Component Test and Verification

Adapted from Z. Navabi
Portions Copyright Z. Navabi, 2006

- We will examine:
  - How Verilog language constructs can be used for application of data to a module under test (MUT)
  - How module responses can be displayed and checked

- The first part: Data application and response monitoring
- The second part: The use of assertion verification for giving a better observability to our design modules
Verilog simulation environments provide two kinds of display of simulation results:
- Graphical
- Textual

Some also provide tools for editing input test data to a design module that is being tested.
These tools are referred to as **Waveform Editors**.

Waveform editors have 2 problems:
- Usually are good only for small designs.
- Each simulation environment uses a different procedure for waveform editing.

This problem can be solved by use of **Verilog Testbenches**.

---

A **Verilog Testbench** is:
- A Verilog module
- Instantiates Module Under Test (MUT).
- Applies data to MUT.
- Monitors the output.

A module and its testbench forms a **Simulation Module** in which MUT is tested for the same data regardless of what simulation environment is used.
Testbench

Combinational Circuit Testing

Sequential Circuit Testing

Combinational Circuit Testing

Combinational Circuit Testing
Combinational Circuit Testing

module alu_4bit (a, b, f, oe, y, p, ov, a_gt_b, a_eq_b, a_lt_b);
input [3:0] a, b;
input [1:0] f;
input oe;
output [3:0] y;
output p, ov, a_gt_b, a_eq_b, a_lt_b;
// . . .
endmodule

* alu_4bit Module Declaration

Combinational Circuit Testing

module test_alu_4bit;
reg [3:0] a=4'b1011, b=4'b0110;
reg [1:0] f=2'b00;
reg oe=1;
wire [3:0] y;
wire p, ov, a_gt_b, a_eq_b, a_lt_b;
alu_4bit cut(a, b, f, oe, y, p, ov, a_gt_b, a_eq_b, a_lt_b);
..............
..............
endmodule

* Testbench for alu_4bit
Combinational Circuit Testing

```
initial begin
  #20 b=4'b1011;
  #20 b=4'b1110;
  #80 oe=1'b0;
  #20 $finish;
end
always #23 f = f +1;
endmodule
```

* Testbench for alu_4bit (Continued)

**Combinational Circuit Testing**

- Application of data to the \( b \) data input and \( oe \) output-enable
- Every 20 ns a new value is assigned to \( b \)
- The $finish statement is reached at 160 ns. At this time all active procedural blocks stop and simulation terminates.
- After 20 ns the simulation is finished
- Allows effects of the last input change to be shown in simulation results

---

**Combinational Circuit Testing**

- \( f \) changes every 23 ns causing various ALU functions to be examined
- At 140 ns \( oe \) changes to 0 causing the \( y \) output become \( Z \)

---

**ALU Simulation Results**
Sequential Circuit Testing

- Testing sequential circuits involves synchronization of Clock with other data inputs.
Sequential Circuit Testing

```
module #([3:0] poly=0)
  input [3:0] d_in, output

always @(posedge clk)
  if (rst)
    d_out = 4'b0000;
  else
    d_out = d_in ^ ({4'b0, d_out[3:1]} & poly);
endmodule
```

* `misr` Sequential Circuit multiple input signature register

Sequential Circuit Testing

```
module test_misr;
  reg clk=0, rst=0;
  reg [3:0] d_in;
  wire [3:0] d_out;

  misr #(4'b1100) MUT (clk, rst, d_in, d_out);

........................................
........................................

* A Testbench for missr
```

The circuit has a `poly` parameter that determines its signature and data compression. With each clock a new signature will be calculated with the new data and existing `misr` register data.

Specification of the `poly` parameter
Sequential Circuit Testing

* A Testbench for misr (Continued)

The timing is so chosen to cover at least one positive clock edge.

In order to reduce chance of several inputs changing at the same time, we usually use prime numbers for timing of sequential circuit inputs.

$d_{in}$ input is assigned a new value every 37 ns.

Toggles every 11ns.

Sequential Circuit Testing

* Testing misr
Testbench Techniques

- Test Data
- Simulation Control
- Limiting Data Sets
- Applying Synchronized Data
- Synchronized Display of Results
- An Interactive Testbench
- Random Time Intervals
- Buffered Data Application

---

module moore_detector (input x, rst, clk, output z);

parameter [1:0] a=0, b=1, c=2, d=3;
reg [1:0] current;

always @(posedge clk )
if (rst) current = a;
..........................
..........................
..........................

endmodule

---

* 101 Moore Detector for Test

---
if (rst) current = a;
else case (current)
    a : current = x ? b : a;
    b : current = x ? b : c;
    c : current = x ? d : a;
    d : current = x ? b : c;
default : current = x;
endcase
assign z = (current==d) ? 1'b1 : 1'b0;
endmodule

* 101 Moore Detector for Test (Continued)
module test_moore_detector;
    reg x, reset, clock;
    wire z;

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

endmodule

* Basic Data Generation

initial begin
    clock=1'b0;
    x=1'b0;
    reset=1'b1;
end

initial #24 reset=1'b0;
always #5 clock=~clock;
always #7 @=-x;
endmodule

* Basic Data Generation (Continued)
Simulation Control

Testbench Techniques

- Test Data
- Limiting Data Sets
- Synchronized Display of Results
- Random Time Intervals

Simulation Control

- Applying Synchronized Data
- An Interactive Testbench
- Buffered Data Application

Simulation Control

```
module test_moore_detector;

reg x=0, reset=1, clock=0;
wire z;

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

initial #24 reset=1'b0;
always #5 clock=~clock;
always #7 x=~x;
initial #189 $stop;
endmodule
```

* Testbench with $stop Simulation Control

Simulation Control Tasks are $stop and $finish

A stopped simulation can be resumed.

The first time the flow of a procedural block reaches $stop at 189 ns, simulation stops.

Uses another initial block that stops the simulation at 189 ns.
module test_moore_detector;
  reg x=0, reset=1, clock=0;
  wire z;

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

  ............................................................
  ............................................................
endmodule

* Testbench with $finish Simulation Control

Another testbench for moore_detector with $finish Control Task

initial begin
  #24 reset=1'\b0;
  #165 $finish;
end
always #5 clock=~clock;
always #7 x=~x;
endmodule

* Testbench with $finish Simulation Control (Continued)

This testbench combines the initial blocks of deactivating reset and simulation control into one initial block.

Simulation terminates at 165 ns.

A finished simulation cannot be resumed.
Limiting Data Sets

- Testbench Techniques
  - Test Data
  - Simulation Control
  - Limiting Data Sets
  - Applying Synchronized Data
  - Synchronized Display of Results
  - An Interactive Testbench
  - Random Time Intervals
  - Buffered Data Application

---

### Module: test_moore_detector

```verilog
module test_moore_detector;
    reg x=0, reset=1, clock=0;
    wire z;

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

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

* Testbench Using repeat to Limit Data Sets

---

Instead of setting 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 cause clock to toggle 13 times every 5 ns.

In large circuits, random data is more useful for data inputs than for control signals.
* Where several sets of data are to be applied:
  * Synchronization of data with the system clock becomes difficult.
  * Changing the clock frequency would require changing the timing of all data inputs of the module being tested.
Applying Synchronized Data

```verilog
module test_moore_detector;
    reg x=0, reset=1, clock=0;
    wire z;

    moore_detector uut( x, reset, clock, z );

    initial #24 reset=0;
    initial repeat(13) #5 clock=~clock;
    initial forever @(posedge clock) #3 x=$random;

endmodule
```

* Synchronizing Data with Clock

This testbench uses an event control statement to synchronize data applied to `x` with the clock. This loop waits for the positive edge of the clock, and 3 ns after it, a new random data is generated for `x`. This 3ns delay makes it possible to use this testbench for simulating post-synthesis designs as well as behavioral descriptions. The setup and hold time delay ensures that the changing of data and clock do not coincide.

Synchronized Display of Results

- Test Data
- Simulation Control
- Limiting Data Sets
- Applying Synchronized Data
- Synchronized Display of Results
- An Interactive Testbench
- Random Time Intervals
- Buffered Data Application
Synchronized Display of Results

module test_moore_detector;
    reg x=0, reset=1, clock=0;
    wire z;

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

    initial #24 reset=0;
    initial repeat(13) #5 clock=~clock;
    initial forever @(posedge clock) #1 $random;
    initial forever @(posedge clock) #1 $displayb(z);
endmodule

* Testbench Displaying Output

This testbench uses an event control statement for synchronized observation. Ins after the positive edge of the clock, when the circuit output is supposed to have its new stable value, the z output is displayed using the $display task.

This testbench is also usable for the post-synthesis simulation of moore_detector.

Synchronized Display of Results

module test_moore_detector;
    reg x=0, reset=1, clock=0;
    wire z;

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

    .........................
    .........................
endmodule

* Testbench Displays Design Variables when they change

This testbench is developed for observing states of moore_detector.
Synchronized Display of Results

- Starts $monitor task in the background
- $monitor("New state is x and occurs at $t, MUT.current, $time);
- always @(z) $display("Output changes at $t to $b", z);
- endmodule

* Testbench Displays Design Variables when they change

- Uses $display to display the output
- Sensitive to z
- Event Based
- In binary format
- Flow Based
- Hierarchial Naming
- In binary format
- With the time unit

New state is x and occurs at
Output changes at
New state is 0 and occurs at
New state is 1 and occurs at
New state is 2 and occurs at
Output changes at
New state is 3 and occurs at

* Test Results of Testbench
An Interactive Testbench

module moore_detector (input x, start, rst, clk, output z);

parameter a=0, b=1, c=2, d=3, e=4;

reg [2:0] current;

always @(posedge clk )
    .........................
    .........................
    .........................
    ...........

endmodule

* Moore Sequence Detector Detecting 1101
An Interactive Testbench

```verilog
if ( rst ) current <= a;
else if ( ~start ) current <= a;
else case ( current )
  a : current <= x ? b : a ;
b : current <= x ? c : a ;
c : current <= x ? c : d ;
d : current <= x ? e : a ;
e : current <= x ? c : a ;
default: current <= a;
endcase
assign z = (current==e);
endmodule
```

* Moore Sequence Detector Detecting 1101 (Continued)

An Interactive Testbench

```verilog
module test_moore_detector;
  reg x=0, start, reset=1, clock=0;
  wire z;

  moore_detector MUT ( x, start, reset, clock, z );
  ................................
  ................................
  ................................
endmodule
```

* An Interactive Testbench
An Interactive Testbench

initial begin
    #24 reset=1'b0; start=1'b1;
    wait(z == 1'b1);
    #11 start=1'b0;
    #13 start=1'b1;
    repeat(3) begin
        #11 start=1'b0;
        #13 start=1'b1;
        wait(z == 1'b1);
    end
    #50 $stop;
end

To get the machine started

Waits for z to become 1,
After it restarts the machine

Repeats the process of starting the machine and waiting for z to become 1
3 more times

After 50 ns simulation is stopped

* An Interactive Testbench (Continued)

always
    #5 clock=~clock;
always
    #7 x=$random;
endmodule

always blocks to generate clock and x input

* An Interactive Testbench (Continued)
An Interactive Testbench

```
module test_moore_detector;
    reg x=0, start, reset=1, clock=0;
    wire z;

    moore_detector MUT ( x, start, reset, clock, z );

    initial begin
        #24 reset=1'b0; start=1'b1;
    end

endmodule
```

* Interactive Testbench Using Display Tasks
An Interactive Testbench

Hierarchical Naming

When current becomes e

z is displayed

always begin

wait(MUT.current == MUT.e);

$display

("$display task shows: The output is %b, z);

$strobe

("$strobe task shows: The output is %b, z);

#2 $stop;

end

always #5 clock=~clock;

always #7 x=$random;

endmodule

* Interactive Testbench Using Display Tasks (Continued)

Random Time Intervals

Testbench Techniques

Test Data

Simulation Control

Limiting Data Sets

Applying Synchronized Data

Synchronized Display of Results

An Interactive Testbench

Random Time Intervals

Buffered Data Application
module test_moore_detector;
    reg x, start, reset, clock;
    wire z;
    reg [3:0] t;
    moore_detector MUT ( x, start, reset, clock, z );
    ....................
    ....................
endmodule

* Testbench using Random Time Intervals

initial begin:running
    clock <= 1'b0; x <= 1'b0;
    reset <= 1'b1; reset <= #7 1'b0;
    start <= 1'b0; start <= #17 1'b1;
    repeat (13) begin
        @(posedge clock );
        @(negedge clock );
    end
    start=1'b0;
    #5;
    $finish;
end

* Testbench using Random Time Intervals (Continued)
Random Time Intervals

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

This block generates data on x as long as the $finish statement is not reached

Generates Random data on t

Uses t to delay assignments of random values to x

* Testbench using Random Time Intervals (Continued)

Buffered Data Application

Testbench Techniques

Test Data Simulation Control
Limiting Data Sets Applying Synchronized Data
Synchronized Display of Results An Interactive Testbench
Random Time Intervals Buffered Data Application
Buffered Data Application

```verilog
module test_moore_detector;

reg x=0, rst, start, clk=0;
wire z;
reg [18:0] buffer;

moore_detector MUT ( x, start, rst, clk, z );

endmodule
```

* Testbench Applying Buffered Data

This testbench uses a buffer to hold data to be applied to the MUT data input.

```
initial begin
rst=1'b1; start=1'b0;
#29 rst=1'b0;
#29 start=1'b1
#500 $stop;
end
```

19-bit `buffer` is initialized with test data.

Start and stop control of the state machine.

As data is shifted, `buffer` is rotated in order for the applied buffered data to be able to repeat.

We are sure a correct sequence is applied to our MUT and can more easily check for expected results.

Each bit of the buffer is shifted out onto the `x` input of MUT ins after the positive edge of clock.
Design Verification

- **Formal verification:**
  - A way of automating design verification by eliminating testbenches and problems associated with their data generation and response observation.
  - Tools do not perform simulation, but come up with a Yes/No answer for every property the design is being checked for.
  - Eliminating data generation and response observation

- **Assertion verification:**
  - Reduce or eliminate efforts needed for analyzing output responses
  - While the design is being simulated with its testbench data, assertion monitors continuously check for correct design behavior.
  - In conditions that the design is misbehaving, the monitor is said to fire to alert the designer of the problem.

Assertion Verification

- Assertion Verification Benefits
- Open Verification Library
- Using Assertion Monitors
- Assertion Templates
Assertion Verification

* Unlike simulation, here in-code monitors issue a message if something happens that is not expected.
* In Verilog, monitors are modules.
* The present set of assertion monitors are available in a library referred to as OVL (Open Verification Library).
* For using assertions designer compiles OVL and his or her own assertion library into a simulation library.
* If a signal does not have a value expected by a monitor, the assertion monitor displays a message and the time that the violation of the property has occurred.

Assertion Verification Benefits

- Open Verification Library
- Using Assertion Monitors
- Assertion Templates
- Assertion Verification Benefits

CSE 467 Verilog Digital System Design
Assertion Verification Benefits

- Ways in which assertion monitors are helpful:
  - **Designer Discipline**: With placing an assertion in a design, a designer is disciplining him/her-self to look into the design more carefully and extract properties.
  - **Observability**: Assertions add monitoring points to a design that make it more observable.
  - **Formal Verification Ready**: Having inserted assertion monitors to a design, readies it for verification by a formal verification tool.
  - **Executable Comments**: Assertion monitors can be regarded as comments that explain some features or behavior of a design.
  - **Self Contained Designs**: A design with assertion monitors has the design description and its test procedure all in one Verilog module.

Open Verification Library

- Assertion Verification
- Open Verification Library
- Using Assertion Monitors
- Assertion Templates

Using Assertion Monitors
Open Verification Library

| assert_always      | assert_always_on_edge      |
| assert_change      | assert_cycle_sequence      |
| assert_decrement   | assert_delta               |
| assert_even_parity | assert_fifo_index           |
| assert_frame       | assert_handshake           |
| assert_implication | assert_increment           |
| assert_never       | assert_never_at_x_or_z     |
| assert_next        | assert_no_overflow         |
| assert_no_transition | assert_no_underflow     |
| assert_odd_parity  | assert_one_cold            |
| assert_one_hot     | assert_proposition         |
| assert_quiescent_state | assert_range            |
| assert_time        | assert_transition          |
| assert_unchange    | assert_width               |
| assert_win_change  | assert_win_unchange        |
| assert_window      | assert_zero_one_hot        |

* Assertions

An Assertion is placed in code like a module instantiation.

assert-name

#(static-parameters)

instance-name

(dynamic-arguments);

* Assertion Module Instantiation

An Assertion is placed in code like a module instantiation.

Assertion module name comes first.

Followed by static_parameters like vector size and options

Any Unique name is allowed

Reference and monitor signals and other dynamic arguments
Using Assertion Monitors

Assertion Verification

Benefits

Open Verification Library

Assertion Templates

Using Assertion Monitors

assert_always

assert_change

assert_one_hot

assert_cycle_sequence

assert_next
**assert_always**

**Assertion Monitors**

- **assert_always**
- **assert_change**
- **assert_one_hot**
- **assert_cycle_sequence**
- **assert_next**

*General Format for *assert_always* Assertion Monitor*

```verilog
#(
    severity_level, property_type,  
    msg, coverage_level )
instance_name ( clk, reset_n, test_expr )
```

If the test expression fails, the assertion fires and its corresponding message (msg) is displayed.

Continuously checks its `test_expr` to make sure it is always true on the edge of the specified clock (clk).
**assert_always**

```verilog
module BCD_Counter (input rst, clk,
                     output reg [3:0] cnt);

always @(posedge clk) begin
    if (rst || cnt >= 10) cnt = 0;
    else cnt = cnt + 1;
end

assert_always #1, 0, "Err: Non BCD Count", 0)
    AA1 (clk, 1'b1, (cnt >= 0) & (cnt <= 9));
endmodule
```

- This counter counts between 0 and 9
- BCD with `assert_always`
- The assertion monitor here uses `severity_level 1`
- Indicates that the assertion is to be monitored at all times
- The test expression:
  - The monitor checks that on every rising edge of `clk`, `cnt` must be between 0 and 9

---

**assert_always**

```verilog
module BCD_Counter_Tester;
    reg r, c;
    wire [3:0] count;

    BCD_Counter UUT (r, c, count);
        initial begin
            r = 0; c = 0;
        end
        initial repeat (200) #17 c = ~c;
        initial repeat (03) #807 r = ~r;
endmodule
```

- BCD Counter Testbench
- Even though checking simulation results is done in a semi-automatic fashion, test data generation is still done manually by the designer
assert_change

Assertion Monitors

assert_always

assert_change

assert_one_hot

assert_cycle_sequence

assert_next

assert_change

Verifies that within a given number of clocks after the start event, the test expression changes.

assert_change

#( severity_level, width, num_cks,
    action_on_new_start, property_type,
    msg, coverage_level )
instance_name ( clk, reset_n, start_event,
    test_expr )

* General Format for assert_change Assertion Monitor
```verilog
module Walking_One (input rst, clk, output reg [7:0] wo);
    always @(negedge clk) begin
        if (rst) wo <= 8'b10000000;
        else wo <= {wo[0], wo[7:1]};
    end

    assert_change #(1, 1, 7, 0, 0, "Err: Bit 0 is not changing", 0)
        AC1 (~clk, ~rst, {rst==0}, wo[0]);
    assert_one_hot #(1, 8, 0, "Err: Multiple active bits", 0)
        AOH (~clk, ~rst, wo);
endmodule

* Walking One Circuit with Assertions

---

A shift register that walks a 1 with every clock.

A 1 is loaded into the left-most bit of wo with the rst signal.

Check that from the time that (rst == 0) and while (~rst), it takes at most 7 negative clock edges for wo[0] to change.
It is the responsibility of the testbench developer to make sure enough data is applied to cause design errors to trigger assertion monitors.

When \( \text{rst} \) becomes 1, the falling edge of the clock puts a 1 into \( \text{wo}[7] \).

When \( \text{rst} \) becomes 0, this 1 starts walking, it takes 7 clock edges for this 1 to walk to bit 0 of \( \text{wo} \).
assert_one_hot

Assertion Monitors

assert_always
assert_change
assert_one_hot
assert_cycle_sequence
assert_next

assert_one_hot

Checks that while the monitor is active, only one bit of its n-bit test expression is 1.

---

assert_one_hot

#( severity_level, width, property_type, msg, coverage_level )
instance_name ( clk, reset_n, test_expr )

* General Format for assert_one_hot Assertion Monitor
module Walking_One (input rst, clk, output reg [7:0] wo);
    always @(negedge clk) begin
        if (rst) wo <= 8’b10000000;
        else wo <= {wo[0], wo[7:1]};
    end

    assert_change #(1, 1, 7, 0, 0, "Err: Bit 0 is not changing", 0)
            ACL (~clk, ~rst, (rst==0), wo[0]);
    assert_one_hot #(1, 8, 0, "Err: Multiple active bits", 0)
            AOH (~clk, ~rst, wo);
endmodule

* Walking One Circuit with Assertions
assert_one_hot

module gray_counter (input [3:0] d_in, clk, rst, ld, output reg [3:0] q);
reg [3:0] mem[0:15];
reg [3:0] im_q;
initial $readmemh("mem.dat", mem);
always @(d_in or ld or q)
if (ld)
    im_q = d_in;
else
    im_q = mem[q];
end

............................
* Gray Code Counter

............................
always @(posedge clk)
if (rst)
    q <= 4'b0000;
else
    q <= im_q;
end
always @(posedge clk)
reg [3:0] old;
assert_one_hot #(1, 4, 0, "Err: Not Gray", 0)
    AOH (~clk, ~rst, (old ^ q));
endmodule

assert_one_hot

The mem.dat file that contains consecutive Gray code numbers is read into mem.
With each clock, the next Gray count is looked up from mem.

In order to check for the correct Gray code sequencing, some auxiliary logic have been used to prepare the test expression.
Consecutive Gray code numbers are only different in one bit, their XOR must be one-hot.
assert_cycle_sequence

Checks for a sequence of events in a given number of clocks.

Like other assertion monitors, this monitor has an enabling input that is usually driven by the inactive level of a circuit's reset input.

assert_cycle_sequence

#( severity_level, num_cks, necessary_condition, property_type, msg, coverage_level )
instance_name ( clk, reset_n, event_sequence )

* General Format for assert_cycle_sequence
After the reset state, the machine searches for 110 sequence.

* A Sequencing State Machine

---

**assert_cycle_sequence**

```verilog
module Sequencing_Machine (input x, start, rst, clk, output z);

parameter a=0, b=1, c=2, d=3, e=4;

reg [2:0] current;

............................
............................
............................

* Verilog Code of Sequencing_Machine
```
assert_cycle_sequence

always @( posedge clk )
  if ( rst ) current <= a;
  else if ( ~start ) current <= a;
  else case ( current )
    a : current <= x ? b : a ;
    b : current <= x ? c : a ;
    c : current <= x ? c : d ;
    d : current <= e ;
    e : current <= a ;
    default: current <= a;
endcase

* Verilog Code of Sequencing Machine (Continued)
### assert_cycle_sequence

- **Sequencing Machine State Transitions**

If the machine enters state 4 (e), then state 0 (a) is entered.

### assert_next

- **Assertion Monitors**

  - **assert_always**
  - **assert_change**
  - **assert_one_hot**
  - **assert_cycle_sequence**
  - **assert_next**
assert_next

Verilog Code of Sequencing Machine (Continued)

```verilog
assign z = (current==e);
assert_cycle_sequence
#(1, 3, 0, 0, "Err: State sequence not followed", 0)
instance_name ( clk, reset_n, start_event, test_expr)
assert_next
#(1, 2, 1, 0, 0, "Err: Output state not reached", 0)
ANL (clk, ~rst, (current==c & x==0), (z==1));
endmodule
```

* Verilog Code of *Sequencing Machine* (Continued)
• Hardware features designers may need to verify, and how assertions can be used for their verification
Assertion Templates

Reset Sequence

Initial Resetting

Implication

Valid States

Resetting

Assertion Templates

Valid States

Implication

Initial Resetting

Resetting

Assertion Templates

Valid States

Implication

Initial Resetting

Resetting

Assertion Templates
# Reset Sequence

```verilog
module Sequencing_Machine(input x, start, rst, clk, output z);

parameter a=0, b=1, c=2, d=3, e=4;

// . . .

assert_cycle_sequence
#(1, 4, 0, 0, "Err:Resetting does not occur")
ACS2 (clk, ~rst, {(x==0), (x==0), (x==0), (current==a)});

// . . .
endmodule
```

- Assertion Reset Sequence

Often controllers have a resetting sequence that with certain sequence of inputs, the machine ends in.

Verifies that if in three consecutive clocks, `x` is 0, in the fourth clock the current state of the machine becomes `a`.

## Initial Resetting

- Reset Sequence
- Implication
- Valid States
- Initial Resetting

### Assertion Templates

- Reset Sequence
- Initial Resetting
module mealy_detector (input x, rst, clk, output z);
localparam [1:0] reset = 0, // 0 = 0 0
              got1 = 1, // 1 = 0 1
              got10 = 2; // 2 = 1 0
reg [1:0] current;
............................
............................
............................
............................
............................
............................
endmodule

* Hard-Reset Assertion

always @(posedge clk) begin
if (rst) current <= reset;
else case (current)
  reset: if (x==1'b1) current <= got1;
         else current <= reset;
  got1: if (x==1'b0) current <= got10;
         else current <= got1;
  got10: if (x==1'b1) current <= got1;
         else current <= reset;
  default: current <= reset;
endcase end

* Hard-Reset Assertion (Continued)
Initial Resetting

assign z = (current==got10 & x==1'b1) ? 1'b1 : 1'b0;
assert_next
#(1, 1, 1, 0, 0, 0)

AN1 (clk, [1'b1] rst, (current==reset));

* Hard-Reset Assertion  (Continued)

Assertion Monitor is always active.
Checks if rst is 1 then the next current state becomes reset

Implication

Assertion Templates

Reset Sequence

Initial Resetting

Implication

Valid States
**Implication**

```
assert_implication
    #(
        severity_level, property_type,
        msg, coverage_level
    )
    instance_name ( clk, reset_n,
        antecedent_expr,
        consequence_expr)
```

* General Format for `assert_implication` Assertion Monitor

**Checks on the specified clock edge for the antecedent expression to be true. If it is, then it checks for the consequence expression to be true. If so, it will stay quiet, otherwise it will fire.**

---

**Implication**

```
module mealy_detector2 (input x, rst, clk, output z);
    // . . .
    assert_implication
        #(1, 0, "Err: Output not asserted", 0) AI1 (clk, 1'b1, (current==got10 && x), (z==1));
    // . . .
endmodule
```

* Asserting Implication

**Checks the output value in the got10 state while x is 1**

**Always Active**
Valid States

Assertion Templates

- Reset Sequence
- Initial Resetting
- Implication
- Valid States

Valid States

- General Format for `assert_no_overflow` Assertion Monitor

```verilog
assert_no_overflow #(
  severity_level, width, min, max,
  property_type,
  msg, coverage_level
)
instance_name ( clk, reset_n, test_expr )
```

If the states of the machine being tested are consecutive binary numbers

In the sequential circuit testing it often becomes necessary to check for the machine's valid states and issue a warning if the machine enters an invalid state.
Valid States

Of the four possible states, it is using only three: reset, got1, got10.

```verilog
module mealy_detector2 (input x, rst, clk, output z);

// . . .
assert_no_overflow #(1, 2, 0, 2, 0, “Err: Invalid state”), 0)

ANV1 (clk, 1′b1, current);
// . . .
endmodule
```

- Checking for Invalid States

Text Based Testbenches

- Verilog has an extensive set of tasks for reading and writing external files:
  - Opening and closing files,
  - Positioning a pointer in a file,
  - Writing or appending a file
  - Reading files
Summary

- This chapter discussed:
  - The use of Verilog constructs for developing testbenches
  - Data generation and response analysis by use of Verilog
  - How assertion monitors could be used for reducing efforts needed for response analysis of a unit-under-test.
  - Developing good testbenches for complex designs requires design observability given to designers by use of assertions.