LOW-LEVEL DESIGN ENTRY

The purpose of design entry is to describe a microelectronic system to a set of electronic-design automation (EDA) tools. Electronic systems used to be, and many still are, constructed from off-the-shelf components, such as TTL ICs. Design entry for these systems now usually consists of drawing a picture, a schematic . The schematic shows how all the components are connected together, the connectivity of an ASIC. This type of design-entry process is called schematic entry , or schematic capture . A circuit schematic describes an ASIC in the same way an architect's plan describes a building.

The circuit schematic is a picture, an easy format for us to understand and use, but computers need to work with an ASCII or binary version of the schematic that we call a netlist . The output of a schematic-entry tool is thus a netlist file that contains a description of all the components in a design and their interconnections.

Not all the design information may be conveyed in a circuit schematic or netlist, because not all of the functions of an ASIC are described by the connectivity information. For example, suppose we use a programmable ASIC for some random logic functions. Part of the ASIC might be designed using a text language. In this case design entry also includes writing the code. What if an ASIC in our system contains a programmable memory (PROM)? Is the PROM microcode, the '1's and '0's, part of design entry? The operation of our system is certainly dependent on the correct programming of the PROM. So perhaps the PROM code ought to be considered part of design entry. On the other hand nobody would consider the operating-system code that is loaded into a RAM on an ASIC to be a part of design entry. Obviously, then, there are several different forms of design entry. In each case it is important to make sure that you have completely specified the system-not only so that it can be correctly constructed, but so that someone else can understand how the system is put together. Design entry is thus an important part of documentation .

Until recently most ASIC design entry used schematic entry. As ASICs have become more complex, other design-entry methods are becoming common. Alternative design-entry methods can use graphical methods, such as a schematic, or text files, such as a programming language. Using a hardware description language (HDL) for design entry allows us to generate netlists directly using logic synthesis . We will concentrate on low-level design-entry methods together with their advantages and disadvantages in this chapter.

9.1 Schematic Entry

9.2 Low-Level Design Languages

9.3 PLA Tools

9.4 EDIF

9.1 Schematic Entry

Schematic entry is the most common method of design entry for ASICs and is likely to be useful in one form or another for some time. HDLs are replacing conventional gate-level schematic entry, but new graphical tools based on schematic entry are now being used to create large amounts of HDL code.

Circuit schematics are drawn on schematic sheets . Standard schematic sheet sizes (Table 9.1) are ANSI A-E (more common in the United States) and ISO A4-A0 (more common in Europe). Usually a frame or border is drawn around the schematic containing boxes that list the name and number of the schematic page, the

designer, the date of the drawing, and a list of any modifications or changes.

TABLE 9.1 ANSI (American National Standards Institute) and ISO (International Standards Organization) schematic sheet sizes.

ANSI sheet	Size (inches)	ISO sheet	Size (cm)
A	8.5 ¥ 11	A5	21.0 ¥ 14.8
B	11 ¥ 17	A4	29.7 ¥ 21.0
C	17 ¥ 22	A3	42.0 ¥ 29.7
D	22 ¥ 34	A2	59.4 ¥ 42.0
E	34 ¥ 44	A1	84.0 ¥ 59.4
		A0	118.9 ¥ 84.0

Figure 9.1 shows the " spades" and " shovels," the recognized symbols for AND, NAND, OR, and NOR gates. One of the problems with these recommendations is that the corner points of the shapes do not always lie on a grid point (using a reasonable grid size).

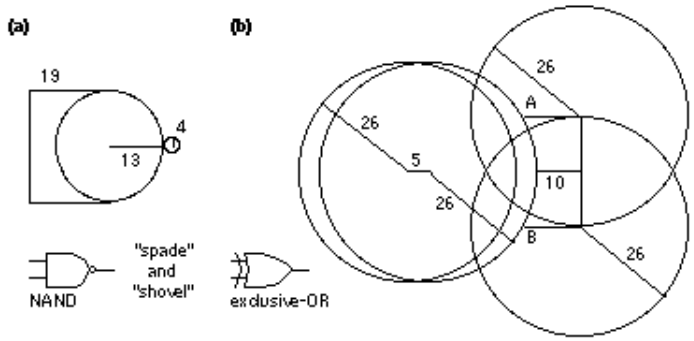

FIGURE 9.1 IEEE-recommended dimensions and their construction for logic-gate symbols. (a) NAND gate (b) exclusive-OR gate (an OR gate is a subset).

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.

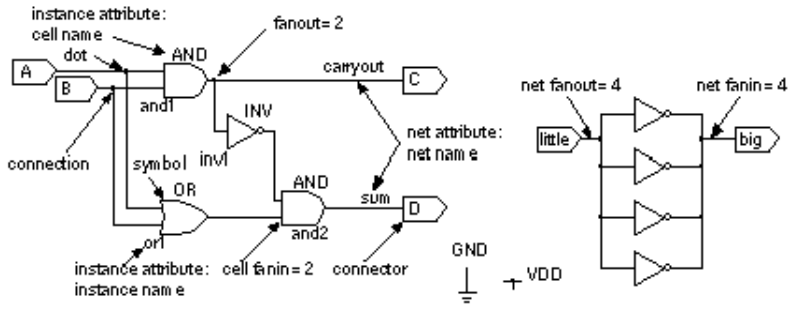

FIGURE 9.2 Terms used in circuit schematics.

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.

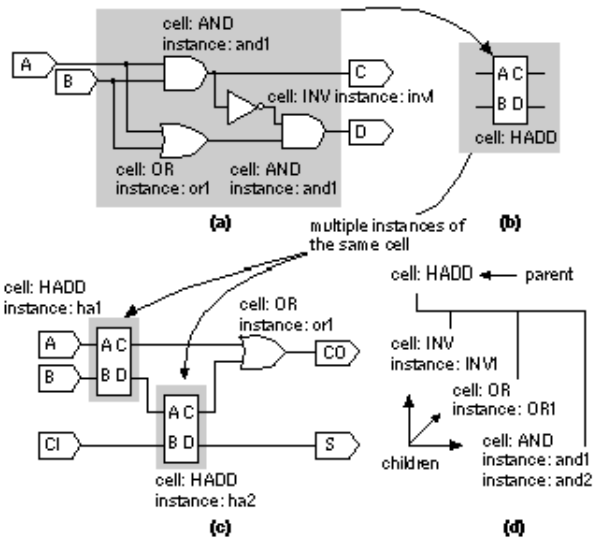

FIGURE 9.3 Schematic example showing hierarchical design. (a) The schematic of a half-adder, the subschematic of cell HADD. (b) A schematic symbol for the half adder. (c) A schematic that uses the half-adder cell. (d) The hierarchy of cell HADD.

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 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 offices 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 subschematic 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 .

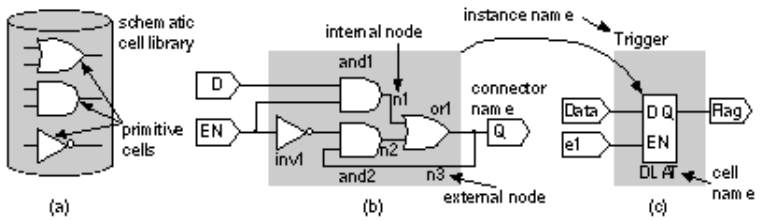

FIGURE 9.4 A cell and its subschematic. (a) A schematic library containing icons for the primitive cells. (b) A subschematic for a cell, DLAT, showing the instance names for the primitive cells. (c) A symbol for cell DLAT.

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.

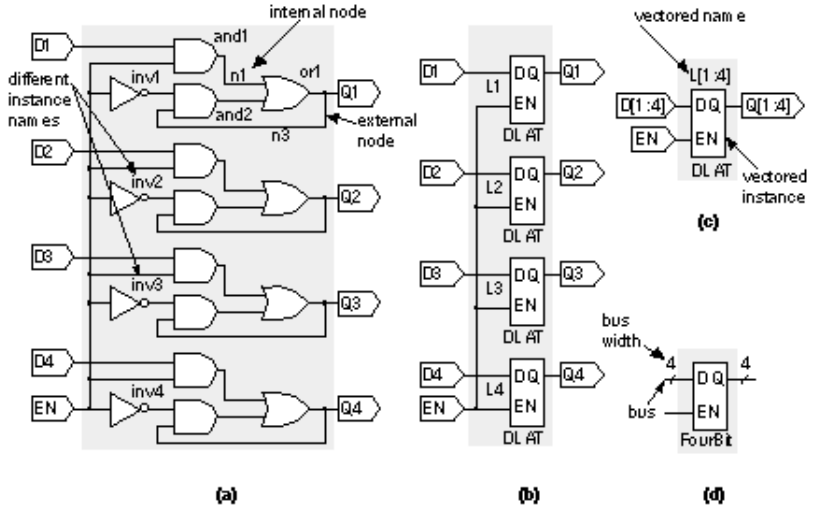

FIGURE 9.5 A 4-bit latch: (a) drawn as a flat schematic from gate-level primitives, (b) drawn as four instances of the cell symbol DLAT, (c) drawn using a vectored instance of the DLAT cell symbol with cardinality of 4, (d) drawn using a new cell symbol with cell name FourBit.

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).

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.

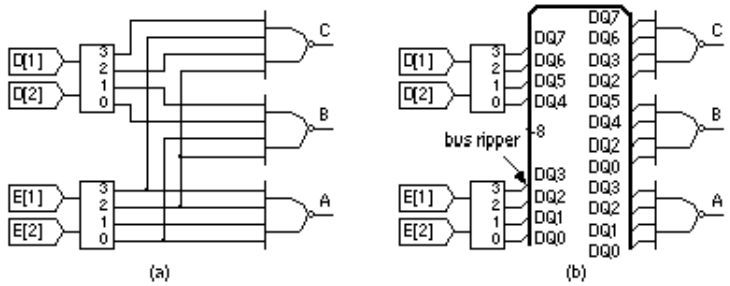

FIGURE 9.6 An example of the use of a bus to simplify a schematic. (a) An address decoder without using a bus. (b) A bus with bus rippers simplifies the schematic and reduces the possibility of making a mistake in creating and reading the schematic.

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.

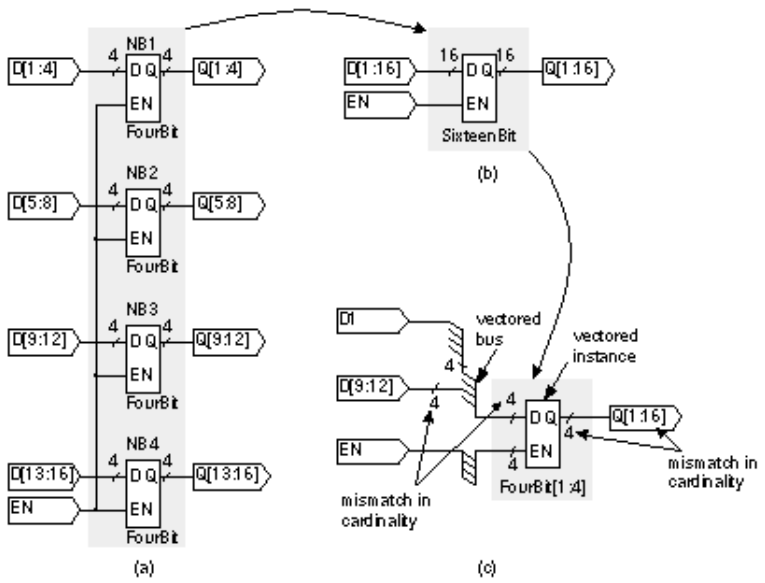

FIGURE 9.7 A 16-bit latch: (a) drawn as four instances of cell FourBit; (b) drawn as a cell named SixteenBit; (c) drawn as four multiple instances of cell FourBit.

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 difficult. 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):

TABLE 9.2 ABEL.

Statement	Example	Comment
Module	module MyModule	You can have multiple modules.
Title	title ’Title in a String’	A string is a character series between quotes.
Device	MYDEV device ’22V10’ ;	MYDEV is Device ID for documentation. 22V10 is checked by the compiler.
Comment	"comments go between double quotes" "end of line is end of comment	The end of a line signifies the end of a comment; there is no need for an end quote.
@ALTERNATE	@ALTERNATE "use alternate symbols	operator alternate default AND * & OR + # NOT / ! XOR :+: $ XNOR :*: !$
Pin declaration	MYINPUT pin 2; I3, I4 pin 3, 4 ; /MYOUTPUT pin 22; IO3,IO4 pin 21,20 ;	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	equations IO4 = HELPER ; HELPER = /I4 ;	Defines combinational logic. Two-pass logic

Assignments	MYOUTPUT = /MYINPUT ;	Equals ’=’ is unlocked assignment.
	IO3 := I4 ;	Clocked assignment operator (registered IO)
Signal sets	D = [D0, D1, D2, D3] ; Q = [Q0, Q1, Q2, Q3];	A signal set, an ABEL bus
	Q := D ;	4-bit-wide register
Suffix	MYOUTPUT.RE = CLR ;	Register reset
	MYOUTPUT.PR = PRE ;	Register preset
Addition	COUNT = [D0, D1, D2];	Can’t use @ALTERNATE
	COUNT := COUNT + 1;	if you use ’+’ to add.
Enable	ENABLE IO3 = IO2;	Three-state enable (ENABLE is a keyword).
	IO3 = MYINPUT;	IO3 must be a three-state pin.
Constants	K = [1, 0, 1] ;	K is 5.
Relational	IO# = D == K5 ;	Operators: == != < > <= >=
End	end MyModule	Last statement in module

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

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 = OxFF
	add_zero = !(Address:&);	True if Address = Ox00
	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.

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

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.

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

ABEL	CUPL
device_id device 'P2500B';	
"device_id used for JEDEC filename	
I1,I2,I3,I17,I18 pin 1,2,3,17,18;	device V2500B;
O4,O5 pin 4,5 istype 'reg_d,buffer';	pin [1,2,3,17,18] = [I1,I2,I3,I17,I18];
O6,O7 pin 6,7 istype 'com';	pin [7,6,5,4] = [O7,O6,O5,O4];

O4Q2,O7Q2 node 41,44 istype ’reg_d’; pinnode [41,65,44] = [O4Q2,O4Q1,O7Q2];

O6F2 node 43 istype ’com’; pinnode [43,68] = [O6Q2,O7Q1];

O7Q1 node 220 istype ’reg_d’;

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:

TABLE 9.7 PALASM 2.

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

name.CLKF	Register clock control
name.RSTF	Register reset control
name.SETF	Register set control

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;

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.

---

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.

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; F2 = !B&C; 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	0	1	1-- 100	-11 101
						11- 001	.e
						.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 minterm. 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); F2 = !B&C; 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 .

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 The hierarchical nature of an EDIF file.

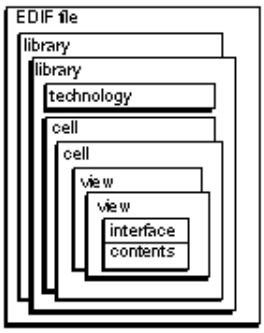

Figure 9.8 illustrates the hierarchy of the EDIF file. Within an EDIF file are one or more libraries of cell descriptions. Each library contains technology information that is used in describing the characteristics of the cells it contains. Each cell description contains one or more user-named views 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.

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 ± 1 ¥ 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.

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

```
(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")))
 (library xc4000d
  (edifLevel 0)
  (technology
   (numberDefinition
   (simulationInfo
    (logicValue  H)
    (logicValue  L)))
  (cell
   (rename INV "inv")
   (cellType GENERIC)
   (view
COMPASS_mde_view
    (viewType
NETLIST)
    (interface
     (port I
      (direction
INPUT))
     (port O
      (direction
OUTPUT))
      (designator
"@@Label")))))
 (library working
  (edifLevel 0)
  (technology
(numberDefinition )
   (simulationInfo
    (logicValue  H)
    (logicValue
L)))
  (cell
   (rename
HALFGATE_P
"halfgate_p")
   (cellType
GENERIC)
   (view
COMPASS_nls_view
    (viewType
NETLIST)
```

(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.

(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.

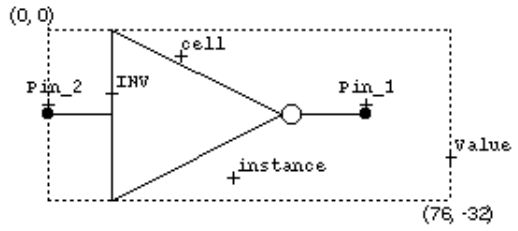

FIGURE 9.9 An EDIF view of an inverter icon. The coordinates shown are in EDIF units. The crosses

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.

TABLE 9.12 EDIF file for a Compass standard-cell schematic icon.

```
(edif pvsc370d
 (edifVersion 2 0 0)
 (edifLevel 0)
 (keywordMap
  (keywordLevel 0))
 (status
  (written
   (timeStamp 1993 2 9
22 38 36)
   (program "COMPASS"
    (version "v8"))
   (author "mikes")))
 (library pvsc370d
  (edifLevel 0)
  (technology
   (numberDefinition )
   (figureGroup
connector_FG
    (color 100 100
100)
    (textHeight 30)
    (visible
     (true )))
   (figureGroup
icon_FG
    (color 100 100
100)
    (textHeight 30)
    (visible
     (true )))
   (figureGroup
instance_FG
    (color 100 100
100)
    (textHeight 30)
    (visible
     (true )))
  (figureGroup
t_FG
   (color 100 100
0)
   (textHeight 30)
   (visible
    (true )))
  (figureGroup
s_FG
   (color 100 100
0)
   (textHeight 30)
   (visible
    (true ))
   (pathWidth 4)))
 (cell an02d1
  (cellType GENER-
)
  (view Icon_view
   (viewType SCHE-
TIC)
   (interface
    (port A2
     (direction IN-
T))
    (port A1
     (direction IN-
T))
    (port Z
     (direction
TPUT))
    (property label
     (string ""))
    (symbol
     (portImplemen-
ion
     (name A2
      (display
nector_FG
       (origin
        (pt -5 1))))
     (connectLoca-
n
      (figure
nector_FG
       (dot
        (pt 0 0)))))
   (portImplementa-
n
     (name A1
      (display
nector_FG
       (origin
        (pt -5
)))
     (connectLoca-
n
      (figure
nector_FG
       (dot
        (pt 0
))))
    (portImplementa-
n
     (name Z
      (display
nector_FG
       (origin
        (pt 60
)))
     (connectLoca-
```

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
connector_FG	pin	net_FG	wire
icon_FG	device	bus_FG	not used
instance_FG	instance		

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.

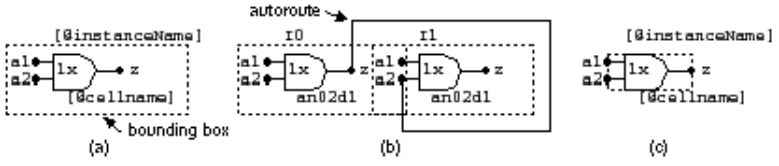

FIGURE 9.10 The bounding box problem. (a) The original bounding box for the an02d1 icon.

(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.

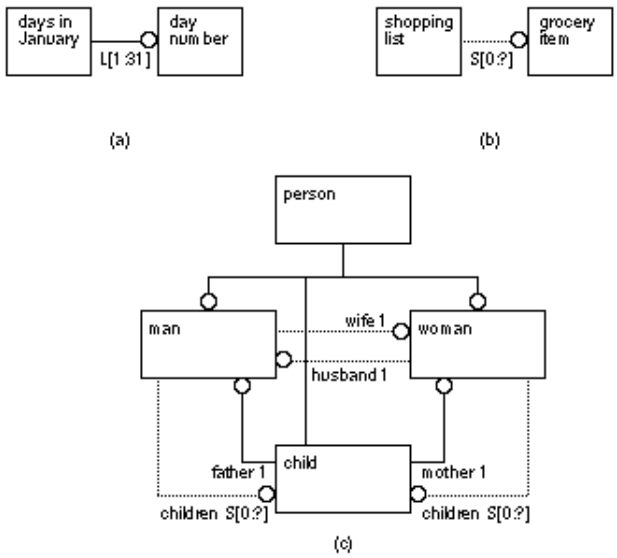

FIGURE 9.11 Examples of EXPRESS-G. (a) Each day in January has a number from 1 to 31. (b) A shopping list may contain a list of items. (c) An EXPRESS-G model for a family.

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;

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).

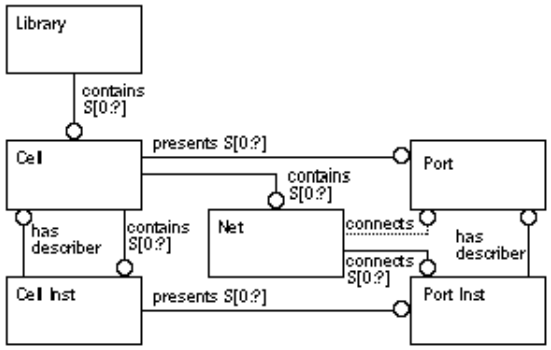

FIGURE 9.12 The original "five-box" model of electrical connectivity. There are actually six boxes or types in this figure; the Library type was added later.

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 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 .

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|use
)

(library libraryNameDef

edifLevel

technology

{<status>|cell|comment|userdata})

(technology numberDefinition

{figureGroup|fabricate|

<simulationInfos>|<physicalDesignRule>|comm
)

(cell cellNameDef

cellType

{<status>|view|<viewMap>|property|comment|
)

(view viewNameDef

viewType

interface

{<status>|<contents>|comment|property|userda
)

(interface

{port|portBundle|<symbol>|<protectionFrame

<arrayRelatedInfo>|parameter|joined|mustJoi

permutable|timing|simulate|<designator>|prop
)

(contents

{instance|offPageConnector|figure|section|

net|netBundle|page|commentGraphics|portImp

timing|simulate|when|follow|logicPort|<boundi

comment|userdata})

(viewMap

{portMap|portBackAnnotate|instanceMap|inst

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 3 0 0 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 [reference location].

EDIF is maintained by the EIA, EIA Standards Sales Office, 2001 Pennsylvania Ave., N.W., Washington, DC 20006, (202) 457-4966 [reference location]:

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 [reference location