; signal c : out BIT);
  function V_And(a, b : BIT) return BIT;
end;

package body And_Pkg is
  procedure V_And(a, b : BIT; signal c : out BIT) is
    begin c <= a and b; end;
  function V_And(a, b : BIT) return BIT is
    begin return a and b; end;
end And_Pkg;
```

The software determines where it stores the design units that we analyze. Suppose the package `Add_Pkg_Fn` is in library `MyLib`. Then we need a library clause (before each design unit) and use clause with a selected name to use the package:

```
library MyLib; -- use MyLib.Add_Pkg.all; -- use all the package
use MyLib.Add_Pkg_Fn.add; -- just function 'add' from the package

entity Lib_1 is port (s : out BIT_VECTOR(3 downto 0) := "0000"); end;
architecture Behave of Lib_1 is begin process
begin s <= add ("0001", "0010", "1000"); wait; end process; end;
```

The VHDL software dictates how you create the library `MyLib` from the library work and the actual name and directory location for the physical file or directory on the disk that holds the library. The mechanism to create the links between the file and directory names in the computer world and the library names in the VHDL world depends on the software. There are three common methods:

- Use a UNIX environment variable (`SETENV MyLib ~/MyDirectory/MyLibFile`, for example).
- Create a separate file that establishes the links between the filename known to the operating system and the library name known to the VHDL software.
- Include the links in an initialization file (often with an `.ini` suffix).

10.7 Interface Declarations

An **interface declaration** declares **interface objects** that may be interface constants, signals, variables, or files [VHDL 87LRM4.3.3, 93LRM4.3.2]. **Interface constants** are generics of a design entity, a component, or a block, or parameters of subprograms. **Interface signals** are ports of a design entity, component, or block,

and parameters of subprograms. **Interface variables** and **interface files** are parameters of subprograms.

Each interface object has a **mode** that indicates the direction of information flow. The most common modes are `in` (the default), `out`, `inout`, and `buffer` (a fifth mode, `linkage`, is used to communicate with other languages and is infrequently used in ASIC design). The restrictions on the use of objects with these modes are listed in Table 10.10. An interface object is **read** when you use it on the RHS of an assignment statement, for example, or when the object is associated with another interface object of modes `in`, `inout` (or `linkage`). An interface object is **updated** when you use it on the LHS side of an assignment statement or when the object is associated with another interface object of mode `out`, `buffer`, `inout` (or `linkage`). The restrictions on reading and updating objects generate the diagram at the bottom of Table 10.10 that shows the 10 allowed types of interconnections (these rules for modes `buffer` and `inout` are the same). The interface objects (Inside and Outside) in the example in this table are ports (and thus interface signals), but remember that interface objects may also be interface constants, variables, and files.

There are other special-case rules for reading and updating interface signals, constants, variables, and files that I shall cover in the following sections. The situation is like the spelling rule, “i before e except after c.” Table 10.10 corresponds to the rule “i before e.”

10.7.1 Port Declaration

Interface objects that are signals are called **ports** [VHDL 93LRM1.1.1.2]. You may think of ports as “connectors” and you must declare them as follows:

```
port (port_interface_list)
interface_list ::= [10.12]
    port_interface_declaration {; port_interface_declaration}
```

A **port interface declaration** is a list of ports that are the inputs and outputs of an entity, a block, or a component declaration:

```
interface_declaration ::= [10.13]
    [signal]
    identifier {, identifier}:[in|out|inout|buffer|linkage]
    subtype_indication [bus] [:= static_expression]
```

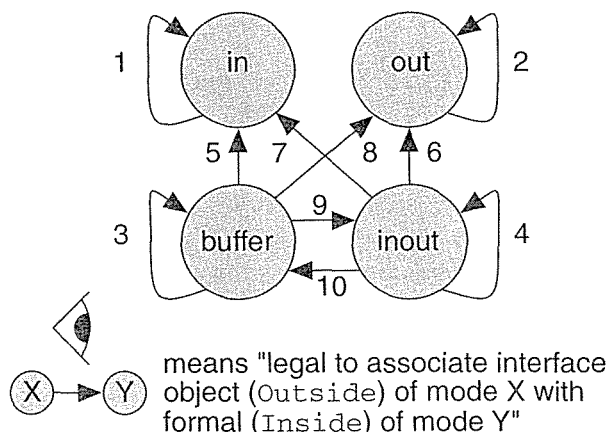
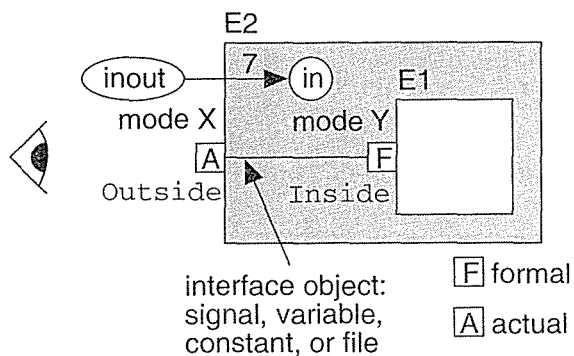
Each port forms an **implicit signal declaration** and has a **port mode**. I shall discuss bus, which is a **signal kind**, in Section 10.13.1. Here is an example of an entity declaration that has five ports:

```
entity Association_1 is
    port (signal X, Y : in BIT := '0'; Z1, Z2, Z3 : out BIT);
end;
```

TABLE 10.10 Modes of interface objects and their properties.

```
entity E1 is port (Inside : in BIT); end; architecture Behave of E1 is begin end;
entity E2 is port (Outside : inout BIT := '1'); end; architecture Behave of E2 is
component E1 port (Inside: in BIT); end component; signal UpdateMe : BIT; begin
I1 : E1 port map (Inside => Outside); -- formal/local (mode in) => actual (mode inout)
UpdateMe <= Outside; -- OK to read Outside (mode inout)
Outside <= '0' after 10 ns; -- and OK to update Outside (mode inout)
end;
```

Possible modes of interface object, Outside	in (default)	out	inout	buffer
Can you read Outside (RHS of assignment)?	Yes	No	Yes	Yes
Can you update Outside (LHS of assignment)?	No	Yes	Yes	Yes
Modes of Inside that Outside may connect to (see below) ¹	in	out	any	any



¹There are additional rules for interface objects that are signals (ports)—see Tables 10.11 and 10.12.

In the preceding declaration the keyword `signal` is redundant (because all ports are signals) and may be omitted. You may also omit the port mode `in` because it is the default mode. In this example, the input ports `x` and `y` are driven by a **default value** (in general a **default expression**) of `'0'` if (and only if) the ports are left unconnected or **open**. If you do leave an input port open, the port must have a default expression.

You use a **port map** and either **positional association** or **named association** to connect the formals of an entity with the locals of a component. Port maps also associate (connect) the locals of a component with the actuals of an instance. For an example of formal, local, and actual ports, and explanation of their function, see Section 10.5, where we declared an entity `AndGate`. The following example shows

how to bind a component to the entity `AndGate` (in this case we use the **default binding**) and associate the ports. Notice that if we mix positional and named association then all positional associations must come first.

```

use work.all; -- makes analyzed design entity AndGate(Simple) visible.
architecture Netlist of Association_1 is
-- The formal port clause for entity AndGate looks like this:
-- port (And_in_1, And_in_2: in BIT; And_out : out BIT); -- Formals.
component AndGate port
    (And_in_1, And_in_2 : in BIT; And_out : out BIT); -- Locals.
end component;
begin
-- The component and entity have the same names: AndGate.
-- The port names are also the same: And_in_1, And_in_2, And_out,
-- so we can use default binding without a configuration.
-- The last (and only) architecture for AndGate will be used: Simple.
A1:AndGate port map (X, Y, Z1); -- positional association
A2:AndGate port map (And_in_2=>Y, And_out=>Z2, And_in_1=>X); -- named
A3:AndGate port map (X, And_out => Z3, And_in_2 => Y); -- both
end;

```

The interface object rules of Table 10.10 apply to ports. The rule that forbids updating an interface object of mode `in` prevents modifying an input port (by placing the input signal on the left-hand side of an assignment statement, for example). Less obviously, you cannot read a port of mode `out` (that is you cannot place an output signal on the right-hand side of an assignment statement). This stops you from accidentally reading an output signal that may be connected to a net with multiple drivers. In this case the value you would read (the unresolved output signal) might not be the same as the resolved signal value. For example, in the following code, since `Clock` is a port of mode `out`, you cannot read `Clock` directly. Instead you can transfer `Clock` to an intermediate variable and read the intermediate variable instead:

```

entity ClockGen_1 is port (Clock : out BIT); end;
architecture Behave of ClockGen_1 is
begin process variable Temp : BIT := '1';
    begin
-- Clock <= not Clock;    -- Illegal, you cannot read Clock (mode out),
    Temp := not Temp;    -- use a temporary variable instead.
    Clock <= Temp after 10 ns; wait for 10 ns;
    if (now > 100 ns) then wait; end if; end process;
end;

```

Table 10.10 lists the restrictions on reading and updating interface objects including interface signals that form ports. Table 10.11 lists additional special rules for reading and updating the attributes of interface signals.

TABLE 10.11 Properties of ports.

Example entity declaration:

```
entity E is port (F_1:BIT; F_2:out BIT; F_3:inout BIT; F_4:buffer BIT); end; -- formals
```

Example component declaration:

```
component C port (L_1:BIT; L_2:out BIT; L_3:inout BIT; L_4:buffer BIT); -- locals
end component;
```

Example component instantiation:

```
I1 : C port map
(L_1 => A_1, L_2 => A_2, L_3 => A_3, L_4 => A_4); -- locals => actuals
```

Example configuration:

```
for I1 : C use entity E(Behave) port map
(F_1 => L_1, F_2 => L_2, F_3 => L_3, F_4 => L_4); -- formals => locals
```

Interface object, port F	F_1	F_2	F_3	F_4
Mode of F	in (default)	out	inout	buffer
Can you read attributes of F? [VHDL LRM4.3.2]	Yes, but not the attributes: 'STABLE 'QUIET 'DELAYED 'TRANSACTION	Yes, but not the attributes: 'STABLE 'QUIET 'DELAYED 'TRANSACTION 'EVENT 'ACTIVE 'LAST_EVENT 'LAST_ACTIVE 'LAST_VALUE	Yes, but not the attributes: 'STABLE 'QUIET 'DELAYED 'TRANSACTION	Yes

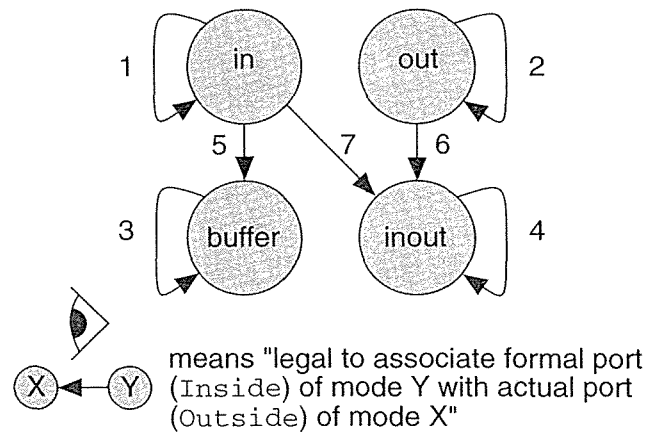
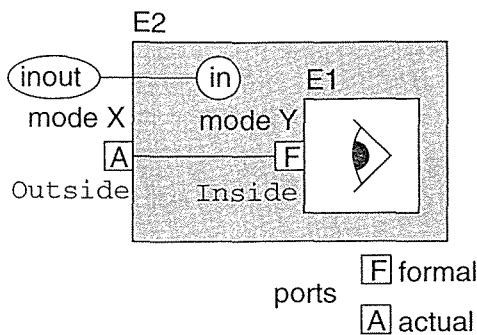
There is one more set of rules that apply to port connections [VHDL LRM 1.1.1.2]. If design entity E2 contains an instance, I1, of design entity E1, then the formals (of design entity E1) are associated with actuals (of instance I1). The actuals (of instance I1) are themselves formal ports (of design entity E2). The restrictions illustrated in Table 10.12 apply to the modes of the port connections from E1 to E2 (looking from the inside to the outside).

Notice that the allowed connections diagrammed in Table 10.12 (looking from inside to the outside) are a superset of those of Table 10.10 (looking from the outside to the inside). Only the seven types of connections shown in Table 10.12 are allowed between the ports of nested design entities. The additional rule that ports of mode `buffer` may only have one source, together with the restrictions on port mode interconnections, limits the use of ports of mode `buffer`.

TABLE 10.12 Connection rules for port modes.

```
entity E1 is port (Inside : in BIT); end; architecture Behave of E1 is begin end;
entity E2 is port (Outside : inout BIT := '1'); end; architecture Behave of E2 is
component E1 port (Inside : in BIT); end component; begin
I1 : E1 port map (Inside => Outside); -- formal/local (mode in) => actual (mode inout)
end;
```

Possible modes of interface object, Inside	in (default)	out	inout	buffer
Modes of Outside that Inside may connect to (see below)	in	inout	out	buffer
	buffer	inout	inout ¹	buffer ²



¹A signal of mode inout can be updated by any number of sources [VHDL 87LRM 4.3.3, 93LRM4.3.2].

²A signal of mode buffer can be updated by at most one source [VHDL LRM1.1.1.2].

10.7.2 Generics

Ports are signals that carry changing information between entities. A **generic** is similar to a port, except generics carry constant, static information. A generic is an interface constant that, unlike normal VHDL constants, may be given a value in a component instantiation statement or in a configuration specification. You declare generics in an entity declaration and you use generics in a similar fashion to ports. The following example uses a generic parameter to alter the size of a gate:

```
entity AndGateNWide is
  generic (N : NATURAL := 2);
  port (Inputs : BIT_VECTOR(1 to N); Result : out BIT);
end;
```

Notice that the **generic interface list** precedes the port interface list. Generics are useful to carry timing (delay) information, as in the next example:

```
entity AndT is
  generic (TPD : TIME := 1 ns);
  port (a, b : BIT := '0'; q: out BIT);
```

```

end;
architecture Behave of AndT is
  begin q <= a and b after TPD;
end;

entity AndT_Test_1 is end;
architecture Netlist_1 of AndT_Test_1 is
  component MyAnd
    port (a, b : BIT; q : out BIT);
  end component;
  signal a1, b1, q1 : BIT := '1';
  begin
    And1 : MyAnd port map (a1, b1, q1);
  end Netlist_1;

configuration Simplest_1 of AndT_Test_1 is use work.all;
  for Netlist_1 for And1 : MyAnd
    use entity AndT(Behave) generic map (2 ns);
  end for; end for;
end Simplest_1;

```

The configuration declaration, `Simplest_1`, changes the default delay (equal to 1ns, declared as a default expression in the entity) to 2ns. Techniques based on this method are useful in ASIC design. Prelayout simulation uses the default timing values. Back-annotation alters the delay in the configuration for postlayout simulation. When we change the delay we only need to reanalyze the configuration, not the rest of the ASIC model.

There was initially no standard in VHDL for how timing generics should be used, and the lack of a standard was a major problem for ASIC designers. The IEEE 1076.4 VITAL standard addresses this problem (see Section 13.5.5).

10.8 Type Declarations

In some programming languages you must declare objects to be integer, real, Boolean, and so on. VHDL (and ADA, the DoD programming language to which VHDL is related) goes further: You must declare the **type** of an object, and there are strict rules on mixing objects of different types. We say VHDL is strongly typed. For example, you can use one type for temperatures in Centigrade and a different type for Fahrenheit, even though both types are real numbers. If you try to add a temperature in Centigrade to a temperature in Fahrenheit, VHDL catches your error and tells you that you have a type mismatch.

This is the formal (expanded) BNF definition of a **type declaration**:

```

type_declaration ::= [10.14]
  type identifier ;
| type identifier is

```

```

(identifier|'graphic_character' {, identifier|'graphic_character'}) ;
| range_constraint ;          | physical_type_definition ;
| record_type_definition ;   | access_subtype_indication ;
| file_of_type_name ;        | file_of_subtype_name ;
| array_index_constraint_of_element_subtype_indication ;
| array
  (type_name|subtype_name range <>
   {, type_name|subtype_name range <>}) of
   element_subtype_indication ;

```

There are four **type classes** in VHDL [VHDL LRM3]: **scalar types**, **composite types**, **access types**, and **file types**. The scalar types are: **integer type**, **floating-point type**, **physical type**, and **enumeration type**. Integer and enumeration types are **discrete types**. Integer, floating-point, and physical types are **numeric types**. The **range** of an integer is implementation dependent but is guaranteed to include -2147483647 to $+2147483647$. Notice the integer range is symmetric and equal to $-(2^{31}-1)$ to $(2^{31}-1)$. Floating-point size is implementation dependent, but the range includes the bounds $-1.0E38$ and $+1.0E38$, and must include a minimum of six decimal digits of precision. Physical types correspond to time, voltage, current, and so on and have dimensions—a unit of measure (seconds, for example). Access types are pointers, useful in abstract data structures, but less so in ASIC design. File types are used for file I/O.

You may also declare a subset of an existing type, known as a **subtype**, in a **subtype declaration**. We shall discuss the different treatment of types and subtypes in expressions in Section 10.12.

Here are some examples of scalar type [VHDL LRM4.1] and subtype declarations [VHDL LRM4.2]:

```

entity Declaration_1 is end; architecture Behave of Declaration_1 is
type F is range 32 to 212; -- Integer type, ascending range.
type C is range 0 to 100; -- Range 0 to 100 is the range constraint.
subtype G is INTEGER range 9 to 0; -- Base type INTEGER, descending.
-- This is illegal: type Bad100 is INTEGER range 0 to 100;
-- don't use INTEGER in declaration of type (but OK in subtype).
type Rainbow is (R, O, Y, G, B, I, V); -- An enumeration type.
-- Enumeration types always have an ascending range.
type MVL4 is ('X', '0', '1', 'Z');
-- Note that 'X' and 'x' are different character literals.
-- The default initial value is MVL4'LEFT = 'X'.
-- We say '0' and '1' (already enumeration literals
-- for predefined type BIT) are overloaded.
-- Illegal enumeration type: type Bad4 is ("X", "0", "1", "Z");
-- Enumeration literals must be character literals or identifiers.
begin end;

```

The most common composite type is the **array type** [VHDL LRM3.2.1]. The following examples illustrate the semantics of array declarations:

```
entity Arrays_1 is end; architecture Behave of Arrays_1 is
type Word is array (0 to 31) of BIT; -- a 32-bit array, ascending
type Byte is array (NATURAL range 7 downto 0) of BIT; -- descending
type BigBit is array (NATURAL range <>) of BIT;
-- We call <> a box, it means the range is undefined for now.
-- We call BigBit an unconstrained array.
-- This is OK, we constrain the range of an object that uses
-- type BigBit when we declare the object, like this:
subtype Nibble is BigBit(3 downto 0);
type T1 is array (POSITIVE range 1 to 32) of BIT;
-- T1, a constrained array declaration, is equivalent to a type T2
-- with the following three declarations:
subtype index_subtype is POSITIVE range 1 to 32;
type array_type is array (index_subtype range <>) of BIT;
subtype T2 is array_type (index_subtype);
-- We refer to index_subtype and array_type as being
-- anonymous subtypes of T1 (since they don't really exist).
begin end;
```

You can assign values to an array using **aggregate notation** [VHDL LRM7.3.2]:

```
entity Aggregate_1 is end; architecture Behave of Aggregate_1 is
type D is array (0 to 3) of BIT; type Mask is array (1 to 2) of BIT;
signal MyData : D := ('0', others => '1'); -- positional aggregate
signal MyMask : Mask := (2 => '0', 1 => '1'); -- named aggregate
begin end;
```

The other composite type is the **record type** that groups elements together:

```
entity Record_2 is end; architecture Behave of Record_2 is
type Complex is record real : INTEGER; imag : INTEGER; end record;
signal s1 : Complex := (0, others => 1); signal s2: Complex;
begin s2 <= (imag => 2, real => 1); end;
```

10.9 Other Declarations

A declaration is one of the following [VHDL LRM4]:

```
declaration ::= [10.15]
  type_declaration      | subtype_declaration | object_declaration
| interface_declaration | alias_declaration   | attribute_declaration
| component_declaration | entity_declaration
| configuration_declaration
| subprogram_declaration | package_declaration
| group_template_declaration | group_declaration
```


I discussed entity, configuration, component, package, interface, type, and subtype declarations in Sections 10.5–10.8. Next I shall discuss the other types of declarations (except for groups or group templates [VHDL 93LRM4.6–4.7], new to VHDL-93, that are not often used in ASIC design).

10.9.1 Object Declarations

There are four **object classes** in VHDL: **constant**, **variable**, **signal**, and **file** [VHDL LRM 4.3.1.1–4.3.1.3]. You use a **constant declaration**, **signal declaration**, **variable declaration**, or **file declaration** together with a type. Signals can only be declared in the **declarative region** (before the first `begin`) of an architecture or block, or in a package (not in a package body). Variables can only be declared in the declarative region of a process or subprogram (before the first `begin`). You can think of signals as representing real wires in hardware. You can think of variables as memory locations in the computer. Variables are more efficient than signals because they require less overhead.

You may assign an (explicit) **initial value** when you declare a type. If you do not provide initial values, the (implicit) **default initial value** of a type or subtype `T` is `T'LEFT` (the leftmost item in the range of the type). For example:

```
entity Initial_1 is end; architecture Behave of Initial_1 is
type Fahrenheit is range 32 to 212;      -- Default initial value is 32.
type Rainbow is (R, O, Y, G, B, I, V);  -- Default initial value is R.
type MVL4 is ('X', '0', '1', 'Z');      -- MVL4'LEFT = 'X'.
begin end;
```

The details of initialization and assignment of initial values are important—it is difficult to implement the assignment of initial values in hardware—instead it is better to mimic the hardware and use explicit reset signals.

Here are the formal definitions of constant and signal declarations:

```
constant_declaration ::= constant [10.16]
identifier {, identifier}:subtype_indication [:= expression] ;
```

```
signal_declaration ::= signal [10.17]
identifier {, identifier}:subtype_indication [register|bus] [:=expression];
```

I shall explain the use of signals of kind `register` or `bus` in Section 10.13.1. Signal declarations are **explicit signal declarations** (ports declared in an interface declaration are implicit signal declarations). Here is an example that uses a constant and several signal declarations:

```
entity Constant_2 is end;
library IEEE; use IEEE.STD_LOGIC_1164.all;
architecture Behave of Constant_2 is
constant Pi : REAL := 3.14159;          -- A constant declaration.
signal B : BOOLEAN; signal s1, s2: BIT;
signal sum : INTEGER range 0 to 15;    -- Not a new type.
signal SmallBus : BIT_VECTOR (15 downto 0); -- 16-bit bus.
```

```
signal GBus : STD_LOGIC_VECTOR (31 downto 0) bus; -- A guarded signal.
begin end;
```

Here is the formal definition of a variable declaration:

```
variable_declaration ::= [shared] variable [10.18]
identifier {, identifier}:subtype_indication [:= expression] ;
```

A **shared variable** can be used to model a varying quantity that is common across several parts of a model, temperature, for example, but shared variables are rarely used in ASIC design. The following examples show that variable declarations belong inside a process statement, after the keyword `process` and before the first appearance of the keyword `begin` inside a process:

```
library IEEE; use IEEE.STD_LOGIC_1164.all; entity Variables_1 is end;
architecture Behave of Variables_1 is begin process
  variable i : INTEGER range 1 to 10 := 10; -- Initial value = 10.
  variable v : STD_LOGIC_VECTOR (0 to 31) := (others => '0');
  begin wait; end process; -- The wait stops an endless cycle.
end;
```

10.9.2 Subprogram Declarations

VHDL code that you use several times can be declared and specified as **subprograms** (functions or procedures) [VHDL LRM2.1]. A **function** is a form of expression, may only use parameters of mode `in`, and may not contain delays or sequence events during simulation (no `wait` statements, for example). Functions are useful to model combinational logic. A **procedure** is a form of statement and allows you to control the scheduling of simulation events without incurring the overhead of defining several separate design entities. There are thus two forms of **subprogram declaration**: a **function declaration** or a **procedure declaration**.

```
subprogram_declaration ::= subprogram_specification ; ::= [10.19]
procedure
  identifier|string_literal [(parameter_interface_list)]
| [pure|impure] function
  identifier|string_literal [(parameter_interface_list)]
return type_name|subtype_name;
```

Here are a function and a procedure declaration that illustrate the difference:

```
function add(a, b, c : BIT_VECTOR(3 downto 0)) return BIT_VECTOR is
-- A function declaration, a function can't modify a, b, or c.

procedure Is_A_Eq_B (signal A, B : BIT; signal Y : out BIT);
-- A procedure declaration, a procedure can change Y.
```

Parameter names in subprogram declarations are called **formal parameters** (or formals). During a call to a subprogram, known as **subprogram invocation**, the passed values are **actual parameters** (or actuals). An **impure** function, such as the function `now` or a function that writes to or reads from a file, may return different values each time it is called (even with the same actuals). A **pure** function (the default) returns the same value if it is given the same actuals. You may call subprograms recursively. Table 10.13 shows the properties of subprogram parameters.

TABLE 10.13 Properties of subprogram parameters.

Example subprogram declarations:

```
function my_function(Ff) return BIT is -- Formal function parameter, Ff.
procedure my_procedure(Fp);          -- Formal procedure parameter, Fp.
```

Example subprogram calls:

```
my_result := my_function(Af); -- Calling a function with an actual parameter, Af.
MY_LABEL:my_procedure(Ap);    -- Using a procedure with an actual parameter, Ap.
```

Mode of Ff or Fp (formals)	in	out	inout	No mode
Permissible classes for Af (function actual parameter)	constant (default) signal	Not allowed	Not allowed	file
Permissible classes for Ap (procedure actual parameter)	constant (default) variable signal	constant variable (default) signal	constant variable (default) signal	file
Can you read attributes of Ff or Fp (formals)?	Yes, except: 'STABLE 'QUIET 'DELAYED 'TRANSACTION of a signal	Yes, except: 'STABLE 'QUIET 'DELAYED 'TRANSACTION 'EVENT 'ACTIVE 'LAST_EVENT 'LAST_ACTIVE 'LAST_VALUE of a signal	Yes, except: 'STABLE 'QUIET 'DELAYED 'TRANSACTION of a signal	

A subprogram declaration is optional, but a **subprogram specification** must be included in the **subprogram body** (and must be identical in syntax to the subprogram declaration—see BNF [10.19]):

```
subprogram_body ::= [10.20]
    subprogram_specification is
    {subprogram_declaration|subprogram_body
    |type_declaration|subtype_declaration
    |constant_declaration|variable_declaration|file_declaration
```

```

|alias_declaration|attribute_declaration|attribute_specification
|use_clause|group_template_declaration|group_declaration}
begin
  {sequential_statement}
end [procedure|function] [identifier|string_literal] ;

```

You can include a subprogram declaration or subprogram body in a package or package body (see Section 10.6) or in the declarative region of an entity or process statement. The following is an example of a function declaration and its body:

```

function subset0(sout0 : in BIT) return BIT_VECTOR -- declaration
-- Declaration can be separate from the body.

function subset0(sout0 : in BIT) return BIT_VECTOR is -- body
variable y : BIT_VECTOR(2 downto 0);
begin
if (sout0 = '0') then y := "000"; else y := "100"; end if;
return result;
end;

procedure clockGen (clk : out BIT) -- Declaration

procedure clockGen (clk : out BIT) is -- Specification
begin -- Careful this process runs forever:
  process begin wait for 10 ns; clk <= not clk; end process;
end;

```

One reason for having the optional (and seemingly redundant) subprogram declaration is to allow companies to show the subprogram declarations (to document the interface) in a package declaration, but to hide the subprogram bodies (the actual code) in the package body. If a separate subprogram declaration is present, it must **conform** to the specification in the subprogram body [VHDL 93LRM2.7]. This means the specification and declaration must be almost identical; the safest method is to copy and paste. If you define common procedures and functions in packages (instead of in each entity or architecture, for example), it will be easier to reuse subprograms. In order to make a subprogram included in a package body visible outside the package, you must declare the subprogram in the package declaration (otherwise the subprogram is **private**).

You may call a function from any expression, as follows:

```

entity F_1 is port (s : out BIT_VECTOR(3 downto 0) := "0000"); end;
architecture Behave of F_1 is begin process
function add(a, b, c : BIT_VECTOR(3 downto 0)) return BIT_VECTOR is
begin return a xor b xor c; end;
begin s <= add("0001", "0010", "1000"); wait; end process; end;

package And_Pkg is
  procedure V_And(a, b : BIT; signal c : out BIT);
  function V_And(a, b : BIT) return BIT;

```

```

end;

package body And_Pkg is
  procedure V_And(a,b : BIT;signal c : out BIT) is
    begin c <= a and b; end;
  function V_And(a,b : BIT) return BIT is
    begin return a and b; end;
end And_Pkg;

entity F_2 is port (s: out BIT := '0'); end;
use work.And_Pkg.all; -- use package already analyzed
architecture Behave of F_2 is begin process begin
s <= V_And('1', '1'); wait; end process; end;

```

I shall discuss the two different ways to call a procedure in Sections 10.10.4 and 10.13.3.

10.9.3 Alias and Attribute Declarations

An **alias declaration** [VHDL 87LRM4.3.4, 93LRM4.3.3] names parts of a type:

```

alias_declaration ::= [10.21]
alias
  identifier|character_literal|operator_symbol [ :subtype_indication]
  is name [signature];

```

(the subtype indication is required in VHDL-87, but not in VHDL-93).

Here is an example of alias declarations for parts of a floating-point number:

```

entity Alias_1 is end; architecture Behave of Alias_1 is
begin process variable Nmbr: BIT_VECTOR (31 downto 0);
-- alias declarations to split Nmbr into 3 pieces :
alias Sign :      BIT is Nmbr(31);
alias Mantissa :  BIT_VECTOR (23 downto 0) is Nmbr (30 downto 7);
alias Exponent :  BIT_VECTOR ( 6 downto 0) is Nmbr ( 6 downto 0);
begin wait; end process; end; -- the wait prevents an endless cycle

```

An **attribute declaration** [VHDL LRM4.4] defines attribute properties:

```

attribute_declaration ::= [10.22]
  attribute identifier:type_name ; | attribute identifier:subtype_name ;

```

Here is an example:

```

entity Attribute_1 is end; architecture Behave of Attribute_1 is
begin process type COORD is record X, Y : INTEGER; end record;
attribute LOCATION : COORD; -- the attribute declaration
begin wait ; -- the wait prevents an endless cycle
end process; end;

```

You define the attribute properties in an **attribute specification** (the following example specifies an attribute of a component label). You probably will not need to use your own attributes very much in ASIC design.

```
attribute LOCATION of adder1 : label is (10,15);
```

You can then refer to your attribute as follows:

```
positionOfComponent := adder1'LOCATION;
```

10.9.4 Predefined Attributes

The predefined attributes for scalar and array types in VHDL-93 are shown in Table 10.14 [VHDL 93LRM14.1]. There are two attributes, 'STRUCTURE and 'BEHAVIOR, that are present in VHDL-87, but removed in VHDL-93. Both of these attributes apply to architecture bodies. The attribute name A'BEHAVIOR is TRUE if the architecture A does not contain component instantiations. The attribute name A'STRUCTURE is TRUE if the architecture A contains only passive processes (those with no assignments to signals) and component instantiations. These two attributes were not widely used. The attributes shown in Table 10.14, however, are used extensively to create packages and functions for type conversion and overloading operators, but should not be needed by an ASIC designer. Many of the attributes do not correspond to “real” hardware and cannot be implemented by a synthesis tool.

The attribute 'LEFT is important because it determines the default initial value of a type. For example, the default initial value for type BIT is BIT'LEFT, which is '0'. The predefined attributes of signals are listed in Table 10.15. The most important signal attribute is 'EVENT, which is frequently used to detect a clock edge. Notice that Clock'EVENT, for example, is a function that returns a value of type BOOLEAN, whereas the otherwise equivalent not(Clock'STABLE), is a signal. The difference is subtle but important when these attributes are used in the wait statement that treats signals and values differently.

10.10 Sequential Statements

A **sequential statement** [VHDL LRM8] is defined as follows:

```
sequential_statement ::= [10.23]
    wait_statement      | assertion_statement
| signal_assignment_statement
| variable_assignment_statement      | procedure_call_statement
| if_statement         | case_statement | loop_statement
| next_statement      | exit_statement
| return_statement    | null_statement | report_statement
```

TABLE 10.14 Predefined attributes for scalar and array types.

Attribute	Kind ¹	Prefix T, A, E ²	Parameter X or N ³	Result type ³	Result
T'BASE	T	any		base(T)	base(T), use only with other attribute
T'LEFT	V	scalar		T	Left bound of T
T'RIGHT	V	scalar		T	Right bound of T
T'HIGH	V	scalar		T	Upper bound of T
T'LOW	V	scalar		T	Lower bound of T
T'ASCENDING	V	scalar		BOOLEAN	True if range of T is ascending ⁴
T'IMAGE(X)	F	scalar	base(T)	STRING	String representation of X in T ⁴
T'VALUE(X)	F	scalar	STRING	base(T)	Value in T with representation X ⁴
T'POS(X)	F	discrete	base(T)	UI	Position number of X in T (starts at 0)
T'VAL(X)	F	discrete	UI	base(T)	Value of position X in T
T'SUCC(X)	F	discrete	base(T)	base(T)	Value of position X in T plus one
T'PRED(X)	F	discrete	base(T)	base(T)	Value of position X in T minus one
T'LEFTOF(X)	F	discrete	base(T)	base(T)	Value to the left of X in T
T'RIGHTOF(X)	F	discrete	base(T)	base(T)	Value to the right of X in T
A'LEFT[(N)]	F	array	UI	T(Result)	Left bound of index N of array A
A'RIGHT[(N)]	F	array	UI	T(Result)	Right bound of index N of array A
A'HIGH[(N)]	F	array	UI	T(Result)	Upper bound of index N of array A
A'LOW[(N)]	F	array	UI	T(Result)	Lower bound of index N of array A
A'RANGE[(N)]	R	array	UI	T(Result)	Range A'LEFT(N) to A'RIGHT(N) ⁵
A'REVERSE_RANGE[(N)]	R	array	UI	T(Result)	Opposite range to A'RANGE[(N)]
A'LENGTH[(N)]	V	array	UI	UI	Number of values in index N of array A
A'ASCENDING[(N)]	V	array	UI	BOOLEAN	True if index N of A is ascending ⁴
E'SIMPLE_NAME	V	name		STRING	Simple name of E ⁴
E'INSTANCE_NAME	V	name		STRING	Path includes instantiated entities ⁴
E'PATH_NAME	V	name		STRING	Path excludes instantiated entities ⁴

¹T=Type, F=Function, V=Value, R=Range.

²any=any type or subtype, scalar=scalar type or subtype, discrete=discrete or physical type or subtype, name=entity name=identifier, character literal, or operator symbol.

³base(T)=base type of T, T=type of T, UI=universal_integer, T(Result)=type of object described in result column.

⁴Only available in VHDL-93. For 'ASCENDING all enumeration types are ascending.

⁵Or reverse for descending ranges.

TABLE 10.15 Predefined attributes for signals.

Attribute	Kind ¹	Parameter T ²	Result type ³	Result/restrictions
S'DELAYED [(T)]	S	TIME	base(S)	S delayed by time T
S'STABLE [(T)]	S	TIME	BOOLEAN	TRUE if no event on S for time T
S'QUIET [(T)]	S	TIME	BOOLEAN	TRUE if S is quiet for time T
S'TRANSACTION	S		BIT	Toggles each cycle if S becomes active
S'EVENT	F		BOOLEAN	TRUE when event occurs on S
S'ACTIVE	F		BOOLEAN	TRUE if S is active
S'LAST_EVENT	F		TIME	Elapsed time since the last event on S
S'LAST_ACTIVE	F		TIME	Elapsed time since S was active
S'LAST_VALUE	F		base(S)	Previous value of S, before last event ⁴
S'DRIVING	F		BOOLEAN	TRUE if every element of S is driven ⁵
S'DRIVING_VALUE	F		base(S)	Value of the driver for S in the current process ⁵

¹ F=function, S=signal.

²Time T≥0 ns. The default, if T is not present, is T=0 ns.

³base(S)=base type of S.

⁴VHDL-93 returns last value of each signal in array separately as an aggregate, VHDL-87 returns the last value of the composite signal.

⁵VHDL-93 only.

Sequential statements may only appear in processes and subprograms. In the following sections I shall describe each of these different types of sequential statements in turn.

10.10.1 Wait Statement

The **wait statement** is central to VHDL, here are the BNF definitions [VHDL 93LRM8.1]:

```
wait_statement ::= [label:] wait [sensitivity_clause] [10.24]
                [condition_clause] [timeout_clause] ;
sensitivity_clause ::= on sensitivity_list
sensitivity_list ::= signal_name { , signal_name }
condition_clause ::= until condition
condition ::= boolean_expression
timeout_clause ::= for time_expression
```

A wait statement **suspends** (stops) a process or procedure (you cannot use a wait statement in a function). The wait statement may be made sensitive to events (changes) on **static** signals (the value of the signal must be known at analysis time)

that appear in the **sensitivity list** after the keyword **on**. These signals form the **sensitivity set** of a wait statement. The process will **resume** (restart) when an event occurs on any signal (and only signals) in the sensitivity set.

A wait statement may also contain a condition to be met before the process resumes. If there is no **sensitivity clause** (there is no keyword **on**) the sensitivity set is made from signals (and only signals) from the **condition clause** that appears after the keyword **until** (the rules are quite complicated [VHDL 93LRM8.1]).

Finally a wait statement may also contain a **timeout** (following the keyword **for**) after which the process will resume. Here is the expanded BNF definition, which makes the structure of the wait statement easier to see (but we lose the definitions of the clauses and the sensitivity list):

```
wait_statement ::= [label:] wait
                [on signal_name {, signal_name}]
                [until boolean_expression]
                [for time_expression] ;
```

For example, the statement, **wait on light**, makes you wait until a traffic light changes (any change). The statement, **wait until light = green**, makes you wait (even at a green light) until the traffic signal changes to green. The statement,

```
if light = (red or yellow) then wait until light = green; end if;
```

accurately describes the basic rules at a traffic intersection.

The most common use of the wait statement is to describe synchronous logic, as in the following model of a D flip-flop:

```
entity DFF is port (CLK, D : BIT; Q : out BIT); end;           --1
architecture Behave of DFF is                                 --2
process begin wait until Clk = '1'; Q <= D ; end process;     --3
end;                                                           --4
```

Notice that the statement in line 3 above, **wait until Clk = '1'**, is equivalent to **wait on Clk until Clk = '1'**, and detects a clock edge and not the clock level. Here are some more complex examples of the use of the wait statement:

```
entity Wait_1 is port (Clk, s1, s2 :in BIT); end;
architecture Behave of Wait_1 is
signal x : BIT_VECTOR (0 to 15);
begin process variable v : BIT; begin
wait;                                     -- Wait forever, stops simulation.
wait on s1 until s2 = '1'; -- Legal, but s1, s2 are signals so
-- s1 is in sensitivity list, and s2 is not in the sensitivity set.
-- Sensitivity set is s1 and process will not resume at event on s2.
wait on s1, s2;                          -- resumes at event on signal s1 or s2.
wait on s1 for 10 ns;                     -- resumes at event on s1 or after 10 ns.
wait on x;                                -- resumes when any element of array x
-- has an event.
```

```

-- wait on x(1 to v); -- Illegal, nonstatic name, since v is a variable.
end process;
end;

entity Wait_2 is port (Clk, s1, s2:in BIT); end;
architecture Behave of Wait_2 is
  begin process variable v : BIT; begin
    wait on Clk; -- resumes when Clk has an event: rising or falling.
    wait until Clk = '1';          -- resumes on rising edge.
    wait on Clk until Clk = '1'; -- equivalent to the last statement.
    wait on Clk until v = '1';
    -- The above is legal, but v is a variable so
    -- Clk is in sensitivity list, v is not in the sensitivity set.
    -- Sensitivity set is Clk and process will not resume at event on v.
    wait on Clk until s1 = '1';
    -- The above is legal, but s1 is a signal so
    -- Clk is in sensitivity list, s1 is not in the sensitivity set.
    -- Sensitivity set is Clk, process will not resume at event on s1.
  end process;
end;

```

You may only use interface signals that may be read (port modes in, inout, and buffer—see Section 10.7) in the sensitivity list of a wait statement.

10.10.2 Assertion and Report Statements

You can use an **assertion statement** to conditionally issue warnings. The **report statement** (VHDL-93 only) prints an expression and is useful for debugging.

```

assertion_statement ::= [label:] assert                                [10.25]
boolean_expression [report expression] [severity expression] ;

report_statement ::=
[label:] report expression [severity expression] ;

```

Here is an example of an assertion statement:

```

entity Assert_1 is port (I:INTEGER:=0); end;
architecture Behave of Assert_1 is
  begin process begin
    assert (I > 0) report "I is negative or zero"; wait;
  end process;
end;

```

The expression after the keyword **report** must be of type **STRING** (the default is "Assertion violation" for the assertion statement), and the expression after the keyword **severity** must be of type **SEVERITY_LEVEL** (default **ERROR** for the assertion statement, and **NOTE** for the report statement) defined in the **STANDARD**

package. The assertion statement prints if the assertion condition (after the keyword `assert`) is `FALSE`. Simulation normally halts for severity of `ERROR` or `FAILURE` (you can normally control this threshold in the simulator).

10.10.3 Assignment Statements

There are two sorts of VHDL assignment statements: one for signals and one for variables [VHDL 93LRM8.4–8.5]. The difference is in the timing of the update of the LHS. A **variable assignment statement** is the closest equivalent to the assignment statement in a computer programming language. Variable assignment statements are always sequential statements and the LHS of a variable assignment statement is always updated immediately. Here is the definition and an example:

```
variable_assignment_statement ::= [10.26]
    [label:] name | aggregate := expression ;

entity Var_Assignment is end;
architecture Behave of Var_Assignment is
    signal s1 : INTEGER := 0;
    begin process variable v1,v2 : INTEGER := 0; begin
        assert (v1/=0) report "v1 is 0" severity note ; -- this prints
        v1 := v1 + 1; -- after this statement v1 is 1
        assert (v1=0) report "v1 isn't 0" severity note ; -- this prints
        v2 := v2 + s1; -- signal and variable types must match
        wait;
    end process;
end;
```

This is the output from Cadence Leapfrog for the preceding example:

```
ASSERT/NOTE (time 0 FS) from :$PROCESS_000 (design unit
WORK.VAR_ASSIGNMENT:BEHAVE) v1 is 0
ASSERT/NOTE (time 0 FS) from :$PROCESS_000 (design unit
WORK.VAR_ASSIGNMENT:BEHAVE) v1 isn't 0
```

A **signal assignment statement** schedules a future assignment to a signal:

```
signal_assignment_statement ::= [10.27]
    [label:] target <=
    [transport | [ reject time_expression ] inertial] waveform ;
```

The following example shows that, even with no delay, a signal is updated at the end of a simulation cycle after all the other assignments have been scheduled, just before simulation time is advanced:

```
entity Sig_Assignment_1 is end;
architecture Behave of Sig_Assignment_1 is
    signal s1,s2,s3 : INTEGER := 0;
    begin process variable v1 : INTEGER := 1; begin
        assert (s1 /= 0) report "s1 is 0" severity note ; -- this prints.
```

```

s1 <= s1 + 1; -- after this statement s1 is still 0.
assert (s1 /= 0) report "s1 still 0" severity note ; -- this prints.
wait;
end process;
end;

```

```

ASSERT/NOTE (time 0 FS) from :$PROCESS_000 (design unit
WORK.SIG_ASSIGNMENT_1:BEHAVE) s1 is 0
ASSERT/NOTE (time 0 FS) from :$PROCESS_000 (design unit
WORK.SIG_ASSIGNMENT_1:BEHAVE) s1 still 0

```

Here is another example to illustrate how time is handled:

```

entity Sig_Assignment_2 is end;
architecture Behave of Sig_Assignment_2 is
  signal s1, s2, s3 : INTEGER := 0;
  begin process variable v1 : INTEGER := 1; begin
    -- s1, s2, s3 are initially 0; now consider the following:
    s1 <= 1 ; -- schedules updates to s1 at end of 0 ns cycle.
    s2 <= s1; -- s2 is 0, not 1.
    wait for 1 ns;
    s3 <= s1; -- now s3 will be 1 at 1 ns.
    wait;
  end process;
end;

```

The Compass simulator produces the following trace file for this example:

Time(fs) + Cycle	s1	s2	s3
0+ 0:	0	0	0
0+ 1: *	1 *	0	0
...			
1000000+ 1:	1	0 *	1

Time is indicated in femtoseconds for each **simulation cycle** plus the number of **delta cycles** (we call this **delta time**, measured in units of **delta**, δ) needed to calculate all transactions on signals. A transaction consists of a new value for a signal (which may be the same as the old value) and the time delay for the value to take effect. An asterisk '*' before a value in the preceding trace indicates that a transaction has occurred and the corresponding signal updated at that time. A transaction that does result in a change in value is an **event**. In the preceding simulation trace for `Sig_Assignment_2:Behave`

- At 0 ns+ 0 δ : all signals are 0.
- At 0 ns+ 1 δ : s1 is updated to 1, s2 is updated to 0 (not to 1).
- At 1 ns+ 1 δ : s3 is updated to a 1.

The following example shows the behavior of the different **delay models**: **transport** and **inertial** (the default):

```
entity Transport_1 is end;
architecture Behave of Transport_1 is
signal s1, SLOW, FAST, WIRE : BIT := '0';
begin process begin
s1 <= '1' after 1 ns, '0' after 2 ns, '1' after 3 ns ;
-- schedules s1 to be '1' at t+1 ns, '0' at t+2 ns, '1' at t+3 ns
wait; end process;
-- inertial delay: SLOW rejects pulsewidths less than 5ns:
process (s1) begin SLOW <= s1 after 5 ns ; end process;
-- inertial delay: FAST rejects pulsewidths less than 0.5ns:
process (s1) begin FAST <= s1 after 0.5 ns ; end process;
-- transport delay: WIRE passes all pulsewidths...
process (s1) begin WIRE <= transport s1 after 5 ns ; end process;
end;
```

Here is the trace file from the Compass simulator:

Time(fs) + Cycle	s1	slow	fast	wire
0+ 0:	'0'	'0'	'0'	'0'
500000+ 0:	'0'	'0'	*'0'	'0'
1000000+ 0:	*'1'	'0'	'0'	'0'
1500000+ 0:	'1'	'0'	*'1'	'0'
2000000+ 0:	*'0'	'0'	'1'	'0'
2500000+ 0:	'0'	'0'	*'0'	'0'
3000000+ 0:	*'1'	'0'	'0'	'0'
3500000+ 0:	'1'	'0'	*'1'	'0'
5000000+ 0:	'1'	'0'	'1'	*'0'
6000000+ 0:	'1'	'0'	'1'	*'1'
7000000+ 0:	'1'	'0'	'1'	*'0'
8000000+ 0:	'1'	*'1'	'1'	*'1'

Inertial delay mimics the behavior of real logic gates, whereas transport delay more closely models the behavior of wires. In VHDL-93 you can also add a separate **pulse rejection limit** for the inertial delay model as in the following example:

```
process (s1) begin RJCT <= reject 2 ns s1 after 5 ns ; end process;
```

10.10.4 Procedure Call

A **procedure call** in VHDL corresponds to calling a subroutine in a conventional programming language [VHDL LRM8.6]. The parameters in a procedure call statement are the actual procedure parameters (or actuals); the parameters in the procedure definition are the formal procedure parameters (or formals). The two are linked

using an association list, which may use either positional or named association (association works just as it does for ports—see Section 10.7.1):

```
procedure_call_statement ::= [10.28]
    [label:] procedure_name [(parameter_association_list)];
```

Here is an example:

```
package And_Pkg is
    procedure V_And(a, b : BIT; signal c : out BIT);
    function V_And(a, b : BIT) return BIT;
end;

package body And_Pkg is
    procedure V_And(a, b : BIT; signal c: out BIT) is
        begin c <= a and b; end;
    function V_And(a, b: BIT) return BIT is
        begin return a and b; end;
end And_Pkg;

use work.And_Pkg.all; entity Proc_Call_1 is end;
architecture Behave of Proc_Call_1 is signal A, B, Y: BIT := '0';
    begin process begin V_And (A, B, Y); wait; end process;
end;
```

Table 10.13 on page 416 explains the rules for formal procedure parameters. There is one other way to call procedures, which we shall cover in Section 10.13.3.

10.10.5 If Statement

An **if statement** evaluates one or more Boolean expressions and conditionally executes a corresponding sequence of statements [VHDL LRM8.7].

```
if_statement ::= [10.29]
    [if_label:] if boolean_expression then {sequential_statement}
        {elsif boolean_expression then {sequential_statement}}
        [else {sequential_statement}]
    end if [if_label];
```

The simplest form of an **if** statement is thus:

```
if boolean_expression then {sequential_statement} end if;
```

Here are some examples of the **if** statement:

```
entity If_Then_Else_1 is end;
architecture Behave of If_Then_Else_1 is signal a, b, c: BIT := '1';
    begin process begin
        if c = '1' then c <= a ; else c <= b; end if; wait;
    end process;
end;
```

```

entity If_Then_1 is end;
architecture Behave of If_Then_1 is signal A, B, Y : BIT := '1';
begin process begin
    if A = B then Y <= A; end if; wait;
end process;
end;

```

10.10.6 Case Statement

A **case statement** [VHDL LRM8.8] is a multiway decision statement that selects a sequence of statements by matching an expression with a list of (locally static [VHDL LRM7.4.1]) choices.

```

case_statement ::= [10.30]
[case_label:] case expression is
    when choice { | choice } => {sequential_statement}
    {when choice { | choice } => {sequential_statement}}
end case [case_label];

```

Case statements are useful to model state machines. Here is an example of a Mealy state machine with an asynchronous reset:

```

library IEEE; use IEEE.STD_LOGIC_1164.all; --1
entity sm_mealy is --2
    port (reset, clock, i1, i2 : STD_LOGIC; o1, o2 : out STD_LOGIC); --3
end sm_mealy; --4
architecture Behave of sm_mealy is --5
type STATES is (s0, s1, s2, s3); signal current, new : STATES; --6
begin --7
synchronous : process (clock, reset) begin --8
    if To_X01(reset) = '0' then current <= s0; --9
    elsif rising_edge(clock) then current <= new; end if; --10
end process; --11
combinational : process (current, i1, i2) begin --12
case current is --13
    when s0 => --14
        if To_X01(i1) = '1' then o2 <='0'; o1 <='0'; new <= s2; --15
        else o2 <= '1'; o1 <= '1'; new <= s1; end if; --16
    when s1 => --17
        if To_X01(i2) = '1' then o2 <='1'; o1 <='0'; new <= s1; --18
        else o2 <='0'; o1 <='1'; new <= s3; end if; --19
    when s2 => --20
        if To_X01(i2) = '1' then o2 <='0'; o1 <='1'; new <= s2; --21
        else o2 <= '1'; o1 <= '0'; new <= s0; end if; --22
    when s3 => o2 <= '0'; o1 <= '0'; new <= s0; --23
    when others => o2 <= '0'; o1 <= '0'; new <= s0; --24
end case; --25
end process; --26
end Behave; --27

```

Each possible value of the case expression must be present once, and once only, in the list of choices (or arms) of the case statement (the list must be **exhaustive**). You can use '|' (that means 'or') or 'to' to denote a range in the expression for choice. You may also use the keyword *others* as the last, default choice (even if the list is already exhaustive, as in the preceding example).

10.10.7 Other Sequential Control Statements

A **loop statement** repeats execution of a series of sequential statements [VHDL LRM8.9]:

```
loop_statement ::= [10.31]
[loop_label:]
[while boolean_expression|for identifier in discrete_range]
loop
  {sequential_statement}
end loop [loop_label];
```

If the **loop variable** (after the keyword *for*) is used, it is only visible inside the loop. A *while* loop evaluates the Boolean expression before each execution of the sequence of statements; if the expression is **TRUE**, the statements are executed. In a *for* loop the sequence of statements is executed once for each value of the discrete range.

```
package And_Pkg is function V_And(a, b : BIT) return BIT; end;
package body And_Pkg is function V_And(a, b : BIT) return BIT is
  begin return a and b; end; end And_Pkg;
entity Loop_1 is port (x, y : in BIT := '1'; s : out BIT := '0'); end;
use work.And_Pkg.all;
architecture Behave of Loop_1 is
  begin loop
    s <= V_And(x, y); wait on x, y;
  end loop;
end;
```

The **next statement** [VHDL LRM8.10] forces completion of the current iteration of a loop (the containing loop unless another loop label is specified). Completion is forced if the condition following the keyword *then* is **TRUE** (or if there is no condition).

```
next_statement ::= [10.32]
[label:] next [loop_label] [when boolean_expression];
```


An **exit statement** [VHDL LRM8.11] forces an exit from a loop.

```
exit_statement ::= [label:] exit [loop_label] [when condition] ; [10.33]
```

As an example:

```
loop wait on Clk; exit when Clk = '0'; end loop;
-- equivalent to: wait until Clk = '0';
```

The **return statement** [VHDL LRM8.12] completes execution of a procedure or function.

```
return_statement ::= [label:] return [expression]; [10.34]
```

A **null statement** [VHDL LRM8.13] does nothing (but is useful in a case statement where all choices must be covered, but for some of the choices you do not want to do anything).

```
null_statement ::= [label:] null; [10.35]
```

10.11 Operators

Table 10.16 shows the predefined VHDL **operators**, listed by their (increasing) order of precedence [VHDL 93LRM7.2]. The shift operators and the xnor operator were added in VHDL-93.

TABLE 10.16 VHDL predefined operators (listed by increasing order of precedence).¹

logical_operator ² ::=	and or nand nor xor <u>xnor</u>
relational_operator ::=	= /= < <= > >=
shift_operator ² ::=	<u>sll</u> <u>srl</u> <u>sla</u> <u>sra</u> <u>rol</u> <u>ror</u>
adding_operator ::=	+ - &
sign ::=	+ -
multiplying_operator ::=	* / mod rem
miscellaneous_operator ::=	** abs not

¹The not operator is a logical operator but has the precedence of a miscellaneous operator.

²Underline means “new to VHDL-93.”

The binary **logical operators** (and, or, nand, nor, xor, xnor) and the unary not logical operator are predefined for types BIT or BOOLEAN and one-dimensional arrays whose element type is BIT or BOOLEAN. The operands must be of the same base type for the binary logical operators and the same length if they are arrays.

Both operands of **relational operators** must be of the same type and the result type is BOOLEAN. The equality operator and inequality operator ('=' and '/=') are defined for all types (other than file types). The remaining relational operators, ordering operators, are predefined for any scalar type, and for any one-dimensional array whose elements are of a discrete type (enumeration or integer type).

The left operand of the **shift operators** (VHDL-93 only) is a one-dimensional array with element type of BIT or BOOLEAN; the right operand must be INTEGER.

The **adding operators** ('+' and '-') are predefined for any numeric type. You cannot use the adding operators on BIT or BIT_VECTOR without overloading. The **concatenation operator** '&' is predefined for any one-dimensional array type. The **signs** ('+' and '-') are defined for any numeric type.

The **multiplying operators** are: '*', '/', mod, and rem. The operators '*' and '/' are predefined for any integer or floating-point type, and the operands and the result are of the same type. The operators mod and rem are predefined for any integer type, and the operands and the result are of the same type. In addition, you can multiply an INTEGER or REAL by any physical type and the result is the physical type. You can also divide a physical type by REAL or INTEGER and the result is the physical type. If you divide a physical type by the same physical type, the result is an INTEGER (actually type UNIVERSAL_INTEGER, which is a predefined anonymous type [VHDL LRM7.5]). Once again—you cannot use the multiplying operators on BIT or BIT_VECTOR types without overloading the operators.

The **exponentiating operator**, '**', is predefined for integer and floating-point types. The right operand, the exponent, is type INTEGER. You can only use a negative exponent with a left operand that is a floating-point type, and the result is the same type as the left operand. The unary operator abs (**absolute value**) is predefined for any numeric type and the result is the same type. The operators abs, '**', and not are grouped as **miscellaneous operators**.

Here are some examples of the use of VHDL operators:

```
entity Operator_1 is end; architecture Behave of Operator_1 is      --1
begin process                                                    --2
variable b : BOOLEAN; variable bt : BIT := '1'; variable i : INTEGER;--3
variable pi : REAL := 3.14; variable epsilon : REAL := 0.01;    --4
variable bv4 : BIT_VECTOR (3 downto 0) := "0001";                --5
variable bv8 : BIT_VECTOR (0 to 7);                               --6
begin                                                            --7

b := "0000" < bv4; -- b is TRUE, "0000" treated as BIT_VECTOR.  --8
b := 'f' > 'g';      -- b is FALSE, 'dictionary' comparison.    --9
bt := '0' and bt;    -- bt is '0', analyzer knows '0' is BIT.   --10
bv4 := not bv4;      -- bv4 is now "1110".                        --11
i := 1 + 2;          -- Addition, must be compatible types.     --12
```

```

i := 2 ** 3;           -- Exponentiation, exponent must be integer. --13
i := 7/3;             -- Division, L/R rounded towards zero, i=2. --14
i := 12 rem 7;        -- Remainder, i=5. In general: --15
                       -- L rem R = L-((L/R)*R). --16
i := 12 mod 7;        -- modulus, i=5. In general: --17
                       -- L mod R = L-(R*N) for an integer N. --18

-- shift := sll | srl | sla | sra | rol | ror (VHDL-93 only) --19
bv4 := "1001" srl 2;  -- Shift right logical, now bv4="0100". --20
-- Logical shift fills with T'LEFT. --21
bv4 := "1001" sra 2;  -- Shift right arithmetic, now bv4="0111". --22
-- Arithmetic shift fills with element at end being vacated. --23
bv4 := "1001" ror 2;  -- Rotate right, now bv4="0110". --24
-- Rotate wraps around. --25
-- Integer argument to any shift operator may be negative or zero. --26

if (pi*2.718)/2.718 = 3.14 then wait; end if; -- This is unreliable.--27
if (abs((pi*2.718)/2.718)-3.14)<epsilon then wait; end if; -- Better.--28

bv8 := bv8(1 to 7) & bv8(0); -- Concatenation, a left rotation. --29

wait; end process; --30
end; --31

```

10.12 Arithmetic

The following example illustrates **type checking** and **type conversion** in VHDL arithmetic operations [VHDL 93LRM7.3.4–7.3.5]:

```

entity Arithmetic_1 is end; architecture Behave of Arithmetic_1 is --1
begin process
  variable i : INTEGER := 1; variable r : REAL := 3.33; --2
  variable b : BIT := '1'; --3
  variable bv4 : BIT_VECTOR (3 downto 0) := "0001"; --4
  variable bv8 : BIT_VECTOR (7 downto 0) := B"1000_0000"; --5
begin --6

-- i := r; -- you can't assign REAL to INTEGER. --7
-- bv4 := bv4 + 2; -- you can't add BIT_VECTOR and INTEGER. --8
-- bv4 := '1'; -- you can't assign BIT to BIT_VECTOR. --9
-- bv8 := bv4; -- an error, the arrays are different sizes.--10

r := REAL(i); -- OK, uses a type conversion. --11
i := INTEGER(r); -- OK (0.5 rounds up or down). --12
bv4 := "001" & '1'; -- OK, you can mix an array and a scalar. --13
bv8 := "0001" & bv4; -- OK, if arguments are the correct lengths.--14
wait; end process; end; --15

```

The next example shows arithmetic operations between types and subtypes, and also illustrates **range checking** during analysis and simulation:

```
entity Arithmetic_2 is end; architecture Behave of Arithmetic_2 is --1
type TC is range 0 to 100; -- Type INTEGER. --2
type TF is range 32 to 212; -- Type INTEGER. --3
subtype STC is INTEGER range 0 to 100; -- Subtype of type INTEGER. --4
subtype STF is INTEGER range 32 to 212; -- Base type is INTEGER. --5
begin process --6
variable t1 : TC := 25; variable t2 : TF := 32; --7
variable st1 : STC := 25; variable st2 : STF := 32; --8
begin --9
-- t1 := t2; -- Illegal, different types. --10
-- t1 := st1; -- Illegal, different types and subtypes. --11
  st2 := st1; -- OK to use same base types. --12
  st2 := st1 + 1; -- OK to use subtype and base type. --13
-- st2 := 213; -- Error, outside range at analysis time. --14
-- st2 := 212 + 1; -- Error, outside range at analysis time. --15
  st1 := st1 + 100; -- Error, outside range at initialization. --16
wait; end process; end;
```

The MTI simulator, for example, gives the following informative error message during simulation of the preceding model:

```
# ** Fatal: Value 25 is out of range 32 to 212
# Time: 0 ns Iteration: 0 Instance:/
# Stopped at Arithmetic_2.vhd line 12
# Fatal error at Arithmetic_2.vhd line 12
```

The assignment `st2 := st1` causes this error (since `st1` is initialized to 25).

Operations between array types and subtypes are a little more complicated as the following example illustrates:

```
entity Arithmetic_3 is end; architecture Behave of Arithmetic_3 is --1
type TYPE_1 is array (INTEGER range 3 downto 0) of BIT; --2
type TYPE_2 is array (INTEGER range 3 downto 0) of BIT; --3
subtype SUBTYPE_1 is BIT_VECTOR (3 downto 0); --4
subtype SUBTYPE_2 is BIT_VECTOR (3 downto 0); --5
begin process --6
variable bv4 : BIT_VECTOR (3 downto 0) := "0001"; --7
variable st1 : SUBTYPE_1 := "0001"; variable t1 : TYPE_1 := "0001"; --8
variable st2 : SUBTYPE_2 := "0001"; variable t2 : TYPE_2 := "0001"; --9
begin --10
  bv4 := st1; -- OK, compatible type and subtype. --11
-- bv4 := t1; -- Illegal, different types. --12
  bv4 := BIT_VECTOR(t1); -- OK, type conversion. --13
  st1 := bv4; -- OK, compatible subtype and base type. --14
-- st1 := t1; -- Illegal, different types. --15
  st1 := SUBTYPE_1(t1); -- OK, type conversion. --16
```

```

-- t1 := st1;           -- Illegal, different types.           --17
-- t1 := bv4;          -- Illegal, different types.           --18
  t1 := TYPE_1(bv4);   -- OK, type conversion.               --19
-- t1 := t2;           -- Illegal, different types.           --20
  t1 := TYPE_1(t2);   -- OK, type conversion.               --21
  st1 := st2;         -- OK, compatible subtypes.            --22
wait; end process; end; --23

```

The preceding example uses `BIT` and `BIT_VECTOR` types, but exactly the same considerations apply to `STD_LOGIC` and `STD_LOGIC_VECTOR` types or other arrays. Notice the use of **type conversion**, written as `type_mark'(expression)`, to convert between **closely related types**. Two types are closely related if they are abstract numeric types (integer or floating-point) or arrays with the same dimension, each index type is the same (or are themselves closely related), and each element has the same type [VHDL 93LRM7.3.5].

10.12.1 IEEE Synthesis Packages

The IEEE 1076.3 standard synthesis packages allow you to perform arithmetic on arrays of the type `BIT` and `STD_LOGIC`.³ The `NUMERIC_BIT` package defines all of the operators in Table 10.16 (except for the exponentiating operator `'**'`) for arrays of type `BIT`. Here is part of the package header, showing the declaration of the two types `UNSIGNED` and `SIGNED`, and an example of one of the function declarations that overloads the addition operator `'+'` for `UNSIGNED` arguments:

```

package Part_NUMERIC_BIT is
type UNSIGNED is array (NATURAL range <> ) of BIT;
type SIGNED is array (NATURAL range <> ) of BIT;
function "+" (L, R : UNSIGNED) return UNSIGNED;
-- other function definitions that overload +, -, =, >, and so on.
end Part_NUMERIC_BIT;

```

The package bodies included in the 1076.3 standard define the functionality of the packages. Companies may implement the functions in any way they wish—as long as the results are the same as those defined by the standard. Here is an example of the parts of the `NUMERIC_BIT` package body that overload the addition operator `'+'` for two arguments of type `UNSIGNED` (even with my added comments the code is rather dense and terse, but remember this is code that we normally never see or need to understand):

```

package body Part_NUMERIC_BIT is
constant NAU : UNSIGNED(0 downto 1) := (others =>'0'); -- Null array.

```

³IEEE Std 1076.3-1997 was approved by the IEEE Standards Board on 20 March 1997. The synthesis package code on the following pages is reprinted with permission from IEEE Std 1076.3-1997, Copyright © 1997 IEEE. All rights reserved.

```

constant NAS : SIGNED(0 downto 1):=(others => '0'); -- Null array.
constant NO_WARNING : BOOLEAN := FALSE; -- Default to emit warnings.

function MAX (LEFT, RIGHT : INTEGER) return INTEGER is
begin -- Internal function used to find longest of two inputs.
if LEFT > RIGHT then return LEFT; else return RIGHT; end if; end MAX;

function ADD_UNSIGNED (L, R : UNSIGNED; C: BIT) return UNSIGNED is
constant L_LEFT : INTEGER := L'LENGTH-1; -- L, R must be same length.
alias XL : UNSIGNED(L_LEFT downto 0) is L; -- Descending alias,
alias XR : UNSIGNED(L_LEFT downto 0) is R; -- aligns left ends.
variable RESULT : UNSIGNED(L_LEFT downto 0); variable CBIT : BIT := C;
begin for I in 0 to L_LEFT loop -- Descending alias allows loop.
RESULT(I) := CBIT xor XL(I) xor XR(I); -- CBIT = carry, initially = C.
CBIT := (CBIT and XL(I)) or (CBIT and XR(I)) or (XL(I) and XR(I));
end loop; return RESULT; end ADD_UNSIGNED;

function RESIZE (ARG : UNSIGNED; NEW_SIZE : NATURAL) return UNSIGNED is
constant ARG_LEFT : INTEGER := ARG'LENGTH-1;
alias XARG : UNSIGNED(ARG_LEFT downto 0) is ARG; -- Descending range.
variable RESULT : UNSIGNED(NEW_SIZE-1 downto 0) := (others => '0');
begin -- resize the input ARG to length NEW_SIZE
if (NEW_SIZE < 1) then return NAU; end if; -- Return null array.
if XARG'LENGTH = 0 then return RESULT; end if; -- Null to empty.
if (RESULT'LENGTH < ARG'LENGTH) then -- Check lengths.
RESULT(RESULT'LEFT downto 0) := XARG(RESULT'LEFT downto 0);
else -- Need to pad the result with some '0's.
RESULT(RESULT'LEFT downto XARG'LEFT + 1) := (others => '0');
RESULT(XARG'LEFT downto 0) := XARG;
end if; return RESULT;
end RESIZE;

function "+" (L, R : UNSIGNED) return UNSIGNED is -- Overloaded '+'.
constant SIZE : NATURAL := MAX(L'LENGTH, R'LENGTH);
begin -- If length of L or R < 1 return a null array.
if ((L'LENGTH < 1) or (R'LENGTH < 1)) then return NAU; end if;
return ADD_UNSIGNED(RESIZE(L, SIZE), RESIZE(R, SIZE), '0'); end "+";

end Part_NUMERIC_BIT;

```

The following conversion functions are also part of the NUMERIC_BIT package:

```

function TO_INTEGER (ARG : UNSIGNED) return NATURAL;
function TO_INTEGER (ARG : SIGNED) return INTEGER;
function TO_UNSIGNED (ARG, SIZE : NATURAL) return UNSIGNED;
function TO_SIGNED (ARG : INTEGER; SIZE : NATURAL) return SIGNED;
function RESIZE (ARG : SIGNED; NEW_SIZE : NATURAL) return SIGNED;
function RESIZE (ARG : UNSIGNED; NEW_SIZE : NATURAL) return UNSIGNED;
-- set XMAP to convert unknown values, default is 'X'-'>'0'
function TO_01(S : UNSIGNED; XMAP : STD_LOGIC := '0') return UNSIGNED;
function TO_01(S : SIGNED; XMAP : STD_LOGIC := '0') return SIGNED;

```

The NUMERIC_STD package is almost identical to the NUMERIC_BIT package except that the UNSIGNED and SIGNED types are declared in terms of the STD_LOGIC type from the Std_Logic_1164 package as follows:

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
package Part_NUMERIC_STD is
type UNSIGNED is array (NATURAL range <>) of STD_LOGIC;
type SIGNED is array (NATURAL range <>) of STD_LOGIC;
end Part_NUMERIC_STD;
```

The NUMERIC_STD package body is similar to NUMERIC_BIT with the addition of a comparison function called STD_MATCH, illustrated by the following:

```
-- function STD_MATCH (L, R: T) return BOOLEAN;
-- T = STD_ULOGIC UNSIGNED SIGNED STD_LOGIC_VECTOR STD_ULOGIC_VECTOR
```

The STD_MATCH function uses the following table to compare logic values:

```
type BOOLEAN_TABLE is array(STD_ULOGIC, STD_ULOGIC) of BOOLEAN;
constant MATCH_TABLE : BOOLEAN_TABLE := (
```

```
-----
-- U   X   0   1   Z   W   L   H   -
-----
(TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE), -- | U |
(TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE), -- | X |
(TRUE, TRUE, FALSE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE), -- | 0 |
(TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE), -- | 1 |
(TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE), -- | Z |
(TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE), -- | W |
(TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE), -- | L |
(TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE), -- | H |
(TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE));-- | - |
```

Thus, for example (notice we need type conversions):

```
IM_TRUE = STD_MATCH(STD_LOGIC_VECTOR ("10HLXWZ-"),
                    STD_LOGIC_VECTOR ("HL10----")) -- is TRUE
```

The following code is similar to the first simple example of Section 10.1, but illustrates the use of the Std_Logic_1164 and NUMERIC_STD packages:

```
entity Counter_1 is end; --1
  library STD; use STD.TEXTIO.all; --2
  library IEEE; use IEEE.STD_LOGIC_1164.all; --3
use work.NUMERIC_STD.all; --4
architecture Behave_2 of Counter_1 is --5
  signal Clock : STD_LOGIC := '0'; --6
  signal Count : UNSIGNED (2 downto 0) := "000"; --7
begin --8
  process begin --9
    wait for 10 ns; Clock <= not Clock; --10
```

```

    if (now > 340 ns) then wait;           --11
    end if;                                --12
end process;                               --13
process begin                              --14
    wait until (Clock = '0');             --15
    if (Count = 7)                         --16
        then Count <= "000";             --17
        else Count <= Count + 1;         --18
    end if;                                --19
end process;                               --20
process (Count) variable L: LINE; begin write(L, now); --21
write(L, STRING(" Count=")); write(L, TO_INTEGER(Count)); --22
writeline(output, L);                    --23
end process;                               --24
end;                                       --25

```

The preceding code looks similar to the code in Section 10.1 (and the output is identical), but there is more going on here:

- Line 3 is a library clause and a use clause for the `std_logic_1164` package, so you can use the `STD_LOGIC` type and the `NUMERIC_STD` package.
- Line 4 is a use clause for `NUMERIC_STD` package that was previously analyzed into the library work. If the package is instead analyzed into the library `IEEE`, you would use the name `IEEE.NUMERIC_STD.all` here. The `NUMERIC_STD` package allows you to use the type `UNSIGNED`.
- Line 6 declares `clock` to be type `STD_LOGIC` and initializes it to `'0'`, instead of the default initial value `STD_LOGIC'LEFT` (which is `'U'`).
- Line 7 declares `Count` to be a 3-bit array of type `UNSIGNED` from `NUMERIC_STD` and initializes it using a bit-string literal.
- Line 10 uses the overloaded `'not'` operator from `std_logic_1164`.
- Line 15 uses the overloaded `'='` operator from `std_logic_1164`.
- Line 16 uses the overloaded `'='` operator from `NUMERIC_STD`.
- Line 17 requires a bit-string literal, you cannot use `Count <= 0` here.
- Line 18 uses the overloaded `'+'` operator from `NUMERIC_STD`.
- Line 22 converts `Count`, type `UNSIGNED`, to type `INTEGER`.

10.13 Concurrent Statements

A **concurrent statement** [VHDL LRM9] is one of the following statements:

```

concurrent_statement ::=                               [10.36]
    block_statement
    | process_statement

```



```

| [ label : ] [ postponed ] procedure_call ;
| [ label : ] [ postponed ] assertion ;
| [ label : ] [ postponed ] conditional_signal_assignment
| [ label : ] [ postponed ] selected_signal_assignment
| component_instantiation_statement
| generate_statement

```

(The presence of the semicolons ‘;’ in some lines and absence in others in the preceding is correct.) The following sections describe each of these statements in turn.

10.13.1 Block Statement

A **block statement** has the following format [VHDL LRM9.1]:

```

block_statement ::=                                     [10.37]
  block_label: block [(guard_expression)] [is]
    [generic (generic_interface_list);
    [generic map (generic_association_list);]
    [port (port_interface_list);
    [port map (port_association_list);]
    {block_declarative_item}
    begin
      {concurrent_statement}
  end block [block_label] ;

```

Blocks may have their own ports and generics and may be used to split an architecture into several hierarchical parts (blocks can also be nested). As a very general rule, for the same reason that it is better to split a computer program into separate small modules, it is usually better to split a large architecture into smaller separate entity–architecture pairs rather than several nested blocks.

A block does have a unique feature: It is possible to specify a **guard expression** for a block. This creates a special signal, **GUARD**, that you can use within the block to control execution [VHDL LRM9.5]. It also allows you to model three-state buses by declaring **guarded signals** (signal kinds **register** and **bus**).

When you make an assignment statement to a signal, you define a **driver** for that signal. If you make assignments to guarded signals in a block, the driver for that signal is turned off, or **disconnected**, when the **GUARD** signal is **FALSE**. The use of guarded signals and guarded blocks can become quite complicated, and not all synthesis tools support these VHDL features.

The following example shows two drivers, A and B, on a three-state bus TSTATE, enabled by signals OEA and OEB. The drivers are enabled by declaring a guard expression after the block declaration and using the keyword guarded in the assignment statements. A **disconnect** statement [VHDL LRM5.3] models the driver delay from driving the bus to the high-impedance state (time to “float”).

```
library ieee; use ieee.std_logic_1164.all;
entity bus_drivers is end;

architecture Structure_1 of bus_drivers is
signal TSTATE: STD_LOGIC bus; signal A, B, OEA, OEB : STD_LOGIC:= '0';
begin
process begin OEA <= '1' after 100 ns, '0' after 200 ns;
OEB <= '1' after 300 ns; wait; end process;
B1 : block (OEA = '1')
disconnect all : STD_LOGIC after 5 ns; -- Only needed for float time.
begin TSTATE <= guarded not A after 3 ns; end block;
B2 : block (OEB = '1')
disconnect all : STD_LOGIC after 5 ns; -- Float time = 5 ns.
begin TSTATE <= guarded not B after 3 ns; end block;
end;
```

Time(fs) + Cycle	1	2	3	4	5	6	7
	tstate	a	b	oea	oeb	b1.GUARD	b2.GUARD
0+ 0:	'U'	'0'	'0'	'0'	'0'	FALSE	FALSE
0+ 1: *	'Z'	'0'	'0'	'0'	'0'	FALSE	FALSE
100000000+ 0:	'Z'	'0'	'0'	*'1'	'0'	* TRUE	FALSE
103000000+ 0: *	'1'	'0'	'0'	'1'	'0'	TRUE	FALSE
200000000+ 0:	'1'	'0'	'0'	*'0'	'0'	* FALSE	FALSE
200000000+ 1: *	'Z'	'0'	'0'	'0'	'0'	FALSE	FALSE
300000000+ 0:	'Z'	'0'	'0'	'0'	*'1'	FALSE *	TRUE
303000000+ 0: *	'1'	'0'	'0'	'0'	'1'	FALSE	TRUE

Notice the creation of implicit guard signals b1.GUARD and b2.GUARD for each guarded block. There is another, equivalent, method that uses the high-impedance value explicitly as in the following example:

```
architecture Structure_2 of bus_drivers is
signal TSTATE : STD_LOGIC; signal A, B, OEA, OEB : STD_LOGIC := '0';
begin
process begin
OEA <= '1' after 100 ns, '0' after 200 ns; OEB <= '1' after 300 ns;
wait; end process;
process(OEA, OEB, A, B) begin
if (OEA = '1') then TSTATE <= not A after 3 ns;
elsif (OEB = '1') then TSTATE <= not B after 3 ns;
else TSTATE <= 'Z' after 5 ns;
```

```

    end if;
end process;
end;
```

This last method is more widely used than the first, and what is more important, more widely accepted by synthesis tools. Most synthesis tools are capable of recognizing the value 'z' on the RHS of an assignment statement as a cue to synthesize a three-state driver. It is up to you to make sure that multiple drivers are never enabled simultaneously to cause contention.

10.13.2 Process Statement

A process statement has the following format [VHDL LRM9.2]:

```

process_statement ::= [10.38]
[process_label:]
[postponed] process [(signal_name {, signal_name})]
[is] {subprogram_declaration | subprogram_body
    | type_declaration | subtype_declaration
    | constant_declaration | variable_declaration
    | file_declaration | alias_declaration
    | attribute_declaration | attribute_specification
    | use_clause
    | group_declaration | group_template_declaration}
begin
    {sequential_statement}
end [postponed] process [process_label];
```

The following process models a 2:1 MUX (combinational logic):

```

entity Mux_1 is port (i0, i1, sel : in BIT := '0'; y : out BIT); end;
architecture Behave of Mux_1 is
    begin process (i0, i1, sel) begin -- i0, i1, sel = sensitivity set
        case sel is when '0' => y <= i0; when '1' => y <= i1; end case;
    end process; end;
```

This process executes whenever an event occurs on any of the signals in the process **sensitivity set** (i0, i1, sel). The execution of a process occurs during a simulation cycle—a delta cycle. Assignment statements to signals may trigger further delta cycles. Time advances when all transactions for the current time step are complete and all signals updated.

The following code models a two-input AND gate (combinational logic):

```

entity And_1 is port (a, b : in BIT := '0'; y : out BIT); end;
architecture Behave of And_1 is
    begin process (a, b) begin y <= a and b; end process; end;
```

The next example models a D flip-flop (sequential logic). The process statement is executed whenever there is an event on clk. The if statement updates the output q with the input d on the rising edge of the signal clk. If the if statement

condition is false (as it is on the falling edge of `clk`), then the assignment statement `q <= d` will not be executed, and `q` will keep its previous value. The process thus requires the value of `q` to be stored between successive process executions, and this implies sequential logic.

```
entity FF_1 is port (clk, d: in BIT := '0'; q : out BIT); end;
architecture Behave of FF_1 is
begin process (clk) begin
    if clk'EVENT and clk = '1' then q <= d; end if;
end process; end;
```

The behavior of the next example is identical to the previous model. Notice that the `wait` statement is at the end of the equivalent process with the signals in the sensitivity set (in this case just one signal, `clk`) included in the sensitivity list (that follows the keyword `on`).

```
entity FF_2 is port (clk, d: in BIT := '0'; q : out BIT); end;
architecture Behave of FF_2 is
begin process begin -- The equivalent process has a wait at the end:
    if clk'event and clk = '1' then q <= d; end if; wait on clk;
end process; end;
```

If we use a `wait` statement in a process statement, then we may not use a process sensitivity set (the reverse is true: If we do not have a sensitivity set for a process, we must include a `wait` statement or the process will execute endlessly):

```
entity FF_3 is port (clk, d: in BIT := '0'; q : out BIT); end;
architecture Behave of FF_3 is
begin process begin -- No sensitivity set with a wait statement.
    wait until clk = '1'; q <= d;
end process; end;
```

If you include ports (interface signals) in the sensitivity set of a process statement, they must be ports that can be read (they must be of mode `in`, `inout`, or `buffer`, see Section 10.7).

10.13.3 Concurrent Procedure Call

A **concurrent procedure call** appears outside a process statement [VHDL LRM9.3]. The concurrent procedure call is a shorthand way of writing an equivalent process statement that contains a procedure call (Section 10.10.4):

```
package And_Pkg is procedure V_And(a,b:BIT; signal c:out BIT); end;
package body And_Pkg is procedure V_And(a,b:BIT; signal c:out BIT) is
    begin c <= a and b; end; end And_Pkg;
use work.And_Pkg.all; entity Proc_Call_2 is end;
architecture Behave of Proc_Call_2 is signal A, B, Y : BIT := '0';
    begin V_And (A, B, Y); -- Concurrent procedure call.
```

```
process begin wait; end process; -- Extra process to stop.
end;
```

10.13.4 Concurrent Signal Assignment

There are two forms of **concurrent signal assignment statement**. A **selected signal assignment statement** is equivalent to a case statement inside a process statement [VHDL LRM9.5.2]:

```
selected_signal_assignment ::= [10.39]
  with expression select
    name|aggregate <= [guarded]
      [transport|reject time expression] inertial]
      waveform when choice { | choice}
      {, waveform when choice { | choice} } ;
```

The following design unit, Selected_1, uses a selected signal assignment. The equivalent unit, Selected_2, uses a case statement inside a process statement.

```
entity Selected_1 is end; architecture Behave of Selected_1 is
  signal y,i1,i2 : INTEGER; signal sel : INTEGER range 0 to 1;
begin with sel select y <= i1 when 0, i2 when 1; end;

entity Selected_2 is end; architecture Behave of Selected_2 is
  signal i1,i2,y : INTEGER; signal sel : INTEGER range 0 to 1;
begin process begin
  case sel is when 0 => y <= i1; when 1 => y <= i2; end case;
  wait on i1, i2;
end process; end;
```

The other form of concurrent signal assignment is a **conditional signal assignment statement** that, in its most general form, is equivalent to an if statement inside a process statement [VHDL LRM9.5.1]:

```
conditional_signal_assignment ::= [10.40]
  name|aggregate <= [guarded]
  [transport|reject time expression] inertial]
  {waveform when boolean_expression else}
  waveform [when boolean_expression];
```

Notice that in VHDL-93 the else clause is optional. Here is an example of a conditional signal assignment, followed by a model using the equivalent process with an if statement:

```
entity Conditional_1 is end; architecture Behave of Conditional_1 is
  signal y,i,j : INTEGER; signal clk : BIT;
begin y <= i when clk = '1' else j; -- conditional signal assignment
end;

entity Conditional_2 is end; architecture Behave of Conditional_2 is
  signal y,i : INTEGER; signal clk : BIT;
begin process begin
```

```

    if clk = '1' then y <= i; else y <= y ; end if; wait on clk;
end process; end;

```

A concurrent signal assignment statement can look just like a sequential signal assignment statement, as in the following example:

```

entity Assign_1 is end; architecture Behave of Assign_1 is
signal Target, Source : INTEGER;
    begin Target <= Source after 1 ns; -- looks like signal assignment
end;

```

However, outside a process statement, this statement is a concurrent signal assignment and has its own equivalent process statement. Here is the equivalent process for the example:

```

entity Assign_2 is end; architecture Behave of Assign_2 is
signal Target, Source : INTEGER;
begin process begin
    Target <= Source after 1 ns; wait on Source;
end process; end;

```

Every process is executed once during initialization. In the previous example, an initial value will be scheduled to be assigned to `Target` even though there is no event on `Source`. If, for some reason, you do not want this to happen, you need to rewrite the concurrent assignment statement as a process statement with a wait statement before the assignment statement:

```

entity Assign_3 is end; architecture Behave of Assign_3 is
signal Target, Source : INTEGER; begin process begin
    wait on Source; Target <= Source after 1 ns;
end process; end;

```

10.13.5 Concurrent Assertion Statement

A concurrent assertion statement is equivalent to a passive process statement (without a sensitivity list) that contains an assertion statement followed by a wait statement [VHDL LRM9.4].

```

concurrent_assertion_statement [10.41]
 ::= [ label : ] [ postponed ] assertion ;

```

If the assertion condition contains a signal, then the equivalent process statement will include a final wait statement with a sensitivity clause. A concurrent assertion statement with a condition that is static expression is equivalent to a process statement that ends in a wait statement that has no sensitivity clause. The equivalent process will execute once, at the beginning of simulation, and then wait indefinitely.

10.13.6 Component Instantiation

A **component instantiation statement** in VHDL is similar to placement of a component in a schematic—an instantiated component is somewhere between a copy of the component and a reference to the component. Here is the definition [VHDL LRM9.6]:

```
component_instantiation_statement ::= [10.42]
instantiation_label:
  [component] component_name
  [entity entity_name [(architecture identifier)]
  [configuration configuration_name
  [generic map (generic_association_list)]
  [port map (port_association_list)] ;
```

We examined component instantiation using a *component_name* in Section 10.5. If we instantiate a component in this way we must declare the component (see BNF [10.9]). To bind a component to an entity–architecture pair we can use a configuration, as illustrated in Figure 10.1, or we can use the default binding as described in Section 10.7. In VHDL-93 we have another alternative—we can directly instantiate an entity or configuration. For example:

```
entity And_2 is port (i1, i2 : in BIT; y : out BIT); end;
architecture Behave of And_2 is begin y <= i1 and i2; end;
entity Xor_2 is port (i1, i2 : in BIT; y : out BIT); end;
architecture Behave of Xor_2 is begin y <= i1 xor i2; end;

entity Half_Adder_2 is port (a,b : BIT := '0'; sum, cry : out BIT); end;
architecture Netlist_2 of Half_Adder_2 is
use work.all; -- need this to see the entities Xor_2 and And_2
begin
  X1 : entity Xor_2(Behave) port map (a, b, sum); -- VHDL-93 only
  A1 : entity And_2(Behave) port map (a, b, cry); -- VHDL-93 only
end;
```

10.13.7 Generate Statement

A **generate statement** [VHDL LRM9.7] simplifies repetitive code:

```
generate_statement ::= [10.43]
generate_label: for generate_parameter_specification
  [if boolean_expression
generate [{block declarative_item} begin]
  {concurrent_statement}
end generate [generate_label] ;
```

Here is an example (notice the labels are required):

```
entity Full_Adder is port (X, Y, Cin : BIT; Cout, Sum: out BIT); end;
architecture Behave of Full_Adder is begin Sum <= X xor Y xor Cin;
Cout <= (X and Y) or (X and Cin) or (Y and Cin); end;
```

```

entity Adder_1 is
  port (A, B : in BIT_VECTOR (7 downto 0) := (others => '0');
        Cin : in BIT := '0'; Sum : out BIT_VECTOR (7 downto 0);
        Cout : out BIT);
end;

architecture Structure of Adder_1 is use work.all;

component Full_Adder port (X, Y, Cin: BIT; Cout, Sum: out BIT);
end component;

signal C : BIT_VECTOR(7 downto 0);
begin AllBits : for i in 7 downto 0 generate
  LowBit : if i = 0 generate
    FA : Full_Adder port map (A(0), B(0), Cin, C(0), Sum(0));
  end generate;
  OtherBits : if i /= 0 generate
    FA : Full_Adder port map (A(i), B(i), C(i-1), C(i), Sum(i));
  end generate;
end generate;
Cout <= C(7);
end;

```

The instance names within a generate loop include the generate parameter. For example for $i=6$, FA'INSTANCE_NAME is

```
:adder_1(structure):allbits(6):otherbits:fa:
```

10.14 Execution

Two successive statements may execute in either a concurrent or sequential fashion depending on where the statements appear.

```
statement_1; statement_2;
```

In **sequential execution**, statement₁ in this sequence is always evaluated before statement 2. In **concurrent execution**, statement₁ and statement₂ are evaluated at the same time (as far as we are concerned—obviously on most computers exactly parallel execution is not possible). Concurrent execution is the most important difference between VHDL and a computer programming language. Suppose we have two signal assignment statements inside a process statement. In this case statement₁ and statement₂ are sequential assignment statements:

```

entity Sequential_1 is end; architecture Behave of Sequential_1 is
  signal s1, s2 : INTEGER := 0;
begin
  process begin
    s1 <= 1;           -- sequential signal assignment 1
    s2 <= s1 + 1;     -- sequential signal assignment 2
    wait on s1, s2 ;
  end process;
end;

```



```

end process;
end;

```

Time(fs) + Cycle	s1	s2
0+ 0:	0	0
0+ 1: *	1 *	1
0+ 2: *	1 *	2
0+ 3: *	1 *	2

If the two statements are outside a process statement they are concurrent assignment statements, as in the following example:

```

entity Concurrent_1 is end; architecture Behave of Concurrent_1 is
signal s1, s2 : INTEGER := 0; begin
  L1 : s1 <= 1;          -- concurrent signal assignment 1
  L2 : s2 <= s1 + 1;   -- concurrent signal assignment 2
end;

```

Time(fs) + Cycle	s1	s2
0+ 0:	0	0
0+ 1: *	1 *	1
0+ 2:	1 *	2

The two concurrent signal assignment statements in the previous example are equivalent to the two processes, labeled as P1 and P2, in the following model.

```

entity Concurrent_2 is end; architecture Behave of Concurrent_2 is
signal s1, s2 : INTEGER := 0; begin
  P1 : process begin s1 <= 1;      wait on s2 ; end process;
  P2 : process begin s2 <= s1 + 1; wait on s1 ; end process;
end;

```

Time(fs) + Cycle	s1	s2
0+ 0:	0	0
0+ 1: *	1 *	1
0+ 2: *	1 *	2
0+ 3: *	1	2

Notice that the results are the same (though the trace files are slightly different) for the architectures `Sequential_1`, `Concurrent_1`, and `Concurrent_2`. Updates to signals occur at the end of the simulation cycle, so the values used will always be the old values. So far things seem fairly simple: We have sequential execution or concurrent execution. However, variables are updated immediately, so the variable values that are used are always the new values. The examples in Table 10.17 illustrate this very important difference.

The various concurrent and sequential statements in VHDL are summarized in Table 10.18.

TABLE 10.17 Variables and signals in VHDL.

Variables	Signals
<pre>entity Execute_1 is end; architecture Behave of Execute_1 is begin process variable v1 : INTEGER := 1; variable v2 : INTEGER := 2; begin v1 := v2; -- before: v1 = 1, v2 = 2 v2 := v1; -- after: v1 = 2, v2 = 2 wait; end process; end;</pre>	<pre>entity Execute_2 is end; architecture Behave of Execute_2 is signal s1 : INTEGER := 1; signal s2 : INTEGER := 2; begin process begin s1 <= s2; -- before: s1 = 1, s2 = 2 s2 <= s1; -- after: s1 = 2, s2 = 1 wait; end process; end;</pre>

TABLE 10.18 Concurrent and sequential statements in VHDL.

Concurrent [VHDL LRM9]	Sequential [VHDL LRM8]	
block	wait	case
process	assertion	loop
concurrent_procedure_call	signal_assignment	next
concurrent_assertion	variable_assignment	exit
concurrent_signal_assignment	procedure_call	return
t	if	null
component_instantiation		
generate		

10.15 Configurations and Specifications

The difference between, the interaction, and the use of component/configuration declarations and specifications is probably the most confusing aspect of VHDL. Fortunately this aspect of VHDL is not normally important for ASIC design. The syntax of component/configuration declarations and specifications is shown in Table 10.19.

- A *configuration declaration* defines a configuration—it is a library unit and is one of the basic units of VHDL code.
- A *block configuration* defines the configuration of a block statement or a design entity. A block configuration appears inside a configuration declaration, a component configuration, or nested in another block configuration.

TABLE 10.19 VHDL binding.

configuration declaration ¹ [VHDL LRM1.3]	<pre> configuration identifier of entity_name is {use_clause attribute_specification group_declaration} block_configuration end <u>configuration</u> [configuration_identifier]; </pre>
block configuration [VHDL LRM1.3.1]	<pre> for architecture_name block_statement_label generate_statement_label [(index_specification)] {use selected_name {, selected_name};} {block_configuration component_configuration} end for ; </pre>
configuration specification ¹ [VHDL LRM5.2]	<pre> for instantiation_label{, instantiation_label}:component_name others:component_name all:component_name [use entity entity_name [(architecture_identifier)] configuration configuration_name open] [generic map (generic_association_list)] [port map (port_association_list)]; </pre>
component declaration ¹ [VHDL LRM4.5]	<pre> component identifier [<u>is</u>] [generic (local_generic_interface_list);] [port (local_port_interface_list);] end component [<u>component_identifier</u>]; </pre>
component configuration ¹ [VHDL LRM1.3.2]	<pre> for instantiation_label {, instantiation_label}:component_name others:component_name all:component_name [[use entity entity_name [(architecture_identifier)] configuration configuration_name open] [generic map (generic_association_list)] [port map (port_association_list)];] [block_configuration] end for; </pre>

¹Underline means "new to VHDL-93".

- A *configuration specification* may appear in the declarative region of a generate statement, block statement, or architecture body.
- A *component declaration* may appear in the declarative region of a generate statement, block statement, architecture body, or package.
- A *component configuration* defines the configuration of a component and appears in a block configuration.

Table 10.20 shows a simple example (identical in structure to the example of Section 10.5) that illustrates the use of each of the preceding constructs.

TABLE 10.20 VHDL binding examples.

	<pre>entity AD2 is port (A1, A2: in BIT; Y: out BIT); end; architecture B of AD2 is begin Y <= A1 and A2; end; entity XR2 is port (X1, X2: in BIT; Y: out BIT); end; architecture B of XR2 is begin Y <= X1 xor X2; end;</pre>
component declaration configuration specification	<pre>entity Half_Adder is port (X, Y: BIT; Sum, Cout: out BIT); end; architecture Netlist of Half_Adder is use work.all; component MX port (A, B: BIT; Z :out BIT);end component; component MA port (A, B: BIT; Z :out BIT);end component; for G1:MX use entity XR2(B) port map(X1 => A,X2 => B,Y => Z); begin G1:MX port map(X, Y, Sum); G2:MA port map(X, Y, Cout); end;</pre>
configuration declaration block configuration component configuration	<pre>configuration C1 of Half_Adder is use work.all; for Netlist for G2:MA use entity AD2(B) port map(A1 => A,A2 => B,Y => Z); end for; end for; end;</pre>

10.16 An Engine Controller

This section describes part of a controller for an automobile engine. Table 10.21 shows a temperature converter that converts digitized temperature readings from a sensor from degrees Centigrade to degrees Fahrenheit.

To save area the temperature conversion is approximate. Instead of multiplying by 9/5 and adding 32 (so 0°C becomes 32°F and 100°C becomes 212°F) we multiply by 1.75 and add 32 (so 100°C becomes 207°F). Since $1.75 = 1 + 0.5 + 0.25$, we can multiply by 1.75 using shifts (for divide by 2, and divide by 4) together with a very simple constant addition (since $32 = "100000"$). Using shift to multiply and divide by powers of 2 is free in hardware (we just change connections to a bus). For

TABLE 10.21 A temperature converter.

```

library IEEE;
use IEEE.STD_LOGIC_1164.all; -- type STD_LOGIC, rising_edge
use IEEE.NUMERIC_STD.all ; -- type UNSIGNED, "+", "/"
entity tconv is generic TPD : TIME:= 1 ns;
  port (T : in UNSIGNED(11 downto 0);
        T_out : out UNSIGNED(11 downto 0));
end;
architecture rtl of tconv is
  constant T2 : UNSIGNED(1 downto 0) := "10" ;
  constant T4 : UNSIGNED(2 downto 0) := "100" ;
  constant T32 : UNSIGNED(5 downto 0) := "100000" ;
begin
  process(T) begin
    T_out <= T + T/T2 + T/T4 + T32 after TPD;
  end process;
end rtl;

```

T = temperature in °C

T_{out} = temperature in °F

The conversion formula from Centigrade to Fahrenheit is:

$$T(^{\circ}\text{F}) = (9/5) \times T(^{\circ}\text{C}) + 32$$

This converter uses the approximation:

$$9/5 \approx 1.75 = 1 + 0.5 + 0.25$$

large temperatures the error approaches 0.05/1.8 or approximately 3 percent. We play these kinds of tricks often in hardware computation. Notice also that temperatures measured in °C and °F are defined as unsigned integers of the same width. We could have defined these as separate types to take advantage of VHDL's type checking.

Table 10.22 describes a digital filter to compute a “moving average” over four successive samples in time ($i(0)$, $i(1)$, $i(2)$, and $i(3)$), with $i(0)$ being the first sample).

The filter uses the following formula:

$$T_{out} \leq (i(0) + i(1) + i(2) + i(3)) / T4$$

Division by $T4 = "100"$ is free in hardware. If instead, we performed the divisions before the additions, this would reduce the number of bits to be added for two of the additions and saves us worrying about overflow. The drawback to this approach is round-off errors. We can use the register shown in Table 10.23 to register the inputs.

Table 10.24 shows a **first-in, first-out** stack (FIFO). This allows us to buffer the signals coming from the sensor until the microprocessor has a chance to read them. The depth of the FIFO will depend on the maximum amount of time that can pass without the microcontroller being able to read from the bus. We have to determine this with statistical simulations taking into account other traffic on the bus.

TABLE 10.22 A digital filter.

```

library IEEE;
use IEEE.STD_LOGIC_1164.all; -- STD_LOGIC type, rising_edge
use IEEE.NUMERIC_STD.all; -- UNSIGNED type, "+" and "/"
entity filter is
  generic TPD : TIME := 1 ns;
  port (T_in : in UNSIGNED(11 downto 0);
        rst, clk : in STD_LOGIC;
        T_out: out UNSIGNED(11 downto 0));
end;
architecture rtl of filter is
type arr is array (0 to 3) of UNSIGNED(11 downto 0);
signal i : arr ;
constant T4 : UNSIGNED(2 downto 0) := "100";
begin
  process(rst, clk) begin
    if (rst = '1') then
      for n in 0 to 3 loop i(n) <= (others =>'0') after TPD;
      end loop;
    else
      if(rising_edge(clk)) then
        i(0) <= T_in after TPD;i(1) <= i(0) after TPD;
        i(2) <= i(1) after TPD;i(3) <= i(2) after TPD;
        end if;
      end if;
    end process;
  process(i) begin
    T_out <= ( i(0) + i(1) + i(2) + i(3) )/T4 after TPD;
  end process;
end rtl;

```

The filter computes a moving average over four successive samples in time.

Notice

$i(0)$ $i(1)$ $i(2)$ $i(3)$
are each 12 bits wide.

Then the sum

$i(0) + i(1) + i(2) + i(3)$
is 14 bits wide, and the average

$(i(0) + i(1) + i(2) + i(3))/T4$

is 12 bits wide.

All delays are generic TPD.

The FIFO has flags, empty and full, that signify its state. It uses a function to increment two circular pointers. One pointer keeps track of the address to write to next, the other pointer tracks the address to read from. The FIFO memory may be implemented in a number of ways in hardware. We shall assume for the moment that it will be synthesized as a bank of flip-flops.

Table 10.25 shows a controller for the two FIFOs. The controller handles the reading and writing to the FIFO. The microcontroller attached to the bus signals which of the FIFOs it wishes to read from. The controller then places the appropriate data on the bus. The microcontroller can also ask for the FIFO flags to be placed in the low-order bits of the bus on a read cycle. If none of these actions are requested by the microcontroller, the FIFO controller three-states its output drivers.

TABLE 10.23 The input register.

```

library IEEE;
use IEEE.STD_LOGIC_1164.all; -- type STD_LOGIC, rising_edge
use IEEE.NUMERIC_STD.all ; -- type UNSIGNED
entity register_in is
generic ( TPD : TIME := 1 ns);
port (T_in : in UNSIGNED(11 downto 0);
clk, rst : in STD_LOGIC; T_out : out UNSIGNED(11 downto 0)); end;
architecture rtl of register_in is
begin
  process(clk, rst) begin
    if (rst = '1') then T_out <= (others => '0') after TPD;
    else
      if (rising_edge(clk)) then T_out <= T_in after TPD; end if;
    end if;
  end process;
end rtl ;

```

12-bit-wide register for the temperature input signals.

If the input is asynchronous (from an A/D converter with a separate clock, for example), we would need to worry about metastability.

All delays are generic TPD.

Table 10.25 shows the top level of the controller. To complete our model we shall use a package for the component declarations:

```

package TC_Components is
component register_in generic (TPD : TIME := 1 ns);
port (T_in : in UNSIGNED(11 downto 0);
      clk, rst : in STD_LOGIC; T_out : out UNSIGNED(11 downto 0));
end component;
component tconv generic (TPD : TIME := 1 ns);
port (T : in UNSIGNED (11 downto 0);
      T_out : out UNSIGNED(11 downto 0));
end component;
component filter generic (TPD : TIME := 1 ns);
port (T_in : in UNSIGNED (11 downto 0);
      rst, clk : in STD_LOGIC; T_out : out UNSIGNED(11 downto 0));
end component;
component fifo generic (width:INTEGER := 12; depth : INTEGER := 16);
port (clk, rst, push, pop : STD_LOGIC;
      Di : UNSIGNED (width-1 downto 0);
      Do : out UNSIGNED (width-1 downto 0);
      empty, full : out STD_LOGIC);
end component;
component fifo_control generic (TPD:TIME := 1 ns);
port (D_1, D_2 : in UNSIGNED(11 downto 0);
      select : in UNSIGNED(1 downto 0); read, f1, f2, e1, e2 : in STD_LOGIC;
      r1, r2, w12 : out STD_LOGIC; D : out UNSIGNED(11 downto 0)) ;

```

TABLE 10.24 A first-in, first-out stack (FIFO).

```

library IEEE; use IEEE.NUMERIC_STD.all ; -- UNSIGNED type
use ieee.std_logic_1164.all; -- STD_LOGIC type, rising_edge
entity fifo is
  generic (width : INTEGER := 12; depth : INTEGER := 16);
  port (clk, rst, push, pop : STD_LOGIC;
        Di : in UNSIGNED (width-1 downto 0);
        Do : out UNSIGNED (width-1 downto 0);
        empty, full : out STD_LOGIC);
end fifo;
architecture rtl of fifo is
  subtype ptype is INTEGER range 0 to (depth-1);
  signal diff, Ai, Ao : ptype; signal f, e : STD_LOGIC;
  type a is array (ptype) of UNSIGNED(width-1 downto 0);
  signal mem : a ;
  function bump(signal ptr : INTEGER range 0 to (depth-1))
  return INTEGER is begin
    if (ptr = (depth-1)) then return 0;
    else return (ptr + 1);
    end if;
end;
begin
  process(f,e) begin full <= f ; empty <= e; end process;
  process(diff) begin
    if (diff = depth -1) then f <= '1'; else f <= '0'; end if;
    if (diff = 0) then e <= '1'; else e <= '0'; end if;
  end process;
  process(clk, Ai, Ao, Di, mem, push, pop, e, f) begin
    if(rising_edge(clk)) then
      if(push='0')and(pop='1')and(e='0') then Do <= mem(Ao); end if;
      if(push='1')and(pop='0')and(f='0') then mem(Ai) <= Di; end if;
    end if ;
  end process;
  process(rst, clk) begin
    if(rst = '1') then Ai <= 0; Ao <= 0; diff <= 0;
    else if(rising_edge(clk)) then
      if (push = '1') and (f = '0') and (pop = '0') then
        Ai <= bump(Ai); diff <= diff + 1;
      elsif (pop = '1') and (e = '0') and (push = '0') then
        Ao <= bump(Ao); diff <= diff - 1;
      end if;
    end if;
  end if;
end process;
end;

```

FIFO (first-in, first-out) register

Reads (pop = 1) and writes (push = 1) are synchronous to the rising edge of the clock. Read and write should not occur at the same time. The width (number of bits in each word) and depth (number of words) are generics.

External signals:

clk, clock
rst, reset active-high
push, write to FIFO
pop, read from FIFO
Di, data in
Do, data out
empty, FIFO flag
full, FIFO flag

Internal signals:

diff, difference pointer
Ai, input address
Ao, output address
f, full flag
e, empty flag

No delays in this model.

TABLE 10.25 A FIFO controller.

```

library IEEE;use IEEE.STD_LOGIC_1164.all;use IEEE.NUMERIC_STD.all;
entity fifo_control is generic TPD : TIME := 1 ns;
  port(D_1, D_2 : in UNSIGNED(11 downto 0);
  sel : in UNSIGNED(1 downto 0) ;
  read , f1, f2, e1, e2 : in STD_LOGIC;
  r1, r2, w12 : out STD_LOGIC; D : out UNSIGNED(11 downto 0)) ;
end;
architecture rtl of fifo_control is
  begin process
    (read, sel, D_1, D_2, f1, f2, e1, e2)
  begin
    r1 <= '0' after TPD; r2 <= '0' after TPD;
    if (read = '1') then
      w12 <= '0' after TPD;
      case sel is
        when "01" => D <= D_1 after TPD; r1 <= '1' after TPD;
        when "10" => D <= D_2 after TPD; r2 <= '1' after TPD;
        when "00" => D(3) <= f1 after TPD; D(2) <= f2 after TPD;
          D(1) <= e1 after TPD; D(0) <= e2 after TPD;
        when others => D <= "ZZZZZZZZZZZZ" after TPD;
      end case;
    elsif (read = '0') then
      D <= "ZZZZZZZZZZZZ" after TPD; w12 <= '1' after TPD;
    else D <= "ZZZZZZZZZZZZ" after TPD;
    end if;
  end process;
end rtl;

```

This handles the reading and writing to the FIFOs under control of the processor (mpu). The mpu can ask for data from either FIFO or for status flags to be placed on the bus.

Inputs:

D_1

data in from FIFO1

D_2

data in from FIFO2

sel

FIFO select from mpu

read

FIFO read from mpu

f1,f2,e1,e2

flags from FIFOs

Outputs:

r1, r2

read enables for FIFOs

w12

write enable for FIFOs

D

data out to mpu bus

```

end component;
end;

```

The following testbench completes a set of reads and writes to the FIFOs:

```

library IEEE;
use IEEE.std_logic_1164.all; -- type STD_LOGIC
use IEEE.numeric_std.all; -- type UNSIGNED
entity test_TC is end;
architecture testbench of test_TC is
  component T_Control port (T_1, T_2 : in UNSIGNED(11 downto 0);
    clk : in STD_LOGIC; sensor: in UNSIGNED( 1 downto 0) ;
    read : in STD_LOGIC; rst : in STD_LOGIC;
    D : out UNSIGNED(11 downto 0)); end component;
  signal T_1, T_2 : UNSIGNED(11 downto 0);

```

TABLE 10.26 Top level of temperature controller.

```

library IEEE; use IEEE.STD_LOGIC_1164.all; use IEEE.NUMERIC_STD.all;
entity T_Control is port (T_in1, T_in2 : in UNSIGNED (11 downto 0);
  sensor: in UNSIGNED(1 downto 0);
  clk, RD, rst : in STD_LOGIC; D : out UNSIGNED(11 downto 0));
end;
architecture structure of T_Control is use work.TC_Components.all;
signal F, E : UNSIGNED (2 downto 1);
signal T_out1, T_out2, R_out1, R_out2, F1, F2, FIFO1, FIFO2 : UNSIGNED(11 downto 0);
signal RD1, RD2, WR: STD_LOGIC ;
begin
RG1 : register_in generic map (1ns) port map (T_in1, clk, rst, R_out1);
RG2 : register_in generic map (1ns) port map (T_in2, clk, rst, R_out2);
TC1 : tconv generic map (1ns) port map (R_out1, T_out1);
TC2 : tconv generic map (1ns) port map (R_out2, T_out2);
TF1 : filter generic map (1ns) port map (T_out1, rst, clk, F1);
TF2 : filter generic map (1ns) port map (T_out2, rst, clk, F2);
FI1 : fifo generic map (12,16) port map (clk, rst, WR, RD1, F1, FIFO1, E(1), F(1));
FI2 : fifo generic map (12,16) port map (clk, rst, WR, RD2, F2, FIFO2, E(2), F(2));
FC1 : fifo_control port map
(FIFO1, FIFO2, sensor, RD, F(1), F(2), E(1), E(2), RD1, RD2, WR, D);
end structure;

```

```

signal clk, read, rst : STD_LOGIC;
signal sensor : UNSIGNED(1 downto 0);
signal D : UNSIGNED(11 downto 0);
begin TT1 : T_Control port map (T_1, T_2, clk, sensor, read, rst, D);
process begin
rst <= '0'; clk <= '0';
wait for 5 ns; rst <= '1'; wait for 5 ns; rst <= '0';
T_1 <= "0000000000011"; T_2 <= "000000000111"; read <= '0';
  for i in 0 to 15 loop -- fill the FIFOs
    clk <= '0'; wait for 5ns; clk <= '1'; wait for 5 ns;
  end loop;
  assert (false) report "FIFOs full" severity NOTE;
  clk <= '0'; wait for 5ns; clk <= '1'; wait for 5 ns;
read <= '1'; sensor <= "01";
  for i in 0 to 15 loop -- empty the FIFOs
    clk <= '0'; wait for 5ns; clk <= '1'; wait for 5 ns;
  end loop;
  assert (false) report "FIFOs empty" severity NOTE;
  clk <= '0'; wait for 5ns; clk <= '1'; wait;
end process;
end;

```

10.17 Summary

Table 10.27 shows the essential elements of the VHDL language. Table 10.28 shows the most important BNF definitions and their locations in this chapter. The key points covered in this chapter are as follows:

- The use of an `entity` and an `architecture`
- The use of a `configuration` to bind entities and their architectures
- The compile, elaboration, initialization, and simulation steps
- Types, subtypes, and their use in expressions
- The logic systems based on `BIT` and `Std_Logic_1164` types
- The use of the IEEE synthesis packages for `BIT` arithmetic
- Ports and port modes
- Initial values and the difference between simulation and hardware
- The difference between a `signal` and a `variable`
- The different assignment statements and the timing of updates
- The `process` and `wait` statements

VHDL is a “wordy” language. The examples in this chapter are complete rather than code fragments. To write VHDL “nicely,” with indentation and nesting of constructs, requires a large amount of space. Some of the VHDL code examples in this chapter are deliberately dense (with reduced indentation and nesting), but the bold keywords help you to see the code structure. Most of the time, of course, we do not have the luxury of bold fonts (or color) to highlight code. In this case, you should add additional space, indentation, nesting, and comments.

Appendix A contains more detailed definitions and technical reference material.

TABLE 10.27 VHDL summary.

VHDL feature	Example	Book	93LRM
Comments	<code>-- this is a comment</code>	10.3	13.8
Literals (fixed-value items)	<code>12 1.0E6 '1' "110" 'Z' 2#1111_1111# "Hello world" STRING("110")</code>	10.4	13.4
Identifiers (case-insensitive, start with letter)	<code>a_good_name Same same 2_Bad bad _bad very_bad</code>	10.4	13.3
Several basic units of code	<code>entity architecture configuration</code>	10.5	1.1-1.3
Connections made through ports	<code>port (signal in i : BIT; out o : BIT);</code>	10.7	4.3
Default expression	<code>port (i : BIT := '1'); -- i='1' if left open</code>	10.7	4.3
No built-in logic-value system. BIT and BIT_VECTOR (STD).	<code>type BIT is ('0', '1'); -- predefined signal myArray: BIT_VECTOR (7 downto 0);</code>	10.8	14.2
Arrays	<code>myArray(1 downto 0) <= ('0', '1');</code>	10.8	3.2.1
Two basic types of logic signals	a signal corresponds to a real wire a variable is a memory location in RAM	10.9	4.3.1.2 4.3.1.3
Types and explicit initial/default value	<code>signal ONE : BIT := '1' ;</code>	10.9	4.3.2
Implicit initial/default value	<code>BIT'LEFT = '0'</code>	10.9	4.3.2
Predefined attributes	<code>clk'EVENT, clk'STABLE</code>	10.9.4	14.1
Sequential statements inside processes model things that happen one after another and repeat	<code>process begin wait until alarm = ring; eat; work; sleep; end process;</code>	10.10	8
Timing with wait statement	<code>wait for 1 ns; -- not wait 1 ns wait on light until light = green;</code>	10.10.1	8.1
Update to signals occurs at the end of a simulation cycle	<code>signal <= 1; -- delta time delay signal <= variable1 after 2 ns;</code>	10.10.3	8.3
Update to variables is immediate	<code>variable := 1; -- immediate update</code>	10.10.3	8.4
Processes and concurrent statements model things that happen at the same time	<code>process begin rain ; end process; process begin sing ; end process; process begin dance; end process;</code>	10.13	9.2
IEEE Std_Logic_1164 (defines logic operators on 1164 types)	<code>STD_ULOGIC, STD_LOGIC, STD_ULOGIC_VECTOR, and STD_LOGIC_VECTOR type STD_ULOGIC is ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-');</code>	10.6	—
IEEE Numeric_Bit and Numeric_Std (defines arithmetic operators on BIT and 1164 types)	<code>UNSIGNED and SIGNED X <= "10" * "01" -- OK with numeric pkgs.</code>	10.12	—

TABLE 10.28 VHDL definitions.

Structure	Page	BNF	Structure	Page	BNF
alias declaration	418	10.21	next statement	429	10.32
architecture body	394	10.8	null statement	430	10.35
assertion statement	423	10.25	package declaration	398	10.11
attribute declaration	418	10.22	port interface declaration	406	10.13
block statement	438	10.37	port interface list	406	10.12
case statement	428	10.30	primary unit	393	10.5
component declaration	395	10.9	procedure call statement	427	10.28
component instantiation	444	10.42	process statement	440	10.38
concurrent statement	437	10.36	return statement	430	10.34
conditional signal assignment	442	10.40	secondary unit	393	10.6
configuration declaration	396	10.10	selected signal assignment	442	10.39
constant declaration	414	10.16	sequential statement	419	10.23
declaration	413	10.15	signal assignment statement	424	10.27
design file	393	10.4	signal declaration	414	10.17
entity declaration	394	10.7	special character	391	10.2
exit statement	430	10.33	subprogram body	416	10.20
generate statement	444	10.43	subprogram declaration	415	10.19
graphic character	391	10.1	type declaration	411	10.14
identifier	392	10.3	variable assignment statement	424	10.26
if statement	427	10.29	variable declaration	415	10.18
loop statement	429	10.31	wait statement	421	10.24

10.18 Problems

*=Difficult **=Very difficult ***=Extremely difficult

10.1 (Hello World, 10 min.) Set up a new, empty, directory (use `mkdir VHDL`, for example) to run your VHDL simulator (the exact details will depend on your computer and simulator). Copy the code below to a file called `hw_1.vhd` in your VHDL directory (leave out comments to save typing). *Hint:* Use the `vi` editor (`i` inserts text, `x` deletes text, `dd` deletes a line, `ESC :w` writes the file, `ESC :q` quits) or use `cat > hw_1.vhd` and type in the code (use `CTRL-D` to end typing) on a UNIX machine. Remember to save in 'Text Only' mode (Frame or MS Word) on an IBM PC or Apple Macintosh.

Analyze, elaborate, and simulate your model (include the output in your answer). Comment on how easy or hard it was to follow the instructions to use the software and suggest improvements.

```
entity HW_1 is end; architecture Behave of HW_1 is
constant M : STRING := "hello, world"; signal Ch : CHARACTER := ' ';
begin process begin
  for i in M'RANGE loop Ch <= M(i); wait for 1 ns; end loop; wait;
end process; end;
```

10.2 (Running a VHDL simulation, 20 min.) Copy the example from Section 10.1 into a file called `Counter1.vhd` in your VHDL directory (leave out the comments to save typing). Complete the compile (analyze), elaborate (build), and execute (initialize and simulate) or other equivalent steps for your simulator. After each step list the contents of your directory VHDL and any subdirectories and files that are created (use `ls -aLR` on a UNIX system).

10.3 (Simulator commands, 10 min.) Make a “cheat sheet” for your simulator, listing the commands that can be used to control simulation.

10.4 (BNF addresses, 10 min.) Create a BNF description of a name including: optional title (Prof., Dr., Mrs., Mr., Miss, or Ms.), optional first name and middle initials (allow up to two), and last name (including unusual hyphenated and foreign names, such as Miss A-S. de La Salle, and Prof. John T. P. McTavish-fFiennes). The lowest level constructs are `letter ::= a-z`, `'.'` (period) and `'-'` (hyphen). Add BNF productions for a postal address in the form: company name, mail stop, street address, address lines (1 to 4), and country.

10.5 (BNF e-mail, 10 min.) Create a BNF description of a valid internet e-mail address in terms of letters, `'@'`, `'.'`, `'gov'`, `'com'`, `'org'`, and `'edu'`. Create a state diagram that “parses” an e-mail address for validity.

10.6 (BNF equivalence) Are the following BNF productions exactly equivalent? If they are not, produce a counterexample that shows a difference.

```
term ::= factor { multiplying_operator factor }
term ::= factor | term multiplying_operator factor
```

10.7 (Environment, 20 min.) Write a simple VHDL model to check and demonstrate that you can get to the IEEE library and have the environment variables, library statements, and such correctly set up for your simulator.

10.8 (Work, 20 min.) Write simple VHDL models to demonstrate that you can retrieve and use previously analyzed design units from the library `work` and that you can also remove design units from `work`. Explain how your models prove that access to `work` is functioning correctly.

10.9 (Packages, 60 min.) Write a simple package (use filename `PackH.vhd`) and package body (filename `PackB.vhd`). Demonstrate that you can store your package (call it `MyPackage`) in the library `work`. Then store, move, or rename (the details will depend on your software) your package to a library called `MyLibrary` in a directory called `MyDir`, and use its contents with a library clause (`library MyLibrary`) and a use clause (`use MyLibrary.MyPackage.all`) in a testbench called `PackTest` (filename `PackT.vhd`) in another directory `MyWork`. You may or may not be amazed at how complicated this can be and how poorly most software companies document this process.

10.10 (**IEEE Std 1164, 60 min.) Prior to VHDL-93 the `xnor` function was not available, and therefore older versions of the `std_logic_1164` library did not provide the `xnor` function for `STD_LOGIC` types either (it was actually included but commented out). Write a simple model that checks to see if you have the newer version of `std_logic_1164`. Can you do this without crashing the simulator?

You are an engineer on a very large project and find that your design fails to compile because your design must use the `xnor` function and the library setup on your company's system still points to the old IEEE `std_logic_1164` library, even though the new library was installed. You are apparently the first person to realize the problem. Your company has a policy that any time a library is changed all design units that use that library must be rebuilt from source. This might require days or weeks of work. Explain in detail, using code, the alternative solutions. What will you recommend to your manager?

10.11 (**VHDL-93 test, 20 min.) Write a simple test to check if your simulator is a VHDL-87 or VHDL-93 environment—without crashing the simulator.

10.12 (Declarations, 10 min.) Analyze the following changes to the code in Section 10.8 and include the simulator output in your answers:

- a. Uncomment the declarations for `Bad100` and `Bad4` in `Declaration_1`.
- b. Add the following to `Constant_2`:

```
signal wacky : wackytype (31 downto 0); -- wacky
```

- c. Remove the library and use clause in `Constant_2`.

10.13 (STRING type, 10 min.) Replace the `write` statement that prints the string " count=" in `Text(Behave)` in Section 10.6.3 with the following, compile it, and explain the result:

```
write(L, " count=" ); -- No type qualification.
```

10.14 (Sequential statements, 10 min.) Uncomment the following line in `wait_1(Behave)` in Section 10.10, analyze the code, and explain the result:

```
wait on x(1 to v); -- v is a variable.
```

10.15 (VHDL logical operators, 10 min.)

a. Explain the problem with the following VHDL statement:

```
Z <= A nand B nand C;
```

b. Explain why this problem does not occur with this statement:

```
Z <= A and B and C;
```

c. What can you say about the logical operators: and, or, nand, nor, xnor, xor?

d. Is the following code legal?

```
Z <= A and B or C;
```

10.16 (*Initialization, 45 min.) Consider the following code:

```
entity DFF_Plain is port (Clk, D : in BIT; Q : out BIT); end;
architecture Bad of DFF_Plain is begin process (Clk) begin
  if Clk = '0' and Clk'EVENT then Q <= D after 1 ns; end if;
end process; end;
```

a. Analyze and simulate this model using a testbench.

b. Rewrite architecture `Bad` using an equivalent process including a wait statement. Simulate this equivalent model and confirm the behaviors are identical.

c. What is the behavior of the output `Q` during initial execution of the process?

d. Why does this happen?

e. Why does this not happen with the following code:

```
architecture Good of DFF_Plain is
begin process begin wait until Clk = '0'; Q <= D after 1 ns;
end process; end;
```

10.17 (Initial and default values, 20 min.) Use code examples to explain the difference between: default expression, default value, implicit default value, initial value, initial value expression, and default initial value.

10.18 (Enumeration types, 20 min.) Explain the analysis results for the following:

```
type MVL4 is ('X', '0', '1', 'Z'); signal test : MVL4;
process begin
  test <= 1; test <= Z; test <= z; test <= '1'; test <= 'Z';
end process;
```

Alter the type declaration to the following, analyze your code again, and comment:

```
type Mixed4 is (X , '0', '1', Z);
```


10.19 (Type declarations, 10 min.) Correct these declarations:

```
type BadArray is array (0 to 7) of BIT_VECTOR;
type Byte is array (NATURAL range 7 downto 0) of BIT;
subtype BadNibble is Byte(3 downto 0);
type BadByte is array (range 7 downto 0) of BIT;
```

10.20 (Procedure parameters, 10 min.) Analyze the following package; explain and correct the error. Finally, build a testbench to check your solution.

```
package And_Pkg_Bad is procedure V_And(a, b : BIT; c: out BIT); end;
package body And_Pkg_Bad is
procedure V_And(a,b : BIT;c : out BIT) is begin c <= a and b;end;
end And_Pkg_Bad;
```

10.21 (Type checking, 20 min.) Test the following code and explain the results:

```
type T is INTEGER range 0 to 32; variable a: T;
a := (16 + 17) - 12; a := 16 - 12 + 17; a := 16 + (17 - 12);
```

10.22 (Debugging VHDL code, 30 min.) Find and correct the errors in the following code. Create a testbench for your code to check that it works correctly.

```
entity UpDownCount_Bad is
port(clock, reset, up: STD_LOGIC; D: STD_LOGIC_VECTOR (7 to 0));
end UpDownCount_Bad;

architecture Behave of UpDownCount_Bad is
begin process (clock, reset, up); begin
if (reset = '0') then D <= '0000000';
elseif (rising_edge(clock)) then
if (up = 1) D <= D+1; else D <= D-1; end if;
end if; end process; end Behave;
```

10.23 (Subprograms, 20 min.) Write and test subprograms for these declarations:

```
function Is_X_Zero (signal X : in BIT) return BIT;
procedure Is_A_Eq_B (signal A, B : BIT; signal Y : out BIT);
```

10.24 (Simulator error messages, 10 min.) Analyze and attempt to simulate `Arithmetic_2(Behave)` from Section 10.12 and compare the error message you receive with that from the MTI simulator (not all simulators are as informative). There are no standards for error messages.

10.25 (Exhaustive property of case statement, 30 min.) Write and simulate a testbench for the state machine of Table 10.8 and include your results. Is every state transition tested by your program and is every transition covered by an assignment statement in the code? (*Hint*: Think very carefully.) Repeat this exercise for the state machine in Section 10.10.6.

10.26 (Default values for inputs, 20 min.) Replace the interface declaration for entity `Half_Adder` in Section 10.5 with the following (to remove the default values):

```
port (X, Y: in BIT ; Sum, Cout: out BIT);
```

Attempt to compile, elaborate, and simulate configuration `Simplest` (the other entities needed, `AndGate` and `XorGate`, must already be in `work` or in the same file). You should get an error at some stage (different systems find this error at different points—just because an entity compiles, that does not mean it is error-free).

The LRM says “... A port of mode in may be unconnected ...only if its declaration includes a **default expression...**” [VHDL 93LRM1.1.1.2].

We face a dilemma here. If we do not drive inputs with test signals and leave an input port unconnected, we can compile the model (since it is syntactically correct) but the model is not semantically correct. On the other hand, if we give the inputs default values, we might accidentally forget to make a connection and not notice.

10.27 (Adder generation, 10 min.) Draw the schematic for `Adder_1` (Structure) of Section 10.13.7, labeling each instance with the VHDL instance name.

10.28 (Generate statement, 20 min.) Draw a schematic corresponding to the following code (label the cells with their instance names):

```
B1: block begin L1 : C port map (T, B, A(0), B(0)) ;
L2: for i in 1 to 3 generate L3 : for j in 1 to 3 generate
L4: if i+j > 4 generate L5: C port map (A(i-1), B(j-1), A(i), B(j)) ;
end generate; end generate; end generate;
L6: for i in 1 to 3 generate L7: for j in 1 to 3 generate
L8: if i+j < 4 generate L9: C port map (A(i+1), B(j+1), A(i), B(j)) ;
end generate; end generate; end generate;
end block B1;
```

Rewrite the code without generate statements. How would you prove that your code really is exactly equivalent to the original?

10.29 (Case statement, 20 min.) Create a package (`my_equal`) that overloads the equality operator so that `'x'='0'` and `'x'='1'` are both `TRUE`. Test your package. Simulate the following design unit and explain the result.

```
entity Case_1 is end; architecture Behave of Case_1 is
signal r : BIT; use work.my_equal.all;
begin process variable twobit:STD_LOGIC_VECTOR(1 to 2); begin
twobit := "X0";
case twobit is
when "10" => r <= '1';
when "00" => r <= '1';
when others => r <= '0';
end case; wait;
end process; end;
```

10.30 (State machine) Create a testbench for the state machine of Section 10.2.5.

10.31 (Mealy state machine, 60 min.) Rewrite the state machine of Section 10.2.5 as a Mealy state machine (the outputs depend on the inputs and on the current state).

10.32 (Gate-level D flip-flop, 30 min.) Draw the schematic for the following D flip-flop model. Create a testbench (check for correct operation with combinations of Clear, Preset, Clock, and Data). Have you covered all possible modes of operation? Justify your answer of yes or no.

```
architecture RTL of DFF_To_Test is
  signal A, B, C, D, QI, QBarI : BIT; begin
  A <= not (Preset and D and B) after 1 ns;
  B <= not (A and Clear and Clock) after 1 ns;
  C <= not (B and Clock and D) after 1 ns;
  D <= not (C and Clear and Data) after 1 ns;
  QI <= not (Preset and B and QBarI) after 1 ns;
  QBarI <= not (QI and Clear and C) after 1 ns;
  Q <= QI; QBar <= QBarI;
end;
```

10.33 (Flip-flop model, 20 min.) Add an asynchronous active-low preset to the D flip-flop model of Table 10.3. Generate a testbench that includes interaction of the preset and clear inputs. What issue do you face and how did you solve it?

10.34 (Register, 45 min.) Design a testbench for the register of Table 10.4. Adapt the 8-bit register design to a 4-bit version with the following interface declaration:

```
entity Reg4 is port (D : in STD_LOGIC_VECTOR(7 downto 0);
  Clk, Pre, Clr : in STD_LOGIC; Q, QB : out STD_LOGIC_VECTOR(7 downto 0));
end Reg8;
```

Create a testbench for your 4-bit register with the following component declaration:

```
component DFF
  port (Preset, Clear, Clock, Data: STD_LOGIC; Q, QBar: out STD_LOGIC_VECTOR);
end component;
```

10.35 (*Conversion functions, 30 min.) Write a conversion function from NATURAL to STD_LOGIC_VECTOR using the following declaration:

```
function Convert (N, L: NATURAL) return STD_LOGIC_VECTOR;
-- N is NATURAL, L is length of STD_LOGIC_VECTOR
```

Write a similar conversion function from STD_LOGIC_VECTOR to NATURAL:

```
function Convert (B: STD_LOGIC_VECTOR) return NATURAL;
```

Create a testbench to test your functions by including them in a package.

10.36 (Clock procedure, 20 min.) Design a clock procedure for a two-phase clock (C1, C2) with variable high times (HT1, HT2) and low times (LT1, LT2) and the following interface. Include your procedure in a package and write a model to test it.

```
procedure Clock (C1, C2 : out STD_LOGIC; HT1, HT2, LT1, LT2 : TIME);
```

10.37 (Random number, 20 min.) Design a testbench for the following procedure:

```
procedure uniform (seed : inout INTEGER range 0 to 15) is
  variable x : INTEGER;
  begin x := (seed*11) + 7; seed := x mod 16;
end uniform;
```

10.38 (Full-adder, 30 min.) Design and test a behavioral model of a full adder with the following interface:

```
entity FA is port (X, Y, Cin : STD_LOGIC; Cout, Sum : out STD_LOGIC);
end;
```

Repeat the exercise for inputs and outputs of type UNSIGNED.

10.39 (8-bit adder testbench, 60 min.) Write out the code corresponding to the generate statements of Adder_1 (Structure) in Section 10.13.7. Write a testbench to check your adder. What problems do you encounter? How thorough do you believe your tests are?

10.40 (Shift-register testbench, 60 min.) Design a testbench for the shift register of Table 10.4. Convert this model to use STD_LOGIC types with the following interface:

```
entity ShiftN is
port (CLK, CLR, LD, SH, DIR : STD_LOGIC;
      D : STD_LOGIC_VECTOR; Q : out STD_LOGIC_VECTOR);
end;
```

10.41 (Multiplier, 60 min.) Design and test a multiplier with the following interface:

```
entity Mult8 is
port (A, B : STD_LOGIC_VECTOR(3 downto 0);
      Start, CLK, Reset : in STD_LOGIC;
      Result : out STD_LOGIC_VECTOR(7 downto 0); Done : out BIT);
end;
```

- a. Create testbench code to check your model.
- b. Catalog each compile step with the syntax errors as you debug your code.
- c. Include a listing of the first code you write together with the final version.

An interesting class project is to collect statistics from other students working on this problem and create a table showing the types and frequency of syntax errors made with each compile step, and the number of compile steps required. Does this

information suggest ways that you could improve the compiler, or suggest a new type of tool to use when writing VHDL?

10.42 (Port maps, 5 min.) What is wrong with this VHDL statement?

```
U1 : nand2 port map (a <= set, b <= qb, c <= q);
```

10.43 (DRIVING_VALUE, 15 min.) Use the VHDL-93 attribute Clock'DRIVING_VALUE to rewrite the following clock generator model without using a temporary variable.

```
entity ClockGen_2 is port (Clock : out BIT); end;
architecture Behave of ClockGen_2 is
begin process variable Temp : BIT := '1'; begin
  Temp := not Temp ; Clock <= Temp after 10 ns; wait for 10 ns;
  if (now > 100 ns) then wait; end if; end process;
end;
```

10.44 (Records, 15 min.) Write an architecture (based on the following skeleton) that uses the record structure shown:

```
entity Test_Record_1 is end; architecture Behave of Test_Record_1 is
begin process type Coordinate is record X, Y : INTEGER; end record;
-- a record declaration for an attribute declaration:
attribute Location:Coordinate; -- an attribute declaration
begin wait; end process; end Behave;
```

10.45 (**Communication between processes, 30 min.) Explain and correct the problem with the following skeleton code:

```
variable v1 : INTEGER := 1; process begin v1 := v1+3; wait; end process;
process variable v2 : INTEGER := 2; begin v2 := v1 ; wait; end process;
```

10.46 (*Resolution, 30 min.) Explain and correct the problems with the following:

```
entity R_Bad_1 is port (i : in BIT; o out BIT); end;
architecture Behave of R_Bad_1 is
begin o <= not i after 1 ns; o <= i after 2 ns; end;
```

10.47 (*Inputs, 30 min.) Analyze the following and explain the result:

```
entity And2 is port (A1, A2: in BIT; ZN: out BIT); end;
architecture Simple of And2 is begin ZN <= A1 and A2; end;

entity Input_Bad_1 is end; architecture Netlist of Input_Bad_1 is
component And2 port (A1, A2 : in BIT; ZN : out BIT); end component;
signal X, Z : BIT begin G1 : And2 port map (X, X, Z); end;
```

10.48 (Association, 15 min.) Analyze the following and explain the problem:

```
entity And2 is port (A1, A2 : in BIT; ZN : out BIT); end;
architecture Simple of And2 is begin ZN <= A1 and A2; end;

entity Assoc_Bad_1 is port (signal X, Y : in BIT; Z : out BIT); end;
architecture Netlist of Assoc_Bad_1 is
component And2 port (A1, A2 : in BIT; ZN : out BIT); end component;
```

```

begin
G1: And2 port map (X, Y, Z);
G2: And2 port map (A2 => Y, ZN => Z, A1 => X);
G3: And2 port map (X, ZN => Z, A2 => Y);
end;

```

10.49 (Modes, 30 min.) Analyze and explain the errors in the following:

```

entity And2 is port (A1, A2 : in BIT; ZN : out BIT); end;
architecture Simple of And2 is begin ZN <= A1 and A2; end;

entity Mode_Bad_1 is port (X : in BIT; Y : out BIT; Z : inout BIT); end;
architecture Netlist of Mode_Bad_1 is
component And2 port (A1, A2 : in BIT; ZN : out BIT); end component;
begin G1 : And2 port map (X, Y, Z); end;

entity Mode_Bad_2 is port (X : in BIT; Y : out BIT; Z : inout BIT); end;
architecture Netlist of Mode_Bad_1 is
component And2 port (A1, A2 : in BIT; ZN : inout BIT); end component;
begin G1 : And2 port map (X, Y, Z); end;

```

10.50 (*Mode association, 60 min.) Analyze and explain the errors in the following code. The number of errors, types of error, and the information in the error messages given by different simulators vary tremendously in this area.

```

entity Allmode is port
(I : in BIT; O : out BIT; IO : inout BIT; B : buffer BIT);
end;
architecture Simple of Allmode is begin O<=I; IO<=I; B<=I; end;

entity Mode_1 is port
(I : in BIT; O : out BIT; IO : inout BIT; B : buffer BIT);
end;
architecture Netlist of Mode_1 is
component Allmode port
(I : in BIT; O : out BIT; IO : inout BIT; B : buffer BIT); end
component;
begin
G1:Allmode port map (I , O , IO, B );
G2:Allmode port map (O , IO, B , I );
G3:Allmode port map (IO, B , I , O );
G4:Allmode port map (B , I , O , IO);
end;

```

10.51 (**Declarations, 60 min.) Write a tutorial (approximately two pages of text, five pages with code) with examples explaining the difference between: a component declaration, a component configuration, a configuration declaration, a configuration specification, and a block configuration.

10.52 (**Guards and guarded signals, 60 min.) Write some simple models to illustrate the use of guards, guarded signals, and the disconnect statement. Include an experiment that shows and explains the use of the implicit signal GUARD in assignment statements.

10.53 (**Std_logic_1164, 120 min.) Write a short (two pages of text) tutorial, with (tested) code examples, explaining the `std_logic_1164` types, their default values, the difference between the 'u`logic`' and 'logic' types, and their vector forms. Include an example that shows and explains the problem of connecting a `std_logic_vector` to a `std_ulogic_vector`.

10.54 (Data swap, 20 min.) Consider the following code:

```
library ieee; use ieee.std_logic_1164.all;
package config is
type type1 is record
f1 : std_logic_vector(31 downto 0); f2 : std_logic_vector(3 downto 0);
end record;
type type2 is record
f1 : std_logic_vector(31 downto 0); f2 : std_logic_vector(3 downto 0);
end record;
end config;
library ieee; use ieee.STD_LOGIC_1164.all; use work.config.all;
entity Swap_1 is
port (Data1 : type1; Data2 : type2; sel : STD_LOGIC;
Data1Swap : out type1; Data2Swap : out type2); end Swap_1;

architecture Behave of Swap_1 is begin
Swap: process (Data1, Data2, sel) begin case sel is
when '0' => Data1Swap <= Data1; Data2Swap <= Data2;
when others => Data1Swap <= Data2; Data2Swap <= Data1;
end case; end process Swap; end Behave;
```

Compile this code. What is the problem? Suggest a fix. Now write a testbench and test your code. Have you considered all possibilities?

10.55 (**RTL, 30 min.) “RTL stands for **register-transfer level**. ...when referencing VHDL, the term means that the description includes only concurrent signal assignment statements and possibly block statements. In particular, VHDL data flow descriptions explicitly do not contain either process statements (which describe behavior) or component instantiation statements (which describe structure)” (Dr. VHDL from VHDL International).

- a. With your knowledge of process statements and components, comment on Dr. VHDL’s explanation.
- b. In less than 100 words offer your own definition of the difference between RTL, data flow, behavioral, and structural models.

10.56 (*Operators mod and rem, 20 min.) Confirm and explain the following:

```
i1 := (-12) rem 7;           -- i1 = -5
i2 := 12    rem (-7);        -- i2 = 5
i3 := (12)  rem (-7);        -- i3 = -5
i4 := 12    mod 7;           -- i4 = 5
i5 := (-12) mod 7;           -- i5 = 2
i6 := 12    mod (-7);        -- i6 = -2
i7 := (12)  mod (-7);        -- i7 = -5
```

Evaluate $-5 \text{ rem } 2$ and explain the result.

10.57 (**Event and stable, 60 min.) Investigate the differences between `clk'EVENT` and `clk'STABLE`. Write a minitutorial (in the form of a “cheat sheet”) with examples showing the differences and potential dangers of using `clk'STABLE`.

10.58 (PREP benchmark #2, 60 min.) The following code models a benchmark circuit used by **PREP** to measure the capacity of FPGAs. Rewrite the concurrent signal assignment statements (labeled `mux` and `comparator`) as equivalent processes. Draw a datapath schematic corresponding to `PREP2(Behave_1)`. Write a testbench for the model. Finally (for extra credit) rewrite the model and testbench to use `STD_LOGIC` instead of `BIT` types.

```
library ieee; use ieee.STD_LOGIC_1164.all;
use ieee.NUMERIC_BIT.all; use ieee.NUMERIC_STD.all;
entity PREP2 is
port(CLK,Reset,Sel,Ldli,Ldhi : BIT; D1,D2 : STD_LOGIC_VECTOR(7 downto 0);
     DQ:out STD_LOGIC_VECTOR(7 downto 0));
end;

architecture Behave_1 of PREP2 is
signal EQ : BIT; signal y,lo,hi,Q_i : STD_LOGIC_VECTOR(7 downto 0);
begin
outputDriver: Q <= Q_i;
mux: with Sel select y <= hi when '0', D1 when '1';
comparator: EQ <= '1' when Q_i = lo else '0';
     register: process(Reset, CLK) begin
         if Reset = '1' then hi <= "00000000"; lo <= "00000000";
         elsif CLK = '1' and CLK'EVENT then
             if Ldhi='1' then hi<=D2;end if;if Ldlo='1' then lo<=D2;end if;
         end if;
     end process register;
     counter: process(Reset, CLK) begin
         if Reset = '1' then Q_i <= "00000000";
         elsif CLK = '1' and CLK'EVENT then
             if EQ = '1' then Q_i <= y;
             elsif EQ = '0' then Q_i <= Q_i + "00000001";
             end if;
         end if;
     end process counter;
end;
```

10.59 (PREP #3, state machine) Draw the state diagram for the following PREP benchmark (see Problem 10.58). Is this a Mealy or Moore machine? Write a testbench and test this code.

```
library ieee; use ieee.STD_LOGIC_1164.all;
entity prep3_1 is port(Clk, Reset: STD_LOGIC;
     I : STD_LOGIC_VECTOR(7 downto 0); O : out STD_LOGIC_VECTOR(7 downto 0));
end prep3_1;
architecture Behave of prep3_1 is
```



```

type STATE_TYPE is (sX,s0,sa,sb,sc,sd,se,sf,sg);
signal state : STATE_TYPE; signal Oi : STD_LOGIC_VECTOR(7 downto 0);
begin
  O <= Oi;
  process (Reset, Clk) begin
    if (Reset = '1') then state <= s0; Oi <= (others => '0');
    elsif rising_edge(Clk) then
      case state is
        when s0 =>
          if (I = X"3c") then state <= sa; Oi <= X"82";
          else state <= s0; Oi <= (others => '0');
          end if;
        when sa =>
          if (I = X"2A") then state <= sc; Oi <= X"40";
          elsif (I = X"1F") then state <= sb; Oi <= X"20";
          else state <= sa; Oi <= X"04";
          end if;
        when sb =>
          if (I = X"AA") then state <= se; Oi <= X"11";
          else state <= sf; Oi <= X"30";
          end if;
        when sc => state <= sd; Oi <= X"08";
        when sd => state <= sg; Oi <= X"80";
        when se => state <= s0; Oi <= X"40";
        when sf => state <= sg; Oi <= X"02";
        when sg => state <= s0; Oi <= X"01";
        when others => state <= sX; Oi <= (others => 'X');
      end case;
    end if;
  end process;
end;

```

10.60 (Edge detection, 30 min) Explain the construction of the IEEE 1164 function to detect the rising edge of a signal, `rising_edge(s)`. List all the changes in signal `s` that correspond to a rising edge.

```

function rising_edge (signal s : STD_ULOGIC) return BOOLEAN is
begin return
  (s'EVENT and (To_X01(s) = '1') and (To_X01(s'LAST_VALUE) = '0'));
end;

```

10.61 (*Real, 10 min.) Determine the smallest real in your VHDL environment.

10.62 (*Stop, 30 min.) How many ways are there to stop a VHDL simulator?

10.63 (*Arithmetic package, 60 min.) Write a function for an arithmetic package to subtract two's complement numbers. Create a test bench to check your function. Your declarations in the package header should look like this:

```

type TC is array (INTEGER range <>) of STD_LOGIC;
function "-"(L : TC; R : TC) return TC;

```

10.64 (**Reading documentation, hours) There are a few gray areas in the interpretation of the VHDL-87 LRM some of which were clarified in the VHDL-93 revision. One VHDL system has a “**compatibility mode**” that allows alternative interpretations. For each of the following “issues” taken from the actual tool documentation try to interpret what was meant, determine the interpretation taken by your own software, and then rewrite the explanation clearly using examples.

- a. * “Unassociated variable and signal parameters. Compatibility mode allows variable and signal parameters to subprograms to be unassociated if they have a default value. Otherwise, an error is generated.”

Example answer: Consider the following code:

```
package Util_2 is
procedure C(signal Clk : out BIT; signal P : TIME := 10 ns);
end Util_2;
package body Util_2 is
procedure C(signal Clk : out BIT; signal P : TIME := 10 ns) is
begin loop Clk <= '1' after P/2, '0' after P;
wait for P; end loop; end; end Util_2;
entity Test_Compatibility_1 is end; use work.Util_2.all;
architecture Behave of Test_Compatibility_1 is
signal v,w,x,y,z : BIT; signal s : TIME := 5 ns;
begin process variable v : TIME := 5 ns; begin
C(v, s);           -- parameter s is OK since P is declared as signal
-- C(w, v);        -- would be OK if P is declared as variable instead
-- C(x, 5 ns);     -- would be OK if P is declared as constant instead
-- C(y);           -- unassociated, an error if P is signal or variable
-- C(z,open);     -- open, an error if P is signal or variable
end process; end;
```

The Compass Scout simulator (which does not have a compatibility mode) generates an error during analysis if a signal or variable subprogram parameter is open or unassociated (a constant subprogram parameter may be unassociated or open).

- b. * “Allow others in an aggregate within a record aggregate. The LRM [7.3.2.2] defines nine situations where others may appear in an aggregate. In compatibility mode, a tenth case is added. In this case, others is allowed in an aggregate that appears as an element association in a record element.”
- c. * “BIT' ('1') parsed as BIT ' ('1') . The tick (') character is being used twice in this example. In the first case as an attribute indicator, in the second case, to form a character literal. Without the compatibility option, the analyzer adopts a strict interpretation of the LRM, and without white space around the first tick, the fragment is parsed as BIT ' ('1') , that is, the left parenthesis (' (') is the character literal.”
- d. ** “Generate statement declarative region. Generate statements form their own declarative region. In compatibility mode, configuration specifications will apply to items being instantiated within a generate statement.”

- e. ** “Allow type conversion functions on open parameters. If a parameter is specified as open, it indicates a parameter without an explicit association. In such cases, the presence of a type conversion function is meaningless. Compatibility mode allows the type conversion functions.”
- f. *** “Entity class flexibility. Section [3.1.2] of the LRM defines the process of creating a new integer type. The type name given is actually assigned to a subtype name, related to an anonymous base type. This implies that the entity class used during an attribute specification [LRM 5.1] should indicate subtype, rather than type. Because the supplied declaration was type rather than subtype, compatibility mode allows type.”
- g. *** “Allowing declarations beyond an all/others specification. Section [5.1] of the LRM states that the first occurrence of the reserved word `all` or `others` in an attribute specification terminates the declaration of the related entity class. The LRM declares that the entity/architecture and package/package body library units form single declaration regions [LRM 10.1] that are the concatenation of the two individual library declarative regions. For example, if a signal attribute specification with `all` or `others` was specified in the entity, it would be impossible to declare a signal in the architecture. In compatibility mode, this LRM limitation is removed.”
- h. *** “User-defined attributes on overloaded functions. In compatibility mode, user-defined attributes are allowed to be associated with overloaded functions. Note: Even in compatibility mode, there is no way to retrieve the different attributes.”

10.65 (*1076 interpretations, 30 min.) In a DAC paper, the author writes: ‘It was experienced that (company R) might have interpreted IEEE 1076 differently than (company S) did, e.g. concatenations (&) are not allowed in “case selector” expressions for (company S).’ Can you use concatenation in your VHDL tool for either the expression or choices for a case statement?

10.66 (**Interface declarations, 15 min.) Analyze the following and comment:

```
entity Interface_1 is
  generic (I : INTEGER; J : INTEGER := I; K, L : INTEGER);
  port (A : BIT_VECTOR; B : BIT_VECTOR(A'RANGE); C : BIT_VECTOR (K to L));
  procedure X(P, Q : INTEGER; R : INTEGER range P to Q);
  procedure Y(S : INTEGER range K to L);
end Interface_1;
```

10.67 (**Wait statement, 10 min.) Construct the sensitivity set and thus the sensitivity list for the following wait statement (that is, rewrite the wait statement in the form `wait on sensitivity_list until condition`).

```
entity Complex_Wait is end; --1
architecture Behave of Complex_Wait is --2
  type A is array (1 to 5) of BOOLEAN; --3
```

```

function F (P : BOOLEAN) return BOOLEAN;           --4
signal S : A; signal i, j : INTEGER range 1 to 5;  --5
begin process begin                                --6
    wait until F(S(3)) and (S(i) or S(j));         --7
end process;                                       --8
end;                                               --9

```

10.68 (**Shared variables, 20 min.) Investigate the following code and comment:

```

architecture Behave of Shared_1 is
subtype S is INTEGER range 0 to 1; shared variable C : S := 0; begin
process begin C := C + 1; wait; end process;
process begin C := C - 1; wait; end process;
end;

```

10.69 (Undocumented code and ranges, 20 min.) Explain the purpose of the following function (part of a package from a well-known synthesis company) with a parameter of type SIGNED. Write a testbench to check your explanation. Investigate what happens when you call this function with a string-literal argument, for example with the statement `X <= IM("11100")`. What is the problem and why does it happen? Rewrite the code, including documentation, to avoid this problem.

```

type SIGNED is array (NATURAL range <> ) of BIT;

function IM (L : SIGNED) return INTEGER is variable M : INTEGER;
begin M := L'RIGHT-1;
    for i in L'LEFT-1 downto L'RIGHT loop
        if (L(i) = (not L(L'LEFT))) then M := i; exit; end if;
    end loop; return M;
end;

```

10.70 (Timing parameters, 20 min.) Write a model and a testbench for a two-input AND gate with separate rising (t_{PLH}) and falling (t_{PHL}) delays using the following interface:

```

entity And_Process is
generic (tpLH, tpHL : TIME); port (a, b : BIT; z : out BIT) end;

```

10.71 (Passive code in entities, 30 min.) Write a procedure (CheckTiming, part of a package Timing_Pkg) to check that two timing parameters (t_{PLH} and t_{PHL}) are both greater than zero. Include this procedure in a two-input AND gate model (And_Process). Write a testbench to show your procedure and gate model both work. Rewrite the entity for And_Process to include the timing check as part of the entity declaration. You are allowed to include **passive code** (no assignments to signals and so on) directly in each entity. This avoids having to include the timing checks in each architecture.

10.72 (Buried code, 30 min.) Some companies bury instructions to the software within their packages. Here is an example of part of the arithmetic package from an imaginary company called SissyN:

```

function UN_plus(A, B : UN) return UN is          --1
variable CRY : STD_ULOGIC; variable X,SUM : UN (A'LEFT downto 0); --2
-- pragma map_to_operator ADD_UN_OP              --3
-- pragma type_function LEFT_UN_ARG             --4
-- pragma return_port_name Z                    --5
begin                                           --6
-- sissyn synthesis_off                         --7
if (A(A'LEFT) = 'X' or B(B'LEFT) = 'X') then SUM := (others => 'X'); --8
return(SUM);                                   --9
end if;                                        --10
-- sissyn synthesis_on                          --11
CRY := '0'; X := B;                             --12
for i in 0 to A'LEFT loop                       --13
SUM(i) := A(i) xor X(i) xor carry;              --14
CRY := (A(i) and X(i)) or (A(i) and CRY) or (CRY and X(i)); --15
end loop; return SUM;                           --16
end;                                           --17

```

Explain what this function does. Can you now hazard a guess at what each of the comments means? What are the repercussions of using comments in this fashion?

10.73 (*Deferred constants, 15 min.) “If the assignment symbol ‘:=’ followed by an expression is not present in a constant declaration, then the declaration declares a **deferred constant**. Such a constant declaration may only appear in a package declaration. The corresponding full constant declaration, which defines the value of the constant, must appear in the body of the package” [VHDL 93LRM4.3.1.1].

```

package Constant is constant s1, s2 : BIT_VECTOR; end Constant;

package body Constant is
constant s0 : BIT_VECTOR := "00"; constant s1 : BIT_VECTOR := "01";
end Constant;

```

It is tempting to use deferred constants to hide information. However, there are problems with this approach. Analyze the following code, explain the results, and correct the problems:

```

entity Deferred_1 is end; architecture Behave of Deferred_1 is
use work.all; signal y,i1,i2 : INTEGER; signal sel : INTEGER range 0 to 1;
begin with sel select y <= i1 when s0, i2 when s1; end;

```

10.74 (**Viterbi code, days) Convert the Verilog model of the Viterbi decoder in Chapter 11 to VHDL. This problem is tedious without the help of some sort of **Verilog to VHDL conversion** process. There are two main approaches to this problem. The first uses a synthesis tool to read the behavioral Verilog and write structural VHDL (the Compass ASIC Synthesizer can do this, for example). The second approach uses conversion programs (Alternative System Concepts Inc. at

<http://www.ascinc.com> is one source). Some of these companies allow you to e-mail code to them and they will automatically return a translated version.

10.75 (*Wait statement, 30 min.) Rewrite the code below using a single wait statement and write a testbench to prove that both approaches are exactly equivalent:

```
entity Wait_Exit is port (Clk : in BIT); end;
architecture Behave of Wait_Exit is
begin process begin
    loop wait on Clk; exit when Clk = '1'; end loop;
end process;
end;
```

10.76 (Expressions, 10 min.) Explain and correct the problems with the following:

```
variable b : BOOLEAN; b := "00" < "11"; --1
variable bv8 : BIT_VECTOR (7 downto 0) := "1000_0000"; --2
```

10.77 (Combinational logic using case statement, 10 min.) A Verilog user suggests the following method to model combinational logic. What are the problems with this approach? Can you get it to work?

```
entity AndCase is port (a, b : BIT; y : out BIT); end;
architecture Behave of AndCase is begin process (a , b) begin
    case a & b is
        when '1'&'1' => y <= '1'; when others => y <= '0';
    end case;
end process; end;
```

10.78 (*Generics and back-annotation, 60 min.)

- Construct design entities `And_3(Behave)`, a two-input AND gate, and `Xor_3(Behave)`, a two-input XOR gate. Include generic constants to model the propagation delay from each input to the output separately. Use the following entity declaration for `And_3`:

```
entity And_3 is port (I1, I2 : BIT; O : out BIT);
generic (I1toO, I2toO : DELAY_LENGTH := 0.4 ns); end;
```

- Create and test a package, `P_1`, that contains `And_3` and `Xor_3` as components.
- Create and test a design entity `Half_Adder_3(Structure_3)` that uses `P_1`, with the following interface:

```
entity Half_Adder_3 is port (X, Y : BIT; Sum, Carry : out BIT); end;
```

- Modify and test the architecture `Structure_3` for `Half_Adder_3` so that you can use the following configuration:

```
configuration Structure_3 of Half_Adder_3 is
for Structure_3
for L1 : XOR generic map (0.66 ns,0.69 ns); end for;
for L2 : AND generic map (0.5 ns, 0.6 ns) port map (I2 => HI); end for;
end for; end;
```

10.79 (SNUG'95, *60 min.) In 1995 John Cooley organized a contest between VHDL and Verilog for ASIC designers. The goal was to design the fastest 9-bit counter in under one hour using Synopsys synthesis tools and an LSI Logic vendor technology library. The VHDL interface is as follows:

```
library ieee; use ieee.std_logic_1164.all;
-- use ieee.std_logic_arith.all; -- substitute your package here
entity counter is port (
data_in      : in std_logic_vector(8 downto 0);
up           : in std_logic;
down        : in std_logic;
clock       : in std_logic;
count_out   : inout std_logic_vector(8 downto 0);
carry_out   : out std_logic;
borrow_out  : out std_logic;
parity_out  : out std_logic ); end counter;
architecture example of counter is begin
-- insert your design here
end example;
```

The counter is positive-edge triggered, counts up with `up = '1'` and down with `down = '1'`. The contestants had the advantage of a predefined testbench with a set of test vectors, you do not. Design a model for the counter and a testbench. How confident are you that you have thoroughly tested your model? (In the real contest none of the VHDL contestants managed to even complete a working design in under one hour. In addition, the VHDL experts that had designed the testbench omitted a test case for one of the design specifications.)

10.80 (*A test procedure, 45 min.) Write a procedure `all` (for a package test) that serially generates all possible input values for a signal spaced in time by a delay, `dly`. Use the following interface:

```
library ieee; use ieee.std_logic_1164.all; package test is
procedure all (signal SLV : out STD_LOGIC_VECTOR; dly : in TIME);
end package test ;
```

10.81 (Direct instantiation, 20 min.) Write an architecture for a full-adder, entity `Full_Adder_2`, that directly instantiates units `And_2(Behave)` and `Xor_2(Behave)`. This is only possible in a VHDL-93 environment.

```
entity And_2 is port (i1, i2 : BIT; y : out BIT); end;
entity Xor_2 is port (i1, i2 : BIT; y : out BIT); end;
entity Full_Adder_2 is port (a, b, c : BIT ; sum, cout : out BIT); end;
```

10.82 (**Shift operators for 1164, 60 min.) Write a package body to implement the VHDL-93 shift operators, `sll` and `srl`, for the type `STD_LOGIC_VECTOR`. Use the following package header:

```
package 1164_shift is
function "sll"(x : STD_LOGIC_VECTOR; n : INTEGER)
return STD_LOGIC_VECTOR;
```

```
function "srl"(x : STD_LOGIC_VECTOR; n : INTEGER)
  return STD_LOGIC_VECTOR;
end package 1164_shift;
```

10.83 (**VHDL wait statement, 60 min.) What is the problem with the following VHDL code? *Hint:* You may need to consult the VHDL LRM.

```
procedure p is begin wait on b; end;
process (a) is begin procedure p; end process;
```

10.84 (**Null range, 45 min.) A range such as 1 **to** -1 or 0 **downto** 1 is a **null range** (0 **to** 0 is a legal range). Write a one-page summary on null ranges, including code examples. Is a null range treated as an ascending or descending range?

10.85 (**Loops, 45 min.) Investigate the following issues with loops, including code examples and the results of analysis and simulation:

- Try to alter the loop parameter within a loop. What happens?
- What is the type of the loop parameter?
- Can the condition inside a loop depend on a loop parameter?
- What happens in a **for** loop if the range is null?
- Can you pass a loop parameter out of a procedure as a procedure parameter?

10.86 (Signals and variables, 30 min.) Write a summary on signals and variables, including code examples.

10.87 (Type conversion, 60 min.) There are some very subtle rules involving type conversion, [VHDL 93LRM7.3.5]. Does the following work? Explain the type conversion rules.

```
BV <= BIT_VECTOR("1111");
```

10.19 Bibliography

The definitive reference guide to VHDL is the IEEE VHDL LRM [IEEE, 1076-1993]. The LRM is initially difficult to read because it is concise and precise (the LRM is intended for tool builders and experienced tool users, not as a tutorial). The LRM does form a useful reference—as does a dictionary for serious users of any language. You might think of the LRM as a legal contract between you and the company that sells you software that is compliant with the standard. VHDL software uses the terminology of the LRM for error messages, so it is necessary to understand the terms and definitions of the LRM. The WAVES standard [IEEE 1029.1-1991] deals with the problems of interfacing VHDL testbenches to testers.

VHDL International maintains VIUF (VHDL International Users' Forum) Internet Services (<http://www.vhdl.org>) and links to other groups working on VHDL including the IEEE synthesis packages, IEEE WAVES packages, and IEEE VITAL packages (see also Appendix A).

The frequently asked questions (FAQ) list for the VHDL newsgroup `comp.lang.vhdl` is a useful starting point (the list is archived at `gopher://kona.ee.pitt.edu/h0/NewsGroupArchives`). Information on character sets and the problems of exchanging information across national boundaries can be found at `ftp://watsun.cc.columbia.edu/kermit/charsets`.

10.20 References

Page numbers in brackets after the reference indicate the location in the chapter body.

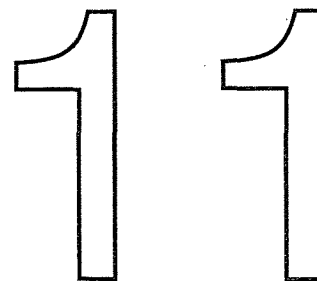
IEEE 1029.1-1991. *IEEE Standard for Waveform and Vector Exchange (WAVES)*. IEEE Std 1029.1-1991. The Institute of Electrical and Electronics Engineers, Inc., New York. Available from The Institute of Electrical and Electronics Engineers, Inc., 345 East 47th Street, New York, NY 10017 USA.

IEEE 1076-1993. *IEEE Standard VHDL Language Reference Manual (ANSI)*. IEEE Std. 1076-1993. The Institute of Electrical and Electronics Engineers, Inc., New York. Available from The Institute of Electrical and Electronics Engineers, Inc., 345 East 47th Street, New York, NY 10017 USA. [p. 380]

IEEE 1076.2-1996. *Standard VHDL Language Mathematical Packages*. IEEE Ref. AD129-NYF. Approved by IEEE Standards Board on 19 September 1996. [p. 404].

ISO 8859-1. 1987 (E). Information Processing—8-bit single-byte coded graphic character sets—Part 1: Latin Alphabet No. 1. American National Standards Institute, Hackensack, NJ; 1987. Available from Sales Department, American National Standards Institute, 105-111 South State Street, Hackensack, NJ 07601 USA. [p. 391]

VERILOG HDL



- 11.1 A Counter
- 11.2 Basics of the Verilog Language
- 11.3 Operators
- 11.4 Hierarchy
- 11.5 Procedures and Assignments
- 11.6 Timing Controls and Delay
- 11.7 Tasks and Functions
- 11.8 Control Statements
- 11.9 Logic-Gate Modeling
- 11.10 Modeling Delay
- 11.11 Altering Parameters
- 11.12 A Viterbi Decoder
- 11.13 Other Verilog Features
- 11.14 Summary
- 11.15 Problems
- 11.16 Bibliography
- 11.17 References

In this chapter we look at the **Verilog** hardware description language. Gateway Design Automation developed Verilog as a simulation language. The use of the Verilog-XL simulator is discussed in more detail in Chapter 13. Cadence purchased Gateway in 1989 and, after some study, placed the Verilog language in the public domain. Open Verilog International (OVI) was created to develop the Verilog language as an IEEE standard. The definitive reference guide to the Verilog language is now the Verilog LRM, IEEE Std 1364-1995 [1995].¹ This does not mean that all Verilog simulators and tools adhere strictly to the IEEE Standard—we must abide by the reference manual for the software we are using. Verilog is a fairly simple language to learn, especially if you are familiar with the C programming language. In this chapter we shall concentrate on the features of Verilog applied to high-level design entry and synthesis for ASICs.

¹Some of the material in this chapter is reprinted with permission from IEEE Std 1364-1995, © Copyright 1995 IEEE. All rights reserved.

11.1 A Counter

The following Verilog code models a “black box” that contains a 50MHz clock (period 20 ns), counts from 0 to 7, resets, and then begins counting at 0 again:

```

`timescale 1ns/1ns // Set the units of time to be nanoseconds. //1
module counter; //2
    reg clock; // Declare a reg data type for the clock. //3
    integer count; // Declare an integer data type for the count. //4
initial // Initialize things; this executes once at t=0. //5
    begin //6
        clock = 0; count = 0; // Initialize signals. //7
        #340 $finish; // Finish after 340 time ticks. //8
    end //9
/* An always statement to generate the clock; only one statement
follows the always so we don't need a begin and an end. */ //10
always //11
    #10 clock = ~ clock; // Delay (10ns) is set to half the clock cycle. //12
/* An always statement to do the counting; this executes at the same
time (concurrently) as the preceding always statement. */ //13
always //14
    begin //15
        // Wait here until the clock goes from 1 to 0. //16
        @ (negedge clock); //17
        // Now handle the counting. //18
        if (count == 7) //19
            count = 0; //20
        else //21
            count = count + 1; //22
        $display("time = ", $time, " count = ", count); //23
    end //24
endmodule //25

```

Verilog **keywords** (reserved words that are part of the Verilog language) are shown in bold type in the code listings (but not in the text). References in this chapter such as [Verilog LRM 1.1] refer you to the IEEE Verilog LRM.

The following output is from the Cadence Verilog-XL simulator. This example includes the system input so you can see how the tool is run and when it is finished. Some of the banner information is omitted in the listing that follows to save space (we can use “quiet” mode using a ‘-q’ flag, but then the version and other useful information is also suppressed):

```

> verilog counter.v
VERILOG-XL 2.2.1 Apr 17, 1996 11:48:18
... Banner information omitted here...
Compiling source file "counter.v"
Highest level modules:

```

counter

```

time =                20 count =                1
time =                40 count =                2
(... 12 lines omitted...)
time =                300 count =                7
time =                320 count =                0
L10 "counter.v": $finish at simulation time 340
223 simulation events
CPU time: 0.6 secs to compile + 0.2 secs to link + 0.0 secs in
simulation
End of VERILOG-XL 2.2.1   Apr 17, 1996  11:48:20
>

```

Here is the output of the VeriWell simulator from the console window (future examples do not show all of the compiler output— just the model output):

```

Veriwell -k VeriWell.key -l VeriWell.log -s :counter.v
... banner information omitted ....
Memory Available: 0
Entering Phase I...
Compiling source file : :counter.v
The size of this model is [1%, 1%] of the capacity of the free version

```

```

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

```

```

C1> .
time =                20 count =                1
time =                40 count =                2
(... 12 lines omitted...)
time =                300 count =                7
time =                320 count =                0
Exiting VeriWell for Macintosh at time 340
0 Errors, 0 Warnings, Memory Used: 29468
Compile time = 0.6, Load time = 0.7, Simulation time = 4.7

```

```

Normal exit
Thank you for using VeriWell for Macintosh

```

11.2 Basics of the Verilog Language

A Verilog **identifier** [Verilog LRM2.7], including the names of variables, may contain any sequence of letters, digits, a dollar sign '\$', and the underscore '_' symbol. The first character of an identifier must be a letter or underscore; it cannot be a dollar sign '\$', for example. We cannot use characters such as '-' (hyphen), brackets, or '#' (for active-low signals) in Verilog names (escaped identifiers are an exception). The following is a shorthand way of saying the same thing:

```

identifier ::= simple_identifier | escaped_identifier
simple_identifier ::= [ a-zA-Z_ ][ a-zA-Z_$ ]
escaped_identifier ::=
    \ {Any_ASCII_character_except_white_space} white_space
white_space ::= space | tab | newline

```

(In the 1995 LRM the underscore '_' is missing from the first bracket.) If we think of ' ::= ' as an equal sign, then the preceding "equation" defines the syntax of an identifier. Usually we use the Backus–Naur form (BNF) to write these equations. We also use the BNF to describe the syntax of VHDL. There is an explanation of the BNF in Appendix A. Verilog syntax definitions are given in Appendix B. In Verilog all names, including keywords and identifiers, are case-sensitive. Special commands for the simulator (a system task or a system function) begin with a dollar sign '\$' [Verilog LRM 2.7]. Here are some examples of Verilog identifiers:

```

module identifiers;                                     //1
/* Multiline comments in Verilog                       //2
   look like C comments and // is OK in here. */      //3
// Single-line comment in Verilog.                    //4
reg legal_identifier,two__underscores;                //5
reg _OK,OK_,OK_$,OK_123,CASE_SENSITIVE, case_sensitive; //6
reg \clock ,\a*b ; // Add white_space after escaped identifier. //7
//reg $_BAD,123_BAD; // Bad names even if we declare them! //8
initial begin                                         //9
legal_identifier = 0; // Embedded underscores are OK, //10
two__underscores = 0; // even two underscores in a row. //11
_OK = 0; // Identifiers can start with underscore //12
OK_ = 0; // and end with underscore. //13
OK$ = 0; // $ sign is OK, but beware foreign keyboards. //14
OK_123 =0; // Embedded digits are OK. //15
CASE_SENSITIVE = 0; // Verilog is case-sensitive (unlike VHDL). //16
case_sensitive = 1; //17
\clock = 0; // An escaped identifier with \ breaks rules, //18
\a*b = 0; // but be careful to watch the spaces! //19
$display("Variable CASE_SENSITIVE= %d",CASE_SENSITIVE); //20
$display("Variable case_sensitive= %d",case_sensitive); //21
$display("Variable \clock = %d",\clock ); //22
$display("Variable \a*b = %d",\a*b ); //23

```

```
end //24
endmodule //25
```

The following is the output from this model (future examples in this chapter list the simulator output directly after the Verilog code).

```
Variable CASE_SENSITIVE= 0
Variable case_sensitive= 1
Variable /clock = 0
Variable \a*b = 0
```

11.2.1 Verilog Logic Values

Verilog has a predefined logic-value system or **value set** [Verilog LRM 3.1] that uses four logic values: '0', '1', 'x', and 'z' (lowercase 'x' and lowercase 'z'). The value 'x' represents an uninitialized or an unknown logic value—an unknown value is either '1', '0', 'z', or a value that is in a state of change. The logic value 'z' represents a high-impedance value, which is usually treated as an 'x' value. Verilog uses a more complicated internal logic-value system in order to resolve conflicts between different drivers on the same node. This hidden logic-value system is useful for switch-level simulation, but for most ASIC simulation and synthesis purposes we do not need to worry about the internal logic-value system.

11.2.2 Verilog Data Types

There are several **data types** in Verilog—all except one need to be declared before we can use them. The two main data types are **nets** and **registers** [Verilog LRM 3.2]. Nets are further divided into several net types. The most common and important net types are: **wire** and **tri** (which are identical); **supply1** and **supply0** (which are equivalent to the positive and negative power supplies respectively). The **wire** data type (which we shall refer to as just **wire** from now on) is analogous to a wire in an ASIC. A wire cannot store or hold a value. A wire must be continuously driven by an assignment statement (see Section 11.5). The default initial value for a wire is 'z' [Verilog LRM3.6]. There are also **integer**, **time**, **event**, and **real** data types.

```
module declarations_1; //1
wire pwr_good, pwr_on, pwr_stable; // Explicitly declare wires. //2
integer i; // 32-bit, signed (2's complement). //3
time t; // 64-bit, unsigned, behaves like a 64-bit reg. //4
event e; // Declare an event data type. //5
real r; // Real data type of implementation defined size. //6
// An assign statement continuously drives a wire: //7
assign pwr_stable = 1'b1; assign pwr_on = 1; // 1 or 1'b1 //8
assign pwr_good = pwr_on & pwr_stable; //9
initial begin //10
i = 123.456; // There must be a digit on either side //11
r = 123456e-3; // of the decimal point if it is present. //12
```

```

t = 123456e-3; // Time is rounded to 1 second by default. //13
$display("i=%0g",i," t=%6.2f",t," r=%f",r); //14
#2 $display("TIME=%0d",$time," ON=",pwr_on, //15
" STABLE=",pwr_stable," GOOD=",pwr_good); //16
$finish; end //17
endmodule //18

i=123 t=123.00 r=123.456000
TIME=2 ON=1 STABLE=1 GOOD=1

```

A **register** data type is declared using the keyword `reg` and is comparable to a variable in a programming language. On the LHS of an assignment a register data type (which we shall refer to as just `reg` from now on) is updated immediately and holds its value until changed again. The default initial value for a `reg` is 'x'. We can transfer information directly from a wire to a `reg` as shown in the following code:

```

module declarations_2; //1
reg Q, Clk; wire D; //2
// Drive the wire (D): //3
assign D = 1; //4
// At a +ve clock edge assign the value of wire D to the reg Q: //5
always @(posedge Clk) Q = D; //6
initial Clk = 0; always #10 Clk = ~ Clk; //7
initial begin #50; $finish; end //8
always begin //9
$display("T=%2g", $time," D=",D," Clk=",Clk," Q=",Q); #10; end //10
endmodule //11

T= 0 D=z Clk=0 Q=x
T=10 D=1 Clk=1 Q=x
T=20 D=1 Clk=0 Q=1
T=30 D=1 Clk=1 Q=1
T=40 D=1 Clk=0 Q=1

```

We shall discuss assignment statements in Section 11.5. For now, it is important to recognize that a `reg` is not always equivalent to a hardware register, flip-flop, or latch. For example, the following code describes purely combinational logic:

```

module declarations_3; //1
reg a,b,c,d,e; //2
initial begin //3
#10; a = 0;b = 0;c = 0;d = 0; #10; a = 0;b = 1;c = 1;d = 0; //4
#10; a = 0;b = 0;c = 1;d = 1; #10; $stop; //5
end //6
always begin //7
@(a or b or c or d) e = (a|b)&(c|d); //8
$display("T=%0g", $time," e=",e); //9
end //10
endmodule //11

T=10 e=0

```

```
T=20 e=1
T=30 e=0
```

A single-bit wire or reg is a **scalar** (the default). We may also declare a wire or reg as a **vector** with a **range** of bits [Verilog LRM 3.3]. In some situations we may use implicit declaration for a scalar wire; it is the only data type we do not always need to declare. We must use explicit declaration for a vector wire or any reg. We may access (or **expand**) the range of bits in a vector one at a time, using a **bit-select**, or as a contiguous subgroup of bits (a continuous sequence of numbers—like a straight in poker) using a **part-select** [Verilog LRM 4.2]. The following code shows some examples:

```
module declarations_4; //1
wire Data; // A scalar net of type wire. //2
wire [31:0] ABus, DBus; // Two 32-bit-wide vector wires: //3
// DBus[31] = leftmost = most-significant bit = msb //4
// DBus[0] = rightmost = least-significant bit = lsb //5
// Notice the size declaration precedes the names. //6
// wire [31:0] TheBus, [15:0] BigBus; // This is illegal. //7
reg [3:0] vector; // A 4-bit vector register. //8
reg [4:7] nibble; // msb index < lsb index is OK. //9
integer i; //10
initial begin //11
i = 1; //12
vector = 'b1010; // Vector without an index. //13
nibble = vector; // This is OK too. //14
#1; $display("T=%0g", $time, " vector=", vector, " nibble=", nibble); //15
#2; $display("T=%0g", $time, " Bus=%b", DBus[15:0]); //16
end //17
assign DBus [1] = 1; // This is a bit-select. //18
assign DBus [3:0] = 'b1111; // This is a part-select. //19
// assign DBus [0:3] = 'b1111; // Illegal : wrong direction. //20
endmodule //21

T=1 vector=10 nibble=10
T=3 Bus=zzzzzzzzzzzz1111
```

There are no multidimensional arrays in Verilog, but we may declare a **memory** data type as an **array** of registers [Verilog LRM 3.8]:

```
module declarations_5; //1
reg [31:0] VideoRam [7:0]; // An 8-word by 32-bit wide memory. //2
initial begin //3
VideoRam[1] = 'bxz; // We must specify an index for a memory. //4
VideoRam[2] = 1; //5
VideoRam[7] = VideoRam[VideoRam[2]]; // Need 2 clock cycles for this. //6
VideoRam[8] = 1; // Careful! the compiler won't complain about this! //7
// Verify what we entered: //8
```



```

$display("VideoRam[0] is %b",VideoRam[0]);           //9
$display("VideoRam[1] is %b",VideoRam[1]);           //10
$display("VideoRam[2] is %b",VideoRam[2]);           //11
$display("VideoRam[7] is %b",VideoRam[7]);           //12
end                                                    //13
endmodule                                             //14

VideoRam[0] is xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
VideoRam[1] is xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
VideoRam[2] is 00000000000000000000000000000001
VideoRam[7] is xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

```

We may also declare an **integer array** or **time array** in the same way as an array of `reg`, but there are no real arrays [Verilog LRM 3.9]:

```

module declarations_6;                               //1
integer Number [1:100];                             // Notice that size follows name //2
time Time_Log [1:1000];                             // - as in an array of reg. //3
// real Illegal [1:10];                             // Illegal. There are no real arrays.//4
endmodule                                            //5

```

11.2.3 Other Wire Types

There are the following other Verilog wire types (rarely used in ASIC design) [Verilog LRM 3.7.2]:

- `wand`, `wor`, `triand`, and `trior` model wired logic. Wiring, or dotting, the outputs of two gates generates a logic function (in emitter-coupled logic, ECL, or in an EPROM, for example). This is one area in which the logic values 'z' and 'x' are treated differently.
- `tri0` and `tri1` model resistive connections to VSS or VDD.
- `triereg` is like a wire but associates some capacitance with the net, so it can model charge storage.

There are also other keywords that may appear in declarations:

- `scalared` and `vectored` are properties of vectors [Verilog LRM 3.3.2].
- `small`, `medium`, and `large` model the charge strength of `triereg` connections [Verilog LRM 7].

11.2.4 Numbers

Constant numbers are integer or real constants [Verilog LRM 2.5]. **Integer constants** are written as

width'radix value

where `width` and `radix` are optional. The **radix** (or base) indicates the type of number: **decimal** (d or D), **hex** (h or H), **octal** (o or O), or **binary** (b or B). A number may be **sized** or **unsized**. The length of an unsized number is implementation dependent.

We can use '1' and '0' as numbers since they cannot be identifiers, but we must write 1'bx and 1'bz for 'x' and 'z'. A number may be declared as a **parameter** [Verilog LRM 3.10]. A parameter assignment belongs inside a module declaration and has **local scope** [Verilog LRM3.11]. **Real constants** are written using decimal (100.0) or scientific notation (1e2) and follow IEEE Std 754-1985 for double-precision floating-point numbers. Reals are rounded to the nearest integer, ties (numbers that end in .5) round away from zero [Verilog LRM 3.9.2], but not all implementations follow this rule (the output from the following code is from VeriWell, which rounds ties toward zero for negative integers).

```

module constants; //1
parameter H12_UNSIZE = 'h 12; // Unsized hex 12 = decimal 18. //2
parameter H12_SIZE = 6'h 12; // Sized hex 12 = decimal 18. //3
// Note: a space between base and value is OK. //4
// Note: '' (single apostrophes) are not the same as the ' character. //5
parameter D42 = 8'B0010_1010; // bin 101010 = dec 42 //6
// OK to use underscores to increase readability. //7
parameter D123 = 123; // Unsized decimal (the default). //8
parameter D63 = 8'o 77; // Sized octal, decimal 63. //9
// parameter ILLEGAL = 1'o9; // No 9's in octal numbers! //10
// A = 'hx and B = 'ox assume a 32 bit width. //11
parameter A = 'h x, B = 'o x, C = 8'b x, D = 'h z, E = 16'h ????. //12
// Note the use of ? instead of z, 16'h ????. is the same as 16'h zzzz. //13
// Also note the automatic extension to a width of 16 bits. //14
reg [3:0] B0011,Bxxx1,Bzzz1; real R1,R2,R3; integer I1,I3,I_3; //15
parameter BXZ = 8'b1x0x1z0z; //16
initial begin //17
B0011 = 4'b11; Bxxx1 = 4'bx1; Bzzz1 = 4'bz1; // Left padded. //18
R1 = 0.1e1; R2 = 2.0; R3 = 30E-01; // Real numbers. //19
I1 = 1.1; I3 = 2.5; I_3 = -2.5; // IEEE rounds away from 0. //20
end //21
initial begin #1; //22
$display //23
("H12_UNSIZE, H12_SIZE (hex) = %h, %h",H12_UNSIZE, H12_SIZE); //24
$display("D42 (bin) = %b",D42," (dec) = %d",D42); //25
$display("D123 (hex) = %h",D123," (dec) = %d",D123); //26
$display("D63 (oct) = %o",D63); //27
$display("A (hex) = %h",A," B (hex) = %h",B); //28
$display("C (hex) = %h",C," D (hex) = %h",D," E (hex) = %h",E); //29
$display("BXZ (bin) = %b",BXZ," (hex) = %h",BXZ); //30
$display("B0011, Bxxx1, Bzzz1 (bin) = %b, %b, %b",B0011,Bxxx1,Bzzz1); //31
$display("R1, R2, R3 (e, f, g) = %e, %f, %g", R1, R2, R3); //32
$display("I1, I3, I_3 (d) = %d, %d, %d", I1, I3, I_3); //33
end //34
endmodule //35

H12_UNSIZE, H12_SIZE (hex) = 00000012, 12
D42 (bin) = 00101010 (dec) = 42
D123 (hex) = 0000007b (dec) = 123

```

```

D63 (oct) = 077
A (hex) = xxxxxxxx B (hex) = xxxxxxxx
C (hex) = xx D (hex) = zzzzzzzz E (hex) = zzzz
BXZ (bin) = 1x0x1z0z (hex) = XZ
B0011, Bxxx1, Bzzz1 (bin) = 0011, xxx1, zzz1
R1, R2, R3 (e, f, g) = 1.000000e+00, 2.000000, 3
I1, I3, I_3 (d) =          1,          3,          -2

```

11.2.5 Negative Numbers

Integer numbers are **signed** (two's complement) or **unsigned**. The following example illustrates the handling of negative constants [Verilog LRM 3.2.2, 4.1.3]:

```

module negative_numbers; //1
parameter PA = -12, PB = -'d12, PC = -32'd12, PD = -4'd12; //2
integer IA , IB , IC , ID ; reg [31:0] RA , RB , RC , RD ; //3
initial begin #1; //4
IA = -12; IB = -'d12; IC = -32'd12; ID = -4'd12; //5
RA = -12; RB = -'d12; RC = -32'd12; RD = -4'd12; #1; //6
$display("          parameter    integer    reg[31:0]"); //7
$display ("-12          =",PA,IA,,,RA); //8
$displayh("          ",,,,PA,,,,,IA,,,,,RA); //9
$display ("- 'd12      =", ,PB,IB,,,RB); //10
$displayh("          ",,,,PB,,,,,IB,,,,,RB); //11
$display ("-32'd12    =", ,PC,IC,,,RC); //12
$displayh("          ",,,,PC,,,,,IC,,,,,RC); //13
$display ("-4'd12     =",,,,,,,,,PD,ID,,,RD); //14
$displayh("          ",,,,,,,,,PD,,,,,ID,,,,,RD); //15
end //16
endmodule //17

```

	parameter	integer	reg[31:0]
-12	= -12	-12	4294967284
	ffffffff4	ffffffff4	ffffffff4
- 'd12	= 4294967284	-12	4294967284
	ffffffff4	ffffffff4	ffffffff4
-32'd12	= 4294967284	-12	4294967284
	ffffffff4	ffffffff4	ffffffff4
-4'd12	= 4	-12	4294967284
	4	ffffffff4	ffffffff4

Verilog only “keeps track” of the sign of a negative constant if it is (1) assigned to an integer or (2) assigned to a parameter without using a base (essentially the same thing). In other cases (even though the bit representations may be identical to the signed number—hexadecimal `ffffffff4` in the previous example), a negative constant is treated as an unsigned number. Once Verilog “loses” the sign, keeping track of signed numbers becomes your responsibility (see also Section 11.3.1).

11.2.6 Strings

The code listings in this book use Courier font. The ISO/ANSI standard for the ASCII code defines the characters, but not the appearance of the graphic symbol in any particular font. The confusing characters are the quote and accent characters:

```

module characters; /*                                     //1
" is ASCII 34 (hex 22), double quote.                   //2
' is ASCII 39 (hex 27), tick or apostrophe.             //3
/ is ASCII 47 (hex 2F), forward slash.                  //4
\ is ASCII 92 (hex 5C), back slash.                     //5
` is ASCII 96 (hex 60), accent grave.                   //6
| is ASCII 124 (hex 7C), vertical bar.                   //7
There are no standards for the graphic symbols for codes above 128. //8
^ is 171 (hex AB), accent acute in almost all fonts.    //9
" is 210 (hex D2), open double quote, like 66 (in some fonts). //10
" is 211 (hex D3), close double quote, like 99 (in some fonts). //11
' is 212 (hex D4), open single quote, like 6 (in some fonts). //12
' is 213 (hex D5), close single quote, like 9 (in some fonts). //13
*/ endmodule                                           //14

```

Here is an example showing the use of **string constants** [Verilog LRM 2.6]:

```

module text;                                           //1
parameter A_String = "abc"; // string constant, must be on one line //2
parameter Say = "Say \"Hey!\""; //3
// use escape quote \" for an embedded quote           //4
parameter Tab = "\t"; // tab character                 //5
parameter NewLine = "\n"; // newline character         //6
parameter BackSlash = "\\"; // back slash             //7
parameter Tick = "\047"; // ASCII code for tick in octal //8
// parameter Illegal = "\500"; // illegal - no such ASCII code //9
initial begin //10
$display("A_String(str) = %s ",A_String," (hex) = %h ",A_String); //11
$display("Say = %s ",Say," Say \"Hey!\""); //12
$display("NewLine(str) = %s ",NewLine," (hex) = %h ",NewLine); //13
$display("\\(str) = %s ",BackSlash," (hex) = %h ",BackSlash); //14
$display("Tab(str) = %s ",Tab," (hex) = %h ",Tab,"1 newline..."); //15
$display("\n"); //16
$display("Tick(str) = %s ",Tick," (hex) = %h ",Tick); //17
#1.23; $display("Time is %t", $time); //18
end //19
endmodule //20

A_String(str) = abc (hex) = 616263
Say = Say \"Hey!\" Say "Hey!"
NewLine(str) = \n (hex) = 0a
\\(str) = \\ (hex) = 5c

```

```
Tab(str) = \t (hex) = 09 1 newline...
```

```
Tick(str) = ' (hex) = 27
Time is 1
```

Instead of parameters you may use a **define directive** that is a **compiler directive**, and not a statement [Verilog LRM 16]. The define directive has **global scope**:

```
module define; //1
`define G_BUSWIDTH 32 // Bus width parameter (G_ for global). //2
/* Note: there is no semicolon at end of a compiler directive. The
character ` is ASCII 96 (hex 60), accent grave, it slopes down from
left to right. It is not the tick or apostrophe character ' (ASCII 39
or hex 27)*/ //3
wire [`G_BUSWIDTH:0]MyBus; // A 32-bit bus. //4
endmodule //5
```

11.3 Operators

An expression uses any of the three types of operators: unary operators, binary operators, and a single ternary operator [Verilog LRM 4.1]. The Verilog operators are similar to those in the C programming language—except there is no autoincrement (++) or autodecrement (--) in Verilog. Table 11.1 shows the operators in their (increasing) order of precedence and Table 11.2 shows the unary operators. Here is an example that illustrates the use of the Verilog operators:

```
module operators; //1
parameter A10xz = {1'b1,1'b0,1'bx,1'bz}; // Concatenation and //2
parameter A01010101 = {4{2'b01}}; // replication, illegal for real. //3
// Arithmetic operators: +, -, *, /, and modulus % //4
parameter A1 = (3+2) %2; // The sign of a % b is the same as sign of a. //5
// Logical shift operators: << (left), >> (right) //6
parameter A2 = 4 >> 1; parameter A4 = 1 << 2; // Note: zero fill. //7
// Relational operators: <, <=, >, >= //8
initial if (1 > 2) $stop; //9
// Logical operators: ! (negation), && (and), || (or) //10
parameter B0 = !12; parameter B1 = 1 && 2; //11
reg [2:0] A00x; initial begin A00x = 'b111; A00x = !2'bx1; end //12
parameter C1 = 1 || (1/0); /* This may or may not cause an //13
error: the short-circuit behavior of && and || is undefined. An //14
evaluation including && or || may stop when an expression is known //15
to be true or false. */ //16
// == (logical equality), != (logical inequality) //17
```

TABLE 11.1 Verilog operators (in increasing order of precedence).

```

?: (conditional) [legal for real; associates right to left (others associate left to right)]
|| (logical or) [A smaller operand is zero-filled from its msb (0-fill); legal for real]
&& (logical and)[0-fill, legal for real]
| (bitwise or) ~| (bitwise nor) [0-fill]
^ (bitwise xor) ^~ ~^ (bitwise xnor, equivalence) [0-fill]
& (bitwise and) ~& (bitwise nand) [0-fill]
== (logical) != (logical) === (case) !== (case) [0-fill, logical versions are legal for real]
< (lt) <= (lt or equal) > (gt) >= (gt or equal) [0-fill, all are legal for real]
<< (shift left) >> (shift right) [zero fill; no -ve shifts; shift by x or z results in unknown]
+ (addition) - (subtraction) [if any bit is x or z for + - * / % then entire result is unknown]
* (multiply) / (divide) % (modulus) [integer divide truncates fraction; + - * / legal for real]
Unary operators: ! ~ & ~& | ~| ^ ^~ ^~ + - [see Table 11.2 for precedence]

```

TABLE 11.2 Verilog unary operators.

Operator	Name	Examples
!	logical negation	!123 is 'b0 [0, 1, or x for ambiguous; legal for real]
~	bitwise unary negation	~1'b10xz is 1'b01xx
&	unary reduction and	& 4'b1111 is 1'b1, & 2'bx1 is 1'bx, & 2'bz1 is 1'bx
~&	unary reduction nand	~& 4'b1111 is 1'b0, ~& 2'bx1 is 1'bx
	unary reduction or	Note:
~	unary reduction nor	Reduction is performed left (first bit) to right
^	unary reduction xor	Beware of the non-associative reduction operators
^~ ^~	unary reduction xnor	z is treated as x for all unary operators
+	unary plus	+2'bxz is +2'bxz [+m is the same as m; legal for real]
-	unary minus	-2'bxz is x [-m is unary minus m; legal for real]

```

parameter Ax = (1==1'bx); parameter Bx = (1'bx!=1'bz); //18
parameter D0 = (1==0); parameter D1 = (1==1); //19
// === case equality, !== (case inequality) //20
// The case operators only return true (1) or false (0). //21
parameter E0 = (1===1'bx); parameter E1 = 4'b01xz === 4'b01xz; //22
parameter F1 = (4'bxxxx === 4'bxxxx); //23

```

```

// Bitwise logical operators: //24
// ~ (negation), & (and), | (inclusive or), //25
// ^ (exclusive or), ~^ or ^~ (equivalence) //26
parameter A00 = 2'b01 & 2'b10; //27
// Unary logical reduction operators: //28
// & (and), ~& (nand), | (or), ~| (nor), //29
// ^ (xor), ~^ or ^~ (xnor) //30
parameter G1= & 4'b1111; //31
// Conditional expression f = a ? b : c [if (a) then f=b else f=c] //32
// if a=(x or z), then (bitwise) f=0 if b=c=0, f=1 if b=c=1, else f=x//33
reg H0, a, b, c; initial begin a=1; b=0; c=1; H0=a?b:c; end //34
reg[2:0] J01x, Jxxx, J01z, J011; //35
initial begin Jxxx = 3'bxxx; J01z = 3'b01z; J011 = 3'b011; //36
J01x = Jxxx ? J01z : J011; end // A bitwise result. //37
initial begin #1; //38
$display("A10xz=%b",A10xz," A01010101=%b",A01010101); //39
$display("A1=%0d",A1," A2=%0d",A2," A4=%0d",A4); //40
$display("B1=%b",B1," B0=%b",B0," A00x=%b",A00x); //41
$display("C1=%b",C1," Ax=%b",Ax," Bx=%b",Bx); //42
$display("D0=%b",D0," D1=%b",D1); //43
$display("E0=%b",E0," E1=%b",E1," F1=%b",F1); //44
$display("A00=%b",A00," G1=%b",G1," H0=%b",H0); //45
$display("J01x=%b",J01x); end //46
endmodule //47

A10xz=10xz A01010101=01010101
A1=1 A2=2 A4=4
B1=1 B0=0 A00x=00x
C1=1 Ax=x Bx=x
D0=0 D1=1
E0=0 E1=1 F1=1
A00=00 G1=1 H0=0
J01x=01x

```

11.3.1 Arithmetic

Arithmetic operations on n -bit objects are performed modulo 2^n in Verilog,

```

module modulo; reg [2:0] Seven; //1
initial begin //2
#1 Seven = 7; #1 $display("Before=", Seven); //3
#1 Seven = Seven + 1; #1 $display("After =", Seven); //4
end //5
endmodule //6

Before=7
After =0

```

Arithmetic operations in Verilog (addition, subtraction, comparison, and so on) on vectors (reg or wire) are predefined (Tables 11.1 and 11.2 show which operators are legal for `real`). This is a very important difference for ASIC designers from the situation in VHDL. However, there are some subtleties with Verilog arithmetic and negative numbers that are illustrated by the following example (based on an example in the LRM [Verilog LRM4.1.3]):

```

module LRM_arithmetic;                                     //1
integer IA, IB, IC, ID, IE; reg [15:0] RA, RB, RC;      //2
initial begin                                           //3
IA = -4'd12;           RA = IA / 3; // reg is treated as unsigned. //4
RB = -4'd12;           IB = RB / 3; //                               //5
IC = -4'd12 / 3;       RC = -12 / 3; // real is treated as signed //6
ID =   -12 / 3;        IE = IA / 3; // (two's complement).        //7
end                                                       //8
initial begin #1;                                       //9
$display("                hex    default");             //10
$display("IA = -4'd12    = %h%d", IA, IA);              //11
$display("RA = IA / 3    =    %h    %d", RA, RA);       //12
$display("RB = -4'd12    =    %h    %d", RB, RB);       //13
$display("IB = RB / 3    = %h%d", IB, IB);              //14
$display("IC = -4'd12 / 3 = %h%d", IC, IC);             //15
$display("RC = -12 / 3   =    %h    %d", RC, RC);       //16
$display("ID = -12 / 3   = %h%d", ID, ID);             //17
$display("IE = IA / 3    = %h%d", IE, IE);             //18
end                                                       //19
endmodule                                               //20

                hex    default
IA = -4'd12     = ffffffff4    -12
RA = IA / 3     =    fffc      65532
RB = -4'd12     =    fff4      65524
IB = RB / 3     = 00005551     21841
IC = -4'd12 / 3 = 55555551 1431655761
RC = -12 / 3    =    fffc      65532
ID = -12 / 3    = ffffffffcc    -4
IE = IA / 3     = ffffffffcc    -4

```

We might expect the results of all these divisions to be $-4 = -12/3$. For integer assignments, the results are correctly signed (ID and IE). Hex `fffc` (decimal 65532) is the 16-bit two's complement of -4 , so RA and RC are also correct if we keep track of the signs ourselves. The integer result IB is incorrect because Verilog treats RB as an unsigned number. Verilog also treats `-4'd12` as an unsigned number in the calculation of IC. Once Verilog "loses" a sign, it cannot get it back (see also Section 11.2.5).

11.4 Hierarchy

The **module** is the basic unit of code in the Verilog language [Verilog LRM 12.1],

```

module holiday_1(sat, sun, weekend);           //1
    input sat, sun; output weekend;           //2
    assign weekend = sat | sun;               //3
endmodule                                     //4

```

We do not have to explicitly declare the scalar wires: `saturday`, `sunday`, `weekend` because, since these wires appear in the module interface, they must be declared in an `input`, `output`, or `inout` statement and are thus implicitly declared. The **module interface** provides the means to interconnect two Verilog modules using **ports** [Verilog LRM 12.3]. Each port must be explicitly declared as one of **input**, **output**, or **inout**. Table 11.3 shows the characteristics of ports. Notice that a `reg` cannot be an `input` port or an `inout` port. This is to stop us trying to connect a `reg` to another `reg` that may hold a different value.

TABLE 11.3 Verilog ports.

Verilog port	input	output	inout
Characteristics	wire (or other net)	reg or wire (or other net) We can read an output port inside a module	wire (or other net)

Within a module we may **instantiate** other modules, but we cannot declare other modules. Ports are linked using **named association** or **positional association**,

```

`timescale 100s/1s // Units are 100 seconds with precision of 1s. //1
module life; wire [3:0] n; integer days; //2
    wire wake_7am, wake_8am; // Wake at 7 on weekdays else at 8. //3
    assign n = 1 + (days % 7); // n is day of the week (1-7) //4
always@(wake_8am or wake_7am) //5
    $display("Day=",n," hours=%0d ",($time/36)%24," 8am = ", //6
        wake_8am," 7am = ",wake_7am," m2.weekday = ", m2.weekday); //7
    initial days = 0; //8
    initial begin #(24*36*10);$finish; end // Run for 10 days. //9
    always #(24*36) days = days + 1; // Bump day every 24hrs. //10
    rest m1(n, wake_8am); // Module instantiation. //11
// Creates a copy of module rest with instance name m1, //12
// ports are linked using positional notation. //13
    work m2(.weekday(wake_7am), .day(n)); //14
// Creates a copy of module work with instance name m2, //15

```

```

// Ports are linked using named association. //16
endmodule //17

module rest(day, weekend); // Module definition. //1
// Notice the port names are different from the parent. //2
    input [3:0] day; output weekend; reg weekend; //3
    always begin #36 weekend = day > 5; end // Need a delay here. //4
endmodule //5

module work(day, weekday); //1
    input [3:0] day; output weekday; reg weekday; //2
    always begin #36 weekday = day < 6; end // Need a delay here. //3
endmodule //4

Day= 1 hours=0    8am = 0    7am = 0    m2.weekday = 0
Day= 1 hours=1    8am = 0    7am = 1    m2.weekday = 1
Day= 6 hours=1    8am = 1    7am = 0    m2.weekday = 0
Day= 1 hours=1    8am = 0    7am = 1    m2.weekday = 1

```

The port names in a module definition and the port names in the parent module may be different. We can **associate** (link or map) ports using the same order in the instantiating statement as we use in the module definition—such as instance `m1` in module `life`. Alternatively we can associate the ports by naming them—such as instance `m2` in module `life` (using a period `.` before the port name that we declared in the module definition). Identifiers in a module have local scope. If we want to refer to an identifier outside a module, we use a **hierarchical name** [Verilog LRM12.4] such as `m1.weekend` or `m2.weekday` (as in module `life`), for example. The compiler will first search downward (or inward) then upward (outward) to resolve a hierarchical name [Verilog LRM 12.4–12.5].

11.5 Procedures and Assignments

A Verilog **procedure** [Verilog LRM 9.9] is an `always` or `initial` statement, a `task`, or a `function`. The statements within a sequential block (statements that appear between a `begin` and an `end`) that is part of a procedure execute sequentially in the order in which they appear, but the procedure executes concurrently with other procedures. This is a fundamental difference from computer programming languages. Think of each procedure as a microprocessor running on its own and at the same time as all the other microprocessors (procedures). Before I discuss procedures in more detail, I shall discuss the two different types of assignment statements:

- *continuous assignments* that appear outside procedures
- *procedural assignments* that appear inside procedures

To illustrate the difference between these two types of assignments, consider again the example used in Section 11.4:

```

module holiday_1(sat, sun, weekend); //1
    input sat, sun; output weekend; //2
    assign weekend = sat | sun; // Assignment outside a procedure. //3
endmodule //4

```

We can change `weekend` to a `reg` instead of a `wire`, but then we must declare `weekend` and use a procedural assignment (inside a procedure—an `always` statement, for example) instead of a continuous assignment. We also need to add some delay (one time tick in the example that follows); otherwise the computer will never be able to get out of the `always` procedure to execute any other procedures:

```

module holiday_2(sat, sun, weekend); //1
    input sat, sun; output weekend; reg weekend; //2
    always #1 weekend = sat | sun; // Assignment inside a procedure. //3
endmodule //4

```

We shall cover the continuous assignment statement in the next section, which is followed by an explanation of sequential blocks and procedural assignment statements. Here is some skeleton code that illustrates where we may use these assignment statements:

```

module assignments //1
//... Continuous assignments go here. //2
always // beginning of a procedure //3
    begin // beginning of sequential block //4
//... Procedural assignments go here. //5
    end //6
endmodule //7

```

Table 11.4 at the end of Section 11.6 summarizes assignment statements, including two more forms of assignment—you may want to look at this table now.

11.5.1 Continuous Assignment Statement

A **continuous assignment statement** [Verilog LRM 6.1] assigns a value to a `wire` in a similar way that a real logic gate drives a real wire,

```

module assignment_1(); //1
wire pwr_good, pwr_on, pwr_stable; reg Ok, Fire; //2
assign pwr_stable = Ok & (!Fire); //3
assign pwr_on = 1; //4
assign pwr_good = pwr_on & pwr_stable; //5
initial begin Ok = 0; Fire = 0; #1 Ok = 1; #5 Fire = 1; end //6
initial begin $monitor("TIME=%0d", $time, " ON=", pwr_on, " STABLE=", //7
    pwr_stable, " OK=", Ok, " FIRE=", Fire, " GOOD=", pwr_good); //8

```



```

T=20 Clk=0 Y=1
T=30 Clk=1 Y=1
T=35 Clk=1 Y=0
T=40 Clk=0 Y=0
T=50 Clk=1 Y=0
T=55 Clk=1 Y=1
T=60 Clk=0 Y=1

```

11.5.3 Procedural Assignments

A **procedural assignment** [Verilog LRM 9.2] is similar to an assignment statement in a computer programming language such as C. In Verilog the value of an expression on the RHS of an assignment within a procedure (a procedural assignment) updates a reg (or memory element) on the LHS. In the absence of any *timing controls* (see Section 11.6), the reg is updated immediately when the statement executes. The reg holds its value until changed by another procedural assignment. Here is the BNF definition:

```
blocking_assignment ::= reg_lvalue = [delay_or_event_control] expression
```

(Notice this BNF definition is for a *blocking* assignment—a type of procedural assignment—see Section 11.6.4.) Here is an example of a procedural assignment (notice that a wire can only appear on the RHS of a procedural assignment):

```

module procedural_assign; reg Y, A; //1
always @(A) //2
    Y = A; // Procedural assignment. //3
initial begin A=0; #5; A=1; #5; A=0; #5; $finish; end //4
initial $monitor("T=%2g", $time, "A=", A, "Y=", Y); //5
endmodule //6

T= 0 A=0 Y=0
T= 5 A=1 Y=1
T=10 A=0 Y=0

```

11.6 Timing Controls and Delay

The statements within a sequential block are executed in order, but, in the absence of any delay, they all execute at the same simulation time—the current **time step**. In reality there are delays that are modeled using a timing control.

11.6.1 Timing Control

A **timing control** is either a delay control or an event control [Verilog LRM 9.7]. A **delay control** delays an assignment by a specified amount of time. A **timescale**

compiler directive is used to specify the units of time followed by the precision used to calculate time expressions,

```
`timescale 1ns/10ps // Units of time are ns. Round times to 10 ps.
```

Time units may only be s, ns, ps, or fs and the multiplier must be 1, 10, or 100. We can delay an assignment in two different ways:

- Sample the RHS immediately and then delay the assignment to the LHS.
- Wait for a specified time and then assign the value of the RHS to the LHS.

Here is an example of the first alternative (an **intra-assignment delay**):

```
x = #1 y;           // intra-assignment delay
```

The second alternative is **delayed assignment**:

```
#1 x = y;          // delayed assignment
```

These two alternatives are not the same. The intra-assignment delay is equivalent to the following code:

```
begin              // Equivalent to intra-assignment delay.
  hold = y;        // Sample and hold y immediately.
  #1;              // Delay.
  x = hold;        // Assignment to x. Overall same as x = #1 y.
end
```

In contrast, the delayed assignment is equivalent to a delay followed by an assignment as follows:

```
begin              // Equivalent to delayed assignment.
  #1;              // Delay.
  x = y;           // Assign y to x. Overall same as #1 x = y.
end
```

The other type of timing control, an **event control**, delays an assignment until a specified event occurs. Here is the formal definition:

```
event_control ::= @ event_identifier | @ (event_expression)
event_expression ::= expression | event_identifier
                 | posedge expression | negedge expression
                 | event_expression or event_expression
```

(Notice there are two different uses of 'or' in this simplified BNF definition—the last one, in bold, is part of the Verilog language, a keyword.) A positive edge (denoted by the keyword **posedge**) is a transition from '0' to '1' or 'x', or a transition from 'x' to '1'. A negative edge (**negedge**) is a transition from '1' to '0' or

'x', or a transition from 'x' to '0'. Transitions to or from 'z' do not count. Here are examples of event controls:

```

module delay_controls; reg X, Y, Clk, Dummy;           //1
always #1 Dummy=!Dummy; // Dummy clock, just for graphics. //2
// Examples of delay controls:                       //3
always begin #25 X=1;#10 X=0;#5; end                 //4
// An event control:                                 //5
always @(posedge Clk) Y=X; // Wait for +ve clock edge. //6
always #10 Clk = !Clk; // The real clock.           //7
initial begin Clk = 0;                               //8
    $display("T   Clk X Y");                         //9
    $monitor("%2g", $time,,, Clk,,, X,, Y);          //10
    $dumpvars;#100 $finish; end                      //11
endmodule                                           //12

```

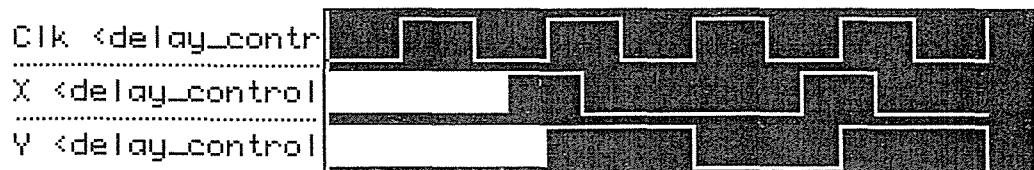
```

T   Clk X Y
0   0   x x
10  1   x x
20  0   x x
25  0   1 x
30  1   1 1
35  1   0 1
40  0   0 1
50  1   0 0
60  0   0 0
65  0   1 0
70  1   1 1
75  1   0 1
80  0   0 1
90  1   0 0

```

The dummy clock in `delay_controls` helps in the graphical waveform display of the results (it provides a one-time-tick timing grid when we zoom in, for example). Figure 11.1 shows the graphical output from the Waves viewer in VeriWell (white is used to represent the initial unknown values). The assignment statements to 'x' in the always statement repeat (every $25 + 10 + 5 = 40$ time ticks).

FIGURE 11.1 Output from the module `delay_controls`.



Events can be declared (as **named events**), triggered, and detected as follows:

```

module show_event; //1
reg clock; //2
event event_1, event_2; // Declare two named events. //3
always @(posedge clock) -> event_1; // Trigger event_1. //4
always @ event_1 //5
begin $display("Strike 1!!"); -> event_2; end // Trigger event_2. //6
always @ event_2 begin $display("Strike 2!!"); //7
$finish; end // Stop on detection of event_2. //8
always #10 clock = ~ clock; // We need a clock. //9
initial clock = 0; //10
endmodule //11

Strike 1!!
Strike 2!!

```

11.6.2 Data Slip

Consider this model for a shift register and the simulation output that follows:

```

module data_slip_1 (); reg Clk, D, Q1, Q2; //1
/***** bad sequential logic below *****/ //2
always @(posedge Clk) Q1 = D; //3
always @(posedge Clk) Q2 = Q1; // Data slips here! //4
/***** bad sequential logic above *****/ //5
initial begin Clk = 0; D = 1; end always #50 Clk = ~Clk; //6
initial begin $display("t Clk D Q1 Q2"); //7
$monitor("%3g", $time, Clk, D, Q1, Q2); //8
initial #400 $finish; // Run for 8 cycles. //9
initial $dumpvars; //10
endmodule //11

t Clk D Q1 Q2
0 0 1 x x
50 1 1 1 1
100 0 1 1 1
150 1 1 1 1
200 0 1 1 1
250 1 1 1 1
300 0 1 1 1
350 1 1 1 1

```

The first clock edge at $t = 50$ causes $Q1$ to be updated to the value of D at the clock edge (a '1'), and at the same time $Q2$ is updated to this new value of $Q1$. The data, D , has passed through both always statements. We call this problem **data slip**.

If we include delays in the always statements (labeled 3 and 4) in the preceding example, like this—

```
always @(posedge Clk) Q1 = #1 D; // The delays in the assignments //3
always @(posedge Clk) Q2 = #1 Q1; // fix the data slip. //4
```

—we obtain the correct output:

```
t  Clk D Q1 Q2
  0  0  1  x  x
  50 1  1  x  x
  51 1  1  1  x
 100 0  1  1  x
 150 1  1  1  x
 151 1  1  1  1
 200 0  1  1  1
 250 1  1  1  1
 300 0  1  1  1
 350 1  1  1  1
```

11.6.3 Wait Statement

The **wait statement** [Verilog LRM9.7.5] suspends a procedure until a condition becomes true. There must be another concurrent procedure that alters the condition (in this case the variable Done—in general the condition is an expression) in the following wait statement; otherwise we are placed on “infinite hold”:

```
wait (Done) $stop; // Wait until Done = 1 then stop.
```

Notice that the Verilog wait statement does not look for an event or a change in the condition; instead it is level-sensitive—it only cares that the condition is true.

```
module test_dff_wait; //1
reg D, Clock, Reset; dff_wait u1(D, Q, Clock, Reset); //2
initial begin D=1; Clock=0;Reset=1'b1; #15 Reset=1'b0; #20 D=0; end //3
always #10 Clock = !Clock; //4
initial begin $display("T Clk D Q Reset"); //5
    $monitor("%2g", $time, ,Clock, , ,D, ,Q, ,Reset); #50 $finish; end //6
endmodule //7
```

```
module dff_wait(D, Q, Clock, Reset); //1
output Q; input D, Clock, Reset; reg Q; wire D; //2
always @(posedge Clock) if (Reset != 1) Q = D; //3
always begin wait (Reset == 1) Q = 0; wait (Reset != 1); end //4
endmodule //5
```

```
T Clk D Q Reset
  0  0  1  0  1
 10  1  1  0  1
 15  1  1  0  0
 20  0  1  0  0
```

```

30 1 1 1 0
35 1 0 1 0
40 0 0 1 0

```

We must include wait statements in module `dff_wait` above to wait for both `Reset==1` and `Reset==0`. If we were to omit the wait statement for `Reset==0`, as in the following code:

```

module dff_wait(D,Q,Clock,Reset); //1
output Q; input D,Clock,Reset; reg Q; wire D; //2
always @(posedge Clock) if (Reset != 1) Q = D; //3
// We need another wait statement here or we shall spin forever. //4
always begin wait (Reset == 1) Q = 0; end //5
endmodule //6

```

the simulator would cycle endlessly, and we would need to press the 'Stop' button or 'CTRL-C' to halt the simulator. Here is the console window in VeriWell:

```

C1> .
T Clk D Q Reset      <- at this point nothing happens, so press CTRL-C
Interrupt at time 0
C1>

```

11.6.4 Blocking and Nonblocking Assignments

If a procedural assignment in a sequential block contains a timing control, then the execution of the following statement is delayed or **blocked**. For this reason a procedural assignment statement is also known as a **blocking procedural assignment statement** [Verilog LRM 9.2]. We covered this type of statement in Section 11.5.3. The **nonblocking procedural assignment statement** allows execution in a sequential block to continue and registers are all updated together at the end of the current time step. Both types of procedural assignment may contain timing controls. Here is an artificially complicated example that illustrates the different types of assignment:

```

module delay; //1
reg a,b,c,d,e,f,g,bds,bsd; //2
initial begin //3
a = 1; b = 0; // No delay control. //4
#1 b = 1; // Delayed assignment. //5
c = #1 1; // Intra-assignment delay. //6
#1; // Delay control. //7
d = 1; // //8
e <= #1 1; // Intra-assignment delay, nonblocking assignment //9
#1 f <= 1; // Delayed nonblocking assignment. //10
g <= 1; // Nonblocking assignment. //11
end //12
initial begin #1 bds = b; end // Delay then sample (ds). //13
initial begin bsd = #1 b; end // Sample then delay (sd). //14
initial begin $display("t a b c d e f g bds bsd"); //15

```

```

$monitor("%g", $time, , a, b, c, , d, e, f, , g, , bds, , , , bsd); end //16
endmodule //17

t a b c d e f g bds bsd
0 1 0 x x x x x x x
1 1 1 x x x x x 1 0
2 1 1 1 x x x x 1 0
3 1 1 1 1 x x x 1 0
4 1 1 1 1 1 1 1 1 0

```

Many synthesis tools will not allow us to use blocking and nonblocking procedural assignments to the same reg within the same sequential block.

11.6.5 Procedural Continuous Assignment

A **procedural continuous assignment statement** [Verilog LRM 9.3] (sometimes called a quasicontinuous assignment statement) is a special form of the assign statement that we use within a sequential block. For example, the following flip-flop model assigns to q depending on the clear, clr_, and preset, pre_, inputs (in general it is considered very bad form to use a trailing underscore to signify active-low signals as I have done to save space; you might use "_n" instead).

```

module dff_procedural_assign; //1
reg d, clr_, pre_, clk; wire q; dff_clr_pre dff_1(q, d, clr_, pre_, clk); //2
always #10 clk = ~clk; //3
initial begin clk = 0; clr_ = 1; pre_ = 1; d = 1; //4
    #20; d = 0; #20; pre_ = 0; #20; pre_ = 1; #20; clr_ = 0; //5
    #20; clr_ = 1; #20; d = 1; #20; $finish; end //6
initial begin //7
    $display("T CLK PRE_ CLR_ D Q"); //8
    $monitor("%3g", $time, , , clk, , , , pre_ , , , , clr_ , , , , d, , q); end //9
endmodule //10

module dff_clr_pre(q, d, clear_, preset_, clock); //1
output q; input d, clear_, preset_, clock; reg q; //2
always @(clear_ or preset_) //3
    if (!clear_) assign q = 0; // active-low clear //4
    else if (!preset_) assign q = 1; // active-low preset //5
    else deassign q; //6
always @(posedge clock) q = d; //7
endmodule //8

T CLK PRE_ CLR_ D Q
0 0 1 1 1 x
10 1 1 1 1 1
20 0 1 1 0 1
30 1 1 1 0 0
40 0 0 1 0 1
50 1 0 1 0 1

```

```

60 0 1 1 0 1
70 1 1 1 0 0
80 0 1 0 0 0
90 1 1 0 0 0
100 0 1 1 0 0
110 1 1 1 0 0
120 0 1 1 1 0
130 1 1 1 1 1

```

We have now seen all of the different forms of Verilog assignment statements. The following skeleton code shows where each type of statement belongs:

```

module all_assignments //1
//... continuous assignments. //2
always // beginning of procedure //3
  begin // beginning of sequential block //4
  //... blocking procedural assignments. //5
  //... nonblocking procedural assignments. //6
  //... procedural continuous assignments. //7
  end //8
endmodule //9

```

Table 11.4 summarizes the different types of assignments.

TABLE 11.4 Verilog assignment statements.

Type of Verilog assignment	Continuous assignment statement	Procedural assignment statement	Nonblocking procedural assignment statement	Procedural continuous assignment statement
Where it can occur	outside an always or initial statement, task, or function	inside an always or initial statement, task, or function	inside an always or initial statement, task, or function	always or initial statement, task, or function
Example	<code>wire [31:0] DataBus; assign DataBus = Enable ? Data : 32'bz</code>	<code>reg Y; always @(posedge clock) Y = 1;</code>	<code>reg Y; always Y <= 1;</code>	<code>always @(Enable) if(Enable) assign Q = D; else deassign Q;</code>
Valid LHS of assignment	net	register or memory element	register or memory element	net
Valid RHS of assignment	<expression> net, reg or memory element	<expression> net, reg or memory element	<expression> net, reg or memory element	<expression> net, reg or memory element
Book	11.5.1	11.5.3	11.6.4	11.6.5
Verilog LRM	6.1	9.2	9.2.2	9.3

11.7 Tasks and Functions

A **task** [Verilog LRM 10.2] is a type of procedure, called from another procedure. A task has both inputs and outputs but does not return a value. A task may call other tasks and functions. A **function** [Verilog LRM 10.3] is a procedure used in any expression, has at least one input, no outputs, and returns a single value. A function may not call a task. In Section 11.5 we covered all of the different Verilog procedures except for tasks and functions. Now that we have covered timing controls, we can explain the difference between tasks and functions: Tasks may contain timing controls but functions may not. The following two statements help illustrate the difference between a function and a task:

```
Call_A_Task_And_Wait (Input1, Input2, Output);
Result_Immediate = Call_A_Function (All_Inputs);
```

Functions are useful to model combinational logic (rather like a subroutine):

```
module F_subset_decode; reg [2:0]A, B, C, D, E, F; //1
initial begin A = 1; B = 0; D = 2; E = 3; //2
    C = subset_decode(A, B); F = subset_decode(D,E); //3
    $display("A B C D E F"); $display(A,,B,,C,,D,,E,,F); end //4
function [2:0] subset_decode; input [2:0] a, b; //5
    begin if (a <= b) subset_decode = a; else subset_decode = b; end //6
endfunction //7
endmodule //8

A B C D E F
1 0 0 2 3 2
```

11.8 Control Statements

In this section we shall discuss the Verilog `if`, `case`, `loop`, `disable`, `fork`, and `join` statements that control the flow of code execution.

11.8.1 Case and If Statement

An **if statement** [Verilog LRM 9.4] represents a two-way branch. In the following example, `switch` has to be true to execute `'Y = 1'`; otherwise `'Y = 0'` is executed:

```
if(switch) Y = 1; else Y = 0;
```

The **case statement** [Verilog LRM 9.5] represents a multiway branch. A **controlling expression** is matched with **case expressions** in each of the **case items** (or arms) to determine a match,

```
module test_mux; reg a, b, select; wire out; //1
mux mux_1(a, b, out, select); //2
```

```

initial begin #2; select = 0; a = 0; b = 1; //3
    #2; select = 1'bx; #2; select = 1'bz; #2; select = 1; end //4
initial $monitor("T=%2g", $time, " Select=", select, " Out=", out); //5
initial #10 $finish; //6
endmodule //7

module mux(a, b, mux_output, mux_select); //1
output mux_output; reg mux_output; //2
always begin //3
case(mux_select) //4
    0: mux_output = a; //5
    1: mux_output = b; //6
    default mux_output = 1'bx; // If select = x or z set output to x. //7
endcase //8
#1; // Need some delay, otherwise we'll spin forever. //9
end //10
endmodule //11

T= 0 Select=x Out=x
T= 2 Select=0 Out=x
T= 3 Select=0 Out=0
T= 4 Select=x Out=0
T= 5 Select=x Out=x
T= 6 Select=z Out=x
T= 8 Select=1 Out=x
T= 9 Select=1 Out=1

```

Notice that the case statement must be inside a sequential block (inside an always statement). Because the case statement is inside an always statement, it needs some delay; otherwise the simulation runs forever without advancing simulation time. The **casex statement** handles both 'z' and 'x' as don't care (so that they match any bit value), the **casez statement** handles 'z' bits, and only 'z' bits, as don't care. Bits in case expressions may be set to '?' representing don't care values, as follows:

```

casex (instruction_register[31:29])
    3b'??1 : add;
    3b'?1? : subtract;
    3b'1?? : branch;
endcase

```

11.8.2 Loop Statement

A loop statement [Verilog LRM 9.6] is a **for**, **while**, **repeat**, or **forever** statement. Here are four examples, one for each different type of loop statement, each of which performs the same function. The comments with each type of loop statement illustrate how the controls work:

```

module loop_1; //1
integer i; reg [31:0] DataBus; initial DataBus = 0; //2

```

```

initial begin                                                    //3
/***** Insert loop code after here. *****/
/* for(Execute this assignment once before starting loop; exit loop if
this expression is false; execute this assignment at end of loop before
the check for end of loop.) */
for(i = 0; i <= 15; i = i+1) DataBus[i] = 1;                    //4
/***** Insert loop code before here. *****/
end                                                                //5
initial begin                                                    //6
$display("DataBus = %b",DataBus);                                //7
#2; $display("DataBus = %b",DataBus); $finish;                  //8
end                                                                //9
endmodule                                                         //10

```

Here is the while statement code (to replace line 4 in module loop_1):

```

i = 0;
/* while(Execute next statement while this expression is true.) */
while(i <= 15) begin DataBus[i] = 1; i = i+1; end              //4

```

Here is the repeat statement code (to replace line 4 in module loop_1):

```

i = 0;
/* repeat(Execute next statement the number of times corresponding to
the evaluation of this expression at the beginning of the loop.) */
repeat(16) begin DataBus[i] = 1; i = i+1; end                  //4

```

Here is the forever statement code (to replace line 4 in module loop_1):

```

i = 0;
/* A forever statement loops continuously. */
forever begin : my_loop
    DataBus[i] = 1;
    if (i == 15) #1 disable my_loop; // Need to let time advance to exit.
    i = i+1;
end                                                                //4

```

The output for all four forms of looping statement is the same:

```

DataBus = 00000000000000000000000000000000
DataBus = 0000000000000000001111111111111111

```

11.8.3 Disable

The **disable statement** [Verilog LRM 11] stops the execution of a labeled sequential block and skips to the end of the block:

```

forever
begin: microprocessor_block // Labeled sequential block.
    @(posedge clock)
    if (reset) disable microprocessor_block; // Skip to end of block.

```

```

    else Execute_code;
end

```

Use the `disable` statement with caution in ASIC design. It is difficult to implement directly in hardware.

11.8.4 Fork and Join

The `fork` statement and `join` statement [Verilog LRM 9.8.2] allows the execution of two or more parallel threads in a **parallel block**:

```

module fork_1 //1
event eat_breakfast, read_paper; //2
initial begin //3
    fork //4
        @eat_breakfast; @read_paper; //5
    join //6
end //7
endmodule //8

```

This is another Verilog language feature that should be used with care in ASIC design, because it is difficult to implement in hardware.

11.9 Logic-Gate Modeling

Verilog has a set of built-in logic models and you may also define your own models.

11.9.1 Built-in Logic Models

Verilog's built-in logic models are the following **primitives** [Verilog LRM7]:

```
and, nand, nor, or, xor, xnor
```

You may use these primitives as you use modules. For example:

```

module primitive; //1
nand (strong0, strong1) #2.2 //2
    Nand_1(n001, n004, n005), //3
    Nand_2(n003, n001, n005, n002); //4
nand (n006, n005, n002); //5
endmodule //6

```

This module models three NAND gates (Figure 11.2). The first gate (line 3) is a two-input gate named `Nand_1`; the second gate (line 4) is a three-input gate named `Nand_2`; the third gate (line 5) is unnamed. The first two gates have strong drive strengths [Verilog LRM3.4] (these are the defaults anyway) and 2.2 ns delay; the third gate takes the default values for drive strength (strong) and delay (zero). The first port of a primitive gate is always the output port. The remaining ports for a primitive gate (any number of them) are the input ports.

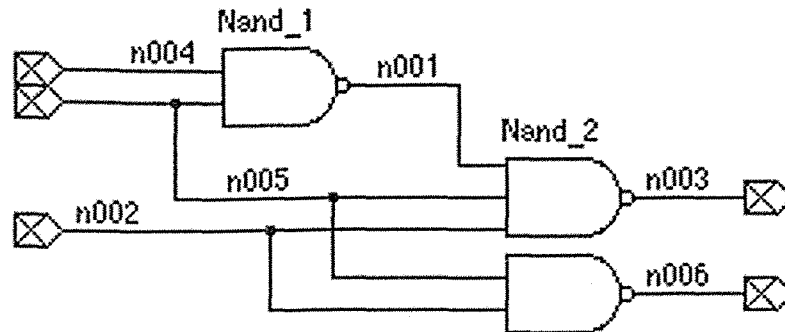


FIGURE 11.2 An example schematic (drawn with Capilano's DesignWorks) to illustrate the use of Verilog primitive gates.

Table 11.5 shows the definition of the and gate primitive (I use lowercase 'and' as the name of the Verilog primitive, rather than 'AND', since Verilog is case-sensitive). Notice that if one input to the primitive 'and' gate is zero, the output is zero, no matter what the other input is.

TABLE 11.5 Definition of the Verilog primitive 'and' gate.

'and'	0	1	x	z
0	0	0	0	0
1	0	1	x	x
x	0	x	x	x
z	0	x	x	x

11.9.2 User-Defined Primitives

We can define primitive gates (a **user-defined primitive** or **UDP**) using a truth-table specification [Verilog LRM8]. The first port of a UDP must be an output port, and this must be the only output port (we may not use vector or inout ports):

```
primitive Adder(Sum, InA, InB);           //1
output Sum; input InA, InB;             //2
table                                    //3
// inputs : output                       //4
00 : 0;                                  //5
01 : 1;                                  //6
10 : 1;                                  //7
```

```

11 : 0; //8
endtable //9
endprimitive //10

```

We may only specify the values '0', '1', and 'x' as inputs in a **UDP truth table**. Any 'z' input is treated as an 'x'. If there is no entry in a UDP truth table that exactly matches a set of inputs, the output is 'x' (unknown).

We can construct a UDP model for sequential logic by including a state in the UDP truth-table definition. The state goes between an input and an output in the table and the output then represents the next state. The following sequential UDP model also illustrates the use of shorthand notation in a UDP truth table:

```

primitive DLatch(Q, Clock, Data); //1
output Q; reg Q; input Clock, Data; //2
table //3
//inputs : present state : output (next state) //4
1 0 : ? : 0; // ? represents 0,1, or x (input or present state). //5
1 1 : b : 1; // b represents 0 or 1 (input or present state). //6
1 1 : x : 1; // Could have combined this with previous line. //7
0 ? : ? : -; // - represents no change in an output. //8
endtable //9
endprimitive //10

```

Be careful not to confuse the '?' in a UDP table (shorthand for '0', '1', or 'x') with the '?' in a constant that represents an extension to 'z' (Section 11.2.4) or the '?' in a case statement that represents don't care values (Section 11.8.1).

For sequential UDP models that need to detect edge transitions on inputs, there is another special truth-table notation (ab) that represents a change in logic value from a to b. For example, (01) represents a rising edge. There are also shorthand notations for various edges:

- * is (??)
- r is (01)
- f is (10)
- p is (01), (0x), or (x1)
- n is (10), (1x), or (x0)

```

primitive DFlipFlop(Q, Clock, Data); //1
output Q; reg Q; input Clock, Data; //2
table //3
//inputs : present state : output (next state) //4
r 0 : ? : 0 ; // rising edge, next state = output = 0 //5
r 1 : ? : 1 ; // rising edge, next state = output = 1 //6
(0x) 0 : 0 : 0 ; // rising edge, next state = output = 0 //7
(0x) 1 : 1 : 1 ; // rising edge, next state = output = 1 //8
(?0) ? : ? : - ; // falling edge, no change in output //9
? (??) : ? : - ; // no clock edge, no change in output //10

```

```

endtable //11
endprimitive //12

```

11.10 Modeling Delay

Verilog has a set of built-in methods to define delays. This is very important in ASIC physical design. Before we start layout, we can use ASIC cell library models written in Verilog that include logic delays as a function of fanout and estimated wiring loads. After we have completed layout, we can extract the wiring capacitance, allowing us to calculate the exact delay values. Using the techniques described in this section, we can then back-annotate our Verilog netlist with postlayout delays and complete a postlayout simulation.

We can complete this back-annotation process in a standard fashion since delay specification is part of the Verilog language. This makes working with an ASIC cell library and the ASIC foundry that will fabricate our ASIC much easier. Typically an ASIC library company might sell us a cell library complete with Verilog models that include all the minimum, typical, and maximum delays as well as the different values for rising and falling transitions. The ASIC foundry will provide us with a delay calculator that calculates the net delays (this is usually proprietary technology) from the layout. These delays are held in a separate file (the **Standard Delay Format, SDF**, is widely used) and then mapped to parameters in the Verilog models. If we complete back-annotation and a postlayout simulation using an approved cell library, the ASIC foundry will “sign off” on our design. This is basically a guarantee that our chip will work according to the simulation. This ability to design sign-off quality ASIC cell libraries is very important in the ASIC design process.

11.10.1 Net and Gate Delay

We saw how to specify a delay control for any statement in Section 11.6. In fact, Verilog allows us to specify minimum, typical, and maximum values for the delay as follows [Verilog LRM7.15]:

```

#(1.1:1.3:1.7) assign delay_a = a; // min:typ:max

```

We can also specify the delay properties of a `wire` in a similar fashion:

```

wire #(1.1:1.3:1.7) a_delay; // min:typ:max

```

We can specify delay in a `wire` declaration together with a continuous assignment as in the following example:

```

wire #(1.1:1.3:1.7) a_delay = a; // min:typ:max

```

but in this case the delay is associated with the driver and not with the `wire`.

In Section 11.9.1 we explained that we can specify a delay for a logic primitive. We can also specify minimum, typical, and maximum delays as well as separate delays for rising and falling transitions for primitives as follows [Verilog LRM4.3]:

```
nand #3.0 nd01(c, a, b);
nand #(2.6:3.0:3.4) nd02(d, a, b); // min:typ:max
nand #(2.8:3.2:3.4, 2.6:2.8:2.9) nd03(e, a, b);
// #(rising, falling) delay
```

The first NAND gate, nd01, has a delay of 3 ns (assuming we specified nanoseconds as the timescale) for both rising and falling delays. The NAND gate nd02 has a triplet for the delay; this corresponds to a minimum (2.6 ns), typical (3.0 ns), and a maximum delay (3.4 ns). The NAND gate nd03 has two triplets for the delay: The first triplet specifies the min/typ/max rising delay ('0' or 'x' or 'z' to '1'), and the second triplet specifies the min/typ/max falling delay ('1' or 'x' or 'z' to '0').

Some primitives can produce a high-impedance output, 'z'. In this case we can specify a triplet of delay values corresponding to rising transition, falling transition, and the delay to transition to 'z' (from '0' or '1' to 'z'—this is usually the delay for a three-state driver to turn off or float). We can do the same thing for net types,

```
wire #(0.5,0.6,0.7) a_z = a; // rise/fall/float delays
```

11.10.2 Pin-to-Pin Delay

The **specify block** [Verilog LRM 13] is a special construct in Verilog that allows the definition of **pin-to-pin delays** across a module. The use of a specify block can include the use of built-in system functions to check setup and hold times, for example. The following example illustrates how to specify pin-to-pin timing for a D flip-flop. We declare the timing parameters first followed by the paths. This example uses the UDP from Section 11.9.2, which does not include preset and clear (so only part of the flip-flop function is modeled), but includes the timing for preset and clear for illustration purposes.

```
module DFF_Spec; reg D, clk; //1
DFF_Part DFF1 (Q, clk, D, pre, clr); //2
initial begin D = 0; clk = 0; #1; clk = 1; end //3
initial $monitor("T=%2g", $time, " clk=", clk, " Q=", Q); //4
endmodule //5

module DFF_Part(Q, clk, D, pre, clr); //1
input clk, D, pre, clr; output Q; //2
DFlipFlop(Q, clk, D); // No preset or clear in this UDP. //3
specify //4
specparam //5
tPLH_clk_Q = 3, tPHL_clk_Q = 2.9, //6
tPLH_set_Q = 1.2, tPHL_set_Q = 1.1; //7
(clk => Q) = (tPLH_clk_Q, tPHL_clk_Q); //8
(pre, clr *> Q) = (tPLH_set_Q, tPHL_set_Q); //9
```

```

        endspecify //10
    endmodule //11

T= 0 clk=0 Q=x
T= 1 clk=1 Q=x
T= 4 clk=1 Q=0

```

There are the following two ways to specify paths (module DFF_part above uses both) [Verilog LRM13.3]:

- $x \Rightarrow y$ specifies a **parallel connection** (or parallel path) between x and y (x and y must have the same number of bits).
- $x * \Rightarrow y$ specifies a **full connection** (or full path) between x and y (every bit in x is connected to y). In this case x and y may be different sizes.

The delay of some logic cells depends on the state of the inputs. This can be modeled using a **state-dependent path delay**. Here is an example:

```

`timescale 1 ns / 100 fs //1
module M_Spec; reg A1, A2, B; M M1 (Z, A1, A2, B); //2
initial begin A1=0;A2=1;B=1;#5;B=0;#5;A1=1;A2=0;B=1;#5;B=0; end //3
initial //4
    $monitor("T=%4g",$realtime," A1=",A1," A2=",A2," B=",B," Z=",Z); //5
endmodule //6

`timescale 100 ps / 10 fs //1
module M(Z, A1, A2, B); input A1, A2, B; output Z; //2
or (Z1, A1, A2); nand (Z, Z1, B); // OAI21 //3
/*A1 A2 B Z Delay=10*100 ps unless indicated in the table below. //4
    0 0 0 1 //5
    0 0 1 1 //6
    0 1 0 1 B:0->1 Z:1->0 delay=t2 //7
    0 1 1 0 B:1->0 Z:0->1 delay=t1 //8
    1 0 0 1 B:0->1 Z:1->0 delay=t4 //9
    1 0 1 0 B:1->0 Z:0->1 delay=t3 //10
    1 1 0 1 //11
    1 1 1 0 */ //12
specify specparam t1 = 11, t2 = 12; specparam t3 = 13, t4 = 14; //13
    (A1 => Z) = 10; (A2 => Z) = 10; //14
    if (~A1) (B => Z) = (t1, t2); if (A1) (B => Z) = (t3, t4); //15
endspecify //16
endmodule //17

T= 0 A1=0 A2=1 B=1 Z=x
T= 1 A1=0 A2=1 B=1 Z=0
T= 5 A1=0 A2=1 B=0 Z=0
T= 6.1 A1=0 A2=1 B=0 Z=1
T= 10 A1=1 A2=0 B=1 Z=1
T= 11 A1=1 A2=0 B=1 Z=0
T= 15 A1=1 A2=0 B=0 Z=0
T=16.3 A1=1 A2=0 B=0 Z=1

```

11.11 Altering Parameters

Here is an example of a module that uses a parameter [Verilog LRM3.10, 12.2]:

```
module Vector_And(Z, A, B); //1
    parameter CARDINALITY = 1; //2
    input [CARDINALITY-1:0] A, B; //3
    output [CARDINALITY-1:0] Z; //4
    wire [CARDINALITY-1:0] Z = A & B; //5
endmodule //6
```

We can override this parameter when we instantiate the module as follows:

```
module Four_And_Gates(OutBus, InBusA, InBusB); //1
    input [3:0] InBusA, InBusB; output [3:0] OutBus; //2
    Vector_And #(4) My_AND(OutBus, InBusA, InBusB); // 4 AND gates //3
endmodule //4
```

The parameters of a module have local scope, but we may override them using a **defparam** statement and a hierarchical name, as in the following example:

```
module And_Gates(OutBus, InBusA, InBusB); //1
    parameter WIDTH = 1; //2
    input [WIDTH-1:0] InBusA, InBusB; output [WIDTH-1:0] OutBus; //3
    Vector_And #(WIDTH) My_And(OutBus, InBusA, InBusB); //4
endmodule //5

module Super_Size; defparam And_Gates.WIDTH = 4; endmodule //1
```

11.12 A Viterbi Decoder

This section describes an ASIC design for a Viterbi decoder using Verilog. Christeen Gray completed the original design as her MS thesis at the University of Hawaii (UH) working with VLSI Technology, using the Compass ASIC Synthesizer and a VLSI Technology cell library. The design was mapped from VLSI Technology design rules to Hewlett-Packard design rules; prototypes were fabricated by Hewlett-Packard (through Mosis) and tested at UH.

11.12.1 Viterbi Encoder

Viterbi encoding is widely used for satellite and other noisy **communications channels**. There are two important components of a channel using Viterbi encoding: the **Viterbi encoder** (at the transmitter) and the **Viterbi decoder** (at the receiver). A

Viterbi encoder includes extra information in the transmitted signal to reduce the probability of errors in the received signal that may be corrupted by noise.

I shall describe an encoder in which every two bits of a data stream are encoded into three bits for transmission. The ratio of input to output information in an encoder is the **rate** of the encoder; this is a rate 2/3 encoder. The following equations relate the three encoder output bits (Y_n^2 , Y_n^1 , and Y_n^0) to the two encoder input bits (X_n^2 and X_n^1) at a time nT :

$$\begin{aligned} Y_n^2 &= X_n^2 \\ Y_n^1 &= X_n^1 \oplus X_{n-2}^1 \\ Y_n^0 &= X_{n-1}^1 \end{aligned} \quad (11.1)$$

We can write the input bits as a single number. Thus, for example, if $X_n^2 = 1$ and $X_n^1 = 0$, we can write $X_n = 2$. Equation 11.1 defines a state machine with two memory elements for the two last input values for X_n^1 : X_{n-1}^1 and X_{n-2}^1 . These two state variables define four states: $\{X_{n-1}^1, X_{n-2}^1\}$, with $S_0 = \{0, 0\}$, $S_1 = \{1, 0\}$, $S_2 = \{0, 1\}$, and $S_3 = \{1, 1\}$. The 3-bit output Y_n is a function of the state and current 2-bit input X_n .

The following Verilog code describes the rate 2/3 encoder. This model uses two D flip-flops as the state register. When reset (using active-high input signal *res*) the encoder starts in state S_0 . In Verilog I represent Y_n^2 by *y2N*, for example.

```

/*****
/* module viterbi_encode
/*****
/* This is the encoder. X2N (msb) and X1N form the 2-bit input
message, XN. Example: if X2N=1, X1N=0, then XN=2. Y2N (msb), Y1N, and
Y0N form the 3-bit encoded signal, YN (for a total constellation of 8
PSK signals that will be transmitted). The encoder uses a state
machine with four states to generate the 3-bit output, YN, from the
2-bit input, XN. Example: the repeated input sequence XN = (X2N, X1N)
= 0, 1, 2, 3 produces the repeated output sequence YN = (Y2N, Y1N,
Y0N) = 1, 0, 5, 4. */
module viterbi_encode(X2N,X1N,Y2N,Y1N,Y0N,clk,res);
input X2N,X1N,clk,res; output Y2N,Y1N,Y0N;
wire X1N_1,X1N_2,Y2N,Y1N,Y0N;
dff dff_1(X1N,X1N_1,clk,res); dff dff_2(X1N_1,X1N_2,clk,res);
assign Y2N=X2N; assign Y1N=X1N ^ X1N_2; assign Y0N=X1N_1;
endmodule

```

Figure 11.3 shows the state diagram for this encoder. The first four rows of Table 11.6 show the four different transitions that can be made from state S_0 . For example, if we reset the encoder and the input is $X_n = 3$ ($X_n^2 = 1$ and $X_n^1 = 1$), then the output will be $Y_n = 6$ ($Y_n^2 = 1, Y_n^1 = 1, Y_n^0 = 0$) and the next state will be S_1 .

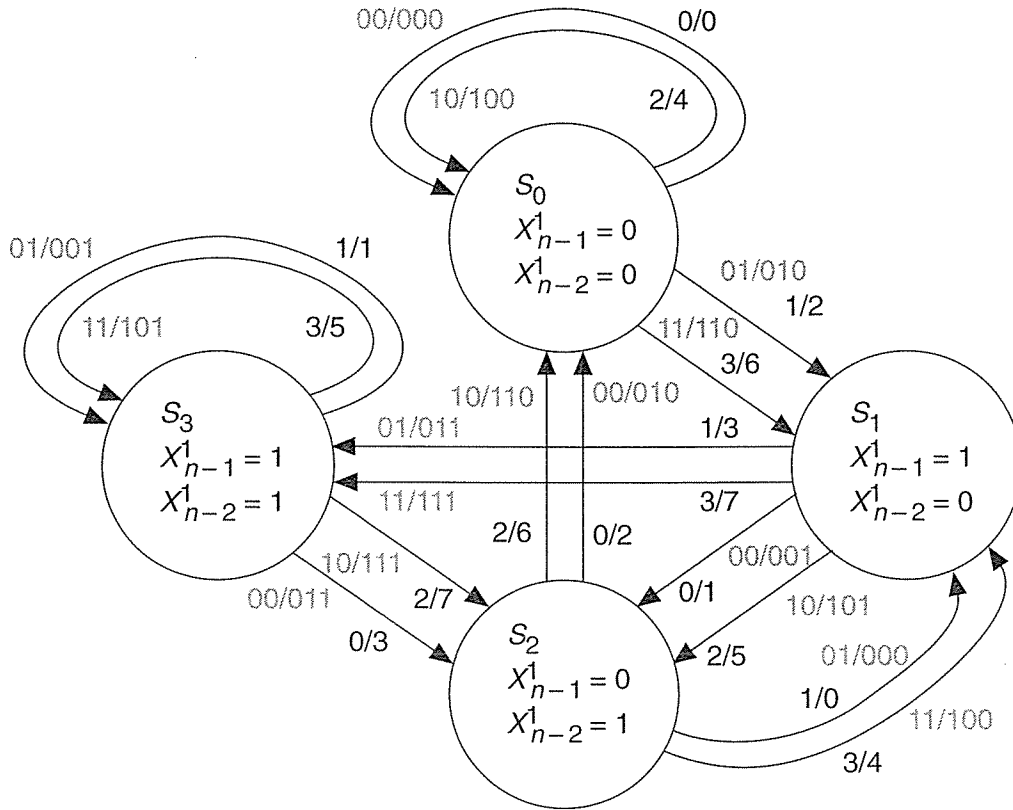


FIGURE 11.3 A state diagram for a rate 2/3 Viterbi encoder. The inputs and outputs are shown in binary as $X_n^2 X_n^1 / Y_n^2 Y_n^1 Y_n^0$, and in decimal as X_n / Y_n .

As an example, the repeated encoder input sequence $X_n = 0, 1, 2, 3, \dots$ produces the encoder output sequence $Y_n = 1, 0, 5, 4, \dots$ repeated. Table 11.7 shows the state transitions for this sequence, including the initialization steps.

Next we transmit the eight possible encoder outputs ($Y_n = 0-7$) as **signals** over our noisy communications channel (perhaps a microwave signal to a satellite) using the **signal constellation** shown in Figure 11.4. Typically this is done using **phase-shift keying (PSK)** with each signal position corresponding to a different phase shift in the transmitted carrier signal.

TABLE 11.6 State table for the rate 2/3 Viterbi encoder.

Present state	Inputs		State variables		Outputs			Next state	
					Y_n^2	Y_n^1	Y_n^0		
	X_n^2	X_n^1	X_{n-1}^1	X_{n-2}^1	$= X_n^2$	$= X_n^1 \oplus X_{n-2}^1$	$= X_{n-1}^1$	$\{X_{n-1}^1, X_{n-2}^1\}$	
S_0	0	0	0	0	0	0	0	00	S_0
S_0	0	1	0	0	0	1	0	10	S_1
S_0	1	0	0	0	1	0	0	00	S_0
S_0	1	1	0	0	1	1	0	10	S_1
S_1	0	0	1	0	0	0	1	01	S_2
S_1	0	1	1	0	0	1	1	11	S_3
S_1	1	0	1	0	1	0	1	01	S_2
S_1	1	1	1	0	1	1	1	11	S_3
S_2	0	0	0	1	0	1	0	00	S_0
S_2	0	1	0	1	0	0	0	10	S_1
S_2	1	0	0	1	1	1	0	00	S_0
S_2	1	1	0	1	1	0	0	10	S_1
S_3	0	0	1	1	0	1	1	01	S_2
S_3	0	1	1	1	0	0	1	11	S_3
S_3	1	0	1	1	1	1	1	01	S_2
S_3	1	1	1	1	1	0	1	11	S_3

FIGURE 11.4 The signal constellation for an 8PSK (phase-shift keyed) code.

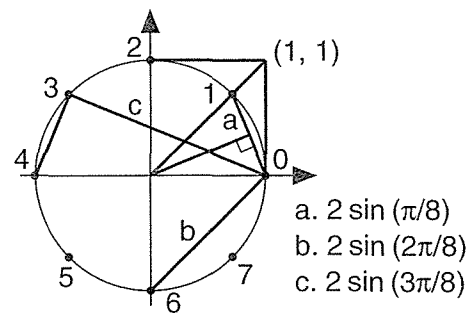


TABLE 11.7 A sequence of transmitted signals for the rate 2/3 Viterbi encoder

Time ns	Inputs		State variables		Outputs			Present state	Next state
	X_n^2	X_n^1	X_{n-1}^1	X_{n-2}^1	Y_n^2	Y_n^1	Y_n^0		
0	1	1	x	x	1	x	x	S_7	S_7
10	1	1	0	0	1	1	0	S_0	S_1
50	0	0	1	0	0	0	1	S_1	S_2
150	0	1	0	1	0	0	0	S_2	S_1
250	1	0	1	0	1	0	1	S_1	S_2
350	1	1	0	1	1	0	0	S_2	S_1
450	0	0	1	0	0	0	1	S_1	S_2
550	0	1	0	1	0	0	0	S_2	S_1
650	1	0	1	0	1	0	1	S_1	S_2
750	1	1	0	1	1	0	0	S_2	S_1
850	0	0	1	0	0	0	1	S_1	S_2
950	0	1	0	1	0	0	0	S_2	S_1

11.12.2 The Received Signal

The noisy signal enters the receiver. It is now our task to discover which of the eight possible signals were transmitted at each time step. First we calculate the distance of each received signal from each of the known eight positions in the signal constellation. Table 11.8 shows the distances between signals in the 8PSK constellation. We are going to assume that there is no noise in the channel to illustrate the operation of the Viterbi decoder, so that the distances in Table 11.8 represent the possible distance measures of our received signal from the 8PSK signals.

The distances, X , in the first column of Table 11.8 are the geometric or algebraic distances. We measure the **Euclidean distance**, $E = X^2$ shown as B (the binary quantized value of E) in Table 11.8. The rounding errors that result from conversion to fixed-width binary are **quantization errors** and are important in any practical implementation of the Viterbi decoder. The effect of the quantization error is to add a form of noise to the received signal.

The following code models the receiver section that digitizes the noisy analog received signal and computes the binary distance measures. Eight binary-distance measures, $in0-in7$, are generated each time a signal is received. Since each of the distance measures is 3 bits wide, there are a total of 24 bits (8×3) that form the digital inputs to the Viterbi decoder.

TABLE 11.8 Distance measures for Viterbi encoding (8PSK).

Signal	Algebraic distance from signal 0	X =Distance from signal 0	Euclidean distance $E=X^2$	B = binary quantized value of E	D = decimal value of B	Quantization error $Q=D-1.75E$
0	$2\sin(0\pi/8)$	0.00	0.00	000	0	0
1	$2\sin(1\pi/8)$	0.77	0.59	001	1	-0.0325
2	$2\sin(2\pi/8)$	1.41	2.00	100	4	0.5
3	$2\sin(3\pi/8)$	1.85	3.41	110	6	0.0325
4	$2\sin(4\pi/8)$	2.00	4.00	111	7	0
5	$2\sin(5\pi/8)$	1.85	3.41	110	6	0.0325
6	$2\sin(6\pi/8)$	1.41	2.00	100	4	0.5
7	$2\sin(7\pi/8)$	0.77	0.59	001	1	-0.0325

```

/*****
/* module viterbi_distances
/*****
/* This module simulates the front end of a receiver. Normally the
received analog signal (with noise) is converted into a series of
distance measures from the known eight possible transmitted PSK
signals: s0,...,s7. We are not simulating the analog part or noise in
this version, so we just take the digitally encoded 3-bit signal, Y,
from the encoder and convert it directly to the distance measures.
d[N] is the distance from signal = N to signal = 0
d[N] = (2*sin(N*PI/8))**2 in 3-bit binary (on the scale 2=100)
Example: d[3] = 1.85**2 = 3.41 = 110
inN is the distance from signal = N to encoder signal.
Example: in3 is the distance from signal = 3 to encoder signal.
d[N] is the distance from signal = N to encoder signal = 0.
If encoder signal = J, shift the distances by 8-J positions.
Example: if signal = 2, in0 is d[6], in1 is D[7], in2 is D[0], etc. */
module viterbi_distances
    (Y2N,Y1N,Y0N,clk,res,in0,in1,in2,in3,in4,in5,in6,in7);
input clk,res,Y2N,Y1N,Y0N; output in0,in1,in2,in3,in4,in5,in6,in7;
reg [2:0] J,in0,in1,in2,in3,in4,in5,in6,in7; reg [2:0] d [7:0];
initial begin d[0]='b000;d[1]='b001;d[2]='b100;d[3]='b110;
d[4]='b111;d[5]='b110;d[6]='b100;d[7]='b001; end
always @(Y2N or Y1N or Y0N) begin
J[0]=Y0N;J[1]=Y1N;J[2]=Y2N;
J=8-J;in0=d[J];J=J+1;in1=d[J];J=J+1;in2=d[J];J=J+1;in3=d[J];
J=J+1;in4=d[J];J=J+1;in5=d[J];J=J+1;in6=d[J];J=J+1;in7=d[J];
end endmodule

```

As an example, Table 11.9 shows the distance measures for the transmitted encoder output sequence $Y_n = 1, 0, 5, 4, \dots$ (repeated) corresponding to an encoder input of $X_n = 0, 1, 2, 3, \dots$ (repeated).

TABLE 11.9 Receiver distance measures for an example transmission sequence.

Time ns	Input X_n	Output Y_n	Present state	Next state	in0	in1	in2	in3	in4	in5	in6	in7
0	3	x	S_7	S_7	x	x	x	x	x	x	x	x
10	3	6	S_0	S_1	4	6	7	6	4	1	0	1
50	0	1	S_1	S_2	1	0	1	4	6	7	6	4
150	1	0	S_2	S_1	0	1	4	6	7	6	4	1
250	2	5	S_1	S_2	6	7	6	4	1	0	1	4
350	3	4	S_2	S_1	7	6	4	1	0	1	4	6
450	0	1	S_1	S_2	1	0	1	4	6	7	6	4
550	1	0	S_2	S_1	0	1	4	6	7	6	4	1
650	2	5	S_1	S_2	6	7	6	4	1	0	1	4
750	3	4	S_2	S_1	7	6	4	1	0	1	4	6
850	0	1	S_1	S_2	1	0	1	4	6	7	6	4
950	1	0	S_2	S_1	0	1	4	6	7	6	4	1

11.12.3 Testing the System

Here is a testbench for the entire system: encoder, receiver front end, and decoder:

```

/*****
/* module viterbi_test_CDD
/*****
/* This is the top-level module, viterbi_test_CDD, that models the
communications link. It contains three modules: viterbi_encode,
viterbi_distances, and viterbi. There is no analog and no noise in
this version. The 2-bit message, X, is encoded to a 3-bit signal, Y.
In this module the message X is generated using a simple counter.
The digital 3-bit signal Y is transmitted, received with noise as an
analog signal (not modeled here), and converted to a set of eight
3-bit distance measures, in0, ..., in7. The distance measures form
the input to the Viterbi decoder that reconstructs the transmitted
signal Y, with an error signal if the measures are inconsistent.
CDD = counter input, digital transmission, digital reception */
module viterbi_test_CDD;

```

```

wire Error; // decoder out
wire [2:0] Y, Out; // encoder out, decoder out
reg [1:0] X; // encoder inputs
reg Clk, Res; // clock and reset
wire [2:0] in0,in1,in2,in3,in4,in5,in6,in7;
always #500 $display("t Clk X Y Out Error");
initial $monitor("%4g", $time, Clk, X, Y, Out, Error);
initial $dumpvars; initial #3000 $finish;
always #50 Clk = ~Clk; initial begin Clk = 0;
X = 3; // No special reason to start at 3.
#60 Res = 1; #10 Res = 0; end // Hit reset after inputs are stable.
always @(posedge Clk) #1 X = X + 1; // Drive the input with a counter.
viterbi_encode v_1
(X[1], X[0], Y[2], Y[1], Y[0], Clk, Res);
viterbi_distances v_2
(Y[2], Y[1], Y[0], Clk, Res, in0, in1, in2, in3, in4, in5, in6, in7);
viterbi v_3
(in0, in1, in2, in3, in4, in5, in6, in7, Out, Clk, Res, Error);
endmodule

```

The Viterbi decoder takes the distance measures and calculates the most likely transmitted signal. It does this by keeping a running history of the previously received signals in a **path memory**. The path-memory length of this decoder is 12. By keeping a history of possible sequences and using the knowledge that the signals were generated by a state machine, it is possible to select the most likely sequences.

TABLE 11.10 Output from the Viterbi testbench

t	Clk	X	Y	Out	Error	t	Clk	X	Y	Out	Error
0	0	3	x	x	0	1351	1	1	0	0	0
50	1	3	x	x	0	1400	0	1	0	0	0
51	1	0	x	x	0	1450	1	1	0	0	0
60	1	0	0	0	0	1451	1	2	5	2	0
100	0	0	0	0	0	1500	0	2	5	2	0
150	1	0	0	0	0	1550	1	2	5	2	0
151	1	1	2	0	0	1551	1	3	4	5	0

Table 11.10 shows part of the simulation results from the testbench, `viterbi_test_CDD`, in tabular form. Figure 11.5 shows the Verilog simulator output from the testbench (displayed using VeriWell from Wellspring).

The system input or message, `x[1:0]`, is driven by a counter that repeats the sequence 0, 1, 2, 3, ... incrementing by 1 at each positive clock edge (with a delay of one time unit), starting with `x` equal to 3 at $t = 0$. The active-high reset signal, `Res`, is asserted at $t = 60$ for 10 time units. The encoder output, `Y[2:0]`, changes at $t = 151$, which is one time unit (the positive-edge-triggered D flip-flop model contains a

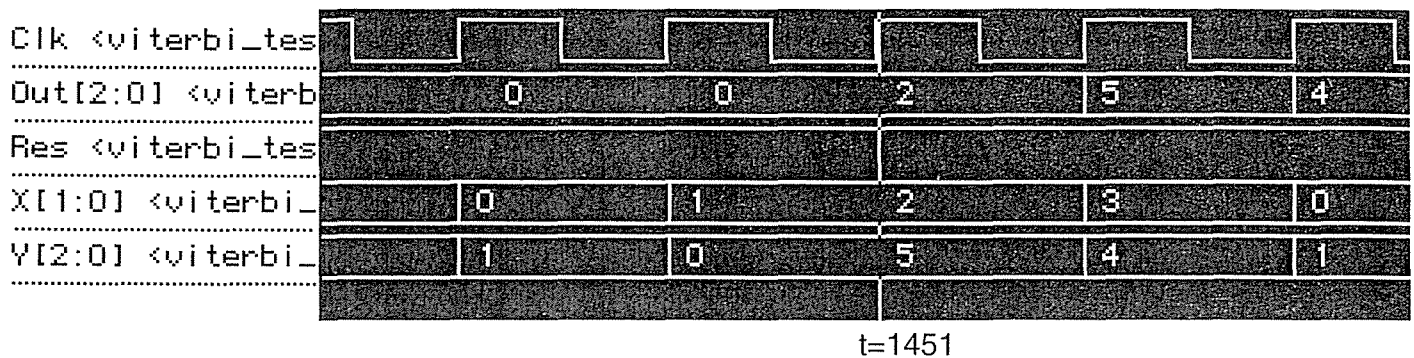
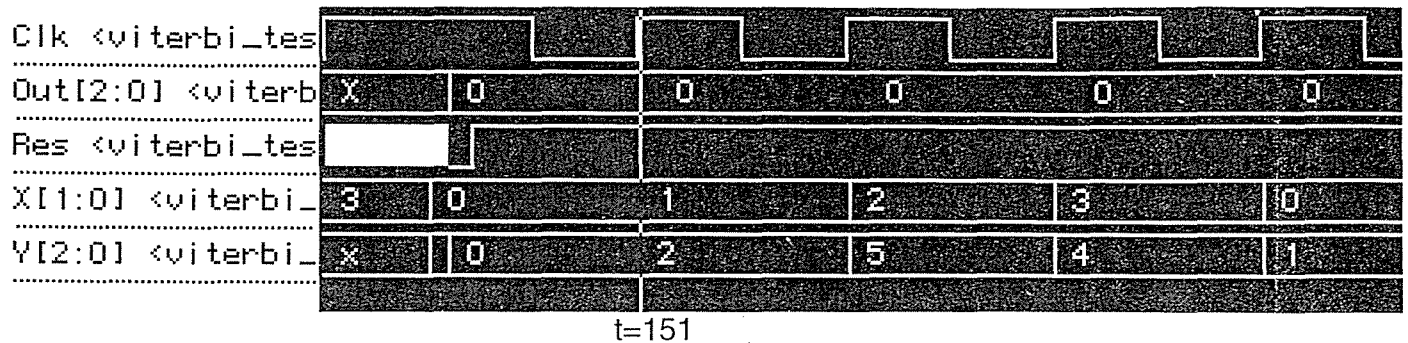


FIGURE 11.5 Viterbi encoder testbench simulation results. (Top) Initialization and the start of the encoder output sequence 2, 5, 4, 1, 0, ... on $Y[2:0]$ at $t = 151$. (Bottom) The appearance of the same encoder output sequence at the output of the decoder, $Out[2:0]$, at $t = 1451$, 1300 time units (13 positive clock edges) later.

one-time-unit delay) after the first positive clock edge (at $t = 150$) following the deassertion of the reset at $t = 70$. The encoder output sequence beginning at $t = 151$ is 2, 5, 4, 1, 0, ... and then the sequence 5, 4, 1, 0, ... repeats. This encoder output sequence is then imagined to be transmitted and received. The receiver module calculates the distance measures and passes them to the decoder. After 13 positive clock-edges (1300 time ticks) the transmitted sequence appears at the output, $out[2:0]$, beginning at $t = 1451$ with 2, 5, 4, 1, 0, ..., exactly the same as the encoder output.

11.12.4 Verilog Decoder Model

The Viterbi decoder model presented in this section is written for both simulation and synthesis. The Viterbi decoder makes extensive use of vector D flip-flops (registers). Early versions of Verilog-XL did not support vector instantiations of modules. In addition the inputs of UDPs may not be vectors and there are no primitive D flip-flops in Verilog. This makes instantiation of a register difficult other than by writing a separate module instance for each flip-flop.

The first solution to this problem is to use flip-flop models supplied with the synthesis tool such as the following:

```
asDff #(3) subout0(in0, sub0, clk, reset);
```

The `asDff` is a model in the Compass ASIC Synthesizer standard component library. This statement triggers the synthesis of three D flip-flops, with an input vector `ina` (with a range of three) connected to the D inputs, an output vector `sub0` (also with a range of three) connected to the Q flip-flop outputs, a common scalar clock signal, `clk`, and a common scalar `reset` signal. The disadvantage of this approach is that the names, functional behavior, and interfaces of the standard components are different for every software system.

The second solution, in new versions of Verilog-XL and other tools that support the IEEE standard, is to use vector instantiation as follows [LRM 7.5.1, 12.1.2]:

```
myDff subout0[0:2] (in0, sub0, clk, reset);
```

This instantiates three copies of a user-defined module or UDP called `myDff`. The disadvantage of this approach is that not all simulators and synthesizers support vector instantiation.

The third solution (which is used in the Viterbi decoder model) is to write a model that supports vector inputs and outputs. Here is an example D flip-flop model:

```

/*****
/*      module dff
/*****
/* A D flip-flop module. */

module dff(D,Q,Clock,Reset); // N.B. reset is active-low.
output Q; input D,Clock,Reset;
parameter CARDINALITY = 1; reg [CARDINALITY-1:0] Q;
wire [CARDINALITY-1:0] D;
always @(posedge Clock) if (Reset != 0) #1 Q = D;
always begin wait (Reset == 0); Q = 0; wait (Reset == 1); end
endmodule

```

We use this model by defining a parameter that specifies the bus width as follows:

```
dff #(3) subout0(in0, sub0, clk, reset);
```

The code that models the entire Viterbi decoder is listed below (Figure 12.6 on page 578 shows the block diagram). Notice the following:

- Comments explain the function of each module.
- Each module is about a page or less of code.

- Each module can be tested by itself.
- The code is as simple as possible avoiding clever coding techniques.

The code is not flexible, because bit widths are fixed rather than using parameters. A model with parameters for rate, signal constellation, distance measure resolution, and path memory length is considerably more complex. We shall use this Viterbi decoder design again when we discuss logic synthesis in Chapter 12, test in Chapter 14, floorplanning and placement in Chapter 16, and routing in Chapter 17.

```
/* Verilog code for a Viterbi decoder. The decoder assumes a rate
2/3 encoder, 8 PSK modulation, and trellis coding. The viterbi module
contains eight submodules: subset_decode, metric, compute_metric,
compare_select, reduce, pathin, path_memory, and output_decision.
```

```
The decoder accepts eight 3-bit measures of ||r-si||**2 and, after
an initial delay of thirteen clock cycles, the output is the best
estimate of the signal transmitted. The distance measures are the
Euclidean distances between the received signal r (with noise) and
each of the (in this case eight) possible transmitted signals s0 to s7.
```

```
Original by Christeen Gray, University of Hawaii. Heavily modified
by MJSS; any errors are mine. Use freely. */
```

```
/* module viterbi */
/* This is the top level of the Viterbi decoder. The eight input
signals {in0,...,in7} represent the distance measures, ||r-si||**2.
The other input signals are clk and reset. The output signals are
out and error. */
```

```
module viterbi
    (in0,in1,in2,in3,in4,in5,in6,in7,
     out,clk,reset,error);
input [2:0] in0,in1,in2,in3,in4,in5,in6,in7;
output [2:0] out; input clk,reset; output error;
wire sout0,sout1,sout2,sout3;
wire [2:0] s0,s1,s2,s3;
wire [4:0] m_in0,m_in1,m_in2,m_in3;
wire [4:0] m_out0,m_out1,m_out2,m_out3;
wire [4:0] p0_0,p2_0,p0_1,p2_1,p1_2,p3_2,p1_3,p3_3;
wire ACS0,ACS1,ACS2,ACS3;
wire [4:0] out0,out1,out2,out3;
wire [1:0] control;
wire [2:0] p0,p1,p2,p3;
wire [11:0] path0;

    subset_decode u1(in0,in1,in2,in3,in4,in5,in6,in7,
        s0,s1,s2,s3,sout0,sout1,sout2,sout3,clk,reset);
    metric u2(m_in0,m_in1,m_in2,m_in3,m_out0,
        m_out1,m_out2,m_out3,clk,reset);
    compute_metric u3(m_out0,m_out1,m_out2,m_out3,s0,s1,s2,s3,
```



```

    p0_0,p2_0,p0_1,p2_1,p1_2,p3_2,p1_3,p3_3,error);
compare_select u4(p0_0,p2_0,p0_1,p2_1,p1_2,p3_2,p1_3,p3_3,
    out0,out1,out2,out3,ACS0,ACS1,ACS2,ACS3);
reduce u5(out0,out1,out2,out3,
    m_in0,m_in1,m_in2,m_in3,control);
pathin u6(sout0,sout1,sout2,sout3,
    ACS0,ACS1,ACS2,ACS3,path0,clk,reset);
path_memory u7(p0,p1,p2,p3,path0,clk,reset,
    ACS0,ACS1,ACS2,ACS3);
output_decision u8(p0,p1,p2,p3,control,out);
endmodule

/*****
/* module subset_decode */
/*****
/* This module chooses the signal corresponding to the smallest of
each set  $\{|r-s0|^{**2}, |r-s4|^{**2}\}$ ,  $\{|r-s1|^{**2}, |r-s5|^{**2}\}$ ,
 $\{|r-s2|^{**2}, |r-s6|^{**2}\}$ ,  $\{|r-s3|^{**2}, |r-s7|^{**2}\}$ . Therefore
there are eight input signals and four output signals for the
distance measures. The signals sout0, ..., sout3 are used to control
the path memory. The statement dff #(3) instantiates a vector array
of 3 D flip-flops. */
module subset_decode
    (in0,in1,in2,in3,in4,in5,in6,in7,
    s0,s1,s2,s3,
    sout0,sout1,sout2,sout3,
    clk,reset);
input [2:0] in0,in1,in2,in3,in4,in5,in6,in7;
output [2:0] s0,s1,s2,s3;
output sout0,sout1,sout2,sout3;
input clk,reset;
wire [2:0] sub0,sub1,sub2,sub3,sub4,sub5,sub6,sub7;

    dff #(3) subout0(in0, sub0, clk, reset);
    dff #(3) subout1(in1, sub1, clk, reset);
    dff #(3) subout2(in2, sub2, clk, reset);
    dff #(3) subout3(in3, sub3, clk, reset);
    dff #(3) subout4(in4, sub4, clk, reset);
    dff #(3) subout5(in5, sub5, clk, reset);
    dff #(3) subout6(in6, sub6, clk, reset);
    dff #(3) subout7(in7, sub7, clk, reset);

    function [2:0] subset_decode; input [2:0] a,b;
    begin
        subset_decode = 0;
        if (a<=b) subset_decode = a; else subset_decode = b;
    end
endfunction

```

```

function set_control; input [2:0] a,b;
begin
    if (a<=b) set_control = 0; else set_control = 1;
end
endfunction

assign s0 = subset_decode (sub0,sub4);
assign s1 = subset_decode (sub1,sub5);
assign s2 = subset_decode (sub2,sub6);
assign s3 = subset_decode (sub3,sub7);
assign sout0 = set_control(sub0,sub4);
assign sout1 = set_control(sub1,sub5);
assign sout2 = set_control(sub2,sub6);
assign sout3 = set_control(sub3,sub7);
endmodule

/*****
/*  module compute_metric
/*****
/* This module computes the sum of path memory and the distance for
each path entering a state of the trellis. For the four states,
there are two paths entering it; therefore eight sums are computed
in this module. The path metrics and output sums are 5 bits wide.
The output sum is bounded and should never be greater than 5 bits
for a valid input signal. The overflow from the sum is the error
output and indicates an invalid input signal.*/
module compute_metric
    (m_out0,m_out1,m_out2,m_out3,
    s0,s1,s2,s3,p0_0,p2_0,
    p0_1,p2_1,p1_2,p3_2,p1_3,p3_3,
    error);
input [4:0] m_out0,m_out1,m_out2,m_out3;
input [2:0] s0,s1,s2,s3;
output [4:0] p0_0,p2_0,p0_1,p2_1,p1_2,p3_2,p1_3,p3_3;
output error;

assign
    p0_0 = m_out0 + s0,
    p2_0 = m_out2 + s2,
    p0_1 = m_out0 + s2,
    p2_1 = m_out2 + s0,
    p1_2 = m_out1 + s1,
    p3_2 = m_out3 + s3,
    p1_3 = m_out1 + s3,
    p3_3 = m_out3 + s1;

function is_error; input x1,x2,x3,x4,x5,x6,x7,x8;
begin
    if (x1||x2||x3||x4||x5||x6||x7||x8) is_error = 1;

```

```

        else is_error = 0;
    end
endfunction

assign error = is_error(p0_0[4],p2_0[4],p0_1[4],p2_1[4],
    p1_2[4],p3_2[4],p1_3[4],p3_3[4]);
endmodule

/*****
/*  module compare_select
/*****
/* This module compares the summations from the compute_metric
module and selects the metric and path with the lowest value. The
output of this module is saved as the new path metric for each
state. The ACS output signals are used to control the path memory of
the decoder. */
module compare_select
    (p0_0,p2_0,p0_1,p2_1,p1_2,p3_2,p1_3,p3_3,
    out0,out1,out2,out3,
    ACS0,ACS1,ACS2,ACS3);
input  [4:0] p0_0,p2_0,p0_1,p2_1,p1_2,p3_2,p1_3,p3_3;
output [4:0] out0,out1,out2,out3;
output ACS0,ACS1,ACS2,ACS3;

function [4:0] find_min_metric; input [4:0] a,b;
begin
    if (a <= b) find_min_metric = a; else find_min_metric = b;
end
endfunction

function set_control; input [4:0] a,b;
begin
    if (a <= b) set_control = 0; else set_control = 1;
end
endfunction

assign out0 = find_min_metric(p0_0,p2_0);
assign out1 = find_min_metric(p0_1,p2_1);
assign out2 = find_min_metric(p1_2,p3_2);
assign out3 = find_min_metric(p1_3,p3_3);
assign ACS0 = set_control (p0_0,p2_0);
assign ACS1 = set_control (p0_1,p2_1);
assign ACS2 = set_control (p1_2,p3_2);
assign ACS3 = set_control (p1_3,p3_3);
endmodule

/*****
/*  module path
/*****
/* This is the basic unit for the path memory of the Viterbi

```

decoder. It consists of four 3-bit D flip-flops in parallel. There is a 2:1 mux at each D flip-flop input. The statement `dff #(12)` instantiates a vector array of 12 flip-flops. */

```

module path(in,out,clk,reset,ACS0,ACS1,ACS2,ACS3);
input [11:0] in; output [11:0] out;
input clk,reset,ACS0,ACS1,ACS2,ACS3; wire [11:0] p_in;

dff #(12) path0(p_in,out,clk,reset);

function [2:0] shift_path; input [2:0] a,b; input control;
begin
    if (control == 0) shift_path = a; else shift_path = b;
end
endfunction

assign p_in[11:9] = shift_path(in[11:9],in[5:3],ACS0);
assign p_in[ 8:6] = shift_path(in[11:9],in[5:3],ACS1);
assign p_in[ 5:3] = shift_path(in[8: 6],in[2:0],ACS2);
assign p_in[ 2:0] = shift_path(in[8: 6],in[2:0],ACS3);
endmodule

/*****
/*  module path_memory
/*****
/* This module consists of an array of memory elements (D
flip-flops) that store and shift the path memory as new signals are
added to the four paths (or four most likely sequences of signals).
This module instantiates 11 instances of the path module. */
module path_memory
    (p0,p1,p2,p3,
    path0,clk,reset,
    ACS0,ACS1,ACS2,ACS3);
output [2:0] p0,p1,p2,p3; input [11:0] path0;
input clk,reset,ACS0,ACS1,ACS2,ACS3;
wire [11:0]out1,out2,out3,out4,out5,out6,out7,out8,out9,out10,out11;
    path x1 (path0,out1 ,clk,reset,ACS0,ACS1,ACS2,ACS3),
        x2 (out1, out2 ,clk,reset,ACS0,ACS1,ACS2,ACS3),
        x3 (out2, out3 ,clk,reset,ACS0,ACS1,ACS2,ACS3),
        x4 (out3, out4 ,clk,reset,ACS0,ACS1,ACS2,ACS3),
        x5 (out4, out5 ,clk,reset,ACS0,ACS1,ACS2,ACS3),
        x6 (out5, out6 ,clk,reset,ACS0,ACS1,ACS2,ACS3),
        x7 (out6, out7 ,clk,reset,ACS0,ACS1,ACS2,ACS3),
        x8 (out7, out8 ,clk,reset,ACS0,ACS1,ACS2,ACS3),
        x9 (out8, out9 ,clk,reset,ACS0,ACS1,ACS2,ACS3),
        x10(out9, out10,clk,reset,ACS0,ACS1,ACS2,ACS3),
        x11(out10,out11,clk,reset,ACS0,ACS1,ACS2,ACS3);
assign p0 = out11[11:9];
assign p1 = out11[ 8:6];
assign p2 = out11[ 5:3];

```

```

assign p3 = out11[ 2:0];
endmodule

/*****
/*  module pathin
/*****
/* This module determines the input signal to the path for each of
the four paths. Control signals from the subset decoder and compare
select modules are used to store the correct signal. The statement
dff #(12) instantiates a vector array of 12 flip-flops. */
module pathin
    (sout0,sout1,sout2,sout3,
     ACS0,ACS1,ACS2,ACS3,
     path0,clk,reset);
input sout0,sout1,sout2,sout3,ACS0,ACS1,ACS2,ACS3;
input clk,reset; output [11:0] path0;
wire [2:0] sig0,sig1,sig2,sig3; wire [11:0] path_in;
dff #(12) firstpath(path_in,path0,clk,reset);

function [2:0] subset0; input sout0;
begin
    if(sout0 == 0) subset0 = 0; else subset0 = 4;
end
endfunction

function [2:0] subset1; input sout1;
begin
    if(sout1 == 0) subset1 = 1; else subset1 = 5;
end
endfunction

function [2:0] subset2; input sout2;
begin
    if(sout2 == 0) subset2 = 2; else subset2 = 6;
end
endfunction

function [2:0] subset3; input sout3;
begin
    if(sout3 == 0) subset3 = 3; else subset3 = 7;
end
endfunction

function [2:0] find_path; input [2:0] a,b; input control;
begin
    if(control==0) find_path = a; else find_path = b;
end
endfunction

assign sig0 = subset0(sout0);

```

```

assign sig1 = subset1(sout1);
assign sig2 = subset2(sout2);
assign sig3 = subset3(sout3);
assign path_in[11:9] = find_path(sig0,sig2,ACS0);
assign path_in[ 8:6] = find_path(sig2,sig0,ACS1);
assign path_in[ 5:3] = find_path(sig1,sig3,ACS2);
assign path_in[ 2:0] = find_path(sig3,sig1,ACS3);
endmodule

/*****
/*  module metric
*****/
/* The registers created in this module (using D flip-flops) store
the four path metrics. Each register is 5 bits wide. The statement
dff #(5) instantiates a vector array of 5 flip-flops. */
module metric
    (m_in0,m_in1,m_in2,m_in3,
     m_out0,m_out1,m_out2,m_out3,
     clk,reset);
input [4:0] m_in0,m_in1,m_in2,m_in3;
output [4:0] m_out0,m_out1,m_out2,m_out3;
input clk,reset;
    dff #(5) metric3(m_in3, m_out3, clk, reset);
    dff #(5) metric2(m_in2, m_out2, clk, reset);
    dff #(5) metric1(m_in1, m_out1, clk, reset);
    dff #(5) metric0(m_in0, m_out0, clk, reset);
endmodule

/*****
/*  module output_decision
*****/
/* This module decides the output signal based on the path that
corresponds to the smallest metric. The control signal comes from
the reduce module. */
module output_decision(p0,p1,p2,p3,control,out);
    input [2:0] p0,p1,p2,p3; input [1:0] control; output [2:0] out;
    function [2:0] decide;
    input [2:0] p0,p1,p2,p3; input [1:0] control;
    begin
        if(control == 0) decide = p0;
        else if(control == 1) decide = p1;
        else if(control == 2) decide = p2;
        else decide = p3;
    end
    endfunction
assign out = decide(p0,p1,p2,p3,control);
endmodule

```

```

/*****
/*  module reduce
/*****
/* This module reduces the metrics after the addition and compare
operations. This algorithm selects the smallest metric and subtracts
it from all the other metrics. */

module reduce
    (in0,in1,in2,in3,
     m_in0,m_in1,m_in2,m_in3,
     control);
    input [4:0] in0,in1,in2,in3;
    output [4:0] m_in0,m_in1,m_in2,m_in3;
    output [1:0] control; wire [4:0] smallest;

    function [4:0] find_smallest;
        input [4:0] in0,in1,in2,in3; reg [4:0] a,b;
        begin
            if(in0 <= in1) a = in0; else a = in1;
            if(in2 <= in3) b = in2; else b = in3;
            if(a <= b) find_smallest = a;
            else find_smallest = b;
        end
    endfunction

    function [1:0] smallest_no;
    input [4:0] in0,in1,in2,in3,smallest;
    begin
        if(smallest == in0) smallest_no = 0;
        else if (smallest == in1) smallest_no = 1;
        else if (smallest == in2) smallest_no = 2;
        else smallest_no = 3;
    end
    endfunction

    assign smallest = find_smallest(in0,in1,in2,in3);
    assign m_in0 = in0 - smallest;
    assign m_in1 = in1 - smallest;
    assign m_in2 = in2 - smallest;
    assign m_in3 = in3 - smallest;
    assign control = smallest_no(in0,in1,in2,in3,smallest);
endmodule

```

11.13 Other Verilog Features

This section covers some of the more advanced Verilog features. **System tasks** and **functions** are defined as part of the IEEE Verilog standard [Verilog LRM14].

11.13.1 Display Tasks

The following code illustrates the **display system tasks** [Verilog LRM 14.1]:

```

module test_display; // display system tasks:
initial begin $display ("string, variables, or expression");
/* format specifications work like printf in C:
    %d=decimal %b=binary %s=string %h=hex %o=octal
    %c=character %m=hierarchical name %v=strength %t=time format
    %e=scientific %f=decimal %g=shortest
examples: %d uses default width %0d uses minimum width
    %7.3g uses 7 spaces with 3 digits after decimal point */
// $displayb, $displayh, $displayo print in b, h, o formats
// $write, $strobe, $monitor also have b, h, o versions
$write("write"); // as $display, but without newline at end of line
$strobe("strobe"); // as $display, values at end of simulation cycle
$monitor(v); // disp. @change of v (except v= $time,$stime,$realtime)
$monitoron; $monitoroff; // toggle monitor mode on/off

end endmodule

```

11.13.2 File I/O Tasks

The following example illustrates the **file I/O system tasks** [Verilog LRM 14.2]:

```

module file_1; integer f1, ch; initial begin f1 = $fopen("f1.out");
if(f1==0) $stop(2); if(f1==2)$display("f1 open");
ch = f1|1; $fdisplay(ch,"Hello"); $fclose(f1); end endmodule

```

```

> vlog file_1.v
> vsim -c file_1
# Loading work.file_1
VSIM 1> run 10
# f1 open
# Hello
VSIM 2> q
> more f1.out
Hello
>

```

The `$fopen` system task returns a 32-bit unsigned integer called a **multichannel descriptor** (`f1` in this example) unique to each file. The multichannel descriptor contains 32 flags, one for each of 32 possible channels or files (subject to limitations of the operating system). Channel 0 is the standard output (normally the screen), which is always open. The first call to `$fopen` opens channel 1 and sets bit 1 of the multichannel descriptor. Subsequent calls set higher bits. The file I/O system tasks: `$fdisplay`, `$fwrite`, `$fmonitor`, and `$fstrobe`; correspond to their display counterparts. The first parameter for the file system tasks is a multichannel descriptor that may have

multiple bits set. Thus, the preceding example writes the string "Hello" to the screen and to file1.out. The task \$fclose closes a file and allows the channel to be reused.

The file I/O tasks \$readmemb and \$readmemh read a text file into a memory. The file may contain only spaces, new lines, tabs, form feeds, comments, addresses, and binary (for \$readmemb) or hex (for \$readmemh) numbers, as in the following example:

```
mem.dat
@2 1010_1111 @4 0101_1111 1010_1111 // @address in hex
xlxl_zzzz 1111_0000 /* x or z is OK */

module load; reg [7:0] mem[0:7]; integer i; initial begin
$readmemb("mem.dat", mem, 1, 6); // start_address=1, end_address=6
for (i= 0; i<8; i=i+1) $display("mem[%0d] %b", i, mem[i]);
end endmodule

> vsim -c load
# Loading work.load
VSIM 1> run 10
# ** Warning: $readmem (memory mem) file mem.dat line 2:
#   More patterns than index range (hex 1:6)
#   Time: 0 ns Iteration: 0 Instance:/
# mem[0] xxxxxxxx
# mem[1] xxxxxxxx
# mem[2] 10101111
# mem[3] xxxxxxxx
# mem[4] 01011111
# mem[5] 10101111
# mem[6] xlxlzzzz
# mem[7] xxxxxxxx
VSIM 2> q
>
```

11.13.3 Timescale, Simulation, and Timing-Check Tasks

There are two **timescale tasks**, \$printtimescale and \$timeformat [Verilog LRM 14.3]. The \$timeformat specifies the %t format specification for the display and file I/O system tasks as well as the time unit for delays entered interactively and from files. Here are examples of the timescale tasks:

```
// timescale tasks:
module a; initial $printtimescale(b.c1); endmodule
module b; c c1 (); endmodule
`timescale 10 ns / 1 fs
module c_dat; endmodule

`timescale 1 ms / 1 ns
module Ttime; initial $timeformat(-9, 5, " ns", 10); endmodule
/* $timeformat [ ( n, p, suffix , min_field_width ) ] ;
```

```

units = 1 second ** (-n), n = 0->15, e.g. for n = 9, units = ns
p = digits after decimal point for %t e.g. p = 5 gives 0.00000
suffix for %t (despite timescale directive)
min_field_width is number of character positions for %t */

```

The **simulation control tasks** are \$stop and \$finish [Verilog LRM 14.4]:

```

module test_simulation_control; // simulation control system tasks:
initial begin $stop; // enter interactive mode (default parameter 1)
$finish(2); // graceful exit with optional parameter as follows:
// 0 = nothing 1 = time and location 2 = time, location, and statistics
end endmodule

```

The **timing-check tasks** [Verilog LRM 14.5] are used in specify blocks. The following code and comments illustrate the definitions and use of timing-check system tasks. The arguments to the tasks are defined and explained in Table 11.11.

TABLE 11.11 Timing-check system task parameters.

Timing task argument	Description of argument	Type of argument
reference_event	to establish reference time	module input or inout (scalar or vector net)
data_event	signal to check against reference_event	module input or inout (scalar or vector net)
limit	time limit to detect timing violation on data_event	constant expression of specparam
threshold	largest pulse width ignored by timing check \$width	constant expression of specparam
notifier	flags a timing violation (before -> after): x->0, 0->1, 1->0, z->z	register

```

module timing_checks (data, clock, clock_1, clock_2); //1
input data, clock, clock_1, clock_2; reg tSU, tH, tHIGH, tP, tSK, tR; //2
specify // timing check system tasks: //3
/* $setup (data_event, reference_event, limit [, notifier]); //4
violation = (T_reference_event)-(T_data_event) < limit */ //5
$setup(data, posedge clock, tSU); //6
/* $hold (reference_event, data_event, limit [, notifier]); //7
violation = //8
    (time_of_data_event)-(time_of_reference_event) < limit */ //9
$hold(posedge clock, data, tH); //10
/* $setuphold (reference_event, data_event, setup_limit, //11
    hold_limit [, notifier]); //12
parameter_restriction = setup_limit + hold_limit > 0 */ //13

```

```

$setuphold(posedge clock, data, tSU, tH); //14
/* $width (reference_event, limit, threshold [, notifier]); //15
violation = //16
    threshold < (T_data_event) - (T_reference_event) < limit //17
reference_event = edge //18
data_event = opposite_edge_of_reference_event */ //19
$width(posedge clock, tHIGH); //20
/* $period (reference_event, limit [, notifier]); //21
violation = (T_data_event) - (T_reference_event) < limit //22
reference_event = edge //23
data_event = same_edge_of_reference_event */ //24
$period(posedge clock, tP); //25
/* $skew (reference_event, data_event, limit [, notifier]); //26
violation = (T_data_event) - (T_reference_event) > limit */ //27
$skew(posedge clock_1, posedged clock_2, tSK); //28
/* $recovery (reference_event, data_event, limit, [, notifier]); //29
violation = (T_data_event) - (T_reference_event) < limit */ //30
$recovery(posedge clock, posedged clock_2, tR); //31
/* $nochange (reference_event, data_event, start_edge_offset, //32
    end_edge_offset [, notifier]); //33
reference_event = posedged | negedge //34
violation = change while reference high (posedged)/low (negedge) //35
+ve start_edge_offset moves start of window later //36
+ve end_edge_offset moves end of window later */ //37
$nochange (posedged clock, data, 0, 0); //38
endspecify endmodule //39

```

You can use **edge specifiers** as parameters for the timing-check events (except for the reference event in \$nochange):

```

edge_control_specifier ::= edge [edge_descriptor {, edge_descriptor}]
edge_descriptor ::= 01 | 0x | 10 | 1x | x0 | x1

```

For example, 'edge [01, 0x, x1] clock' is equivalent to 'posedged clock'. Edge transitions with 'z' are treated the same as transitions with 'x'.

Here is a D flip-flop model that uses timing checks and a **notifier register**. The register, notifier, is changed when a timing-check task detects a violation and the last entry in the table then sets the flip-flop output to unknown.

```

primitive dff_udp(q, clock, data, notifier);
output q; reg q; input clock, data, notifier;
table // clock data notifier:state: q
    r    0    ?    : ? : 0 ;
    r    1    ?    : ? : 1 ;
    n    ?    ?    : ? : - ;
    ?    *    ?    : ? : - ;
    ?    ?    *    : ? : x ; endtable // notifier

```

```

endprimitive

`timescale 100 fs / 1 fs
module dff(q, clock, data); output q; input clock, data; reg notifier;
dff_udp(q1, clock, data, notifier); buf(q, q1);
specify
    specparam tSU = 5, tH = 1, tPW = 20, tPLH = 4:5:6, tPHL = 4:5:6;
    (clock *> q) = (tPLH, tPHL);
    $setup(data, posedge clock, tSU, notifier); // setup: data to clock
    $hold(posedge clock, data, tH, notifier); // hold: clock to data
    $period(posedge clock, tPW, notifier); // clock: period
endspecify
endmodule

```

11.13.4 PLA Tasks

The **PLA modeling tasks** model two-level logic [Verilog LRM 14.6]. As an example, the following eqntott logic equations can be implemented using a PLA:

```
b1 = a1 & a2; b2 = a3 & a4 & a5 ; b3 = a5 & a6 & a7;
```

The following module loads a PLA model for the equations above (in AND logic) using the **array format** (the array format allows only '1' or '0' in the PLA memory, or **personality array**). The file array.dat is similar to the espresso input plane format.

```

array.dat
1100000
0011100
0000111

module pla_1 (a1,a2,a3,a4,a5,a6,a7,b1,b2,b3);
input a1, a2, a3, a4, a5, a6, a7 ; output b1, b2, b3;
reg [1:7] mem[1:3]; reg b1, b2, b3;
initial begin
    $readmemb("array.dat", mem);
    #1; b1=1; b2=1; b3=1;
    $asyn$and$array(mem,{a1,a2,a3,a4,a5,a6,a7},{b1,b2,b3});
end
initial $monitor("%4g", $time, ,b1, ,b2, ,b3);
endmodule

```

The next example illustrates the use of the **plane format**, which allows '1', '0', as well as '?' or 'z' (either may be used for don't care) in the personality array.

```

b1 = a1 & !a2; b2 = a3; b3 = !a1 & !a3; b4 = 1;

module pla_2; reg [1:3] a, mem[1:4]; reg [1:4] b;
initial begin
    $asyn$and$plane(mem,{a[1],a[2],a[3]},{b[1],b[2],b[3],b[4]});
    mem[1] = 3'b10?; mem[2] = 3'b??1; mem[3] = 3'b0?0; mem[4] = 3'b???;

```

```

#10 a = 3'b111; #10 $displayb(a, " -> ", b);
#10 a = 3'b000; #10 $displayb(a, " -> ", b);
#10 a = 3'bxxx; #10 $displayb(a, " -> ", b);
#10 a = 3'b101; #10 $displayb(a, " -> ", b);
end endmodule

111 -> 0101
000 -> 0011
xxx -> xxx1
101 -> 1101

```

11.13.5 Stochastic Analysis Tasks

The **stochastic analysis tasks** model queues [Verilog LRM 14.7]. Each of the tasks return a status as shown in Table 11.12.

TABLE 11.12 Status values for the stochastic analysis tasks.

Status value	Meaning
0	OK
1	queue full, cannot add
2	undefined <code>q_id</code>
3	queue empty, cannot remove
4	unsupported <code>q_type</code> , cannot create queue
5	<code>max_length</code> <= 0, cannot create queue
6	duplicate <code>q_id</code> , cannot create queue
7	not enough memory, cannot create queue

The following module illustrates the interface and parameters for these tasks:

```

module stochastic; initial begin // stochastic analysis system tasks:
/* $q_initialize (q_id, q_type, max_length, status) ;
q_id is an integer that uniquely identifies the queue
q_type 1=FIFO 2=LIFO
max_length is an integer defining the maximum number of entries */
$q_initialize (q_id, q_type, max_length, status) ;

/* $q_add (q_id, job_id, inform_id, status) ;
job_id = integer input
inform_id = user-defined integer input for queue entry */
$q_add (q_id, job_id, inform_id, status) ;

/* $q_remove (q_id, job_id, inform_id, status) ; */
$q_remove (q_id, job_id, inform_id, status) ;

```

```

/* $q_full (q_id, status) ;
status = 0 = queue is not full, status = 1 = queue full */
$q_full (q_id, status) ;

/* $q_exam (q_id, q_stat_code, q_stat_value, status) ;
q_stat_code is input request as follows:
1=current queue length 2=mean inter-arrival time 3=max. queue length
4=shortest wait time ever
5=longest wait time for jobs still in queue 6=ave. wait time in queue
q_stat_value is output containing requested value */
$q_exam (q_id, q_stat_code, q_stat_value, status) ;

end endmodule

```

11.13.6 Simulation Time Functions

The simulation time functions return the time as follows [Verilog LRM 14.8]:

```

module test_time; initial begin // simulation time system functions:
$time ;
// returns 64-bit integer scaled to timescale unit of invoking module

$time ;
// returns 32-bit integer scaled to timescale unit of invoking module

$realtime ;
// returns real scaled to timescale unit of invoking module

end endmodule

```

11.13.7 Conversion Functions

The conversion functions for reals handle real numbers [Verilog LRM 14.9]:

```

module test_convert; // conversion functions for reals:
integer i; real r; reg [63:0] bits;
initial begin #1 r=256;#1 i = $rtoi(r);
#1; r = $itor(2 * i) ; #1 bits = $realtobits(2.0 * r) ;
#1; r = $bitstoreal(bits) ; end
initial $monitor("%3f", $time,,i,,r,,bits); /*
$rtoi converts reals to integers w/truncation e.g. 123.45 -> 123
$itor converts integers to reals e.g. 123 -> 123.0
$realtobits converts reals to 64-bit vector
$bitstoreal converts bit pattern to real
Real numbers in these functions conform to IEEE Std 754. Conversion
rounds to the nearest valid number. */
endmodule

# 0.000000          x 0          x
# 1.000000          x 256        x
# 2.000000          256 256      x
# 3.000000          256 512      x

```

```
# 4.000000      256 512 4652218415073722368
# 5.000000      256 1024 4652218415073722368
```

Here is an example using the conversion functions in port connections:

```
module test_real; wire [63:0]a; driver d (a); receiver r (a);
initial $monitor("%3g", $time, , a, , d.r1, , r.r2); endmodule

module driver (real_net);
output real_net; real r1; wire [64:1] real_net = $realtobits(r1);
initial #1 r1 = 123.456; endmodule

module receiver (real_net);
input real_net; wire [64:1] real_net; real r2;
initial assign r2 = $bitstoreal(real_net);
endmodule

# 0      0 0 0
# 1 4638387860618067575 123.456 123.456
```

11.13.8 Probability Distribution Functions

The probability distribution functions are as follows [Verilog LRM 14.10]:

```
module probability; // probability distribution functions: //1
/* $random [(seed)] returns random 32-bit signed integer //2
seed = register, integer, or time */ //3
reg [23:0] r1,r2; integer r3,r4,r5,r6,r7,r8,r9; //4
integer seed, start, \end , mean, standard_deviation; //5
integer degree_of_freedom, k_stage; //6
initial begin seed=1; start=0; \end =6; mean=5; //7
standard_deviation=2; degree_of_freedom=2; k_stage=1; #1; //8
r1 = $random % 60; // random -59 to 59 //9
r2 = {$random} % 60; // positive value 0-59 //10
r3=$dist_uniform (seed, start, \end ) ; //11
r4=$dist_normal (seed, mean, standard_deviation) ; //12
r5=$dist_exponential (seed, mean) ; //13
r6=$dist_poisson (seed, mean) ; //14
r7=$dist_chi_square (seed, degree_of_freedom) ; //15
r8=$dist_t (seed, degree_of_freedom) ; //16
r9=$dist_erlang (seed, k_stage, mean) ; end //17
initial #2 $display ("%3f", $time, , r1, , r2, , r3, , r4, , r5); //18
initial begin #3; $display ("%3f", $time, , r6, , r7, , r8, , r9); end //19
/* All parameters are integer values. //20
Each function returns a pseudo-random number //21
e.g. $dist_uniform returns uniformly distributed random numbers //22
mean, degree_of_freedom, k_stage //23
(exponential, poisson, chi-square, t, erlang) > 0. //24
seed = inout integer initialized by user, updated by function //25
```

```

start, end ($dist_uniform) = integer bounding return values */ //26
endmodule //27
2.000000      8      57      0      4      9
3.000000      7      3      0      2

```

11.13.9 Programming Language Interface

The C language **Programming Language Interface (PLI)** allows you to access the internal Verilog data structure [Verilog LRM17–23, A–E]. For example, you can use the PLI to implement the following extensions to a Verilog simulator:

- C language delay calculator for a cell library
- C language interface to a Verilog-based or other logic or fault simulator
- Graphical waveform display and debugging
- C language simulation models
- Hardware interfaces

There are three generations of PLI routines (see Appendix B for an example):

- Task/function (TF) routines (or utility routines), the first generation of the PLI, start with 'tf_'.
- Access (ACC) routines, the second generation of the PLI, start with the characters 'acc_' and access delay and logic values. There is some overlap between the ACC routines and TF routines.
- Verilog Procedural Interface (VPI) routines, the third generation of the PLI, start with the characters 'vpi_' and are a superset of the TF and ACC routines.

11.14 Summary

Table 11.13 lists the key features of Verilog HDL. The most important concepts covered in this chapter are:

- Concurrent processes and sequential execution
- Difference between a reg and a wire, and between a scalar and a vector
- Arithmetic operations on reg and wire
- Data slip
- Delays and events

TABLE 11.13 Verilog on one page.

Verilog feature	Example
Comments	<code>a = 0; // comment ends with newline /* This is a multiline or block comment */</code>
Constants: string and numeric	<code>parameter BW = 32 // local, use BW `define G_BUS 32 // global, use `G_BUS 4'b2 1'bx</code>
Names (case-sensitive, start with letter or '_')	<code>_12name A_name \$BAD NotSame notsame</code>
Two basic types of logic signals: wire and reg	<code>wire myWire; reg myReg;</code>
Use a continuous assignment statement with wire	<code>assign myWire = 1;</code>
Use a procedural assignment statement with reg	<code>always myReg = myWire;</code>
Buses and vectors use square brackets	<code>reg [31:0] DBus; DBus[12] = 1'bx;</code>
We can perform arithmetic on bit vectors	<code>reg [31:0] DBus; DBus = DBus + 2;</code>
Arithmetic is performed modulo 2^n	<code>reg [2:0] R; R = 7 + 1; // now R = 0</code>
Operators: as in C (but not ++ or --)	
Fixed logic-value system	1, 0, x (unknown), z (high-impedance)
Basic unit of code is the module	<code>module bake (chips, dough, cookies); input chips, dough; output cookies; assign cookies = chips & dough; endmodule</code>
Ports	input or input/output ports are wire output ports are wire or reg
Procedures model things that happen at the same time and may be sensitive to an edge, <code>posedge</code> , <code>negedge</code> , or to a level.	<code>always @rain sing; always @rain dance; always @(posedge clock) D = Q; // flop always @(a or b) c = a & b; // and gate</code>
Sequential blocks model repeating things: <code>always</code> : executes forever <code>initial</code> : executes once only at start of simulation	<code>initial born; always @alarm_clock begin : a_day metro=commute; boulot=work; dodo=sleep; end</code>
Functions and tasks	<code>function ... endfunction task ... endtask</code>
Output	<code>\$display("a=%f", a); \$dumpvars; \$monitor(a)</code>
Control simulation	<code>\$stop; \$finish // sudden or gentle halt</code>
Compiler directives	<code>`timescale 1ns/1ps // units/resolution</code>
Delay	<code>#1 a = b; // delay then sample b a = #1 b; // sample b then delay</code>

11.15 Problems

*=Difficult, **=Very difficult, ***=Extremely difficult

11.1 (Counter, 30 min.) Download the VeriWell simulator from <http://www.wellspring.com> and simulate the counter from Section 11.1 (exclude the comments to save typing). Include the complete input and output listings in your report.

11.2 (Simulator, 30 min.) Build a “cheat sheet” for your simulator, listing the commands for running the simulator and using it in interactive mode.

11.3 (Verilog examples, 10 min.) The Cadence Verilog-XL simulator comes with a directory `examples`. Make a list of the examples from the `README` files in the various directories.

11.4 (Gotchas, 60 min.) Build a “most common Verilog mistakes” file. Start with:

- Extra or missing semicolon ';'
- Forgetting to declare a `reg`
- Using a `reg` instead of a `wire` for an input or inout port
- Bad declarations: `reg bus[0:31]` instead of `reg [31:0]bus`
- Mixing vector declarations: `wire [31:0]BusA, [15:0]BusB`
- The case-sensitivity of Verilog
- No delay in an `always` statement (simulator loops forever)
- Mixing up ``` (accent grave) for ``define` and `'` (tick or apostrophe) for `1'b1` with `´` (accent acute) or `'` (open single quote) or `'` (close single quote)
- Mixing `"` (double quote) with `"` (open quotes) or `"` (close quotes)

11.5 (Sensitivity, 10 min.) Explore and explain what happens if you write this:

```
always @(a or b or c) e = (a|b)&(c|d);
```

11.6 (Verilog `if` statement, 10 min.) Build test code to simulate the following Verilog fragment. Explain what is wrong and fix the problem.

```
if (i > 0)
    if (i < 2) $display ("i is 1");
else $display ("i is less than 0");
```

11.7 (Effect of delay, 30 min.). Write code to test the four different code fragments shown in Table 11.14 and print the value of 'a' at time = 0 and time = 1 for each case. Explain the differences in your simulation results.

11.8 (Verilog events, 10 min.). Simulate the following and explain the results:

```
event event_1, event_2;
always @ event_1 -> event_2;
initial @event_2 $stop;
initial -> event_1;
```

TABLE 11.14 Code fragments for Problem 11.7.

	(a)	(b)	(c)	(d)
Code fragment	<pre>reg a; initial begin a = 0; a = a + 1; end</pre>	<pre>reg a; initial begin #0 a = 0; #0 a = a + 1; end</pre>	<pre>reg a; initial begin a <= 0; a <= a + 1; end</pre>	<pre>reg a; initial begin #1 a = 0; #1 a = a + 1; end</pre>

11.9 (Blocking and nonblocking assignment statements, 30 min.). Write code to test the different code fragments shown in Table 11.15 and print the value of 'outp' at time = 0 and time = 10 for each case. Explain the difference in simulation results.

TABLE 11.15 Code fragments for Problem 11.9.

	(a)	(b)	(c)	(d)
Code fragment	<pre>reg outp; always begin #10 outp = 0; #10 outp = 1; end</pre>	<pre>reg outp; always begin outp <= #10 1; outp <= #10 0; end</pre>	<pre>reg outp; always begin #10 outp = 0; #10 outp <= 1; end</pre>	<pre>reg outp; always begin #10 outp <= 0; #10 outp = 1; end</pre>

11.10 (Verilog UDPs, 20 min.). Use this primitive to build a half adder:

```
primitive Adder(Sum, InA, InB); output Sum; input InA, InB;
table 00 : 0; 01 : 1; 10 : 1; 11 : 0; endtable
endprimitive
```

Apply unknowns to the inputs. What is the output?

11.11 (Verilog UDPs, 30 min.). Use the following primitive model for a D latch:

```
primitive DLatch(Q, Clock, Data); output Q; reg Q; input Clock, Data;
table 1 0 : ? : 0; 1 1 : ? : 1; 0 1 : ? : -; endtable
endprimitive
```

Check to see what happens when you apply unknown inputs (including clock transitions to unknown). What happens if you apply high-impedance values to the inputs (again including transitions)?

11.12 (Propagation of unknowns in primitives, 45 min.) Use the following primitive model for a D flip-flop:

```
primitive DFF(Q, Clock, Data); output Q; reg Q; input Clock, Data;
```

```

table
r   0 : ? : 0 ;
r   1 : ? : 1 ;
(0x) 0 : 0 : 0 ;
(0x) 1 : 1 : 1 ;
(?0) ? : ? : - ;
? (??) : ? : - ;
endtable
endprimitive

```

Check to see what happens when you apply unknown inputs (including a clock transition to an unknown value). What happens if you apply high-impedance values to the inputs (again including transitions)?

11.13 (D flip-flop UDP, 60 min.) Table 11.16 shows a UDP for a D flip-flop with QN output and asynchronous reset and set.

TABLE 11.16 D flip-flop UDP for Problem 11.13.

```

primitive DFlipFlop2(QN, Data, Clock, Res, Set);
output QN; reg QN; input Data, Clock, Res, Set;
table
//      Data   Clock  Res   Set   :state :next state
      1      (01)   0     0     :?     :0;     // line 1
      1      (01)   0     x     :?     :0;
      ?      ?     0     x     :0     :0;
      0      (01)   0     0     :?     :1;
      0      (01)   x     0     :?     :1;
      ?      ?     x     0     :1     :1;
      1      (x1)   0     0     :0     :0;
      0      (x1)   0     0     :1     :1;
      1      (0x)   0     0     :0     :0;
      0      (0x)   0     0     :1     :1;
      ?      ?     1     ?     :?     :1;
      ?      ?     0     1     :?     :0;
      ?      n     0     0     :?     :-;
      *      ?     ?     ?     :?     :-;
      ?      ?     (?0)  ?     :?     :-;
      ?      ?     ?     (?0)  :?     :-;
      ?      ?     ?     ?     :?     :x;     // line 17
endtable
endprimitive

```

- Explain the purpose of each line in the truth table.
- Write a module to test each line of the UDP.
- Can you find any errors, omissions, or other problems in this UDP?

11.14 (JK flip-flop, 30 min.) Test the following model for a JK flip-flop:

```

module JKFF (Q, J, K, Clk, Rst);
parameter width = 1, reset_value = 0;
input [width-1:0] J, K; output [width-1:0] Q; reg [width-1:0] Q;
input Clk, Rst; initial Q = {width{1'bx}};
always @ (posedge Clk or negedge Rst )
if (Rst==0 ) Q <= #1 reset_value;
else Q <= #1 (J & ~K) | (J & K & ~Q) | (~J & ~K & Q);
endmodule

```

11.15 (Overriding Verilog parameters, 20 min.) The following module has a parameter specification that allows you to change the number of AND gates that it models (the cardinality or width):

```

module Vector_AND(Z, A, B);
parameter card = 2; input [card-1:0] A,B; output [card-1:0] Z;
wire [card-1:0] Z = A & B;
endmodule

```

The next module changes the parameter value by specifying an overriding value in the module instantiation:

```

module Four_AND_Gates(OutBus, InBusA, InBusB);
input [3:0] InBusA, InBusB; output [3:0] OutBus;
Vector_AND #(4) My_AND(OutBus, InBusA, InBusB);
endmodule

```

These next two modules change the parameter value by using a defparam statement, which overrides the declared parameter value:

```

module X_AND_Gates(OutBus, InBusA, InBusB);
parameter X = 2;input [X-1:0] InBusA, InBusB;output [X-1:0] OutBus;
Vector_AND #(X) My_AND(OutBus, InBusA, InBusB);
endmodule

module size; defparam X_AND_Gates.X = 4; endmodule

```

- a. Check that the two alternative methods of specifying parameters are equivalent by instantiating the modules `Four_AND_Gates` and `X_AND_Gates` in another module and simulating.
- b. List and comment on the advantages and disadvantages of both methods.

11.16 (Default Verilog delays, 10 min.). Demonstrate, using simulation, that the following NAND gates have the delays you expect:

```

nand (strong0, strong1) #1
Nand_1(n001, n004, n005),
Nand_2(n003, n001, n005, n002);
nand (n006, n005, n002);

```

11.17 (Arrays of modules, 30 min.) Newer versions of Verilog allow the instantiating of **arrays of modules** (in this book we usually call this a vector since we are

only allowed one row). You specify the number in the array by using a **range** after the instance name as follows:

```
nand #2 nand_array[0:7](zn, a, b);
```

Create and test a model for an 8-bit register using an array of flip-flops.

11.18 (Assigning Verilog real to integer data types, 10 min.) What is the value of `ImInteger` in the following code?

```
real ImReal; integer ImInteger;
initial begin ImReal = -1.5; ImInteger = ImReal; end
```

11.19 (BNF syntax, 10 min.) Use the BNF syntax definitions in Appendix B to answer the following questions. In each case explain how you arrive at the answer:

- What is the highest-level construct?
- What is the lowest-level construct?
- Can you nest `begin` and `end` statements?
- Where is a legal place for a `case` statement?
- Is the following code legal: `reg [31:0] rega, [32:1] regb;`
- Where is it legal to include sequential statements?

11.20 (Old syntax definitions, 10 min.) Prior to the IEEE LRM, Verilog BNF was expressed using a different notation. For example, an event expression was defined as follows:

```
<event_expression> ::= <expression>
  or <<posedge or negedge> <SCALAR_EVENT_EXPRESSION>>
  or <<event_expression> or <event_expression>>
```

Notice that we are using `'or'` as part of the BNF to mean “alternatively” and also `'or'` as a Verilog keyword. The keyword `'or'` is in bold—the difference is fairly obvious. Here is an alternative definition for an event expression:

```
<event_expression> ::= <expression>
||= posedge <SCALAR_EVENT_EXPRESSION>
||= negedge <SCALAR_EVENT_EXPRESSION>
||= <event_expression> <or <event_expression>>*
```

Are these definitions equivalent (given, of course, that we replaced `||=` with `or` in the simplified syntax)? Explain carefully how you would attempt to prove that they are the same.

11.21 (Operators, 20 min.) Explain Table 11.17 (see next page).

11.22 (Unary reduction, 10 min.) Complete Table 11.18 (see next page).

11.23 (Coerced ports, 20 min.) Perform some experiments to test the behavior of your Verilog simulator in the following situation: “NOTE—A port that is declared as input (output) but used as an output (input) or inout may be coerced to inout. If not coerced to inout, a warning must be issued” [Verilog LRM 12.3.6].

TABLE 11.17 Unary operators (Problem 11.21).

	(a)	(b)	(c)	(d)
Code	<pre>module unary; reg [4:0] u; initial u=! 'b011z; initial \$display("%b",u); endmodule</pre>	<pre>module unary; wire u; assign u=! 'b011z; initial \$display("%b",u); endmodule</pre>	<pre>module unary; wire u; assign u=! 'b011z; initial #1 \$display("%b",u); endmodule</pre>	<pre>module unary; wire u; assign u=&'b1; initial #1 \$display("%b",u); endmodule</pre>
Output	0000x	z	x	0

TABLE 11.18 Unary reduction (Problem 11.22).

Operand	&	~&		~	^	~^
4'b0000						
4'b1111						
4'b01x0						
4'bz000						

11.24 (*Difficult delay code, 20 min.) Perform some experiments to explain what this difficult to interpret statement does:

```
#2 a <= repeat(2) @(posedge clk) d;
```

11.25 (Fork-join, 20 min.) Write some test code to compare the behavior of the code fragments shown in Table 11.19.

TABLE 11.19 Fork-and-join examples for Problem 11.25.

	(a)	(b)	(c)	(d)
Code fragment	<pre>fork a = b; b = a; join</pre>	<pre>fork a <= b; b <= a; join</pre>	<pre>fork #1 a = b; #1 b = a; join</pre>	<pre>fork a = #1 b; b = #1 a; join</pre>

11.26 (Blocking and nonblocking assignments, 20 min.) Simulate the following code and explain the results:

```
module nonblocking; reg Y;
  always begin Y <= #10 1; Y <= #20 0; #10; end
  always begin $display($time, "Y=", Y); #10; end
```

```

initial #100 $finish;
endmodule

```

11.27 (*Flip-flop code, 10 min.) Explain why this flip-flop does not work:

```

module Dff_Res_Bad(D,Q,Clock,Reset);
output Q; input D,Clock,Reset; reg Q; wire D;
always @(posedge Clock) if (Reset != 1) Q = D; always if (Reset == 1)
Q = 0;
end endmodule

```

11.28 (D flip-flop, 10 min.) Test the following D flip-flop model:

```

module DFF (D, Q, Clk, Rst);
parameter width = 1, reset_value = 0;
input [width-1:0] D; output [width-1:0] Q; reg [width-1:0] Q;
input Clk,Rst;
initial Q = {width{1'bx}};
always @ ( posedge Clk or negedge Rst )
if ( Rst == 0 ) Q <= #1 reset_value; else Q <= #1 D;
endmodule

```

11.29 (D flip-flop with scan, 10 min.) Explain the following model:

```

module DFFSCAN (D, Q, Clk, Rst, ScEn, ScIn, ScOut);
parameter width = 1, reset_value = 0;
input [width-1:0] D; output [width-1:0] Q; reg [width-1:0] Q;
input Clk,Rst,ScEn,ScIn; output ScOut;
initial Q = {width{1'bx}};
always @ ( posedge Clk or negedge Rst ) begin
if ( Rst == 0 ) Q <= #1 reset_value;
else if (ScEn) Q <= #1 {Q,ScIn};
else Q <= #1 D;
end
assign ScOut=Q[width-1];
endmodule

```

11.30 (Pads, 30 min.) Test the following model for a bidirectional I/O pad:

```

module PadBidir (C, Pad, I, Oen); // active low enable
parameter width=1, pinNumbers="", \strength =1, level="CMOS",
pull="none", externalVdd=5;
output [width-1:0] C; inout [width-1:0] Pad; input [width-1:0] I;
input Oen;
assign #1 Pad = Oen ? {width{1'bz}} : I;
assign #1 C = Pad;
endmodule

```

Construct and test a model for a three-state pad from the above.

11.31 (Loops, 15 min.) Explain and correct the problem in the following code:

```

module Loop_Bad; reg [3:0] i; reg [31:0] DBus;
initial DBus = 0;

```



```

initial begin #1; for (i=0; i<=15; i=i+1) DBus[i]=1; end
initial begin
$display("DBus = %b",DBus); #2; $display("DBus = %b",DBus); $stop;
end endmodule

```

11.32 (Arithmetic, 10 min.) Explain the following:

```

integer IntA;
IntA = -12 / 3; // result is -4
IntA = -'d 12 / 3; // result is 1431655761

```

Determine and explain the values of `intA` and `regA` after each assignment statement in the following code:

```

integer intA; reg [15:0] regA;
intA = -4'd12; regA = intA/3; regA = -4'd12;
intA = regA/3; intA = -4'd12/3; regA = -12/3;

```

11.33 (Arithmetic overflow, 30 min.) Consider the following:

```

reg [7:0] a, b, sum; sum = (a + b) >> 1;

```

The intent is to add `a` and `b`, which may cause an overflow, and then shift `sum` to keep the carry bit. However, because all operands in the expression are of an 8-bit width, the expression `(a + b)` is only 8 bits wide, and we lose the carry bit before the shift. One solution forces the expression `(a + b)` to use at least 9 bits. For example, adding an integer value of 0 to the expression will cause the evaluation to be performed using the bit size of integers [LRM 4.4.2]. Check to see if the following alternatives produce the intended result:

```

sum = (a + b + 0) >> 1;
sum = {0,a} + {0,b} >> 1;

```

11.34 (*Data slip, 60 min.) Table 11.20 shows several different ways to model the connection of a 2-bit shift register. Determine which of these models suffer from data slip. In each case show your simulation results.

11.35 (**Timing, 30 min.) What does a simulator display for the following?

```

assign p = q; initial begin q = 0; #1 q = 1; $display(p); end

```

What is the problem here? Conduct some experiments to illustrate your answer.

11.36 (Port connections, 10 min.) Explain the following declaration:

```

module test (.a(c), .b(c));

```

11.37 (**Functions and tasks, 30 min.) Experiment to determine whether invocation of a function (or task) behaves as a blocking or nonblocking assignment.

11.38 (Nonblocking assignments, 10 min.) Predict the output of the following model:

```

module e1; reg a, b, c;
initial begin a = 0; b = 1; c = 0; end
always c = #5 ~c; always @(posedge c) begin a <= b; b <= a; end
endmodule

```

TABLE 11.20 Data slip (Problem 11.34).

Alternative	Data slip?
1 <code>always @(posedge Clk) begin Q2 = Q1; Q1 = D1; end</code>	
2 <code>always @(posedge Clk) begin Q1 = D1; Q2 = Q1; end</code>	
3 <code>always @(posedge Clk) begin Q1 <= #1 D1; Q2 <= #1 Q1; end</code>	
4 <code>always @(posedge Clk) Q1 = D1; always @(posedge Clk) Q2 = Q1;</code>	Y
5 <code>always @(posedge Clk) Q1 = #1 D1; always @(posedge Clk) Q2 = #1 Q1;</code>	N
6 <code>always @(posedge Clk) #1 Q1 = D1; always @(posedge Clk) #1 Q2 = Q1;</code>	
7 <code>always @(posedge Clk) Q1 <= D1; always @(posedge Clk) Q2 <= Q1;</code>	
8 <code>module FF_1 (Clk, D1, Q1); always @(posedge Clk) Q1 = D1; endmodule module FF_2 (Clk, Q1, Q2); always @(posedge Clk) Q2 = Q1; endmodule</code>	
9 <code>module FF_1 (Clk, D1, Q1); always @(posedge Clk) Q1 <= D1; endmodule module FF_2 (Clk, Q1, Q2); always @(posedge Clk) Q2 <= Q1; endmodule</code>	

11.39 (Assignment timing, 20 min.) Predict the output of the following module and explain the timing of the assignments:

```
module e2; reg a, b, c, d, e, f;
initial begin a = #10 1; b = #2 0; c = #4 1; end
initial begin d <= #10 1; e <= #2 0; f <= #4 1; end
endmodule
```

11.40 (Swap, 10 min.) Explain carefully what happens in the following code:

```
module e3; reg a, b;
initial begin a = 0; b = 1; a <= b; b <= a; end
endmodule
```

11.41 (*Overwriting, 30 min.) Explain the problem in the following code, determine what happens, and conduct some experiments to explore the problem further:

```
module m1; reg a;
initial a = 1;
initial begin a <= #4 0; a <= #4 1; end
endmodule
```

11.42 (*Multiple assignments, 30 min.) Explain what happens in the following:

```
module m2; reg r1; reg [2:0] i;
initial begin
r1 = 0; for (i = 0; i <= 5; i = i+1) r1 <= # (i*10) i[0]; end
endmodule
```

11.43 (Timing, 30min) Write a model to mimic the behavior of a traffic light signal. The clock input is 1 MHz. You are to drive the lights as follows (times that the lights are on are shown in parentheses): green (60s), yellow (1s), red (60s).

11.44 (Port declarations, 30 min.) The rules for port declarations are as follows: “The port expression in the port definition can be one of the following:

- a simple identifier
- a bit-select of a vector declared within the module
- a part-select of a vector declared within the module
- a concatenation of any of the above

Each port listed in the module definition’s list of ports shall be declared in the body of the module as an input, output, or inout (bidirectional). This is in addition to any other declaration for a particular port—for example, a reg, or wire. A port can be declared in both a port declaration and a net or register declaration. If a port is declared as a vector, the range specification between the two declarations of a port shall be identical” [Verilog LRM 12.3.2].

Compile the following and comment (you may be surprised at the results):

```
module stop (); initial #1 $finish; endmodule
module Outs_1 (a); output [3:0] a; reg [3:0] a;
initial a <= 4'b10xz; endmodule
module Outs_2 (a); output [2:0] a; reg [3:0] a;
initial a <= 4'b10xz; endmodule
module Outs_3 (a); output [3:0] a; reg [2:0] a;
initial a <= 4'b10xz; endmodule
module Outs_4 (a); output [2:0] a; reg [2:0] a;
initial a <= 4'b10xz; endmodule
module Outs_5 (a); output a; reg [3:0] a;
initial a <= 4'b10xz; endmodule
module Outs_6 (a[2:0]); output [3:0] a; reg [3:0] a;
initial a <= 4'b10xz; endmodule
module Outs_7 (a[1]); output [3:0] a; reg [3:0] a;
initial a <= 4'b10xz; endmodule
module Outs_8 (a[1]); output a; reg [3:0] a;
always a <= 4'b10xz; endmodule
```

11.45 (Specify blocks, 30 min.)

- a. Describe the pin-to-pin timing of the following module. Build a testbench to demonstrate your explanation.

```
module XOR_spec (a, b, z); input a, b; output z; xor x1 (z, a, b);
specify
  specparam tnr = 1, tnf = 2 specparam tir = 3, tif = 4;
  if ( a)(b => z) = (tir, tif); if ( b)(a => z) = (tir, tif);
  if (~a)(b => z) = (tnr, tnf); if (~b)(a => z) = (tnr, tnf);
endspecify
endmodule
```

- b. Write and test a module for a 2:1 MUX with inputs A0, A1, and sel; output z; and the following delays: A0 to z: 0.3ns (rise) and 0.4ns (fall); A1 to z: 0.2ns (rise) and 0.3 ns (fall); sel to z=0.5 ns.

11.46 (Design contest, **60 min.) In 1995 John Cooley organized a contest between VHDL and Verilog for ASIC designers. The goal was to design the fastest 9-bit counter in under one hour using Synopsys synthesis tools and an LSI Logic vendor technology library. The Verilog interface is as follows:

```
module counter (data_in, up, down, clock,
  count_out, carry_out, borrow_out, parity_out);
  output [8:0] count_out;
  output carry_out, borrow_out, parity_out;
  input [8:0] data_in; input clock, up, down;
  reg [8:0] count_out; reg carry_out, borrow_out, parity_out;
  // Insert your design here.
endmodule
```

The counter is positive-edge triggered, counts up with up='1' and down with down='1'. The contestants had the advantage of a predefined testbench with a set of test vectors; you do not. Design a model for the counter and a testbench.

11.47 (Timing checks, ***60 min.+) Flip-flops with preset and clear require more complex timing-check constructs than those described in Section 11.13.3. The following BNF defines a **controlled timing-check event**:

```
controlled_timing_check_event ::= timing_check_event_control
  specify_terminal_descriptor [ &&& timing_check_condition ]

timing_check_condition ::=
  scalar_expression | ~scalar_expression
| scalar_expression == scalar_constant
| scalar_expression === scalar_constant
| scalar_expression != scalar_constant
| scalar_expression !== scalar_constant
```

The scalar expression that forms the conditioning signal must be a scalar net, or else the least significant bit of a vector net or a multibit expression value is used. The comparisons in the timing check condition may be **deterministic** (using ===, !==, ~, or no operator) or **nondeterministic** (using == or !=). For deterministic comparisons, an 'x' result disables the timing check. For nondeterministic comparisons, an 'x' result enables the timing check.

As an example the following **unconditioned timing check**,

```
$setup(data, posedge clock, 10);
```

performs a setup timing check on every positive edge of clock, as was explained in Section 11.13.3. The following controlled timing check is enabled only when clear is high, which is what is required in a flip-flop model, for example.

```
$setup(data, posedge clock &&& clear, 10);
```

The next example shows two alternative ways to enable a timing check only when clear is low. The second method uses a nondeterministic operator.

```
$setup(data, posedge clock &&& (~clear), 10); // clear=x disables check
$setup(data, posedge clock &&& (clear==0), 10); // clear=x enables check
```

To perform the setup check only when `clear` and `preset` signals are high, you can add a gate outside the `specify` block, as follows:

```
and g1(clear_and_preset, clear, set);
```

A controlled timing check event can then use this `clear_and_preset` signal:

```
$setup(data, posedge clock &&& clear_and_preset, 10);
```

Use the preceding techniques to expand the D flip-flop model, `dff_udp`, from Section 11.13.3 to include asynchronous active-low preset and clear signals as well as an output, `qbar`. Use the following module interface:

```
module dff(q, qbar, clock, data, preset, clear);
```

11.48 (Verilog BNF, 30 min.) Here is the “old” BNF definition of a sequential block (used in the Verilog reference manuals and the OVI LRM). Are there any differences from the “new” version?

```
<sequential_block> ::=
    begin <statement>* end
    or
    begin: <block_IDENTIFIER> <block_declaration>*
        <statement>*
    end

<block_declaration> ::= parameter <list_of_param_assignment>;
    or reg <range>? <attribute_decl>*
        <list_of_register_variable>;
    or integer <attribute_decl>* <list_of_register_variable>;
    or real <attribute_decl>* <list_of_variable_IDENTIFIER>;
    or time <attribute_decl>* <list_of_register_variable>;
    or event <attribute_decl>* <list_of_event_IDENTIFIER>;

<statement> ::=
    <blocking_assignment>;
    or <non-blocking_assignment>;
    or if(<expression>) <statement_or_null>
        <else <statement_or_null> >?
    or <case or casez or casex>
        (<expression>) <case item>+ endcase
    or forever <statement>
    or repeat(<expression>) <statement>
    or while(<expression>) <statement>
    or for(<assignment>;
        <expression>; <assignment>) <statement>
    or wait(<expression>) <statement_or_null>
    or disable <task_IDENTIFIER>;
    or disable <block_IDENTIFIER>;
    or force <assignment>; or release <value>;
    or <timing_control> <statement_or_null>
    or -> <event_IDENTIFIER>;
```

or <sequential_block> or <parallel_block>
 or <task_enable> or <system_task_enable>

11.49 (Conditional compiler directives, 30 min.) The conditional compiler directives: ``define`, ``ifdef`, ``else`, ``endif`, and ``undef`; work much as in C. Write and compile a module that models an AND gate as `'z = a&b'` if the variable `behavioral` is defined. If `behavioral` is not defined, then model the AND gate as `'and a1 (z, a, b)'`.

11.50 (*Macros, 30 min.) According to the IEEE Verilog LRM [16.3.1] you can create a **macro** with parameters using ``define`, as the following example illustrates. This is a particularly difficult area of compliance. Does your software allow the following? You may have to experiment considerably to get this to work. *Hint*: Check to see if your software is substituting for the macro text literally or if it does in fact substitute for parameters.

```
`define M_MAX(a, b)((a) > (b) ? (a) : (b))
`define M_ADD(a,b) (a+b)
module macro;
reg m1, m2, m3, s0, s1;
`define var_nand(delay) nand #delay
`var_nand (2) g121 (q21, n10, n11);
`var_nand (3) g122 (q22, n10, n11);
initial begin s0=0; s1=1;
m1 = `M_MAX (s0, s1); m2 = `M_ADD (s0,s1); m3 = s0 > s1 ? s0 : s1;
end
initial #1 $display(" m1=",m1," m2=",m2," m3=",m3);
endmodule
```

11.51 (**Verilog hazards, 30 min.) The MTI simulator, VSIM, is capable of detecting the following kinds of Verilog hazards:

1. WRITE/WRITE: Two processes writing to the same variable at the same time.
2. READ/WRITE: One process reading a variable at the same time it is being written to by another process. VSIM calls this a READ/WRITE hazard if it executed the read first.
3. WRITE/READ: Same as a READ/WRITE hazard except that VSIM executed the write first.

For example, the following log shows how to simulate Verilog code in hazard mode for the example in Section 11.6.2:

```
> vlib work
> vlog -hazards data_slip_1.v
> vsim -c -hazards data_slip_1
...(lines omitted)...
# 100 0 1 1 x
# ** Error: Write/Read hazard detected on Q1 (ALWAYS 3 followed by
ALWAYS 4)
# Time: 150 ns Iteration: 1 Instance:/
```

```
# 150 1 1 1 1
...(lines omitted)...
```

There are a total of five hazards in the module `data_slip_1`, four are on `Q1`, but there is another. If you correct the code as suggested in Section 11.6.2 and run VSIM, you will find this fifth hazard. If you do not have access to MTI's simulator, can you spot this additional read/write hazard? *Hint*: It occurs at time zero on `clk`. Explain.

11.15.1 The Viterbi Decoder

11.52 (Understanding, 20 min.) Calculate the values shown in Table 11.8 if we use 4 bits for the distance measures instead of 3.

11.53 (Testbenches)

- a. (30 min.) Write a testbench for the encoder, `viterbi_encode`, in Section 11.12 and reproduce the results of Table 11.7.
- b. (30 min.) Write a testbench for the receiver front-end `viterbi_distances` and reproduce the results of Table 11.9 (you can write this stand-alone or use the answer to part a to generate the input). *Hint*: You will need a model for a D flip-flop. The sequence of results is more important than the exact timing. If you do have timing differences, explain them carefully.

11.54 (Things go wrong, 60 min.) Things do not always go as smoothly as the examples in this book might indicate. Suppose you accidentally invert the sense of the reset for the D flip-flops in the encoder. Simulate the output of the faulty encoder with an input sequence $X_n = 0, 1, 2, 3, \dots$ (in other words run the encoder with the flip-flops being reset continually). The output sequence looks reasonable (you should find that it is $Y_n = 0, 2, 4, 6, \dots$). Explain this result using the state diagram of Figure 11.3. If you had constructed a testbench for the entire decoder and did not check the intermediate signals against expected values you would probably never find this error.

11.55 (Subset decoder) Table 11.21 shows the inputs and outputs from the first-stage of the Viterbi decoder, the subset decoder. Calculate the expected output and then confirm your predictions using simulation.

TABLE 11.21 Subset decoder (Problem 11.55).

input	in0	in1	in2	in3	in4	in5	in6	in7	s0	s1	s2	s3	sout0	sout1	sout2	sout3
5	6	7	6	4	1	0	1	4	1	0	1	4				
4	7	6	4	1	0	1	4	6	0	1	4	1				
1	1	0	1	4	6	7	6	4	1	0	1	4				
0	0	1	4	6	7	6	4	1	0	1	4	1				

11.16 Bibliography

The IEEE Verilog LRM [1995] is less intimidating than the IEEE VHDL LRM, because it is based on the OVI LRM, which in turn was based on the Verilog-XL simulator reference manual. Thus it has more of a “User’s Guide” flavor and is required reading for serious Verilog users. It is the only source for detailed information on the PLI.

Phil Moorby was one of the original architects of the Verilog language. The Thomas and Moorby text is a good introduction to Verilog [1991]. The code examples from this book can be obtained from the World Wide Web. Palnitkar’s book includes an example of the use of the PLI routines [1996].

Open Verilog International (OVI) has a Web site maintained by Chronologic (<http://www.chronologic.com/ovi>) with membership information and addresses and an ftp site maintained by META-Software (<ftp://ftp.metasw.com> in `/pub/OVI/`). OVI sells reference material, including proceedings from the International Verilog HDL Conference.

The newsgroup `comp.lang.verilog` (with a FAQ—frequently asked questions) is accessible from a number of online sources. The FAQ includes a list of reference materials and book reviews. Cray Research maintained an archive for `comp.lang.verilog` going back to 1993 but this was lost in January 1997 and is still currently unavailable. Cadence has a discussion group at `talkverilog@cadence.com`. Wellspring Solutions offers VeriWell, a no-cost, limited capability, Verilog simulator for UNIX, PC, and Macintosh platforms.

There is a free, “copylefted” Verilog simulator, `vbs`, written by Jimen Ching and Lay Hoon Tho as part of their Master’s theses at the University of Hawaii, which is part of the `comp.lang.verilog` archive. The package includes explanations of the mechanics of a digital event-driven simulator, including event queues and time wheels.

More technical references are included as part of Appendix B.

11.17 References

- IEEE Std 1364-95, Verilog LRM. 1995. The Institute of Electrical and Electronics Engineers. Available from The Institute of Electrical and Electronics Engineers, Inc., 345 East 47th Street, New York, NY 10017 USA. [cited on p. 479]
- Palnitkar, S. 1996. *Verilog HDL: A Guide to Digital Design and Synthesis*. Upper Saddle River, NJ: Prentice-Hall, 396 p. ISBN 0-13-451675-3.
- Thomas, D. E., and P. Moorby. 1991. *The Verilog Hardware Description Language*. 1st ed. Dordrecht, Netherlands: Kluwer, 223 p. ISBN 0-7923-9126-8, TK7885.7.T48 (1st ed.). ISBN 0-7923-9523-9 (2nd ed.).

LOGIC SYNTHESIS

1 2

- | | | | |
|------|----------------------------------|-------|-------------------------------------|
| 12.1 | A Logic-Synthesis Example | 12.9 | The Multiplier |
| 12.2 | A Comparator/MUX | 12.10 | The Engine Controller |
| 12.3 | Inside a Logic Synthesizer | 12.11 | Performance-Driven Synthesis |
| 12.4 | Synthesis of the Viterbi Decoder | 12.12 | Optimization of the Viterbi Decoder |
| 12.5 | Verilog and Logic Synthesis | 12.13 | Summary |
| 12.6 | VHDL and Logic Synthesis | 12.14 | Problems |
| 12.7 | Finite-State Machine Synthesis | 12.15 | Bibliography |
| 12.8 | Memory Synthesis | 12.16 | References |

Logic synthesis provides a link between an HDL (Verilog or VHDL) and a netlist similarly to the way that a C compiler provides a link between C code and machine language. However, the parallel is not exact. C was developed for use with compilers, but HDLs were not developed for use with logic-synthesis tools. Verilog was designed as a simulation language and VHDL was designed as a documentation and description language. Both Verilog and VHDL were developed in the early 1980s, well before the introduction of commercial logic-synthesis software. Because these HDLs are now being used for purposes for which they were not intended, the state of the art in logic synthesis falls far short of that for computer-language compilers. Logic synthesis forces designers to use a subset of both Verilog and VHDL. This makes using logic synthesis more difficult rather than less difficult. The current state of synthesis software is rather like learning a foreign language, and then having to talk to a five-year-old. When talking to a logic-synthesis tool using an HDL, it is necessary to think like hardware, anticipating the netlist that logic synthesis will produce. This situation should improve in the next five years, as logic synthesizers mature.

Designers use graphic or text design entry to create an HDL **behavioral model**, which does not contain any references to logic cells. State diagrams, graphical data-path descriptions, truth tables, RAM/ROM templates, and gate-level schematics may be used together with an HDL description. Once a behavioral HDL model is complete, two items are required to proceed: a **logic synthesizer** (software and documentation) and a cell library (the logic cells—NAND gates and such) that is called the **target library**. Most synthesis software companies produce only software. Most ASIC vendors produce only cell libraries. The behavioral model is simulated to check that the design meets the specifications and then the logic synthesizer is used to generate a netlist, a **structural model**, which contains only references to logic cells. There is no standard format for the netlists that logic synthesis produces, but EDIF is widely used. Some logic-synthesis tools can also create structural HDL (Verilog, VHDL, or both). Following logic synthesis the design is simulated again, and the results are compared with the earlier behavioral simulation. Layout for any type of ASIC may be generated from the structural model produced by logic synthesis.

12.1 A Logic-Synthesis Example

As an example of logic synthesis, we will compare two implementations of the Viterbi decoder described in Chapter 11. Both versions used logic cells from a VLSI Technology cell library. The first ASIC was designed by hand using schematic entry and a data book. The second version of the ASIC (the one that was fabricated) used Verilog for design entry and a logic synthesizer. Table 12.1 compares the two versions. The synthesized ASIC is 16 percent smaller and 13 percent faster than the hand-designed version.

How does logic synthesis generate smaller and faster circuits? Figure 12.1 shows the schematic for a hand-designed comparator and MUX used in the Viterbi decoder ASIC, called here the comparator/MUX example. The Verilog code and the schematic in Figure 12.1 describe the same function. The comparison,

TABLE 12.1 A comparison of hand design with synthesis (using a 1.0 μ m VLSI Technology cell library).

	Path delay/ ns ⁽¹⁾	No. of standard cells	No. of transistors	Chip area/ mils ²⁽²⁾
Hand design	41.6	1,359	16,545	21,877
Synthesized design	36.3	1,493	11,946	18,322

¹These delays are under nominal operating conditions with no wiring capacitance. This is the only stage at which a comparison could be made because the hand design was not completed.

²Both figures are initial layout estimates using default power-bus and signal routing widths.

in Table 12.2, of the two design approaches shows that the synthesized version is smaller and faster than the hand design, even though the synthesized design uses more cells.

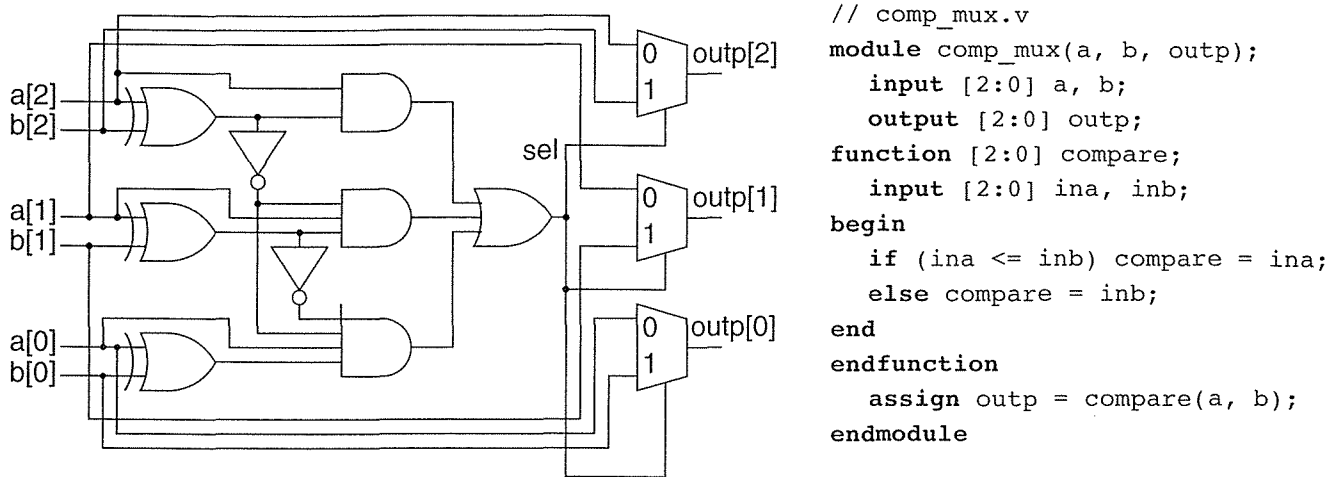


FIGURE 12.1 Schematic and HDL design entry.

TABLE 12.2 Comparison of the comparator/MUX designs using a 1.0 μ m standard-cell library.

	Delay /ns	No. of standard cells	No. of transistors	Area /mils ²
Hand design	4.3	12	116	68.68
Synthesized	2.9	15	66	46.43

12.2 A Comparator/MUX

With the Verilog behavioral model of Figure 12.1 as the input, logic-synthesis software generates logic that performs the same function as the Verilog. The software then optimizes the logic to produce a structural model, which references logic cells from the cell library and details their connections.

Before running a logic synthesizer, it is necessary to set up paths and startup files (`synopsys_dc.setup`, `compass.boom`, `view.ini`, or similar). These files set the target library and directory locations. Normally it is easier to run logic synthesis in text

```

`timescale 1ns / 10ps
module comp_mux_u (a, b, outp);
input  [2:0] a; input  [2:0] b;
output [2:0] outp;
supply1 VDD; supply0 VSS;

in01d0 u2 (.I(b[1]), .ZN(u2_ZN));
nd02d0 u3 (.A1(a[1]), .A2(u2_ZN), .ZN(u3_ZN));
in01d0 u4 (.I(a[1]), .ZN(u4_ZN));
nd02d0 u5 (.A1(u4_ZN), .A2(b[1]), .ZN(u5_ZN));
in01d0 u6 (.I(a[0]), .ZN(u6_ZN));
nd02d0 u7 (.A1(u6_ZN), .A2(u3_ZN), .ZN(u7_ZN));
nd02d0 u8 (.A1(b[0]), .A2(u3_ZN), .ZN(u8_ZN));
nd03d0 u9 (.A1(u5_ZN), .A2(u7_ZN), .A3(u8_ZN),
.ZN(u9_ZN));
in01d0 u10 (.I(a[2]), .ZN(u10_ZN));
nd02d0 u11 (.A1(u10_ZN), .A2(u9_ZN), .ZN(u11_ZN));
nd02d0 u12 (.A1(b[2]), .A2(u9_ZN), .ZN(u12_ZN));
nd02d0 u13 (.A1(u10_ZN), .A2(b[2]), .ZN(u13_ZN));
nd03d0 u14 (.A1(u11_ZN), .A2(u12_ZN), .A3(u13_ZN),
.ZN(u14_ZN));
nd02d0 u15 (.A1(a[2]), .A2(u14_ZN), .ZN(u15_ZN));
in01d0 u16 (.I(u14_ZN), .ZN(u16_ZN));
nd02d0 u17 (.A1(b[2]), .A2(u16_ZN), .ZN(u17_ZN));
nd02d0 u18 (.A1(u15_ZN), .A2(u17_ZN), .ZN(outp[2]));
nd02d0 u19 (.A1(a[1]), .A2(u14_ZN), .ZN(u19_ZN));
nd02d0 u20 (.A1(b[1]), .A2(u16_ZN), .ZN(u20_ZN));
nd02d0 u21 (.A1(u19_ZN), .A2(u20_ZN), .ZN(outp[1]));
nd02d0 u22 (.A1(a[0]), .A2(u14_ZN), .ZN(u22_ZN));
nd02d0 u23 (.A1(b[0]), .A2(u16_ZN), .ZN(u23_ZN));
nd02d0 u24 (.A1(u22_ZN), .A2(u23_ZN), .ZN(outp[0]));

endmodule

```

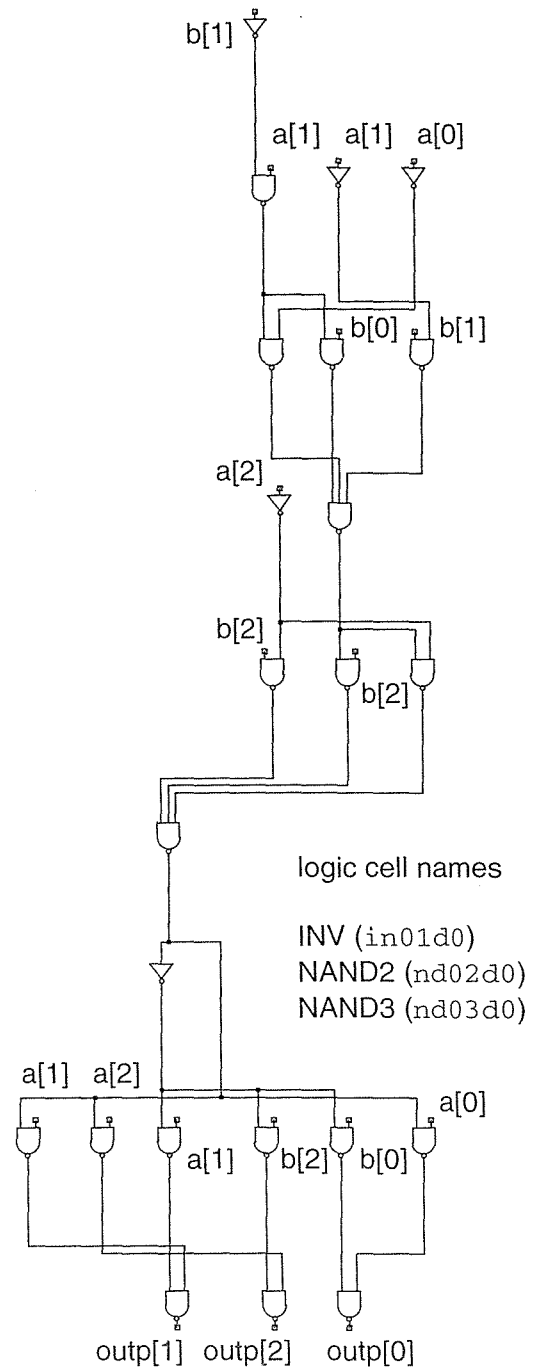


FIGURE 12.2 The comparator/MUX after logic synthesis, but before logic optimization. This figure shows the structural netlist, `comp_mux_u.v`, and its derived schematic.

mode using a script. A **script** is a text file that directs a software tool to execute a series of synthesis commands (we call this a **synthesis run**). Figure 12.2 shows a structural netlist, `comp_mux_u.v`, and the derived schematic after logic synthesis, but

```

`timescale 1ns / 10ps
module comp_mux_o (a, b, outp);
input  [2:0] a; input  [2:0] b;
output [2:0] outp;
supply1 VDD; supply0 VSS;

in01d0 B1_i1 (.I(a[2]),
.ZN(B1_i1_ZN));
in01d0 B1_i2 (.I(b[1]),
.ZN(B1_i2_ZN));
oa01d1 B1_i3 (.A1(a[0]),
.A2(B1_i4_ZN), .B1(B1_i2_ZN),
.B2(a[1]), .ZN(B1_i3_Z);
fn05d1 B1_i4 (.A1(a[1]),
.B1(b[1]), .ZN(B1_i4_ZN));
fn02d1 B1_i5 (.A(B1_i3_ZN),
.B(B1_i1_ZN), .C(b[2]),
.ZN(B1_i5_ZN));
mx21d1 B1_i6 (.I0(a[0]),
.I1(b[0]), .S(B1_i5_ZN),
.Z(outp[0]));
mx21d1 B1_i7 (.I0(a[1]),
.I1(b[1]), .S(B1_i5_ZN),
.Z(outp[1]));
mx21d1 B1_i8 (.I0(a[2]),
.I1(b[2]), .S(B1_i5_ZN),
.Z(outp[2]));

endmodule

```

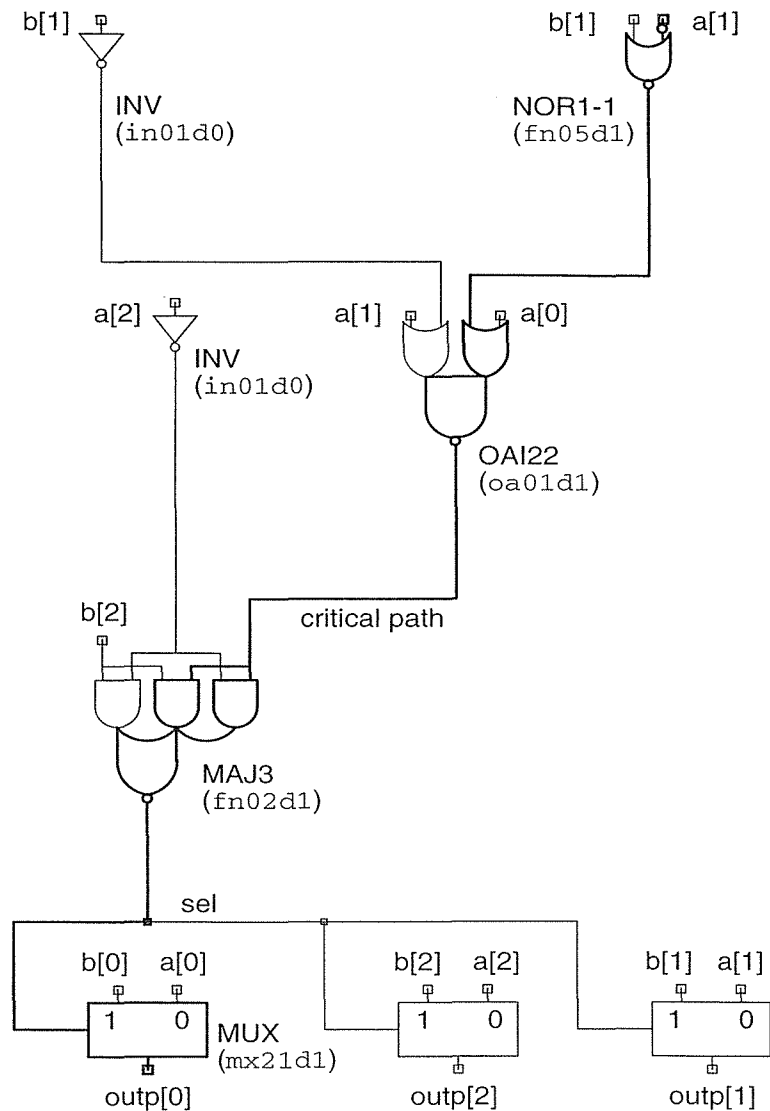


FIGURE 12.3 The comparator/MUX after logic synthesis and logic optimization with the default settings. This figure shows the structural netlist, `comp_mux_o.v`, and its derived schematic.

before any logic optimization. A **derived schematic** is created by software from a structural netlist (as opposed to a schematic drawn by hand). Figure 12.3 shows the structural netlist, `comp_mux_o.v`, and the derived schematic after logic optimization is performed (with the default settings). Figures 12.2 and 12.3 show the results of the two separate steps: logic synthesis and logic optimization. Confusingly, the whole pro-

cess, which includes synthesis and optimization (and other steps as well), is referred to as logic synthesis. We also refer to the software that performs all of these steps (even if the software consists of more than one program) as a logic synthesizer.

Logic synthesis parses (in a process sometimes called **analysis**) and translates (sometimes called **elaboration**) the input HDL to a data structure. This data structure is then converted to a network of generic logic cells. For example, the network in Figure 12.2 uses NAND gates (each with three or fewer inputs in this case) and inverters. This network of generic logic cells is technology-independent since cell libraries in any technology normally contain NAND gates and inverters. The next step, **logic optimization**, attempts to improve this technology-independent network under the controls of the designer. The output of the optimization step is an optimized, but still technology-independent, network. Finally, in the **logic-mapping** step, the synthesizer maps the optimized logic to a specified technology-dependent target cell library. Figure 12.3 shows the results of using a standard-cell library as the target.

Text reports such as the one shown in Table 12.3 may be the only output that the designer sees from the logic-synthesis tool. Often, synthesized ASIC netlists and the derived schematics containing thousands of logic cells are far too large to follow. To make things even more difficult, the net names and instance names in synthesized netlists are automatically generated. This makes it hard to see which lines of code in the HDL generated which logic cells in the synthesized netlist or derived schematic.

In the comparator/MUX example the derived schematics are simple enough that, with hindsight, it is clear that the XOR logic cell used in the hand design is logically inefficient. Using XOR logic cells does, however, result in the simple schematic of Figure 12.1. The synthesized version of the comparator/MUX in Figure 12.3 uses complex combinational logic cells that are logically efficient, but the schematic is not as easy to read. Of course, the computer does not care about this—and neither do we since we usually never see the schematic.

Which version is best—the hand-designed or the synthesized version? Table 12.3 shows statistics generated by the logic synthesizer for the comparator/MUX. To calculate the performance of each circuit that it evaluates during synthesis, there is a **timing-analysis** tool (also known as a **timing engine**) built into the logic synthesizer. The timing-analysis tool reports that the critical path in the optimized comparator/MUX is 2.43 ns. This critical path is highlighted on the derived schematic of Figure 12.3 and consists of the following delays:

- 0.33 ns due to cell `fn05d1`, instance name `B1_i4`, a two-input NOR cell with an inverted input. We might call this a NOR1-1 or $(A + B)'$ logic cell.
- 0.39 ns due to cell `oa01d1`, instance name `B1_i3`, an OAI22 logic cell.
- 1.03 ns due to logic cell `fn02d1`, instance name `B1_i5`, a three-input majority function, MAJ3 (A, B, C).
- 0.68 ns due to logic cell `mx21d1`, instance name `B1_i6`, a 2:1 MUX.

(In this cell library the 'd1' suffix indicates normal drive strength.)

TABLE 12.3 Reports from the logic synthesizer for the Verilog version of the comparator/MUX.

Command	Synthesizer output ¹						
> synthesize	Cell Name	Num Insts	Gate Count Per Cell	Tot Gate Count	Width Per Cell	Total Width	
	-----	-----	-----	-----	-----	-----	
	in01d0	5	.8	3.8	7.2	36.0	
	nd02d0	16	1.0	16.0	9.6	153.6	
	nd03d0	2	1.3	2.5	12.0	24.0	
	-----	-----	-----	-----	-----	-----	
	Totals:	23		22.2		213.6	
> optimize	Cell Name	Num Insts	Gate Count Per Cell	Tot Gate Count	Width Per Cell	Total Width	
	-----	-----	-----	-----	-----	-----	
	fn02d1	1	1.8	1.8	16.8	16.8	
	fn05d1	1	1.3	1.3	12.0	12.0	
	in01d0	2	.8	1.5	7.2	14.4	
	mx21d1	3	2.2	6.8	21.6	64.8	
	oa01d1	1	1.5	1.5	14.4	14.4	
	-----	-----	-----	-----	-----	-----	
	Totals:	8		12.8		122.4	
> report timing	instance name	incr	arrival	trs	rampDel	cap	cell
	inPin --> outPin	(ns)	(ns)		(ns)	(pf)	
	-----	-----	-----	-----	-----	-----	-----
	a[1]	.00	.00	R	.00	.04	comp_m...
	B1_i4						
	A1 --> ZN	.33	.33	R	.17	.03	fn05d1
	B1_i3						
	A2 --> ZN	.39	.72	F	.33	.06	oa01d1
	B1_i5						
	A --> ZN	1.03	1.75	R	.67	.11	fn02d1
	B1_i6						
	S --> Z	.68	2.43	R	.09	.02	mx21d1

¹Cell Name = cell name from the ASIC library (Compass Passport, 0.6 μm high-density, 5 V standard-cell library, cb60hd230); Num Insts = number of cell instances; Gate Count Per Cell = equivalent gates with two-input NAND = 1 gate (with number of transistors \approx equivalent gates \times 4); Width Per Cell = width in μm (cell height in this library is 72λ or $21.6 \mu\text{m}$); incr = incremental delay time due to logic cell delay; trs = transition; R = rising; F = falling; rampDel = ramp delay; cap = capacitance at node or cell output pin.

TABLE 12.4 Logic cell comparisons between the two comparator/MUX designs.

Cell type	Library cell name ¹	² t _{PLH} /ns	t _{PHL} /ns	Gate equivalents in cell ³	Cells used in hand design	Cells used in synthesized design	Gate equivalents used by hand design	Gate equivalents used in synthesized design	Width of cell ⁴ /μm	Width used by hand design /μm	Width of synthesized design /μm
Inverter	in01d0	0.37	0.36	0.8	2	2	1.6	1.6	7.2	14.4	14.4
2-input XOR	xo02d1	0.93	0.62	1.8	3	—	5.3	—	16.8	50.4	—
2-input AND	an02d1	0.34	0.46	1.3	1	—	1.3	—	12.0	12.0	—
3-input AND	an03d1	0.38	0.52	1.5	1	—	1.5	—	14.4	14.4	—
4-input AND	an04d1	0.41	0.98	1.8	1	—	1.8	—	16.8	16.8	—
3-input OR	or03d1	0.60	0.44	1.8	1	—	1.8	—	16.8	16.8	—
2-input MUX	mx21d1	0.69	0.68	2.2	3	3	6.6	6.6	21.6	64.8	64.8
AOI22	oa01d1	0.51	0.42	1.5	—	1	—	1.5	14.4	—	14.4
MAJ3	fn02d1	0.84	0.81	1.8	—	1	—	1.8	16.8	—	16.8
NOR1-1= (A' + B)'	fn05d1 ⁵	0.42	0.46	1.3	—	1	—	1.3	12.0	—	12.0
Totals					12	8	19.8	12.8		189.6	122.4

¹0.6 μm, 5 V, high-density Compass standard-cell library, cb60hd230.

²Average over all inputs with load capacitance equal to two standard loads (one standard load = 0.016 pF).

³2-input NAND = 1 gate equivalent.

⁴Cell height is 72 λ (21.6 μm).

⁵Rise and fall delays are different for the two inputs, A and B, of this cell: t_{PLHA} = 0.48 ns; t_{PLHB} = 0.36 ns; t_{PHLA} = 0.59 ns; t_{PHLB} = 0.33 ns.

Table 12.4 lists the name, type, the number of transistors, the area, and the delay of each logic cell used in the hand-designed and synthesized comparator/MUX. We could have performed this analysis by hand using the cell-library data book and a calculator or spreadsheet, but it would have been tedious work—especially calculating the delays. The computer is excellent at this type of bookkeeping. We can think of the timing engine of a logic synthesizer as a logic calculator.

We see from Table 12.4 that the sum of the widths of all the cells used in the synthesized design (122.4 μm) is less than for the hand design (189.6 μm). All the standard cells in a library are the same height, 72 λ or 21.6 μm, in this case. Thus the synthesized design is smaller. We could estimate the critical path of the hand design

using the information from the cell-library data book (summarized in Table 12.4). Instead we will use the timing engine in the logic synthesizer as a logic calculator to extract the critical path for the hand-designed comparator/MUX.

Table 12.5 shows a timing analysis obtained by loading the hand-designed schematic netlist into the logic synthesizer. Table 12.5 shows that the hand-designed (critical path 2.42 ns) and synthesized versions (critical path 2.43 ns) of the comparator/MUX are approximately the same speed. Remember, though, that we used the default settings during logic optimization. Section 12.11 shows that the logic synthesizer can do much better.

TABLE 12.5 Timing report for the hand-designed version of the comparator/MUX using the logic synthesizer to calculate the critical path (compare with Table 12.3).

Command	Synthesizer output ¹						
> report	instance name						
timing	inPin --> outPin	incr	arrival	trs	rampDel	cap	cell
		(ns)	(ns)		(ns)	(pf)	
	a[1]	.00	.00	F	.00	.04	comp_mux
	B1_i4						
	A1 --> ZN	.61	.61	F	.14	.03	xo02d1
	B1_i3						
	A2 --> ZN	.85	1.46	F	.19	.05	an04d1
	B1_i5						
	A --> ZN	.42	1.88	F	.23	.09	or03d1
	B1_i6						
	S --> Z	.54	2.42	R	.09	.02	mx21d1
	outp[0]	.00	2.42	R	.00	.00	comp_mux

¹See footnote 1 in Table 12.3 for explanations of the abbreviations used in this table.

12.2.1 An Actel Version of the Comparator/MUX

Figure 12.4 shows the results of targeting the comparator/MUX design to the Actel ACT 2/3 FPGA architecture. (The EDIF converter prefixes all internal nodes in this netlist with 'block_0_DEF_NET_'. This prefix was replaced with 'n_' in the Verilog file, `comp_mux_actel_o_ad1_e.v`, derived from the `.ad1` netlist.) As can be seen by comparing the netlists and schematics in Figures 12.3 and 12.4, the results are very different between a standard-cell library and the Actel library. Each of the symbols in the schematic in Figure 12.4 represents the eight-input ACT 2/3 C-Module (see Figure 5.4a). The logic synthesizer, during the technology-mapping step, has decided which connections should be made to the inputs to the combinational logic

```

`timescale 1 ns/100 ps
module comp_mux_actel_o (a, b, outp);
input [2:0] a, b; output [2:0] outp;
wire n_13, n_17, n_19, n_21, n_23, n_27, n_29,
n_31, n_62;

CM8 I_5_CM8(.D0(n_31), .D1(n_62), .D2(a[0]),
.D3(n_62), .S00(n_62), .S01(n_13), .S10(n_23),
.S11(n_21), .Y(outp[0]));
CM8 I_2_CM8(.D0(n_31), .D1(n_19), .D2(n_62),
.D3(n_62), .S00(n_62), .S01(b[1]), .S10(n_31),
.S11(n_17), .Y(outp[1]));
CM8 I_1_CM8(.D0(n_31), .D1(n_31), .D2(b[2]),
.D3(n_31), .S00(n_62), .S01(n_31), .S10(n_31),
.S11(a[2]), .Y(outp[2]));
VCC VCC_I(.Y(n_62));
CM8 I_4_CM8(.D0(a[2]), .D1(n_31), .D2(n_62),
.D3(n_62), .S00(n_62), .S01(b[2]), .S10(n_31),
.S11(a[1]), .Y(n_19));
CM8 I_7_CM8(.D0(b[1]), .D1(b[2]), .D2(n_31),
.D3(n_31), .S00(a[2]), .S01(b[1]), .S10(n_31),
.S11(a[1]), .Y(n_23));
CM8 I_9_CM8(.D0(n_31), .D1(n_31), .D2(a[1]),
.D3(n_31), .S00(n_62), .S01(b[1]), .S10(n_31),
.S11(b[0]), .Y(n_27));
CM8 I_8_CM8(.D0(n_29), .D1(n_62), .D2(n_31),
.D3(a[2]), .S00(n_62), .S01(n_27), .S10(n_31),
.S11(b[2]), .Y(n_13));
CM8 I_3_CM8(.D0(n_31), .D1(n_31), .D2(a[1]),
.D3(n_31), .S00(n_62), .S01(a[2]), .S10(n_31),
.S11(b[2]), .Y(n_17));
CM8 I_6_CM8(.D0(b[2]), .D1(n_31), .D2(n_62),
.D3(n_62), .S00(n_62), .S01(a[2]), .S10(n_31),
.S11(b[0]), .Y(n_21));
CM8 I_10_CM8(.D0(n_31), .D1(n_31), .D2(b[0]),
.D3(n_31), .S00(n_62), .S01(n_31), .S10(n_31),
.S11(a[2]), .Y(n_29));
GND GND_I(.Y(n_31));
endmodule

```

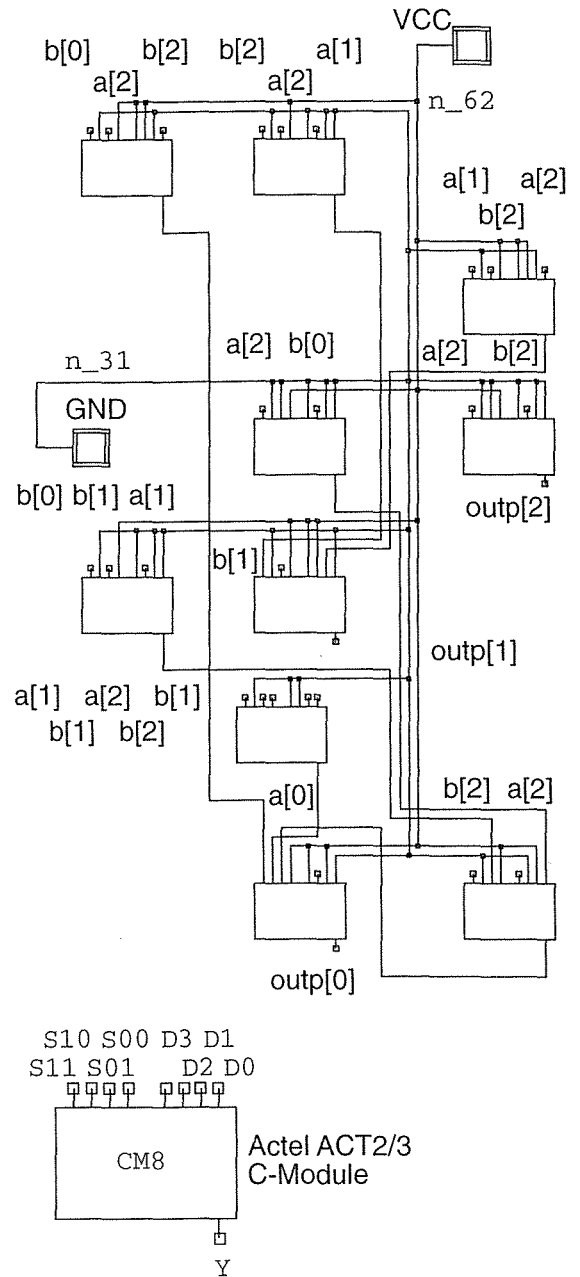


FIGURE 12.4 The Actel version of the comparator/MUX after logic optimization. This figure shows the structural netlist, `comp_mux_actel_o_ad1_e.v`, and its derived schematic.

macro, CM8. The CM8 names and the ACT2/3 C-Module names (in parentheses) correspond as follows: S00(A0), S01(B0), S10(A1), S11(A2), D0(D00), D1(D01), D2(D10), D3(D11), and Y(Y).

12.3 Inside a Logic Synthesizer

The logic synthesizer parses the Verilog of Figure 12.1 and builds an internal data structure (usually a graph represented by linked lists). Such an abstract representation is not easy to visualize, so we shall use pictures instead. The first **Karnaugh map** in Figure 12.5(a) is a picture that represents the `sel` signal (labeled as the input to the three MUXes in the schematic of Figure 12.1) for the case when the inputs are such that $a[2]b[2] = 00$. The signal `sel` is responsible for steering the smallest input, `a` or `b`, to the output of the comparator/MUX. We insert a '1' in the Karnaugh map (which will select the input `b` to be the output) whenever `b` is smaller than `a`. When $a = b$ we do not care whether we select `a` or `b` (since `a` and `b` are equal), so we insert an 'x', a don't care logic value, in the Karnaugh map of Figure 12.5(a). There are four Karnaugh maps for the signal `sel`, one each for the values $a[2]b[2] = 00$, $a[2]b[2] = 01$, $a[2]b[2] = 10$, and $a[2]b[2] = 11$.

Next, **logic minimization** tries to find a minimum cover for the Karnaugh maps—the smallest number of the largest possible circles to cover all the '1's. One possible cover is shown in Figure 12.5(b).

In order to understand the steps that follow we shall use some notation from the **Berkeley Logic Interchange Format (BLIF)** and from the Berkeley tools `misII` and `sis`. We shall use the logic operators (in decreasing order of their precedence): '!' (negation), '*' (AND), '+' (OR). We shall also abbreviate Verilog signal names; writing `a[2]` as `a2`, for example. We can write equations for `sel` and the output signals of the comparator/MUX in the format that is produced by `sis`, as follows (this is the same format as input file for the Berkeley tool `eqntott`):

$$\text{sel} = a1*!b1*!b2 + a0*!b1*!b2 + a0*a1*!b2 + a1*!b1*a2 + a0*!b1*a2 + a0*a1*a2 + a2*!b2; \quad [12.1]$$

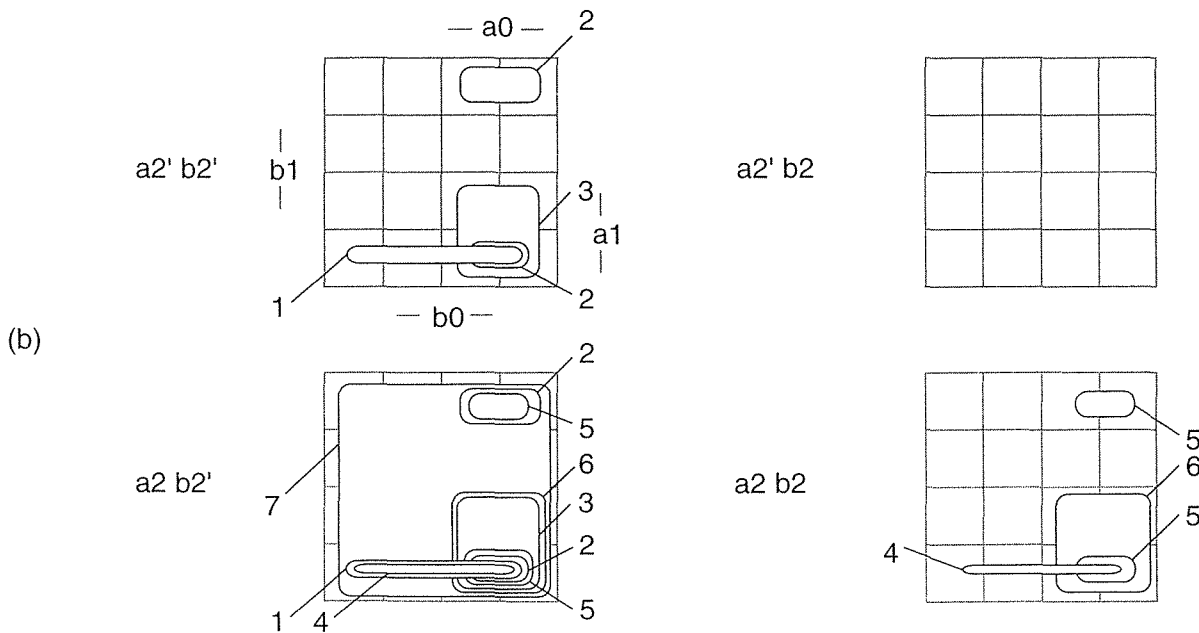
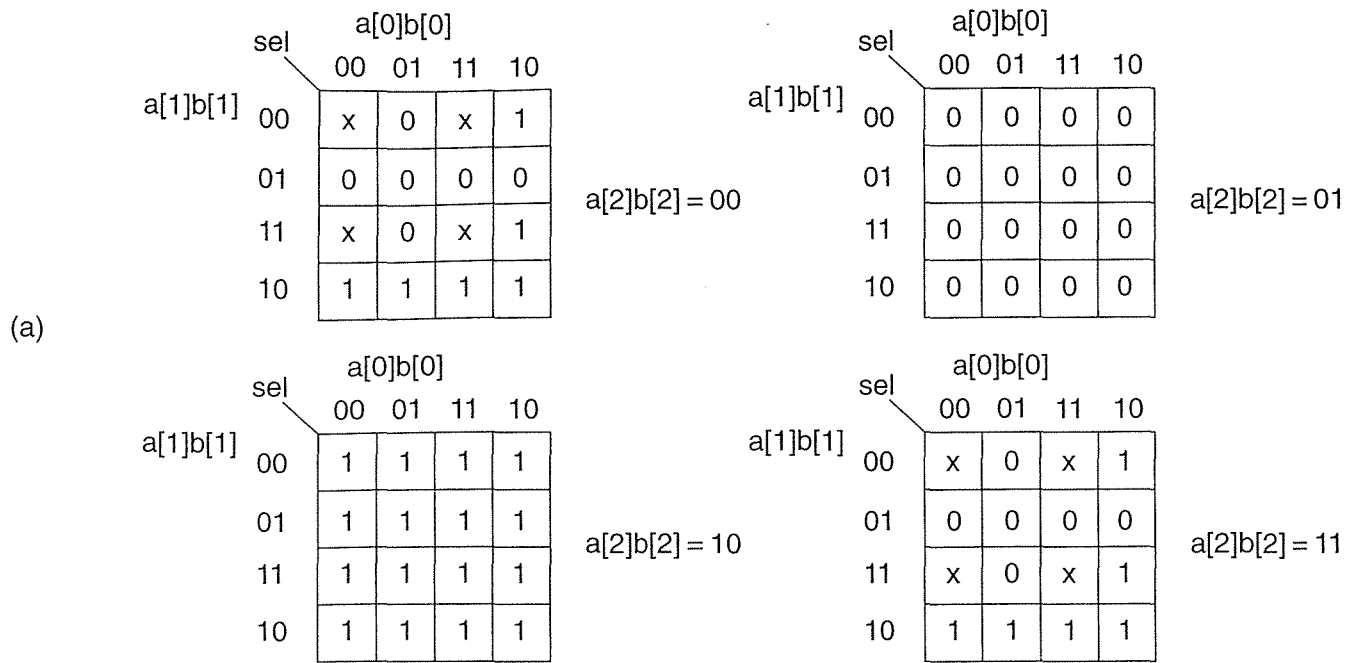
$$\text{outp2} = !\text{sel}*a2 + \text{sel}*b2; \quad [12.2]$$

$$\text{outp1} = !\text{sel}*a1 + \text{sel}*b1; \quad [12.3]$$

$$\text{outp0} = !\text{sel}*a0 + \text{sel}*b0; \quad [12.4]$$

Equations 12.1–12.4 describe the **synthesized network**. There are seven product terms in Eq. 12.1—the logic equation for `sel` (numbered and labeled in the drawing of the cover for `sel` in Figure 12.5). We shall keep track of the `sel` signal separately even though this is not exactly the way the logic synthesizer works—the synthesizer looks at all the signals at once.

Logic optimization uses a series of factoring, substitution, and elimination steps to simplify the equations that represent the synthesized network. A simple analogy would be the simplification of arithmetic expressions. Thus, for example, we can simplify $189/315$ to 0.6 by factoring the top and bottom lines and eliminating common factors as follows: $(3 \times 7 \times 9)/(5 \times 7 \times 9) = 3/5$. Boolean algebra is more complicated than ordinary algebra. To make logic optimization tractable, most tools use algorithms based on algebraic factors rather than Boolean factors.



$$sel = a1!b1!b2 + a0!b1!b2 + a0*a1!b2 + a1!b1*a2 + a0!b1*a2 + a0*a1*a2 + a2!*b1$$

$$\square\square\square\square = \square\square\square\square 1 \square\square\square\square \square\square + \square\square\square\square 2 \square\square\square\square\square\square + \square\square\square\square 3 \square\square\square\square + \square\square\square\square 4 \square\square\square\square\square + \square\square\square\square 5 \square\square\square\square\square + \square\square\square\square 6 \square\square\square\square + \square\square 7$$

FIGURE 12.5 Logic maps for the comparator/MUX. (a) If the input b is less than a, then sel is '1'. If a = b, then sel = 'x' (don't care). (b) A cover for sel.

Logic optimization attempts to simplify the equations in the hope that this will also minimize area and maximize speed. In the synthesis results presented in Table 12.3, we accepted the default optimization settings without setting any constraints. Thus only a minimum amount of logic optimization is attempted that did not alter the synthesized network in this case.

The **technology-decomposition** step builds a generic network from the optimized logic network. The generic network is usually simple NAND gates (*sis* uses either AND, or NOR gates, or both). This generic network is in a technology-independent form. To build this generic network involves creating intermediate nodes. The program *sis* labels these intermediate nodes [n], starting at n = 100.

```

sel = [100] * [101] * [102] ;                               [12.5]
[100] = !( !a2 * [103] );
[101] = !( b2 * [103] );
[102] = !( !a2 * b2 );
[103] = !( [104] * [105] * [106] );
[104] = !( !a1 * b1 );
[105] = !( b0 * [107] );
[106] = !( a0' * [107] );
[107] = !( a1 * !b1 );

outp2 = !( [108] * [109] );                                [12.6]
[108] = !( a2 * !sel );
[109] = !( sel * b2 );

```

There are two other sets of equations, similar to Eq. 12.6, for *outp1* and *outp0*. Notice the polarity of the *sel* signal in Eq. 12.5 is correct and represents an AND gate (a consequence of labeling *sel* as the MUX select input in Figure 12.1).

Next, the **technology-mapping** step (or **logic-mapping** step) implements the technology-independent network by matching pieces of the network with the logic cells that are available in a technology-dependent cell library (an FPGA or standard-cell library, for example). While performing the logic mapping, the algorithms attempt to minimize area (the default constraint) while meeting any other user constraints (timing or power constraints, for example).

Working backward from the outputs the logic mapper recognizes that each of the three output nodes (*outp2*, *outp1*, and *outp0*) may be mapped to a MUX. (We are using the term “node mapping to a logic cell” rather loosely here—an exact parallel is a compiler mapping patterns of source code to object code.) Here is the equation that shows the mapping for *outp2*:

```

outp2 = MUX(a, b, c) = ac + b!c                             [12.7]
a = b2 ; b = a2 ; c = sel

```

The equations for *outp1* and *outp0* are similar.

The node *sel* can be mapped to the three-input majority function as follows:

```

sel = MAJ3(w, x, y) = !(wx + wy + xy)                     [12.8]
w = !a2 ; x = b2 ; y = [103] ;

```

Next node [103] is mapped to an OAI22 cell,

$$\begin{aligned}
 [103] &= \text{OAI22}(w, x, y, z) = !((w + x)(y + z)) = \\
 &(!w!x + !y!z) \qquad \qquad \qquad [12.9] \\
 w &= a0 ; x = a1 ; y = !b1 z = [107] ;
 \end{aligned}$$

Finally, node [107] is mapped to a two-input NOR with one inverted input,

$$[107] = !(b1 + !a1) ; \qquad \qquad \qquad [12.10]$$

Putting Equations 12.7–12.10 together describes the following optimized logic network (corresponding to the structural netlist and schematic shown in Figure 12.3):

$$\begin{aligned}
 \text{sel} &= !(((!a0 * !(a1\&!b1) | (b1*!a1)) * (!a2|b2)) | (!a2*b2)) ; [12.11] \\
 \text{outp2} &= !\text{sel} * a2 | \text{sel} * b2; \\
 \text{outp1} &= !\text{sel} * a1 | \text{sel} * b1; \\
 \text{outp0} &= !\text{sel} * a0 | \text{sel} * b0;
 \end{aligned}$$

The comparator/MUX example illustrates how logic synthesis takes the behavioral model (the HDL input) and, in a series of steps, converts this to a structural model describing the connections of logic cells from a cell library.

When we write a C program we almost never think of the object code that will result. When we write HDL it is always necessary to consider the hardware. In C there is not much difference between $i*j$ and i/j . In an HDL, if i and j are 32-bit numbers, $i*j$ will take up a large amount of silicon. If j is a constant, equal to 2, then $i*j$ take up hardly any space at all. Most logic synthesizers cannot even produce logic to implement i/j . In the following sections we shall examine the Verilog and VHDL languages as a way to communicate with a logic synthesizer. Using one of these HDLs we have to tell the logic synthesizer what hardware we want—we **imply** A. The logic synthesizer then has to figure out what we want—it has to **infer** B. The problem is making sure that we write the HDL code such that $A = B$. As will become apparent, the more clearly we imply what we mean, the easier the logic synthesizer can infer what we want.

12.4 Synthesis of the Viterbi Decoder

In this section we return to the Viterbi decoder from Chapter 11. After an initial synthesis run that shows how logic synthesis works with a real example, we step back and study some of the issues and problems of using HDLs for logic synthesis.

12.4.1 ASIC I/O

Some logic synthesizers can include I/O cells automatically, but the designer may

have to use directives to designate special pads (clock buffers, for example). It may also be necessary to use commands to set I/O cell features such as selection of pull-up resistor, slew rate, and so on. Unfortunately there are no standards in this area. Worse, there is currently no accepted way to set these parameters from an HDL. Designers may also use either generic technology-independent I/O models or instantiate I/O cells directly from an I/O cell library. Thus, for example, in the Compass tools the statement

```
asPadIn #(3,"1,2,3") u0 (in0, padin0);
```

uses a generic I/O cell model, `asPadIn`. This statement will generate three input pads (with pin numbers "1", "2", and "3") if `in0` is a 3-bit-wide bus.

The next example illustrates the use of generic I/O cells from a standard-component library. These components are technology independent (so they may equally well be used with a 0.6 μm or 0.35 μm technology).

```
module allPads(padTri, padOut, clkOut, padBidir, padIn, padClk); //1
    output padTri, padOut, clkOut; inout padBidir; //2
    input [3:0] padIn; input padClk; wire [3:0] in; //3
//compass dontTouch u* //4
// asPadIn #(W, N, L, P) I (toCore, Pad) also asPadInInv //5
// asPadOut #(W, N, L, P) I (Pad, frCore) //6
// asPadTri #(W, N, S, L, P) I (Pad, frCore, OEN) //7
// asPadBidir #(W, N, S, L, P) I (Pad, toCore, frCore, OEN) //8
// asPadClk #(N, S, L) I (Clk, Pad) also asPadClkInv //9
// asPadVxx #(N, subnet) I (Vxx) //10
// W = width, integer (default=1) //11
// N = pin number string, e.g. "1:3,5:8" //12
// S = strength = {2, 4, 8, 16} in mA drive //13
// L = level = {cmos, ttl, schmitt} (default = cmos) //14
// P = pull-up resistor = {down, float, none, up} //15
// Vxx = {Vss, Vdd} //16
// subnet = connect supply to {pad, core, both} //17
    asPadIn #(4,"1:4","", "none") u1 (in, padIn); //18
    asPadOut #(1,"5",13) u2 (padOut, d); //19
    asPadTri #(1,"6",11) u3 (padTri, in[1], in[0]); //20
    asPadBidir #(1,"7",2,"", "") u4 (d, padBidir, in[3], in[2]); //21
    asPadClk #(8) u5 (clk, padClk); //22
    asPadOut #(1, "9") u6 (clkOut, clk); //23
    asPadVdd #("10:11","pads") u7 (vddr); //24
    asPadVss #("12,13","pads") u8 (vssr); //25
    asPadVdd #("14","core") u9 (vddc); //26
    asPadVss #("15","core") u10 (vssc); //27
    asPadVdd #("16","both") u11 (vddb); //28
    asPadVss #("17","both") u12 (vssb); //29
endmodule //30
```

The following code is an example of the contents of a generic model for a three-state I/O cell (provided in a standard-component library or in an I/O cell library):

```

module PadTri (Pad, I, Oen); // active-low output enable           //1
parameter width = 1, pinNumbers = "", \strength = 1,           //2
    level = "CMOS", externalVdd = 5;                             //3
output [width-1:0] Pad; input [width-1:0] I; input Oen;       //4
assign #1 Pad = (Oen ? {width{1'bz}} : I);                     //5
endmodule                                                       //6

```

The module PadTri can be used for simulation and as the basis for synthesizing an I/O cell. However, the synthesizer also has to be told to synthesize an I/O cell connected to a bonding pad and the outside world and not just an internal three-state buffer. There is currently no standard mechanism for doing this, and every tool and every ASIC company handles it differently.

The following model is a generic model for a bidirectional pad. We could use this model as a basis for input-only and output-only I/O cell models.

```

module PadBidir (C, Pad, I, Oen); // active-low output enable   //1
parameter width = 1, pinNumbers = "", \strength = 1,           //2
    level = "CMOS", pull = "none", externalVdd = 5;             //3
output [width-1:0] C; inout [width-1:0] Pad;                  //4
input [width-1:0] I; input Oen;                                //5
assign #1 Pad = Oen ? {width{1'bz}} : I; assign #1 C = Pad;   //6
endmodule                                                       //7

```

In Chapter 8 we used the halfgate example to demonstrate an FPGA design flow—including I/O. If the synthesis tool is not capable of synthesizing I/O cells, then we may have to instantiate them by hand; the following code is a hand-instantiated version of lines 19–22 in module allPads:

```

pc5o05 u2_2 (.PAD(padOut), .I(d));
pc5t04r u3_2 (.PAD(padTri), .I(in[1]), .OEN(in[0]));
pc5b01r u4_3 (.PAD(padBidir), .I(in[3]), .CIN(d), .OEN(in[2]));
pc5d01r u5_in_1 (.PAD(padClk), .CIN(u5toClkBuf[0]));

```

The designer must find the names of the I/O cells (pc5o05 and so on), and the names, positions, meanings, and defaults for the parameters from the cell-library documentation.

I/O cell models allow us to simulate the behavior of the synthesized logic inside an ASIC “all the way to the pads.” To simulate “outside the pads” at a system level, we should use these same I/O cell models. This is important in ASIC design. For example, the designers forgot to put pull-up resistors on the outputs of some of the SparcStation ASICs. This was one of the very few errors in a complex project, but an error that could have been caught if a system-level simulation had included complete I/O cell models for the ASICs.

12.4.2 Flip-Flops

In Chapter 11 we used this D flip-flop model to simulate the Viterbi decoder:

```

module dff(D,Q,Clock,Reset); // N.B. reset is active-low           //1
output Q; input D,Clock,Reset;                                   //2
parameter CARDINALITY = 1; reg [CARDINALITY-1:0] Q;             //3
wire [CARDINALITY-1:0] D;                                       //4
always @(posedge Clock) if (Reset!=0) #1 Q=D;                   //5
always begin wait (Reset==0); Q=0; wait (Reset==1); end         //6
endmodule                                                         //7

```

Most simulators cannot synthesize this model because there are two `wait` statements in one `always` statement (line 6). We could change the code to use flip-flops from the synthesizer standard-component library by using the following code:

```
asDff ff1 (.Q(y), .D(x), .Clk(clk), .Rst(vdd));
```

Unfortunately we would have to change all the flip-flop models from `'dff'` to `'asDff'` and the code would become dependent on a particular synthesis tool. Instead, to maintain independence from vendors, we shall use the following D flip-flop model for synthesis and simulation:

```

module dff(D, Q, Clk, Rst); // new flip-flop for Viterbi decoder //1
parameter width = 1, reset_value = 0; input [width - 1 : 0] D; //2
output [width - 1 : 0] Q; reg [width - 1 : 0] Q; input Clk, Rst; //3
initial Q <= {width{1'bx}};                                       //4
always @ ( posedge Clk or negedge Rst )                           //5
if ( Rst == 0 ) Q <= #1 reset_value; else Q <= #1 D;             //6
endmodule                                                         //7

```

12.4.3 The Top-Level Model

The following code models the top-level Viterbi decoder and instantiates (with instance name `v_1`) a copy of the Verilog module `viterbi` from Chapter 11. The model uses generic input, output, power, and clock I/O cells from the standard-component library supplied with the synthesis software. The synthesizer will take these generic I/O cells and map them to I/O cells from a technology-specific library. We do not need three-state I/O cells or bidirectional I/O cells for the Viterbi ASIC.

```

/* This is the top-level module, viterbi_ASIC.v */               //1
module viterbi_ASIC                                             //2
(padin0, padin1, padin2, padin3, padin4, padin5, padin6, padin7, //3
padOut, padClk, padRes, padError);                               //4
input [2:0] padin0, padin1, padin2, padin3,                       //5
          padin4, padin5, padin6, padin7;                         //6
input padRes, padClk; output padError; output [2:0] padOut;     //7
wire Error, Clk, Res; wire [2:0] Out; // core                    //8

```

```

wire padError, padClk, padRes; wire [2:0] padOut; //9
wire [2:0] in0,in1,in2,in3,in4,in5,in6,in7; // core //10
wire [2:0] //11
    padin0, padin1,padin2,padin3,padin4,padin5,padin6,padin7; //12
// Do not let the software mess with the pads. //13
//compass dontTouch u* //14
    asPadIn    #(3,"1,2,3")    u0    (in0, padin0); //15
    asPadIn    #(3,"4,5,6")    u1    (in1, padin1); //16
    asPadIn    #(3,"7,8,9")    u2    (in2, padin2); //17
    asPadIn    #(3,"10,11,12") u3    (in3, padin3); //18
    asPadIn    #(3,"13,14,15") u4    (in4, padin4); //19
    asPadIn    #(3,"16,17,18") u5    (in5, padin5); //20
    asPadIn    #(3,"19,20,21") u6    (in6, padin6); //21
    asPadIn    #(3,"22,23,24") u7    (in7, padin7); //22
    asPadVdd   #("25","both")  u25   (vddb); //23
    asPadVss   #("26","both")  u26   (vssb); //24
    asPadClk   #("27")         u27   (Clk, padClk); //25
    asPadOut   #(1,"28")       u28   (padError, Error); //26
    asPadin    #(1,"29")       u29   (Res, padRes); //27
    asPadOut   #(3,"30,31,32") u30   (padOut, Out); //28
// Here is the core module: //29
viterbi v_1 //30
    (in0,in1,in2,in3,in4,in5,in6,in7,Out,Clk,Res,Error); //31
endmodule //32

```

At this point we are ready to begin synthesis. In order to demonstrate how synthesis works, I am cheating here. The code that was presented in Chapter 11 has already been simulated and synthesized (requiring several iterations to produce error-free code). What I am doing is a little like the Galloping Gourmet’s television presentation: “And then we put the soufflé in the oven ... and look at the soufflé that I prepared earlier.” The synthesis results for the Viterbi decoder are shown in Table 12.6. Normally the worst thing we can do is prepare a large amount of code, put it in the synthesis oven, close the door, push the “synthesize and optimize” button, and wait. Unfortunately, it is easy to do. In our case it works (at least we may think so at this point) because this is a small ASIC by today’s standards—only a few thousand gates. I made the bus widths small and chose this example so that the code was of a reasonable size. Modern ASICs may be over one million gates, hundreds of times more complicated than our Viterbi decoder example.

The derived schematic for the synthesized core logic is shown in Figure 12.6. There are eight boxes in Figure 12.6 that represent the eight modules in the Verilog code. The schematics for each of these eight blocks are too complex to be useful. With practice it is possible to “see” the synthesized logic from reports such as Table 12.6. First we check the following cells at the top level:

- pc5c01 is an I/O cell that drives the clock node into the logic core. ASIC designers also call an I/O cell a **pad cell**, and often refer to the pad cells (the

TABLE 12.6 Initial synthesis results of the Viterbi decoder ASIC.

Command	Synthesizer output ^{1, 2}					
	Cell Name	Num Insts	Gate Count Per Cell	Tot Gate Count	Width Per Cell	Total Width
> optimize						
	pc5c01	1	315.4	315.4	100.8	100.8
	pc5d01r	26	315.4	8200.4	100.8	2620.8
	pc5o06	4	315.4	1261.6	100.8	403.2
	pv0f	1	315.4	315.4	100.8	100.8
	pvdv	1	315.4	315.4	100.8	100.8
	viterbi_p	1	1880.0	1880.0	18048.0	18048.0

¹See footnote 1 in Table 12.3 for explanations of the abbreviations used in this table.

²I/O cell height (I/O cells have prefixes pc5 and pv) is approximately 650 μm in this cell library.

bonding pads and associated logic) as just “the pads.” From the library data book we find this is a “core-driven, noninverting clock buffer capable of driving 125 pF.” This is a large logic cell and does not have a bonding pad, but is placed in a pad site (a slot in the ring of pads around the perimeter of the die) as if it were an I/O cell with a bonding pad.

- pc5d01r is a 5V CMOS input-only I/O cell with a bus repeater. Twenty-four of these I/O cells are used for the 24 inputs (in0 to in7). Two more are used for Res and clk. The I/O cell for clk receives the clock signal from the bonding pad and drives the clock buffer cell (pc5c01). The pc5c01 cell then buffers and drives the clock back into the core. The power-hungry clock buffer is placed in the pad ring near the VDD and VSS pads.
- pc5o06 is a CMOS output-only I/O cell with 6X drive strength (6 mA AC drive and 4 mA DC drive). There are four output pads: three pads for the signal outputs, outp[2:0], and one pad for the output signal, error.
- pv0f is a power pad that connects all VSS power buses on the chip.
- pvdv is a power pad that connects all VDD power buses on the chip.
- viterbi_p is the core logic. This cell takes its name from the top-level Verilog module (viterbi). The software has appended a “_p” suffix (the default) to prevent input files being accidentally overwritten.

The software does not tell us any of this directly. We learn what is going on by looking at the names and number of the synthesized cells, reading the synthesis tool documentation, and from experience. We shall learn more about I/O pads and the layout of power supply buses in Chapter 16.

Next we examine the cells used in the logic core. Most synthesis tools can produce reports, such as that shown in Table 12.7, which lists all the synthesized cells.

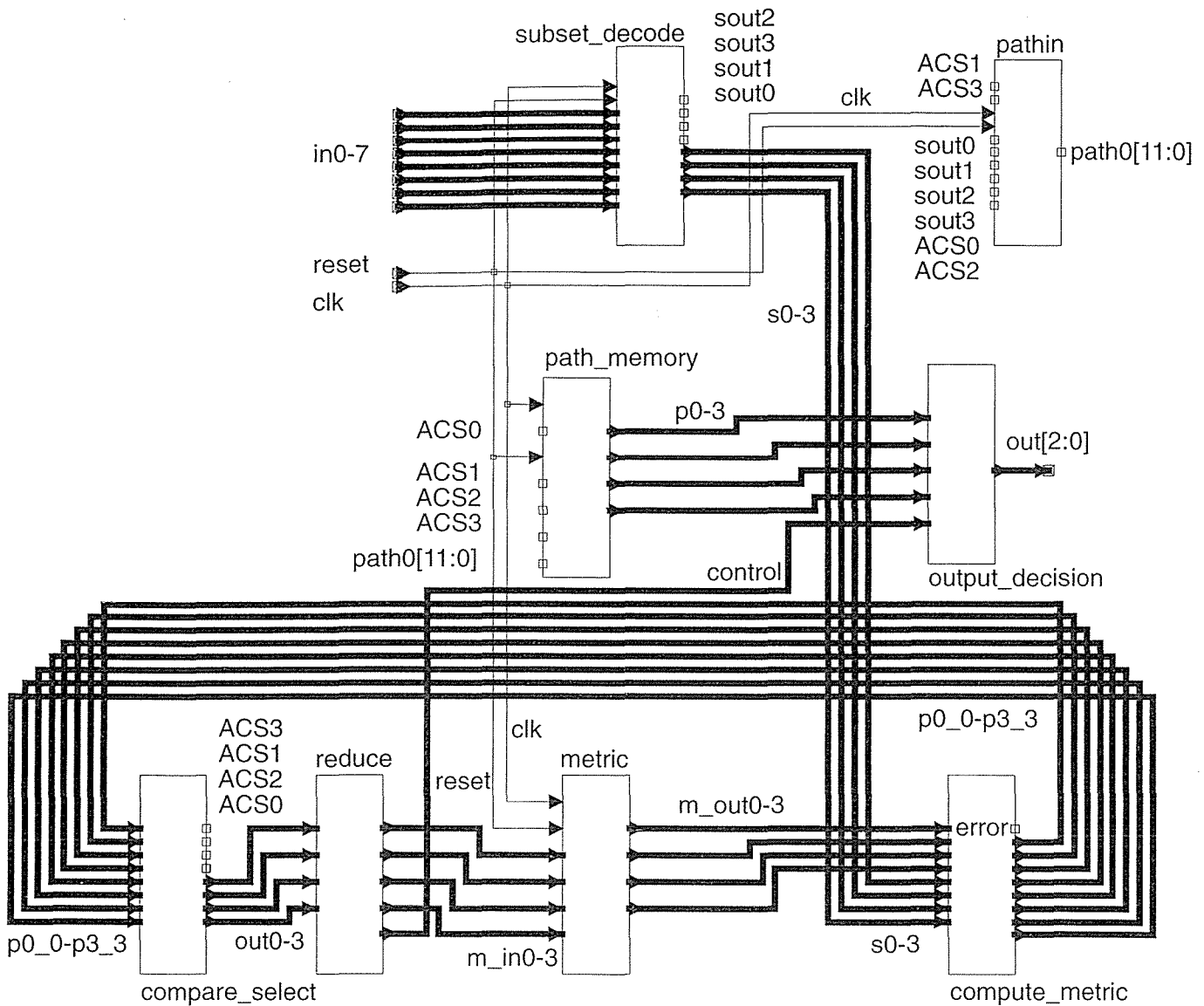


FIGURE 12.6 The core logic of the Viterbi decoder ASIC. Bus names are abbreviated in this figure for clarity. For example the label m_out0-3 denotes the four buses: m_out0 , m_out1 , m_out2 , and m_out3 .

The most important types of cells to check are the sequential elements: flip-flops and latches (I have omitted all but the sequential logic cells in Table 12.7). One of the most common mistakes in synthesis is to accidentally leave variables unassigned in all situations in the HDL. Unassigned variables require memory and will generate unnecessary sequential logic. In the Viterbi decoder it is easy to identify the sequential logic cells that should be present in the synthesized logic because we used the module `dff` explicitly whenever we required a flip-flop. By scanning the code in

Chapter 11 and counting the references to the `dff` model, we can see that the only flip-flops that should be inferred are the following:

- 24 (3×8) D flip-flops in instance `subset_decode`
- 132 (11×12) D flip-flops in instance `path_memory` that contains 11 instances of `path` (12 D flip-flops in each instance of `path`)
- 12 D flip-flops in instance `pathin`
- 20 (5×4) D flip-flops in instance `metric`

The total is $24 + 132 + 12 + 20 = 188$ D flip-flops, which is the same as the number of `dfctnb` cell instances in Table 12.7.

TABLE 12.7 Number of synthesized flip-flops in the Viterbi ASIC.

Command	Synthesizer output ¹					
> report						
area -flat		Num	Gate Count	Tot Gate	Width	Total
	Cell Name	Insts	Per Cell	Count	Per Cell	Width
	-----	-----	-----	-----	-----	-----
	...					
	dfctnb	188	5.8	1081.0	55.2	10377.6
	...					
	-----	-----	-----	-----	-----	-----
	Totals:	1383		12716.5		25485.6

¹See footnote 1 in Table 12.3 for explanations of the abbreviations used in this table. Logic cell `dfctnb` is a D flip-flop with clear in this standard-cell library.

Table 12.6 gives the total width of the standard cells in the logic core after logic optimization as $18,048 \mu\text{m}$. Since the standard-cell height for this library is 72λ ($21.6 \mu\text{m}$), we can make a first estimate of the total logic cell area as

$$(18,048 \mu\text{m}) (21.6 \mu\text{m}) = 390 \text{ k}(\mu\text{m}^2) \approx \frac{390 \text{ k}(\mu\text{m}^2)}{(25.4 \mu\text{m})^2} \approx 600 \text{ mil}^2. \quad (12.12)$$

In the physical layout we shall need additional space for routing. The ratio of routing to logic cell area is called the **routing factor**. The routing factor depends primarily on whether we use two levels or three levels of metal. With two levels of metal the routing factor is typically between 1 and 2. With three levels of metal, where we may use over-the-cell routing, the routing factor is usually zero to 1. We thus expect a logic core area of 600–1000 mils² for the Viterbi decoder using this cell library.

From Table 12.6 we see the I/O cells in this library are $100.8\mu\text{m}$ wide or approximately 4 mil (the width of a single pad site). From the I/O cell data book we find the I/O cell height is $650\mu\text{m}$ (actually $648.825\mu\text{m}$) or approximately 26 mil. Each I/O cell thus occupies 104mil^2 . Our 33 pad sites will thus require approximately 3400mil^2 which is larger than the estimated core logic area.

Let us go back and take a closer look at what it usually takes to get to this point. Remember we used an already prepared Verilog model for the Viterbi decoder.

12.5 Verilog and Logic Synthesis

A top-down design approach using Verilog begins with a single module at the top of the hierarchy to model the input and output response of the ASIC:

```
module MyChip_ASIC(); ... (code to model ASIC I/O) ... endmodule;
```

This top-level Verilog module is used to simulate the ASIC I/O connections and any bus I/O during the earliest stages of design. Often the reason that designs fail is lack of attention to the connection between the ASIC and the rest of the system.

As a designer, you proceed down through the hierarchy as you add lower-level modules to the top-level Verilog module. Initially the lower-level modules are just empty placeholders, or **stubs**, containing a minimum of code. For example, you might start by using inverters just to connect inputs directly to the outputs. You expand these stubs before moving down to the next level of modules.

```
module MyChip_ASIC()
    // behavioral "always", etc. ...
    SecondLevelStub1 port mapping
    SecondLevelStub2 port mapping
    ... endmodule
module SecondLevelStub1() ... assign Output1 = ~Input1; endmodule
module SecondLevelStub2() ... assign Output2 = ~Input2;
endmodule
```

Eventually the Verilog modules will correspond to the various component pieces of the ASIC.

12.5.1 Verilog Modeling

Before we could start synthesis of the Viterbi decoder we had to alter the model for the D flip-flop. This was because the original flip-flop model contained syntax (multiple wait statements in an always statement) that was acceptable to the simulation tool but not by the synthesis tool. This example was artificial because we had already prepared and tested the Verilog code so that it was acceptable to the synthe-

sis software (we say we created **synthesizable** code). However, finding ourselves with nonsynthesizable code arises frequently in logic synthesis. The original OVI LRM included a **synthesis policy**, a set of guidelines that outline which parts of the Verilog language a synthesis tool should support and which parts are optional. Some EDA vendors call their synthesis policy a **modeling style**. There is no current standard on which parts of an HDL (either Verilog or VHDL) a synthesis tool should support.

It is essential that the structural model created by a synthesis tool is **functionally identical**, or **functionally equivalent**, to your behavioral model. Hopefully, we know this is true if the synthesis tool is working properly. In this case the logic is “correct by construction.” If you use different HDL code for simulation and for synthesis, you have a problem. The process of **formal verification** can prove that two logic descriptions (perhaps structural and behavioral HDL descriptions) are identical in their behavior. We shall return to this issue in Chapter 13.

Next we shall examine Verilog and VHDL from the following viewpoint: “How do I write synthesizable code?”

12.5.2 Delays in Verilog

Synthesis tools ignore delay values. They must—how can a synthesis tool guarantee that logic will have a certain delay? For example, a synthesizer cannot generate hardware to implement the following Verilog code:

```

module Step_Time(clk, phase); //1
  input clk; output [2:0] phase; reg [2:0] phase; //2
  always @(posedge clk) begin //3
    phase <= 4'b0000; //4
    phase <= #1 4'b0001; phase <= #2 4'b0010; //5
    phase <= #3 4'b0011; phase <= #4 4'b0100; //6
  end; //7
endmodule //8

```

We can avoid this type of timing problem by dividing a clock as follows:

```

module Step_Count (clk_5x, phase); //1
  input clk_5x; output [2:0] phase; reg [2:0] phase; //2
  always@(posedge clk_5x) //3
  case (phase) //4
    0:phase = #1 1; 1:phase = #1 2; 2:phase = #1 3; 3:phase = #1 4; //5
    default: phase = #1 0; //6
  endcase //7
endmodule //8

```

12.5.3 Blocking and Nonblocking Assignments

There are some synthesis limitations that arise from the different types of Verilog assignment statements. Consider the following shift-register model:

```
module race(clk, q0); input clk, q0; reg q1, q2;
always @(posedge clk) q1 = #1 q0; always @(posedge clk) q2 = #1 q1;
endmodule
```

This example has a **race condition** (or a **race**) that occurs as follows. The synthesizer ignores delays and the two `always` statements are procedures that execute concurrently. So, do we update `q1` first and then assign the new value of `q1` to `q2`? or do we update `q2` first (with the old value of `q1`), and then update `q1`? In real hardware two signals would be racing each other—and the winner is unclear. We must think like the hardware to guide the synthesis tool. Combining the assignment statements into a single `always` statement, as follows, is one way to solve this problem:

```
module no_race_1(clk, q0, q2); input clk, q0; output q2; reg q1, q2;
always @(posedge clk) begin q2 = q1; q1 = q0; end
endmodule
```

Evaluation is sequential within an `always` statement, and the order of the assignment statements now ensures `q2` gets the old value of `q1`—before we update `q1`.

We can also avoid the problem if we use nonblocking assignment statements,

```
module no_race_2(clk, q0, q2); input clk, q0; output q2; reg q1, q2;
always @(posedge clk) q1 <= #1 q0; always @(posedge clk) q2 <= #1 q1;
endmodule
```

This code updates all the registers together, at the end of a time step, so `q2` always gets the old value of `q1`.

12.5.4 Combinational Logic in Verilog

To model combinational logic, the sensitivity list of a Verilog `always` statement must contain only signals with no edges (no reference to keywords `posedge` or `negedge`). This is a **level-sensitive** sensitivity list—as in the following example that implies a two-input AND gate:

```
module And_Always(x, y, z); input x,y; output z; reg z;
always @(x or y) z <= x & y; // combinational logic method 1
endmodule
```

Continuous assignment statements also imply combinational logic (notice that `z` is now a wire rather than a reg),

```
module And_Assign(x, y, z); input x,y; output z; wire z;
assign z <= x & y; // combinational logic method 2 = method 1
endmodule
```


We may also use concatenation or bit reduction to synthesize combinational logic functions,

```
module And_Or (a,b,c,z); input a,b,c; output z; reg [1:0]z;
always @(a or b or c) begin z[1]<= &{a,b,c}; z[2]<= |{a,b,c}; end
endmodule

module Parity (BusIn, outp); input[7:0] BusIn; output outp; reg outp;
always @(BusIn) if (^BusIn == 0) outp = 1; else outp = 0;
endmodule
```

The number of inputs, the types, and the drive strengths of the synthesized combinational logic cells will depend on the speed, area, and load requirements that you set as constraints.

You must be careful if you reference a signal (reg or wire) in a level-sensitive always statement and do not include that signal in the sensitivity list. In the following example, signal b is missing from the sensitivity list, and so this code should be flagged with a warning or an error by the synthesis tool—even though the code is perfectly legal and acceptable to the Verilog simulator:

```
module And_Bad(a, b, c); input a, b; output c; reg c;
always@(a) c <= a & b; // b is missing from this sensitivity list
endmodule
```

It is easy to write Verilog code that will simulate, but that does not make sense to the synthesis software. You must think like the hardware. To avoid this type of problem with combinational logic inside an always statement you should either:

- include all variables in the event expression or
- assign to the variables before you use them

For example, consider the following two models:

```
module CL_good(a, b, c); input a, b; output c; reg c;
always@(a or b)
begin c = a + b; d = a & b; e = c + d; end // c, d: LHS before RHS
endmodule

module CL_bad(a, b, c); input a, b; output c; reg c;
always@(a or b)
begin e = c + d; c = a + b; d = a & b; end // c, d: RHS before LHS
endmodule
```

In `CL_bad`, the signals c and d are used on the right-hand side (RHS) of an assignment statement before they are defined on the left-hand side (LHS) of an assignment statement. If the logic synthesizer produces combinational logic for `CL_bad`, it should warn us that the synthesized logic may not match the simulation results.

When you are describing combinational logic you should be aware of the complexity of logic optimization. Some combinational logic functions are too difficult for the optimization algorithms to handle. The following module, `Achilles`, and

large parity functions are examples of hard-to-synthesize functions. This is because most logic-optimization algorithms calculate the complement of the functions at some point. The complements of certain functions grow exponentially in the number of their product terms.

```
// The complement of this function is too big for synthesis.
module Achilles (out, in); output out; input [30:1] in;
assign out =      in[30]&in[29]&in[28] | in[27]&in[26]&in[25]
                  | in[24]&in[23]&in[22] | in[21]&in[20]&in[19]
                  | in[18]&in[17]&in[16] | in[15]&in[14]&in[13]
                  | in[12]&in[11]&in[10] | in[9] & in[8]&in[7]
                  | in[6] & in[5]&in[4]  | in[3] & in[2]&in[1];
endmodule
```

In a case like this you can isolate the problem function in a separate module. Then, after synthesis, you can use directives to tell the synthesizer not to try and optimize the problem function.

12.5.5 Multiplexers In Verilog

We imply a MUX using a case statement, as in the following example:

```
module Mux_21a(sel, a, b, z); input sel, a, b; output z; reg z;
always @(a or b or sel)
begin case(sel) 1'b0: z <= a; 1'b1: z <= b; end
endmodule
```

Be careful using 'x' in a case statement. Metalogical values (such as 'x') are not “real” and are only valid in simulation (and they are sometimes known as **simbits** for that reason). For example, a synthesizer cannot make logic to model the following and will usually issue a warning to that effect:

```
module Mux_x(sel, a, b, z); input sel, a, b; output z; reg z;
always @(a or b or sel)
begin case(sel) 1'b0: z <= 0; 1'b1: z <= 1; 1'bx: z <= 'x'; end
endmodule
```

For the same reason you should avoid using `casex` and `casez` statements.

An `if` statement can also be used to imply a MUX as follows:

```
module Mux_21b(sel, a, b, z); input sel, a, b; output z; reg z;
always @(a or b or sel) begin if (sel) z <= a else z <= b; end
endmodule
```

However, if you do not always assign to an output, as in the following code, you will get a latch:

```
module Mux_Latch(sel, a, b, z); input sel, a, b; output z; reg z;
always @(a or sel) begin if (sel) z <= a; end
endmodule
```

It is important to understand why this code implies a sequential latch and not a combinational MUX. Think like the hardware and you will see the problem. When `sel` is zero, you can pass through the `always` statement whenever a change occurs on the input `a` without updating the value of the output `z`. In this situation you need to “remember” the value of `z` when `a` changes. This implies sequential logic using `a` as the latch input, `sel` as the active-high latch enable, and `z` as the latch output.

The following code implies an 8:1 MUX with a three-state output:

```
module Mux_81(InBus, sel, OE, OutBit); //1
input [7:0] InBus; input [2:0] Sel; //2
input OE; output OutBit; reg OutBit; //3
always @(OE or sel or InBus) //4
begin //5
if (OE == 1) OutBit = InBus[sel]; else OutBit = 1'bz; //6
end //7
endmodule //8
```

When you synthesize a large MUX the required speed and area, the output load, as well as the cells that are available in the cell library will determine whether the synthesizer uses a large MUX cell, several smaller MUX cells, or equivalent random logic cells. The synthesized logic may also use different logic cells depending on whether you want the fastest path from the select input to the MUX output or from the data inputs to the MUX output.

12.5.6 The Verilog Case Statement

Consider the following model:

```
module case8_oneHot(oneHot, a, b, c, z); //1
input a, b, c; input [2:0] oneHot; output z; reg z; //2
always @(oneHot or a or b or c) //3
begin case(oneHot) //synopsys full_case //4
3'b001: z <= a; 3'b010: z <= b; 3'b100: z <= c; //5
default: z <= 1'bx; endcase //6
end //7
endmodule //8
```

By including the `default` choice, the case statement is **exhaustive**. This means that every possible value of the select variable (`oneHot`) is accounted for in the arms of the case statement. In some synthesizers (Synopsys, for example) you may indicate the arms are exhaustive and imply a MUX by using a **compiler directive** or **synthesis directive**. A compiler directive is also called a **pseudocomment** if it uses

the comment format (such as `//synopsys full_case`). The format of pseudocomments is very specific. Thus, for example, `//synopsys` may be recognized but `// synopsys` (with an extra space) or `//SynopSys` (uppercase) may not. The use of pseudocomments shows the problems of using an HDL for a purpose for which it was not intended. When we start “extending” the language we lose the advantages of a standard and sacrifice portability. A compiler directive in module `case8_oneHot` is unnecessary if the `default` choice is included. If you omit the `default` choice and you do not have the ability to use the `full_case` directive (or you use a different tool), the synthesizer will infer latches for the output `z`.

If the default in a case statement is `'x'` (signifying a **synthesis don't care value**), this gives the synthesizer flexibility in optimizing the logic. It does not mean that the synthesized logic output will be unknown when the default applies. The combinational logic that results from a case statement when a don't care (`'x'`) is included as a default may or may not include a MUX, depending on how the logic is optimized.

In `case8_oneHot` the choices in the arms of the case statement are exhaustive and also mutually exclusive. Consider the following alternative model:

```

module case8_priority(oneHot, a, b, c, z);           //1
input a, b, c; input [2:0] oneHot; output z; reg z; //2
always @(oneHot or a or b or c) begin             //3
case(1'b1) //synopsys parallel_case               //4
    oneHot[0]: z <= a;                             //5
    oneHot[1]: z <= b;                             //6
    oneHot[2]: z <= c;                             //7
    default: z <= 1'bx; endcase                    //8
end                                                 //9
endmodule                                         //10

```

In this version of the case statement the choices are not necessarily mutually exclusive (`oneHot[0]` and `oneHot[2]` may both be equal to `1'b1`, for example). Thus the code implies a priority encoder. This may not be what you intended. Some logic synthesizers allow you to indicate mutually exclusive choices by using a directive (`//synopsys parallel_case`, for example). It is probably wiser not to use these “outside-the-language” directives if they can be avoided.

12.5.7 Decoders In Verilog

The following code models a 4:16 decoder with enable and three-state output:

```

module Decoder_4To16(enable, In_4, Out_16); // 4-to-16 decoder //1
input enable; input [3:0] In_4; output [15:0] Out_16; //2
reg [15:0] Out_16; //3
always @(enable or In_4) //4
    begin Out_16 = 16'hzzzz; //5
        if (enable == 1) //6
            begin Out_16 = 16'h0000; Out_16[In_4] = 1; end //7
    end
endmodule

```

```

    end //8
endmodule //9

```

In line 7 the binary-encoded 4-bit input sets the corresponding bit of the 16-bit output to '1'. The synthesizer infers a three-state buffer from the assignment in line 5. Using the equality operator, '==', rather than the case equality operator, '===', makes sense in line 6, because the synthesizer cannot generate logic that will check for enable being 'x' or 'z'. So, for example, do not write the following (though some synthesis tools will still accept it):

```
if (enable === 1) // can't make logic to check for enable = x or z
```

12.5.8 Priority Encoder in Verilog

The following Verilog code models a priority encoder with three-state output:

```

module Pri_Encoder32 (InBus, Clk, OE, OutBus); //1
input [31:0]InBus; input OE, Clk; output [4:0]OutBus; //2
reg j; reg [4:0]OutBus; //3
    always@(posedge Clk) //4
    begin //5
        if (OE == 0) OutBus = 5'bz ; //6
    else //7
        begin OutBus = 0; //8
            for (j = 31; j >= 0; j = j - 1) //9
                begin if (InBus[j] == 1) OutBus = j; end //10
            end //11
        end //12
    endmodule //13

```

In lines 9–11 the binary-encoded output is set to the position of the lowest-indexed '1' in the input bus. The logic synthesizer must be able to unroll the loop in a for statement. Normally the synthesizer will check for fixed (or static) bounds on the loop limits, as in line 9 above.

12.5.9 Arithmetic in Verilog

You need to make room for the carry bit when you add two numbers in Verilog. You may do this using concatenation on the LHS of an assignment as follows:

```

module Adder_8 (A, B, Z, Cin, Cout); //1
input [7:0] A, B; input Cin; output [7:0] Z; output Cout; //2
assign {Cout, Z} = A + B + Cin; //3
endmodule //4

```

In the following example, the synthesizer should recognize '1' as a carry-in bit of an adder and should synthesize one adder and not two:

```

module Adder_16 (A, B, Sum, Cout); //1
input [15:0] A, B; output [15:0] Sum; output Cout; //2

```

```

reg [15:0] Sum; reg Cout; //3
always @(A or B) {Cout, Sum} = A + B + 1; //4
endmodule //5

```

It is always possible to synthesize adders (and other arithmetic functions) using random logic, but they may not be as efficient as using datapath synthesis (see Section 12.5.12).

A logic synthesizer may infer two adders from the following description rather than shaping a single adder.

```

module Add_A (sel, a, b, c, d, y); //1
input a, b, c, d, sel; output y; reg y; //2
always@(sel or a or b or c or d) //3
    begin if (sel == 0) y <= a + b; else y <= c + d; end //4
endmodule //5

```

To imply the presence of a MUX before a single adder we can use temporary variables. For example, the synthesizer should use only one adder for the following code:

```

module Add_B (sel, a, b, c, d, y); //1
input a, b, c, d, sel; output y; reg t1, t2, y; //2
always@(sel or a or b or c or d) begin //3
    if (sel == 0)    begin t1 = a; t2 = b; end // Temporary //4
    else            begin t1 = c; t2 = d; end // variables. //5
    y = t1 + t2; end //6
endmodule //7

```

If a synthesis tool is capable of performing **resource allocation** and **resource sharing** in these situations, the coding style may not matter. However we may want to use a different tool, which may not be as advanced, at a later date—so it is better to use Add_B rather than Add_A if we wish to conserve area. This example shows that the simplest code (Add_A) does not always result in the simplest logic (Add_B).

Multiplication in Verilog assumes nets are unsigned numbers:

```

module Multiply_unsigned (A, B, Z); //1
input [1:0] A, B; output [3:0] Z; //2
assign Z <= A * B; //3
endmodule //4

```

To multiply signed numbers we need to extend the multiplicands with their sign bits as follows (some simulators have trouble with the concatenation '{ }' structures, in which case we have to write them out “long hand”):

```

module Multiply_signed (A, B, Z); //1
input [1:0] A, B; output [3:0] Z; //2
// 00 -> 00_00  01 -> 00_01  10 -> 11_10  11 -> 11_11 //3
assign Z = { { 2{A[1]} }, A} * { { 2{B[1]} }, B}; //4
endmodule //5

```

How the logic synthesizer implements the multiplication depends on the software.

12.5.10 Sequential Logic in Verilog

The following statement implies a positive-edge-triggered D flip-flop:

```
always@(posedge clock) Q_flipflop = D; // A flip-flop.
```

When you use edges (posedge or negedge) in the sensitivity list of an always statement, you imply a clocked storage element. However, an always statement does not have to be edge-sensitive to imply sequential logic. As another example of sequential logic, the following statement implies a level-sensitive transparent latch:

```
always@(clock or D) if (clock) Q_latch = D; // A latch.
```

On the negative edge of the clock the always statement is executed, but no assignment is made to `Q_latch`. These last two code examples concisely illustrate the difference between a flip-flop and a latch.

Any sequential logic cell or memory element must be initialized. Although you could use an initial statement to simulate power-up, generating logic to mimic an initial statement is hard. Instead use a reset as follows:

```
always@(posedge clock or negedge reset)
```

A problem now arises. When we use two edges, the synthesizer must infer which edge is the clock, and which is the reset. Synthesis tools cannot read any significance into the names we have chosen. For example, we could have written

```
always@(posedge day or negedge year)
```

—but which is the clock and which is the reset in this case?

For most synthesis tools you must solve this problem by writing HDL code in a certain format or pattern so that the logic synthesizer may correctly infer the clock and reset signals. The following examples show one possible pattern or **template**. These templates and their use are usually described in a **synthesis style guide** that is part of the synthesis software documentation.

```
always@(posedge clk or negedge reset) begin // template for reset:
    if (reset == 0) Q = 0; // initialize,
    else Q = D; // normal clocking
end

module Counter_With_Reset (count, clock, reset); //1
input clock, reset; output count; reg [7:0] count; //2
always @ (posedge clock or negedge reset) //3
    if (reset == 0) count = 0; else count = count + 1; //4
endmodule //5

module DFF_MasterSlave (D, clock, reset, Q); // D type flip-flop //1
input D, clock, reset; output Q; reg Q, latch; //2
always @(posedge clock or posedge reset) //3
    if (reset == 1) latch = 0; else latch = D; // the master. //4
```

```

always @(latch) Q = latch; // the slave.           //5
endmodule                                         //6

```

The synthesis tool can now infer that, in these templates, the signal that is tested in the `if` statement is the reset, and that the other signal must therefore be the clock.

12.5.11 Component Instantiation in Verilog

When we give an HDL description to a synthesis tool, it will synthesize a netlist that contains generic logic gates. By generic we mean the logic is technology-independent (it could be CMOS standard cell, FPGA, TTL, GaAs, or something else—we have not decided yet). Only after logic optimization and mapping to a specific ASIC cell library do the speed or area constraints determine the cell choices from a cell library: NAND gates, OAI gates, and so on.

The only way to ensure that the synthesizer uses a particular cell, 'special' for example, from a specific library is to write structural Verilog and instantiate the cell, 'special', in the Verilog. We call this **hand instantiation**. We must then decide whether to allow logic optimization to replace or change 'special'. If we insist on using logic cell 'special' and do not want it changed, we flag the cell with a synthesizer command. Most logic synthesizers currently use a pseudocomment statement or set an attribute to do this.

For example, we might include the following statement to tell the Compass synthesizer—"Do not change cell instance `my_inv_8x`." This is not a standard construct, and it is not portable from tool to tool either.

```

//Compass dontTouch my_inv_8x or // synopsys dont_touch
INVD8 my_inv_8x(.I(a), .ZN(b) );

```

(some compiler directives are trademarks). Notice, in this example, instantiation involves declaring the instance name and defining a structural port mapping.

There is no standard name for technology-independent models or components—we shall call them **soft models** or **standard components**. We can use the standard components for synthesis or for behavioral Verilog simulation. Here is an example of using standard components for flip-flops (remember there are no primitive Verilog flip-flop models—only primitives for the elementary logic cells):

```

module Count4(clk, reset, Q0, Q1, Q2, Q3);           //1
input clk, reset; output Q0, Q1, Q2, Q3; wire Q0, Q1, Q2, Q3; //2
//          Q , D , clk, reset                       //3
asDff dff0( Q0, ~Q0, clk, reset); // The asDff is a //4
asDff dff1( Q1, ~Q1, Q0, reset); // standard component, //5
asDff dff2( Q2, ~Q2, Q1, reset); // unique to one set of tools. //6
asDff dff3( Q3, ~Q3, Q2, reset); //7
endmodule                                           //8

```

The `asDff` and other standard components are provided with the synthesis tool. The standard components have specific names and interfaces that are part of the

software documentation. When we use a standard component such as `asDff` we are saying: “I want a D flip-flop, but I do not know which ASIC technology I want to use—give me a generic version. I do not want to write a Verilog model for the D flip-flop myself because I do not want to bother to synthesize each and every instance of a flip-flop. When the time comes, just map this generic flip-flop to whatever is available in the technology-dependent (vendor-specific) library.”

If we try and simulate `Count4` we will get an error,

```
:Count4.v: L5: error: Module 'asDff' not defined
```

(and three more like this) because `asDff` is not a primitive Verilog model. The synthesis tool should provide us with a model for the standard component. For example, the following code models the behavior of the standard component, `asDff`:

```
module asDff (D, Q, Clk, Rst); //1
parameter width = 1, reset_value = 0; //2
input [width-1:0] D; output [width-1:0] Q; reg [width-1:0] Q; //3
input Clk,Rst; initial Q = {width{1'bx}}; //4
always @ ( posedge Clk or negedge Rst ) //5
    if ( Rst==0 ) Q <= #1 reset_value; else Q <= #1 D; //6
endmodule //7
```

When the synthesizer compiles the HDL code in `Count4`, it does not parse the `asDff` model. The software recognizes `asDff` and says “I see you want a flip-flop.” The first steps that the synthesis software and the simulation software take are often referred to as compilation, but the two steps are different for each of these tools.

Synopsys has an extensive set of libraries, called **DesignWare**, that contains standard components not only for flip-flops but for arithmetic and other complex logic elements. These standard components are kept protected from optimization until it is time to map to a vendor technology. ASIC or EDA companies that produce design software and cell libraries can tune the synthesizer to the silicon and achieve a more efficient mapping. Even though we call them standard components, there are no standards that cover their names, use, interfaces, or models.

12.5.12 Datapath Synthesis in Verilog

Datapath synthesis is used for bus-wide arithmetic and other bus-wide operations. For example, synthesis of a 32-bit multiplier in random logic is much less efficient than using datapath synthesis. There are several approaches to datapath synthesis:

- Synopsys VHDL DesignWare. This models generic arithmetic and other large functions (counters, shift registers, and so on) using standard components. We can either let the synthesis tool map operators (such as '+') to VHDL DesignWare components, or we can hand instantiate them in the code. Many ASIC vendors support the DesignWare libraries. Thus, for exam-

ple, we can instantiate a DesignWare counter in VHDL and map that to a cell predesigned and preoptimized by Actel for an Actel FPGA.

- Compiler directives. This approach uses synthesis directives in the code to steer the mapping of datapath operators either to specific components (a two-port RAM or a register file, for example) or flags certain operators to be implemented using a certain style ('+' to be implemented using a ripple-carry adder or a carry-lookahead adder, for example).
- X-BLOX is a system from Xilinx that allows us to keep the logic of certain functions (counters, arithmetic elements) together. This is so that the layout tool does not splatter the synthesized CLBs all over your FPGA, reducing the performance of the logic.
- LPM (library of parameterized modules) and RPM (relationally placed modules) are other techniques used principally by FPGA companies to keep logic that operates on related data close together. This approach is based on the use of the EDIF language to describe the modules.

In all cases the disadvantage is that the code becomes specific to a certain piece of software. Here are two examples of datapath synthesis directives:

```

module DP_csum(A1,B1,Z1); input [3:0] A1,B1; output Z1; reg [3:0] Z1;
always@(A1 or B1) Z1 <= A1 + B1;//Compass adder_arch cond_sum_add
endmodule

module DP_ripp(A2,B2,Z2); input [3:0] A2,B2; output Z2; reg [3:0] Z2;
always@(A2 or B2) Z2 <= A2 + B2;//Compass adder_arch ripple_add
endmodule

```

These directives steer the synthesis of a conditional-sum adder (usually the fastest adder implementation) or a ripple-carry adder (small but slow).

There are some limitations to datapath synthesis. Sometimes, complex operations are not synthesized as we might expect. For example, a datapath library may contain a subtracter that has a carry input; however, the following code may synthesize to random logic, because the synthesizer may not be able to infer that the signal CarryIn is a subtracter carry:

```

module DP_sub_A(A,B,OutBus,CarryIn); //1
input [3:0] A, B ; input CarryIn ; //2
output OutBus ; reg [3:0] OutBus ; //3
always@(A or B or CarryIn) OutBus <= A - B - CarryIn ; //4
endmodule //5

```

If we rewrite the code and subtract the carry as a constant, the synthesizer can more easily infer that it should use the carry-in of a datapath subtracter:

```

module DP_sub_B (A, B, CarryIn, Z) ; //1
input [3:0] A, B, CarryIn ; output [3:0] Z; reg [3:0] Z; //2
always@(A or B or CarryIn) begin //3
    case (CarryIn) //4

```

```

    1'b1 :      Z <= A - B - 1'b1;           //5
    default :  Z <= A - B - 1'b0; endcase    //6
end                                               //7
endmodule                                       //8

```

This is another example of thinking like the hardware in order to help the synthesis tool infer what we are trying to imply.

12.6 VHDL and Logic Synthesis

Most logic synthesizers insist we follow a set of rules when we use a logic system to ensure that what we synthesize matches the behavioral description. Here is a typical set of rules for use with the IEEE VHDL nine-value system:

- You can use logic values corresponding to states '1', 'H', '0', and 'L' in any manner.
- Some synthesis tools do not accept the uninitialized logic state 'U'.
- You can use logic states 'Z', 'X', 'W', and '-' in signal and variable assignments in any manner. 'Z' is synthesized to three-state logic.
- The states 'X', 'W', and '-' are treated as unknown or don't care values.

The values 'Z', 'X', 'W', and '-' may be used in conditional clauses such as the comparison in an `if` or `case` statement. However, some synthesis tools will ignore them and only match surrounding '1' and '0' bits. Consequently, a synthesized design may behave differently from the simulation if a stimulus uses 'Z', 'X', 'W' or '-'. The IEEE synthesis packages provide the `STD_MATCH` function for comparisons.

12.6.1 Initialization and Reset

You can use a VHDL `process` with a sensitivity list to synthesize clocked logic with a reset, as in the following code:

```

process (signal_1, signal_2) begin
    if (signal_2'EVENT and signal_2 = '0')
        then -- Insert initialization and reset statements.
    elsif (signal_1'EVENT and signal_1 = '1')
        then -- Insert clocking statements.
    end if;
end process;

```

Using a specific pattern the synthesizer can infer that you are implying a positive-edge clock (`signal_1`) and a negative-edge reset (`signal_2`). In order to be able to recognize sequential logic in this way, most synthesizers restrict you to using a maximum of two edges in a sensitivity list.

12.6.2 Combinational Logic Synthesis in VHDL

In VHDL a **level-sensitive process** is a process statement that has a sensitivity list with signals that are not tested for event attributes ('EVENT or 'STABLE, for example) within the process. To synthesize combinational logic we use a VHDL level-sensitive process or a concurrent assignment statement. Some synthesizers do not allow reference to a signal inside a level-sensitive process unless that signal is in the sensitivity list. In this example, signal b is missing from the sensitivity list:

```
entity And_Bad is port (a, b: in BIT; c: out BIT); end And_Bad;

architecture Synthesis_Bad of And_Bad is
  begin process (a) -- this should be process (a, b)
    begin c <= a and b;
    end process;
end Synthesis_Bad;
```

This situation is similar but not exactly the same as omitting a variable from an event control in a Verilog always statement. Some logic synthesizers accept the VHDL version of And_Bad but not the Verilog version or vice versa. To ensure that the VHDL simulation will match the behavior of the synthesized logic, the logic synthesizer usually checks the sensitivity list of a level-sensitive process and issues a warning if signals seem to be missing.

12.6.3 Multiplexers in VHDL

Multiplexers can be synthesized using a case statement (avoiding the VHDL reserved word 'select'), as the following example illustrates:

```
entity Mux4 is port
(i: BIT_VECTOR(3 downto 0); sel: BIT_VECTOR(1 downto 0); s: out BIT);
end Mux4;

architecture Synthesis_1 of Mux4 is
  begin process(sel, i) begin
    case sel is
      when "00" => s <= i(0); when "01" => s <= i(1);
      when "10" => s <= i(2); when "11" => s <= i(3);
    end case;
  end process;
end Synthesis_1;
```

The following code, using a concurrent signal assignment is equivalent:

```
architecture Synthesis_2 of Mux4 is
  begin with sel select s <=
    i(0) when "00", i(1) when "01", i(2) when "10", i(3) when "11";
end Synthesis_2;
```

In VHDL the case statement must be exhaustive in either form, so there is no question of any priority in the choices as there may be in Verilog.

For larger MUXes we can use an array, as in the following example:

```
library IEEE; use ieee.std_logic_1164.all;
entity Mux8 is port
  (InBus : in STD_LOGIC_VECTOR(7 downto 0);
   Sel : in INTEGER range 0 to 7;
   OutBit : out STD_LOGIC);
end Mux8;

architecture Synthesis_1 of Mux8 is
  begin process(InBus, Sel)
    begin OutBit <= InBus(Sel);
  end process;
end Synthesis_1;
```

Most synthesis tools can infer that, in this case, Sel requires three bits. If not, you have to declare the signal as a STD_LOGIC_VECTOR,

```
Sel : in STD_LOGIC_VECTOR(2 downto 0);
```

and use a conversion routine from the STD_NUMERIC package like this:

```
OutBit <= InBus(TO_INTEGER ( UNSIGNED (Sel) ) ) ;
```

At some point you have to convert from an INTEGER to BIT logic anyway, since you cannot connect an INTEGER to the input of a chip! The VHDL case, if, and select statements produce similar results. Assigning don't care bits ('x') in these statements will make it easier for the synthesizer to optimize the logic.

12.6.4 Decoders in VHDL

The following code implies a decoder:

```
library IEEE; --1
use IEEE.STD_LOGIC_1164.all; use IEEE.NUMERIC_STD.all; --2
entity Decoder is port (enable : in BIT; --3
  Din: STD_LOGIC_VECTOR (2 downto 0); --4
  Dout: out STD_LOGIC_VECTOR (7 downto 0)); --5
end Decoder; --6

architecture Synthesis_1 of Decoder is --7
  begin --8
  with enable select Dout <= --9
  STD_LOGIC_VECTOR --10
  (UNSIGNED' --11
  (shift_left --12
  ("00000001", TO_INTEGER (UNSIGNED(Din)) --13
  ) --14
```

```

    )
  )
  when '1',
    "11111111" when '0', "00000000" when others;
end Synthesis_1;

```

There are reasons for this seemingly complex code:

- Line 1 declares the IEEE library. The synthesizer does not parse the VHDL code inside the library packages, but the synthesis company should be able to guarantee that the logic will behave exactly the same way as predicted by a simulator that uses the IEEE libraries and does parse the code.
- Line 2 declares the STD_LOGIC_1164 package, for STD_LOGIC types, and the NUMERIC_STD package for conversion and shift functions. The shift *operators* (sll and so on—the infix operators) were introduced in VHDL-93, they are not defined for STD_LOGIC types in the 1164 standard. The shift *functions* defined in NUMERIC_STD are not operators and are called shift_left and so on. Some synthesis tools support NUMERIC_STD, but not VHDL-93.
- Line 10 performs a type conversion to STD_LOGIC_VECTOR from UNSIGNED.
- Line 11 is a type qualification to tell the software that the argument to the type conversion function is type UNSIGNED.
- Line 12 is the shift function, shift_left, from the NUMERIC_STD package.
- Line 13 converts the STD_LOGIC_VECTOR, Din, to UNSIGNED before converting to INTEGER. We cannot convert directly from STD_LOGIC_VECTOR to INTEGER.
- The others clause in line 18 is required by the logic synthesizer even though type BIT may only be '0' or '1'.

If we model a decoder using a process, we can use a case statement inside the process. A MUX model may be used as a decoder if the input bits are set at '1' (active-high decoder) or at '0' (active-low decoder), as in the following example:

```

library IEEE;
use IEEE.NUMERIC_STD.all; use IEEE.STD_LOGIC_1164.all;

entity Concurrent_Decoder is port (
  enable : in BIT;
  Din : in STD_LOGIC_VECTOR (2 downto 0);
  Dout : out STD_LOGIC_VECTOR (7 downto 0));
end Concurrent_Decoder;

architecture Synthesis_1 of Concurrent_Decoder is
begin process (Din, enable)
  variable T : STD_LOGIC_VECTOR(7 downto 0);
begin
  if (enable = '1') then

```

```

    T := "00000000"; T( TO_INTEGER (UNSIGNED(Din))) := '1';           --13
    Dout <= T ;                                                       --14
  else Dout <= (others => 'Z');                                       --15
  end if;                                                             --16
end process;                                                         --17
end Synthesis_1;                                                    --18

```

Notice that `T` must be a variable for proper timing of the update to the output. The `else` clause in the `if` statement is necessary to avoid inferring latches.

12.6.5 Adders in VHDL

To add two n -bit numbers and keep the overflow bit, we need to assign to a signal or variable with more bits, as follows:

```

library IEEE;                                                         --1
use IEEE.NUMERIC_STD.all; use IEEE.STD_LOGIC_1164.all;              --2

entity Adder_1 is                                                    --3
  port (A, B: in UNSIGNED(3 downto 0); C: out UNSIGNED(4 downto 0)); --4
end Adder_1;                                                         --5

architecture Synthesis_1 of Adder_1 is                               --6
  begin C <= ('0' & A) + ('0' & B);                                   --7
end Synthesis_1;                                                    --8

```

Notice that both `A` and `B` have to be `SIGNED` or `UNSIGNED` as we cannot add `STD_LOGIC_VECTOR` types directly using the IEEE 1164 packages. You will get an error if a result is a different length from the target of an assignment, as in the following example (in which the arguments are not resized):

```

adder_1:      begin C <= A + B;
Error: Width mis-match: right expression is 4 bits wide, c is 5 bits
wide

```

The following code may generate three adders stacked three deep:

```
z <= a + b + c + d;
```

Depending on how the expression is parsed, the first adder may perform $x = a + b$, a second adder $y = x + c$, and a third adder $z = y + d$. The following code should generate faster logic with three adders stacked only two deep:

```
z <= (a + b) + (c + d);
```

12.6.6 Sequential Logic in VHDL

Sensitivity to an edge implies sequential logic in VHDL. A synthesis tool can locate edges in VHDL by finding a process statement that has either:

- no sensitivity list with a `wait until` statement
- a sensitivity list and test for `'EVENT` plus a specific level

Any signal assigned in an edge-sensitive process statement should also be reset—but be careful to distinguish between asynchronous and synchronous resets. The following example illustrates these points:

```

library IEEE; use IEEE.STD_LOGIC_1164.all; entity DFF_With_Reset is
  port(D, Clk, Reset : in STD_LOGIC; Q : out STD_LOGIC);
end DFF_With_Reset;

architecture Synthesis_1 of DFF_With_Reset is
  begin process(Clk, Reset) begin
    if (Reset = '0') then Q <= '0'; -- asynchronous reset
    elsif rising_edge(Clk) then Q <= D;
    end if;
  end process;
end Synthesis_1;

architecture Synthesis_2 of DFF_With_Reset is
  begin process begin
    wait until rising_edge(Clk);
    -- This reset is gated with the clock and is synchronous:
    if (Reset = '0') then Q <= '0'; else Q <= D; end if;
  end process;
end Synthesis_2;

```

Sequential logic results when we have to “remember” something between successive executions of a process statement. This occurs when a process statement contains one or more of the following situations:

- A signal is read but is not in the sensitivity list of a process statement.
- A signal or variable is read before it is updated.
- A signal is not always updated.
- There are multiple wait statements.

Not all of the models that we could write using the above constructs will be synthesizable. Any models that do use one or more of these constructs and that are synthesizable will result in sequential logic.

12.6.7 Instantiation in VHDL

The easiest way to find out how to hand instantiate a component is to generate a structural netlist from a simple HDL input—for example, the following Verilog behavioral description (VHDL could have been used, but the Verilog is shorter):

```

`timescale 1ns/1ns //1
module halfgate (myInput, myOutput); //2
input myInput; output myOutput; wire myOutput; //3
  assign myOutput = ~myInput; //4
endmodule //5

```


We synthesize this module and generate the following VHDL structural netlist:

```

library IEEE; use IEEE.STD_LOGIC_1164.all;           --1
library COMPASS_LIB; use COMPASS_LIB.COMPASS.all;    --2
--compass compile_off -- synopsys etc.             --3
use COMPASS_LIB.COMPASS_ETC.all;                   --4
--compass compile_on -- synopsys etc.              --5
entity halfgate_u is                                --6
--compass compile_off -- synopsys etc.             --7
generic (                                           --8
    myOutput_cap : Real := 0.01;                   --9
    INSTANCE_NAME : string := "halfgate_u" );      --10
--compass compile_on -- synopsys etc.              --11
port ( myInput : in Std_Logic := 'U';              --12
myOutput : out Std_Logic := 'U' );                 --13
end halfgate_u;                                     --14

architecture halfgate_u of halfgate_u is           --15
component in01d0                                    --16
port ( I : in Std_Logic; ZN : out Std_Logic ); end component; --17
begin                                               --18
u2: in01d0 port map ( I => myInput, ZN => myOutput ); --19
end halfgate_u;                                     --20

--compass compile_off -- synopsys etc.             --21
library cb60hd230d;                                 --22
configuration halfgate_u_CON of halfgate_u is      --23
    for halfgate_u                                   --24
        for u2 : in01d0 use configuration cb60hd230d.in01d0_CON --25
            generic map (                            --26
                ZN_cap => 0.0100 + myOutput_cap,    --27
                INSTANCE_NAME => INSTANCE_NAME&"/u2" ) --28
            port map ( I => I, ZN => ZN);            --29
        end for;                                     --30
    end for;                                         --31
end halfgate_u_CON;                                 --32
--compass compile_on -- synopsys etc.               --33

```

This gives a template to follow when hand instantiating logic cells. Instantiating a standard component requires the name of the component and its parameters:

```

component ASDF
generic (WIDTH : POSITIVE := 1;
    RESET_VALUE : STD_LOGIC_VECTOR := "0" );
port (Q : out STD_LOGIC_VECTOR (WIDTH-1 downto 0);
    D : in STD_LOGIC_VECTOR (WIDTH-1 downto 0);
    CLK : in STD_LOGIC;

```

```

        RST : in STD_LOGIC );
end component;

```

Now you have enough information to be able to instantiate both logic cells from a cell library and standard components. The following model illustrates instantiation:

```

library IEEE, COMPASS_LIB; --1
use IEEE.STD_LOGIC_1164.all; use COMPASS_LIB.STDCOMP.all; --2
entity Ripple_4 is --3
    port (Trig, Reset: STD_LOGIC; QN0_5x: out STD_LOGIC; --4
          Q : inout STD_LOGIC_VECTOR(0 to 3)); --5
end Ripple_4; --6
architecture structure of Ripple_4 is --7
    signal QN : STD_LOGIC_VECTOR(0 to 3); --8
    component in01d1 --9
    port ( I : in Std_Logic; ZN : out Std_Logic ); end component; --10
    component in01d5 --11
    port ( I : in Std_Logic; ZN : out Std_Logic ); end component; --12
begin --13
--compass dontTouch inv5x -- synopsys dont_touch etc. --14
-- Named association for hand-instantiated library cells: --15
    inv5x: IN01D5 port map( I=>Q(0), ZN=>QN0_5x ); --16
    inv0 : IN01D1 port map( I=>Q(0), ZN=>QN(0) ); --17
    inv1 : IN01D1 port map( I=>Q(1), ZN=>QN(1) ); --18
    inv2 : IN01D1 port map( I=>Q(2), ZN=>QN(2) ); --19
    inv3 : IN01D1 port map( I=>Q(3), ZN=>QN(3) ); --20
-- Positional association for standard components: --21
--
--           Q           D           Clk   Rst --22
d0: asDFF port map(Q (0 to 0), QN(0 to 0), Trig, Reset); --23
d1: asDFF port map(Q (1 to 1), QN(1 to 1), Q(0), Reset); --24
d2: asDFF port map(Q (2 to 2), QN(2 to 2), Q(1), Reset); --25
d3: asDFF port map(Q (3 to 3), QN(3 to 3), Q(2), Reset); --26
end structure; --27

```

- Lines 5 and 8. Type `STD_LOGIC_VECTOR` must be used for standard component ports, because the standard components are defined using this type.
- Line 5. Mode `inout` has to be used for `Q` since it has to be read/write and this is a structural model. You cannot use mode `buffer` since the formal outputs of the standard components are declared to be of mode `out`.
- Line 14. This synthesis directive prevents the synthesis tool from removing the 5X drive strength inverter `inv5x`. This statement ties the code to a particular synthesis tool.
- Lines 16–20. Named association for the hand-instantiated library cells. The names (`IN01D5` and `IN01D1`) and port names (`I` and `ZN`) come from the cell library data book or from a template (such as the one created for the `IN01D1` logic cell). These statements tie the code to a particular cell library.

- Lines 23–26. Positional port mapping of the standard components. The port locations are from the synthesis standard component library documentation. These asDFF standard components will be mapped to D flip-flop library cells. These statements tie the code to a particular synthesis tool.

You would receive the following warning from the logic synthesizer when it synthesizes this input code (entity `Ripple_4`):

```
Warning: Net has more than one driver: d3_Q[0]; connected to:
ripple_4_p.q[3], inv3.I, d3.Q
```

There is potentially more than one driver on a net because `Q` was declared as `inout`. There are a total of four warnings of this type for each of the flip-flop outputs. You can check the output netlist to make sure that you have the logic you expected as follows (the Verilog netlist is shorter and easier to read):

```
`timescale 1ns / 10ps //1
module ripple_4_u (trig, reset, qn0_5x, q); //2
input trig; input reset; output qn0_5x; inout [3:0] q; //3
wire [3:0] qn; supply1 VDD; supply0 VSS; //4
in01d5 inv5x (.I(q[0]),.ZN(qn0_5x)); //5
in01d1 inv0 (.I(q[0]),.ZN(qn[0])); //6
in01d1 inv1 (.I(q[1]),.ZN(qn[1])); //7
in01d1 inv2 (.I(q[2]),.ZN(qn[2])); //8
in01d1 inv3 (.I(q[3]),.ZN(qn[3])); //9
dfctnb d0(.D(qn[0]),.CP(trig),.CDN(reset),.Q(q[0]),.QN(\d0.QN )); //10
dfctnb d1(.D(qn[1]),.CP(q[0]),.CDN(reset),.Q(q[1]),.QN(\d1.QN )); //11
dfctnb d2(.D(qn[2]),.CP(q[1]),.CDN(reset),.Q(q[2]),.QN(\d2.QN )); //12
dfctnb d3(.D(qn[3]),.CP(q[2]),.CDN(reset),.Q(q[3]),.QN(\d3.QN )); //13
endmodule //14
```

12.6.8 Shift Registers and Clocking in VHDL

The following code implies a serial-in/parallel-out (SIPO) shift register:

```
library IEEE; --1
use IEEE.STD_LOGIC_1164.all; use IEEE.NUMERIC_STD.all; --2

entity SIPO_1 is port ( --3
    Clk : in STD_LOGIC; --4
    SI : in STD_LOGIC; -- serial in --5
    PO : buffer STD_LOGIC_VECTOR(3 downto 0)); -- parallel out --6
end SIPO_1; --7

architecture Synthesis_1 of SIPO_1 is --8
    begin process (Clk) begin --9
        if (Clk = '1') then PO <= SI & PO(3 downto 1); end if; --10
    end process; --11
end Synthesis_1; --12
```

Here is the Verilog structural netlist that results (dfntnb is a positive-edge-triggered D flip-flop without clear or reset):

```

module sipo_1_u (clk, si, po); //1
input clk; input si; output [3:0] po; //2
supply1 VDD; supply0 VSS; //3
dfntnb po_ff_b0 (.D(po[1]),.CP(clk),.Q(po[0]),.QN(\po_ff_b0.QN)); //4
dfntnb po_ff_b1 (.D(po[2]),.CP(clk),.Q(po[1]),.QN(\po_ff_b1.QN)); //5
dfntnb po_ff_b2 (.D(po[3]),.CP(clk),.Q(po[2]),.QN(\po_ff_b2.QN)); //6
dfntnb po_ff_b3 (.D(si),.CP(clk),.Q(po[3]),.QN(\po_ff_b3.QN )); //7
endmodule //8

```

The synthesized design consists of four flip-flops. Notice that (line 6 in the VHDL input) signal PO is of mode buffer because we cannot read a signal of mode out inside a process. This is acceptable for synthesis but not usually a good idea for simulation models. We can modify the code to eliminate the buffer port and at the same time we shall include a reset signal, as follows:

```

library IEEE; --1
use IEEE.STD_LOGIC_1164.all; use IEEE.NUMERIC_STD.all; --2

entity SIPO_R is port ( --3
    clk : in STD_LOGIC ; res : in STD_LOGIC ; --4
    SI : in STD_LOGIC ; PO : out STD_LOGIC_VECTOR(3 downto 0)); --5
end; --6

architecture Synthesis_1 of SIPO_R is --7
    signal PO_t : STD_LOGIC_VECTOR(3 downto 0); --8
begin --9
    process (PO_t) begin PO <= PO_t; end process; --10
    process (clk, res) begin --11
        if (res = '0') then PO_t <= (others => '0'); --12
        elsif (rising_edge(clk)) then PO_t <= SI & PO_t(3 downto 1); --13
        end if; --14
    end process; --15
end Synthesis_1; --16

```

Notice the following:

- Line 10 uses a temporary signal, PO_t, to avoid using a port of mode buffer for the output signal PO. We could have used a variable instead of a signal and the variable would consume less overhead during simulation. However, we must complete an assignment to a variable inside the clocked process (not in a separate process as we can for the signal). Assignment between a variable and a signal inside a single process creates its own set of problems.
- Line 11 is sensitive to the clock, clk, and the reset, res. It is not sensitive to PO_t or SI and this is what indicates the sequential logic.
- Line 13 uses the rising_edge function from the STD_LOGIC_1164 package.

The software synthesizes four positive-edge-triggered D flip-flops for design entity SIPO_R(Synthesis_1) as it did for design entity SIPO_1(Synthesis_1). The difference is that the synthesized flip-flops in SIPO_R have active-low resets. However, the simulation behavior of these two design entities will be different. In SIPO_R, the function `rising_edge` only evaluates to TRUE for a transition from '0' or 'L' to '1' or 'H'. In SIPO_1 we only tested for `Clk = '1'`. Since nearly all synthesis tools now accept `rising_edge` and `falling_edge`, it is probably wiser to use these functions consistently.

12.6.9 Adders and Arithmetic Functions

If you wish to perform `BIT_VECTOR` or `STD_LOGIC_VECTOR` arithmetic you have three choices:

- Use a vendor-supplied package (there are no standard vendor packages—even if a company puts its own package in the IEEE library).
- Convert to `SIGNED` (or `UNSIGNED`) and use the IEEE standard synthesis packages (IEEE Std 1076.3-1997).
- Use overloaded functions in packages or functions that you define yourself.

Here is an example of addition using a ripple-carry architecture:

```

library IEEE; --1
use IEEE.STD_LOGIC_1164.all; use IEEE.NUMERIC_STD.all; --2

entity Adder4 is port ( --3
  in1, in2 : in BIT_VECTOR(3 downto 0) ; --4
  mySum : out BIT_VECTOR(3 downto 0) ) ; --5
end Adder4; --6

architecture Behave_A of Adder4 is --7
  function DIY(L,R: BIT_VECTOR(3 downto 0)) return BIT_VECTOR is --8
  variable sum:BIT_VECTOR(3 downto 0);variable lt,rt,st,cry: BIT; --9
  begin cry := '0'; --10
  for i in L'REVERSE_RANGE loop --11
    lt := L(i); rt := R(i); st := lt xor rt; --12
    sum(i):= st xor cry; cry:= (lt and rt) or (st and cry); --13
  end loop; --14
  return sum; --15
end; --16
begin mySum <= DIY (in1, in2); -- do it yourself (DIY) add --17
end Behave_A; --18

```

This model results in random logic.

An alternative is to use `UNSIGNED` or `UNSIGNED` from the IEEE `NUMERIC_STD` or `NUMERIC_BIT` packages as in the following example:

```

library IEEE; --1
use IEEE.STD_LOGIC_1164.all; use IEEE.NUMERIC_STD.all; --2

```

```

entity Adder4 is port (                                     --3
    in1, in2 : in    UNSIGNED(3 downto 0) ;               --4
    mySum : out     UNSIGNED(3 downto 0) ) ;               --5
end Adder4;                                               --6

architecture Behave_B of Adder4 is                         --7
    begin mySum <= in1 + in2; -- This uses an overloaded '+'. --8
end Behave_B;                                           --9

```

In this case, the synthesized logic will depend on the logic synthesizer.

12.6.10 Adder/Subtractor and Don't Cares

The following code models a 16-bit sequential adder and subtracter. The input signal, `xin`, is added to output signal, `result`, when signal `addsub` is high; otherwise `result` is subtracted from `xin`. The internal signal `addout` temporarily stores the result until the next rising edge of the clock:

```

library IEEE;                                             --1
use IEEE.STD_LOGIC_1164.all; use IEEE.NUMERIC_STD.all;  --2
entity Adder_Subtractor is port (                         --3
    xin : in          UNSIGNED(15 downto 0);              --4
    clk, addsub, clr: in    STD_LOGIC;                    --5
    result : out      UNSIGNED(15 downto 0));              --6
end Adder_Subtractor;                                    --7

architecture Behave_A of Adder_Subtractor is             --8
    signal addout, result_t: UNSIGNED(15 downto 0);      --9
    begin                                                --10
        result <= result_t;                               --11
        with addsub select                                --12
        addout <=  (xin + result_t)    when '1',          --13
                   (xin - result_t)    when '0',          --14
                   (others => '-')     when others;       --15
        process (clr, clk) begin                          --16
            if (clr = '0') then result_t <= (others => '0'); --17
            elsif rising_edge(clk) then result_t <= addout; --18
            end if;                                       --19
        end process;                                      --20
    end Behave_A;                                        --21

```

Notice the following:

- Line 11 is a concurrent assignment to avoid using a port of mode buffer.
- Lines 12–15 define an exhaustive list of choices for the selected signal assignment statement. The default choice sets the result to '-' (don't care) to allow the synthesizer to optimize the logic.

Line 18 includes a reference to signal addout that could be eliminated by moving the selected signal assignment statement inside the clocked process as follows:

```

architecture Behave_B of Adder_Subtractor is                                --1
  signal result_t: UNSIGNED(15 downto 0);                                --2
  begin                                                                    --3
    result <= result_t;                                                  --4
    process (clr, clk) begin                                             --5
      if (clr = '0') then result_t <= (others => '0');                  --6
      elsif rising_edge(clk) then                                        --7
        case addsub is                                                  --8
          when '1'      => result_t <= (xin + result_t);                --9
          when '0'      => result_t <= (xin - result_t);                --10
          when others   => result_t <= (others => '-');                  --11
        end case;                                                       --12
      end if;                                                            --13
    end process;                                                         --14
  end Behave_B;                                                         --15

```

This code is simpler than architecture Behave_A, but the synthesized logic should be identical for both architectures. Since the logic that results is an adder/subtractor followed by a register (bank of flip-flops) the Behave_A model more clearly reflects the hardware.

12.7 Finite-State Machine Synthesis

There are three ways to synthesize a **finite-state machine (FSM)**:

1. Omit any special synthesis directives and let the logic synthesizer operate on the state machine as though it were random logic. This will prevent any reassignment of states or state machine optimization. It is the easiest method and independent of any particular synthesis tool, but is the most inefficient approach in terms of area and performance.
2. Use directives to guide the logic synthesis tool to improve or modify state assignment. This approach is dependent on the software that you use.
3. Use a special state-machine compiler, separate from the logic synthesizer, to optimize the state machine. You then merge the resulting state machine with the rest of your logic. This method leads to the best results but is harder to use and ties your code to a particular set of software tools, not just the logic synthesizer.

Most synthesis tools require that you write a state machine using a certain style—a special format or template. Synthesis tools may also require that you declare an FSM, the encoding, and the state register using a synthesis directive or special software command. Common FSM encoding options are:

- **Adjacent encoding** assigns states by the minimum logic difference in the state transition graph. This normally reduces the amount of logic needed to decode each state. The minimum number of bits in the state register for an FSM with n states is $\log 2n$. In some tools you may increase the state register width up to n to generate encoding based on Gray codes.
- **One-hot encoding** sets one bit in the state register for each state. This technique seems wasteful. For example, an FSM with 16 states requires 16 flip-flops for one-hot encoding but only four if you use a binary encoding. However, one-hot encoding simplifies the logic and also the interconnect between the logic. One-hot encoding often results in smaller and faster FSMs. This is especially true in programmable ASICs with large amounts of sequential logic relative to combinational logic resources.
- **Random encoding** assigns a random code for each state.
- **User-specified encoding** keeps the explicit state assignment from the HDL.
- **Moore encoding** is useful for FSMs that require fast outputs. A Moore state machine has outputs that depend only on the current state (Mealy state machine outputs depend on the current state and the inputs).

You need to consider how the reset of the state register will be handled in the synthesized hardware. In a programmable ASIC there are often limitations on the polarity of the flip-flop resets. For example, in some FPGAs all flip-flop resets must all be of the same polarity (and this restriction may or may not be present or different for the internal flip-flops and the flip-flops in the I/O cells). Thus, for example, if you try to assign the reset state as '0101', it may not be possible to set two flip-flops to '0' and two flip-flops to '1' at the same time in an FPGA. This may be handled by assigning the reset state, `resSt`, to '0000' or '1111' and inverting the appropriate two bits of the state register wherever they are used.

You also need to consider the initial value of the state register in the synthesized hardware. In some reprogrammable FPGAs, after programming is complete the flip-flops may all be initialized to a value that may not correspond to the reset state. Thus if the flip-flops are all set to '1' at start-up and the reset state is '0000', the initial state is '1111' and not the reset state. For this reason, and also to ensure fail-safe behavior, it is important that the behavior of the FSM is defined for every possible value of the state register.

12.7.1 FSM Synthesis in Verilog

The following FSM model uses **paired processes**. The first process synthesizes to sequential logic and the second process synthesizes to combinational logic:

```

`define resSt 0 //1
`define S1 1 //2
`define S2 2 //3
`define S3 3 //4

module StateMachine_1 (reset, clk, yOutReg); //5
  input reset, clk; output yOutReg; //6
  reg yOutReg, yOut; reg [1:0] curSt, nextSt; //7
  always @(posedge clk or posedge reset) //8
  begin:Seq //Compass statemachine oneHot curSt //9
    if (reset == 1) //10
      begin yOut = 0; yOutReg = yOut; curSt = `resSt; end //11
    else begin //12
      case (curSt) //13
        `resSt:yOut = 0;`S1:yOut = 1;`S2:yOut = 1;`S3:yOut = 1; //14
        default:yOut = 0; //15
      endcase //16
      yOutReg = yOut; curSt = nextSt; // ... update the state. //17
    end //18
  end //19
  always @(curSt or yOut) // Assign the next state: //20
  begin:Comb //21
    case (curSt) //22
      `resSt:nextSt = `S3; `S1:nextSt = `S2; //23
      `S2:nextSt = `S1; `S3:nextSt = `S1; //24
      default:nextSt = `resSt; //25
    endcase //26
  end //27
endmodule //28

```

Synopsys uses separate pseudocomments to define the states and state vector as in the following example:

```

module StateMachine_2 (reset, clk, yOutReg); //1
  input reset, clk; output yOutReg; reg yOutReg, yOut; //2
  parameter [1:0] //synopsys enum states //3
    resSt = 2'b00, S1 = 2'b01, S2 = 2'b10, S3 = 2'b11; //4
  reg [1:0] /* synopsys enum states */ curSt, nextSt; //5
  //synopsys state_vector curSt //6
  always @(posedge clk or posedge reset) begin //7
    if (reset == 1) //8
      begin yOut = 0; yOutReg = yOut; curSt = resSt; end //9
    else begin //10
      case (curSt) resSt:yOut = 0;S1:yOut = 1;S2:yOut = 1;S3:yOut = 1; //11

```

```

        default:yOut = 0; endcase //12
    yOutReg = yOut; curSt = nextSt; end //13
end //14
always @(curSt or yOut) begin //15
    case (curSt) //16
        resSt:nextSt = S3; S1:nextSt = S2; S2:nextSt = S1; S3:nextSt = S1; //17
        default:nextSt = S1; endcase //18
    end //19
endmodule //20

```

To change encoding we can assign states explicitly by altering lines 3–4 to the following, for example:

```

parameter [3:0] //synopsys enum states
    resSt = 4'b0000, S1 = 4'b0010, S2 = 4'b0100, S3 = 4'b1000;

```

12.7.2 FSM Synthesis in VHDL

The first architecture that follows is a template for a Moore state machine:

```

library IEEE; use IEEE.STD_LOGIC_1164.all; --1
entity SM1 is --2
    port (aIn, clk : in Std_logic; yOut: out Std_logic); --3
end SM1; --4

architecture Moore of SM1 is --5
    type state is (s1, s2, s3, s4); --6
    signal pS, nS : state; --7
    begin --8
        process (aIn, pS) begin --9
            case pS is --10
                when s1 => yOut <= '0'; nS <= s4; --11
                when s2 => yOut <= '1'; nS <= s3; --12
                when s3 => yOut <= '1'; nS <= s1; --13
                when s4 => yOut <= '1'; nS <= s2; --14
            end case; --15
        end process; --16
        process begin --17
            -- synopsys etc. --18
            --compass Statemachine adj pS --19
            wait until clk = '1'; pS <= nS; --20
        end process; --21
    end Moore; --22

```

An example input, `aIn`, is included but not used in the next state assignments. A reset is also omitted to further simplify this example.

An FSM compiler **extracts** the state machine. Some companies use FSM compilers that are separate from the logic synthesizers (and priced separately) because

the algorithms for FSM optimization are different from those for optimizing combinational logic. We can see what is happening by asking the Compass synthesizer to write out intermediate results. The synthesizer extracts the FSM and produces the following output in a state-machine language used by the tools:

```
sm sm1_ps_sm;
inputs; outputs yout_smo; clock clk;
STATE S1 { let yout_smo=0 ; } --> S4;
STATE S2 { let yout_smo=1 ; } --> S3;
STATE S3 { let yout_smo=1 ; } --> S1;
STATE S4 { let yout_smo=1 ; } --> S2;
end
```

You can use this language to modify the FSM and then use this modified code as an input to the synthesizer if you wish. In our case, it serves as documentation that explains the FSM behavior.

Using one-hot encoding generates the following structural Verilog netlist (dfntnb is positive-edge-triggered D flip-flop, and nd03d0 is a three-input NAND):

```
dfntnb sm_ps4(.D(sm_ps1_Q),.CP(clk),.Q(sm_ps4_Q),.QN(sm_ps4_QN));
dfntnb sm_ps3(.D(sm_ps2_Q),.CP(clk),.Q(sm_ps3_Q),.QN(sm_ps3_QN));
dfntnb sm_ps2(.D(sm_ps4_Q),.CP(clk),.Q(sm_ps2_Q),.QN(sm_ps2_QN));
dfntnb sm_ps1(.D(sm_ps3_Q),.CP(clk),.Q(sm_ps1_Q),.QN(\sm_ps1.QN ));
nd03d0 i_6(.A1(sm_ps4_QN),.A2(sm_ps3_QN),.A3(sm_ps2_QN),.ZN(yout_smo));
```

(Each example shows only the logic cells and their interconnection in the Verilog structural netlists.) The synthesizer has assigned one flip-flop to each of the four states to form a 4-bit state register. The FSM output (renamed from `yout` to `yout_smo` by the software) is taken from the output of the three-input NAND gate that decodes the outputs from the flip-flops in the state register.

Using adjacent encoding gives a simpler result,

```
dfntnb sm_ps2(.D(i_4_ZN),.CP(clk),.Q(\sm_ps2.Q),.QN(sm_ps2_QN));
dfntnb sm_ps1(.D(sm_ps1_QN),.CP(clk),.Q(\sm_ps1.Q),.QN(sm_ps1_QN));
oa04d1 i_4(.A1(sm_ps1_QN),.A2(sm_ps2_QN),.B(yout_smo),.ZN(i_4_ZN));
nd02d0 i_5(.A1(sm_ps2_QN),.A2(sm_ps1_QN),.ZN(yout_smo));
```

(`oa04d1` is an OAI21 logic cell, `nd02d0` is a two-input NAND). In this case binary encoding for the four states uses only two flip-flops. The two-input NAND gate decodes the states to produce the output. The OAI21 logic cell implements the logic that determines the next state. The combinational logic in this example is only slightly more complex than that for the one-hot encoding, but, in general, combinational logic for one-hot encoding is simpler than the other forms of encoding.

Using the option 'moore' for Moore encoding, we receive the following message from the FSM compiler:

```
The states were assigned these codes:
0?? : S1      100 : S2      101 : S3      110 : S4
```

The FSM compiler has assigned three bits to the state register. The first bit in the state register is used as the output. We can see more clearly what has happened by looking at the Verilog structural netlist:

```
dfntnb sm_ps3(.D(i_6_ZN),.CP(clk),.Q(yout_smo),.QN(sm_ps3_QN));
dfntnb sm_ps2(.D(sm_ps3_QN),.CP(clk),.Q(sm_ps2_Q),.QN(\sm_ps2.QN ));
dfntnb sm_ps1(.D(i_5_ZN),.CP(clk),.Q(sm_ps1_Q),.QN(\sm_ps1.QN ));
nr02d0 i_5(.A1(sm_ps3_QN),.A2(sm_ps2_Q),.ZN(i_5_ZN));
nd02d0 i_6(.A1(sm_ps1_Q),.A2(yout_smo),.ZN(i_6_ZN));
```

The output, `yout_smo`, is now taken directly from a flip-flop. This means that the output appears after the clock edge with no combinational logic delay (only the clock-to-Q delay). This is useful for FSMs that are required to produce outputs as soon as possible after the active clock edge (in PCI bus controllers, for example).

The following code is a template for a Mealy state machine:

```
library IEEE; use IEEE.STD_LOGIC_1164.all;           --1
entity SM2 is                                       --2
  port (aIn, clk : in Std_logic; yOut: out Std_logic); --3
end SM2;                                           --4

architecture Mealy of SM2 is                       --1
  type state is (s1, s2, s3, s4);                 --2
  signal pS, nS : state;                          --3
  begin                                           --4
  process(aIn, pS) begin                          --5
  case pS is                                       --6
  when s1 => if (aIn = '1')                       --7
    then yOut <= '0'; nS <= s4;                 --8
    else yOut <= '1'; nS <= s3;                 --9
    end if;                                       --10
  when s2 => yOut <= '1'; nS <= s3;              --11
  when s3 => yOut <= '1'; nS <= s1;              --12
  when s4 => if (aIn = '1')                       --13
    then yOut <= '1'; nS <= s2;                 --14
    else yOut <= '0'; nS <= s1;                 --15
    end if;                                       --16
  end case;                                       --17
  end process;                                    --18
  process begin                                   --19
  wait until clk = '1' ;                          --20
  --Compass Statemachine oneHot pS                --21
  pS <= nS;                                       --22
```

```

    end process;                                --23
end Mealy;                                     --24

```

12.8 Memory Synthesis

There are several approaches to memory synthesis:

1. Random logic using flip-flops or latches
2. Register files in datapaths
3. RAM standard components
4. RAM compilers

The first approach uses large vectors or arrays in the HDL code. The synthesizer will map these elements to arrays of flip-flops or latches depending on how the timing of the assignments is handled. This approach is independent of any software or type of ASIC and is the easiest to use but inefficient in terms of area. A flip-flop may take up 10 to 20 times the area of a six-transistor static RAM cell.

The second approach uses a synthesis directive or hand instantiation to synthesize a memory to a datapath component. Usually the datapath components are constructed from latches in a regular array. These are slightly more efficient than a random arrangement of logic cells, but the way we create the memory then depends on the software and the ASIC technology we are using.

The third approach uses standard components supplied by an ASIC vendor. For example, we can instantiate a small RAM using CLBs in a Xilinx FPGA. This approach is very dependent on the technology. For example, we could not easily transfer a design that uses Xilinx CLBs as SRAM to an Actel FPGA.

The last approach, using a custom RAM compiler, is the most area-efficient approach. It depends on having the capability to call a compiler from within the synthesis tool or to instantiate a component that has already been compiled.

12.8.1 Memory Synthesis in Verilog

Most synthesizers implement a Verilog memory array, such as the one shown in the following code, as an array of latches or flip-flops.

```
reg [31:0] MyMemory [3:0]; // a 4 x 32-bit register
```

For example, the following code models a small RAM, and the synthesizer maps the memory array to sequential logic:

```

module RAM_1(A, CEB, WEB, OEB, INN, OUTT);           //1
    input [6:0] A; input CEB,WEB,OEB; input [4:0]INN; //2
    output [4:0] OUTT;                               //3
    reg [4:0] OUTT; reg [4:0] int_bus; reg [4:0] memory [127:0]; //4
always@(negedge CEB) begin                          //5
    if (CEB == 0) begin                              //6

```

```

        if (WEB == 1) int_bus = memory[A];           //7
        else if (WEB == 0) begin memory[A] = INN; int_bus = INN; end //8
        else int_bus = 5'bxxxxx;                   //9
    end                                             //10
end                                               //11
always@(OEB or int_bus) begin                    //12
    case (OEB) 0 : OUTT = int_bus;                //13
        default : OUTT = 5'bzzzzz; endcase       //14
    end                                           //15
endmodule                                        //16

```

Memory synthesis using random control logic and transparent latches for each bit is reasonable only for small, fast register files, or for local RAM on an MGA or CBIC. For large RAMs synthesized memory becomes very expensive and instead you should normally use a dedicated RAM compiler.

Typically there will be restrictions on synthesizing RAM with multiple read/writes:

- If you write to the same memory in two different processes, be careful to avoid address contention.
- You need a multiport RAM if you read or write to multiple locations simultaneously.
- If you write and read the same memory location, you have to be very careful. To mimic hardware you need to read before you write so that you read the old memory value. If you attempt to write before reading, the difference between blocking and nonblocking assignments can lead to trouble.

You cannot make a memory access that depends on another memory access in the same clock cycle. For example, you cannot do this:

```
memory[i + 1] = memory[i]; // needs two clock cycles
```

or this:

```
pointer = memory[memory[i]]; // needs two clock cycles
```

For the same reason (but less obviously) we cannot do this:

```
pc = memory[addr1]; memory[addr2] = pc + 1; // not on the same cycle
```

12.8.2 Memory Synthesis in VHDL

VHDL allows multidimensional arrays so that we can synthesize a memory as an array of latches by declaring a two-dimensional array as follows:

```

type memStor is array(3 downto 0) of integer; -- This is OK.
subtype MemReg is STD_LOGIC_VECTOR(15 downto 0); -- So is this.
type memStor is array(3 downto 0) of MemReg;
-- other code...
signal Mem1 : memStor;

```

As an example, the following code models a standard-cell RAM:

```

library IEEE; --1
use IEEE.STD_LOGIC_1164.all; --2
package RAM_package is --3
constant numOut : INTEGER := 8; --4
constant wordDepth: INTEGER := 8; --5
constant numAddr : INTEGER := 3; --6
subtype MEMV is STD_LOGIC_VECTOR(numOut-1 downto 0); --7
type MEM is array (wordDepth-1 downto 0) of MEMV; --8
end RAM_package; --9

library IEEE; --10
use IEEE.STD_LOGIC_1164.all; use IEEE.NUMERIC_STD.all; --11
use work.RAM_package.all; --12
entity RAM_1 is --13
    port (signal A : in STD_LOGIC_VECTOR(numAddr-1 downto 0); --14
          signal CEB, WEB, OEB : in STD_LOGIC; --15
          signal INN : in MEMV; --16
          signal OUTT : out MEMV); --17
end RAM_1; --18

architecture Synthesis_1 of RAM_1 is --19
    signal i_bus : MEMV; -- RAM internal data latch --20
    signal mem : MEM; -- RAM data --21
begin --22
    process begin --23
        wait until CEB = '0'; --24
        if WEB = '1' then i_bus <= mem(TO_INTEGER(UNSIGNED(A))); --25
        elsif WEB = '0' then --26
            mem(TO_INTEGER(UNSIGNED(A))) <= INN; --27
            i_bus <= INN; --28
        else i_bus <= (others => 'X'); --29
        end if; --30
    end process; --31

    process(OEB, int_bus) begin -- control output drivers: --32
        case (OEB) is --33
            when '0' => OUTT <= i_bus; --34
            when '1' => OUTT <= (others => 'Z'); --35
            when others => OUTT <= (others => 'X'); --36
        end case; --37
    end process; --38
end Synthesis_1; --39

```

12.9 The Multiplier

This section looks at the messages that result from attempting to synthesize the VHDL code from Section 10.2, “A 4-bit Multiplier.” The following examples use the line numbers that were assigned in the comments at the end of each line of code in Tables 10.1–10.9. The first problem arises in the following code (line 7 of the full adder in Table 10.1):

```
Sum <= X xor Y xor Cin after TS;
```

Warning: AFTER clause in a waveform element is not supported

This is not a serious problem if you are using a synchronous design style. If you are, then your logic will work whatever the delays (it may run slowly but it will work).

The next problem is from lines 3–4 of the 8-bit MUX in Table 10.5,

```
port (A, B : in BIT_VECTOR (7 downto 0); Sel : in BIT := '0'; Y : out
      BIT_VECTOR (7 downto 0));
```

Warning: Default values on interface signals are not supported

The synthesis tool cannot mimic the behavior of a default value on a port in the software model. The default value is the value given to an input if nothing is connected ('open' in VHDL). In hardware either an input is connected or it is not. If it is connected, there will be a voltage on the wire. If it is not connected, the node will be floating. Default values are useful in VHDL—without a default value on an input port, an entity–architecture pair will not compile. The default value may be omitted in this model because this input port is connected at the next higher level of hierarchy.

The next problem illustrates what happens when a designer fails to think like the hardware (from line 3 of the zero-detector in Table 10.6),

```
port (X:BIT_VECTOR; F:out BIT );
```

Error: An index range must be specified for this data type

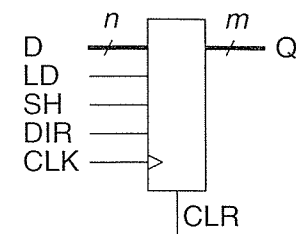
This code has the advantage of being flexible, but the synthesizer needs to know exactly how wide the bus will be. There are two other similar errors in `shiftn`, the variable-width shift register (from lines 4–5 in Table 10.7). There are also three more errors generated by the same problem in the component statement for `AllZero` (from lines 4–5 of package `Mult_Components`) and the component statement for `shiftn` (from lines 10–11 of package `Mult_Components`).

All of these index range problems may be fixed by sacrificing the flexible nature of the code and specifying an index range explicitly, as in the following example:

```
port (X:BIT_VECTOR(7 downto 0); F:out BIT );
```


Table 12.8 shows the synthesizable version of the shift-register model. The constrained index ranges in lines 6, 7, 11, 18, 22, and 23 fix the problem, but are rather ugly. It would be better to use generic parameters for the input and output bus widths. However, a shift register with different input and output widths is not that common so, for now, we will leave the code as it is.

TABLE 12.8 A synthesizable version of the shift register shown in Table 10.7.

<pre> entity ShiftN is generic (TCQ:TIME := 0.3 ns; TLQ:TIME := 0.5 ns; TSQ:TIME := 0.7 ns); port(CLK, CLR, LD, SH, DIR: in BIT; D: in BIT_VECTOR(3 downto 0); Q: out BIT_VECTOR(7 downto 0)); end ShiftN; architecture Behave of ShiftN is begin Shift: process (CLR, CLK) variable St: BIT_VECTOR(7 downto 0); begin if CLR = '1' then St := (others => '0'); Q <= St after TCQ; elsif CLK'EVENT and CLK='1' then if LD = '1' then St := (others => '0'); St(3 downto 0) := D; Q <= St after TLQ; elsif SH = '1' then case DIR is when '0'=>St:='0' & St(7 downto 1); when '1'=>St:=St(6 downto 0) & '0'; end case; Q <= St after TSQ; end if; end if; end process; end; </pre>	<pre> --1 --2 --3 --4 --5 --6 --7 --8 --9 --10 --11 --12 --13 --14 --15 --16 --17 --18 --19 --20 --21 --22 --23 --24 --25 --26 --27 --28 --29 </pre>		<pre> CLK Clock CLR Clear, active high LD Load, active high SH Shift, active high DIR Direction, 1=left D Data in Q Data out </pre> <p>Shift register. Input width = 4. Output width = 8. Output is left-shifted or right-shifted under control of DIR. Unused MSBs are zero-padded during load. Clear is asynchronous. Load is synchronous.</p> <p>Timing: TCQ (CLR to Q) = 0.3 ns TLQ (LD to Q) = 0.5 ns TSQ (SH to Q) = 0.7 ns</p>
---	--	---	---

The next problem occurs because VHDL is not a synthesis language (from lines 6–7 of the variable-width shift register in Table 10.7),

```
begin assert (D'LENGTH <= Q'LENGTH)
  report "D wider than output Q" severity Failure;
```

Warning: Assertion statements are ignored

Error: Statements in entity declarations are not supported

The synthesis tool warns us it does not know how to generate hardware that writes to our screen to implement an assertion statement. The error occurs because a synthesis tool cannot support any of the passive statements (no assignments to signals, for example) that VHDL allows in an entity declaration. Synthesis software usually provides a way around these problems by providing switches to turn the synthesizer on and off. For example, we might be able to write the following:

```
//Compass compile_off
begin assert (D'LENGTH <= Q'LENGTH)
  report "D wider than output Q" severity Failure;
//Compass compile_on
```

The disadvantage of this approach is that the code now becomes tied to a particular synthesis tool. The alternative is to move the statement to the architecture to eliminate the error, and ignore the warning.

The next error message is, at first sight, confusing (from lines 15–16 of the variable-width shift register in Table 10.7),

```
if CLR = '1' then St := (others => '0'); Q <= St after TCQ;
```

Error: Illegal use of aggregate with the choice "others": the derived subtype of an array aggregate that has a choice "others" must be a constrained array subtype

This error message is precise and uses the terminology of the LRM but does not reveal the source of the problem. To discover the problem we work backward through the model. We declared variable `St` as follows (lines 12–13 of Table 10.7):

```
subtype OutB is NATURAL range Q'LENGTH-1 downto 0;
  variable St: BIT_VECTOR(OutB);
```

(to keep the model flexible). Continuing backward we see `Q` is declared as type `BIT_VECTOR` with no index range as follows (lines 4–5 of Table 10.7):

```
port(CLK, CLR, LD, SH, DIR: in BIT;
      D: in BIT_VECTOR; Q: out BIT_VECTOR);
```

The error is thus linked to the previous problem (undeclared bus widths) in this entity–architecture pair. Because the synthesizer does not know the width of `Q`, it does not know how many '0's to put in `St` when it has to implement `St := (others => '0')`. There is one more error like this one in the second assignment to `St` (line 19 in Table 10.7). Again the problem may be solved by sacrificing flexibility and constraining the width of `Q` to be a fixed value.

The next warning involves names (line 5 in Table 10.9),

```
signal SRA, SRB, ADDout, MUXout, REGout: BIT_VECTOR(7 downto 0);
```

Warning: Name is reserved word in VHDL-93: sra

This problem can be fixed by (a) changing the signal name, (b) using an escaped name, or (c) accepting that this code will not work in a VHDL-93 environment.

Finally, there is the following warning (line 6 in Table 10.9):

```
signal Zero, Init, Shift, Add, Low: BIT := '0'; signal High: BIT := '1';
```

Warning: Initial values on signals are only for simulation and setting the value of undriven signals in synthesis. A synthesized circuit can not be guaranteed to be in any known state when the power is turned on.

Signals Low and High are used to tie inputs to a logic '0' and to a logic '1', respectively. This is because VHDL-87 does not allow '1' or '0', which are literals, as actual parameters. Thus one way to solve this problem is to change to a VHDL-93 environment, where this restriction was lifted. Some synthesis systems handle VDD and GND nets in a specific fashion. For example, VDD and GND may be declared as constants in a synthesis package. It does not really matter how inputs are connected to VDD and GND as long as they are connected in the synthesized logic.

12.9.1 Messages During Synthesis

After fixing the error and warning messages, we can synthesize the multiplier. During synthesis we see these messages:

```
These unused instances are being removed: in full_adder_p_dup8: u5, u2,
u3, u4
```

```
These unused instances are being removed: in dffclr_p_dup1: u2
```

and seven more similar to this for dffclr_p_dup2: u2 to dffclr_p_dup8: u2. We are suspicious because we did not include any redundant or unused logic in our input code. Let us dig deeper.

Turning to the second set of messages first, we need to discover the locations of dffclr_p_dup1: u2 and the other seven similarly named unused instances. We can ask the synthesizer to produce the following hierarchy map of the design:

```
***** Hierarchy of cell "mult8_p" *****
mult8_p
  adder8_p
    | full_adder_p [x8]
  allzero_p
  mux8_p
  register8_p
    | dffclr_p [x8]
```

```

    shiftn_p [x2]
    sm_1_p

```

The eight unused instances in question are inside the 8-bit shift register, `register8_p`. The only models in this shift register are eight copies of the D flip-flop model, `DFFClr`. Let us look more closely at the following code:

```

architecture Behave of DFFClr is                                --1
signal Qi : BIT;                                              --2
begin QB <= not Qi; Q <= Qi;                                  --3
process (CLR, CLK) begin                                       --4
    if CLR = '1' then Qi <= '0' after TRQ;                    --5
    elsif CLK'EVENT and CLK = '1' then Qi <= D after TCQ;    --6
    end if;                                                    --7
end process;                                                  --8
end;                                                            --9

```

The synthesizer infers an inverter from the first statement in line 3 (`QB <= not Qi`). What we meant to imply (A) was: “I am trying to describe the function of a D flip-flop and it has two outputs; one output is the complement of the other.” What the synthesizer inferred (B) was: “You described a D flip-flop with an inverter connected to Q.” Unfortunately A does not equal B.

Why were four cell instances (`u5`, `u2`, `u3`, `u4`) removed from inside a cell with instance name `full_adder_p_dup8`? The top-level cell `mult8_p` contains cell `adder8_p`, which in turn contains `full_adder_p [x8]`. This last entry in the hierarchy map represents eight occurrences or instances of cell `full_adder_p`. The logic synthesizer appends the suffix `'_p'` by default to the names of the design units to avoid overwriting any existing netlists (it also converts all names to lowercase). The synthesizer has then added the suffix `'dup8'` to create the instance name `full_adder_p_dup8` for the eighth copy of cell `full_adder_p`.

What is so special about the eighth instance of `full_adder_p` inside cell `adder8_p`? The following (line 13 in Table 10.9) instantiates `Adder8`:

```

A1:Adder8 port map(A=>SRB,B=>REGout,Cin=>Low,Cout=>OFL,Sum=>ADDout);

```

The signal `OFL` is declared but not used. This means that the formal port name `Cout` for the entity `Adder8` in Table 10.2 is unconnected in the instance `full_adder_p_dup8`. Since the carry-out bit is unused, the synthesizer deletes some logic. Before dismissing this message as harmless, let us look a little closer. In the architecture for entity `Adder8` we wrote:

```

Cout <= (X and Y) or (X and Cin) or (Y and Cin) after TC;

```

In one of the instances of `Adder8`, named `full_adder_p_dup8`, this statement is redundant since we never use `Cout` in that particular cell instance. If we look at the synthesized netlist for `full_adder_p_dup8` before optimization, we find four

NAND cells that produce the signal `Cout`. During logic optimization the synthesizer removes these four instances. Their instance names are `full_adder_p_dup8:u2`, `u3`, `u4`, `u5`.

12.10 The Engine Controller

This section returns to the example from Section 10.16, “An Engine Controller.” This ASIC gathers sampled temperature measurements from sensors, converts the temperature values from Fahrenheit to Centigrade, averages them, and stores them in a FIFO before passing the values to a microprocessor on a three-state bus. We receive the following message from the logic synthesizer when we use the FIFO-controller code shown in Table 10.25:

```
Warning: Made latches to store values on: net d(4), d(5), d(6), d(7),
d(8), d(9), d(10), d(11), in module fifo_control
```

This message often indicates that we forgot to initialize a variable.

Here is the part of the code from Table 10.25 that assigns to the vector `D` (the error message for `d` is in lowercase—remember VHDL is case insensitive):

```
case sel is
  when "01" => D <= D_1 after TPD; r1 <= '1' after TPD;
  when "10" => D <= D_2 after TPD; r2 <= '1' after TPD;
  when "00" => D(3) <= f1 after TPD; D(2) <= f2 after TPD;
               D(1) <= e1 after TPD; D(0) <= e2 after TPD;
  when others => D <= "ZZZZZZZZZZZZ" after TPD;
end case;
```

When `sel = "00"`, there is no assignment to `D(4)` through `D(11)`. This did not matter in the simulation, but to reproduce the exact behavior of the HDL code the logic synthesizer generates latches to remember the values of `D(4)` through `D(11)`.

This problem may be corrected by replacing the “00” choice with the following:

```
when "00" => D(3) <= f1 after TPD; D(2) <= f2 after TPD;
              D(1) <= e1 after TPD; D(0) <= e2 after TPD;
              D(11 downto 4) <= "ZZZZZZZZ" after TPD;
```

The synthesizer recognizes the assignment of the high-impedance logic value ‘`Z`’ to a signal as an indication to implement a three-state buffer. However, there are two kinds of three-state buffers: core logic three-state buffers and three-state I/O cells. We want a three-state I/O cell containing a bonding pad and not a three-state buffer located in the core logic. If we synthesize the code in Table 10.25, we get a three-state buffer in the core. Table 12.9 shows the modified code that will synthesize to three-state I/O cells. The signal `OE_b` drives the output enable (active-low) of the three-state buffers. Table 12.10 shows the top-level code including all the I/O cells.

TABLE 12.9 A modified version of the FIFO controller to drive three-state I/O cells.

```

library IEEE;use IEEE.STD_LOGIC_1164.all;use IEEE.NUMERIC_STD.all;           --1
entity fifo_control is generic TPD:TIME := 1 ns;                             --2
  port(D_1, D_2: in UNSIGNED(11 downto 0);                                   --3
        sel : in UNSIGNED(1 downto 0) ;                                    --4
        read , f1, f2, e1, e2 : in STD_LOGIC;                             --5
        r1, r2, w12:out STD_LOGIC; D: out UNSIGNED(11 downto 0);          --6
        OE:out STD_LOGIC ) ;                                              --7
end;                                                                           --8
architecture rtl of fifo_control is                                          --9
  begin process (read, sel, D_1, D_2, f1, f2, e1, e2)                      --10
  begin                                                                       --11
    r1 <= '0' after TPD; r2 <= '0' after TPD; OE_b <= '0' after TPD;      --12
    if (read = '1') then                                                    --13
      w12 <= '0' after TPD;                                                --14
      case sel is                                                           --15
        when "01" =>  D <= D_1 after TPD; r1 <= '1' after TPD;          --16
        when "10" =>  D <= D_2 after TPD; r2 <= '1' after TPD;          --17
        when "00" =>  D(3) <= f1 after TPD; D(2) <= f2 after TPD;        --18
                     D(1) <= e1 after TPD; D(0) <= e2 after TPD;        --19
                     D(11 downto 4) <= "00000000" after TPD;           --20
        when others => OE_b <= '1' after TPD;                              --21
      end case;                                                             --22
    elsif (read = '0') then                                                --23
      OE_b <= '0' after TPD; w12 <= '1' after TPD;                       --24
    else OE_b <= '0' after TPD;                                           --25
    end if;                                                                  --26
  end process;                                                             --27
end rtl;                                                                     --28

```

12.11 Performance-Driven Synthesis

Many logic synthesizers allow the use of directives. The pseudocomment in the following code directs the logic synthesizer to minimize the delay of an addition:

```

module add_directive (a, b, z); input [3:0] a, b; output [3:0] z;
  //compass maxDelay 2 ns
  //synopsys and so on.
  assign z = a + b;
endmodule

```

TABLE 12.10 The top-level VHDL code for the engine controller ASIC.

```

library COMPASS_LIB, IEEE ; --1
use IEEE.STD.all; use IEEE.NUMERIC_STD.all; --2
use COMPASS_LIB.STDCOMP.all; use COMPASS_LIB.COMPASS.all; --3
--4
entity t_control_ASIC is port( --5
    PadTri : out                STD_LOGIC_VECTOR (11 downto 0) ; --6
    PadClk, PadInreset, PadInreadv : in    STD_LOGIC_VECTOR ( 0 downto 0) ; --7
    PadInp1, PadInp2 : in          STD_LOGIC_VECTOR (11 downto 0) ; --8
    PadInSens : in                STD_LOGIC_VECTOR ( 1 downto 0) ) ; --9
end t_control_ASIC ; --10
--11
architecture structure of t_control_ASIC is --12
for all : asPadIn    use entity COMPASS_LIB.aspadIn(aspadIn) ; --13
for all : asPadClk  use entity COMPASS_LIB.aspadClk(aspadClk); --14
for all : asPadTri  use entity COMPASS_LIB.aspadTri(aspadTri) ; --15
for all : asPadVdd  use entity COMPASS_LIB.aspadVdd(aspadVdd) ; --16
for all : asPadVss  use entity COMPASS_LIB.aspadVss(aspadVss) ; --17
component pc3c01 port ( cclk : in STD_LOGIC; cp : out STD_LOGIC ); end component; --18
component t_control port(T_in1, T_in2 : in UNSIGNED(11 downto 0); --19
    SENSOR: in UNSIGNED( 1 downto 0) ; clk, rd, rst : in STD_LOGIC; --20
    D : out UNSIGNED(11 downto 0); oe_b : out STD_LOGIC ); end component ; --21
signal T_in1_sv, T_in2_sv :          STD_LOGIC_VECTOR(11 downto 0) ; --22
signal T_in1_un, T_in2_un :          UNSIGNED(11 downto 0) ; --23
signal sensor_sv :                  STD_LOGIC_VECTOR(1 downto 0) ; --24
signal sensor_un :                  UNSIGNED(1 downto 0) ; --25
signal clk_sv, rd_fifo_sv, reset_sv : STD_LOGIC_VECTOR (0 downto 0) ; --26
signal clk_core, oe_b :              STD_LOGIC ; --27
signal D_un : UNSIGNED(11 downto 0) ; signal D_sv : STD_LOGIC_VECTOR(11 downto 0) ; --28
begin --compass dontTouch u* -- synopsys dont_touch etc. --29
u1 : asPadIn generic map(12,"2:13")   port map(t_in1_sv,PadInp1) ; --30
u2 : asPadIn generic map(12,"14:25")  port map(t_in2_sv,PadInp2) ; --31
u3 : asPadIn generic map(2,"26:27")   port map(sensor_sv, PadInSens ) ; --32
u4 : asPadIn generic map(1,"29")      port map(rd_fifo_sv, PadInReadv ) ; --33
u5 : asPadIn generic map(1,"30")      port map(reset_sv, PadInreset ) ; --34
u6 : asPadIn generic map(1,"32")      port map(clk_sv, PadClk) ; --35
u7 : pc3c01                            port map(clk_sv(0), clk_core) ; --36
u8 : asPadTri generic map(12,"35:38,41:44,47:50") port map(PadTri,D_sv,oe_b); --37
u9 : asPadVdd generic map("1,31,34,40,45,52") port map(Vdd) ; --38
u10: asPadVss generic map("28,33,39,46,51,53") port map(Vss) ; --39
T_in1_un <= UNSIGNED(T_in1_sv) ; T_in2_un <= UNSIGNED(T_in2_sv) ; --40
sensor_un <= UNSIGNED(sensor_sv) ; D_sv <= STD_LOGIC_VECTOR(D_un) ; --41
v_1 : t_control port map --42
    (T_in1_un,T_in2_un,sensor_un, Clk_core, rd_fifo_sv(0), reset_sv(0),D_un, oe_b) ; --43
end; --44

```

These directives become complicated when we need to describe complex timing constraints. Figure 12.7(a) shows an example of a more flexible method to measure and specify delay using **timing arcs** (or timing paths). Suppose we wish to improve the performance of the comparator/MUX example from Section 12.2. First we define a **pathcluster** (a group of circuit nodes—see Figure 12.7b). Next, we specify the **required time** for a signal to reach the output nodes (the **end set**) as 2 ns. Finally, we specify the **arrival time** of the signals at all the inputs as 0 ns. We have thus constrained the delay of the comparator/MUX to be 2 ns—measured between any input and any output. The logic-optimization step will simplify the logic network and then map it to a cell library while attempting to meet the timing constraints.

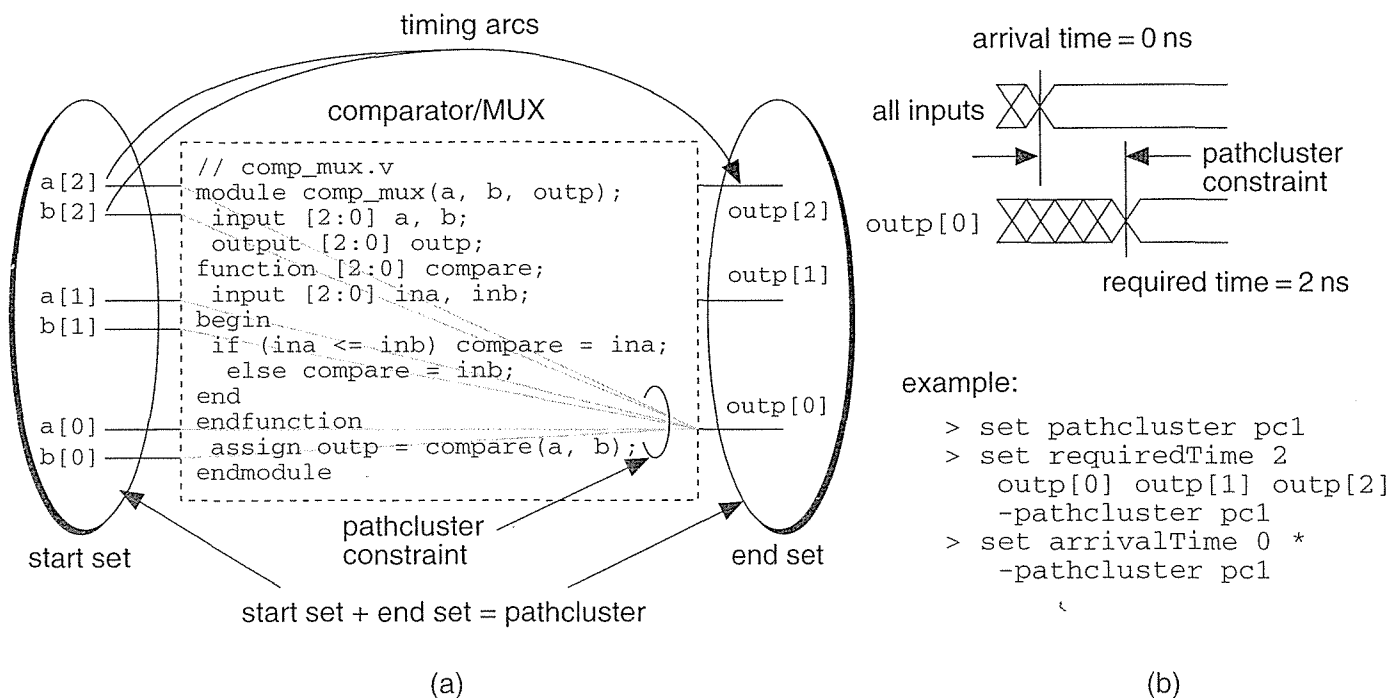


FIGURE 12.7 Timing constraints. (a) A pathcluster. (b) Defining constraints.

Table 12.11 shows the results of a timing-driven logic optimization for the comparator/MUX. Comparing these results with the default optimization results shown in Table 12.3 reveals that the timing has dramatically improved (critical path delay was 2.43 ns with default optimization settings, and the delay varies between 0.31 ns and 1.64 ns for the timing-driven optimization).

Figure 12.8 shows that timing-driven optimization and the subsequent mapping have simplified the logic considerably. For example, the logic for outp[2] has been

TABLE 12.11 Timing-driven synthesis reports for the comparator/MUX example of Section 12.2.

Command	Synthesizer output ¹					
> set pathcluster pcl						
> set requiredTime 2 outp[0] outp[1] outp[2] -pathcluster pcl						
> set arrivalTime 0 * -pathcluster pcl						
> optimize		Num	Gate Count	Tot Gate	Width	Total
	Cell Name	Insts	Per Cell	Count	Per Cell	Width
	-----	-----	-----	-----	-----	-----
	an02d1	1	1.3	1.3	12.0	12.0
	in01d0	2	.8	1.5	7.2	14.4
	mx21d1	2	2.2	4.5	21.6	43.2
	nd02d0	2	1.0	2.0	9.6	19.2
	oa03d1	1	1.8	1.8	16.8	16.8
	oa04d1	1	1.3	1.3	12.0	12.0
	-----	-----	-----	-----	-----	-----
	Totals:	9		12.2		117.6
> report	path cluster name: pcl					
timing	path type: maximum					
-allpaths	-----					
	end node			current	required	slack
	-----			-----	-----	-----
	outp[1]			1.64	2.00	.36 MET
	outp[0]			1.64	2.00	.36 MET
	outp[2]			.31	2.00	1.69 MET

¹See footnote 1 in Table 12.3 for explanations of the abbreviations used in this table.

reduced to a two-input AND gate. Using *sis* reveals how optimization works in this case. Table 12.12 shows the equations for the intermediate signal *se1* and the three comparator/MUX outputs in the BLIF. Thus, for example, the following line of the BLIF code in Table 12.12 (the first line following *.names a0 b0 a1 b1 a2 b2 se1*) includes the term $a0 \cdot b0' \cdot a1' \cdot b1' \cdot a2' \cdot b2'$ in the equation for *se1*:

```
100000 1
```

There are six similar lines that describe the six other product terms for *se1*. These seven product terms form a cover for *se1* in the Karnaugh maps of Figure 12.5.

In addition *sis* must be informed of the don't care values (called the **external don't care set**) in these Karnaugh maps. This is the function of the PLA-format input that follows the *.exdc* line. Now *sis* can simplify the equations including the don't care values using a standard script, *rugged.script*, that contains a sequence

```

`timescale 1ns / 10ps
module comp_mux_o (a, b, outp);
input [2:0] a; input [2:0] b;
output [2:0] outp;
supply1 VDD; supply0 VSS;

mx21d1 B1_i1 (.I0(a[0]),
.I1(b[0]), .S(B1_i6_ZN),
.Z(outp[0]));
oa03d1 B1_i2 (.A1(B1_i9_ZN),
.A2(a[2]), .B1(a[0]), .B2(a[1]),
.C(B1_i4_ZN), .ZN(B1_i2_ZN));
nd02d0 B1_i3 (.A1(a[1]),
.A2(a[0]), .ZN(B1_i3_ZN));
nd02d0 B1_i4 (.A1(b[1]),
.A2(B1_i3_ZN), .ZN(B1_i4_ZN));
mx21d1 B1_i5 (.I0(a[1]),
.I1(b[1]), .S(B1_i6_ZN),
.Z(outp[1]));
oa04d1 B1_i6 (.A1(b[2]),
.A2(B1_i7_ZN), .B(B1_i2_ZN),
.ZN(B1_i6_ZN));
in01d0 B1_i7 (.I(a[2]),
.ZN(B1_i7_ZN));
an02d1 B1_i8 (.A1(b[2]),
.A2(a[2]), .Z(outp[2]));
in01d0 B1_i9 (.I(b[2]),
.ZN(B1_i9_ZN));

endmodule

```

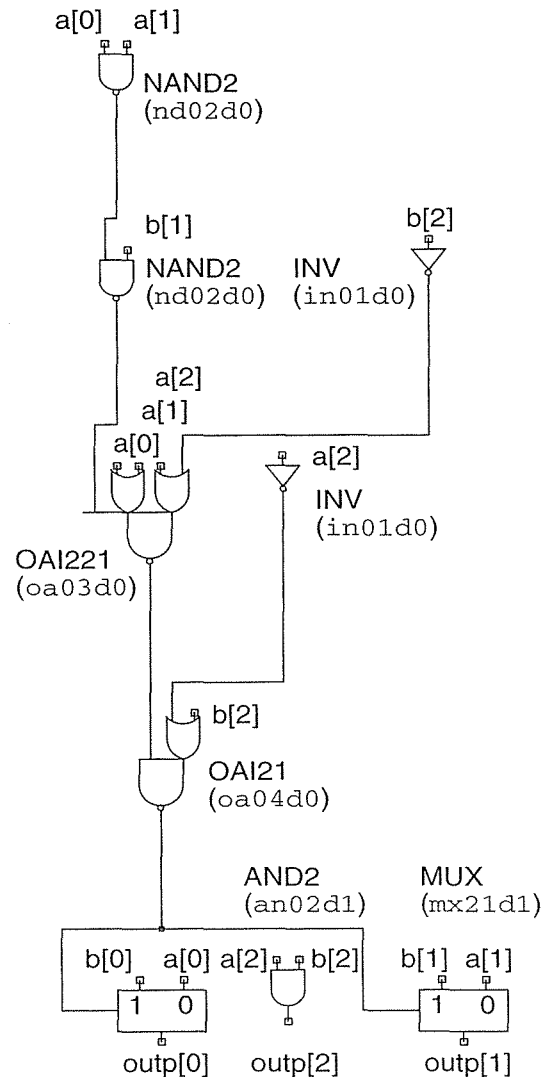


FIGURE 12.8 The comparator/MUX example of Section 12.2 after logic optimization with timing constraints. The figure shows the structural netlist, `comp_mux_o.v`, and its derived schematic. Compare this with Figures 12.2 and 12.3.

of `sis` commands. This particular script uses a series of factoring and substitution steps. The output (Table 12.12) reveals that `sis` finds the same equation for `outp[2]` (named `outp2` in the `sis` equations):

$$\{outp2\} = a2 \ b2$$

The other logic equations in Table 12.12 that `sis` produces are also equivalent to the logic in Figure 12.8. The technology-mapping step hides the exact details of the conversion between the internal representation and the optimized logic.

TABLE 12.12 Optimizing the comparator/MUX equations using `sis`.

sis input file (BLIF)	sis results
<code>.model comp_mux</code>	<code></usr/user1/msmith/sis> sis</code>
<code>.inputs a0 b0 a1 b1 a2 b2</code>	<code>UC Berkeley, SIS Development Version</code>
<code>.outputs outp0 outp1 outp2</code>	<code>(compiled 11-Oct-95 at 11:50 AM)</code>
<code>.names a0 b0 a1 b1 a2 b2 sel</code>	<code>sis> read_blif comp_mux.blif</code>
<code>100000 1</code>	<code>sis> print</code>
<code>101100 1</code>	<code>{outp0} = a0 sel' + b0 sel</code>
<code>--1000 1</code>	<code>{outp1} = a1 sel' + b1 sel</code>
<code>----10 1</code>	<code>{outp2} = a2 sel' + b2 sel</code>
<code>100011 1</code>	<code>sel = a0 a1 a2 b0' b1 b2</code>
<code>101111 1</code>	<code>+ a0 a1 a2' b0' b1 b2'</code>
<code>--1011 1</code>	<code>+ a0 a1' a2 b0' b1' b2</code>
<code>.names sel a0 b0 outp0</code>	<code>+ a0 a1' a2' b0' b1' b2'</code>
<code>1-1 1</code>	<code>+ a1 a2 b1' b2</code>
<code>01- 1</code>	<code>+ a1 a2' b1' b2'</code>
<code>.names sel a1 b1 outp1</code>	<code>+ a2 b2'</code>
<code>1-1 1</code>	<code>sis> source script.rugged</code>
<code>01- 1</code>	<code>sis> print</code>
<code>.names sel a2 b2 outp2</code>	<code>{outp0} = a0 sel' + b0 sel</code>
<code>1-1 1</code>	<code>{outp1} = a1 sel' + b1 sel</code>
<code>01- 1</code>	<code>{outp2} = a2 b2</code>
<code>.exdc</code>	<code>sel = [9] a2 b0'</code>
<code>.names a0 b0 a1 b1 a2 b2 sel</code>	<code>+ [9] b0' b2'</code>
<code>000000 1</code>	<code>+ a1 a2 b1'</code>
<code>110000 1</code>	<code>+ a1 b1' b2'</code>
<code>001100 1</code>	<code>+ a2 b2'</code>
<code>111100 1</code>	<code>[9] = a1 + b1'</code>
<code>000011 1</code>	<code>sis> quit</code>
<code>110011 1</code>	<code></usr/user1/msmith/sis></code>
<code>001111 1</code>	
<code>111111 1</code>	
<code>.end</code>	

12.12 Optimization of the Viterbi Decoder

Returning to the Viterbi decoder example (from Section 12.4), we first set the **environment** for the design using the following worst-case conditions: a die temperature of 25°C (fastest logic) to 120°C (slowest logic); a power supply voltage of $V_{DD}=5.5\text{ V}$ (fastest logic) to $V_{DD}=4.5\text{ V}$ (slowest logic); and worst process (slowest logic) to best process (fastest logic). Assume that this ASIC should run at a clock frequency of at least 33 MHz (clock period of 30 ns). An initial synthesis run gives a criti-

cal path delay at nominal conditions (the default setting) of about 25 ns and nearly 35 ns under worst-case conditions using a high-density 0.6 μm standard-cell target library.

Estimates (using simulation and calculation) show that data arrives at the input pins 5 ns (worst-case) after the rising edge of the clock. The reset signal arrives 10 ns (worst-case) after the rising edge of the clock. The outputs of the Viterbi decoder must be stable at least 4 ns before the rising edge of the clock. This allows these signals to be driven to another ASIC in time to be clocked. These timing constraints are particularly devastating. Together they effectively reduce the clock period that is available for use by 9 ns. However, these figures are typical for board-level delays.

The initial synthesis runs reveal the critical path is through the following six modules:

```
subset_decode -> compute_metric ->
compare_select -> reduce -> metric -> output_decision
```

The logic synthesizer can do little or no optimization across these module boundaries. The next step, then, is to rearrange the design hierarchy for synthesis. **Flattening** (merging or ungrouping) the six modules into a new cell, called *critical*, allows the synthesizer to reduce the critical path delay by optimizing one large module.

At present the last module in the critical path is *output_decision*. This combinational logic adds 2–3 ns to the output delay requirement of 4 ns (this means the outputs of the module *metric* must be stable 6–7 ns before the rising clock edge). Registering the output reduces this overhead and removes the module *output_decision* from the critical path. The disadvantage is an increase in latency by one clock cycle, but the latency is already 12 clock cycles in this design. If registering the output decreases the critical path delay by more than a factor of 12/13, performance will still improve.

To register the output, alter the code (on pages 575–576) as follows:

```
module viterbi_ASIC
...
wire [2:0] Out, Out_r; // Change: add Out_r.
...
    asPadOut    #(3,"30,31,32") u30 (padOut, Out_r); // Change: Out_r.
    Outreg o_1 (Out, Out_r, Clk, Res); // Change: add output register.
...
endmodule

module Outreg (Out, Out_r, Clk, Res); // Change: add this module.
input [2:0] Out; input Clk, Rst; output [2:0] Out_r;
    dff #(3) reg1(Out, Out_r, Clk, Res);
endmodule
```

These changes move the performance closer to the target. Prelayout estimates indicate the die perimeter required for the I/O pads will allow more than enough area to

hold the core logic. Since there is unused area in the core, it makes sense to switch to a high-performance standard-cell library with a slightly larger cell height (96λ versus 72λ). This cell library is less dense, but faster.

Typically, at this point, the design is improved by altering the HDL, the hierarchy, and the synthesis controls in an iterative manner until the desired performance is achieved. However, remember there is still no information from the layout. The best that can be done is to estimate the contribution of the interconnect using wire-load models. As soon as possible the netlist should be passed to the floorplanner (or the place-and-route software in the absence of a floorplanner) to generate better estimates of interconnect delays.

TABLE 12.13 Critical-path timing report for the Viterbi decoder.

Instance name		Delay information ¹					
		incr	arrival	trs	rampDel	cap(pF)	cell
v_1.u100	inPin --> outPin						
u1.subout5.Q_ff_b0	CP --> QN	1.65	1.65	F	.20	.10	dfctnb
B1_i67	A1 --> ZN	.63	2.27	R	.14	.08	ao01d1
B1_i66	B --> ZN	.84	3.12	F	.15	.08	ao04d1
B1_i64	B2 --> ZN	.91	4.03	F	.35	.17	fn03d1
B1_i68	I --> ZN	.39	4.43	R	.23	.12	in01d1
B1_i316	S --> Z	.91	5.33	F	.34	.17	mx21d1
u3.add_rip1.u4	B0 --> CO	2.20	7.54	F	.24	.14	ad02d1
... 28 other cell instances omitted ...							
u5.sub_rip1.u6	B0 --> CO	2.25	23.17	F	.23	.13	ad02d1
u5.sub_rip1.u8	CI --> CO	.53	23.70	F	.21	.09	ad01d1
B1_i301	A1 --> Z	.69	24.39	R	.19	.07	xo02d1
u2.metric3.Q_ff_b4	setup: D --> CP	.17	24.56	R	.00	.00	dfctnb
	slack: MET		.44				

¹See the text for explanations of the column headings.

Table 12.13 is a timing report for the Viterbi decoder, which shows the critical path starts at a sequential logic cell (a D flip-flop in the present example), ends at a sequential logic cell (another D flip-flop), with 37 other combinational logic cells in-between. The first delay is the clock-to-Q delay of the first flip-flop. The last delay is the setup time of the last flip-flop. The critical path delay is 24.56 ns, which gives a **slack** of 0.44 ns from the constraint of 25 ns (reduced from 30 ns to give an extra margin). We have **met** the timing constraint (otherwise we say it is **violated**).

In Table 12.13 all instances in the critical path are inside instance `v_1.u100`. Instance name `u100` is the new cell (cell name `critical`) formed by merging six blocks in module `viterbi` (instance name `v_1`).

The second column in Table 12.13 shows the timing arc of the cell involved on the critical path. For example, `CP --> QN` represents the path from the clock pin, `CP`, to the flip-flop output pin, `QN`, of a D flip-flop (cell name `dfctnb`). The pin names and their functions come from the library data book. Each company adopts a different naming convention (in this case `CP` represents a positive clock edge, for example). The conventions are not always explicitly shown in the data books but are normally easy to discover by looking at examples. As another example, `B0 --> CO` represents the path from the B input to the carry output of a 2-bit full adder (cell name `ad02d1`).

The third column (`incr`) represents the incremental delay contribution of the logic cell to the critical path.

The fourth column (`arrival`) shows the arrival time of the signal at the output pin of the logic cell. This is the cumulative delay to that point on the critical path.

The fifth column (`trs`) describes whether the transition at the output node is rising (`R`) or falling (`F`). The timing analyzer examines each possible combination of rising and falling delays to find the critical path.

The sixth column (`rampDel`) is a measure of the input slope (ramp delay, or slew rate). In submicron ASIC design this is an important contribution to delay.

The seventh column (`Cap`) is the capacitance at the output node of the logic cell. This determines the logic cell delay and also the signal slew rate at the node.

The last column (`cell`) is the cell name (from the cell-library data book). In this library suffix `'d1'` represents normal drive strength with `'d0'`, `'d2'`, and `'d5'` being the other available strengths.

12.13 Summary

A logic synthesizer may contain over 500,000 lines of code. With such a complex system, complex inputs, and little feedback at the output there is a danger of the “garbage in, garbage out” syndrome. Ask yourself “What do I expect to see at the output?” and “Does the output make sense?” If you cannot answer these questions, you should simplify the input (reduce the width of the buses, simplify or partition the code, and so on). The worst thing you can do is write and simulate a huge amount of code, read it into the synthesis tool, and try and optimize it all at once with the default settings.

With experience it is possible to recognize what the logic synthesizer is doing by looking at the number of cells, their types, and the drive strengths. For example, if there are many minimum drive strength cells on the critical path it is usually an indication that the synthesizer has room to increase speed by substituting cells with stronger drive. This is not always true, sometimes a higher-drive cell may actually

slow down the circuit. This is because adding the larger cell increases load capacitance, but not enough drive to make up for it. This is why logical effort is a useful measure.

Because interconnect delay is increasingly dominant, it is important to begin the physical design steps as early as possible. Ideally floorplanning and logic synthesis should be completed at the same time. This ensures that the estimated interconnect delays are close to the actual delays after routing is complete.

12.14 Problems

* = Difficult, ** = Very difficult, *** = Extremely difficult

12.1 (Comparator/MUX)

- a. (30 min.) Build a DesignWorks (or use another tool) model for the schematic in Figure 12.1 and simulate the operation of this circuit to check that it performs the same function as the Verilog code. *Hint:* You could also use VeriWell to simulate the Verilog netlist.
- b. (30 min.) Simulate the schematic (or the Verilog netlist) shown in Figure 12.2 and check that it performs the comparator/MUX function correctly.
- c. (30 min.) Simulate the schematic (or the Verilog netlist) of Figure 12.3. If you have access to a logic synthesizer and cell library, you might resynthesize the comparator/MUX and compare the results with those shown in Figures 12.2 and 12.3.
- d. (20 min.) Build a schematic (or Verilog model) for macro `cm8` in Figure 12.4.
- e. (30 min.) Simulate the schematic (or Verilog netlist) shown in Figure 12.4.

12.2 (*Verilog assignments, 15 min.) Simulate and test the following model paying attention to initialization. Attempt to synthesize it. Explain your results.

```

module dff (D, Q, Clk, Rst);
  parameter width = 1, reset_value = 0; input [width - 1 : 0] D;
  output [width - 1 : 0] Q; reg [width - 1 : 0] Q; input Clk,Rst;
  initial Q = {width{1'bx}};
  always @ ( posedge Clk or negedge Rst )
    if ( Rst == 0 ) Q <= #1 reset_value; else Q <= #1 D;
endmodule

```

12.3 (Digital filter) (30 min.) Write HDL code to model the following filter:

```
y0 <= c(0) * x(0) + c(1) * x(1) + b(2) * x(2) ;
```

Use $c(0) = -4$, $c(0) = +5$, $c(0) = -3$, but make your code flexible so that these coefficients may be changed. (120 min.) Simulate, test, and synthesize your model. *Hint:* You should use the transfer equation in your code (Verilog or VHDL).

12.4 (Hand design, 60 min.) Use hand calculation and gate delay values obtained from a data book to estimate the critical path of the comparator/MUX shown in Figure 12.1. Assume the critical path (the one with the longest delay) is from the $a[2]$ input (the input with the largest load) \rightarrow XOR \rightarrow inverter \rightarrow four-input NAND \rightarrow three-input OR \rightarrow select input of two-input MUX (the symbol \rightarrow means “through the”). You will need to find the t_{PHL} (falling) and t_{PLH} (rising) propagation delays for each gate used in the critical path. Do not adjust the delays for the loading (fanout) at the output of each gate on the critical path, assume a loading equal to one input of a two-input NAND gate. Change the AND–NOR gate combination to a NAND–NAND gate combination and recalculate the critical path delay.

12.5 (Critical path, 30 min.) Enter the schematic shown in Figure 12.1 and, using a gate-level simulator or a timing analyzer, obtain the delays from the a and b inputs to the outputs. What is the critical path?

12.6 (Verilog sensitivity list, 30 min.) Simulate the following Verilog module with the test pattern shown and explain your results.

```
module and2_bad(a, b, c); input a, b; output c; reg c;
// test pattern: (a b) = (1 1) (0 1) (0 0) (1 0) (1 1)
always@(a) c <= a & b;
endmodule
```

Can you synthesize this module as it is? What is the error message if you get one? If you can synthesize this module as it is, simulate the synthesized logic and compare the output with the Verilog simulation.

12.7 (Verilog decoder, 30 min.) Synthesize the following Verilog module with minimum-area constraint and then with maximum-speed constraint. Compare the resulting logic in each case.

```
module Decoder4_to_16(Enable, In_4, Out_16);
input Enable; input [3:0] In_4; output [15:0] Out_16;
reg [15:0] Out_16;
always @(Enable or In_4)
begin
Out_16 = 16'hzzzz;
if (Enable == 1) begin Out_16 = 16'h0000; Out_16[In_4] = 1; end
end
endmodule
```

What happens if you change the `if` statement to `if (Enable === 1)`?

12.8 (Verilog eight-input MUX, 20 min.) Synthesize the following code with maximum-speed constraint and then minimum-area constraint. Compare the results.

```
module Mux8_to_1(InBus, Select, OutEnable, OutBit);
```



```

input [7:0] InBus; input [2:0] Select; input OutEnable;
output OutBit; reg OutBit;
always @(OutEnable or Select or InBus)
  begin
    if (OutEnable == 1) OutBit = InBus[Select]; else OutBit = 1'bz;
  end
endmodule

```

12.9 (Verilog parity generator, 30 min.) Synthesize the following code with maximum-speed constraint and then minimum-area constraint. Compare the results.

```

module Parity (BusIn, ParityB);
  input[8:0] BusIn; output ParityB; reg ParityB;
  always @(BusIn) if (^Busin == 0 ) ParityB = 1; else ParityB = 0;
endmodule

```

12.10 (Verilog edges and levels, 30 min.) What is the function of the following model? List the cells produced by a logic synthesizer, their function, and an explanation of why they were synthesized.

```

module DD(D, C, R, Q, QB); input D, C, R; output Q, QB; reg Q, QB, L;
always @(posedge C or posedge R) if (R == 1) L = 0; else L = D;
always @(L) begin Q = L; QB = ~L; end
endmodule

```

12.11 (Verilog adders, 120 min.) Synthesize the following code with maximum-speed constraint and then minimum-area constraint. What type of adder architecture does the synthesis tool produce in each case (ripple-carry, lookahead, etc.)? Show exactly how you reached your conclusion. If you can, use either synthesis tool directives, shell commands, or standard components (Synopsys DesignWare or Xilinx X-BLOX, for example) to direct the synthesis tool to a specific adder implementation. Check that when you optimize the synthesized logic the adder architecture is not broken up. Next, if you can, find a way to make the synthesis tool break up the adder and reoptimize the logic. Does anything change?

```

module adder1(a, a, outp);
  input [3:0] a; input [3:0] b; output [3:0] outp; reg [3:0] outp;
  // if you can, change the next line to drive your synthesis tool
  // pragma|compass|synopsys|whatever max_delay constraint
  begin
    outp <= a + b; // Map me to DesignWare, X-Blox etc., if you can.
  end
endmodule

```

12.12 (Elementary gates in Verilog, 60 min.) Synthesize and optimize the following (you will have to write some more code to go around these statements):

```
And3 = &{In1,In2,In3}; Or3 = |{In1,In2,In3}; Xor3 = ^{In1,In2,In3};
```

This should produce three-input AND, OR, and XOR gates. Now synthesize and optimize eight-input AND, OR, and XOR gates in the same way with minimum-area

constraint and then maximum-speed constraint. Compare your results. How and why do the synthesis results and your answers change if you place a large capacitive load on the outputs. *Hint*: Try a load equivalent to 16 minimum-size inverters. Can you explain these results using logical effort?

12.13 (Synthesizable VHDL, 20 min.) Complete the following code fragment and try to synthesize the VHDL:

```
process begin
    wait until Clk = '1';
    Phase <= "0" after 0 ns; Phase <= "1" after 10 ns;
end process;
```

What is the error message? Synthesize this code, and explain the results:

```
process begin
    wait until Clk_x_2 = '1';
    case (Phase) is
        when '0' => Phase <= '1'; when others => Phase <= '0';
    end case;
end process;
```

12.14 (VHDL and process sensitivity list, 15 min.) Simulate the following code with the test input vectors shown:

```
entity AND2 is port (a, b in : BIT; c out : BIT); end AND2;
architecture Bad_behavior of AND2 is begin
    test inputs: (a b) = (1 1) (0 1) (0 0) (1 0) (1 1)
    process (a) begin c <= a and b; end process;
end;
```

Now try to synthesize this code. Do you get an error message? If not, try simulating the synthesized logic and compare with your earlier simulation results.

12.15 (MUX logic, 20 min.) Synthesize the following VHDL:

```
entity MuxLogic is
    port (InBus : in BIT_VECTOR(3 downto 0);
          Sel : in BIT_VECTOR(1 downto 0);
          OutBit : out BIT);
end MuxLogic;

architecture Synthesis_1 of MuxLogic is
    begin process (Sel, InBus)
        begin
            case Sel is
                when "00" => OutBit <= not(InBus(0));
                when "01" => OutBit <= InBus(1) and InBus(2);
                when "10" => OutBit <= InBus(2) or InBus(1);
                when "11" => OutBit <= InBus(3) xor InBus(0);
            end case;
        end process;
    end;
```

```
end process;
end Synthesis_1;
```

Does the synthesizer implement the case statement using a MUX? Explain your answer carefully by using the synthesis reports and the synthesized netlist. Try synthesizing again with minimum-area constraint and then maximum-speed constraint. Does this alter the implementation chosen by the synthesis tool? Explain.

12.16 (Arithmetic overflow in VHDL) Synthesize the following model (you will need arithmetic packages):

```
entity Adder1 is port (InBusA,
  InBusB : in Std_logic_vector(3 downto 0);
  OutBus : out Std_logic_vector(3 downto 0));
end Adder1;

architecture Behavior of Adder1 is begin OutBus <= InBusA + InBusB;
end Behavior;
```

Repeat the synthesis with the following modification and explain the difference:

```
OutBus : out Std_logic_vector(4 downto 0));
```

Finally, make the following additional modification and explain all your results:

```
OutBus <= ( "0" & InBusA) + ("0" & InBusB) ;
```

12.17 (Verilog integers, 30 min.) Consider the following Verilog module:

```
module TestIntegers (clk, out)
  integer i; reg [1:0] out; input clk; output out;
  always @(posedge clk) begin i = i + 1; out = i; end
endmodule
```

Write a test module for TestIntegers and simulate the behavior. Try to synthesize TestIntegers and explain what happens.

12.18 (Verilog shift register, 30 min.) Consider this code for a shift register:

```
module Shift1 (clk, q0, q1, q2)
  input clk, q0; output q2, q1; reg q2, q1;
  always (@ posedge clk) q1 = q0; always (@ posedge clk) q2 = q1;
endmodule
```

Write a module Test to exercise this module. Does it simulate correctly? Can you synthesize your code for Shift1 as it is? Change the body of the code as follows (call this module Shift2):

```
always (@ posedge clk) q1 = #1 q0; always (@ posedge clk) q2 = #1 q1;
```

Does this simulate correctly? Now change the code as follows (Shift3):

```
always (@ posedge clk) begin q1 = q0; q2 = q1; end
```

Does this simulate correctly? Can you synthesize Shift3? Finally, change the code to the following (Shift4):

```
always (@posedge clk) q1 <= q0; always (@posedge clk) q2 <= q1
```

Does this simulate correctly? Can you synthesize Shift4?

12.19 (Reset, 20 min.) Use simulation results to explain the difference between:

```
always (@posedge clk) if(clr) Q = 0;
always (@posedge clk) if(rst) Q = 1;
```

and

```
always (@posedge clk) begin if (clr) Q = 0; if (rst) Q = 1; end
```

12.20 (Verilog assignments, 30 min.) Consider the following Verilog module:

```
module TestAssign1(sel) input sel; reg outp;
  always @sel begin outp <= 1; if (sel) outp <= 0; end
endmodule
```

Write a module to drive TestAssign1 and simulate your code. Now consider the following modification (call this TestAssign2):

```
if (sel) outp <= 0; else outp <= 1;
```

Simulate TestAssign2 and compare your results. Try to synthesize TestAssign1 and TestAssign2. Comment on any problems you have and how you resolved them. Compare the behavior of the synthesized logic with the simulations.

12.21 (VHDL sequential logic, 60 min.) Consider the following processes:

```
S1: process (clk) begin
if clk'EVENT and clk = '1' then count <= count + inc; end if;
end process;

S2: process (clk) begin
  if rst = '1' then count <= 0;
  elsif clk'EVENT and clk = '1' then count <= count + inc;
end if;
end process;

S3: process (clk, rst) begin
  if rst = '1' then count <= 0; elsif clk'EVENT and clk = '1' then
    count <= count + inc; sum <= count + sum;
  end if;
end process;

S4: process (clk) begin
if clk'EVENT and clk = '1' then if rst = '1' then count <= 0;
  else count <= count + inc; end if;
end if;
end process;
```

```

S5: process (clk, rst) begin
  if rst = '1' then count <= 0;
  elsif clk'EVENT and clk = '1' then count <= count + inc;
  else count <= count + 1;
  end if;
end process;

S6: process (clk, rst) begin
  if rst = '1' then count <= 0;
  elsif clk'EVENT and clk = '1' then count <= count + inc;
  end if; inc <= not dec;
end process;

```

Write code to drive each of these processes and simulate them. Explain any errors or problems you encounter. Try to synthesize your code and check that the results behave correctly and match the simulation results. Explain any differences in behavior or any problems you encounter.

12.22 (Verilog signed multiplication, 30 min.) Show, by simulation, that the following code performs signed multiplication. Synthesize the code and compare the results with the simulation.

```

module Smpy (in1, in2, out); input [2:0] in1, in2; output [5:0] out;
assign out = {{3{in1[2]}},in1}*{{3{in2[2]}},in2};
endmodule

```

12.23 (Verilog arithmetic, 30 min.) Synthesize the following code and explain in detail the implementation that results:

```

module Arithmetic (in_4, out_2, out_3, out_7, out_14);
input [3:0] in_4; output [7:0] out_2, out_3, out_7, out_14;
assign out_2 = in_4*2; assign out_3 = in_4*3; assign out_7 = in_4*7;
assign out_14 = in_4 * 4'b1110;
endmodule

```

12.24 (Verilog overflow bit, 15 min.) Synthesize the following code and explain the implementation that results:

```

module Overflow (a, b, sum, cout);
input [7:0] a, b; output [7:0] sum; output cout;
assign {cout, sum} = a + b;
endmodule

```

12.25 (*VHDL latches, 60 min.) Consider the following two architectures:

```

entity latch1 is port(data: in BIT_VECTOR(1 to 4);
  reset: in BIT; delay: out BIT_VECTOR(1 to 4));
end latch1;

architecture Synthesis_1 of latch1 is
begin S1: process (data, reset) variable hold : BIT_VECTOR (1 to 4);

```

```

begin
    if reset = '1' then hold := "0000"; end if;
    delay <= hold; hold := data;
end process;
end Synthesis_1;

architecture Synthesis_2 of latch1 is
begin S2: process (data, enable, reset)
variable hold : BIT_VECTOR (1 to 4);
begin
if enable = '1' then hold := data; end if;
delay <= hold;
if reset = '0' then hold := "0000";
end process;
end Synthesis_2;

```

Try to synthesize both versions. Does the synthesizer accept the code? *Hint:* It should not. Explain any problems that you encounter, and how to correct them. Resynthesize your working code.

12.26 (*VHDL data slip, 60 min.) Consider the following process, a shift register core:

```

S1: process (data, enable) begin
if enable = '1' then Q <= Q(7 downto 0) & data; end if;
end process;

```

Complete the VHDL code and simulate to ensure your model operates correctly. Try to synthesize your code and compare the operation of the resulting implementation with the simulation results. Explain any problems you encounter.

12.27 (**Synchronous logic, hours) Investigate the following alternative ways to synthesize synchronous logic in VHDL. *Hint:* A few of these methods are illegal in both VHDL-87 and VHDL-93, some methods are only illegal in VHDL-87. Create a table for Q1-Q17 that summarizes your results. Assume all signals are STD_LOGIC. Can you create any more methods (positive-edge only)?

```

-- Let me count the ways to count.
-- Using wait statement:
process begin wait on clk; Q1 <= D; end process; -- 2 edges
process begin wait on clk until clk = '1'; Q2 <= D; end process;
process begin wait until clk = '1'; Q3 <= D; end process;
process begin wait until clk = '1' and clk'EVENT; Q4 <= D;
end process;
-- Using process and sensitivity list:
process(clk) begin if clk'EVENT and clk = '1' then Q5 <= D; end if;
end process;
process(clk) begin if not clk'STABLE and clk = '1' then Q6 <= D;
end if; end process;
process(clk) begin
if clk'LAST_VALUE = '0' and clk = '1' then Q7 <= D; end if;

```

```

end process;
-- Using rising_edge function from STD_LOGIC_1164:
process(clk) begin if rising_edge(clk) then Q8 <= D; end if;
end process;
process begin wait until rising_edge(clk); Q9 <= D; end process;
process begin wait on rising_edge(clk); Q10 <= D; end process;
-- rising_edge expanded:
process(clk) begin
if clk'EVENT and To_X01(clk) = '1'
    and To_X01(clk'LAST_VALUE) = '0' then Q11 <= D; end if;
end process;
-- Using concurrent signal assignments:
Q12 <= D when clk'EVENT and clk = '1'; -- VHDL-93 only (...else)
Q13 <= D when clk'EVENT and clk = '1' else Q13; -- need buffer
Q14 <= D when clk'EVENT and clk = '1' else unaffected; -- VHDL-93
Q15 <= D when clk'EVENT and clk = '1'
    else Q15'DRIVING_VALUE; -- VHDL-93
-- Using blocks:
F1:block(not clk'STABLE and clk = '1')
    begin Q16 <= guarded D;end block;
F2:block(clk'EVENT and clk = '1')
    begin Q17 <= guarded D;end block;
-- The bizarre and variations using '0', 'L', 'H', and '1':
process(clk) begin
if clk'LAST_VALUE = 'L' and clk = 'H' or clk = '1' then Q18 <= D;
    end if; end process;
process begin wait until clk = 'H' or clk = '1';
Q19 <= D; end process;
-- More?

```

12.28 (*State assignment, 30 min) If we have a state machine with r states and S_0 variables, how many different state assignments are there, for $S_0=1$ and $r=2$? List the different state assignments with $S_0=2$, $r=3$ and for $S_0=2$, $r=4$. How many of these are distinct? For five states and three state variables there are 6720 different state assignments, of which 140 are distinct. For nine states and four state variables there are over 4×10^9 different possible state assignments and nearly 11 million of these are distinct. This makes the task of performing sequential logic synthesis by exhaustively considering all possible state assignments virtually impossible. *Hint:* McCluskey's book discusses the problem of state assignment [1965, pp. 266–267].

12.29 (*Synthesis scripts, hours) Write and document a script to synthesize the Viterbi decoder using a logic synthesizer of your choice.

12.30 (*Floorplanning, hours) Write and document a script to perform timing-driven synthesis and floorplanning for the Viterbi decoder.

12.31 (***)Patents, 120 min.) Obtain a copy of U.S. Patent 5,530,841 “Method for converting a hardware independent user description of a logic circuit into hardware components.” This patent caused controversy during the approval of the IEEE synthesis packages. Research this topic (including a visit to the Web site of the synthesis package working group and checking other synthesis patents). Do you feel (as an engineer) that the IEEE should be concerned?

12.15 Bibliography

One way to learn more about logic synthesis is to obtain a copy of `misII` or `sis` (or their newest derivatives) from the University of California at Berkeley (UCB). These tools form the basis of most commercially available logic-synthesis software. Included with the `sis` distribution is a PostScript copy of a tutorial paper (available also as ERL Memorandum UCB/ERL M92/41) on logic synthesis by the UCB synthesis group. The internal help in `sis` explains the theory and purpose of each command. In addition each logic-synthesis step is available separately so it is possible to see the logic being synthesized, optimized, and mapped.

Programmable ASIC vendors, Xilinx, Altera, and Actel have each produced reports explaining how to use Synopsys, Mentor, Cadence, and other synthesis tools with their products. These are available on these companies' Web sites.

Brayton [1984] describes the detailed operation of `espresso`, one of the first logic-minimization programs, and the foundation of most modern commercial logic-synthesis tools. Edited books by Birtwistle and Subrahmanyam [1988] and Dutton [1991] contain a collection of papers on logic synthesis. The book by Thomas et al. [1990] describes an early logic-synthesis system. A tutorial paper by Brayton, Hachtel, and Sangiovanni-Vincentelli [1990] is an advanced description of multilevel logic optimization. In this chapter we have focused on RTL synthesis; the edited books by Camposano and Wolf [1991]; Walker and Camposano [1991]; and Michel, Lauther, and Duzy [1992] contain papers on higher-level, or behavioral-level synthesis. Edwards provides an overview of synthesis including references to earlier work [1992]. Gebotys and Elmasry [1992] cover system-level synthesis. Sasao [1993] is a selection of papers from a conference on logic synthesis. Kurup and Abbasi [1995] describe the Synopsys logic-synthesis tools. The book by Murgai et al. [1995] focuses on logic synthesis for FPGAs. De Micheli's book [1994] is a detailed work on logic-synthesis algorithms. Ashar et al. [1992] and Lavagno and Sangiovanni-Vincentelli [1993] cover sequential logic synthesis in their books. The book by Airiau, Berge, and Olive [1994] covers VHDL-93 from the perspective of logic synthesis. A book by Knapp [1996], describing the Synopsys behavioral compiler, is the closest to this book's treatment of logic synthesis, and includes several practical examples.

I have included references for a number of books (some not yet published) that I was unable to obtain before this book went to press including titles by Rushton

[1995] on logic synthesis using VHDL; Saucier [1995] on architectural synthesis; Hachtel and Somenzi [1996] on verification; Romdhane, Madisetti, and Hines [1996] on behavioral synthesis; and Villa et al. [1997] on FSM synthesis. I have included as much information as possible for these references including the LOC catalog information (it is possible to obtain an ISBN before publication).

12.16 References

- Airiau, R., J.-M. Berge, and V. Olive. 1994. *Circuit Synthesis with VHDL*. Boston, 221 p. ISBN 0792394291. TK7885.7.A37.
- Ashar, P. et al. 1992. *Sequential Logic Synthesis*. Norwell, MA: Kluwer, 225 p. ISBN 0-7923-9187-X. TK7868.L6.A84.
- Birtwistle, G., and P. A. Subrahmanyam (Ed.). 1988. *VLSI Specification, Verification, and Synthesis*. Boston: Kluwer, 404 p. ISBN 0898382467. TK7874.V564. A collection of papers presented at a workshop held in Calgary, Canada, Jan. 1987.
- Brayton, R. K. 1984. *Logic Minimization Algorithms for VLSI Synthesis*. Boston: Kluwer, 193 p. ISBN 0-89838-164-9. TK7868.L6L626. Includes an extensive bibliography. A complete description of espresso, the basis of virtually all commercial logic-synthesis tools. Difficult to read at first, but an excellent and clear description of the development of the algorithms used for two-level logic minimization.
- Brayton, R. K., G. D. Hachtel, and A. L. Sangiovanni-Vincentelli. 1990. "Multilevel logic synthesis." *Proceedings of the IEEE*, Vol. 78, no. 2, pp. 264–300.
- Camposano, R., and W. Wolf (Ed.). 1991. *High-level VLSI Synthesis*. Boston: Kluwer, 390 p. ISBN 0792391594. TK7874.H5243.
- De Micheli, G. 1994. *Synthesis and Optimization of Digital Circuits*. New York: McGraw-Hill, 579 p. ISBN 0070163332. TK7874.65.D4.
- Dutton, R. W. (Ed.). 1991. *VLSI Logic Synthesis and Design*. IOS Press. ISBN 905199046-4.
- Edwards, M. D. 1992. *Automated-Logic Synthesis Techniques for Digital Systems*. New York: McGraw-Hill, 186 p. ISBN 0-07-019417-3. TK7874.6.E34. Also Macmillan Press, Basingstoke, England, 1992. Includes an introduction to logic minimization and synthesis, and the topic of synthesis and testing.
- Gebotys, C. H., and M. I. Elmasry. 1992. *Optimal VLSI Architectural Synthesis: Area, Performance, and Testability*. Boston, 289 p. ISBN 079239223X. QA76.9.A73.G42.
- Hachtel, G. D., and F. Somenzi. 1996. *Logic Synthesis and Verification Algorithms*. Boston: Kluwer, 564 p. ISBN 0792397460. TK7874.75.H33.16 pages of references.
- Knapp, D. W. 1996. *Behavioral Synthesis: Digital System Design using the Synopsys Behavioral Compiler*. Upper Saddle River, NJ: Prentice-Hall, 231 p. ISBN 0-13-569252-0. A description of the Synopsys software. Includes the following code examples: FIR and IIR filters; Inverse Discrete Cosine Transform; random logic for a Data Encryption Standard (DES) ASIC; and a packet router. Appendix A contains a description of the details of creating DesignWare components. Appendix B describes the subsets of VHDL and Verilog that are understood by the Synopsys compiler. Includes a diskette containing the code from the book.
- Kurup, P., and T. Abbasi. 1995. *Logic Synthesis Using Synopsys*. Boston: Kluwer, 304 p. ISBN 0-7923-9582-4. TK7874.6.K87. Hints, tips, and problems with Synopsys synthesis tools.

- Synopsys has a technical support site on the World Wide Web for registered users of their tools. See also 2nd ed., 1997 ISBN 079239786X.
- Lavagno, L., and A. Sangiovanni-Vincentelli. 1993. *Algorithms for Synthesis and Testing of Asynchronous Circuits*. Boston: Kluwer, 339 p. ISBN 0792393643. TK7888.4.L38.
- McCluskey, E. J. 1965. *Introduction to the Theory of Switching Circuits*. New York: McGraw-Hill, 318 p. TK7888.3.M25.
- Michel, P., U. Lauther, and P. Duzy (Ed.). 1992. *The Synthesis Approach to Digital System Design*. Norwell: Kluwer, 415 p. ISBN 0792391993. TK7868.D5.S96. Includes 30 pages of references.
- Murgai, R., et al. 1995. *Logic Synthesis for Field-Programmable Gate Arrays*. Boston: Kluwer, 427 p. ISBN 0-7923-9596-4. TK7895.G36M87.
- Romdhane, M. S. B., V. K. Madiseti, and J. W. Hines. 1996. *Quick-Turnaround ASIC Design in VHDL: Core-Based Behavioral Synthesis*. Boston: Kluwer, 180 p. ISBN 0792397444. TK7874.6.R66. Includes 6 pages of references.
- Rushton, A. 1995. *VHDL for Logic Synthesis: An Introductory Guide for Achieving Design Requirements*. New York: McGraw-Hill, 254 p. ISBN 0077090926. TK7885.7.R87.
- Sasao, T. (Ed.). 1993. *Logic Synthesis and Optimization*. Boston: Kluwer. ISBN 0-7923-9308-2. TK7868.L6.L627. Papers from the International Symposium on Logic Synthesis and Microprocessor Architecture, Iizuka, Japan, July 1992.
- Saucier, G. 1995. *Logic and Architecture Synthesis*. New York: Chapman & Hall. ISBN 0412726904. Not cataloged by the Library of Congress at the time of this book's publication.
- Thomas, D. E., et al. 1990. *Algorithmic and Register-Transfer Level Synthesis: The System Architect's Workbench*. Boston: Kluwer. ISBN 0792390539. TK7874.A418.
- Villa, T., et al. 1997. *Synthesis of Finite State Machines: Logic Optimization*. Boston: Kluwer. ISBN 0792398920. TK7868.L6.S944. In Library of Congress catalog, but was not available at the time of this book's publication.
- Walker, R. A., and R. Camposano (Ed.). 1991. *A Survey of High-Level Synthesis Systems*. Boston: Kluwer, 182 p. ISBN 0792391586. TK7874.S857.

SIMULATION

13

- | | | | |
|------|----------------------------|-------|-----------------------------|
| 13.1 | Types of Simulation | 13.8 | Formal Verification |
| 13.2 | The Comparator/MUX Example | 13.9 | Switch-Level Simulation |
| 13.3 | Logic Systems | 13.10 | Transistor-Level Simulation |
| 13.4 | How Logic Simulation Works | 13.11 | Summary |
| 13.5 | Cell Models | 13.12 | Problems |
| 13.6 | Delay Models | 13.13 | Bibliography |
| 13.7 | Static Timing Analysis | 13.14 | References |

Engineers used to prototype systems to check their designs, often using a breadboard with connector holes, allowing them to plug in ICs and wires. Breadboarding was feasible when it was possible to construct systems from a few off-the-shelf TTL parts. It is impractical for prototyping an ASIC. Instead most ASIC design engineers turn to **simulation** as the modern equivalent of breadboarding.

13.1 Types of Simulation

Simulators are usually divided into the following categories or **simulation modes**:

- Behavioral simulation
- Functional simulation
- Static timing analysis
- Gate-level simulation
- Switch-level simulation
- Transistor-level or circuit-level simulation

This list is ordered from high-level to low-level simulation (high-level being more abstract, and low-level being more detailed). Proceeding from high-level to low-level simulation, the simulations become more accurate, but they also become progressively more complex and take longer to run. While it is just possible to perform a behavioral-level simulation of a complete system, it is impossible to perform a circuit-level simulation of more than a few hundred transistors.

There are several ways to create an imaginary simulation model of a system. One method models large pieces of a system as black boxes with inputs and outputs. This type of simulation (often using VHDL or Verilog) is called **behavioral simulation**. **Functional simulation** ignores timing and includes **unit-delay simulation**, which sets delays to a fixed value (for example, 1 ns). Once a behavioral or functional simulation predicts that a system works correctly, the next step is to check the timing performance. At this point a system is partitioned into ASICs and a **timing simulation** is performed for each ASIC separately (otherwise the simulation run times become too long). One class of timing simulators employs **timing analysis** that analyzes logic in a static manner, computing the delay times for each path. This is called **static timing analysis** because it does not require the creation of a set of test (or stimulus) vectors (an enormous job for a large ASIC). Timing analysis works best with synchronous systems whose maximum operating frequency is determined by the longest path delay between successive flip-flops. The path with the longest delay is the **critical path**.

Logic simulation or **gate-level simulation** can also be used to check the timing performance of an ASIC. In a gate-level simulator a logic gate or logic cell (NAND, NOR, and so on) is treated as a black box modeled by a function whose variables are the input signals. The function may also model the delay through the logic cell. Setting all the delays to unit value is the equivalent of functional simulation. If the timing simulation provided by a black-box model of a logic gate is not accurate enough, the next, more detailed, level of simulation is **switch-level simulation** which models transistors as switches—on or off. Switch-level simulation can provide more accurate timing predictions than gate-level simulation, but without the ability to use logic-cell delays as parameters of the models. The most accurate, but also the most complex and time-consuming, form of simulation is **transistor-level simulation**. A transistor-level simulator requires models of transistors, describing their nonlinear voltage and current characteristics.

Each type of simulation normally uses a different software tool. A **mixed-mode simulator** permits different parts of an ASIC simulation to use different simulation modes. For example, a critical part of an ASIC might be simulated at the transistor level while another part is simulated at the functional level. Be careful not to confuse mixed-level simulation with a mixed analog/digital simulator, these are **mixed-level simulators**.

Simulation is used at many stages during ASIC design. Initial **prelayout simulations** include logic-cell delays but no interconnect delays. Estimates of capacitance may be included after completing logic synthesis, but only after physical design is it possible to perform an accurate **postlayout simulation**.

13.2 The Comparator/MUX Example

As an example we borrow the model from Section 12.2, “A Comparator/MUX,”

```
// comp_mux.v //1
module comp_mux(a, b, outp); input [2:0] a, b; output [2:0] outp; //2
function [2:0] compare; input [2:0] ina, inb; //3
begin if (ina <= inb) compare = ina; else compare = inb; end //4
endfunction //5
assign outp = compare(a, b); //6
endmodule //7
```

We can use the following testbench to generate a sequence of input values (we call these **input vectors**) that test or **exercise** the behavioral model, `comp_mux.v`:

```
// testbench.v //1
module comp_mux_testbench; //2
integer i, j; //3
reg [2:0] x, y, smaller; wire [2:0] z; //4
always @(x) $display("t x y actual calculated"); //5
initial $monitor("%4g", $time, x, y, z, smaller); //6
initial $dumpvars; initial #1000 $finish; //7
initial //8
begin //9
for (i = 0; i <= 7; i = i + 1) //10
begin //11
for (j = 0; j <= 7; j = j + 1) //12
begin //13
x = i; y = j; smaller = (x <= y) ? x : y; //14
#1 if (z != smaller) $display("error"); //15
end //16
end //17
end //18
comp_mux v_1 (x, y, z); //19
endmodule //20
```

The results from the behavioral simulation are as follows:

```
t x y actual calculated
0 0 0 0 0
1 0 1 0 0
... 60 lines omitted...
62 7 6 6 6
63 7 7 7 7
```

We included a delay of one Verilog time unit in line 15 of the testbench model (allowing time to progress), but we did not specify the units—they could be nanoseconds or days. Thus, behavioral simulation can only tell us if our design does not work; it cannot tell us that real hardware will work.

13.2.1 Structural Simulation

We use logic synthesis to produce a structural model from a behavioral model. The following comparator/MUX model is adapted from the example in Section 12.11, “Performance-Driven Synthesis” (optimized for a 0.6 μm standard-cell library):

```

`timescale 1ns / 10ps // comp_mux_o2.v //1
module comp_mux_o (a, b, outp); //2
input [2:0] a; input [2:0] b; //3
output [2:0] outp; //4
supply1 VDD; supply0 VSS; //5
mx21d1 b1_i1 (.i0(a[0]), .i1(b[0]), .s(b1_i6_zn), .z(outp[0])); //6
oa03d1 b1_i2 (.a1(b1_i9_zn), .a2(a[2]), .b1(a[0]), .b2(a[1]), //7
.c(b1_i4_zn), .zn(b1_i2_zn)); //8
nd02d0 b1_i3 (.a1(a[1]), .a2(a[0]), .zn(b1_i3_zn)); //9
nd02d0 b1_i4 (.a1(b[1]), .a2(b1_i3_zn), .zn(b1_i4_zn)); //10
mx21d1 b1_i5 (.i0(a[1]), .i1(b[1]), .s(b1_i6_zn), .z(outp[1])); //11
oa04d1 b1_i6 (.a1(b[2]), .a2(b1_i7_zn), .b(b1_i2_zn), //12
.zn(b1_i6_zn)); //13
in01d0 b1_i7 (.i(a[2]), .zn(b1_i7_zn)); //14
an02d1 b1_i8 (.a1(b[2]), .a2(a[2]), .z(outp[2])); //15
in01d0 b1_i9 (.i(b[2]), .zn(b1_i9_zn)); //16
endmodule //17

```

Logic simulation requires Verilog models for the following six logic cells: mx21d1 (2:1 MUX), oa03d1 (OAI221), nd02d0 (two-input NAND), oa04d1 (OAI21), in01d0 (inverter), and an02d1 (two-input AND). These models are part of an ASIC library (often encoded so that they cannot be seen) and thus, from this point on, the designer is dependent on a particular ASIC library company. As an example of this dependence, notice that some of the names in the preceding code have changed from uppercase (in Figure 12.8 on p. 624) to lowercase. Verilog is case sensitive and we are using a cell library that uses lowercase. Most unfortunately, there are no standards for names, cell functions, or the use of case in ASIC libraries.

The following code (a simplified model from a 0.8 μm standard-cell library) models a 2:1 MUX and uses fixed delays:

```

`timescale 1 ns / 10 ps //1
module mx21d1 (z, i0, i1, s); input i0, i1, s; output z; //2
not G3(N3, s); //3
and G4(N4, i0, N3), G5(N5, s, i1), G6(N6, i0, i1); //4
or G7(z, N4, N5, N6); //5
specify //6
(i0*>z) = (0.279:0.504:0.900, 0.276:0.498:0.890); //7
(i1*>z) = (0.248:0.448:0.800, 0.264:0.476:0.850); //8
(s*>z) = (0.285:0.515:0.920, 0.298:0.538:0.960); //9

```

```
endspecify //10
endmodule //11
```

This code uses Verilog primitive models (not, and, or) to describe the behavior of a MUX, but this is not how the logic cell is implemented.

To simulate the optimized structural model, module `comp_mux_o2.v`, we use the library cell models (module `mx21d1` and the other five that are not shown here) together with the following new testbench model:

```
`timescale 1 ps / 1 ps // comp_mux_testbench2.v //1
module comp_mux_testbench2; //2
integer i, j; integer error; //3
reg [2:0] x, y, smaller; wire [2:0] z, ref; //4
always @(x) $display("t x y derived reference"); //5
// initial $monitor("%8.2f", $time/1e3, x, y, z, ref); //6
initial $dumpvars; //7
initial begin //8
    error = 0; #1e6 $display("%4g", error, " errors"); //9
    $finish; //10
end //11
initial begin //12
    for (i = 0; i <= 7; i = i + 1) begin //13
        for (j = 0; j <= 7; j = j + 1) begin //14
            x = i; y = j; #10e3; //15
            $display("%8.2f", $time/1e3, x, y, z, ref); //16
            if (z != ref) //17
                begin $display("error"); error = error + 1; end //18
            end //19
        end //20
    end //21
end //21
comp_mux_o v_1 (x, y, z); // comp_mux_o2.v //22
reference v_2 (x, y, ref); //23
endmodule //24

// reference.v //1
module reference(a, b, outp); //2
input [2:0] a, b; output [2:0] outp; //3
    assign outp = (a <= b) ? a : b; // different from comp_mux //4
endmodule //5
```

In this testbench we have instantiated two models: a *reference model* (module `reference`) and a *derived model* (module `comp_mux_o`, the optimized structural model). The high-level behavioral model that represents the initial system specification (module `reference`) may be different from the model that we use as input to the logic-synthesis tool (module `comp_mux`). Which is the real reference model? We

postpone this question until we discuss *formal verification* in Section 13.8. For the moment, we shall simply perform simulations to check the reference model against the derived model. The simulation results are as follows:

```
t          x y derived reference
  10.00  0 0 0          0
  20.00  0 1 0          0
... 60 lines omitted...
 630.00  7 6 6          6
 640.00  7 7 7          7
      0 errors
```

(A summary is printed at the end of the simulation to catch any errors.) The next step is to examine the timing of the structural model (by switching the leading `'//'` from line 6 to 16 in module `comp_mux_testbench2`). It is important to simulate using the worst-case delays by using a command-line **switch** as follows: `verilog +maxdelays`. We can then find the longest path delay by searching through the simulator output, part of which follows:

```
t          x y derived reference
... lines omitted...
 260.00  3 2 1          2
 260.80  3 2 3          2
 260.85  3 2 2          2
 270.00  3 3 2          3
 270.80  3 3 3          3
 280.00  3 4 3          3
 280.85  3 4 0          3
 283.17  3 4 3          3
... lines omitted...
```

At time 280 ns, the input vectors, x and y , switch from $(x = 3, y = 3)$ to $(x = 3, y = 4)$. The output of the derived model (which should be equal to the smaller of x and y) is the same for both of these input vectors and should remain unchanged. In fact there is a glitch at the output of the derived model, as it changes from 3 to 0 and back to 3 again, taking 3.17 ns to settle to its final value (this is the longest delay that occurs using this testbench). The glitch occurs because one of the input vectors (input y) changes from `'011'` (3 in decimal) to `'100'` (decimal 4). Changing several input bits simultaneously causes the output to vacillate.

Notice that the nominal and worst-case simulations will not necessarily give the same longest path delay. In addition the longest path delay found using this testbench is not necessarily the critical path delay. For example, the longest, and therefore critical, path delay might result from a transition from $x = 3, y = 4$ to $x = 4, y = 3$ (to choose a random but possible candidate set of input vectors). This testbench does not include tests with such transitions. To find the critical path using logic simulation requires simulating all possible input transitions ($64 \times 64 = 4096$) and then sifting through the output to find the critical path.

Vector-based simulation (or **dynamic simulation**) can show us that our design functions correctly—hence the name functional simulation. However, functional simulation does not work well if we wish to find the critical path. For this we turn to a different type of simulation—static simulation or static timing analysis.

TABLE 13.1 Timing analysis of the comparator/MUX structural model, `comp_mux_o2.v`, from Figure 12.8.

Command	Timing analyzer/logic synthesizer output ¹						
> report	instance name						
timing	inPin --> outPin	incr	arrival	trs	rampDel	cap	cell
		(ns)	(ns)		(ns)	(pF)	
	a[0]	.00	.00	R	.00	.12	comp_m...
	b1_i3						
	A2 --> ZN	.31	.31	F	.23	.08	nd02d0
	b1_i4						
	A2 --> ZN	.41	.72	R	.26	.07	nd02d0
	b1_i2						
	C --> ZN	1.36	2.08	F	.13	.07	oa03d1
	b1_i6						
	B --> ZN	.94	3.01	R	.24	.14	oa04d1
	b1_i5						
	S --> Z	1.04	4.06	F	.08	.04	mx21d1
	outp[0]	.00	4.06	F	.00	.00	comp_m...

¹Using a 0.8 μm standard-cell library, VLSI Technology `vsc450`. Worst-case environment: worst-case process, $V_{DD}=4.75\text{ V}$, and $T=70^\circ\text{C}$. No wire capacitance, no input or output capacitance, prop-ramp timing model. The structural model was synthesized and optimized using a 0.6 μm library, but this timing analysis was performed using the 0.8 μm library. This is because the library models are simpler for the 0.8 μm library and thus easier to explain in the text.

13.2.2 Static Timing Analysis

A timing analyzer answers the question: “What is the longest delay in my circuit?” Table 13.1 shows the timing analysis of the comparator/MUX structural model, module `comp_mux_o2.v`. The longest or critical path delay is 4.06 ns under the following worst-case operating conditions: worst-case process, $V_{DD}=4.75\text{ V}$, and $T=70^\circ\text{C}$ (the same conditions as used for the library data book delay values). The timing analyzer gives us only the critical path and its delay. A timing analyzer does not give us the input vectors that will activate the critical path. In fact input vectors may not exist to activate the critical path. For example, it may be that the decimal values of the input vectors to the comparator/MUX may never differ by more than four, but the timing-analysis tool cannot use this information. Future timing-analysis tools may consider such factors, called **Boolean relations**, but at present they do not.

Section 13.2.1 explained why dynamic functional simulation does not necessarily find the critical path delay. Nevertheless, the difference between the longest path delay found using functional simulation, 3.17 ns, and the critical path delay reported by the static timing-analysis tool, 4.06 ns, is surprising. This difference occurs because the timing analysis accounts for the loading of each logic cell by the input capacitance of the logic cells that follow, but the simplified Verilog models used for functional simulation in Section 13.2.1 did not include the effects of capacitive loading. For example, in the model for the logic cell `mx21d1`, the (rising) delay from the `i0` input to the output `z`, was fixed at 0.900 ns worst case (the maximum delay value is the third number in the first triplet in line 7 of module `mx21d1`). Normally library models include another portion that adjusts the timing of each logic cell—this portion was removed to simplify the model `mx21d1` shown in Section 13.2.1.

Most timing analyzers do not consider the function of the logic when they search for the critical path. Thus, for example, the following code models `z = NAND(a, NOT(a))`, which means that the output, `z`, is always '1'.

```

module check_critical_path_1 (a, z); //1
input a; output z; supply1 VDD; supply0 VSS; //2
nd02d0 b1_i3 (.a1(a), .a2(b), .zn(z)); // 2-input NAND //3
in01d0 b1_i7 (.i(a), .zn(b)); // inverter //4
endmodule //5

```

A timing-analyzer report for this model might show the following critical path:

inPin --> outPin	incr (ns)	arrival (ns)	trs	rampDel (ns)	cap (pf)	cell
a	.00	.00	R	.00	.08	check_...
b1_i7						
I --> ZN	.38	.38	F	.30	.07	in01d0
b1_i3						
A2 --> ZN	.28	.66	R	.13	.04	nd02d0
z	.00	.66	R	.00	.00	check_...

Paths such as this, which are impossible to activate, are known as **false paths**. Timing analysis is essential to ASIC design but has limitations. A timing-analysis tool is more logic calculator than logic simulator.

13.2.3 Gate-Level Simulation

To illustrate the differences between functional simulation, timing analysis, and gate-level simulation, we shall simulate the comparator/MUX critical path (the path is shown in Table 13.1). We start by trying to find vectors that activate this critical path by working forward from the beginning of the critical path, the input `a[0]`, toward the end of the critical path, output `outp[0]`, as follows:

1. Input `a[0]` to the two-input NAND, `nd02d0`, cell instance `b1_i3`, changes from a '0' to a '1'. We know this because there is an 'R' (for rising) under

the `trs` (for transition) heading on the first line of the critical path timing analysis report in Table 13.1.

2. Input `a[1]` to the two-input NAND, `nd02d0`, cell instance `b1_i3`, must be a '1'. This allows the change on `a[0]` to propagate toward the output, `outp[0]`.
3. Similarly, input `b[1]` to the two-input NAND, cell instance `b1_i4`, must be a '1'.
4. We skip over the required inputs to cells `b1_i2` and `b1_i6` for the moment.
5. From the last line of Table 13.1 we know the output of MUX, `mx21d1`, cell instance `b1_i5`, changes from '1' to a '0'. From the previous line in Table 13.1 we know that the select input of this MUX changes from '0' to a '1'. This means that the final value of input `b[0]` (the `i1` input, selected when the select input is '1') must be '0' (since this is the final value that must appear at the MUX output). Similarly, the initial value of `a[0]` must be a '1'.

We have now contradicted ourselves. In step 1 we saw that the initial value of `a[0]` must be a '0'. The critical path is thus a false path. Nevertheless we shall proceed. We set the initial input vector to (`a = '110'`, `b = '111'`) and then to (`a = '111'`, `b = '110'`). These vectors allow the change on `a[0]` to propagate to the select signal of the MUX, `mx21d1`, cell instance `b1_i5`. In decimal we are changing `a` from 6 to 7, and `b` from 7 to 6; the output should remain unchanged at 6. The simulation results from the gate-level simulator we shall use (CompassSim) can be displayed graphically or in the text form that follows:

```
...
# The calibration was done at Vdd=4.65V, Vss=0.1V, T=70 degrees C
Time = 0:0 [0 ns]
      a = 'D6 [0] (input)(display)
      b = 'D7 [0] (input)(display)
      outp = 'Buuu ('Du) [0] (display)
      outp --> 'B1uu ('Du) [.47]
      outp --> 'B11u ('Du) [.97]
      outp --> 'D6 [4.08]
      a --> 'D7 [10]
      b --> 'D6 [10]
      outp --> 'D7 [10.97]
      outp --> 'D6 [14.15]
Time = 0:0 +20ns [20 ns]
```

The code 'Buuu denotes that the output is initially, at $t=0$ ns, a binary vector of three unknown or **unsettled** signals. The output bits become valid as follows: `outp[2]` at 0.47 ns, `outp[1]` at 0.97 ns, and `outp[0]` at 4.08 ns. The output is stable at 'D6 (decimal 6) or '110' at $t=10$ ns when the input vectors are changed in an attempt to activate the critical path. The output glitches from 'D6 ('110') to 'D7 ('111') at $t=10.97$ ns and back to 'D6 again at $t=14.15$ ns. Thus, the output bit, `outp[0]`, takes a total of 4.15 ns to settle.

Can we explain this behavior? The data book entry for the `mx21d1` logic cell gives the following equation for the rising delay as a function of `C1d` (the load capacitance, excluding the output capacitance of the logic cell itself, expressed in picofarads):

$$t_{IOZ} (IO \rightarrow Z) = 0.90 + 0.07 + (1.76 \times C1d) \text{ ns} \quad (13.1)$$

The capacitance, `C1d`, at the output of each MUX is zero (because nothing is connected to the outputs). From Eq. 13.1, the path delay from the input, `a[0]`, to the output, `outp[0]`, is thus 0.97 ns. This explains why the output, `outp[0]`, changes from '0' to '1' at $t = 10.97$ ns, 0.97 ns after a change occurs on `a[0]`.

The gate-level simulation predicts that the input, `a[0]`, to the MUX will change before the changes on the inputs have time to propagate to the MUX select. Finally, at $t = 14.15$ ns, the MUX select will change and switch the output, `outp[0]`, back to '0' again. The total delay for this input vector stimulus is thus 4.15 ns. Even though this path is a false path (as far as timing analysis is concerned), it is a critical path. It is indeed necessary to wait for 4.15 ns before using the output signal of this circuit. A timing analyzer can only offer us a guarantee that there is no other path that is slower than the critical path.

13.2.4 Net Capacitance

The timing analyzer predicted a critical path delay of 4.06 ns compared to the gate-level simulation prediction of 4.15 ns. We can check our results by using another gate-level simulator (QSim) which uses a slightly different algorithm. Here is the output (with the same input vectors as before):

```
@nodes
a R10 W1; a[2] a[1] a[0]
b R10 W1; b[2] b[1] b[0]
outp R10 W1; outp[2] outp[1] outp[0]
@data
      .00          a -> 'D6
      .00          b -> 'D7
      .00          outp -> 'Du
      .53          outp -> 'Du
      .93          outp -> 'Du
      4.42         outp -> 'D6
     10.00         a -> 'D7
     10.00         b -> 'D6
     11.03         outp -> 'D7
     14.43         outp -> 'D6
### END OF SIMULATION TIME = 20 ns
@end
```

The output is similar but gives yet another value, 4.43 ns, for the path delay. Can this be explained? The simulator prints the following messages as a clue:

```
defCapacitance      = .1E-01 pF
incCapacitance     = .1E-01 pF/pin
```

The simulator is adding capacitance to the outputs of each of the logic cells to model the parasitic **net capacitance** (interconnect capacitance or wire capacitance) that will be present in the physical layout. The simulator adds 0.01 pF (`defCapacitance`) on each node and another 0.01 pF (`incCapacitance`) for each pin (logic cell input) attached to a node. The model that predicts these values is known as a **wire-load model**, **wire-delay model**, or **interconnect model**. Changing the wire-load model parameters to zero and repeating the simulation changes the critical-path delay to 4.06 ns, which agrees exactly with the logic-synthesizer timing analysis. This emphasizes that the net capacitance may contribute a significant delay.

The library data book (VLSI Technology, `vsc450`) lists the cell input and output capacitances. For example, the values for the `nd02d0` logic cell are as follows:

$$C_{in}(\text{inputs, } a1 \text{ and } a2) = 0.042 \text{ pF} \quad C_{out}(\text{output, } z_n) = 0.038 \text{ pF} \quad (13.2)$$

Armed with this information, let us return to the timing analysis report of Table 13.1 on page 647 (the part of this table we shall focus on follows) and examine how a timing analyzer handles net capacitance.

inPin --> outPin	incr (ns)	arrival (ns)	trs	rampDel (ns)	cap (pf)	cell
a[0]	.00	.00	R	.00	.12	comp_m...
b1_i3						
A2 --> ZN	.31	.31	F	.23	.08	nd02d0
...						

The total capacitance at the output node of logic cell instance `b1_i3` is 0.08 pF. This figure is the sum of the logic cell (`nd02d0`) output capacitance of cell instance `b1_i3` (equal to 0.038 pF) and `C1d`, the input capacitance of the next cell, `b1_i2` (also an `nd02d0`), equal to 0.042 pF.

The capacitance at the input node, `a[0]`, is equal to the sum of the input capacitances of the logic cells connected to that node. These capacitances (and their sources) are as follows:

1. 0.042 pF (the `a2` input of the two-input NAND, instance `b1_i3`, cell `nd02d0`)
2. 0.038 pF (the `i0` input of the 2:1 MUX, instance `b1_i1`, cell `mx21d1`)
3. 0.038 pF (the `b1` input of the OAI221, instance `b1_i2`, cell `oa03d1`)

The sum of these capacitances is the 0.12 pF shown in the timing-analysis report.

Having explained the capacitance figures in the timing-analysis report, let us turn to the delay figures. The fall-time delay equation for a `nd02d0` logic cell (again from the `vsc450` library data book) is as follows:

$$\tau_D(\text{AX} \rightarrow \text{ZN}) = 0.08 + 0.11 + (2.89 \times \text{C1d}) \text{ ns} \quad (13.3)$$

Notice $0.11 \text{ ns} = 2.89 \text{ nspF}^{-1} \times 0.038 \text{ pF}$, and this figure in Eq. 13.3 is the part of the cell delay attributed to the cell output capacitance. The ramp delay in the timing analysis (under the heading `rampDe1` in Table 13.1) is the sum of the last two terms in Eq. 13.3. Thus, the ramp delay is $0.11 + (2.89 \times 0.042) = 0.231 \text{ ns}$ (since `C1d` is 0.042 pF). The total delay (under `incr` in Table 13.1) is $0.08 + 0.231 = 0.31 \text{ ns}$.

There are thus the following four figures for the critical path delay:

1. 4.06 ns from a static timing analysis using the logic-synthesizer timing engine (worst-case process, $V_{DD} = 4.50 \text{ V}$, and $T = 70^\circ\text{C}$). No wire capacitance.
2. 4.15 ns from a gate-level functional simulation (worst-case process, $V_{SS} = 0.1 \text{ V}$, $V_{DD} = 4.65 \text{ V}$, and $T = 70^\circ\text{C}$). No wire capacitance.
3. 4.43 ns from a gate-level functional simulation. Default wire-capacitance model ($0.01 \text{ pF} + 0.01 \text{ pF/pin}$).
4. 4.06 ns from a gate-level functional simulation. No wire capacitance.

Normally we do not check our simulation results this thoroughly. However, we can only trust the tools if we understand what they are doing, how they work, their limitations, and we are able to check that the results are reasonable.

13.3 Logic Systems

Digital signals are actually analog voltage (or current) levels that vary continuously as they change. **Digital simulation** assumes that digital signals may only take on a set of **logic values** (or **logic states**—here we will consider the two terms equivalent) from a **logic system**. A logic system must be chosen carefully. Too many values will make the simulation complicated and slow. With too few values the simulation may not accurately reflect the hardware performance.

A **two-value logic system** (or two-state logic system) has a logic value '0' corresponding to a **logic level** 'zero' and a logic value '1' corresponding to a logic level 'one'. However, when the power to a system is initially turned on, we do not immediately know whether the logic value of a flip-flop output is '1' or '0' (it will be one or the other, but we do not know which). To model this situation we introduce a logic value 'x', with an unknown logic level, or **unknown**. An unknown can **propagate** through a circuit. For example, if the inputs to a two-input NAND gate are logic values '1' and 'x', the output is logic value 'x' or unknown. Next, in order to model a three-state bus, we need a **high-impedance state**. A high-impedance state may have a logic level of 'zero' or 'one', but it is not being driven—we say

it is floating. This will occur if none of the gates connected to a three-state bus is driving the bus. A **four-value logic system** is shown in Table 13.2.

TABLE 13.2 A four-value logic system.

Logic state	Logic level	Logic value
0	zero	zero
1	one	one
X	zero or one	unknown
Z	zero, one, or neither	high impedance

13.3.1 Signal Resolution

What happens if multiple drivers try to drive different logic values onto a bus? Table 13.3 shows a **signal-resolution function** for a four-value logic system that will predict the result.

TABLE 13.3 A resolution function $R\{A, B\}$ that predicts the result of two drivers simultaneously attempting to drive signals with values A and B onto a bus.

$R\{A, B\}$	B = 0	B = 1	B = X	B = Z
A = 0	0	X	X	0
A = 1	X	1	X	1
A = X	X	X	X	X
A = Z	0	1	X	Z

A resolution function, $R\{A, B\}$, must be **commutative** and **associative**. That is,

$$R\{A, B\} = R\{B, A\} \quad \text{and} \quad R\{R\{A, B\}, C\} = R\{A, R\{B, C\}\}. \quad (13.4)$$

Equation 13.4 ensures that, if we have three (or more) signals to resolve, it does not matter in which order we resolve them. Suppose we have four drivers on a bus driving values '0', '1', 'X', and 'Z'. If we use Table 13.3 three times to resolve these signals, the answer is always 'X' whatever order we use.

13.3.2 Logic Strength

In CMOS logic we use n -channel transistors to produce a logic level 'zero' (with a forcing strength) and we use p -channel transistors to force a logic level 'one'. An

n -channel transistor provides a weak logic level 'one'. This is a new logic value, a **resistive 'one'**, which has a logic level of 'one', but with **resistive strength**. Similarly, a p -channel transistor produces a **resistive 'zero'**. A resistive strength is not as strong as a forcing strength. At a high-impedance node there is nothing to keep the node at any logic level. We say that the logic strength is **high impedance**. A high-impedance strength is the weakest strength and we can treat it as either a very high-resistance connection to a power supply or no connection at all.

TABLE 13.4 A 12-state logic system.

Logic strength	Logic level		
	zero	unknown	one
strong	S0	SX	S1
weak	W0	WX	W1
high impedance	Z0	ZX	Z1
unknown	U0	UX	U1

With the introduction of logic strength, a logic value may now have two properties: level and strength. Suppose we were to measure a voltage at a node N with a digital voltmeter (with a very high input impedance). Suppose the measured voltage at node N was 4.98 V (and the measured positive supply, $V_{DD} = 5.00$ V). We can say that node N is a logic level 'one', but we do not know the logic strength. Now suppose you connect one end of a 1 k Ω resistor to node N , the other to GND, and the voltage at N changes to 4.95 V. Now we can say that whatever is driving node N has a strong forcing strength. In fact, we know that whatever is driving N is capable of supplying a current of at least $4.95 \text{ V} / 1 \text{ k}\Omega \approx 5 \text{ mA}$. Depending on the logic-value system we are using, we can assign a logic value to N . If we allow all possible combinations of logic level with logic strength, we end up with a matrix of logic values and logic states. Table 13.4 shows the 12 states that result with three logic levels (zero, one, unknown) and four logic strengths (strong, weak, high-impedance, and unknown). In this logic system, node N has logic value S1—a logic level of 'one' with a logic strength of 'strong'.

The Verilog logic system has three logic levels that are called '1', '0', and 'x'; and the eight logic strengths shown in Table 13.5. The designer does not normally see the logic values that result—only the three logic levels.

The IEEE Std 1164-1993 logic system defines a variable type, `std_ulogic`, with the nine logic values shown in Table 13.6. When we wish to simulate logic cells using this logic system, we must define the primitive-gate operations. We also

TABLE 13.5 Verilog logic strengths.

Logic strength	Strength number	Models	Abbreviation	
supply drive	7	power supply	supply	Su
strong drive	6	default gate and assign output strength	strong	St
pull drive	5	gate and assign output strength	pull	Pu
large capacitor	4	size of trireg net capacitor	large	La
weak drive	3	gate and assign output strength	weak	We
medium capacitor	2	size of trireg net capacitor	medium	Me
small capacitor	1	size of trireg net capacitor	small	Sm
high impedance	0	not applicable	highz	Hi

TABLE 13.6 The nine-value logic system, IEEE Std 1164-1993.

Logic state	Logic value	Logic state	Logic value
'0'	strong low	'X'	strong unknown
'1'	strong high	'W'	weak unknown
'L'	weak low	'Z'	high impedance
'H'	weak high	'_'	don't care
		'U'	uninitialized

need to define the process of **VHDL signal resolution** using **VHDL signal-resolution functions**. For example, the function in the IEEE Std_Logic_1164 package that defines the and operation is as follows¹:

```
function "and"(l,r : std_ulogic_vector) return std_ulogic_vector is --1
  alias lv : std_ulogic_vector (1 to l'LENGTH) is l; --2
  alias rv : std_ulogic_vector (1 to r'LENGTH) is r; --3
variable result : std_ulogic_vector (1 to l'LENGTH); --4
constant and_table : stdlogic_table := ( --5
----- --6
--| U X 0 1 Z W L H - | | --7
----- --8
( 'U', 'U', '0', 'U', 'U', 'U', '0', 'U', 'U' ), -- | U | --9
( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ), -- | X | --10
( '0', '0', '0', '0', '0', '0', '0', 'U', '0' ), -- | 0 | --11
```

¹IEEE Std 1164-1993, © Copyright 1993 IEEE. All rights reserved.

```

( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ), -- | 1 |           --12
( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ), -- | Z |           --13
( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ), -- | W |           --14
( '0', '0', '0', '0', '0', '0', '0', '0', '0' ), -- | L |           --15
( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ), -- | H |           --16
( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ), -- | - | );         --17
begin                                                                --18
    if (l'LENGTH /= r'LENGTH) then assert false report              --19
"arguments of overloaded 'and' operator are not of the same        --20
length"                                                              --21
        severity failure;                                          --22
    else                                                            --23
        for i in result'RANGE loop                                  --24
            result(i) := and_table ( lv(i), rv(i) );                --25
        end loop;                                                  --26
    end if;                                                         --27
    return result;                                                 --28
end "and";                                                         --29

```

If a = 'X' and b = '0', then (a and b) is '0' no matter whether a is, in fact, '0' or '1'.

13.4 How Logic Simulation Works

The most common type of digital simulator is an **event-driven simulator**. When a circuit node changes in value the time, the node, and the new value are collectively known as an **event**. The event is scheduled by putting it in an **event queue** or **event list**. When the specified time is reached, the logic value of the node is changed. The change affects logic cells that have this node as an input. All of the affected logic cells must be **evaluated**, which may add more events to the event list. The simulator keeps track of the current time, the current **time step**, and the event list that holds future events. For each circuit node the simulator keeps a record of the logic state and the strength of the source or sources driving the node. When a node changes logic state, whether as an input or as an output of a logic cell, this causes an event.

An **interpreted-code simulator** uses the HDL model as data, compiling an executable model as part of the simulator structure, and then executes the model. This type of simulator usually has a short compile time but a longer execution time compared to other types of simulator. An example is Verilog-XL. A **compiled-code simulator** converts the HDL model to an intermediate form (usually C) and then uses a separate compiler to create executable binary code (an executable). This results in a longer compile time but shorter execution time than an interpreted-code simulator. A **native-code simulator** converts the HDL directly to an executable and offers the shortest execution time.

The logic cells for each of these types of event-driven simulator are modeled using a primitive modeling language (primitive in the sense of “fundamental”). There are no standards for this primitive modeling language. For example, the following code is a primitive model of a two-input NAND logic cell:

```
model nd01d1 (a, b, zn)
function (a, b) !(a & b); function end
model end
```

The model has three ports: a, b, and zn. These ports are connected to nodes when a NAND gate is instantiated in an input structural netlist,

```
nand nd01d1(a2, b3, r7)
```

An event occurs when one of the circuit nodes a2 or b3 changes, and the function defined in the primitive model is called. For example, when a2 changes, it affects the port a of the model. The function will be called to set zn to the logical NAND of a and b. The implementation of the primitive functions is unique to each simulator and carefully coded to reduce execution time.

The data associated with an event consists of the affected node, a new logic value for the node, a time for the change to take effect, and the node that caused the event. Written in C, the data structure for an event might look like the following:

```
struct Event {
    event_ptr fwd_link, back_link; /* event list */
    event_ptr node_link;          /* list of node events */
    node_ptr event_node;          /* node for the event */
    node_ptr cause;               /* node causing event */
    port_ptr port;                /* port which caused this event */
    long event_time;              /* event time, in units of delta */
    char new_value;               /* new value: '1' '0' etc. */
};
```

The event list keeps track of logic cells whose outputs are changing and the new values for each output. The **evaluation list** keeps track of logic cells whose inputs have changed. Using separate event and evaluation lists avoids any dependence on the order in which events are processed, since the evaluations occur only after all nodes have been updated. The sequence of event-list processing followed by the evaluation-list processing is called a **simulation cycle**, or an **event–evaluation cycle** (or event–eval cycle for short).

Delays are tracked using a **time wheel** divided into ticks or slots, with each slot representing a unit of time. A software pointer marks the current time on the timing wheel. As simulation progresses, the pointer moves forward by one slot for each time step. The event list tracks the events pending and, as the pointer moves, the simulator processes the event list for the current time.

13.4.1 VHDL Simulation Cycle

We shall use VHDL as an example to illustrate the steps in a **simulation cycle** (which is precisely defined in the LRM). In VHDL, before simulation begins, the design hierarchy is first **elaborated**. This means all the pieces of the model code (entities, architectures, and configurations) are put together. Then the nets in the model are initialized just before simulation starts. The simulation cycle is then continuously repeated during which processes are executed and signals are updated. A VHDL simulation cycle consists of the following steps:

1. The current time, t_c is set equal to t_n .
2. Each active signal in the model is updated and events may occur as a result.
3. For each process P, if P is currently sensitive to a signal S, and an event has occurred on signal S in this simulation cycle, then process P resumes.
4. Each resumed process is executed until it suspends.
5. The time of the next simulation cycle, t_n , is set to the earliest of:
 - a. the next time at which a driver becomes active or
 - b. the next time at which a process resumes
6. If $t_n = t_c$, then the next simulation cycle is a **delta cycle**.

Simulation is complete when we run out of time ($t_n = \text{TIME 'HIGH}$) and there are no active drivers or process resummptions at t_n (there are some slight modifications to these rules involving postponed processes—which we rarely use in ASIC design).

Time in an event-driven simulator has two dimensions. A **delta cycle** takes **delta time**, which does not result in a change in real time. Each event that occurs at the same **time step** executes in delta time. Only when all events have been completed and signals updated does real time advance to the next time step.

13.4.2 Delay

In VHDL you may assign a **delay mechanism** to an assignment statement. **Transport delay** is characteristic of wires and transmission lines that exhibit nearly infinite frequency response and will transmit any pulse, no matter how short. **Inertial delay** more closely models the real behavior of logic cells. Typically, a logic cell will not transmit a pulse that is shorter than the switching time of the circuit, and this is the default **pulse-rejection limit**. If we explicitly specify a pulse-rejection limit, the assignment will not transmit a pulse shorter than the limit. As an example, the following three assignments are equivalent to each other:

```
Op <= Ip after 10 ns;                                --1
Op <= inertial Ip after 10 ns;                       --2
Op <= reject 10 ns inertial Ip after 10 ns;         --3
```

Every assignment that uses transport delay can be written using inertial delay with a pulse-rejection limit, as the following examples illustrate.

```
-- Assignments using transport delay:           --1
Op <= transport Ip after 10 ns;                 --2
Op <= transport Ip after 10 ns, not Ip after 20 ns; --3
-- Their equivalent assignments:               --4
Op <= reject 0 ns inertial Ip after 10 ns;     --5
Op <= reject 0 ns inertial Ip after 10 ns, not Ip after 10 ns; --6
```

13.5 Cell Models

There are several different kinds of logic cell models:

- Primitive models, which are produced by the ASIC library company and describe the function and properties of each logic cell (NAND, D flip-flop, and so on) using primitive functions.
- Verilog and VHDL models that are produced by an ASIC library company from the primitive models.
- Proprietary models produced by library companies that describe either small logic cells or larger functions such as microprocessors.

A logic cell model is different from the cell **delay model**, which is used to calculate the delay of the logic cell, from the **power model**, which is used to calculate power dissipation of the logic cell, and from the interconnect **timing model**, which is used to calculate the delays between logic cells (we return to these in Section 13.6).

13.5.1 Primitive Models

The following is an example of a **primitive model** from an ASIC library company (Compass Design Automation). This particular model (for a two-input NAND cell) is complex because it is intended for a 0.35 μm process and has some advanced delay modeling features. The contents are not important to an ASIC designer, but almost all of the information about a logic cell is derived from the primitive model. The designer does not normally see this primitive model; it may only be used by an ASIC library company to generate other models—Verilog or VHDL, for example.

```
Function
(timingModel = oneOf("ism","pr"); powerModel = oneOf("pin"); )
Rec
Logic = Function (A1; A2; )Rec ZN = not (A1 AND A2); End; End;
miscInfo = Rec Title = "2-Input NAND, 1X Drive"; freq_fact = 0.5;
tml = "nd02d1 nand 2 * zn a1 a2";
MaxParallel = 1; Transistors = 4; power = 0.179018;
Width = 4.2; Height = 12.6; productName = "stdcell135"; libraryName =
"cb35sc"; End;
```

```

Pin = Rec
A1 = Rec input; cap = 0.010; doc = "Data Input"; End;
A2 = Rec input; cap = 0.010; doc = "Data Input"; End;
ZN = Rec output; cap = 0.009; doc = "Data Output"; End; End;
Symbol = Select
timingModel
On pr Do Rec
tA1D_fr = |( Rec prop = 0.078; ramp = 2.749; End);
tA1D_rf = |( Rec prop = 0.047; ramp = 2.506; End);
tA2D_fr = |( Rec prop = 0.063; ramp = 2.750; End);
tA2D_rf = |( Rec prop = 0.052; ramp = 2.507; End); End
On ism Do Rec
tA1D_fr = |( Rec A0 = 0.0015; dA = 0.0789; D0 = -0.2828;
dD = 4.6642; B = 0.6879; Z = 0.5630; End );
tA1D_rf = |( Rec A0 = 0.0185; dA = 0.0477; D0 = -0.1380;
dD = 4.0678; B = 0.5329; Z = 0.3785; End );
tA2D_fr = |( Rec A0 = 0.0079; dA = 0.0462; D0 = -0.2819;
dD = 4.6646; B = 0.6856; Z = 0.5282; End );
tA2D_rf = |( Rec A0 = 0.0060; dA = 0.0464; D0 = -0.1408;
dD = 4.0731; B = 0.6152; Z = 0.4064; End ); End; End;
Delay = |( Rec from = pin.A1; to = pin.ZN;
edges = Rec fr = Symbol.tA1D_fr; rf = Symbol.tA1D_rf; End; End, Rec
from = pin.A2; to = pin.ZN; edges = Rec fr = Symbol.tA2D_fr; rf =
Symbol.tA2D_rf; End; End );
MaxRampTime = |( Rec check = pin.A1; riseTime = 3.000; fallTime =
3.000; End, Rec check = pin.A2; riseTime = 3.000; fallTime = 3.000;
End, Rec check = pin.ZN; riseTime = 3.000; fallTime = 3.000; End );
DynamicPower = |( Rec rise = { ZN }; val = 0.003; End); End; End

```

This primitive model contains the following information:

- The logic cell name, the logic cell function expressed using primitive functions, and port names.
- A list of supported delay models (*ism* stands for input-slope delay model, and *pr* for prop-ramp delay model—see Section 13.6).
- Miscellaneous data on the logic cell size, the number of transistors and so on—primarily for use by logic-synthesis tools and for data book generation.
- Information for power dissipation models and timing analysis.

13.5.2 Synopsys Models

The ASIC library company may provide **vendor models** in formats unique to each CAD tool company. The following is an example of a Synopsys model derived from a primitive model similar to the example in Section 13.5.1. In a Synopsys library,

each logic cell is part of a large file that also contains wire-load models and other characterization information for the cell library.

```
cell (nd02d1) {
/* title : 2-Input NAND, 1X Drive */
/* pmd checksum : 'HBA7EB26C */
area : 1;
  pin(a1) { direction : input; capacitance : 0.088;
    fanout_load : 0.088; }
  pin(a2) { direction : input; capacitance : 0.087;
    fanout_load : 0.087; }
  pin(zn) { direction : output; max_fanout : 1.786;
    max_transition : 3; function : "(a1 a2)";
  timing() {
    timing_sense : "negative_unate"
    intrinsic_rise : 0.24 intrinsic_fall : 0.17
    rise_resistance : 1.68 fall_resistance : 1.13
    related_pin : "a1" }
  timing() { timing_sense : "negative_unate"
    intrinsic_rise : 0.32 intrinsic_fall : 0.18
    rise_resistance : 1.68 fall_resistance : 1.13
    related_pin : "a2"
} } } /* end of cell */
```

This file contains the only information the Synopsys logic synthesizer, simulator, and other design tools use. If the information is not in this model, the tools cannot produce it. You can see that not all of the information from a primitive model is necessarily present in a vendor model.

13.5.3 Verilog Models

The following is a Verilog model for an inverter (derived from a primitive model):

```
`celldefine //1
`delay_mode_path //2
`suppress_faults //3
`enable_portfaults //4
`timescale 1 ns / 1 ps //5
module in01d1 (zn, i); input i; output zn; not G2(zn, i); //6
specify specparam //7
InCap$i = 0.060, OutCap$zn = 0.038, MaxLoad$zn = 1.538, //8
R_Ramp$i$zn = 0.542:0.980:1.750, F_Ramp$i$zn = 0.605:1.092:1.950; //9
specparam cell_count = 1.000000; specparam Transistors = 4 ; //10
specparam Power = 1.400000; specparam MaxLoadedRamp = 3 ; //11
(i => zn) = (0.031:0.056:0.100, 0.028:0.050:0.090); //12
endspecify //13
endmodule //14
`nosuppress_faults //15
```

```

`disable_portfaults //16
`endcelldefine //17

```

This is very similar in form to the model for the MUX of Section 13.2.1, except that this model includes additional timing parameters (at the beginning of the `specify` block). These timing parameters were omitted to simplify the model of Section 13.2.1 (see Section 13.6 for an explanation of their function).

There are no standards on writing Verilog logic cell models. In the Verilog model, `in01d1`, fixed delays (corresponding to zero load capacitance) are embedded in a `specify` block. The parameters describing the delay equations for the timing model and other logic cell parameters (area, power-model parameters, and so on) are specified using the Verilog `specparam` feature. Writing the model in this way allows the model information to be accessed using the Verilog PLI routines. It also allows us to back-annotate timing information by overriding the data in the `specify` block.

The following Verilog code tests the model for logic cell `in01d1`:

```

`timescale 1 ns / 1 ps //1
module SDF_b; reg A; in01d1 i1 (B, A); //2
initial begin A = 0; #5; A = 1; #5; A = 0; end //3
initial $monitor("T=%6g", $realtime, " A=", A, " B=", B); //4
endmodule //5

T=      0 A=0 B=x
T= 0.056 A=0 B=1
T=      5 A=1 B=1
T=  5.05 A=1 B=0
T=     10 A=0 B=0
T=10.056 A=0 B=1

```

In this case the simulator has used the fixed, typical timing delays (0.056 ns for the rising delay, and 0.05 ns for the falling delay—both from line 12 in module `in01d1`). Here is an example SDF file (filename `SDF_b.sdf`) containing back-annotation timing delays:

```

(DELAYFILE
  (SDFVERSION "3.0") (DESIGN "SDF.v") (DATE "Aug-13-96")
  (VENDOR "MJSS") (PROGRAM "MJSS") (VERSION "v0")
  (DIVIDER .) (TIMESCALE 1 ns)
  (CELL (CELLTYPE "in01d1")
    (INSTANCE SDF_b.i1)
    (DELAY (ABSOLUTE
      (IOPATH i zn (1.151:1.151:1.151) (1.363:1.363:1.363))
    ))
  )
)

```

(Notice that since Verilog is case sensitive, the instance names and node names in the SDF file are also case sensitive.) This SDF file describes the path delay between input (pin `i`) and output (pin `zn`) as 1.151 ns (rising delay—minimum, typical, and

maximum are identical in this simple example) and 1.363 ns (falling delay). These delays are calculated by a **delay calculator**. The delay calculator may be a stand-alone tool or part of the simulator. This tool calculates the delay values by using the delay parameters in the logic cell model (lines 8–9 in module `in01d1`).

We call a system task, `$sdf_annotate`, to perform back-annotation,

```

`timescale 1 ns / 1 ps //1
module SDF_b; reg A; in01d1 i1 (B, A); //2
initial begin //3
$sdf_annotate ( "SDF_b.sdf", SDF_b, , "sdf_b.log", "minimum", , ); //4
A = 0; #5; A = 1; #5; A = 0; end //5
initial $monitor("T=%6g", $realtime, " A=", A, " B=", B); //6
endmodule //7

```

Here is the output (from MTI V-System/Plus) including back-annotated timing:

```

T=      0 A=0 B=x
T=  1.151 A=0 B=1
T=      5 A=1 B=1
T=  6.363 A=1 B=0
T=     10 A=0 B=0
T=11.151 A=0 B=1

```

The delay information from the SDF file has been passed to the simulator.

Back-annotation is not part of the IEEE 1364 Verilog standard, although many Verilog-compatible simulators do support the `$sdf_annotate` system task. Many ASIC vendors require the use of Verilog to complete a back-annotated timing simulation before they will accept a design for manufacture. Used in this way Verilog is referred to as a **golden simulator**, since an ASIC vendor uses Verilog to judge whether an ASIC design fabricated using its process will work.

13.5.4 VHDL Models

Initially VHDL did not offer a standard way to perform back-annotation. Here is an example of a VHDL model for an inverter used to perform a back-annotated timing simulation using an Altera programmable ASIC:

```

library IEEE; use IEEE.STD_LOGIC_1164.all;
library COMPASS_LIB; use COMPASS_LIB.COMPASS_ETC.all;
entity bknot is
  generic (derating : REAL := 1.0; z1_cap : REAL := 0.000;
    INSTANCE_NAME : STRING := "bknot");
  port (Z2 : in Std_Logic; Z1 : out STD_LOGIC);
end bknot;
architecture bknot of bknot is
  constant tplh_z2_z1 : TIME := (1.00 ns + (0.01 ns * Z1_Cap)) * derating;
  constant tphl_z2_z1 : TIME := (1.00 ns + (0.01 ns * Z1_Cap)) * derating;
begin
  process (Z2)

```

```

variable int_Z1 : Std_Logic := 'U';
variable tplh_Z1, tphl_Z1, Z1_delay : time := 0 ns;
variable CHANGED : BOOLEAN;
begin
int_Z1 := not (Z2);
if Z2'EVENT then
    tplh_Z1 := tplh_Z2_Z1; tphl_Z1 := tphl_Z2_Z1;
end if;
Z1_delay := F_Delay(int_Z1, tplh_Z1, tphl_Z1);
Z1 <= int_Z1 after Z1_delay;
end process;
end bknot;
configuration bknot_CON of bknot is for bknot end for;
end bknot_CON;

```

This model accepts two generic parameters: load capacitance, `Z1_cap`, and a derating factor, `derating`, used to adjust postlayout timing delays. The proliferation of different VHDL back-annotation techniques drove the VHDL community to develop a standard method to complete back-annotation—VITAL.

13.5.5 VITAL Models

VITAL is the VHDL Initiative Toward ASIC Libraries, IEEE Std 1076.4 [1995].² VITAL allows the use of sign-off quality ASIC libraries with VHDL simulators. **Sign-off** is the transfer of a design from a customer to an ASIC vendor. If the customer has completed simulation of a design using **sign-off quality** models from an approved cell library and a golden simulator, the customer and ASIC vendor will sign off the design (by signing a contract) and the vendor guarantees that the silicon will match the simulation.

VITAL models, like Verilog models, may be generated from primitive models. Here is an example of a VITAL-compliant model for an inverter,

```

library IEEE; use IEEE.STD_LOGIC_1164.all;           --1
use IEEE.VITAL_timing.all; use IEEE.VITAL_primitives.all; --2
entity IN01D1 is                                     --3
    generic (                                        --4
        tpd_I          : VitalDelayType01 := (0 ns, 0 ns); --5
        tpd_I_ZN      : VitalDelayType01 := (0 ns, 0 ns) ); --6
    port (                                             --7
        I              : in STD_LOGIC := 'U';         --8
        ZN            : out STD_LOGIC := 'U' );      --9
attribute VITAL_LEVEL0 of IN01D1 : entity is TRUE;  --10
end IN01D1;                                         --11
architecture IN01D1 of IN01D1 is                   --12
attribute VITAL_LEVEL1 of IN01D1 : architecture is TRUE; --13
signal I_ipd      : STD_LOGIC := 'X';             --14
begin                                                    --15

```

²IEEE Std 1076.4-1995, © 1995 IEEE. All rights reserved.

```

WIREDELAY:block                                     --16
  begin VitalWireDelay(I_ipd, I, tpd_I); end block;  --17
VITALbehavior : process (I_ipd)                    --18
variable ZN_zd      : STD_LOGIC;                   --19
variable ZN_GlitchData : VitalGlitchDataType;     --20
begin                                                       --21
ZN_zd := VitalINV(I_ipd);                               --22
VitalPathDelay01(                                       --23
  OutSignal      => ZN,                                  --24
  OutSignalName  => "ZN",                                --25
  OutTemp        => ZN_zd,                               --26
  Paths          => (0 => (I_ipd'LAST_EVENT, tpd_I_ZN, TRUE)), --27
  GlitchData     => ZN_GlitchData,                      --28
  DefaultDelay   => VitalZeroDelay01,                   --29
  Mode           => OnEvent,                             --30
  MsgOn          => FALSE,                               --31
  XOn            => TRUE,                                --32
  MsgSeverity    => ERROR);                             --33
  end process;                                         --34
end IN01D1;                                           --35

```

The following testbench, SDF_testbench, contains an entity, SDF, that in turn instantiates a copy of an inverter, in01d1:

```

library IEEE; use IEEE.STD_LOGIC_1164.all;          --1
entity SDF is port ( A : in STD_LOGIC; B : out STD_LOGIC ); --2
end SDF;                                             --3
architecture SDF of SDF is                           --4
component in01d1 port ( I : in STD_LOGIC; ZN : out STD_LOGIC ); --5
end component;                                       --6
  begin il: in01d1 port map ( I => A, ZN => B);       --7
end SDF;                                             --8

library STD; use STD.TEXTIO.all;                    --1
library IEEE; use IEEE.STD_LOGIC_1164.all;         --2
entity SDF_testbench is end SDF_testbench;         --3
architecture SDF_testbench of SDF_testbench is     --4
component SDF port ( A : in STD_LOGIC; B : out STD_LOGIC ); --5
end component;                                       --6
signal A, B : STD_LOGIC := '0';                    --7
begin                                               --8
  SDF_b : SDF port map ( A => A, B => B);           --9
  process begin                                     --10
    A <= '0'; wait for 5 ns; A <= '1';            --11
    wait for 5 ns; A <= '0'; wait;                 --12
  end process;                                       --13
  process (A, B) variable L: LINE; begin           --14
    write(L, now, right, 10, TIME'(ps));          --15
    write(L, STRING(" A=")); write(L, TO_BIT(A)); --16
    write(L, STRING(" B=")); write(L, TO_BIT(B)); --17
  end process;

```

```

        writeline(output, L);                                --18
    end process;                                           --19
end SDF_testbench;                                       --20

```

Here is an SDF file (`SDF_b.sdf`) that contains back-annotation timing information (min/typ/max timing values are identical in this example):

```

(DELAYFILE
  (SDFVERSION "3.0") (DESIGN "SDF.vhd") (DATE "Aug-13-96")
  (VENDOR "MJSS") (PROGRAM "MJSS") (VERSION "v0")
  (DIVIDER .) (TIMESCALE 1 ns)
  (CELL (CELLTYPE "in01d1")
    (INSTANCE i1)
    (DELAY (ABSOLUTE
      (IOPATH i zn (1.151:1.151:1.151) (1.363:1.363:1.363))
      (PORT i (0.021:0.021:0.021) (0.025:0.025:0.025))
    ))
  )
)

```

(VHDL is case insensitive, but to allow the use of an SDF file with both Verilog and VHDL we must maintain case.) As in the Verilog example in Section 13.5.3 the logic cell delay (from the input pin of the inverter, `i`, to the output pin, `zn`) follows the `IOPATH` keyword. In this example there is also an interconnect delay that follows the `PORT` keyword. The interconnect delay has been placed, or lumped, at the input of the inverter. In order to include back-annotation timing using the SDF file, `SDF_b.sdf`, we use a command-line switch to the simulator. In the case of MTI V-System/Plus the command is as follows:

```

<msmith/MTI/vital> vsim -c -sdfmax /sdf_b=SDF_b.sdf sdf_testbench
...
#      0 ps A=0 B=0
#      0 ps A=0 B=0
#    1176 ps A=0 B=1
#     5000 ps A=1 B=1
#     6384 ps A=1 B=0
#    10000 ps A=0 B=0
#    11176 ps A=0 B=1

```

We have to explain to the simulator where in the design hierarchy to apply the timing information in the SDF file. The situation is like giving someone directions “Go North on the M1 and turn left at the third intersection,” but where do we start? London or Birmingham? VHDL needs much more precise directions. Using VITAL we say we back-annotate to a **region**. The switch `/sdf_b=SDF_b.sdf` specifies that all instance names in the SDF file, `SDF_b.sdf`, are relative to the region `/sdf_b`. The region refers to instance name `sdf_b` (line 9 in entity `SDF_testbench`), which is an instance of component `SDF`. Component `SDF` in turn contains an instance

of a component, `in01d1`, with instance name `i1` (line 7 in architecture `SDF`). Through this rather (for us) difficult-to-follow set of directions, the simulator knows that

```
... (CELL (CELLTYPE "in01d1") (INSTANCE i1) ...
```

refers to (SDF) cell or (VHDL) component `in01d1` with instance name `i1` in instance `SDF_b` of the compiled model `sdf_testbench`.

Notice that we cannot use an SDF file of the following form (as we did for the Verilog version of this example):

```
... (CELL (CELLTYPE "in01d1") (INSTANCE SDF_b.i1) ...
```

There is no instance in the VHDL model “higher” than instance name `SDF_b` that we can use as a starting point for VITAL back-annotation. In the Verilog SDF file we can refer to the name of the top-level module (`SDF_b` in line 2 in module `SDF_b`). We cannot do this in VHDL; we must name an instance. The result is that, unless you are careful in constructing the hierarchy of your VHDL design, you may not be able to use the same SDF file for back-annotating both VHDL and Verilog.

13.5.6 SDF in Simulation

SDF was developed to handle back-annotation, but it is also used to describe forward-annotation of timing constraints from logic synthesis. Here is an example of an SDF file that contains the timing information for the halfgate ASIC design:

```
(DELAYFILE
  (SDFVERSION "1.0")
  (DESIGN "halfgate_ASIC_u")
  (DATE "Aug-13-96")
  (VENDOR "Compass")
  (PROGRAM "HDL Asst")
  (VERSION "v9r1.2")
  (DIVIDER .)
  (TIMESCALE 1 ns)
  (CELL (CELLTYPE "in01d0")
    (INSTANCE v_1.B1_i1)
    (DELAY (ABSOLUTE
      (IOPATH I ZN (1.151:1.151:1.151) (1.363:1.363:1.363))
    ))
  )
  (CELL (CELLTYPE "pc5o06")
    (INSTANCE u1_2)
    (DELAY (ABSOLUTE
      (IOPATH I PAD (1.216:1.216:1.216) (1.249:1.249:1.249))
    ))
  )
  (CELL (CELLTYPE "pc5d01r")
    (INSTANCE u0_2)
```

```

        (DELAY (ABSOLUTE
            (IOPATH PAD CIN (.169:.169:.169) (.199:.199:.199))
        ))
    )
)

```

This SDF file describes the delay due to the input pad (cell pc5d01r, instance name u0_2), our inverter (cell in01d0, instance name v_1.B1_i1), and the output pad (cell pc5o06, instance name u1_2). Since this SDF file was produced before any physical layout, there are no estimates for interconnect delay. The following partial SDF file illustrates how interconnect delay can be specified in SDF.

```

(DELAYFILE
    ...
    (PROCESS "FAST-FAST")
    (TEMPERATURE 0:55:100)
    (TIMESCALE 100ps)
(CELL (CELLTYPE "CHIP")
    (INSTANCE TOP)
    (DELAY (ABSOLUTE
        (INTERCONNECT A.INV8.OUT B.DFF1.Q (:0.6:) (:0.6:))
    )))
)

```

This SDF file specifies an interconnect delay (using the keyword `INTERCONNECT`) of 60 ps (0.6 units with a timescale of 100 ps per unit) between the output port of an inverter with instance name `A.INV8` (note that `'.'` is the hierarchy divider) in block A and the Q input port of a D flip-flop (instance name `B.DFF1`) in block B.

The triplet notation (`min : typ : max`) in SDF corresponds to minimum, typical, and maximum values of a parameter. Specifying two triplets corresponds to rising (the first triplet) and falling delays. A single triplet corresponds to both. A third triplet corresponds to turn-off delay (transitions to or from `'Z'`). You can also specify six triplets (rising, falling, `'0'` to `'Z'`, `'Z'` to `'1'`, `'1'` to `'Z'`, and `'Z'` to `'0'`). When only the typical value is specified, the minimum and maximum are set equal to the typical value.

Logic cell delays can use several models in SDF. Here is one example:

```

(INSTANCE B.DFF1)
(DELAY (ABSOLUTE
    (IOPATH (POSEDGE CLK) Q (12:14:15) (11:13:15))))
)

```

The `IOPATH` construct specifies a delay between the input pin and the output pin of a cell. In this example the delay is between the positive edge of the clock (input port) and the flip-flop output.

The following example SDF file is for an AO221 logic cell:

```

(DELAYFILE
(DSIGN "MYDESIGN")
(DATE "26 AUG 1996")
(VENDOR "ASICS_INC")
)

```

```

(PROGRAM "SDF_GEN")
(VERSION "3.0")
(DIVIDER .)
(VOLTAGE 3.6:3.3:3.0)
(PROCESS "-3.0:0.0:3.0")
(TEMPERATURE 0.0:25.0:115.0)
(TIMESCALE )
(CELL
  (CELLTYPE "AOI221")
  (INSTANCE X0)
  (DELAY (ABSOLUTE
    (IOPATH A1 Y (1.11:1.42:2.47) (1.39:1.78:3.19))
    (IOPATH A2 Y (0.97:1.30:2.34) (1.53:1.94:3.50))
    (IOPATH B1 Y (1.26:1.59:2.72) (1.52:2.01:3.79))
    (IOPATH B2 Y (1.10:1.45:2.56) (1.66:2.18:4.10))
    (IOPATH C1 Y (0.79:1.04:1.91) (1.36:1.62:2.61))
  )))

```

13.6 Delay Models

We shall use the term *timing model* to describe delays outside logic cells and the term *delay model* to describe delays inside logic cells. These terms are not standard and often people use them interchangeably. There are also different terms for various types of delay:

- A **pin-to-pin delay** is a delay between an input pin and an output pin of a logic cell. This usually represents the delay of the logic cell excluding any delay contributed by interconnect.
- A **pin delay** is a delay lumped to a certain pin of a logic cell (usually an input). This usually represents the delay of the interconnect, but may also represent the delay of the logic cell.
- A **net delay** or **wire delay** is a delay outside a logic cell. This always represents the delay of interconnect.

In this section we shall focus on delay models and logic cell delays. In Chapter 3 we modeled logic cell delay as follows (Eq. 3.10):

$$t_{PD} = R(C_{out} + C_p) + t_q \quad (13.5)$$

A linear delay model is also known as a **prop-ramp delay model**, because the delay comprises a fixed propagation delay (the intrinsic delay) and a ramp delay (the extrinsic delay). As an example, the data book entry for the inverter, cell `in01d0`, in a $0.8\ \mu\text{m}$ standard-cell library gives the following delay information (with delay measured in nanoseconds and capacitance in picofarads):

$$\text{RISE} = 0.10 + 0.07 + (1.75 \times \text{Cld}) \quad \text{FALL} = 0.09 + 0.07 + (1.95 \times \text{Cld}) \quad (13.6)$$

The first two terms in each of these equations represents the intrinsic delay, with the last term in each equation representing the extrinsic delay. We see that the `Cld` corresponds to C_{out} , $R_{pu} = 1.75\ \text{k}\Omega$, and $R_{pd} = 1.95\ \text{k}\Omega$ (R_{pu} is the pull-up resistance and R_{pd} is the pull-down resistance).

From the data book the pin capacitances for this logic cell are as follows:

$$\text{pin I (input)} = 0.060\ \text{pF} \quad \text{pin ZN (output)} = 0.038\ \text{pF} \quad (13.7)$$

Thus, $C_p = 0.038\ \text{pF}$ and we can calculate the component of the intrinsic delay due to the output pin capacitance as follows:

$$C_p \times R_{pu} = 0.038 \times 1.75 = 0.0665\ \text{ns} \quad \text{and} \quad C_p \times R_{pd} = 0.038 \times 1.95 = 0.0741\ \text{ns} \quad (13.8)$$

Suppose t_{qr} and t_{qf} are the parasitic delays for the rising and falling waveforms respectively. By comparing the data book equations for the rise and fall delays with Eq. 13.5 and 13.8, we can identify $t_{qr} = 0.10\ \text{ns}$ and $t_{qf} = 0.09\ \text{ns}$.

Now we can explain the timing section of the `in01d0` model (Section 13.5.3),

```

specify specparam                                     //1
InCap$i = 0.060, OutCap$zn = 0.038, MaxLoad$zn = 1.538, //2
R_Ramp$i$zn = 0.542:0.980:1.750, F_Ramp$i$zn = 0.605:1.092:1.950; //3
specparam cell_count = 1.000000; specparam Transistors = 4 ; //4
specparam Power = 1.400000; specparam MaxLoadedRamp = 3 ; //5
(i=>zn)=(0.031:0.056:0.100, 0.028:0.050:0.090); //6

```

The parameter `OutCap$zn` is C_p . The maximum value of the parameter `R_Rampizn` is R_{pu} , and the maximum value of parameter `F_Rampizn` is R_{pd} . Finally, the maximum values of the fixed-delay triplets correspond to t_{qr} and t_{qf} .

13.6.1 Using a Library Data Book

ASIC library data books typically contain two types of information for each cell in the library—capacitance loading and delay. Table 13.7 shows the input capacitances for the inverter family for both an **area-optimized library** (small) and a **performance-optimized library** (fast).

From Table 13.7, the input capacitance of the small library version of the `inv1` (a 1X inverter gate) is $0.034\ \text{pF}$. Any logic cell that is driving an `inv1` from the small library sees this as a load capacitance. This capacitance consists of the gate capacitance of a p -channel transistor, the gate capacitance of an n -channel transistor,

and the internal cell routing. Similarly, 0.145 pF is the input capacitance of a fast inv1. We can deduce that the transistors in the fast library are approximately $0.145/0.034 \approx 4$ times larger than those in the small version. The small library and fast library may not have the same cell height (they usually do not), so that we cannot mix cells from different libraries in the same standard-cell area.

TABLE 13.7 Input capacitances for an inverter family (pF).¹

Library	inv1	invh	invs	inv8	inv12
Area	0.034	0.067	0.133	0.265	0.397
Performance	0.145	0.292	0.584	1.169	1.753

¹Suffix '1' denotes normal drive strength, suffix 'h' denotes high-power drive strength (approximately $\times 2$), suffix 's' denotes superpower drive strength (approximately $\times 4$), and a suffix 'm' ($m=8$ or 12) denotes inverter blocks containing m inverters.

The delay table for a 2:1 MUX is shown in Table 13.8. For example, D0/ to z/, indicates the path delay from the D0 input rising to the z output rising. Rising delay is denoted by '/' and falling delay by '\'.

TABLE 13.8 Delay information for a 2:1 MUX.

From input	To output	Propagation delay			
		Area		Performance	
		Extrinsic/ nspF ⁻¹	Intrinsic / ns	Extrinsic / ns	Intrinsic / ns
D0\	Z\	2.10	1.42	0.5	0.8
D0/	Z/	3.66	1.23	0.68	0.70
D1\	Z\	2.10	1.42	0.50	0.80
D1/	Z/	3.66	1.23	0.68	0.70
SD\	Z\	2.10	1.42	0.50	0.80
SD\	Z/	3.66	1.09	0.70	0.73
SD/	Z\	2.10	2.09	0.5	1.09
SD/	Z/	3.66	1.23	0.68	0.70

/ = rising and \ = falling.

Both intrinsic delay and extrinsic delay values are given in Table 13.8. For example, the delay t_{PD} (from DO\ to z\) of a 2:1 MUX from the small library is

$$t_{PD} = 1.42 \text{ ns} + (2.10 \text{ ns/pF}) \times C_L \text{ (pF)}. \quad (13.9)$$

ASIC cell libraries may be characterized and the delay information presented in several ways in a data book. Some manufacturers simulate under worst-case slow conditions (4.5 V, 100°C, and slow process conditions, for example) and then derate each delay value to convert delays to nominal conditions (5.0 V, 25°C, and nominal process). This allows nominal delays to be used in the data book while maintaining accurate predictions for worst-case behavior. Other manufacturers characterize using nominal conditions and include worst-case values in the data book. In either case, we always design with worst-case values. Data books normally include process, voltage, and temperature derating factors as tables or graphs such as those shown in Tables 13.9 and 13.10.

For example, suppose we are measuring the performance of an ASIC on the bench and the lab temperature (25 °C) and the power supply voltage (5 V) correspond to nominal operating conditions. We shall assume, in the absence of other information, that we have an ASIC from a nominal process lot. We have data book values given as worst case (worst-case temperature, 100 °C; worst-case voltage, 4.5 V; slow process) and we wish to find nominal values for delay to compare them with our measured results. From Table 13.9 the derating factor from nominal process to slow process is 1.31. From Table 13.10 the derating factor from 100 °C and 4.5 V to nominal (25 °C and 5 V) is 1.60. The derating factor from nominal to worst-case (data book values) is thus:

$$\text{worst-case} = \text{nominal} \times 1.31 \text{ (slow process)} \times 1.60 \text{ (4.5 V, 100 °C)}. \quad (13.10)$$

To get from the data book values to nominal operating conditions we use the following equation:

$$\text{nominal} = \text{worst-case} / (1.31 \times 1.60) = 0.477 \times \text{worst-case}. \quad (13.11)$$

13.6.2 Input-Slope Delay Model

It is increasingly important for submicron technologies to account for the effects of the rise (and fall) time of the input waveforms to a logic cell. The nonlinear delay model described in this section was developed by Mike Misheloff at VLSI Technology and then at Compass. There are, however, no standards in this area—each ASIC company has its own, often proprietary, model.

We begin with some definitions:

- D_{t0} is the time from the beginning of the input to beginning of the output.
- D_{t1} is the time from the beginning of the input to the end of the output.
- I_R is the time from the beginning to the end of the input ramp.

TABLE 13.9 Process derating factors.

Process	Derating factor
Slow	1.31
Nominal	1.0
Fast	0.75

TABLE 13.10 Temperature and voltage derating factors.

Temperature/°C	Supply voltage				
	4.5V	4.75V	5.00V	5.25V	5.50V
-40	0.77	0.73	0.68	0.64	0.61
0	1.00	0.93	0.87	0.82	0.78
25	1.14	1.07	1.00	0.94	0.90
85	1.50	1.40	1.33	1.26	1.20
100	1.60	1.49	1.41	1.34	1.28
125	1.76	1.65	1.56	1.47	1.41

In these definitions “beginning” and “end” refer to the projected intersections of the input waveform or the output waveform with V_{DD} and V_{SS} as appropriate. Then we can calculate the delay, D (measured with 0.5 trip points at input and output), and output ramp, O_R , as follows:

$$D = (D_{t1} + D_{t0} - I_R) / 2 \quad (13.12)$$

$$\text{and } O_R = D_{t1} - D_{t0}. \quad (13.13)$$

Experimentally we find that the times, D_{t0} and D_{t1} , are accurately modeled by the following equations:

$$D_{t0} = A_0 + D_0 C_L + B \times \min(I_R, C_R) + Z \times \max(0, I_R - C_R) \quad (13.14)$$

and

$$D_{t1} = A_1 + B I_R + D_1 C_L. \quad (13.15)$$

C_R is the critical ramp that separates two regions of operation, we call these slow ramp and fast ramp. A sensible definition for C_R is the point at which the end of the input ramp occurs at the same time the output reaches the 0.5 trip point. This leads to the following equation for C_R :

$$C_R = \frac{A_0 + A_1 + (D_0 + D_1) C_L}{2(1 - B)}. \quad (13.16)$$

It is convenient to define two more parameters:

$$d_A = A_1 - A_0 \quad \text{and} \quad d_D = D_1 - D_0. \quad (13.17)$$

In the region that $C_R > I_R$, we can simplify Eqs. 13.14 and by using the definitions in Eq. 13.17, as follows:

$$D = (D_{t1} + D_{t0} - I_R)/2 = A_0 + D_0 C_L + d_A/2 + d_D C_L/2 \quad (13.18)$$

$$\text{and } O_R = D_{t1} - D_{t0} = d_A + d_D C_L. \quad (13.19)$$

Now we can understand the timing parameters in the primitive model in Section 13.5.1. For example, the following parameter, τ_{A1D_fr} , models the falling input to rising output waveform delay for the logic cell (the units are a consistent set: all times are measured in nanoseconds and capacitances in picofarads):

$$A_0 = 0.0015; d_A = 0.0789; D_0 = -0.2828; d_D = 4.6642; B = 0.6879; Z = 0.5630;$$

The input-slope model predicts delay in the fast-ramp region, $D_{ISM}(50\%, FR)$, as follows (0.5 trip points):

$$\begin{aligned} D_{ISM}(50\%, FR) &= A_0 + D_0 C_L + 0.5 O_R = A_0 + D_0 C_L + d_A/2 + d_D C_L/2 \\ &= 0.0015 + 0.5 \times 0.0789 + (-0.2828 + 0.5 \times 4.6642) C_L \\ &= 0.041 + 2.05 C_L. \end{aligned} \quad (13.20)$$

We can adjust this delay to 0.35/0.65 trip points as follows:

$$\begin{aligned} D_{ISM}(65\%, FR) &= A_0 + D_0 C_L + 0.65 O_R \\ &= 0.0015 + 0.65 \times 0.0789 + (-0.2828 C_L + 0.65 \times 4.6642) C_L \\ &= 0.053 + 2.749 C_L. \end{aligned} \quad (13.21)$$

We can now compare Eq. 13.21 with the prop-ramp model. The prop-ramp parameters for this logic cell (from the primitive model in Section 13.5.1) are:

$$\tau_{A1D_fr} = |(\text{Rec prop} = 0.078; \text{ ramp} = 2.749; \text{ End});$$

These parameters predict the following prop-ramp delay (0.35/0.65 trip points):

$$D_{PR}(65\%) = 0.078 + 2.749 C_L. \quad (13.22)$$

The input-slope delay model and the prop-ramp delay model predict similar delays in the fast-ramp region, but for slower inputs the differences can become significant.

13.6.3 Limitations of Logic Simulation

Table 13.11 shows the switching characteristics of a two-input NAND gate (1X drive) from a commercial 1 μm gate-array family. The difference in propagation delay (with $FO = 0$) between the inputs A and B is

$$(0.25 - 0.17) \times 2 / (0.25 + 0.17) = 38\%.$$

This difference is taken into account only by a pin-to-pin delay model.

TABLE 13.11 Switching characteristics of a two-input NAND gate.

Symbol	Parameter	Fanout					K /nspF ⁻¹
		FO = 0 /ns	FO = 1 /ns	FO = 2 /ns	FO = 4 /ns	FO = 8 /ns	
t_{PLH}	Propagation delay, A to X	0.25	0.35	0.45	0.65	1.05	1.25
t_{PHL}	Propagation delay, B to X	0.17	0.24	0.30	0.42	0.68	0.79
t_r	Output rise time, X	1.01	1.28	1.56	2.10	3.19	3.40
t_f	Output fall time, X	0.54	0.69	0.84	1.13	1.71	1.83

FO = fanout in standard loads (one standard load = 0.08 pF). Nominal conditions: $V_{DD} = 5$ V, $T_A = 25$ °C.

Timing information for most gate-level simulators is calculated once, before simulation, using a delay calculator. This works as long as the logic cell delays and signal ramps do not change. There are some cases in which this is not true. Table 13.12 shows the switching characteristics of a half adder. In addition to pin-to-pin timing differences there is a timing difference depending on state. For example, the pin-to-pin timing from input pin A to the output pin S depends on the state of the input pin B. Depending on whether B = '0' or B = '1' the difference in propagation delay (at FO = 0) is

$$(0.93 - 0.58) \times 2 / (0.93 + 0.58) = 46 \%$$

This **state-dependent timing** is not taken into account by simple pin-to-pin delay models and is not accounted for by most gate-level simulators.

13.7 Static Timing Analysis

We return to the comparator/MUX example to see how timing analysis is applied to sequential logic. We shall use the same input code (`comp_mux.v` in Section 13.2), but this time we shall target the design to an Actel FPGA.

Before routing we obtain the following static timing analysis:

```
Instance name      in pin-->out pin   tr      total   incr    cell
-----
END_OF_PATH
outp_2_           R           27.26
OUT1              : D--->PAD        R           27.26   7.55   OUTBUF
I_1_CM8          : S11--->Y        R           19.71   4.40   CM8
I_2_CM8          : S11--->Y        R           15.31   5.20   CM8
I_3_CM8          : S11--->Y        R           10.11   4.80   CM8
```

TABLE 13.12 Switching characteristics of a half adder.

Symbol	Parameter	Fanout					K /nspF ⁻¹
		FO = 0 /ns	FO = 1 /ns	FO = 2 /ns	FO = 4 /ns	FO = 8 /ns	
t_{PLH}	Delay, A to S (B = '0')	0.58	0.68	0.78	0.98	1.38	1.25
t_{PHL}	Delay, A to S (B = '1')	0.93	0.97	1.00	1.08	1.24	0.48
t_{PLH}	Delay, B to S (B = '0')	0.89	0.99	1.09	1.29	1.69	1.25
t_{PHL}	Delay, B to S (B = '1')	1.00	1.04	1.08	1.15	1.31	0.48
t_{PLH}	Delay, A to CO	0.43	0.53	0.63	0.83	1.23	1.25
t_{PHL}	Delay, A to CO	0.59	0.63	0.67	0.75	0.90	0.48
t_r	Output rise time, X	1.01	1.28	1.56	2.10	3.19	3.40
t_f	Output fall time, X	0.54	0.69	0.84	1.13	1.71	1.83

FO = fanout in standard loads (one standard load = 0.08 pF). Nominal conditions: $V_{DD}=5\text{ V}$, $T_A=25\text{ }^\circ\text{C}$.

```

IN1          : PAD--->Y          R      5.32   5.32   INBUF
a_2_        R      0.00   0.00
BEGIN_OF_PATH

```

The estimated prelayout critical path delay is nearly 30 ns including the I/O-cell delays (ACT 3, worst-case, standard speed grade). This limits the operating frequency to 33 MHz (assuming we can get the signals to and from the chip pins with no further delays—highly unlikely). The operating frequency can be increased by pipelining the design as follows (by including three register stages: at the inputs, the outputs, and between the comparison and the select functions):

```

// comp_mux_rrr.v //1
module comp_mux_rrr(a, b, clock, outp); //2
input [2:0] a, b; output [2:0] outp; input clock; //3
reg [2:0] a_r, a_rr, b_r, b_rr, outp; reg sel_r; //4
wire sel = ( a_r <= b_r ) ? 0 : 1; //5
always @ (posedge clock) begin a_r <= a; b_r <= b; end //6
always @ (posedge clock) begin a_rr <= a_r; b_rr <= b_r; end //7
always @ (posedge clock) outp <= sel_r ? b_rr : a_rr; //8
always @ (posedge clock) sel_r <= sel; //9
endmodule //10

```

Following synthesis we optimize module `comp_mux_rrr` for maximum speed. Static timing analysis gives the following preroute critical paths:

```

-----INPAD to SETUP longest path-----
Rise delay, Worst case

```