Design Verification

- Use of assertions in circuits and testbenches
- Testing small combinational circuits
- Testing state machines
- Techniques for making testing more automatic
- Strategies for larger circuits

Jon Turner
Importance of Verification

- Bugs in digital circuits can be very costly
  - “re-spins” of custom circuits can cost several million $$
  - delays caused by re-spins can lower product sales
  - buggy products may have to be recalled and replaced
    - bugs in safety-critical products may lead to injuries and associated legal liabilities

- Companies invest lots of effort in circuit verification
  - projects often have more verification engineers than design engineers
  - verification engineers actively try to “break” circuits

- How are circuits checked for correctness?
  - simulation – both at system level and individual modules
  - formal verification – useful, but limited application
Assertions in Circuit Specifications

- **Check range of input argument**
  ```vhdl
  procedure update(i: in unsigned(4 downto 0); v: in unsigned(7 downto 0)) is begin
    assert(0 <= to_integer(i) and to_integer(i) <= 23);
    if table(to_integer(i)).valid = '1' then
      table(to_integer(i)).value <= v;
    end if;
  end procedure update;
  ```

- **Check consistency of procedure outputs**
  ```vhdl
  procedure minMax(a, b: in signed(7 downto 0); x, y: out signed(7 downto 0)) is begin
    if a < b then x <= a; y <= b;
    else x <= b; y <= a;
  end if;
  assert(x <= y);
  end procedure minMax;
  ```
Testing Combinational Circuits

- Small circuits can be tested exhaustively
  - and gives us confidence in correctness of larger versions
- Example: 4 bit 2s complement circuit
  - complement data input when control input (c) is high,
    else output=input
  - note: 5 inputs, so 32 distinct input values

```vhdl
entity negate is
  generic (size: integer);
  port (
    c: in std_logic;  -- negate when c = high, otherwise dout=din
    din : in signed(size-1 downto 0);
    dout : out signed(size-1 downto 0));
  end negate;
architecture al of negate is begin
  dout <= (din'range => '0') - din when c = '0' else din;
end al;
```

note: logic reversed
Testbench for Negate

```vhdl
library ieee;
...
use work.txt_util.all;
...
entity testNegate is end testNegate;
architecture al of testNegate is
...
begin
  uut: negate generic map (size => 4) port map(c, din, dout);
  tb : process begin
    c <= '0';
    for i in -8 to 7 loop
      din <= to_signed(i,4);
      wait for pause;
      assert (dout = din) report "error when c=0 i=" & str(i);
    end loop;
    c <= '1';
    for i in -8 to 7 loop .. end loop;
    assert (false) report "Simulation ended normally." severity failure;
  end process;
end al;
```

Simulation of Negate Circuit

Bug apparent in waveform window

Also reported on simulation console due to assertion violations
Testbench for FindFirstOne Circuit

- Circuit determines index of rightmost 1 in word

```vhdl
architecture al of testFindFirstOne is begin ...
  process
    function mask(x: word; i: integer) return word ...
    begin
      din <= x"0000"; wait for 10 ns;
      for i in 0 to wordSize-1 loop
        dIn <= mask(x"a35d",i); wait for 10 ns;
        for j in 0 to int(firstOneIndex) - 1 loop
          assert dIn(j) = '0'
          report "detected error when i=" & str(i);
        end loop;
        assert (valid = '0' and dIn = (dIn'range => '0')) or
               (valid = '1' and dIn(int(firstOneIndex)) = '1'
               report "detected error when i=" & str(i);
      end loop;
  end process;
end al;
```

- `mask(x,i)` is copy of `x` with 0s in bits 0..i
- Clear bits from right causing result to increase
- `firstOneIndex` is result
- Assertion checks that when valid is high `dIn(firstOneIndex)=1`
- Assertion checks that bits before `firstOneIndex` are 0
Simulation of FindFirstOne Circuit

- Note how input values affected by masking.
- Result increases as rightmost bit shifts to right.
Exercises

1. In the testbench on slide 5, write out what should appear in the second loop, including an appropriate `assert` statement.

2. Write an implementation of the `mask` function used in the testbench for `findFirstOne`.

3. Consider a combinational circuit called `findLastPulse` that takes a 16 bit input `dlIn` and has a valid output and a four bit output called `pulseIndex`, which identifies the position of the leftmost “pulse” in `dlIn`, where a pulse is the bit sequence 010. So for example, if `dlIn=1101001101001010`, then `pulseIndex` should be 12, since middle bit of the pulse is at bit position 12. If `dlIn` contains no pulse, `valid` should be low, otherwise it should be high. Write a function `doTest` with a single argument `testInput`. `doTest` should perform a single test on the circuit and should check the circuit outputs to make sure that they are consistent with the input. So for example, if the valid output of the circuit is high and the `pulseIndex` output is 7, `doTest` should verify that bit 7 of `testInput` is high, bits 6 and 8 are low and that there is no other pulse to the left of bit 7. Use an assertion to print a message if a violation is detected.
Exhaustively Testing Larger Circuits

- Some circuits are too big to check manually, but can still be checked exhaustively using assertions
- Eight bit barrel shifter
  - 8 data inputs, 8 data outputs and 3 bit shift control
  - output is rotated version of input
- Eleven input bits means 2048 input combinations
  - way too many to check manually
  - but not too many for simulator to check using assertions
- To enable automated checking, use two shifters
  - connect input of second shifter to output of first
  - shift inputs $i$ and $j$, where $i+j=8$, so output of second shifter equals input of first
Testbench for Barrel Shifter

architecture ... begin
-- instantiate two shifters
   uut1: barrelShift port map (dIn, shift1, dMid);
   uut2: barrelShift port map (dMid, shift2, dOut);
process begin
   wait for 100 ns;
   for i in 0 to 255 loop
      din <= to_unsigned(i,8);
      shift1 <= "000"; shift2 <= "000"; wait for pause;
      assert dIn = dMid
      report "error detected when i=" & str(i) & " and shift1=" & str(to_integer(shift1));
      for j in 1 to 7 loop
         shift1 <= to_unsigned(j,3); wait for 1 ps;
         shift2 <= "000" - shift1; wait for pause;
         assert dIn = dout
         report "error detected when i=" & str(i) & " and i=" & str(to_integer(shift1));
      end loop;
   end loop;
end loop;
"
Barrel Shifter Simulation

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>200 ns</th>
<th>250 ns</th>
</tr>
</thead>
<tbody>
<tr>
<td>shift1[2:0]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>shift2[2:0]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>din[7:0]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dmid[7:0]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dout[7:0]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Shift values add to 0
- Output of second matches input of first
Testing State Machines

- General strategy is to visit every state and follow every possible transition (including self-loops)
- Use state diagram to formulate plan
- Example – Mealy-mode arbiter circuit
  - start in idle and choose inputs that traverse all transitions
    - include self-loops
    - to be complete, test all input combinations per transition
  - possible sequence
    - 00, 10, 10, 11, 00, 10, 00, 00, 01, 01, 11, 00, 01, 00, 11, 00, 11, 00
architecture al of testArbiter is ...

type testData is record requests: SLV(0 to 1); grants: SLV(0 to 1); end;
type testVec is array(natural range <>) of testData;
constant tv: testVec := {
    ('00', "00"), ('10', "10"), ('10', "10"), ('11', "10"), ...
};

begin ...

process
    procedure doTest(td: testData) is
        variable r0, r1, g0, g1: std_logic;
    begin
        r0 := td.requests(0); r1 := td.requests(1);
        g0 := td.grants(0); g1 := td.grants(1);
        request0 <= r0; request1 <= r1; wait for 1 ps;
        assert grant0 = g0 and grant1 = g1
        report "error detected when r0,r1=" & str(r0) & "," & str(r1) & 
        " expected " & str(g0) & "," & str(g1) & 
        " actual " & str(grant0) & "," & str(grant1);
    wait for clk_period;
    end;
begin
    wait for 100 ns;
    reset <= '1'; wait for clk_period; reset <= '0';
    for i in tv.range loop doTest(tv(i)); end loop;
    ...
end; end;
Arbiter Simulation
Testing Serial State Machines

- Circuits that process numerical data bit-by-bit can be tedious to test
  - automate process to make testing easier
- Example – serial inRange circuit
  - inputs hi, x and lo presented one bit at a time, msb first
  - output=1 initially, but becomes 0 if x is not between lo & hi
  - test using multiple input sequences that traverse all paths
Testbench for InRange

architecture a1 of testInRange is

type testData is record
  lo, x, hi, result: std_logic_vector(7 downto 0);
end record inVec;
type testVec is array(natural range <>) of testData;
constant tv: testVec := (  
  (x"55", x"55", x"5a", x"ff"),
  (x"55", x"5a", x"5a", x"ff"), ...
);
begin
  process begin
    wait for pause;
    for i in td'low to td'high loop
      reset <= '1'; wait for pause; reset <= '0'; wait for pause;
      for j in 7 downto 0 loop
        lo <= tv(i).lo(j); x <= tv(i).x(j); hi <= tv(i).hi(j);
        wait for pause;
        assert (InRangeOut = tv(i).result(j)) report ... end
      end loop;
      lo <= '0'; x <= '0'; hi <= '0';
      end loop;
  end a1;
Simulation Output
Testing State Machines with Data

- Useful to have testbench check internal data
  » equip module with extra output(s)
  » can be commented out when not testing
- For pulse count circuit, monitor internal counter
  » testbench verifies both event output and count values
begin
  uut: pulseCount port map (counter, clk, reset, N, dIn, eventOut);
  ...
process
  procedure doTest(count: std_logic_vector(3 downto 0);
    dinBits: std_logic_vector(0 to 19);
    evBits: std_logic_vector(0 to 19);
    cntVals: std_logic_vector(79 downto 0)) is
begin
  reset <= '1'; wait for pause;
  reset <= '0'; N <= count; wait for clk_period; N <= (others => 0);
  for i in 0 to 19 loop
    dIn <= dinBits(i); wait for clk_period;
    assert eventOut = evBits(i) and
    counter = cntVals(4*(19-i)+3 downto 4*(19-i))
    report "error detected at step " & str(i) ...
  end loop;
end;
begin
  wait for 100 ns;
  ...
  doTest(x"5","01110010100110111111",
         "0000000000000000001111",
         x"555544432221111000");
  ...
end;
Sample Simulation Output

count tracking pulses as expected
Strategies for Larger Circuits

- For state machines, check controller transitions, but selectively test internal data values
  - focus on “corner cases” (atypical/abnormal inputs)
  - cases that trigger state/output changes
  - check for common errors (e.g. off-by-one errors)
- Random sample of possible input values
  - using random number generator
  - for large samples, use assertions to check results
  - provides systematic way to check large sample of inputs and is often very effective in identifying errors
  - limitation is that random inputs may miss cases most likely to be source of errors
Unit Testing

- Circuits often have many separate modules
- Important to test individual modules, as well as complete circuit
  - easy to miss errors in individual modules when testing the complete circuit
    - errors may be subtle and get overlooked; or may be hidden
  - difficult to exercise individual modules thoroughly in context of complete system
    - individual module tests make it easier to isolate errors
- CAD tools allow multiple testbenches in a project
  - each testbench instantiates and exercises one module
  - to simulate a module, first select its testbench
Exercises

1. Consider a VHDL module defining a combinational circuit called \textit{reverseFt}, that has an \( n \) bit input \( x \) and an \( n \) bit output \( y \), where the bits of \( y \) are the reverse of the bits in \( x \). \( n \) is a generic parameter of the component. Write a testbench that tests this component for all possible input values when \( n=8 \). Your testbench should check the output automatically and should use just one instance of the component.

2. Write a second testbench for the circuit in the previous example that uses two copies of the component.

3. The testbench for the \textit{pulseCount} circuit shows one call to the \textit{doTest} procedure. This test does not cover all the transitions in the state diagram for \textit{pulseCount}. Write another call to \textit{doTest} that uses the remaining transitions.