EECS 150 Spring 2003

Lab Lecture 4

Verilog Design Synthesis

2/13/2003

Greg Gibeling
(Original By Sandro Pintz,
Adaptions from John Wawryznek)

Today

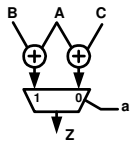
- "Think Hardware"
  - Tips for making EECS150 an easy class
- Simulation
- Blocking vs Non-Blocking
- The Combo Lock
- FSMs in Verilog
- Kramnik!

MOTIVATION

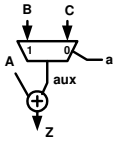
- Finite State Machine Design
- Design Partitioning
- Design Entry
- Synthesis
- Mapping, Placing and Routing

"Think Hardware" (1)

```
always @ (/**/)
      if (a) Z = A+B;
      Else Z = A+C;
```

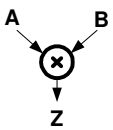



```
always @ (/**/) begin
      if (a) aux = B;
      else aux = C;
      Z = A + aux;
end
```

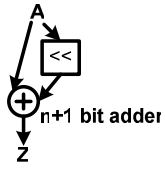


"Think Hardware" (2)

```
assign B = 3;
assign Z = A * B;
```

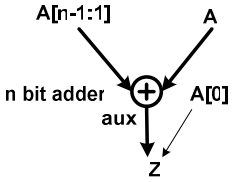



```
assign Z = A + (2 * A);
Or
assign Z = A + (A << 1);
Or
assign Z = A + {A, 1'b0};
```



"Think Hardware" (3)

```
assign aux = {1'b0, A[n-1:1]} + A;
assign Z = {aux, A[0]};
```



Simulation (1)

- Event Driven Simulation
- Order of execution in time tick is not fixed
- Simulator dependent (ouch!)
- Deadlocks can happen in perfectly good design
- **Simulation and Synthesis can differ functionally**

Simulation (2)

- When an event happens put in queue
- When bored get next event
- Figure out the consequences
- This means non-blocking assignments really are executed in any old order, but the results are as-if they were executed in parallel

Administrativia

- Midterm!
  - Thursday 2/19/2004 in class
  - ALL LECTURE MATERIAL COVERED
  - Emphasis on material 2/12 and before
  - TA Review Session
- Monday is a holiday
  - Come to any other lab/discussion

Blocking vs Non-Blocking (1)

```
always @ (b) begin
   a = b;
   c = a;
end
```

- Result
  - c = a = b

```
always @ (posedge Clock) begin
   a <= b;
   c <= a;
end
```

- Result:
  - a = (old?) b
  - c = old a

Blocking vs Non-Blocking (2)

- Use Non-Blocking for FlipFlop Inference:
  - posedge/negedge require nonblocking
  - Otherwise synthesis and simulation will not match
- Use "#1" to visual causality!

```
always @ (posedge Clock) begin
   b <= #1 a;  /* b and c will be flip flops */
   c <= #1 b;
end
```

Blocking vs Non-Blocking (3)

- If you use Blocking for FlipFlops:

You will not get what you want

```
always @ (posedge Clock) begin
   b = a; /* Only c will be a flip flop,  */
   c = b; /* b will go away after synthesis. */
end
/* 'b' is not needed at all */
```

```
always @ (posedge Clock) begin
          c = b; /* c and b will be flip flops */
          b = a;
end
```

Blocking vs Non-Blocking (4)

Race Conditions

file xyz.v:

```
module XYZ(A, B, Clock);
        input   B, Clock;
        output  A;
        reg     A;
        always @ (posedge Clock)
                A = B;
endmodule
```

file abc.v:

```
module ABC(B, C, Clock);
        input   C, Clock;
        output  B;
        reg     B;
        always @ (posedge Clock)
                B = C;
endmodule
```

THIS IS WRONG!!

Blocking vs Non-Blocking (5)

Race Conditions

file xyz.v:

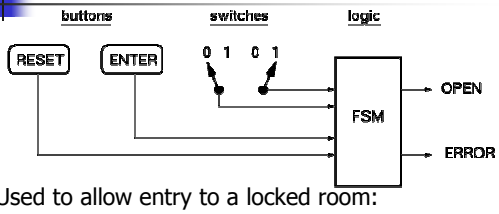
```
module XYZ(A, B, Clock);
        input   B, Clock;
        output  A;
        reg     A;
        always @ (posedge Clock)
                A <= B;
endmodule
```

file abc.v:

```
module ABC(B, C, Clock);
        input   C, Clock;
        output  B;
        reg     B;
        always @ (posedge Clock)
                B <= C;
endmodule
```

THIS IS CORRECT!!

Combination Lock (1)



- Used to allow entry to a locked room:

  2-bit serial combination. Example 01,11:

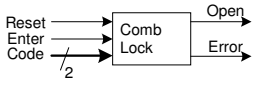
  1. Set switches to 01, press ENTER
  2. Set switches to 11, press ENTER
  3. OPEN is asserted (OPEN=1).

     If wrong code, ERROR is asserted (after second combo word entry).

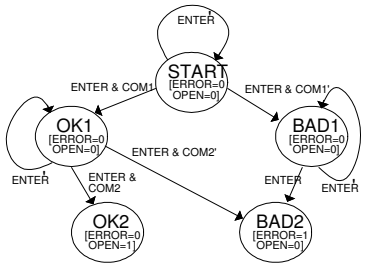
     Press Reset at anytime to try again.

Combination Lock (2)

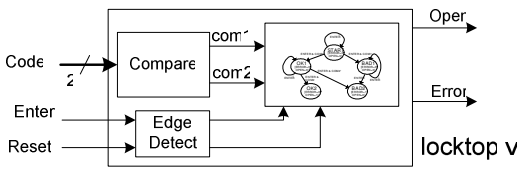
Input Signal	Description
Reset	Clear any entered numbers
Enter	Read the switches (enter a number in the combination)
Code[1:0]	Two binary switches
Output signal	**Description**
Open	Lock opens
Error	Incorrect combination



Combination Lock (3)



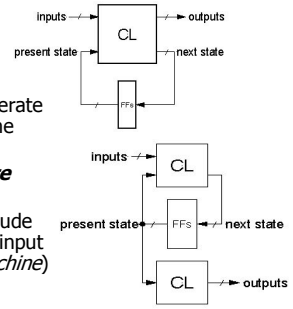
Combination Lock (4)



FSM Implementation Notes

- General FSM form:
- All examples so far generate output based only on the present state:
- Commonly name ***Moore Machine***

  (If output functions include both present state and input then called a *Mealy Machine*)



FSMs in Verilog (1)

- Two always blocks
  - One is **`CurrentState`** register (clocked)
  - Other is combinational
    - Generates **`NextState`**
    - Generates Outputs
- USE MOORE MACHINES
  - Avoid combinational loops

FSMs in Verilog (2)

```
module MyFSM(In, Out, Clock, Reset);
    input       In, Clock, Reset;
    output      Out;

    parameter   IDLE =      1'b0,
                RUNNING =   1'b1;

    reg         CurrentState, NextState, Out;

    always @ (posedge Clock) begin
        if (Reset) CurrentState <=      IDLE;
        else CurrentState <=    NextState;
    end
    …
```

FSMs in Verilog (3)

```
    …
    always @ (CurrentState or In) begin
        NextState =     CurrentState;
        Out =           1'b0;

        // A case block goes here
    end
endmodule
```

FSMs in Verilog (4)

```
case (CurrentState)
    IDLE: begin
        if (In) NextState =     RUNNING;
        Out =                   1'b0;
    end
    RUNNING: begin
        if (In) NextState =     IDLE;
        Out =                   1'b1;
    end
    default: begin
        NextState =             1'bX;
        Out =                   1'bX;
    end
endcase
```

Kramnik

- Windows Terminal Server
- You can log in from home
  - Transfer Files
  - Run Simulations
  - Don't bother with synthesis obviously
- Instructions will be posted
- Server will be available shortly
  - We're still upgrading software