EECS150 - Digital Design
Lecture 7 - Computer Aided Design (CAD) - Part II (Logic Simulation)

Feb 8, 2011

John Wawrzynek
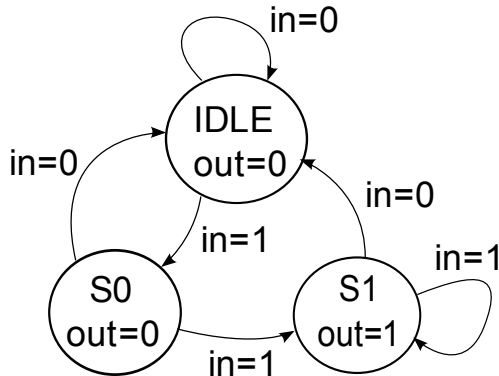
Procedural Assignments

The sequential semantics of the blocking assignment allows variables to be multiply assigned within a single always block. Unexpected behavior can result from mixing these assignments in a single block. Standard rules:

---

i. **Use blocking assignments to model combinational logic within an always block ("=").**

ii. **Use non-blocking assignments to implement sequential logic ("<=").**

iii. **Do not mix blocking and non-blocking assignments in the same always block.**

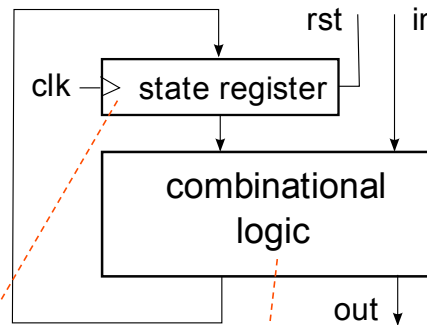iv. **Do not make assignments to the same variable from more than one always block.**

Finite State Machine Review

in=0

IDLE
out=0

in=0

in=0

in=1

S0
out=0

in=1

S1
out=1

in=1

in=1

rst   in

clk → state register

combinational
logic

out ↓

**Holds a symbol to
keep track of which
bubble the FSM is in.**

**CL functions to determine output
value and next state based on input
and current state.**

What does this one do?

Every Synchronous Digital System is a FSM.

out = f(in, current state)

next state = f(in, current state)

Finite State Machines

```
module FSM1(clk, rst, in, out);
input clk, rst;
input in;
output out;
```

**Must use reset to force
to initial state.**

reset not always shown in STD

```
// Defined state encoding:
parameter IDLE = 2'b00;
parameter S0 = 2'b01;
parameter S1 = 2'b10;
reg out;
reg [1:0] state, next_state;
```
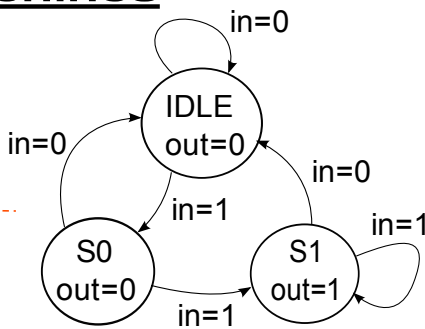
**Constants local
to this module.**

out not a register, but assigned in always block

**Combinational logic
signals for transition.**

THE register to hold the "state" of the FSM.

in=0

IDLE
out=0

in=0

in=0

in=1

S0
out=0

in=1

S1
out=1

in=1

```
// always block for state register
always @(posedge clk)
     if (rst) state <= IDLE;
     else state <= next_state;
```

**A separate always block should be used for combination logic part of FSM. Next
state and output generation. (Always blocks in a design work in parallel.)**

FSM CL block rewritten


```
always @*
 begin
  next_state = IDLE;
  out = 1'b0;
  case (state)
   IDLE   : if (in == 1'b1) next_state = S0;
   S0     : if (in == 1'b1) next_state = S1;
   S1     : begin
             out = 1'b1;
             if (in == 1'b1) next_state = S1;
            end
   default: ;
  endcase
 end
endmodule
```
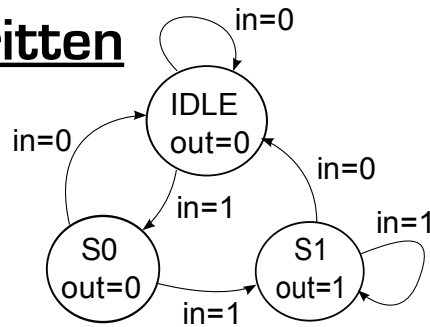
* for sensitivity list

Normal values: used unless specified below.

Within case only need to specify exceptions to the normal values.

Note: The use of "blocking assignments" allow signal values to be "rewritten", simplifying the specification.

Encoder Example

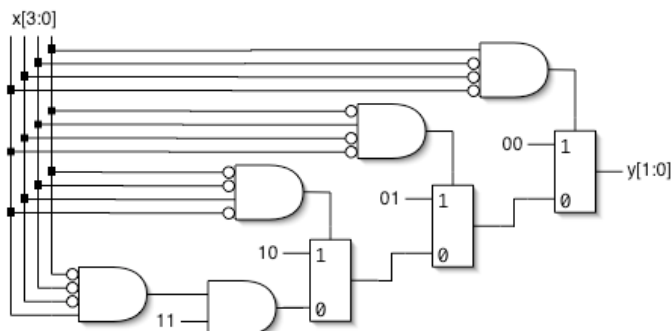Nested IF-ELSE might lead to "priority logic"

Example: 4-to-2 encoder

```
always @(x)
begin : encode
if (x == 4'b0001) y = 2'b00;
else if (x == 4'b0010) y = 2'b01;
else if (x == 4'b0100) y = 2'b10;
else if (x == 4'b1000) y = 2'b11;
else y = 2'bxx;
end
```
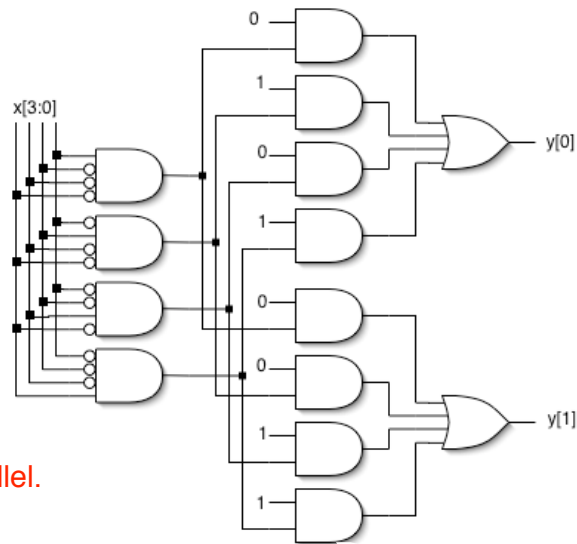
This style of cascaded logic may adversely affect the performance of the circuit.

Encoder Example (cont.)

To avoid "priority logic" use the case construct:

```
always @(x)
begin : encode
case (x)
4'b0001: y = 2'b00;
4'b0010: y = 2'b01;
4'b0100: y = 2'b10;
4'b1000: y = 2'b11;
default: y = 2'bxx;
endcase
end
```
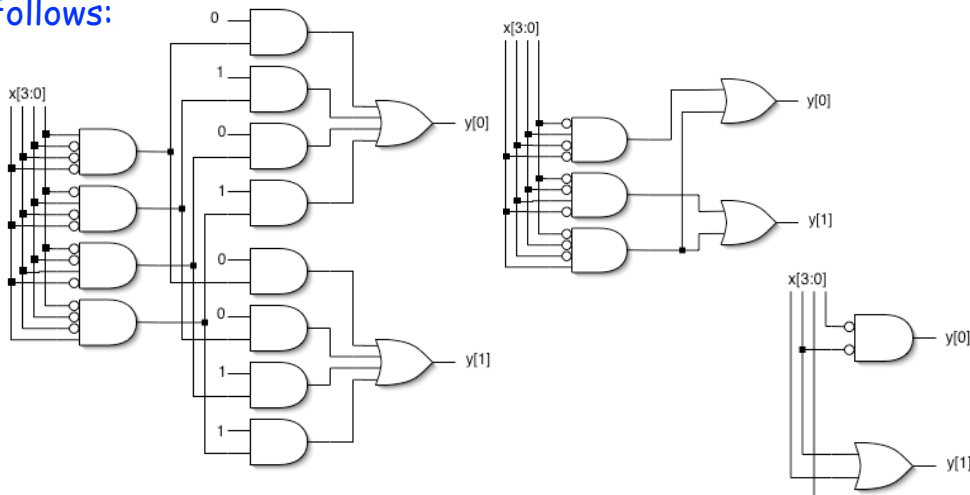

All cases are matched in parallel.

Encoder Example (cont.)

This circuit would be simplified during synthesis to take advantage of constant values and other Boolean equalities as follows:
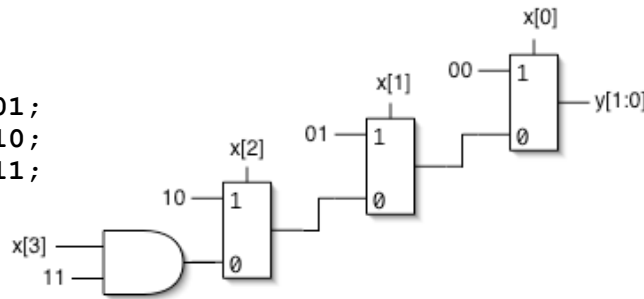

A similar simplification would be applied to the if-else version also.

Encoder Example (cont.)

If you can guarantee that only one 1 appears in the input, then simpler logic can be generated:

```
always @(x)
begin : encode
if (x[0]) y = 2'b00;
else if (x[1]) y = 2'b01;
else if (x[2]) y = 2'b10;
else if (x[3]) y = 2'b11;
else y = 2'bxx;
end
```

If the input applied has more than one 1, then this version functions as a "priority encoder". The least significant 1 gets priority (the more significant 1's are ignored). Again the circuit will be simplified when possible.

Verilog in EECS150

- Primarily use **behavior modeling**. With **instantiation** to build hierarchy and map to FPGA resources not supported by synthesis.
- Primary Style Guidelines:
  - Favor continuous assign and avoid always blocks unless:
    - no other alternative: ex: state elements, case
    - they help readability and clarity of code: ex: large nested if-else-if
  - Use named ports.
  - Separate CL logic specification from state elements.
  - Follow our rules for procedural assignments.
- Verilog is a big language. This is only an introduction.
  - Our text book is a good source. Read and use chapter 4.
  - When needed look at online IEEE Std 1364-2001 document.
  - Be careful of what you read on the web! Many bad examples out there.
  - We will be introducing more useful constructs throughout the semester. Stay tuned!
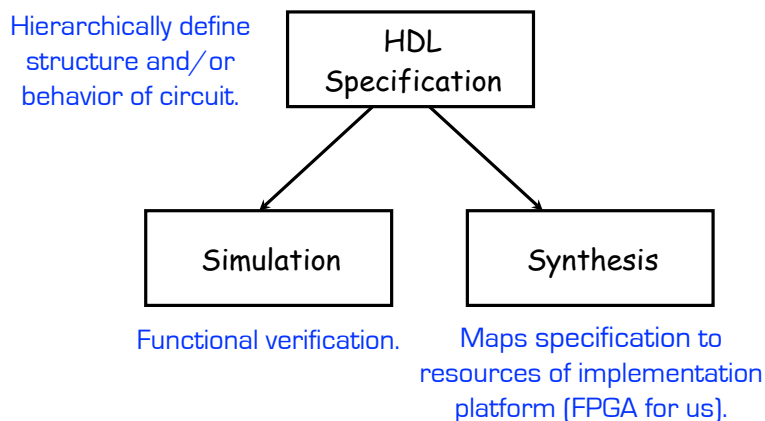
Final thoughts on Verilog Examples

Verilog may look like C, but it describes hardware! (Except in simulation test-benches - which actually behave like programs.)

Multiple physical elements with parallel activities and temporal relationships.

A large part of digital design is knowing how to write Verilog that gets you the desired circuit. <u>First understand the circuit you want, then figure out how to code it in Verilog.</u> If you do one of these activities without the other, you will struggle. These two activities will merge at some point for you.

Be suspicious of the synthesis tools! Check the output of the tools to make sure you get what you want.

EECS150 Design Methodology

Hierarchically define structure and/or behavior of circuit.

```
        HDL
    Specification
```

```
Simulation          Synthesis
```

Functional verification.

Maps specification to resources of implementation platform (FPGA for us).

Let's look at the other branch.

Design Verification

- Industrial design teams spend a large percentage of the design time on design verification:
  - Removing functional bugs, messaging the design to meet performance, cost, and power constraints.
- Particularly important for IC design, less so for FPGAs.
- A variety of tools and strategies are employed.
  - **Simulation**: software that interprets the design description and mimics signal behavior and timing (and power consumption).
    - Simulation provides better controllability and observability over real hardware. Saves on wasted development time and money.
  - **Emulation**: hardware platform (usually FPGAs) are used to mimic behavior of another system. Fast simulation.
  - **Static Analysis**: tools examines circuit structure and reports on expected performance, power, or compares alternative design representations looking for differences.
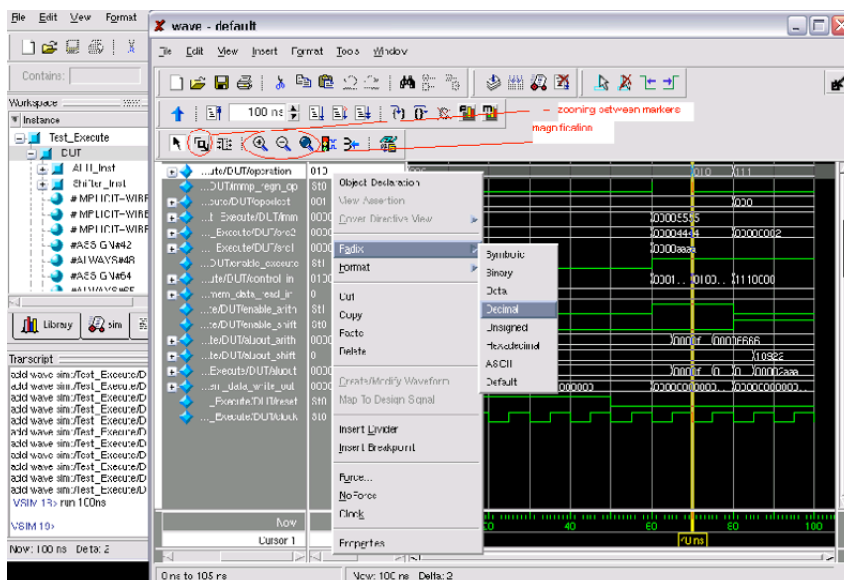
Simulation

Verilog/VHDL simulators use 4 signals values:

0, 1, X (unknown), Z (undriven)

Simulation engine algorithm typically "discrete event simulation"

ModelSim: waveform viewer, GUI.
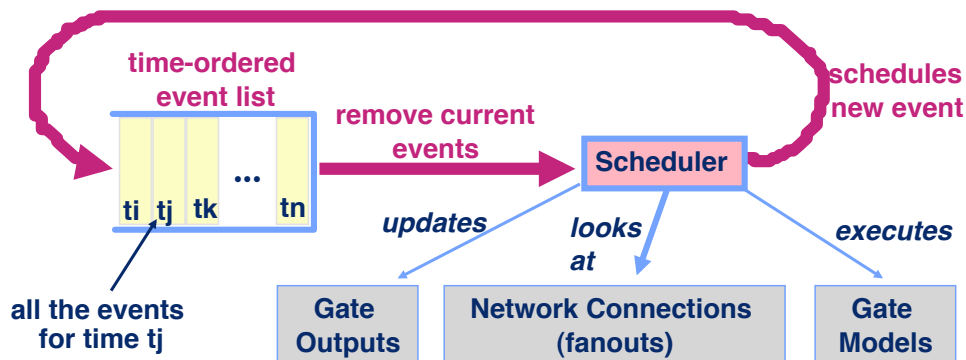
Discrete Event Simulation Engine

- A time-ordered list of events is maintained

  Event: a value-change scheduled to occur at a given time

  All events for a given time are kept together

- The scheduler removes events for a given time ...

  ... propagates values, executes models, and creates new events ...

Slide from Thomas. *The Verilog Hardware Description Language*, By Thomas and Moorby, Kluwer Academic Publishers

Simulation Testing Strategies

- **Unit Testing**: Large systems are often too complex to test all at once, so an bottom-up hierarchical approach. Sub-modules are tested in isolation.

- **Combinational Logic blocks**: when practical, exhaustive testing. Otherwise a combination of random and directed tests.

- **Finite state machines**: test every possible transition and output.

- **Processors**: use software to expose bugs.

- In all cases, the simulated output values are checked against the expected values. Expected values are derived through a variety of means:

  – behavior model running along side the design under test

  – precomputed inputs and outputs (vectors)

  – co-simulation. Ex: C-language model runs along side ModelSim

Testbench

Top-level modules written specifically to test other modules.

```
module testmux;          ----- Generally no ports.
  reg a, b, s;
  wire f;
  reg expected;

  mux2 myMux (.select(s), .in0(a), .in1(b), .out(f));

  initial
      begin
          s=0; a=0; b=1; expected=0;
          #10 a=1; b=0; expected=1;
          #10 s=1; a=0; b=1; expected=1;
      end
  initial
      $monitor(
          "select=%b in0=%b in1=%b out=%b, expected out=%b time=%d",
          s, a, b, f, expected, $time);
  endmodule // testmux
```

Usually never synthesized to circuits. Therefore free to use "simulation only" language constructs.

Instantiation of DUT (device under test).

Initial block similar to "always" block without a trigger. It triggers once automatically at the beginning of simulation. (Also supported on FPGAs).

"#n" used to advance time in simulation. Delays some action by a number of simulation time units.

Note use of blocking assignments.

Note multiple initial blocks.

$monitor triggers whenever any of its inputs change. Sends output to console.

A variety of other "system functions exist for displaying output and controlling the simulation.

Most simulators also include a way to view waveforms of a set of signals.

Mux4 Testbench

```
module testmux4;
  reg [5:0] count = 6'b000000;
  reg a, b, c, d, expected;
  reg [1:0] S;
  wire f;
  mux4 myMux (.select(S), .in0(a), .in1(b), .in2(c), .in3(d), .out(f));
  initial
    begin
      repeat(64)
        begin
          {S, d, c, b, a} = count[5:0];
          case (S)
            2'b00:  expected = a;
            2'b01:  expected = b;
            2'b10:  expected = c;
            2'b11:  expected = d;
          endcase // case(S)
          #8 $strobe( "select=%b in0=%b in1=%b in2=%b in3=%b out=%b,
                      expected=%b time=%d",S, a, b, c, d, f, expected, $time);
          #2 count = count + 1'b1;
        end
      $stop;
    end
endmodule
```

Declaration and initialization all at once. Generally not available in synthesis.

DUT instantiation

Enumerate all possible input patterns.

Apply pattern to DUT

Behavioral model of mux4

$strobe displays data at a selected time. That time is just before simulation time is advanced (after all other events).

Wait a bit, then bump count.

Delay to allow mux outputs to stabilize. Here we assume mux delay < 8ns.

Alternative to $strobe in this case,

**#8 if (f != expected) $display("Mismatch: ...);**

FSM Testbench Example

```
module testFSM;
  reg in;
  wire out;
  reg clk=0, rst;
  reg expect;
  FSM1 myFSM (.out(out), .in(in), .clk(clk), .rst(rst));
  always #5 clk = ~clk;
  initial
    begin
      rst=1;
      #10 in=0; rst=0; expect=0;
      #10 in=1; rst=0; expect=0;
      #10 in=0; rst=0; expect=0;
      #10 in=1; rst=0; expect=0;
      #10 in=1; rst=0; expect=1;
      #10 in=1; rst=0; expect=1;
      #10 in=0; rst=0; expect=0;
      #10 $stop;
    end
  always
    begin
      #4 $strobe($time," in=%b, rst=%b, expect=%b out=%b", in, rst, expect, out);
      #6 ;
    end
endmodule
```

Test all arcs.
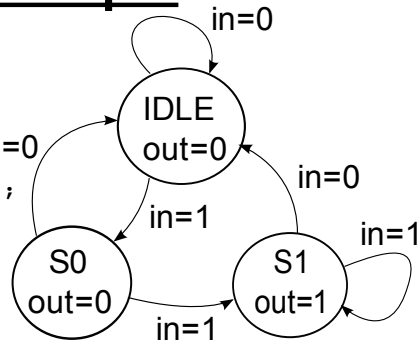
DUT instantiation

100MHz clk signal

start in IDLE
self-loop
transition to S0
transition to IDLE
transition to S0
transition to S1
self-loop
transition to IDLE

Note: Input changes are forced to occur on negative edge of clock.

Strobe output occurs 1ns before rising edge of clock.

Debug is easier if you have access to state value also. Either 1) bring out to ports, or 2) use waveform viewer.

State diagram: IDLE (out=0) with self-loop in=0; IDLE to S0 on in=1; S0 (out=0) to IDLE on in=0; S0 to S1 on in=1; S1 (out=1) self-loop in=1; S1 to IDLE on in=0.

19

Final Words (for now) on Simulation

Testing is not always fun, but you should view it as part of the design process. Untested potentially buggy designs are a dime-a-dozen. Verified designs have real value.

Devising a test strategy is an integral part of the the design process. It shows that you have your head around the design. It should not be an afterthought.