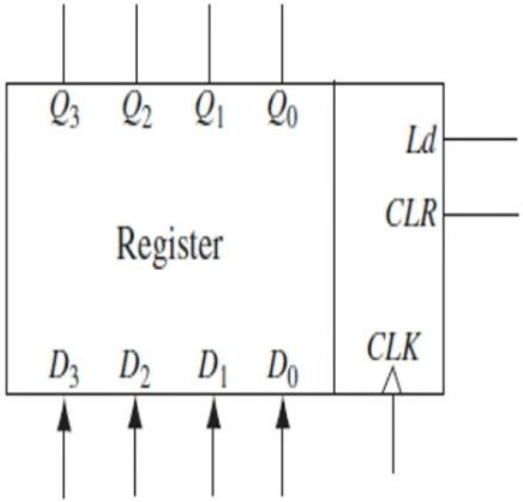
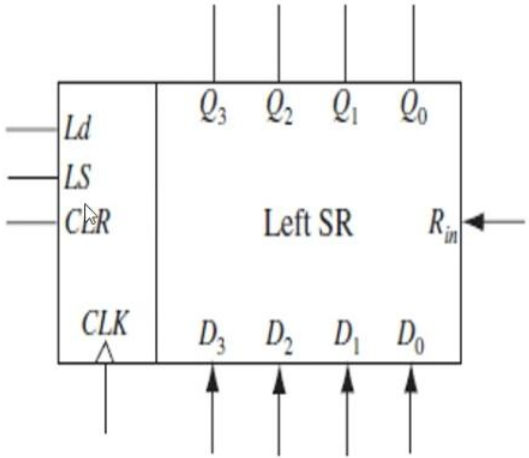
Register with Synchronous Clear and Load



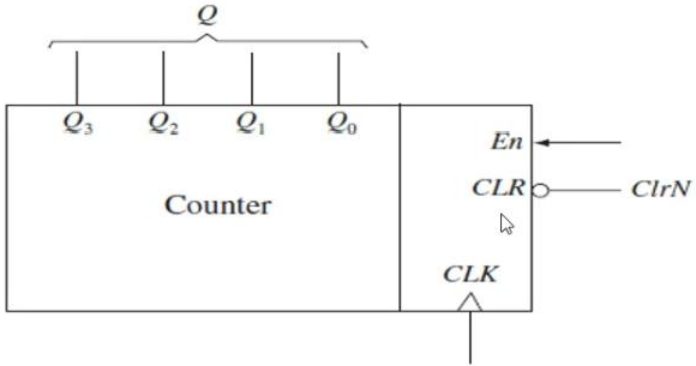

```
always @ (posedge CLK)
begin
 if (CLR)        Q <= 4'b0000;
 else if (Ld)    Q <= D;
end
```

Left shift register with Synchronous Clear and load



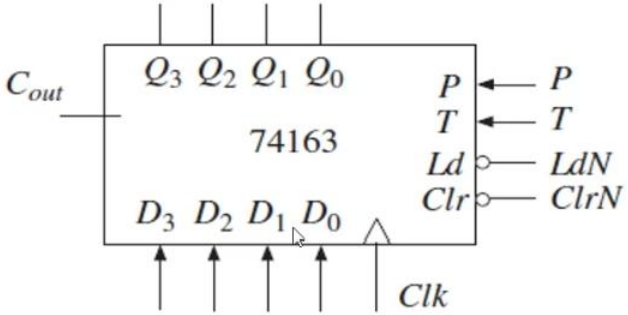

```
always @ (posedge CLK)
begin
 if (CLR)        Q <= 4'b0000;
 else if (Ld)    Q <= D;
 else if (LS)    Q <= {Q[2:0], Rin};
end
```

Verilog Code for a Simple Synchronous Counter



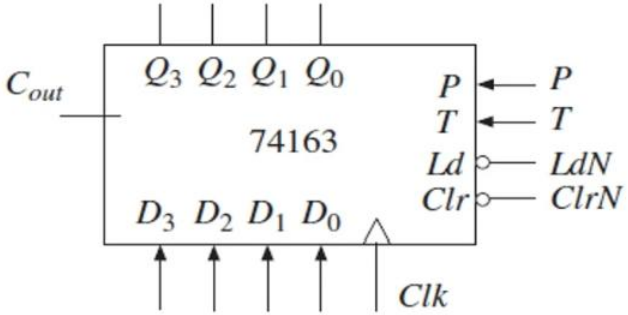

```
reg Q[3:0];

always @ (posedge CLK)
begin
 if (~ClrN)     Q <= 4'b0000;
 else if (En)   Q <= Q + 1;
end
```

4 bit fully synchronous counter



Control Signals			Next State				
$ClrN$	LdN	PT	Q_3^+	Q_2^+	Q_1^+	Q_0^+	
0	X	X	0	0	0	0	(clear)
1	0	X	D_3	D_2	D_1	D_0	(parallel load)
1	1	0	Q_3	Q_2	Q_1	Q_0	(no change)
1	1	1	present state + 1				(increment count)

4 bit fully synchronous counter



Control Signals			Next State				
$ClrN$	LdN	PT	Q_3^+	Q_2^+	Q_1^+	Q_0^+	
0	X	X	0	0	0	0	(clear)
1	0	X	D_3	D_2	D_1	D_0	(parallel load)
1	1	0	Q_3	Q_2	Q_1	Q_0	(no change)
1	1	1	present state + 1				(increment count)

This counter has four control inputs—*ClrN*, *LdN*, *P*, and *T*. Both *P* and *T* are used to enable the counting function. While *P* is an actual enable signal to the 4-bit generic counter, *T* is used for a carry connection signal when cascading multiple counters. Operation of the counter is as follows:

1. If $ClrN = 0$, all flip-flops are set to 0 following the rising clock edge.
2. If $ClrN = 1$ and $LdN = 0$, the *D* inputs are transferred (loaded) in parallel to the flip-flops following the rising clock edge.
3. If $ClrN = LdN = 1$ and $P = T = 1$, the count is enabled and the counter state will be incremented by 1 following the rising clock edge.

If $T = 1$, the counter generates a carry (C_{out}) in state 15; consequently

$$C_{out} = Q_3 Q_2 Q_1 Q_0 T$$

You

```
// 74163 FULLY SYNCHRONOUS COUNTER
module c74163(LdN, ClrN, P, T, Clk, D, Cout, Qout);
input          LdN;
input          ClrN;
input          P;
input          T;
input          Clk;
input [3:0]    D;
output         Cout;
output [3:0]   Qout;

reg [3:0]      Q;

assign Qout = Q;
assign Cout = Q[3] & Q[2] & Q[1] & Q[0] & T;

always @(posedge Clk)
begin
  if (~ClrN)          Q <= 4'b0000;
  else if (~LdN)      Q <= D;
  else if (P & T)     Q <= Q + 1;
end

endmodule
```

Avoiding Unwanted Latches

Verilog signals retain their current values until they are changed. This can result in the creation of unwanted latches when the Verilog code is synthesized. For example, in a combinational always block created using the statements

```
always @ (Sel or I0 or I1 or I2)
begin
 if      (Sel == 2'b00)  F = I0;
 else if (Sel == 2'b01)  F = I1;
 else if (Sel == 2'b10)  F = I2;
end
```

there would be latches to hold the value of *F* when Sel changes to `2'b11`. Someone probably wrote this code intending a MUX. Circuits with latches are not combinational hardware any more. The latch creates a variety of timing problems and unexpected behavior. Instead of being 1 bit wide, if F is 8 bits wide, eight latches would be created.

One can avoid unwanted latches by assigning a value to combinational signals in every possible execution path in an always block intended to create combinational hardware. Hence one should include an **else** clause in every **if** statement or explicitly include all possible cases of the inputs. For example, the code

```
always @ (Sel or I0 or I1 or I2 or I3)
begin
 if        (Sel == 2'b00)  F = I0;
 else if   (Sel == 2'b01)  F = I1;
 else if   (Sel == 2'b10)  F = I2;
 else if   (Sel == 2'b11)  F = 0;
end
```

would not create any latches but would create a MUX. For **if** then **else** statements and **case** statements, it is important to have all cases specified.

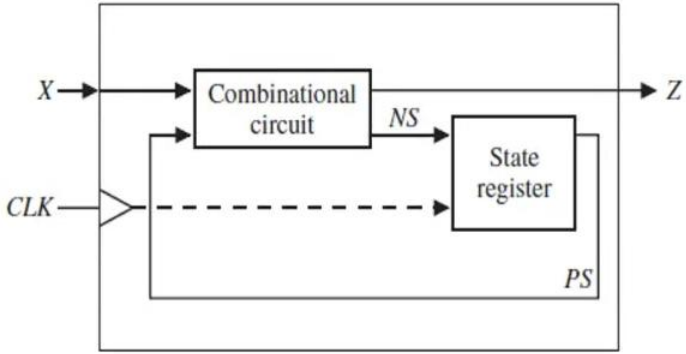
Another method to avoid latches is by initializing at the beginning of the always statement. For instance, the following code is latch free, even though only three of the four possible cases are specified.

```
always @ (Sel or I0 or I1 or I2 or I3)
F = 0;
begin
 if        (Sel == 2'b00)  F = I0;
 else if   (Sel == 2'b01)  F = I1;
 else if   (Sel == 2'b10)  F = I2;
end
```

2.15.1 Modeling a Sequential Machine

In this section, we discuss several ways of writing Verilog descriptions for sequential machines. Let us assume that we have to write a **behavioral** model for a Mealy sequential circuit represented by the state table in Figure 2-51 (one may note that this is the BCD to Excess-3 code converter designed in Chapter 1). A block diagram of this state machine is also shown in Figure 2-51. This view of the circuit can be used to write its entity description. Please note that the current state and next state are not visible externally.

	NS		*Z*	
PS	$X = 0$	$X = 1$	$X = 0$	$X = 1$
S_0	S_1	S_2	1	0
S_1	S_3	S_4	1	0
S_2	S_4	S_4	0	1
S_3	S_5	S_5	0	1
S_4	S_5	S_6	1	0
S_5	S_0	S_0	0	1
S_6	S_0	—	1	—



The following are important coding practices while writing synthesizable Verilog for sequential hardware:

(a) Use an edge-triggered clock in the sensitivity list using the **posedge** or **negedge** keywords.
(b) Use non-blocking assignments—that is, "<=" inside always blocks although it is possible to get sequential hardware by certain uses of the blocking "=" operator.
(c) Do not mix blocking and non-blocking statements in an always block.
(d) Do not make assignments to the same variable from more than one always block. This is not a compile-time error and hence may go unnoticed.
(e) Avoid unwanted latches by assigning a value to combinational output signals in every possible execution path in the always block. This can be done by

i. including else clauses for if statements,
ii. specifying all cases for case statements or have a `default` clause at the end, or
iii. unconditionally assigning default values to all combinational output signals at the beginning of the always block.

There are several ways to model this sequential machine. One approach would be to use two **always** blocks to represent the two parts of the circuit. One **always** block models the combinational part of the circuit and generates the next state information and outputs. The other **always** block models the state register and updates the state at the appropriate edge of the clock. Figure 2-52 illustrates such a model for this Mealy machine. The first always block represents the combinational circuit. At the behavioral level, we will represent the state and next state of the circuit by integer signals initialized to 0. Please remember that this initialization is meaningful only for simulations. Since the circuit outputs, *Z* and *Nextstate*, can change when either the *State* or *X* changes, the sensitivity list includes both *State* and *X*. The case statement tests the value of *State*, and depending on the value of *X*, *Z* and *Nextstate* are assigned new values. The second always block represents the state register. Whenever the rising edge of the clock occurs, *State* is updated to the value of *Nextstate*, so *CLK* appears in the sensitivity list. The second always block will simulate correctly if written as

```
always @ (CLK)          // State Register
begin                   // rising edge of clock (simulation)
  State <= Nextstate;
end
```

but in order to synthesize with edge-triggered flip-flops, the `event expression` (`posedge`) or `negedge` must be used, as in

```
always @ (posedge CLK)          // State Register
begin                           // (synthesis)
  State <= Nextstate;           // rising edge of clock
end
```

You

Constants

Verilog provides constants in addition to variables and nets. Verilog provides three different ways to define constant values, one of which is using `**define** as in

```
`define    constant_name   constant_value
```

The `define is one of the compiler directives in Verilog and is used to define a number or an expression for a meaningful string. The ` in `define is called *grave accent* (ASCII 0x60). It is different from the character (‘), which is the *apostrophe* character (ASCII 0x27). More on compiler directives can be found in Chapter 8.

The `define compiler directive replaces 'constant_name with constant_value. For example:

```
`define wordsize 16
 reg [1:`wordsize] data;
```

causes the string wordsize to be replaced by 16. It then shows how data is declared to be a reg of width wordsize.

Another method to create constants is to use the parameter keyword as follows:

```
parameter constant_name = constant_value;
```

You

For example,

```
parameter msb = 15; // defines msb as a constant value 15
parameter [31:0] decim = 1'b1; // value converted to 32 bits
```

Another method to make constants is using `localparam`.

```
localparam constant_name = constant_value;
```

The localparam is similar to the parameter, but it cannot be directly changed. The localparam can be used to define constants that should not be changed.

Verilog can define constant values in a module using the parameter. The parameter can be used to customize the module instances. Typical uses of parameters are to specify delays and width of variables.

For example,

```
parameter msb = 15; // defines msb as a constant value 15
parameter [31:0] decim = 1'b1; // value converted to 32 bits
```

Another method to make constants is using `localparam`.

```
localparam constant_name = constant_value;
```

The localparam is similar to the parameter, but it cannot be directly changed. The localparam can be used to define constants that should not be changed.

Verilog can define constant values in a module using the parameter. The parameter can be used to customize the module instances. Typical uses of parameters are to specify delays and width of variables.

Arrays

A key feature of VLSI circuits is the repeated use of similar structures. Arrays in Verilog can be used while modeling the repetition. Digital systems often use memory arrays. Verilog arrays can be used to create memory arrays and specify the values to be stored in these arrays. In order to use an array in Verilog, we must declare the array upper and lower bound. There are two positions to declare the array bounds:

In one option, the array bounds are declared between the variable type (`reg or net`) and the variable name, and the array bound means the number of bits for the declared variable. If the array bound is defined as [7:0] as shown in the following example,

```
reg    [7:0]    eight_bit_register;
```

the register variable `eight_bit_register` can store one byte (eight bits) of information. The 8-bit register can be initialized to hold the value `00000001` using the following statement:

```
eight_bit_register = 8'b00000001;
```

As a second option, array bounds can be declared after the name of the array. In the declaration that follows, `rega` is an array of n 1-bit registers while `regb` is a single n-bit register.

```
reg  rega [1:n]; // This is an array of n 1-bit registers
reg [1:n] regb; // This is an n-bit register
```

We can define multiple 8-bit registers in one array declaration. In this case, additional upper and lower bound(s) must be declared after the name of the array. In the example that follows, 16 registers are declared; each register can store one-byte (8-bit) vector information.

```
reg  [7:0]  eight_bit_register_array [15:0];
```

The foregoing declaration means that each of the 16 variables in the array can have 8-bit vector information. This array can be initialized as follows:

```
eight_bit_register_array[15] = 8'b00001100;
eight_bit_register_array[14] = 8'b00000000;

                . . . . . . . .
eight_bit_register_array[1]  = 8'b11001100;
eight_bit_register_array[0]  = 8'b00010001;
```

You

Arrays can be created of various data types. Arrays of wires and integers can be declared as follows:

```
wire wire_array[5:0]; // declares an array of 6 wires
integer inta[1:64]; // declares an array of 64 integer values
```

Matrices

Multidimensional array types may also be defined with two or more dimensions. The following example defines a 2-dimensional array variable in an initial statement, which is a matrix of integers with four rows and three columns with 8-bit elements:

```
reg  [7:0]  matrixA [0:3][0:2] = { { 1,  2,  3},
                                   { 4,  5,  6},
                                   { 7,  8,  9},
                                   {10, 11, 12}};
```

The array element *matrixA[3][1]* references the element in the fourth row and second column, which has a value of 11.

Look-Up Table Method Using Arrays and Parameters

The array construct together with parameter can be used to create look-up tables which can be used to create combinational circuits using the ROM or Look-up Table (LUT) method.

You

The Verilog code for the parity generator is illustrated in Figure 2-61. The first 4 bits of the output are identical to the input. Hence, instead of storing all 5 bits of the output, one might store only the parity bit and then concatenate it to the input bits. The parameter construct is used to define the ParityBit which is 1-bit constant, and the 4-bit input data and 1-bit ParityBit are concatenated to make a parity code as an output.

FIGURE 2-61: Parity Code Generator Using the LUT Method

```
module parity_gen(X, Y);
input  [3:0] X;
output [4:0] Y;

wire          ParityBit;

parameter [0:15] OT = {1'b1, 1'b0, 1'b0, 1'b1, 1'b0, 1'b1, 1'b1, 1'b0, 1'b0, 1'b1,
1'b1, 1'b0, 1'b1, 1'b0, 1'b0, 1'b1};

assign ParityBit = OT[X];
assign Y = {X, ParityBit};

endmodule
```

You

2.18 Loops in Verilog

Often, one has systems where some activity is happening in a repetitive fashion. Verilog loop statements can be used to express this behavior. A loop statement is a sequential statement. Verilog has several kinds of loop statements including **for** loops and **while** loops. There is also a **repeat** loop.

Forever Loop (Infinite Loop)

Infinite loops are undesirable in common computer languages, but they can be useful in hardware modeling where a device works continuously and continues to work until the power is off. Below is an example for a forever loop:

```
begin
  clk = 1'b0;
  forever #10 clk = ~clk;
end
```

For Loop

One way to augment the basic loop is to use the **for** loop, where the number of invocations of the loop can be specified. Syntax is similar to C language except that begin and end are used instead of { } for more than one statement. Note that

```
for (initial_statement; expression; incremental_statement)
begin

   sequential statement(s);

end
```

The following example is of initializing an array variable, `eight_bit_register_array`, using a **for** loop.

```
reg  [7:0]  eight_bit_register_array [15:0];

for (i=0; i<16; i=i+1)
begin

  register A[i] = 8'b00000000;

end
```

One could use this type of loop in behavioral models. The following excerpt models a 4-bit adder. The loop index (i) will be initialized to 0 when the **for** loop is

```
    Cout = (A[i] && B[i]) || (A[i] && Cin) || (B[i] && Cin);
      sum[i] = A[i] ^ B[i] ^ Cin;
      Cin = Cout;

  end
```

You could also use the **for** loop to create multiple copies of a basic cell. When the foregoing code is synthesized, the synthesizer typically provides four copies of a 1-bit adder connected in a ripple carry fashion. If you actually desire to create just one copy of the cell and simply use it multiple times as a serial adder, then you have to design a sequential circuit. The loop construct will not synthesize into that behavior.

While Loop

As in **while** loops in most languages, a condition is tested before each iteration. The loop is terminated if the condition is false. The general form of a while loop is

```
while condition
begin

  sequential statements;

end
```

Example

Parity bits are often used in digital communication for error detection and correction. The simplest of these involve transmitting one additional bit with the data, a parity bit. Use Verilog arrays to represent a parity generator that generates a 5-bit-odd-parity generation for a 4-bit input number using the look-up table (LUT) method.

Answer: The input word is a 4-bit binary number. A 5-bit odd-parity representation will contain exactly an odd number of 1s in the output word. This can be accomplished by the ROM or LUT method using a look-up table of size 16 entries × 5 bits. The look-up table is indicated in Figure 2-60.

Input (LUT Address)				Output (LUT Data)				
A	B	C	D	P	Q	R	S	T
0	0	0	0	0	0	0	0	1
0	0	0	1	0	0	0	1	0
0	0	1	0	0	0	1	0	0
0	0	1	1	0	0	1	1	1
0	1	0	0	0	1	0	0	0
0	1	0	1	0	1	0	1	1
0	1	1	0	0	1	1	0	1
0	1	1	1	0	1	1	1	0
1	0	0	0	1	0	0	0	0
1	0	0	1	1	0	0	1	1
1	0	1	0	1	0	1	0	1
1	0	1	1	1	0	1	1	0
1	1	0	0	1	1	0	0	1
1	1	0	1	1	1	0	1	0
1	1	1	0	1	1	1	0	0
1	1	1	1	1	1	1	1	1

You

until either the count is 0 or stop is 1.

FIGURE 2-62: Use of While Loop

```
`define MAX 100

  module counter_100;

  integer count;

  initial begin
    count = 0;
    while (count < `MAX) begin
      count = count + 1;
    end // while

    $display("number = %d ", count);
  end // initial begin

  endmodule
```

Repeat Loop

The repeat loop repeats the sequential statement(s) for specified times. The num

until either the count is 0 or stop is 1.

FIGURE 2-62: Use of While Loop

```
`define MAX 100

  module counter_100;

  integer count;

  initial begin
    count = 0;
    while (count < `MAX) begin
      count = count + 1;
    end // while

    $display("number = %d ", count);
  end // initial begin

  endmodule
```

Repeat Loop

The repeat loop repeats the sequential statement(s) for specified times. The num

are used for the sum and carry bits, the update of carry out happens instantaneously. Code like this often appears in Verilog tasks and functions (described in Chapter 8).

```
for (i=0; i<4; i=i+1)
begin

 Cout = (A[i] && B[i]) || (A[i] && Cin) || (B[i] && Cin);
   sum[i] = A[i] ^ B[i] ^ Cin;
   Cin = Cout;

end
```

You could also use the **for** loop to create multiple copies of a basic cell. When the foregoing code is synthesized, the synthesizer typically provides four copies of a 1-bit adder connected in a ripple carry fashion. If you actually desire to create just one copy of the cell and simply use it multiple times as a serial adder, then you have to design a sequential circuit. The loop construct will not synthesize into that behavior.

While Loop

As in **while** loops in most languages, a condition is tested before each iteration. The loop is terminated if the condition is false. The general form of a while loop is

```
while condition
begin
```

```
    end // while

    $display("number = %d ", count);
  end // initial begin

  endmodule
```

Repeat Loop

The repeat loop repeats the sequential statement(s) for specified times. The number of repetitions is set by a constant value or a logical expression.

```
repeat( 8 )
  begin

    x = x + 1;
    y = y + 2;

  end
```

bench are interfaced to the adder, as shown in Figure 2-63. The module on the right side is the module under test.

FIGURE 2-63: Interfacing of Signals While Using a Test Bench to Test a 4-Bit Adder

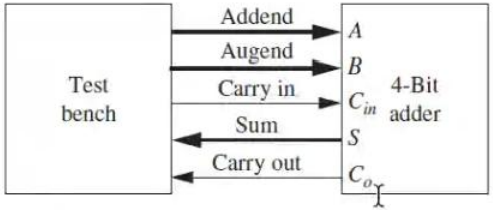


Figure 2-64 shows an example of a test bench for a 4-bit adder without using assertion monitors.

Assertion statements available in languages such as System Verilog provide an easy mechanism to specify properties that the design must adhere to and to verify them. Verilog does not have assertion statements. Hence in this test bench, we simply provide test inputs and check whether the module outputs match the expected outputs. It may be noticed that the test bench module does not have external inputs

```
module TestAdder_v2; //Statement 1

parameter  N = 11;  //Statement 2
reg  [3:0] addend;
reg  [3:0] augend;
reg        cin;
wire [3:0] sum;
wire       cout;

reg [3:0] addend_array[1:N];
reg [1:N] cin_array;
reg [3:0] augend_array[1:N];
reg [3:0] sum_array[1:N];
reg [1:N] cout_array;

initial
begin
  //initialization of addend_array
  addend_array[1] = 4'b0111;
  addend_array[2] = 4'b1101;
  addend_array[3] = 4'b0101;
  addend_array[4] = 4'b1101;
  addend_array[5] = 4'b0111;
```

```
wire        cout;

reg [3:0] addend_array[1:N];
reg [1:N] cin_array;
reg [3:0] augend_array[1:N];
reg [3:0] sum_array[1:N];
reg [1:N] cout_array;

initial
begin
  //initialization of addend_array
  addend_array[1] = 4'b0111;
  addend_array[2] = 4'b1101;
  addend_array[3] = 4'b0101;
  addend_array[4] = 4'b1101;
  addend_array[5] = 4'b0111;
  addend_array[6] = 4'b1000;
  addend_array[7] = 4'b0111;
  addend_array[8] = 4'b1000;
  addend_array[9] = 4'b0000;
  addend_array[10] = 4'b1111;
  addend_array[11] = 4'b0000;
```

```
  sum_array[5] = 4'b1111;
  sum_array[6] = 4'b1111;
  sum_array[7] = 4'b1111;
  sum_array[8] = 4'b0000;
  sum_array[9] = 4'b1110;
  sum_array[10] = 4'b1111;
  sum_array[11] = 4'b0000;

  //initialization of cout_array (expected carry output)
  cout_array[1] = 1'b0;
  cout_array[2] = 1'b1;
  cout_array[3] = 1'b1;
  cout_array[4] = 1'b1;
  cout_array[5] = 1'b0;
  cout_array[6] = 1'b0;
  cout_array[7] = 1'b0;
  cout_array[8] = 1'b1;
  cout_array[9] = 1'b0;
  cout_array[10] = 1'b1;
  cout_array[11] = 1'b0;
end
```

```
integer i;

always
begin
  for(i = 1 ; i <= N ; i = i + 1)
  begin
    $display(i);
    addend <= addend_array[i];//apply an addend test vector
    augend <= augend_array[i];//apply an augend test vector
    cin <= cin_array[i];//apply a carry in

    #(40);//adder expected to take 40 time units

    if(!(sum == sum_array[i] & cout == cout_array[i]))
    begin
      $write("ERROR: ");
      $display("Wrong Answer ");
    end
    else begin
      $display("Correct!!");
    end
  end
  $display("Test Finished");
end
```