CS61C : Machine Structures

Lecture 4.2.1

Verilog II

2004-07-14

Kurt Meinz

inst.eecs.berkeley.edu/~cs61c

Review (1/3)

- Verilog allows both structural and behavioral descriptions, helpful in testing
- Syntax a mixture of C (operators, for, while, if, print) and Ada (begin… end, case…endcase, module …endmodule)
- Some special features only in Hardware Description Languages
  - # time delay, initial vs. always
- Verilog can describe everything from single gate to full computer system; you get to design a simple processor

Review (2/3): Behavorial Mux in Verilog

```
module my_mux(Z, A, B, Sel);
 input A, B, Sel
 output Z;
 reg Z;
 always @ (A or B or Sel)
     if (Sel)
       #2 Z = A;
     else
       #2 Z = B;
endmodule
```

- Sensitivity list: Signals to which the always block will respond.

Review (3/3): Verilog Time

- Verilog describes hardware
- An essential part of description is delay
- In both structural and behavioral Verilog, determine time with #n : event will take place in n time units
  - structural: not `#2(notX, X)` says notX does not change until time advances 2 u
  - behavioral: `#2 Z = A ^ B;` says Z does not change until time advances 2 units
  - Use `‘timescale`

Outline

- Dataflow
- Testing
- Verilog FSMs
- Verilog CPU Blocks

Digital Design – Tools - HDLs

- Hardware Description Languages:
  - Contain hierarchical netlist support
    - ✓ - E.g. “Structural Verilog”
  - Contain support for logic testing, rough prototyping, architectural simulation
    - ✓ - E.g. “Behavioral Verilog”
  - Other integrated design paradigms (and mappings to netlists)
    - ?? - E.g. “Verilog Dataflow”

Ex: Verilog Dataflow Paradigm

```
module my_mux_data(Z, A, B, Sel);
 input A, B, Sel
 output Z;
 wire Z;
 assign #5 Z = Sel ? A : B;
endmodule
```

- Continuous Assignment:
  - Limited to simple operations
    - Simple translation to structural
  - Excellent for CL.

Testing in Verilog

Need separate code to test functional modules (just like C/Java)

- Since hardware is hard to build, major emphasis on testing in HDL
- Testing modules called "test benches" in Verilog; like a bench in a lab dedicated to testing
- Can use time to say how things change

Verilog Testing Idiom

- 1) Instantiate module in testbench
- 2) Write behavioral test cases
  - Stateless: Test all input combos
  - Stateful: Test all transitions from all states
    - Sound hard? It is!
- 3) Compute expected value behaviorally
- 4) Compare expect vs. actual

1) Instantiate module

- Create a test module that instantiates xor:

```
module testxor;
 reg x, y, expected; wire z;
 xor myxor(.x(x), .y(y), .z(z));
 /* add testing code */
endmodule;
```

- Syntax: declare registers, instantiate module.

2 & 3) Generate inputs and expect

- Now we write code to try different inputs by assigning to registers:

```
…
initial
  begin
    x=0; y=0; expected=0;
#10 y=1; expected=1;
#10 x=1; y=0;
#10 y=1; expected=0;
  end
```


4) Testing continued

- Use `$monitor` to watch some signals and see every time they change:

```
…
initial
$monitor(
"x=%b, y=%b, z=%b, exp=%b, time=%d",
x, y, z, expected, $time);
```

- Our code now iterates over all inputs and for each one: prints out the inputs, the gate output, and the expected output.
- `$time` is system function gives current simulation time

4) Testing Output

```
x=0, y=0, z=0, exp=0, time=0
x=0, y=1, z=1, exp=1, time=10
x=1, y=0, z=1, exp=1, time=20
x=1, y=1, z=0, exp=0, time=30
```

- Expected value matches actual value, so Verilog works

Improvements on Testing

- 1) Had to explicitly set inputs by

```
initial
  begin
    x=0; y=0; expected=0;
#10 y=1; expected=1;
#10 x=1; y=0;
#10 y=1; expected=0;
```

- Better:
  - Use counter with #bits = #inputs, iterate through all counter values.
    - Will hit all values

Improvements on Testing

- 2) Had to generate expected for each input combination

```
initial
  begin
    x=0; y=0; expected=0;
#10 y=1; expected=1;
#10 x=1; y=0;
#10 y=1; expected=0;
```

- Better:
  - Generate expected behaviorally to test structural implementation.

Improvements on Testing

- 3) Lot's of output we don't care about:

  x=0, y=0, z=0, exp=0, time=0

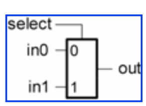
  x=0, y=1, z=1, exp=1, time=10

  x=1, y=0, z=1, exp=1, time=20

  x=1, y=1, z=0, exp=0, time=30

- Better:
  - Print only if actual differs from expected.

Improved Testing Example (1/2)




```
//Test bench for 2-input multiplexor.
// Tests all input combinations.
module testmux2;
 reg [2:0] c;
 wire f, in0, in1, sel;
 reg expected;
 assign in0 = c[0];
 assign in1 = c[1];
 assign sel = c[2];
 mux2 myMux (.select(sel), .in0(in0),
 .in1(in1), .out(f));
 initial
   begin
   $display("Start Test of mux2.");
    #100 c = 3'b000; expected=1'b0; ...
```

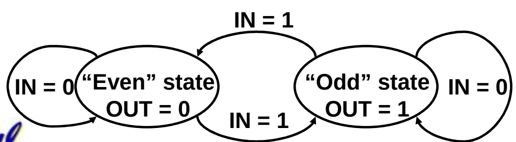

Improved Testing Example (2/2)

```
… begin
    #100 c = 3'b000; expected=1'b0;
    repeat(7)
      begin
        #100
         c = c + 3'b001; // update inputs
          if (sel) expected=in1;
          else expected=in0; // update exp
        #10 // let values settle for delay
          if (expected != f)
           $display("ERROR: Expected %d, got %d.
                  Inputs: sel:%d in0:%d in1:%d",
                  expected, f, sel, in0, in1);
      end
     $display("Finished Test of mux2.");
      #10 $finish;
  end endmodule
```

☹*Can you see the bug?*

Finite State Machines (FSM) (1/10)

- Tutorial example is bit-serial parity checker
  - computes parity of string of bits sequentially, finding the exclusive-or of all the bits
  - Parity of "1" means that the string has an odd number of 1's



FSM (2/10): Review: State Elements

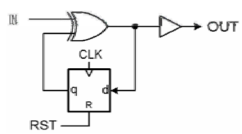
```
//Behavioral model of D-type flip-flop:
// positive edge-triggered,
// synchronous active-high reset.
module DFF (CLK,Q,D,RST);
 input D;
 input CLK, RST;
 output Q;
 reg Q;
 always @ (posedge CLK)
  if (RST) #1 Q = 0; else #1 Q = D;
endmodule // DFF
```

- Loaded on positive edge of clock

FSM (3/10): Structural Implementation

```
//Structural model of serial parity checker.
module parityChecker (OUT, IN, CLK, RST);
 output OUT;
 input IN;
 input CLK, RST;
 wire currentState, nextState;
 DFF state (.CLK(CLK), .Q(currentState),
 .D(nextState), .RST(RST));
 xor (nextState, IN, currentState);
 buf (OUT, currentState);
endmodule // parity
```

Verilog doesn't like it when you feed outputs back internally…



FSM (4/10): Testing

```
//Test-bench for parityChecker.
// Set IN=0. Assert RST. Verify OUT=0.
   // IN=0, RST=1 [OUT=0]
// Keep IN=0. Verify no output change.
   // IN=0, RST=0 [OUT=0]
// Assert IN. Verify output change.
   // IN=1 [OUT=1]
// Set IN=0.Verify no output change.
   // IN=0[OUT=1]
// Assert IN. Verify output change.
   // IN=1 [OUT=0]
// Keep IN=1. Back to ODD.
   // IN=1[OUT=1]
// Assert RST. Verify output change.
   // IN=0, RST=1 [OUT=0]
```

- Comments give what to expect in test

FSM (5/10): Testing

```
...
module testParity0;
 reg IN;
 wire OUT;
 reg CLK=0, RST;
 reg expect;
 parityChecker myParity (OUT, IN, CLK,
 RST);
 always #5 CLK = ~CLK;
...
```

- Note initializing `CLK` in `reg` declaration
- No begin ... end for `always` since only 1 statement

FSM (6/10): Testing

```
initial
 begin
  IN=0; RST=1; expect=0;
  #10 IN=0; RST=0; expect=0;
  #10 IN=1; RST=0; expect=1;
  #10 IN=0; RST=0; expect=1;
  #10 IN=1; RST=0; expect=0;
  #10 IN=1; RST=0; expect=1;
  #10 IN=0; RST=1; expect=0;
  #10 $finish;
 end
initial
$monitor($time," IN=%b, RST=%b, expect=%b
OUT=%b", IN, RST, expect, OUT);
endmodule // testParity0
```

FSM (7/10): Testing Output

```
5 IN=0, RST=1, expect=0 OUT=0
10 IN=0, RST=0, expect=0 OUT=0
20 IN=1, RST=0, expect=1 OUT=0
25 IN=1, RST=0, expect=1 OUT=1
30 IN=0, RST=0, expect=1 OUT=1
40 IN=1, RST=0, expect=0 OUT=1
45 IN=1, RST=0, expect=0 OUT=0
50 IN=1, RST=0, expect=1 OUT=0
55 IN=1, RST=0, expect=1 OUT=1
60 IN=0, RST=1, expect=0 OUT=1
65 IN=0, RST=1, expect=0 OUT=0
```

- Output not well formated; so uses $write, $strobe line up expected, actual outputs

FSM (8/10):Testing: Alternative Approach

```
...
    #10 $finish;
  end
 always
  begin
   $write($time," IN=%b, RST=%b, expect=%b
   ", IN, RST, expect);
   #5 $strobe($time," OUT=%b", OUT);
   #5 ;
  end
endmodule // testParity2
```

- `$write` does not force new line, `$write` and `$strobe` only prints on command, not on every update

FSM (9/10) Testing: New Output

```
0 IN=0, RST=1, expect=0 5 OUT=0
10 IN=0, RST=0, expect=0 15 OUT=0
20 IN=1, RST=0, expect=1 25 OUT=1
30 IN=0, RST=0, expect=1 35 OUT=1
40 IN=1, RST=0, expect=0 45 OUT=0
50 IN=1, RST=0, expect=1 55 OUT=1
60 IN=0, RST=1, expect=0 65 OUT=0
```

- `$write` does not force new line, `$write` and `$strobe` only prints on command, not on every update

FSM (10/10): Lessons

- Testing can be very hard
  - Function of size of state
  - To Test:
    - Test all transitions from all states <or>
    - Prove symmetry of states

Verilog Odds and Ends (1/2)

- Memory is register with second dimension
  `reg [31:0] memArray [0:255];`
- Can assign to group on Left Hand Side
  `{Cout, sum} = A + B + Cin;`
- Can connect logical value 0 or 1 to port via `supply0` or `supply1`
- If you need some temporary variables (e.g., for loops), can declare them as `integer`
- Since variables declared as number of bits, to place a string need to allocate 8 * number of characters

```
    reg [1 : 6*8] Msg;
    Msg = "abcdef";
```


Verilog Odds and Ends (2/2)

```
     ...
     case (select)
      2'b00: expected = a;
      2'b01: expected = b;
      2'b10: expected = c;
      2'b11: expected = d;
       default: expect = a;
     endcase; // case(select)
```

- Verilog case statement different from C
- Has optional `default` case when no match to other cases
- Case very useful in instruction interpretation; case using opcode, each case an instruction