

Chapter 2

Verilog HDL for Design and Test

In Chapter 1, we discussed the basics of test and presented ways in which hardware description languages (HDLs) could be used to improve various aspects of digital system testing. The emphasis of this chapter is on Verilog that is a popular HDL for design. The purpose is to give an introduction of the language while elaborating on ways it can be used for improving methodologies related to digital system testing. After, the basic concepts of HDL modeling, the main aspects of describing combinational and sequential circuits using different levels of abstraction, and the semantics of simulation in Verilog language are expressed. Then, we get into the testbench techniques and virtual tester development, which are heavily utilized in the presentation of test techniques in the rest of this book. Finally, a brief introduction to the procedural language interface (PLI) of Verilog and the basics of implementing test programs in PLI is given. The examples we present in this chapter for illustrating Verilog language and modeling features are used in the rest of this book as circuits that are to be tested. The HDL codes for such examples are presented here. Verilog coding techniques for gate-level components that we use for describing our netlists in the chapters that follow are also shown here.

2.1 Motivations of Using HDLs for Developing Test Methods

Generally speaking, tools and methodologies design and test engineers use are different, and there has always been a gap between design and test tools and methods. This gap results in inconsistencies in the process of design and test, such as designs that are hard to test or the time needed to convert design to the format compatible for testing. On the other hand, we have seen in new design methodologies that incorporating test in design must start from the beginning of the design process [1, 2]. It is desirable to bring testing in the hands of designers, which certainly requires that testing is applied at the level and with the language of the designers. This way, designers will be able to combine design and test phases.

Using RT-level HDLs in test and DFT, helps advancing test methods to RTL, and at the same time alleviates the need for the use of software languages and reformatting designs for the evaluation and application of test techniques. Furthermore, actual test data can be applied to post-manufacturing model of a component, while keeping other component models at the design level, and still simulating in the same environment and keeping the same testbench. This also allows reuse of design test data, and migration of testbenches from the design stage to post-manufacturing test. In a mixed-level design, these advantages make it possible to test a single component described at the gate level while leaving others in RTL or even at the system level.

On the other hand, when we try to develop test methods in an HDL environment, we are confronted with the limitations of HDL simulation tools. Such limitations include the overhead that test methods put on the simulation speed and the inability to describe complex data structures. PLI provides a library of C language functions that can directly access data within an instantiated

Verilog HDL data structure and overcomes the HDL limitations. With PLI, the advantages of doing testable hardware design in an HDL and having a software environment for the manipulation and evaluation of designs can be achieved at the same time. Therefore, not only the design core and its testbench can be developed in a uniform programming environment, but also all the facilities of software programming (such as complex data structures and utilization of functions) are available. PLI provides the necessary accesses to the internal data structure of the compiled design, so test methods can be performed in such a mixed environment more easily and without having to mingle with the original design.

In this book, by means of the PLI interface, a mixed HDL/PLI test environment is proposed and the implementations of several test applications are exercised. In the sections that follow, a brief description of HDL coding and using testbench techniques combined with PLI utilities for developing test methods are given.

2.2 Using Verilog in Design

For decades, HDLs have been used to model the hardware designs as an IEEE standard [3]. Using HDLs and their simulators, digital designers are capable of partitioning their designs into components that work concurrently and are able to communicate with each other. HDL simulators can simulate the design in the presence of the real hardware delays and can imitate concurrency by switching between design parts in small time slots called “delta” delays [4]. In the following subsections, the basic features of Verilog HDL for simulation and synthesis are described.

2.2.1 Using Verilog for Simulation

The basic structure of Verilog in which all hardware components and testbenches are described is called a *module*. Language constructs, in accordance to Verilog syntax and semantics form the inside of a module. These constructs are designed to facilitate the description of hardware components for simulation, synthesis, and specification of testbenches to specify test data and monitor circuit responses. A module that encloses a design’s description can be described to test the module under design, in which case it is regarded as the testbench of the design. Figure 2.1 shows a simulation

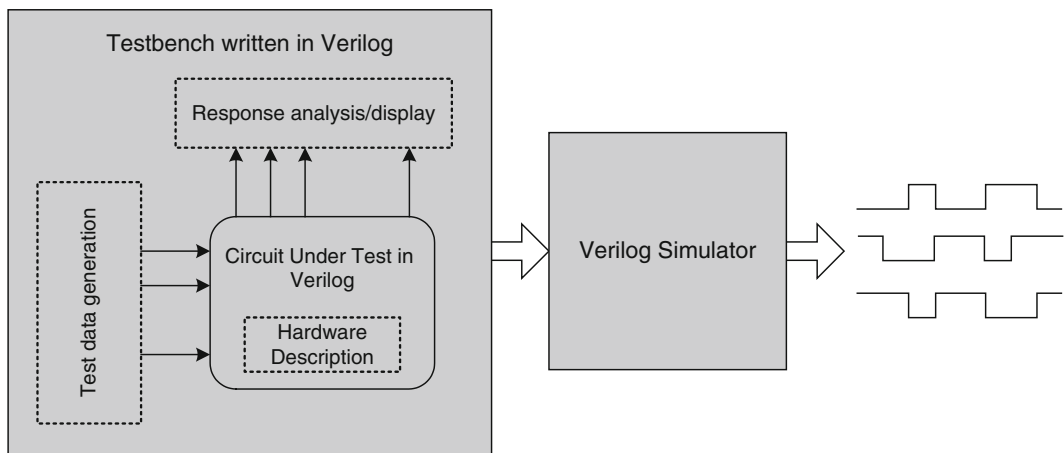


Fig. 2.1 Simulation in Verilog

model that consists of a design with a Verilog testbench. Verilog constructs (shown by dotted lines) of the Verilog model being tested are responsible for the description of its hardware, while language constructs used in a testbench are in charge of providing appropriate input data or applying data stored in a text file to the module being tested, and analysis or display of its outputs. Simulation output is generated in the form of a waveform for visual inspection or data files for record or for machine readability.

2.2.2 Using Verilog for Synthesis

After a design passes basic the functional validations, it must be synthesized into a netlist of components of a target library. The target library is the specification of the hardware that the design is being synthesized to. Verilog constructs used in the Verilog description of a design for its verification or those for timing checks and timing specifications are not synthesizable. A Verilog design that is to be synthesized must use language constructs that have a clear hardware correspondence.

Figure 2.2 shows a block diagram specifying the synthesis process. Circuit being synthesized and specification of the target library are the inputs of a synthesis tool. The outputs of synthesis are a netlist of components of the target library, and timing specification and other physical details of the synthesized design. Often synthesis tools have an option to generate this netlist in Verilog.

2.2.2.1 Postsynthesis Simulation

When the netlist is provided by the synthesis tool that uses Verilog for the description of the netlist components (Fig. 2.3), the same testbench prepared for the pre-synthesis simulation can be used with this gate-level description. This simulation, which is often regarded as post-synthesis simulation, uses timing information generated by the synthesis tool and yields simulation results with detailed timing.

Since the same testbench of the high-level design is applied to the gate-level description, the resulted waveform or printed data must be the same. This can be seen when comparing Fig. 2.1 with Fig. 2.3, while the only difference is that the post-synthesis simulation includes timing details.

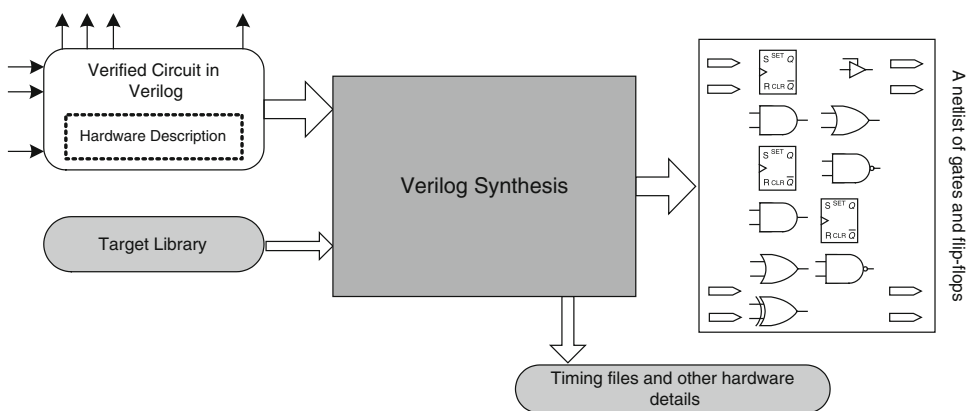


Fig. 2.2 Synthesis of a Verilog design

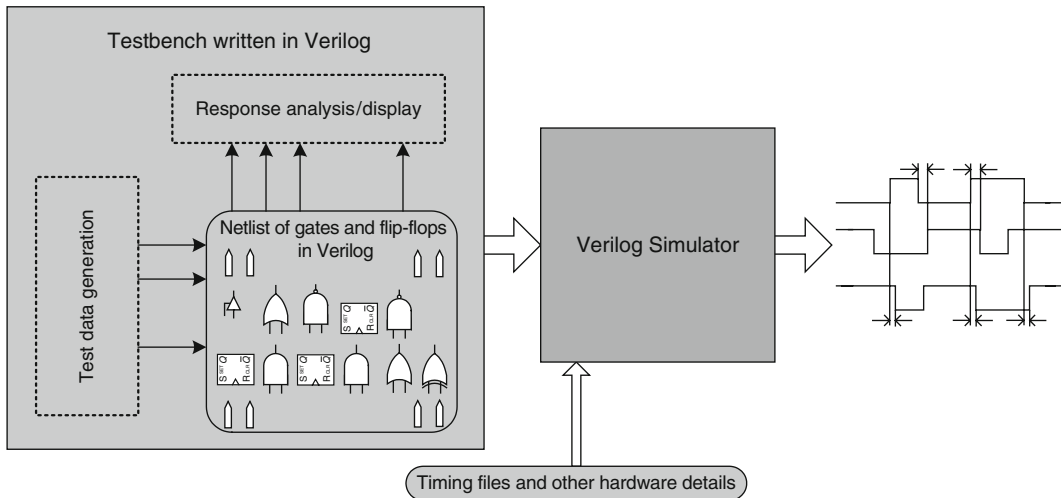


Fig. 2.3 Postsynthesis simulation in Verilog

2.3 Using Verilog in Test

As mentioned, HDL capabilities can be utilized to enhance exercising existing test methods and to develop new ones with little effort. The subsections that follow illustrate some possible usages of Verilog in the test of digital systems.

2.3.1 Good Circuit Analysis

An important tool in testing is one that generates good circuit responses from a circuit's golden model. This response is to be compared with responses from faulty circuits. By applying testbench data to the golden model, it is possible to record the good behavior of the circuit for future use. The golden signatures can also be generated this way. A signature is the result of an accumulative compression on all the outputs of the golden model. Later, when checking if a circuit is faulty or not, the same input data and the same signature collection algorithm must be applied to the design under test. By comparing the obtained signature with the recorded golden signature, the presence or absence of faults in the circuit can be verified. The precision of this fault detection depends on the compression algorithm that is used to collect the signature and on the number of test data that is applied to make this signature.

Another application of HDL simulation for testing is signature generation for various test sets or for different test procedures. Figure 2.4 depicts the good circuit analysis and its results.

2.3.2 Fault List Compilation and Testability Analysis

Fault list compilation is also one of the basic utilities that is needed to perform other test applications. For this purpose, the design described at the gate level, which is normally resulted from synthesis of a behavioral model of the design, can be used. Having fault models available for the gate models used in the gate-level description of the design, possible faults for the entire design can be generated. The capability of exploring the netlist of the circuit under test is very useful in fault compilation. Using these capabilities, the fault list of the design under test can be generated and recorded in a text

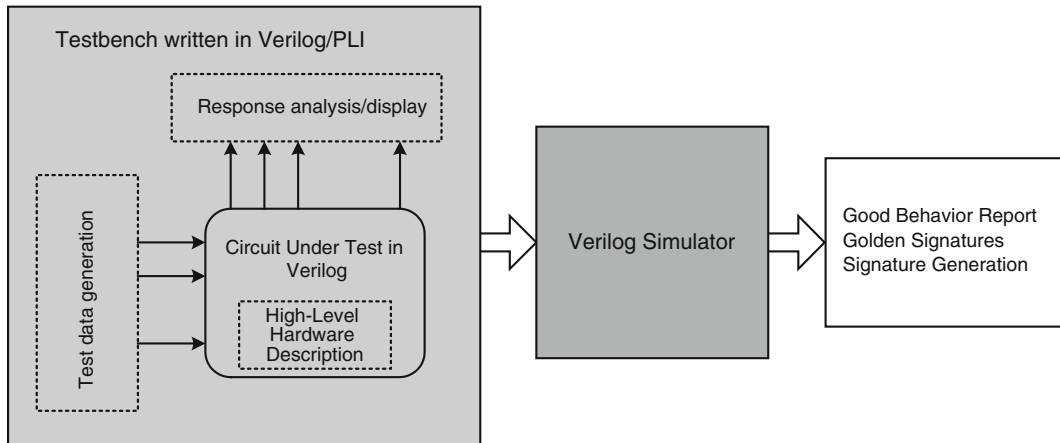


Fig. 2.4 Good circuit analysis using Verilog

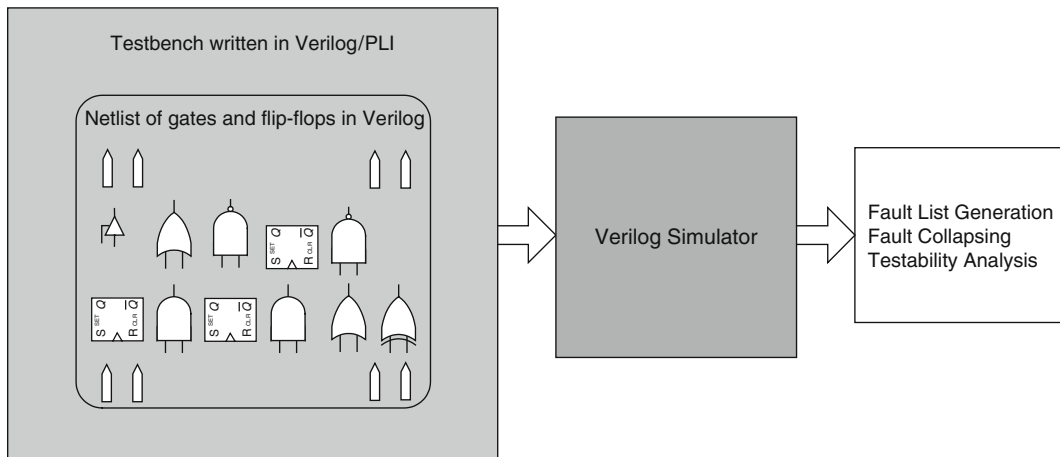


Fig. 2.5 Fault list compilation and testability measurement using Verilog

file as the fault list (Fig. 2.5). In order to reduce test time, fault collapsing, which is also implementable in the HDL environment, is performed.

Certain test applications, such as test generation or testability hardware insertion methods, need measurements to estimate how testable their internal nodes are. Methods used for fault compilations can also be used for applications such as this.

2.3.3 Fault Simulation

As mentioned, an HDL environment is able to generate a list of faults. This list can be used in an HDL simulation environment for fault simulation of the circuit under test. To complement the facilities that the HDL and its environment provide, we have developed Verilog PLI functions for producing fault models of a CUT for the purpose of fault simulation. The PLI functions inject faults in the good circuit model to create a faulty model of the CUT.

Assuming test data and the fault list and a mechanism for fault injection (FI) are available, fault simulation can be implemented in an HDL testbench. This testbench needs to instantiate golden and

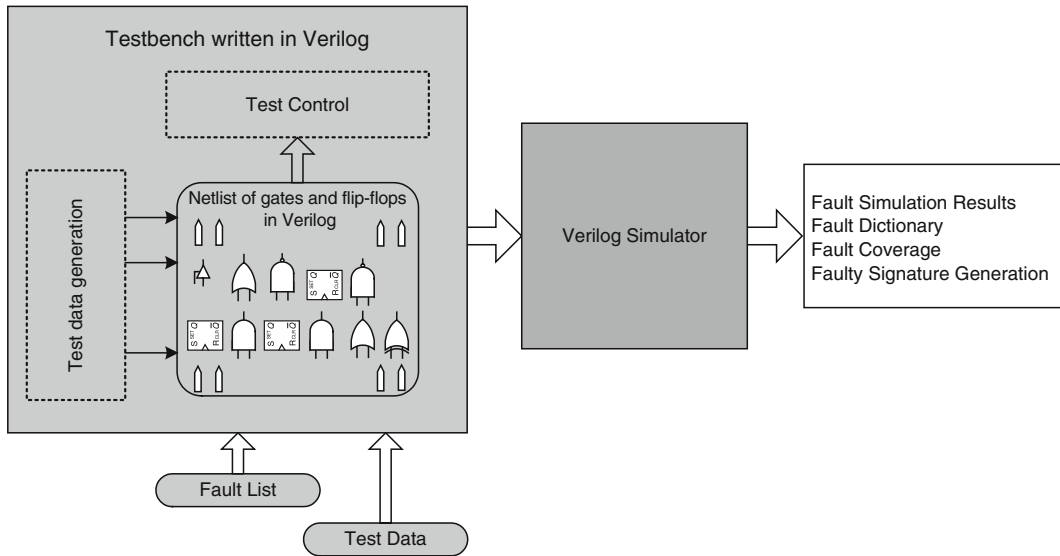


Fig. 2.6 Fault simulation using Verilog

faultable models of the circuit, and must be able to inject faults and remove them for creating various faulty models (see Fig. 2.6).

An important application of fault simulation is the calculation of fault coverage, which is a measure of the number of faults detected versus those that are not. An HDL simulation tool, with a proper testbench that instantiates a CUT, can calculate fault coverage for a test vector or a test set of the CUT.

With fault simulation, it is possible to generate a faulty signature for every one of the CUT's faults. A database containing tests, faults and their faulty signatures is called a fault dictionary that is another output that can be expected from an HDL simulation tool. When dealing with an actual faulty CUT, by performing fault simulation, collecting its signature, and comparing the resulted signature with the signatures saved in the fault dictionary, the CUT's fault can be identified and located.

2.3.4 Test Generation

Another application of Verilog PLI for test applications is test generation. The same netlist that was used for fault simulation is instantiated in the testbench of Fig. 2.7. This environment is able to inject a fault, generate some kind of random or pseudo random test data, and check if the test vector detects the injected fault. We can also find the number of undetected faults that a test vector detects. The result can be a collection of test vectors that detect a good number of circuit faults. This collection is a test set produced by HDL simulation of CUT.

2.3.5 Testability Hardware Design

Efficient design of hardware that makes a design testable is possible in an HDL environment. By means of the testability measurements and other information provided by simulating a design, we can decide on the type and the place of the testability hardware that we intend to insert into the original design. After that, by applying test generation and fault simulation applications provided in this environment, a proper

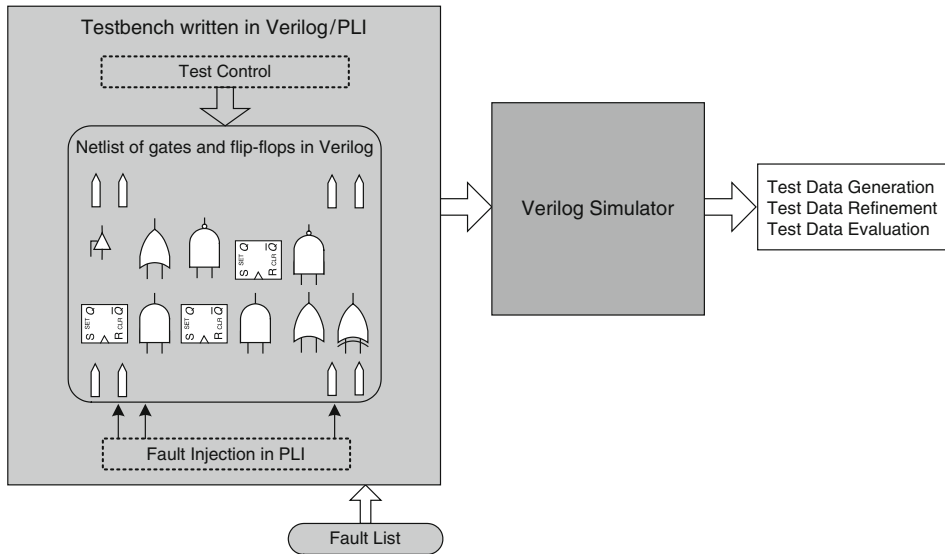


Fig. 2.7 Test generation using Verilog

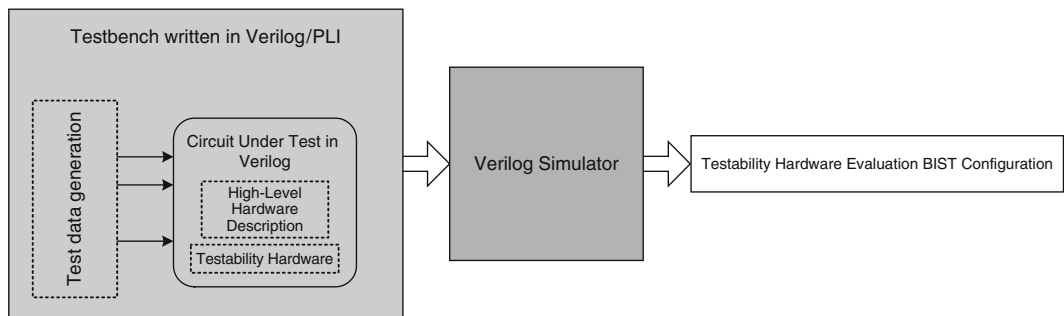


Fig. 2.8 Testability hardware design using Verilog

test set can be found for the new circuit, and the testbench can act as a virtual tester for the DFT-inserted circuit. In this case, various testability factors of the new circuit, such as new testability measurements, fault coverage, test time, and even power consumption estimation during test, can be obtained.

Along with this DFT evaluation, changing the configuration of the testability hardware is also possible. For this purpose, the important parameters of the DFT, such as the place for inserting test points, the length and the number of scan chains, and the number of clocks in the BIST circuit, can be changed until the best possible configuration is obtained (Fig. 2.8).

2.4 Basic Structures of Verilog

As mentioned, all the design and test processes described in this book are implemented in Verilog. The following subsections cover the basics of this language and the rules of describing designs in various levels of abstraction. For more details on HDL modeling and testbenches, the reader is encouraged to refer to [5, 6].

In the examples in this chapter, Verilog keywords and reserved words are shown in **bold**. Verilog is case sensitive. It allows letters, numbers, and special character “_” to be used for names. Names are used for modules, parameters, ports, variables, wires, signals, and instance of gates and modules.

2.4.1 Modules, Ports, Wires, and Variables

The main structure used in Verilog for the description of hardware components and their testbenches is a *module*. A module can describe a hardware component as simple as a transistor or a network of complex digital systems. As shown in Fig. 2.9, modules begin with the **module** keyword and end with **endmodule**. A complete design may consist of several modules. A design file describing a design takes the .v extension. For describing a system, it is usually best to include only one module in a design file.

A design may be described in a hierarchy of other modules. The top-level module is the complete design and modules lower in the hierarchy are the design’s components. Module instantiation is the construct used for bringing a lower level module into a higher level one. Figure 2.9 shows a hierarchy of several nested modules.

The first part of a module description that begins with the **module** keyword and ends with a semicolon is regarded as its header. As shown in Fig. 2.10, in addition to the **module** keyword, a

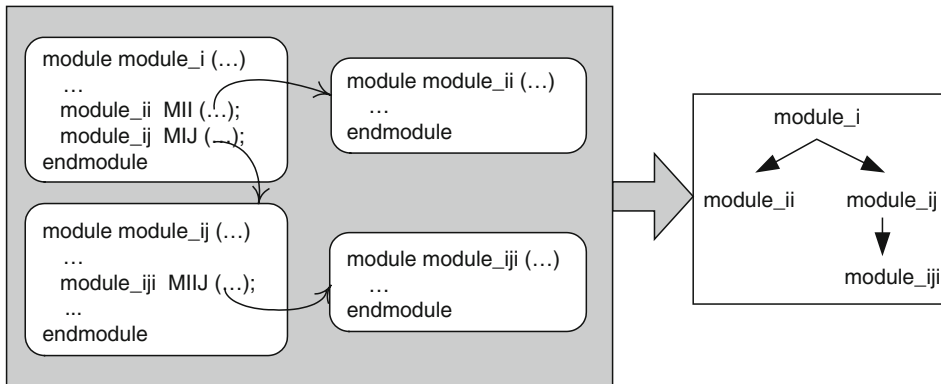


Fig. 2.9 Module outline and hierarchy

```

module acircuit (input a, b, input[7:0] av, bv, output w, output[7:0] wv);
    wire d, c;
    wire [7:0] dv;
    reg e;
    reg [7:0] ev;

    assign d = a & b;
    assign dv = av & bv;
    assign w [6:0] = av [7:1] & dv [7:1];
    assign cv[7] = d ^ bv[3];

    always @(av,bv,a,b) begin
        ev = {av[3:0],bv[7:4]}
        e = a | b;
    end

    assign wv = ev;

endmodule

```

Fig. 2.10 Port, wire, and variable declaration

module header includes the module name and list of its ports. Port declarations specifying the mode of a port (i.e. input, output, etc.), and its length can be included in the header or as separate declarations in the body of the module. Module declarations appear after the module header. A port may be **input**, **output**, or **inout**. The latter type is used for bidirectional input/output lines. The size of a multibit port comes in a pair of numbers separated by a colon and bracketed by square brackets. The number on the left of the colon is the index of the left most bit of the vector, and that on the right is the index of the right most bit of the vector.

In addition to ports not declared in the module header, this part can include declaration of signals used inside the module or temporary variables. Wires (that are called **net** in Verilog) are declared by their types, **wire**, **wand**, or **wor**; and variables are declared as **reg**. Wires are used for interconnections and have properties of actual signals in a hardware component. Variables are used for behavioral descriptions and are similar to variables in software languages. Figure 2.10 shows several wire and variable declarations.

Wires represent simple interconnection wires, busses, and simple gate or complex logical expression outputs. When wires are used on the left-hand side of **assign** statements, they represent outputs of logical structures. Wires can be used in scalar or vector form. Multiple concurrent assignments to a **net** are allowed and the value that the wire receives is the resolution of all concurrent assignments to the **net**. Figure 2.10 includes several examples of wires used on the right and left hand sides of **assign** statements.

In contrast to a **net**, a **reg** variable type does not represent an actual wire and is primarily used as variables are used in a software language. In Verilog, we use a **reg** type variable for temporary variables, intermediate values, and storage of data. A **reg** type variable can only be used in a procedural body of Verilog. Multiple concurrent assignments to a **reg** should be avoided.

In the vector form, inputs, outputs, wires, and variables may be used as a complete vector, part of a vector, or a bit of the vector. The latter two are referred to as part-select and bit-select. Examples of part-select and bit-select on right and left hand sides of an **assign** statement are shown in Fig. 2.10. The statement that assigns the *ev* **reg**, besides part-select indexing, illustrates concatenation of *av*[3:0] and *bv*[7:4] and assigning the result to *ev*. This structure especially is useful to model swapping and shifting operations.

2.4.2 Levels of Abstraction

Operation of a module can be described at the gate level, using Boolean expressions, at the behavioral level, or a mixture of various levels of abstraction. Figure 2.11 shows three ways the same operation can be described. Module *simple_1a* uses Verilog's gate primitives, *simple_1b* uses concurrent statements, and *simple_1c* uses a procedural statement. Module *simple_1a* describes instantiation of three gate primitives of Verilog. In contrast, *simple_1b* uses Boolean expressions to describe the same functions for the outputs of the circuit. The third description, *simple_1c*, uses a conditional **if** statement inside a procedural statement to generate proper function on one output, and uses a procedural Boolean function for forming the other circuit output.

2.4.3 Logic Value System

Verilog uses a 4-value logic value system. Values in this system are 0, 1, Z, and X. Value 0 is for logical 0 which in most cases represents a path to ground (Gnd). Value 1 is logical 1 and it represents a path to supply (Vdd). Value Z is for float, and X is used for uninitialized, undefined, undriven, unknown, and value conflicts. Values Z and X are used for wired-logic, busses, initialization values, tristate structures, and switch-level logic.

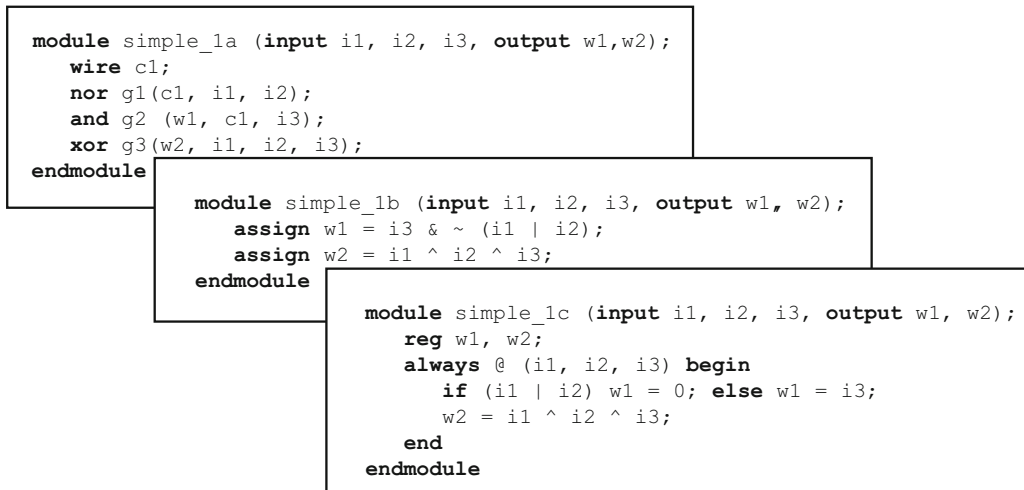


Fig. 2.11 Module definition alternatives

A gate input, or a variable or signal in an expression on the right-hand side of an assignment can take any of the four logic values. Output of a two-valued primitive gate can only take 0, 1, and X while output of a tristate gate or a transistor primitive can also take a Z value. A right-hand-side expression can evaluate to any of the four logic values and can thus assign 0, 1, Z, or X to its left-hand-side **net** or **reg**.

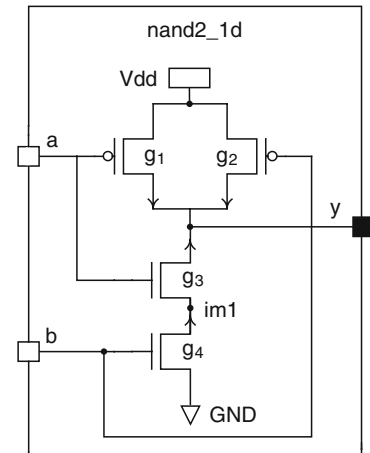
2.5 Combinational Circuits

A combinational circuit can be represented by its gate-level structure, its Boolean functionality, or description of its behavior. At the gate level, interconnection of its gates are shown; at the functional level, Boolean expressions representing its outputs are written; and at the behavioral level a software-like procedural description represents its functionality. At the beginning of this section, implementation of a NAND gate using primitive transistors of Verilog as a glance to transistor-level design is illustrated. Afterward, the implementation of a 2-to-1 multiplexer is described in various levels of abstraction to cover important concepts in combinational circuits. Examples for combining various forms of descriptions and instantiation of existing components are also shown.

2.5.1 Transistor-level Description

Verilog has primitives for unidirectional and bidirectional MOS and CMOS structures [7]. As an example of instantiation of primitive transistors of Verilog, consider the two-input CMOS NAND gate shown in Fig. 2.12.

The Verilog code of Fig. 2.13 describes this CMOS NAND gate. Logically, nMOS transistors in a CMOS structure push 0 into the output of the gate. Therefore, in the Verilog code of the CMOS NAND, input to output direction of nMOS transistors are from *Gnd* toward *y*. Likewise, nMOS

Fig. 2.12 CMOS NAND gate

```

module cmos_nand ( input a, b, output y );
    wire im1;
    supply1 vdd;
    supply0 gnd;

    pmos # (4, 5)
        g1 (y, vdd, a),
        g2 (y, vdd, b);
    nmos # (3, 4)
        g3 (im1, gnd, b),
        g4 (w, im1, a);
endmodule

```

Fig. 2.13 CMOS NAND Verilog description

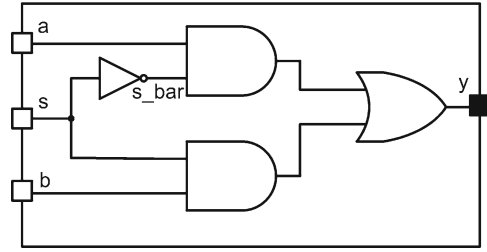
transistors push a 1 value into *y*, and therefore, their inputs are considered the *Vdd* node and their outputs are connected to the *y* node. The *im1* signal is an intermediate **net** and is explicitly declared.

2.5.2 Gate-level Description

We use the multiplexer circuit of Fig. 2.14 to illustrate how primitive gates are used in a design. The description shown in Fig. 2.15 corresponds to this circuit. The module description has inputs and outputs according to the schematic of Fig. 2.14.

The statement that begins in Line 6 and ends in Line 8 instantiates two **and** primitives. The construct that follows the primitive name specifies 0-to-1 and 1-to-0 propagation delays for the instantiated primitive ($t_{plh} = 2$, $t_{phl} = 4$). This part is optional and if eliminated, 0 values are assumed t_{plh} and t_{phl} delays.

Line 7 shows inputs and outputs of one of the two instances of the **and** primitive. The output is *im1* and inputs are module input ports *a* and *b*. The port list on Line 7 must be followed by a comma if other instances of the same primitive are to follow, otherwise a semicolon should be used, like the end of Line 9. Line 8 specifies input and output ports of the other instance of the **and** primitive. Line 10 is for instantiation of the **or** primitive at the output of the majority gate. The output of this gate is

Fig. 2.14 2-to-1 multiplexer circuit

```

module mux2to1 ( a, b, s, y );
  input a, b, s;
  output y;

  not #(1,1) (s_bar, s);           //Line 05
  and #(2,4) (im1, a, s_bar ),    //Line 06
      ( im2, b, s );              //Line 07
  or  #(3,5) ( y, im1, im2);      //Line 08
endmodule

```

Fig. 2.15 Verilog code for the multiplexer circuit

y that comes first in the port list, and is followed by inputs of the gate. In this example, intermediate signals for interconnection of gates are *im1*, *im2*, and *s_bar*. Scalar interconnecting wires need not be explicitly declared in Verilog. The two **and** instances could be written as two separate statements, like instantiation of the **or** primitive. If we were to specify different delay values for the two instances of the **and** primitive, we had to have two separate primitive instantiation statements.

2.5.3 Equation-level Description

At a higher level than gates and transistors, a combinational circuit may be described by the use of Boolean, logical, and arithmetic expressions. For this purpose, the Verilog concurrent **assign** statement is used. Table 2.1 shows Verilog operators that can be used with **assign** statements.

Figure 2.16 shows a 2-to-1 multiplexer using a conditional operator. The expression shown reads as follows: **if** s is 1, then y is $i1$ **else** it becomes $i0$.

If there is more than one **assign** statement, because of the concurrency property of Verilog, the order in which they appear in module is not important. These statements are sensitive to events on their right-hand sides. When a change of value occurs on any of the right-hand-side **net** or variables, the statement is evaluated and the resulting value is scheduled for the left-hand side **net**.

2.5.4 Procedural Level Description

At a higher level of abstraction than describing hardware with gates and expressions, Verilog provides constructs for procedural description of hardware. Unlike gate instantiations and **assign**

Table 2.1 Verilog operators

Bitwise operators	&		^	~	~^	^~	
Reduction operators	&	~&		~	^	~^	^~
Arithmetic operators	+	-	*	/	%		
Logical operators	&&		!				
Compare operators	<	>	<=	>=	++		
Shift operators	>>	<<					
Concatenation operators	{ }	{n{ }}					
Conditional operators	?:						

```

module mux2_1 (input [3:0] i0, i1, input s, output [3:0] y );
    assign y = s ? i1 : i0;
endmodule

```

Fig. 2.16 A 2-to-1 Multiplexer using condition operator

statements that correspond to concurrent substructures of a hardware component, procedural statements describe the hardware by its behavior. Also, unlike concurrent statements that appear directly in a module body, procedural statements must be enclosed in procedural blocks before they can be put inside a module.

The main procedural block in Verilog is the **always** block. This is considered a concurrent statement that runs concurrent with all other statements in a module. Within this statement, procedural statements like **if-else** and **case** statements are used and are executed sequentially. If there are more than one procedural statement inside a procedural block, they must be bracketed by **begin** and **end** keywords.

Unlike assignments in concurrent bodies that model driving logic for left-hand-side wires, assignments in procedural blocks are assignments of values to variables that hold their assigned values until a different value is assigned to them. A variable used on the left hand side of a procedural assignment must be declared as **reg**.

An event control statement is considered a procedural statement, and is used inside an **always** block. This statement begins with an at-sign, and in its simplest form, includes a list of variables in the set of parenthesis that follow the at-sign, e.g., @ (v1, v2,...).

When the flow of the program execution within an **always** block reaches an event-control statement, the execution halts (suspends) until an event occurs on one of the variables in the enclosed list of variables. If an event-control statement appears at the beginning of an **always** block, the variable list it contains is referred to as the *sensitivity list* of the **always** block. For combinational circuit modeling, all variables that are read inside a procedural block must appear on its sensitivity list.

2.5.4.1 Multiplexer Example

As an example of a procedural block, consider the 2-to-1 multiplexer of Fig. 2.17. This example uses an **if-else** construct to set *y* to *i0* or *i1* depending on the value of *s*. As in the previous examples, all circuit variables that participate in the determination of value of *y* appear on the sensitivity list of the **always** block. Also since *y* appears on the left-hand side of a procedural assignment, it is declared as **reg**.

The **if-else** statement shown in Fig. 2.17 has a condition part that uses an equality operator. If the condition is true (i.e., *s* is 0), the block of statements that follow it will be taken, otherwise the block of statements after the **else** is taken. In both cases, the block of statements must be bracketed by **begin** and **end** keywords if there are more than one statement in a block.

```

module mux2_1 (input i0, i1, output reg s, y );
    always @( i0, i1, s ) begin
        if ( s==1'b0 )
            y = i0;
        else
            y = i1;
        end
    endmodule

```

Fig. 2.17 Procedural multiplexer

```

module alu_4bit (input [3:0] a, b, input [1:0] f, output reg [3:0] y );
    always @ ( a or b or f ) begin
        case ( f )
            2'b00 : y = a + b;
            2'b01 : y = a - b;
            2'b10 : y = a & b;
            2'b11 : y = a ^ b;
            default: y = 4'b0000;
        endcase
    end
endmodule

```

Fig. 2.18 Procedural ALU

2.5.4.2 Procedural ALU Example

The **if-else** statement, used in the previous example, is easy to use, descriptive, and expandable. However, when many choices exist, a **case** statement which is more structured may be a better choice. The ALU description of Fig. 2.18 uses a **case** statement to describe an ALU with add, subtract, AND, and XOR functions. The **case** statement shown in the **always** block uses f to select one of ALU functions in the **case** alternatives. The last alternative is the **default** alternative that is taken when f does not match any of the alternatives that appear before it. This is necessary to make sure that unspecified input values (here, those that contain X and/or Z) cause the assignment of the default value to the output and do not leave it unspecified.

2.5.5 Instantiating Other Modules

We have shown how primitive gates can be instantiated in a module and wired with other parts of the module. The same applies to instantiating a module within another. For regular structures, Verilog provides repetition constructs for instantiating multiple copies of the same module, primitive, or set of constructs. Examples in this section illustrate some of these capabilities.

2.5.5.1 ALU Example Using Adder

The ALU of Fig. 2.18 starts from scratch and implements every function it needs inside the module. If we have a situation that we need to use a specific design from a given library, or we have a function that is too complex to be repeated everywhere it is used, we can make it into a module and instantiate it when we need to use it.

```

module ALU_Adder (input [7:0] a,b, input addsub, // Line 01
                  output gt, zero, co, output [7:0] r );
    wire [7:0] b_bbar;
    add_8bit ADD (a, b_bbar, addsub, r, co);           // Line 04
    assign b_bbar = addsub ? ~b : b;                 // Line 05
    assign gt = (a>b);
    assign zero = (r == 0);
endmodule

```

Fig. 2.19 ALU Verilog code using instantiating an adder

Figure 2.19 shows another version of the above ALU circuit. In this new version, addition is handled by instantiation of a predesigned adder (*add_8bit*). Instantiation of a component, such as *add_8bit* in the above example, starts with the component name, an instance name (*ADD*), and the port connection list. The latter part decides how local variables of a module are mapped to the ports of the component being instantiated. The above example uses an ordered list, in which a local variable, e.g., *b_bbar*, takes the same position as the port of the component it is connecting to, e.g., *b*. Alternatively, a named port connection such as that shown below can be used.

$$\text{add_8bit ADD } (.a(a),.b(b_bbar),.ci(addsub),.s(r),.co(co));$$

Using this format allows port connections to be made in any order. Each connection begins with a dot, followed by the name of the port of the instantiated component, e.g. *b*, and followed by a set of parenthesis enclosing the local variable that is connected to the instantiated component, e.g. *b_bbar*. This format is less error-prone than the ordered connection.

2.5.5.2 Iterative Instantiation

Verilog uses the **generate** statement for describing regular structures that are composed of smaller sub-components. An example is a large memory array or a systolic array multiplier. In such cases, a cell unit of the array is described, and by means of several **generate** statements, it is repeated in several directions to cover the entire array of the hardware.

Here, we show the description of a parametric *n*-bit AND gate using this construct. Obviously, *n*-input gates can be easily obtained by using vector inputs and outputs for Verilog primitives. However, the example shown in Fig. 2.20 besides illustrating the iterative **generate** statement of Verilog, introduces the structure of components that are used in this book to describe gate-level circuits for test applications. This description is chosen due to the PLI requirements for implementing test applications that are discussed later.

The code of Fig. 2.20 uses the **parameter** construct to prepare parametric size and delays for this AND gate. In the body of this module on Line 8, a variable for generating *n* instances of **and** primitive is declared using the **genvar** declaration. The **generate** statement that begins on Line 10 loops *n* times to generate an instance of the **and** gate in every iteration. Together, the *and_0* instance and the **generate** statement make enough **and** gates to AND together bits 0 to *n*-1 of input vector *in*. This is done by use of the intermediate wire, *mwire*. Since the resulted *and_n* must represent a bitwise function, *mwire net* is declared to accumulate bit-by-bit AND results. Line 9 shows the first two bits of the *in* input vector ANDed using the **and** primitive, and the result is collected in bit 0 of *mwire*. After that, each instanced **and** in the **generate** statement takes the next bit from *in* and ANDs it with the calculated bit of *mwire* to generate the next bit of *mwire*. The resulted hardware for this parametric *and_n* gate, is concatenation of 2-input **and** primitives that AND all bits of the *in* input vector.

The complete component library for test applications of this book can be found in Appendix B.

```

module and_n
  #(parameter n = 2, tphl = 1, tplh = 1) (out,in);

  input [n-1:0] in;
  output out;
  wire [n-2:0] mwire;

  genvar i;                                     //Line 08
  and and_0 (mwire [0], in [0], in [1]);         //Line 09
  generate                                       //Line 10
    for (i=1; i <= n-2; i=i+1) begin : AND_N      //Line 11
      and inst (mwire [i], mwire [i-1], in [i+1]); //Line 12
    end
  endgenerate

  bufif1 #(tplh, tphl) inst(out, mwire [n-2], 1'b1); //Line 16

endmodule

```

Fig. 2.20 Using iterative instantiation for test primitive AND gate

2.6 Sequential Circuits

As with any digital circuit, a sequential circuit can be described in Verilog by the use of gates, Boolean expressions, or behavioral constructs (e.g., the **always** statement). While gate-level descriptions enable a more detailed description of timing and delays because of complexity of clocking and register and flip-flop controls, these circuits are usually described by the use of procedural **always** blocks. This section shows various ways sequential circuits are described in Verilog.

2.6.1 Registers and Shift Registers

Figure 2.21 shows an 8-bit register with *set* and *reset* inputs that are synchronized with the clock. The *set* input puts all 1s in the register, and the *reset* input resets it to all 0s. The sensitivity list of the procedural statement shown includes **posedge** of *clk*. This **always** statement only wakes up when *clk* makes a 0 to 1 transition. When this statement does wake up, the value of *d* is put into *q*. Obviously, this behavior implements a rising-edge register. Instead of **posedge**, the use of **negedge** would implement a falling-edge register.

In order to provide procedural description for shift registers the concatenation construct can be used as shown in Fig. 2.22. This partial code, that can be used in the body of an **always** statement like that of Fig. 2.21, does a left-shift if *shift_left* is 1, and right shifts, otherwise.

2.6.2 State Machine Coding

Along with simple sequential circuits, such as registers, shift registers, and counters, Verilog constructs enable the designer to model finite state machines of any type. State machines can be modeled as Moore or Mealy machines. In both cases, based on the current state of the sequential circuit and its input, the next state is decided. The difference is in the determination of outputs. Unlike a Moore machine that has outputs that are only determined by the current state of the machine, in a

```

module register (input [7:0] d, input clk, set, reset, output reg [7:0] q);
    always @ ( posedge clk ) begin
        if ( set )
            q <= 8'b1;
        else if ( reset )
            q <= 8'b0;
        else
            q <= d;
    end
endmodule

```

Fig. 2.21 An 8-bit register

```

if ( shift_left )
    q <= {q[6:0], s_in};
else
    q <= {s_in, q[7:1]};

```

Fig. 2.22 Concatenation for a 8-bit shift register

Mealy machine, the outputs are declared regarding the state the machine is in as well as the inputs of the circuit. This makes Mealy outputs not fully synchronized with the circuit clock.

This section shows coding for state machines and introduces the Huffman coding style. The example we use is a Residue-5 divider. The coding styles used here apply to such controllers and are used in later sections of this chapter to describe the controller of a simple adding machine. It must be mentioned that the Residue-5 example presented here is one of the test cases for the application of test methods in this book. Simpler and more detailed examples can be found in [6].

2.6.2.1 Residue-5 Divider

The Residue-5 divider is a circuit that performs the integral division modulo-5 on the sequences coming on its input. For this purpose, the circuit divides the first received input by five and stores the remainder. For the next data on the input port, the circuit adds the new value to the stored remainder, divides the result by 5, and stores the new remainder. This circuit can be modeled using a finite state machine. The remainder stored in this circuit shows its internal state and its output. State diagram for the Residue-5 divider using 2-bit input x is depicted in Figs. 2.23 and 2.24. For the sake of readability, Fig. 2.23 just includes arcs related to two states.

The machine has five states that are labeled, *Zero*, *One*, *Two*, *Three*, and *Four*; each of which shows the resulted Residue-5 remainder. In the Moore state machine modeling, the output depends just on the current state, so in Fig. 2.23 the output is defined for each state. In addition to the x input, the machine has a *reset* input that forces the machine into its *Zero* state. The resetting of the machine is synchronized with the circuit clock.

2.6.2.2 The Moore Implementation of Residue-5 in Verilog

The Verilog code of the Moore machine of Fig. 2.24 is shown in Fig. 2.25. After the declaration of inputs and outputs of this module, **parameter** declaration declares five states of the machine as 3-bit parameters. The square-brackets following the **parameter** keyword specify the size of parameters being declared. Following parameter declarations in the code of Fig. 2.25, the 3-bit *current* **reg** type

Fig. 2.23 A part of Residue-5 Moore state machine

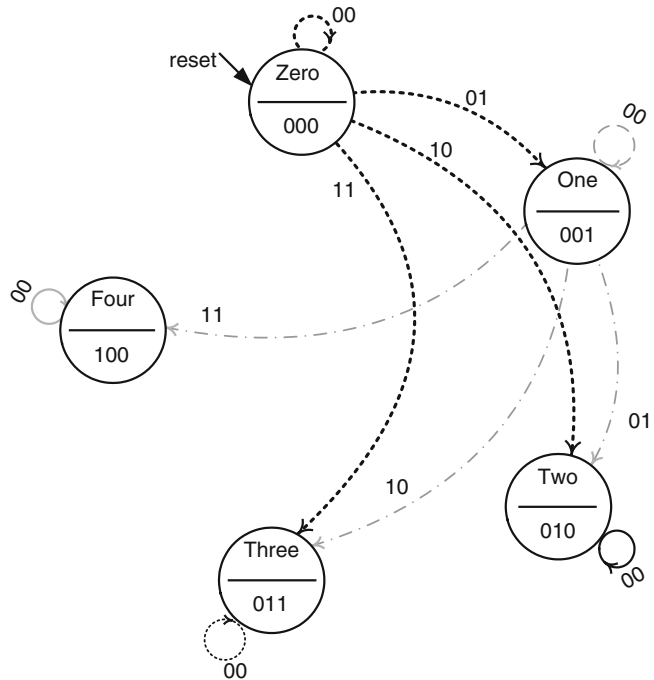
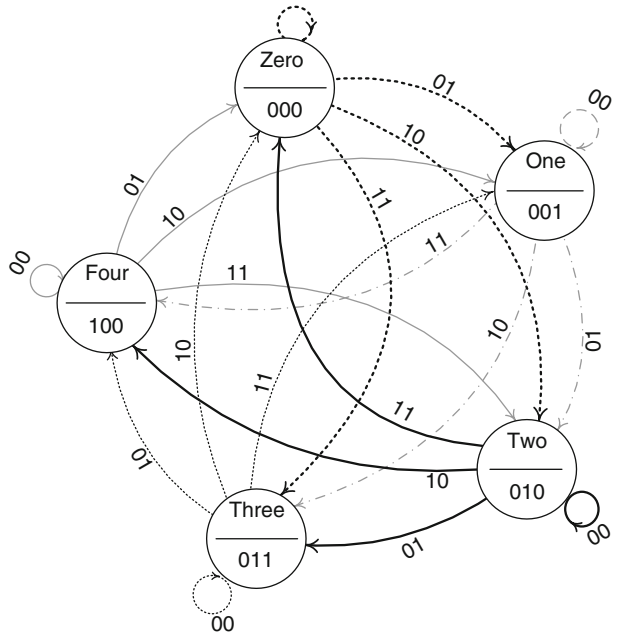


Fig. 2.24 Complete Residue-5 Moore state machine



variable is declared. This variable holds the current state of the state machine. The body of the code of this circuit has an **always** block and an **assign** statement.

The **assign** statement shown in Fig. 2.25 puts the proper value on the output regarding the current state. This statement is concurrent with the **always** block that is responsible for making the state transitions. The **always** block used in the module of Fig. 2.25 describes state transitions of the state

```

module residue5(input clk, reset, input[1:0] x, output[2:0] out);
reg[2:0] current;
parameter Zero = 3'b000, One = 3'b001, Two = 3'b010,
           Three = 3'b011, Four = 3'b100;
always @(posedge clk) begin
    if(reset == 1)
        current <= Zero;
    else
        case(current)
            Zero: case(x)
                2'b00: current <= Zero;
                2'b01: current <= One;
                2'b10: current <= Two;
                2'b11: current <= Three;
            endcase
            One: case(x)
                2'b00: current <= One;
                2'b01: current <= Two;
                2'b10: current <= Three;
                2'b11: current <= Four;
            endcase
            Two: case(x)
                2'b00: current <= Two;
                2'b01: current <= Three;
                2'b10: current <= Four;
                2'b11: current <= Zero;
            endcase
            Three: case(x)
                2'b00: current <= Three;
                2'b01: current <= Four;
                2'b10: current <= Zero;
                2'b11: current <= One;
            endcase
            Four: case(x)
                2'b00: current <= Four;
                2'b01: current <= Zero;
                2'b10: current <= One;
                2'b11: current <= Two;
            endcase
            default: current <= Zero;
        endcase
    end
    assign out = current;
endmodule

```

Fig. 2.25 Moore machine Verilog code

diagram of Fig. 2.24. The main task of this procedural block is to inspect input conditions (values on *reset* and *x*) during the present state of the machine defined by *current* and set values into *current* for the next state of the machine.

The flow into the **always** block begins with the positive edge of *clk*. Since all activities of this machine are synchronized with the clock, only *clk* appears on the sensitivity list of the **always** block. Upon entry into this block, the *reset* input is checked and if it is active, *current* is set to *Zero* (*Zero* is a declared **parameter** and its value is 0). The value put into *current* in this pass through the **always** block gets checked in the next pass with the next edge of the clock. Therefore, assignments to *current* are regarded as the next-state assignment. When such an assignment is made, the **case** statement skips the rest of the code of the **always** block, and this **always** block will next be entered with the next positive edge of *clk*. Upon entry into the **always** block, if *reset* is not 1, program flow reaches the **case** statement that checks the value of *current* against the five states of the machine.

Figure 2.26 shows the Verilog code of the *Two* state and its diagram from the state diagram of Fig. 2.24. As shown, the **case** alternative that corresponds to the *Two* state specifies the next values for that state. Determination of the next state is based on the value of *x*. If *x* is 1, the next state becomes *Three*, and if *x* is 2, the next state becomes *Four*, and so on. As shown in the **assign** statement in Fig. 2.25, the output bits of this circuit are taken directly from the *current* register.

This same machine can be described in Verilog in several different forms. A finite state machine can also be described as a Mealy machine. As mentioned, in this case the output depends not only on the current state, but also on the input of the circuit. In Mealy machines, the output becomes available one cycle sooner than that of a Moore machine, causing fewer states than Moore.

2.6.2.3 Huffman Coding Style

The Huffman model for a digital system characterizes it as a combinational block with feedbacks through an array of registers. Verilog coding of digital systems, according to the Huffman model, uses an **always** statement for describing the register part and another concurrent statement for describing the combinational part. This model of representing a digital component is very useful for test purposes, as we see in the chapters that follow.

We describe the state machine of Fig. 2.24 to illustrate this style of coding. Figure 2.27 shows the combinational and register part partitioning that we use for describing this machine.

Fig. 2.26 Next values from state *two*

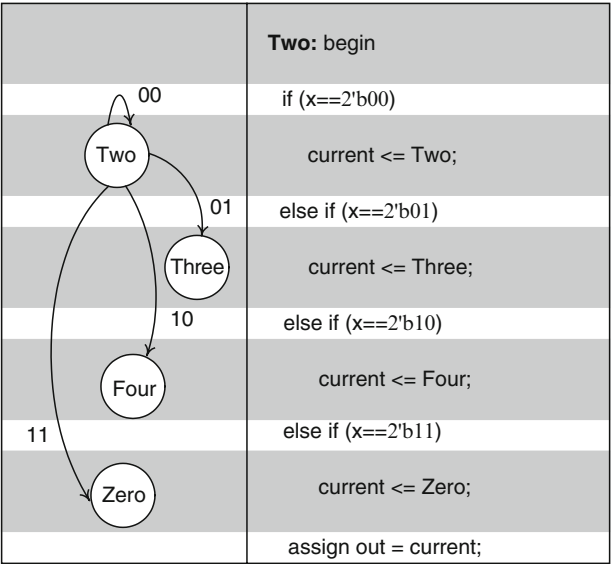
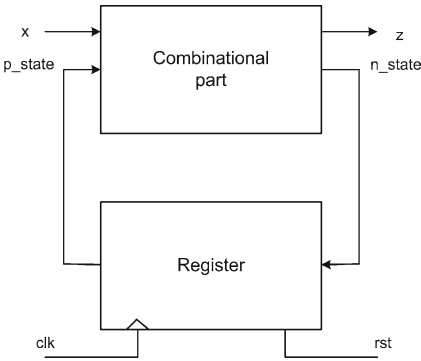


Fig. 2.27 Huffman partitioning of Residue-5 divider



```

module residue5_huffman(input clk, rst, input[1:0] x, output[2:0] out);
  reg[2:0] n_state, p_state;
  parameter Zero = 3'b000, One = 3'b001, Two = 3'b010,
             Three = 3'b011, Four = 3'b100;
  always@(p_state, x) begin
    n_state = Zero;
    case(p_state)
      Zero:...n_state = ...
      One:... n_state = ...
      Two:... n_state = ...
      Three:... n_state = ...
      Four:... n_state = ...
      default:...
    endcase
  end// Combinational part
  always@(posedge clk, posedge rst) begin
    if(rst)
      p_state = Zero;
    else
      p_state = n_state;
    end// Register part

  assign out = p_state;
endmodule

```

Fig. 2.28 Verilog Huffman coding style

The *combinational* block uses x and p_state as input and generates out and n_state . The *register* block clocks n_state into p_state , and resets p_state when rst is active.

Figure 2.28 shows the Verilog code of Fig. 2.24 according to the partitioning of Fig. 2.27. As shown, parameter declaration declares the states of the machine. Following this declaration, n_state and p_state variables are declared as 3-bit **regs** that hold values corresponding to the five states of the Moore Residue-5 divider. The *combinational* **always** block follows this **reg** declaration. Since this is purely a combinational block, it is sensitive to all its inputs, namely, x and p_state . Immediately following the block heading, n_state is set to its inactive or reset value. This is done so that this variable is always reset with the clock to make sure it does not retain its old value. Note that retaining old values implies latches, which is not what we want in our combinational block.

The body of the combinational **always** block of Fig. 2.28 contains a **case** statement that uses the p_state input of the **always** block for its **case** expression. This expression is checked against the states of the Moore machine. As in the other styles discussed before, this **case** statement has **case** alternatives for all of the states. For brevity, the statements in the **case** alternatives are not shown. These statements set the n_state variable using the same procedure as setting the *current* variable in Fig. 2.25. In a block corresponding to a **case** alternative, based on input values, n_state is assigned values. Unlike the other style where *current* is used both for the present and next states, here we use two different variables, p_state and n_state .

The next procedural block shown in Fig. 2.28 handles the register part of the Huffman model of Fig. 2.27. In this part, n_state is treated as the register input and p_state as its output. On the positive edge of the clock, p_state is either set to the *Zero* state (000) or is loaded with contents of n_state . Together, *combinational* and *register* blocks describe our state machine in a very modular fashion.

As with the other style we presented, a separate **assign** statement (or any other concurrent statement) is used for the assignment of values to the output. The advantage of this style of coding is in its modularity and defined tasks of each block. State transitions are handled by the *combinational* block and clocking is done by the *register* block. Changes in clocking, resetting, enabling, or presetting the machine only affect the coding of the *register* block. In this code, the a synchronous resetting is applied.

2.7 A Complete Example (Adding Machine)

In this section, the complete RTL design of a simple CPU is described. Although this design has the structure of a simple CPU, since its ALU actually just performs adding operation, we refer to it as *Adding Machine*. In this part, almost all Verilog constructs explained in this chapter are exercised. Furthermore, the basics of RTL design and datapath and controller partitioning are introduced. Later, this Adding Machine is used as one of the test cases in this book for demonstrating test methods.

2.7.1 Control/Data Partitioning

The first step in an RT-level design is the partitioning of the design into a data part and a control part. The data part consists of data components and the bussing structure of the design, and the control part is usually a state machine generating control signals that control the flow of data in the data part [8].

Figure 2.29 shows a general sketch of an RT-level design that is partitioned into its data and control parts. As shown in this figure, a processor is divided into *datapath* and *controller* parts. The datapath has storage elements (registers) to store intermediate data, handles transfer of data between its storage components, and performs arithmetic or logical operations on data that it stores. The datapath also has communication lines for transfer of data; these lines are referred to as *busses*. Activities in the datapath include reading from and writing into data registers, bus communications, and distributing control signals generated by the controller to the individual data components.

The controller commands the datapath to perform proper operation(s) according to the instruction it is executing. Control signals carry these commands from the controller to the datapath. Control signals are generated by the controller state machine that, at all times, knows the status of the task that is being executed and the sort of the information that is stored in datapath registers. Controller is the thinking part of a design.

2.7.2 Adding Machine Specification

The design of Adding Machine begins with the specification of the design, including the number of general purpose registers and the instruction format. The machine has two 8-bit external data buses (input bus and output bus) and a 6-bit address bus. The address bus connects to the memory in order to address locations that are being read from or written into. Data read from the memory

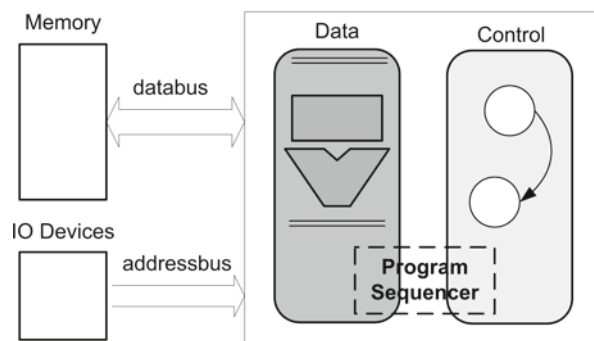


Fig. 2.29 Control/data partitioning for Adding Machine

Table 2.2 Adding machine instruction set

Opcode	Instruction	Instruction class	Description
00	add imm	Arithmetic	$AC \leftarrow AC + imm$
01	lda adr	Data-transfer	$AC \leftarrow Mem[adr]$
10	sta adr	Data-transfer	$Mem[adr] \leftarrow AC$
11	jmp adr	Control-flow	$PC \leftarrow adr$

are instructions and instruction operands, and data written into the memory are instruction results and temporary information. Adding Machine also communicates with its IO devices through its external busses. The address bus addresses a specific device or a device register while the data bus contains data that is to be written or read from the device.

Each instruction of Adding Machine is 8 bits wide, and occupies a memory word. The instruction format of the machine has an explicit operand (immediate data or memory location the address of which is specified in the instruction) and an implicit operand. Adding Machine has four instructions, divided into three classes of arithmetic (**add**), data transfer (**lda**, **sta**), and control-flow instructions (**jmp**).

Adding Machine instructions are described below. A tabular list and summary of this instruction set is shown in Table 2.2.

- **add imm**: adds the *imm* data with an 8-bit register named accumulator (*AC*) and stores the result back in *AC*.
- **lda adr**: reads the content of the memory location addressed by *adr* and writes it into *AC*.
- **sta adr**: writes the content of *AC* into the memory location addressed by *adr*.
- **jmp adr**: jump to the memory location addressed by *adr*.

2.7.3 CPU Implementation

In the following subsections, the Verilog implementation of the Adding Machine in register transfer level of abstraction is described.

2.7.3.1 Datapath Design

As mentioned, Adding Machine has an 8-bit register called accumulator (*AC*). All data transfers and arithmetic instructions use *AC* as an operand. In a real CPU, there may be multiple accumulators or an array of registers that is referred to as a register file.

To store the instruction that is read from the memory, a register is used at the output of the memory unit called instruction register (*IR*). The program counter (*PC*) is implemented as a counter that is incremented for program sequencing. Using these registers, the implementation of datapath is shown in Fig. 2.30. The input data bus connects to the input of *IR* in order to bring the instruction read from the memory into this register. Similarly, this bus connects to *AC* to bring data read from the memory into the *AC* register. The control signal for loading *IR* and *AC* are *ld_ir* and *ld_ac*, respectively. *PC* has three control signals *ld_pc*, *inc_pc*, and *clr_pc* to load, increment, and clear it, respectively. The right most 6 bits of *IR* connect to the input of *PC* for the execution of the *jmp* instruction. When a bus has more than one source driving it, e.g., *IR* and *PC* driving *adr_bus*, a multiplexer and control signals from the controller select the source.

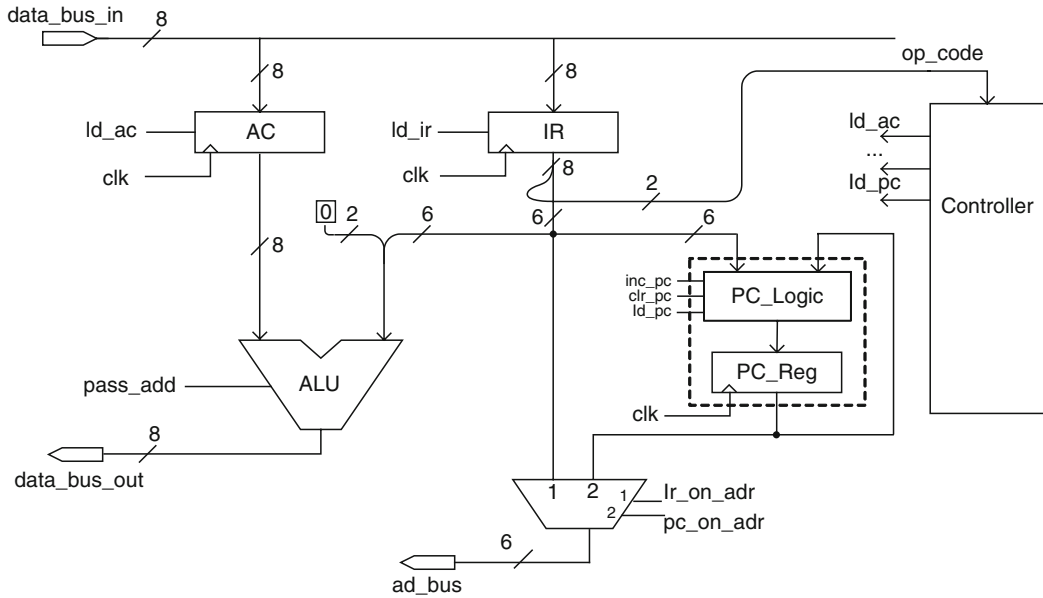


Fig. 2.30 Adding machine multicycle datapath

2.7.3.2 Controller Design

After the design of the datapath and figuring control signals and their role in activities in the datapath, the design of the controller becomes a simple matter. The block diagram of this part is shown in Fig. 2.31.

The controller of our Adding Machine has four states, *Reset*, *Fetch*, *Decode*, and *Execute*. As the machine cycles through these states, various control signals are issued. In state *Reset*, for example, the *clr_pc* control signal is issued. State *Fetch* issues *pc_on_adr*, *rd_mem*, *ld_ir*, and *inc_pc* to read memory from the present *PC* location, route it to *IR*, load it into *IR*, and increment *PC* for the next memory fetch. Depending on *op_code* bits, that are the controller inputs, the *Execute* state of the controller issues control signals for the execution of **lda**, **sta**, **add**, and **jmp** instructions. The *Decode* state is a simple wait state.

The next section discusses details of the controller signals and their role in execution of these instructions. As before, our processor description has a datapath and a control component. The controller is described using a state machine coding style. At the end, the description of our small example is completed by wiring datapath and controller in a top-level Verilog module.

2.7.3.3 Datapath HDL Description

Datapath components of Adding Machine are described by **always** and **assign** statements according to their functionalities described above. Afterward, these modules are instantiated into the datapath module. Figure 2.32 shows the Verilog code of the datapath. Structure and signal names in this description are according to those shown in Fig. 2.30.

2.7.3.4 Controller HDL Description

The controller code for our Adding Machine example is shown in Fig. 2.33. This code corresponds to the right-hand side control block in Fig. 2.29 which is shown in more details in Fig. 2.31.

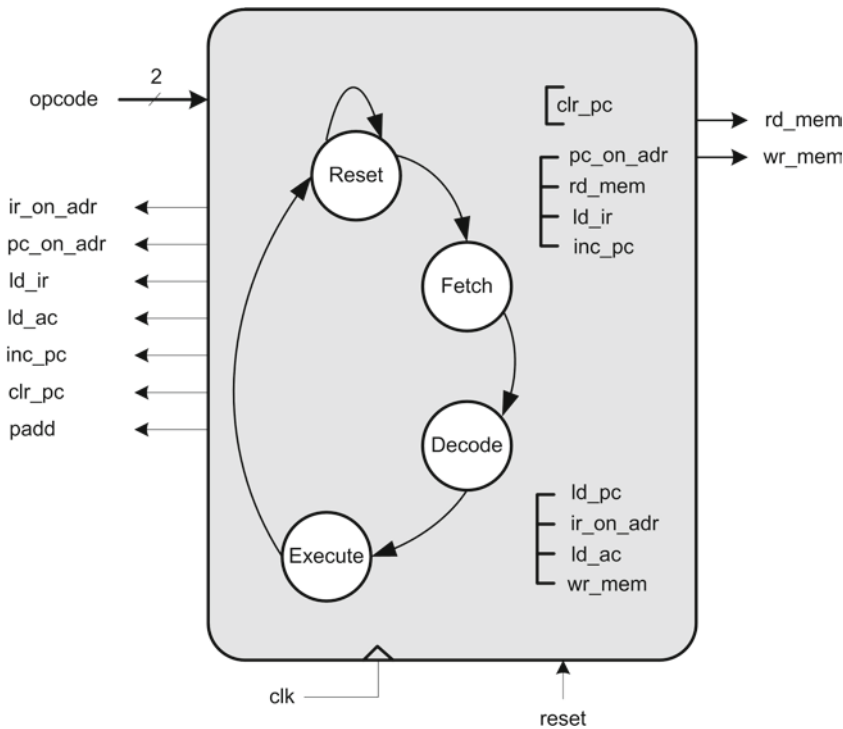


Fig. 2.31 Simple CPU Adding Machine multicycle controller

```

module DataPath ( clk, ir_on_adr, pc_on_adr, ld_ir, ld_ac, ld_pc, inc_pc,
                  clr_pc, pass_add, adr_bus, op_code, data_bus_in, data_bus_out);

    input  clk, ir_on_adr, pc_on_adr, ld_ir, ld_ac, ld_pc, inc_pc, clr_pc,
           pass_add;
    output [5:0] adr_bus;
    output [1:0] op_code;
    input  [7:0] data_bus_in;
    output [7:0] data_bus_out;

    wire [7:0] ir_out;
    wire [5:0] pc_out;
    wire [7:0] a_side;

    IR ir( data_bus_in, ld_ir, clk, ir_out );
    PC pc( ir_out[5:0], ld_pc, inc_pc, clr_pc, clk, pc_out );
    AC ac( data_bus_in, ld_ac, clk, a_side );
    ALU alu( a_side, {2'b00,ir_out[5:0]}, pass_add, data_bus_out );

    assign adr_bus = ir_on_adr ? ir_out[5:0] : pc_on_adr ? pc_out : 6'b0;
    assign op_code = ir_out[7:6];
endmodule

```

Fig. 2.32 Datapath HDL description

In addition to *clk* and *reset*, the controller has the *op_code* input that is driven by *IR* and comes to the controller from the *DataPath* module (see Fig. 2.30).

The sequencing of control states is implemented by a Huffman style Verilog code. In this style, an always block (*registering*) handles the assignment of values to *present_state*, and another **always**

```

`define Reset 2'b00
`define Fetch 2'b01
`define Decode 2'b10
`define Execute 2'b11

module Controller (reset, clk, op_code, rd_mem, wr_mem, ir_on_adr,
    pc_on_adr, ld_ir, ld_ac, ld_pc, inc_pc, clr_pc,
    pass_add );

    input reset, clk;
    input [1:0]op_code;
    output rd_mem, wr_mem, ir_on_adr, pc_on_adr, ld_ir, ld_ac, ld_pc;
    output inc_pc, clr_pc, pass_add;
    reg rd_mem, wr_mem, ir_on_adr, pc_on_adr, ld_ir, ld_ac;
    reg ld_pc, inc_pc, clr_pc, pass_add;
    reg [1:0] present_state, next_state;

    always @( posedge clk )begin : registering
        if (reset )
            present_state <= `Reset;
        else
            present_state <= next_state;
    end
    always @(present_state) begin : combinational
        rd_mem=1'b0; wr_mem=1'b0; ir_on_adr=1'b0; pc_on_adr=1'b0;
        ld_ir=1'b0; ld_ac=1'b0;
        ld_pc=1'b0; inc_pc=1'b0; clr_pc=1'b0; pass_add=1'b0;

        case( present_state )
            `Reset : begin
                next_state = `Fetch; clr_pc = 1'b1;
            end
            `Fetch : begin
                next_state = Decode ; pc_on_adr=1'b1; rd_mem=1'b1;
                ld_ir=1'b1; inc_pc=1;
            end
            Decode : begin
                next_state = `Execute;
            end
            `Execute: begin
                next_state = `Fetch;
                case( op_code )
                    2'b00: begin // lda
                        ir_on_adr=1'b1; rd_mem=1'b1; ld_ac=1'b1;
                    end
                    2'b01: begin // sta
                        ir_on_adr=1'b1; pass_add = 1'b0;
                        wr_mem=1'b1;
                    end
                    2'b10: ld_pc=1'b1; // jmp
                    2'b11: begin // add
                        pass_add=1'b1; ld_ac=1'b1;
                    end
                endcase
            end
        endcase
    end
endmodule

```

Fig. 2.33 Controller HDL description

statement (*combinational*) uses this register output as the input of a combinational logic determining *next_state*. This combinational block also sets values to control signals that are outputs of the controller.

In the body of the *combinational always* block, a **case** statement checks *present_state* against the states of the machine (*Reset*, *Fetch*, *Decode*, and *Execute*), and activates the proper control signals.

The *Reset* state activates *clr_pc* to clear *PC* and sets *Fetch* as the next state of the machine. In the *Fetch* state, *pc_on_adr*, *rd_mem*, *ld_ir*, and *inc_pc* become active, and *Decode* is set to become the next state of the machine. By activating *pc_on_adr* and *rd_mem*, the *PC* output goes on the memory address and a read operation is issued. Assuming the memory responds in the same clock, contents of memory at the *PC* address will be put on *data_bus_in*. This bus is connected to the input of *IR* and issuance of *ld_ir* loads its contents into this register. The next state of the controller is *Decode* that makes the new contents of *IR* available for the controller. In the *Execute* state, a newly fetched instruction in *IR* decides on control signals to issue to execute the instruction.

In the *Execute* state, *op_code* is used in a **case** expression to decide on control signals to issue depending on the opcode of the fetched instruction. The **case** alternatives in this statement are four *op_code* values of 00, 01, 10, and 11 that correspond to **lda**, **sta**, **jmp**, and **add** instructions.

For **lda**, *ir_on_adr*, *rd_mem*, and *ld_ac* are issued. These control signals cause the address from *IR* to be placed on the *adr_bus* address bus, memory read to take place and data from memory to be loaded into *AC*.

The controller executes the **sta** instruction by issuing *pass_add*, *ir_on_adr*, and *wr_mem*. As shown in Fig. 2.33, these signals take contents of *AC* to the input bus of the memory (i.e., *data_bus_out*), and *wr_mem* causes the writing into the memory to take place. Note that *pass_add* causes *AC* to pass through *ALU* unchanged. The **jmp** instruction is executed by enabling *PC* load input, which takes the jump address from *IR* (see Fig. 2.33).

The last instruction of this machine is **add**, for execution of which, *pass_add* and *ld_ac* are issued. This instruction adds data in the upper 6 bits of *IR* with *AC* and loads the result into *AC*.

2.7.3.5 The Complete HDL Design

The top-level module for our Adding Machine example is shown in Fig. 2.34. In the *CPU* module shown, *DataPath* and *Controller* modules are instantiated. Port connections of the *Controller*

```

module CPU( reset,clk,adr_bus,rd_mem,wr_mem,data_bus_in,data_bus_out );
input reset;
input clk;
input [7:0]data_bus_in;
output [5:0]adr_bus;
output rd_mem;
output wr_mem;
output[7:0]data_bus_out;
wire ir_on_adr, pc_on_adr, ld_ir, ld_ac, ld_pc, inc_pc, clr_pc, pass_add;
wire [1:0] op_code;

Controller cu ( reset, clk, op_code, rd_mem, wr_mem, ir_on_adr, pc_on_adr,
               ld_ir, ld_ac, ld_pc, inc_pc, clr_pc, pass_add );

DataPath dp ( clk, ir_on_adr, pc_on_adr, ld_ir, ld_ac, ld_pc, inc_pc,
             clr_pc, pass_add, adr_bus, op_code, data_bus_in, data_bus_out );

endmodule

```

Fig. 2.34 Adding Machine top-level module

include its output control signals, the *op_code* input from *DataPath* and the *reset* external input. Port connections of *DataPath* consist of *adr_bus* and *data_bus_in* and *data_bus_out* external busses, *op_code* output, and control signal inputs.

2.8 Testbench Techniques

The previous sections described Verilog for designing combinational and sequential circuits, as well as complete systems. This section discusses about testbenches and their role in simulation. However, the primary intention of this part is to show how testbench techniques could help us to develop test environments and virtual testers for digital circuit testing. This section shows how Verilog language constructs can be used for the application of data to a module under test, and how module responses can be displayed and checked.

A Verilog testbench is a Verilog module that instantiates a module under test (MUT), applies data to it and monitors its output. Because a testbench is in Verilog, it can go from one simulation environment to another. A module and its corresponding testbench form a simulation model in which MUT is tested regardless of what simulation environment is used.

Based on these considerations, testbenches could play a very important role in the development of test applications in HDL environments. Therefore, a test designer must understand testbenches and language constructs that are used for testing a design module. The basics of testbench techniques in Verilog HDL are discussed in this section, and more complete testbenches to develop test applications are illustrated in the next chapters.

2.8.1 Testbench Techniques

All that a testbench covers can be categorized in instantiating a module, applying generated or existing data to the inputs of the MUT, delay management, and then collecting the responses of the circuit and, if required, comparing them with the expected responses. Therefore, testbench techniques can be categorized in order to answer the following questions: 1) How is the data generated or provided, 2) How are the circuit responses getting reported, 3) What are data generation and response collection sensitive, and 4) What language constructs are to be used to manage the termination of a testbench?

Answers to the above questions are discussed in the rest of this section, and for preparing for the materials that follow Short answer for the above questions are given in the following.

1. The methods to provide data include *deterministic* – assigning a specific data to inputs, *arithmetic* – for example, using a counter to provide new data, *periodic* – toggling the value of a signal in certain periods, *random* – for example, using **\$random** task function of Verilog, and *Text IO* – reading data from a stored text file, e.g. using **\$fscanf** or **\$fread**.
2. To report the circuit responses, Verilog display utilities such, as **\$display** or **\$monitor** can be used. These tasks, show the results in the simulator's console. Another way is to use *Text IO* to record the responses in text files for future references, e.g., using **\$fdisplay** or **\$fwrite**.
3. It is important to decide on the conditions that test data are applied to a design under test, and conditions for collection of its responses. Various choices for such conditions are: a) End of a delay, which can be based on different time slots, equal time slots, or random amount of delay, and b) Change of a signal which is appropriate to make handshaking and synchronization between the testbench and the design under test.

4. While applying data and collecting responses, the duration of running a testbench must also be specified. The methods to manage the end time of a testbench include **\$stop**, **\$finish** or managing iterations using **repeat** or **for** construct.

Examples of the above items will be seen in the testbenches that are discussed in the following sections for testing combinational and sequential circuits.

2.8.2 A Simple Combinational Testbench

Developing a testbench for a combinational circuit is straightforward; however, selection of data and how much testing should be done depends on the MUT and its functionality. Previously, a simple ALU was described (Fig. 2.18) that we use here to test, and its header is repeated in Fig. 2.35 for reference. The *alu_4bit* module is a four function ALU. Data inputs are *a* and *b*, and its function input is *f*.

A testbench for *alu_4bit* is shown in Fig. 2.36. Variables corresponding to inputs and outputs of the MUT are declared in the testbench. Variables connecting to the inputs are declared as **reg** and outputs as **wire**. Instantiation of *alu_4bit*, shown in the testbench, associates local **regs** and **wires** with the ports of this module.

Variables that are associated with the inputs of *alu_4bit* have been given initial values when declared. Application of data to the *b* input is done in an **initial** statement. For the first 60 ns and every 20 ns, a new value is assigned to *b*, and after 20 ns the testbench finishes the simulation. This last 20 ns wait, allows effects of the last input change to be shown in the simulation run results.

Application of data to the *f* input of *alu_4bit* is done in an **always** statement. Starting with the initial value of 0, *f* is incremented by 1 every 23 ns. The **\$finish** statement in the **initial** block is reached at 80 ns. At this time, all active procedural blocks stop and simulation terminates. Simulation

```

module alu_4bit (input [3:0] a, b, input [1:0] f, output reg [3:0] y );

//...

endmodule

```

Fig. 2.35 alu_4bit module declaration

```

module test_alu_4bit;
  reg [3:0] a=4'b1011, b=4'b0110;
  reg [1:0] f=2'b00;
  wire [3:0] y;

  alu_4bit MUT( a, b, f, y);

  initial begin
    #20 b=4'b1011;
    #20 b=4'b1110;
    #20 b=4'b1110;
    #20 $finish;
  end
  always #23 f = f + 1;

endmodule

```

Fig. 2.36 Testbench for *alu_4bit*

control tasks are **\$stop** and **\$finish**. The first time the flow of a procedural block reaches such a task, simulation stops or finishes. A stopped simulation can be resumed, but a finished one cannot. In this example, the data generation for *b* is deterministic, and its data application condition is based on different time slots (we used 20 ns intervals). For the *f*input, data generation is arithmetic, and data application is based on equal time slots (periodic 23 ns).

2.8.3 A Simple Sequential Testbench

Test of sequential circuits involves synchronization of clock with other data inputs. We use the *residue5* module as an example here. As shown in the header of this circuit, repeated in Fig. 2.37 for reference, it has a clock input, a reset, data input, and output.

Figure 2.38 shows a testbench for the Residue-5 circuit. As before, variables corresponding to the ports of MUT are declared in the testbench. When the *residue5* module is instantiated, these variables are connected to its actual ports.

The **initial** block of this testbench generates a positive pulse on *rst* that begins at 13 ns and ends at 63 ns. The timing is so chosen to cover at least one positive clock edge so that the synchronous *rst* input can initialize the states of the Residue-5 circuit. The *d_in* data input begins with value X and is initialized to 2'b01 while *rst* is 1.

In addition to the **initial** block, *test_residue5* module includes two **always** blocks that generate data on *d_in* and *clk*. Clock is given a periodic signal that toggles every 11 ns. The Residue-5 *d_in* input is assigned a new value every 37 ns. In order to reduce the chance of changing several inputs at the same time, we usually use prime numbers for the timing of sequential circuit inputs.

```

module residue5(input clk, reset, input[1:0] x, output[2:0] out);
    reg[2:0] current;
    //...
endmodule

```

Fig. 2.37 *Residue-5* sequential circuit

```

module test_residue5;
    reg clk, rst;
    reg [1:0] d_in;
    wire [2:0] d_out;

    residue5 MUT ( clk, rst, d_in, d_out );

    initial begin
        clk=1'b0
    end
    initial begin
        #13 rst=1'b1;
        #19 d_in = 2'b01;
        #31 rst=0'b0;
        #330 $finish;
    end
    always #37 d_in = d_in+1;
    always #11 clk = ~clk;

endmodule

```

Fig. 2.38 A testbench for the *residue5* module

Instead of initializing **reg** variables when they are declared, we have used an **initial** block for this purpose. It is important to initialize variables, like the *clk* clock, for which their old values are used for determining their new values. If not done so, *clk* would start with value X and complementing it would never change its value. The **always** block shown generates a periodic signal with a period of 22 ns to provide a free running clock.

The waveform generated on *d_in* may or may not be able to test the whole functionality of this state machine. However, periods of *clk* and *d_in*, and the testbench duration can be changed to make this happen.

2.8.4 Limiting Data Sets

Instead of setting a simulation time limit, a testbench can put a limit on the number of data put on inputs of a MUT. This will also be able to stop simulation from running forever.

Figure 2.39 shows a testbench for our MUT that uses **\$random** to generate random data on the *x* input of the circuit. The **repeat** statements in the **initial** blocks shown cause *clock* to toggle 13 times every 5 ns, and *x* to receive a random data 10 times every 7 ns. Instead of a deterministic set of data to guarantee a deterministic test state, random data are used here. This strategy makes it easier to generate data, but due to unpredictable inputs, makes the analysis of circuit responses more difficult. In large circuits, using random data is more useful, and is usually more appropriate to set data inputs and not control signals. The testbench of Fig. 2.39 stops at 70 ns.

2.8.5 Synchronized Data and Response Handling

The previous examples of testbenches for MUT used independent timings for the clock and data. Where several sets of data are to be applied, synchronization of data with the system clock becomes difficult. Furthermore, changing the clock frequency would require changing the timing of all data inputs of the module being tested.

The testbench of Fig. 2.40 uses an event control statement to synchronize data applied to *x* with the clock that is generated in the testbench. The *clock* signal is generated in an **initial** statement using the **repeat** construct. An **always** statement is used for generation of random data on *x*. This loop waits for the positive edge of *clock* and 3 ns after the clock edge, and a new random data is generated for *x*. The stable data after the positive edge of the clock will be used by *residue5* on the

```

module test_residue5;
    reg reset=1, clock=0;
    reg [1:0] x;
    wire [2:0] z;

    residue5 MUT (clock, reset, x, z);

    initial #24 reset=1'b0;
    initial repeat (13) #5 clock=~clock;
    initial repeat (10) #7 x=$random;

endmodule

```

Fig. 2.39 Testbench using **repeat** to limit data sets

```

module test_residue5;
    reg  reset=1, clock=0;
    reg  [1:0] x;
    wire  [2:0] z;

    residue5 MUT ( clock, reset, x, z );

    initial #24 reset=0;
    initial repeat(13) #5 clock=~clock;
    always @(posedge clock) #3 x=$random;
    initial forever @(posedge clock) #1 $displayb(z);
    always @(z) $display("Output changes at %t to %b", $time, z);
    initial $monitor("New state is %d and occurs at %t", MUT.current, $time);
endmodule

```

Fig. 2.40 Synchronizing data with clock

next leading edge of the clock. This technique of data application guarantees that changing of data and clock do not coincide.

In this testbench, 1 ns after the positive edge of the clock, that is when the circuit output is supposed to have its new stable value, the *z* output is displayed using the **\$display** task. This method is appropriate for behavioral simulation, but when dealing with synthesized circuit which includes internal delays, calculating the exact time in which response is ready would not be easy and reliable. A more convenient way to display new output values is to wait for an event on the output *z*, which means that it has received a new value. This can be complemented by displaying the time of change using the **\$time**, task.

Using hierarchical naming, this testbench can be used for displaying internal variables and signals of *MUT*. The **initial** statement containing **\$monitor** is responsible for displaying *MUT*, *current*, which is the current state of *residue5* addressed by its hierarchical name. The **initial** statement starts **\$monitor** in the background. Display occurs when the task is started and when an event occurs on one of the variables in the task arguments. The **%b**, **%d**, and **%t** format specifications in this testbench cause the related signals to be reported as binary, decimal, and in time unit, respectively.

2.8.6 Random Time Intervals

We have shown how **\$random** can be used for generation of random data. The testbench we are discussing in this section uses random delays for assigning values to *x*.

Figure 2.41 shows a testbench for the Residue-5 circuit that uses **\$random** for its delay control. As shown, the *running initial* statement applies appropriate initial values to inputs of the *MUT*. In this procedural block, nonblocking assignments cause intra-assignment delay values to be regarded as absolute timing values. Then, the testbench waits for 13 complete clock pulses before it finishes the simulation. As shown, an **always** block concurrent with the *running* block continuously generates clock pulses of 5 ns duration.

Also concurrent with these blocks is another **always** block that generates random data on *t*, and uses *t* to delay the assignment of random values to *x*. This block generates data on the *x* input for as long as the **\$finish** statement in the *running* block is not reached. Assume that it is desirable to check if the state machine of *residue5* ever meets state *Three* or not. The last **always** block in this testbench waits on observing 2'b11 on the internal state of the Residue-5 circuit (the *current reg*), and if it is found, it will be reported.

```

module test_residue5;
    reg reset, clock;
    reg [1:0] x;
    wire [2:0] z;

    residue5 MUT ( clock, reset, x, z );

    initial begin :running
        clock = 1'b0; x = 1'b0;
        reset = 1'b1; reset = #7 1'b0;
        repeat (13) begin
            @( posedge clock );
            @( negedge clock );
        end
        #5;
        $finish;
    end

    always #5 clock=~clock;
    always begin
        t = $random;
        # (t) x=$random;
    end

    always begin
        wait (MUT.current == 2'b11);
        $display("state is 2'b11");
    end
endmodule

```

Fig. 2.41 Testbench using random time intervals

2.8.7 Text IO

Input and output from external files are discussed here. In VHDL, this is referred to as Text IO, and we use the same terminology here. The input side of Text IO means that instead of generating test data, a testbench can apply data to the MUT from a pre recorded text file. This is equivalent to a *stored vector* testing that is done by an ATE. In this book, using this type of providing data is very common.

Figure 2.42 shows a testbench that uses Text IO to read data and expected output of the MUT. Three file pointers *dataFile*, *responseFile*, and *reportFile* of type **integer** are declared and are assigned in the first **initial** block to three physical text files “*Res5.dat*,” “*Res5.rsp*,” and “*Res5.rpt*,” respectively. This assignment is performed by using the **\$fopen** task function. The second argument in these statements shows the mode of opening file, which could be “*r*” as read, “*w*” as write, and “*a*” as append.

The next **initial** block is responsible for reading data and the expected output from the related text files, managing the required delay and then collecting the responses of the MUT and comparing them with the expected values. All of the mentioned processes continue until the end of one or more of the input files is reached; this condition is checked with the **\$feof** task function in the condition part of the **while** statement.

The data reading from the input files can be done using **\$fscanf** or **\$fread**. In this case, **\$fscanf** is used. This task function has an integer return value which shows if the reading was successful or not. Therefore, variable *status* of type **integer** should be declared and used here. After reading and applying data from *dataFile* to *d_in*, and reading the expected response from *responseFile*, the testbench waits for the **posedge** of the clock to make sure that the input is affected, and the internal state of the circuit has been changed. Then after a very short delay, the expected and the actual responses of MUT

```

module test_residue5;
    reg  rst, clk;
    reg  [1:0] d_in;
    wire [2:0] d_out;
    reg  [2:0] expected_out;
    integer dataFile, responseFile, reportFile, status;

    residue5 MUT (clk, rst, d_in, d_out);

    initial begin
        clk = 0;
        rst = 1'b1; #7; rst = 1'b0;
        dataFile = $fopen("Res5.dat", "r");
        responseFile = $fopen("Res5.rsp", "r");
        reportFile = $fopen("Res5.rpt", "w");
    end

    always #5 clock=~clock;

    initial begin
        while((!$feof(dataFile) && (!$feof(responseFile)))
            begin
                status = $fscanf(dataFile, "%b\n", d_in);
                status = $fscanf(responseFile, "%b\n", expected_out);
                @posedge(clk);
                #1; if(expected_out == d_out)
                    $display("correct output = %d", d_out);
                else
                    $fdisplay(reportFile, "wrong output.. d_out= %b\t
                                expected_out = %b\n", d_out, expected_out);
            end
            #1;
            $finish;
        end

    endmodule

```

Fig. 2.42 Testbench using text IO

can be compared. Since the internal state of the circuit changes right at the **posedge** of the clock, this 1 ns delay guarantees that we are not looking at the previous state of the circuit. The correctness of results is reported on the console of the simulator using **\$display** or **\$monitor**, and in case that they are not equal, it can be reported in a text file using **\$fprintf**, **\$fwrite**, or **\$fdisplay**. Concurrent with the mentioned **initial** blocks, an **always** block generates a periodic clock with a 10 ns period.

2.8.8 Simulation Code Coverage

A good testbench that can verify the correctness of a design should guarantee that is able to exercise most of the design under test and especially its critical parts. The percentage of the statements, blocks, paths, etc., in a design that are covered using a testbench is the *code coverage* of that testbench. Most of the simulation environments provide tools to estimate the code coverage for testbenches. During the compilation part in an HDL simulator, the kind of required code coverage can be specified; then, the simulator calculates the specified type of code coverage for the instantiated design. If the resulted code coverage is less than expected, it can be decided that the testbench is not a good quality testbench. The parameters that most of the HDL simulators support for code coverage include *statement coverage*, *condition coverage*, *block coverage*, and *branch coverage*.

Code coverage matrices measure how much of the design a testbench covers. On the other hand, if we want to estimate how much of the possible design faults this testbench covers, we must apply this testbench to the post-synthesis model of the design, and simulate it. Reports generated by this simulation are called fault coverage. We may think, or hope, that high-level code coverage and low-level fault coverage somehow correspond. Although this correspondence is very weak, but using code coverage we can have a sense of how good a testbench would be for gate-level fault simulation. The advantage of using high-level simulation is that it is much faster than the gate-level simulation. Therefore, the fast behavioral testbench can be performed as an estimation of a good testbench; it can get matured in this level of simulation at a lower cost and then get adjusted for covering more faults.

As an example, Figs. 2.43 and 2.44 show the Verilog code of a *Comparator* and its block diagram on which various code coverages are depicted.

```

module Comparator (input a, b, output a_gtoreq_b, a_lt_b);

    always @ ( a, b ) begin
        if ( a < b ) begin
            a_gtoreq_b = 0;
            a_lt_b = 1;
        end
        else begin
            a_gtoreq_b = 1;
            a_lt_b = 0;
        end
    end
endmodule

```

Fig. 2.43 Behavioral code of a comparator

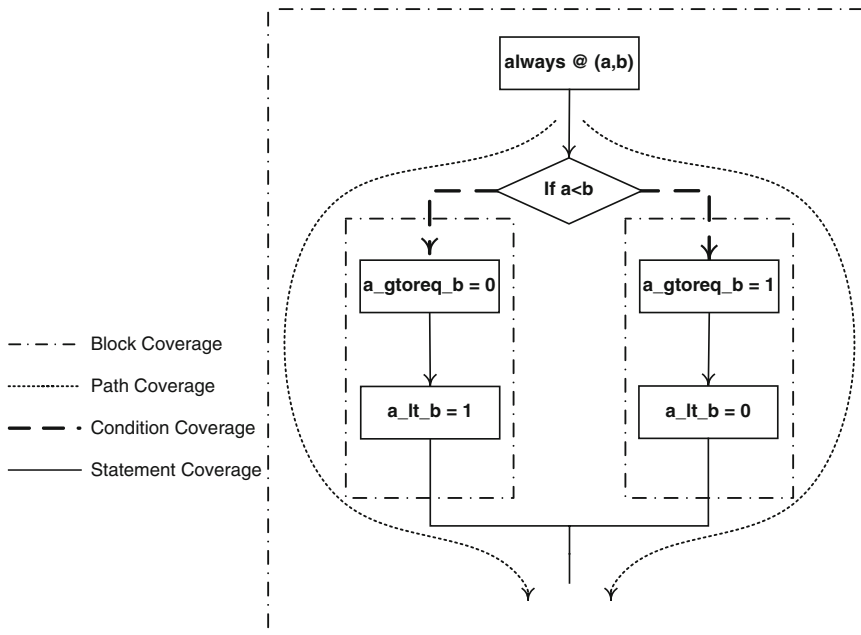


Fig. 2.44 Types of code coverage for a comparator

In these figures, output *a_gtoreq_b* becomes 1 when the first input is greater than or equal to the second input, and *a_lt_b* checks if *a* is less than *b* or not. Figure 2.44 shows the block diagram related to this code, and various types of code coverages are specified with different line styles.

Condition coverage means how many of the edges of all the conditions in the code will be visited with the related testbench. For example in Fig. 2.44, if the testbench is such that *a* is always less than *b*, then the left-hand side branch of the condition is never covered.

In this block diagram, *statements* are identified using solid lines. *Statement coverage* specifies how many of these statements can be examined by the testbench.

Two curved dotted lines on the sides of the block diagram show two *paths* in this code. *Path coverage* shows how many paths in a design are covered using a certain testbench. For example, in a **case** statement, the code branches out to many paths and they converge at the same place, and there might be some paths from the divergence to the convergence point that have not been examined.

Finally, the dash-and-dot lines in Fig. 2.44 represent the *blocks* of the code in which their coverage can be calculated using the *block coverage* option of the simulator. To this point, we have discussed the HDL techniques to develop testbenches useful for HDL design. However, as mentioned at the beginning of this chapter, the main objective of using HDLs and testbenches in this book is utilizing their facilities to implement existing test applications and developing new ones. As mentioned in Sect. 2.2, an HDL environment can provide utilities to develop fault simulation, test generation, DFT evaluation and configuration, and various other test applications. However, some facilities are required for test purposes that Verilog HDL basic constructs do not provide. For example, Verilog is not able to model a defective wire without making changes in the components of the original design [9]. In addition, a number of test utilities such as fault compilation and testability measurements need to explore the gate-level netlist of a design at a reasonable cost. In the standard Verilog language, this cannot easily be done since it does not have mechanisms for creating software-like structures. Fortunately, these drawbacks of HDL environment are compensated for by using the PLI of Verilog. PLI also has other capabilities that facilitate integration of design and test [10]. The following section briefly introduces PLI and its features and illustrates how it can be useful for providing a convenient environment for test application development.

2.9 PLI Basics

Procedural language interface PLI provides a library of C language functions that can directly access data within an instantiated Verilog HDL data structure [11] and provides mechanisms to invoke C or C++ functions from a Verilog testbench. Therefore, not only the design core and its testbench can be developed in a uniform programming environment, but also all the facilities of software programming (such as complex data structures and utilization of functions) become available by the use of PLI. A function invoked in Verilog code is called a system call. An example of a built-in system call is **\$display**, **\$stop**, **\$random**, which were introduced in the testbench section above. PLI allows the user to create custom system calls, for tasks that the standard Verilog language does not support.

Verilog PLI has been in use since the mid-1980s. This standard comprises of three primary generations of the Verilog PLI: a) *Task/function* routines (*tf*), b) *Access* routines (*acc*), and c) *VPI* routines. The *tf* and *acc* libraries construct the PLI 1.0 standard, which is vast and old. The next set of routines, which was introduced with the latest release of Verilog 2001 is called *vpi* routines. These are small and down-to-point PLI routines that make the new version, PLI 2.0.

Test applications in this book are developed using Access (*acc*) routines. The *acc* routines are C programming language functions that start with **acc_**. These routines provide direct access to a Verilog HDL structural description. Using the *acc* routines, we can access and modify information, such as delay and logic values on various objects in a Verilog HDL description. More information about these routines can be found in the next subsection.

2.9.1 Access Routines

Access routines are C programming language routines that provide procedural access to information within Verilog-HDL. Access routines perform one of two operations, read or write. Using read operations, certain data and information can be obtained about particular objects in the circuit directly from its internal data structure. The objects that access routines can perform read operations for, included module instances, module ports, module paths, intermodule paths, top-level modules, primitive instances, primitive terminals, **nets**, **regs**, **parameters**, **specparams**, timing checks, named events, integer, and real and time variables. Write operations replace new data or information for objects in the circuit by directly changing the related variables into the internal data structures. Access routines can write to intermodule paths, module paths, primitive instances, timing checks, register logic values, and sequential UDP logic values.

According to the operation performed by access routines, they are classified into six categories: 1) *Fetch* routines return a variety of information about different objects in the design hierarchy, 2) *Handle* routines return handles – the pointer to an object in the data structure, to a variety of objects in the design hierarchy, 3) *Modify* routines alter the values of a variety of objects in the design hierarchy, 4) *Next* routines when used inside a loop construct can find each object of a given type that is related to a particular reference object in the design hierarchy; for example, ports of a module, the instantiated modules within it – which are called its children, or the module which instantiated this module – which is called its parent, 5) *Utility* routines perform a variety of operations, such as initializing and configuring the access routine environment, and 6) *Vcl* or Value Change Link (VCL) allows a PLI application to monitor the value changes of selected objects. VCL can monitor value changes for events, scalar and vector registers, scalar nets, bit-selects of expanded vector nets, and unexpanded vector nets. On the other hand, VCL cannot extract information about the following objects: bit-selects of unexpanded vector nets or registers, part-selects, and memories.

2.9.2 Steps for HDL/PLI Implementation

Figure 2.45 shows the general view of implementing and running test programs in a mixed HDL/PLI environment. All test applications in this book are implemented based on this block diagram.

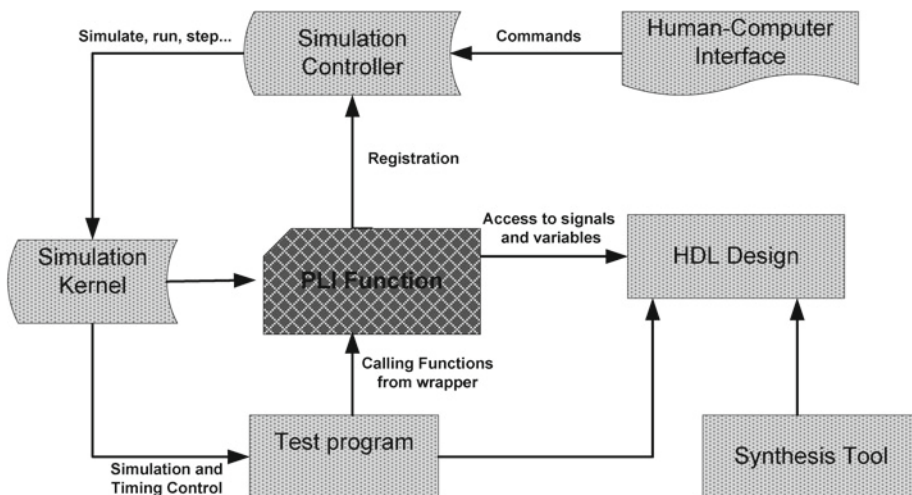


Fig. 2.45 The general view of running test programs in mixed HDL/PLI environment

The main part of this block diagram is the PLI function; the PLI functions should be written and compiled using a C compiler. A number of examples for writing PLI functions are given in the following. After completing the C code for the PLI function using the *acc* routines, the provided function must register its system tasks and functions with the HDL simulator. Registering each system task and function must be performed by filling the entries of an array of *s_tfcell* structures shown in Fig. 2.46. The resulted *struct* must take place at the end of the C code which implements the PLI function.

To make these steps more clear, a simple PLI function (perhaps the simplest possible) is shown in Fig. 2.47. This is just a simple PLI call for printing a message. The first line of this code includes the **veriusers.h** header to be able to use the **io_printf** function. **veriusers.h** and **acc_user.h** are two header files in the directory of the HDL simulator installation path, and should be included in the C code of the PLI function to have access to the routines of these libraries.

The last part of this C code performs the registration of these PLI functions with the HDL simulator. This code varies from one simulator to another, and we have shown this for Mentor Graphics's *ModelSim* simulator. As depicted in this code, there must be one entry for each declared function in this code, and the last entry of the **veriusertfs** must always be zero. The first field in each entry shows the type of the declared function, which for common uses we usually set it as **usertask**. This value means that the registered task does not return any value. The last field declares the name that the PLI function will be invoked with in the HDL testbench (notice the \$ character at the beginning of these names). The fifth field is the name of the C function that describes the PLI function.

```
typedef struct t_tfcell
{
    PLI_INT16 type;      /* USERTASK, USERFUNCTION, or USERREALFUNCTION */
    PLI_INT16 data;      /* passed as data argument of callback function */
    p_tffn checktf;      /* argument checking callback function */
    p_tffn sizetf;       /* function return size callback function */
    p_tffn calltf;       /* task or function call callback function */
    p_tffn mistf;        /* miscellaneous reason callback function */
    char * tfname;       /* name of system task or function */
}

```

Fig. 2.46 *t_tfcell* struct for registering the PLI function with the HDL simulator

```
#include "..\\HDLsimulatorInstallationPath\\include\\veriusers.h"
static PLI_INT32 Start()
{
    io_printf("Starting...\\n");
    return 0;
}
static PLI_INT32 End()
{
    io_printf("Ending...\\n");
    return 0;
}

s_tfcell veriusertfs[] =
{
    {usertask, 0, 0, 0, Start, 0, "$printStart"},
    {usertask, 0, 0, 0, End, 0, "$printEnd"},
    {0} /*last entry must be 0 */
};

```

Fig. 2.47 PLI coding and registration for a very simple function

```

module verilog_test();

initial
    $printStart();
    // . . .
    // . . .
    $printEnd();
    $stop();
endmodule

```

Fig. 2.48 PLI function call in HDL testbench

After providing this C code and compiling it with the C compiler, it must be built to generate a Dynamic Linked Library (*.dll*) file. When the C part is done, we get to the HDL simulator part. The resulted *.dll* file must be placed in the working directory of the HDL simulator. For invoking the prepared PLI function, the pseudo code in Fig. 2.48 can be used.

In order to simulate this testbench in the presence of the PLI *.dll* in ModelSim simulation environment, the following command must be performed in the simulator console.

```
vsim -c -pli dllFileName TestbenchName
```

In this line, **vsim** is the simulation command, **-c** is for the command mode, **-pli** means in the presence of the *.dll* file the name of which appears next, and finally the name of the top-level module or the testbench must be declared. In order to link more than one PLI *.dll* file to the HDL project, the following command should be used.

```
vsim -c -pli dllFileName_1 -pli dllFileName_2 ... -pli dllFileName_n TestbenchName
```

By running this command, the simulation of the testbench and designs added to the project is done, and it can be run like any other normal HDL project. As a result of running this testbench, the following lines will be printed on the simulator console.

```

Starting...
Ending...

```

In the next subsection, the implementation of fault injection and removal as more complex PLI functions and also a very important part of test applications are discussed.

2.9.3 Fault Injection in the HDL/PLI Environment

The most important utilities for implementing most of test algorithms are fault injection (FI) and fault removal (FR) functions. As mentioned, PLI provides mechanisms for reading and writing **net** and **reg** values. Therefore, we can force and release values in the data structures corresponding to **nets**, which give us the capabilities for FI and FR on and from circuit lines. In PLI, a **handle** is a pointer to a specific object in the design hierarchy. **handles** give information about a unique instance of a special object to *acc* routines. They contain useful information such as how and where we can find data about the object. For reading and writing information about an object, most *acc* routines require a handle argument. For each input argument of a PLI function, a variable of type **handle** will be used.

The FI and FR processes are done simply by using the **acc_set_value** PLI routine that sets the desired value on the target wire or removes the value from it. In order to implement PLI *InjectFault* and *RemoveFault*, there are two **structs** named **s_setval_value** and **s_setval_delay**, for which several fields must be set. However, the most important fields that need to be mentioned are the **model** field in **acc_setval_delay** and the **value** field of **acc_setval_value**. Figure 2.49 depicts these two **structs** for one of the input ports of an AND gate.

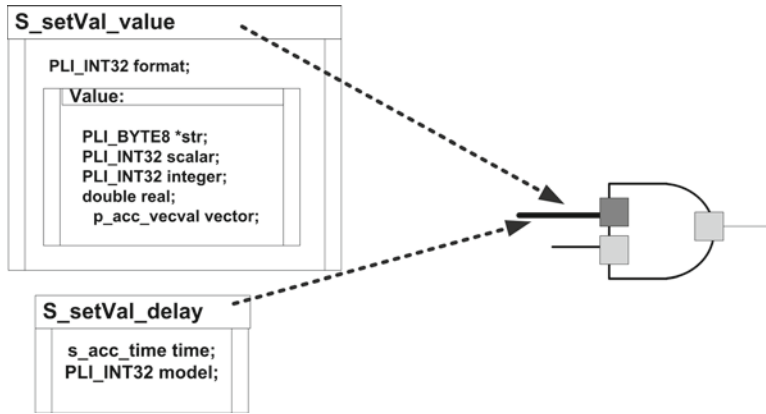


Fig. 2.49 PLI structures for stuck-at fault injection and fault removal

```

static PLI_INT32 injectFault ()
{
    arg1 = acc_handle_by_name ((char*) acc_fetch_tfarg_str(1), null);
    arg2 = acc_handle_tfarg(2);

    //get value of arg2 into value_of_arg2
    value_of_arg2.format = accScalarVal;
    acc_fetch_value (arg2, "%", &value_of_arg2);

    //prepare a data
    value_for_arg1.format = accScalarVal;
    value_for_arg1.value.scalar = Value_for_arg3.value.scalar;

    //prepare delay mode
    delay_of_arg1.model = accForceFlag;
    delay_of_arg1.time.type = accSimTime;
    delay_of_arg1.time.low = 0;
    delay_of_arg1.time.high = 0;

    //Put it in arg1
    acc_set_value (arg1, &value_for_arg1, &delay_for_arg1);

    acc_close();
    return 0;
}

```

Fig. 2.50 Fault injection PLI code

In FI, the model field must be defined as **accForceFlag**. This means that the desired value will be forced on the wire until it is removed by calling a PLI function for FR. During this time, the wire will not take values assigned to it by the Verilog simulator. The C code for the PLI FI is shown in Fig. 2.50.

The *removeFault* function sets the **s_setval_delay** model field to **accReleaseFlag**. Once this is done, values coming from HDL simulator will again appear on the wire. In other words, by putting the desired fault value on a variable of type **s_setvalue_value** and setting the model field of a variable of type **s_setvalue_delay** to **accForceFlag** or **accReleaseFlag**, FI and FR can be achieved. Figure 2.51 shows that the faulty value of the selected wire is applied by the PLI **\$InjectFault** function. Only after calling **\$RemoveFault** for that wire, it will accept the normal values, propagated to it by the HDL testbench.

Figures 2.52 and 2.53, respectively illustrate the usage of inject and remove fault on the wires of a full adder and the resulted waveform.

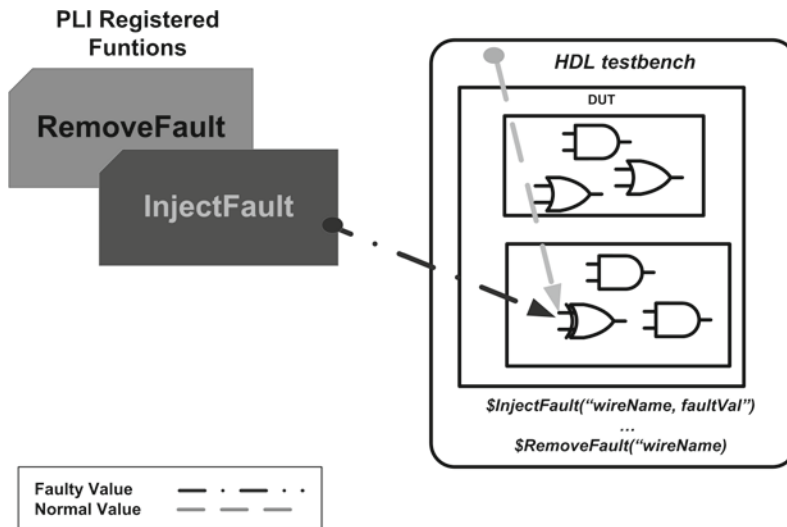


Fig. 2.51 Stuck-at fault injection and fault removal mechanism

```

module testbench();
reg a, b, cin;
wire sum_f, sum_g;
wire co_f, co_g;

FA FA_golden(a, b, cin, sum_g, co_g);
FA FA_faultable(a, b, cin, sum_f, co_f);

initial begin
    #20;
    $InjectFault("testbench.FA_faultable.s", 1'b0);
    repeat(10) begin
        #150;
        {a,b,cin} = $random();
    end
    $RemoveFault("testbench.FA_faultable.s");
    repeat(10) begin
        #150;
        {a,b,cin} = $random();
    end
    $stop;
end
endmodule

```

Fig. 2.52 Fault injection and removal for a full adder – testbench

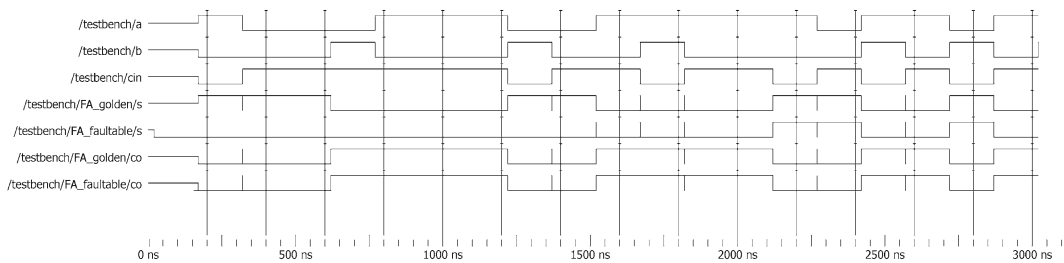


Fig. 2.53 Fault injection and removal for a full adder – waveform

In the testbench of Fig. 2.52 after 20 ns, the *FA_faultable.s* which is the *sum* port in this instance of the full adder, is stuck to 0 utilizing the PLI **\$InjectFault** function and stays in this state for 1,500 ns. The waveform shows that while *FA_golden.s* is obtaining the related output values during this period, the result of adding is not reflected on the *FA_faultable.s*, and its value is always 0 until the simulation time of 1,520 ns. At this time, the PLI **\$RemoveFault** function, removes the injected fault and from this moment to the end of simulation both faultable and golden instances of the full adder obtain the same values on their *sum* port.

2.10 Summary

In this chapter, the basics of Verilog HDL design and testbench techniques and its PLI are discussed. The overall guidelines to use this environment for design and test of digital circuits are shown and developing test applications in this environment is expressed by implementing the FI and FR utilities. All mentioned concepts of this chapter are used in the rest of this book to describe and enhance test techniques.

References

1. The International Technology Roadmap for Semiconductors (ITRS) website. (2007) [Online]. Available: <http://www.itrs.net/>
2. Ungar LY, Ambler T (2007) "Economics of Built-In Self-Test," *ITC*
3. IEEE Std 1364-2001, IEEE Standard Verilog Language Reference Manual, SH94921-TBR (print) SS94921-TBR (electronic), ISBN 0-7381-2827-9 (print and electronic), 2001
4. Stephen B, Zvonko V (2002) *Fundamentals of digital Logic with Verilog design*, McGraw-Hill; ISBN: 0-07-283878-7
5. Zainalabedin N (2006) *Verilog digital system design: RT level synthesis, testbench, and verification*, McGraw Hill, ISBN: 0-07-144564-1
6. Zainalabedin N (2007) *Embedded core design with FPGAs*, McGraw Hill, ISBN: 978-0071474818
7. Neil H.E. Weste and David Harris, *CMOS VLSI Design: A Circuits and Systems Perspective (3rd Edition)*, Addison Wesley; 3rd edition (May 11, 2004), ISBN: 0321149017
8. Patterson DA, Hennessy JL, Ashenden PJ, Larus JR, Sorin DJ *Computer Organization and Design: The Hardware/Software Interface, Third Edition*, Morgan Kaufmann; 3 edition (August 2, 2004), ISBN: 1558606041
9. Hesscot CJ, Ness DC, Lilja DJ (2005) "A methodology for stochastic fault simulation in vlsi processor architectures," *MoBs*
10. Riahi PA, Navabi Z, Lombardi F (2005) "Simulating Faults of Combinational IP Core-based SOC's in a PLI Environment," *DFT*
11. IEEE Std 1364-2001, IEEE Standard Procedural Language Interface Reference Manual, clause 20 through clause 25