Verilog is the Key Tool

- Behavioral Verilog is synthesized into Structural Verilog
- Structural Verilog represents net-lists
  - From Behavioral
  - From Schematics
  - High-level (Synthesizer will flatten these)
- Verilog is used for testing all designs
  - Behavioral & Structural & Schematic & High-level
  - NC_Verilog, vcs (Synopsys Verilog simulator), modelsim (Mentor Verilog simulator)

Verilog has a Split Personality

- Hardware Description Language (HDL)
  - Reliably & Readably
    - Create hardware
    - Document hardware
  - Testbench creation language
    - Create external test environment
      - Time & Voltage
      - Files & messages
  - Are these two tasks
    - Related?
    - Compatible?

Verilog as HDL

- Want high level modeling
  - unification at all levels
    - from fast functional simulation, accurate device simulation
    - support simulation based validation (verification?)
  - How could we do this?
    - behavioral model mapped to transistors
    - pragmas: throughput, latency, cycle time, power…
  - Reality
    - we rely on designers to do most of these xforms
    - therefore:
      - different algorithms => try before you buy…
      - use only a subset of the language.
      - RTL and schematic design used to support Verilog
      - System-C and other HLD models for co-simulation, etc.

This lecture is only about synthesis...

```verilog
Module name (args...);
begin
  parameter ...; // define parameters
  input ...; // define inputs
  output ...; // define outputs
  wire ...; // internal wires
  reg ...; // internal regs, possibly output
  // the parts of the module body are
  // executed concurrently
<continuous assignments>
<always blocks>
endmodule
```
Quick Review

- Continuous assignments to wire vars
  - assign variable = exp;
  - Results in combinational logic
- Procedural assignment to reg vars
  - Always inside procedural blocks (always blocks in particular for synthesis)
  - blocking
    - variable = exp;
  - non-blocking
    - variable <= exp;
  - Can result in combinational or sequential logic

Verilog Description Styles

- Verilog supports a variety of description styles
  - Structural
    - explicit structure of the circuit
    - e.g., each logic gate instantiated and connected to others
  - Behavioral
    - program describes input/output behavior of circuit
    - many structural implementations could have same behavior
    - e.g., different implementation of one Boolean function

Synthesis: Data Types

- Possible Values (wire and reg):
  - 0: logic 0, false
  - 1: logic 1, true
  - Z: High impedance
- Digital Hardware
  - The domain of Verilog
  - Either logic (gates)
  - Or storage (registers & latches)
- Verilog has two relevant data types
  - wire
  - reg

Parameters

- Used to define constants
  - parameter size = 16, foo = 8;
  - wire [size-1:0] bus; \ defines a 15:0 bus

Synthesis: Assign Statement

- The assign statement creates combinational logic
  - assign LHS = expression;
    - LHS can only be wire type
    - expression can contain either wire or reg type mixed with operators
  - wire a,c; reg b; output out;
    - assign a = b & c;
    - assign out = ~(a & b); \ output as wire
  - wire [15:0] sum, a, b;
    - wire cin, cout;
    - assign {cout,sum} = a + b + cin;
Synthesis: Basic Operators

- **Bit-Wise Logical**
  - \( \neg \) (not), \& (and), \(|\) (or), \(^\) (xor), \(^~\) or \(\neg^\) (xnor)

- **Simple Arithmetic Operators**
  - Binary: +, -
  - Unary: -
  - Negative numbers stored as 2’s complement

- **Relational Operators**
  - <, >, <=, >=, ==, !=

- **Logical Operators**
  - ! (not), && (and), || (or)

```plaintext
assign a = (b > 4'b0110) && (c <= 4'd5);
assign a = (b > 4'b0110) && !c > 4'd5);
```

Synthesis: Operand Length

- Operator length is set to the longest member (both RHS & LHS are considered).

```plaintext
wire [3:0] sum, a, b; wire cin, cout, d, e, f, g;
assign sum = f & a;
assign sum = f | a;
assign sum = {d, e, f, g} & a;
assign sum = {4'b1} | b;
assign sum = {4{f == g}} & (a + b);
assign sum[0] = g & a[2];
assign sum[2:0] = {3{g}} & a[3:1];
```

Synthesis: More Operators

- **Concatenation**
  - \{a,b\} \{4{a==b}\} \{ a,b,4'b1001,\{4{a==b}\} \}

- **Shift (logical shift)**
  - << left shift
  - >> right shift

```plaintext
assign a = b >> 2; // shift right 2, division by 4
assign a = b << 1; // shift left 1, multiply by 2
```

- **Arithmetic**

```plaintext
assign a = b * c; // multiply b times c
assign a = b * 'd2; // multiply b times constant (2)
assign a = b / 'b10; // divide by 2 (constant only)
assign a = b % 'h3; // b modulo 3 (constant only)
```

Synthesis: Extra Operators

- **Funky Conditional**
  - cond_exp ? true_expr : false_expr

```plaintext
wire [3:0] a,b,c; wire d;
assign a = (b == c) ? (c + 'd1): 'o5;
```

- **Reduction Logical**

```plaintext
assign d = &a || ^b ^ ^~c;
```

Synthesis: Assign Statement

- The assign statement is sufficient to create all combinational logic

```plaintext
What about this:
assign a = ~(b & c);
assign c = ~(d & a);
```
The assign statement is sufficient to create all combinational logic.

What about this:
```verilog
assign a = ~(b & c);
assign c = ~(d & a);
```

// Behavioral model of NAND gate
```verilog
module NAND (out, in1, in2);
output out;
input in1, in2;
assign out = ~(in1 & in2);
endmodule
```

// Behavioral model of NAND gate
```verilog
module NAND (out, in1, in2);
output out;
input in1, in2;

// Uses Verilog built-in nand function
// syntax is func id (args);
nand i0(out, in1, in2);
endmodule
```

// Behavioral model of NAND gate
```verilog
module NAND (out, in1, in2);
output out;
input in1, in2;
wire w1;

// call existing modules by name
// module-name ID (signal-list);
AND2X1 u1(w1, in1, in2);
INVX1 u2(out, w1);
endmodule
```

Multiple input gates
```verilog
<gatename> [delay] [id] (out, in1, in2, ...);
```
and, or, nand, nor, xor, xnor

Multiple output gates
```verilog
<gatename> [delay] [id] (out1, out2, ... outn, in);
```
buf, not

Tristate gates
```verilog
<gatename> [delay] [id] (out, in, ctrl);
```
bufif1, bufif0, notif1, notif0

// Structural Module for NAND gate
```verilog
module NAND (out, in1, in2);
output out;
input in1, in2;
wire w1;

// call existing modules by name
// module-name ID (signal-list);
AND2X1 u1(.Q(w1), .A(in1), .B(in2));
INVX1 u2(.A(w1), .Q(out));
endmodule
```
**Primitive Gates**

- Delay: three types for gates
  - `#(delaytime)` same delay for all transitions
  - `#(rise, fall)` different delay for rise and fall
  - `#(rise, fall, turnoff)` for tristate gates
- Each delay number can be:
  - single number i.e. `#(2)` or `#(2,3)`
  - min/typ/max triple i.e. `#(2:3:4)` or `#(2:3:4, 3:4:5)`

**Simple Behavioral Module**

```verbatim
// Behavioral model of NAND gate
module NAND (out, in1, in2);
output out;
input in1, in2;
nand _i0(out, in1, in2);
// include specify block for timing
specify
          (in1 => out) = (1.0, 1.0);
          (in2 => out) = (1.0, 1.0);
     endspecify
endmodule
```

**Specify Block Types**

- Parallel Connection => (one to one)
- Full Connection *> (one to many)

**Parallel Specify**

```verbatim
module A ( q, a, b, c, d )
input a, b, c, d;
output q;
wire e, f;
// specify block containing delay statements
specify
          ( a => q ) = 6; // delay from a to q
          ( b => q ) = 7; // delay from b to q
          ( c => q ) = 7; // delay from c to q
          ( d => q ) = 6; // delay from d to q
     endspecify
// module definition
or o1( e, a, b );
or o2( f, c, d );
exor ex1( q, e, f );
endmodule
```
Full Specify

```
module A (q, a, b, c, d)
input a, b, c, d;
output q;
wire e, f;
// specify block containing full connections
specify
(a, d => q) = 6; // delay from a and d to q
(b, c => q) = 7; // delay from b and c to q
endspecify
// module definition
or o1(e, a, b);
or o2(f, c, d);
exor ex1(q, e, f);
endmodule
```

Specify needs a gate!

```
module DCX1 (CLR, D, CLK, Q);
    input CLR, D, CLK;
    output Q;
    reg Q_i;
    always @(posedge CLK or negedge CLR)
        if (CLR == 0)
            Q_i = 1'b0;
        else
            Q_i = D;
    buf _i0(Q, Q_i);
    specify
        (CLK => Q) = (1.0, 1.0);
        (CLR => Q) = (1.0, 1.0);
    $setuphold(posedge CLK, D, 0.1, 0.0);
    $recovery(negedge CLR, posedge CLK, 0.0);
endspecify
endmodule
```

Procedural Assignment

- Assigns values to register types
- They involve data storage
  - The register holds the value until the next procedural assignment to that variable
- The occur only within procedural blocks
  - initial and always
  - initial is NOT supported for synthesis!
- They are triggered when the flow of execution reaches them

Simple Behavioral Module

```
// Behavioral model of NAND gate
module NAND (out, in1, in2);
    output out;
    input in1, in2;
    nand _i0(out, in1, in2);
    // include specify block for timing
    specify
        (in1 => out) = (1.0, 1.0);
        (in2 => out) = (1.0, 1.0);
    endspecify
endmodule
```

Always Blocks

- When is an always block executed?
  - always
    - Starts at time 0
  - always @(a or b or c)
    - Whenever there is a change on a, b, or c
    - Used to describe combinational logic
  - always @(posedge foo)
    - Whenever foo goes from low to high
    - Used to describe sequential logic
  - always @(negedge bar)
    - Whenever bar goes from high to low
# Synthesis: Always Statement

- The always statement creates...
  - `always @sensitivity
    LHS = expression;
  - `sensitivity controls when LHS can only be reg type
  - expression can contain either wire or reg type mixed with operators

- Logic
  - `reg c, b; wire a;
    always @(a, b) c = ~(a & b);
    always @* c = ~(a & b);

- Storage
  - `reg Q; wire clk;
    always @(posedge clk) Q <= D;

## Procedural Control Statements

- Conditional Statement
  - `if ( <expression> ) <statement>
  - `if ( <expression> ) <statement> else <statement>
    - “else” is always associated with the closest previous if that lacks an else.
    - You can use begin-end blocks to make it more clear
  - `if (index >0)
    if (rega > regb)
      result = rega;
    else result = regb;

## Multi-Way Decisions

- Standard if-else-if syntax
  - `if ( <expression> )
    <statement>
  - `else if ( <expression> )
    <statement>
  - `else if ( <expression> )
    <statement>
  - `else <statement>

## Procedural NAND gate

- // Procedural model of NAND gate
  - `module NAND (out, in1, in2);
    output out;
    reg out;
    input in1, in2;
    // always executes when in1 or in2 change value
    always @(in1 or in2)
    begin
      out <= ~(in1 & in2);
    end
  endmodule

## Synthesis: NAND gate

- // Procedural model of NAND gate
  - `input in1, in2;
    reg n1, n2; // is this a flip-flop?
    wire n3, n4;
    always @(in1 or in2) n1 = ~(in1 & in2);
    always @* n2 = ~(in1 & in2);
    assign n3 = ~(in1 & in2);
    nand u1(n4, in1, in2);
  // Notice always block for combinational logic
  - Full sensitivity list, but `@* works (2001 syntax)
  - Can then use the always goodies
  - Is this a good coding style?
Procedural Assignments
- Assigns values to reg types
  - Only usable inside a procedural block Usually synthesizes to a register
    - But, under the right conditions, can also result in combinational circuits
- Blocking procedural assignment
  - LHS = timing-control exp \( a = #10 \ 1; \)
  - Must be executed before any assignments that follow (timing control is optional)
  - Assignments proceed in order even if no timing is given
- Non-Blocking procedural assignment
  - LHS <= timing-control exp \( b <= 2; \)
  - Evaluated simultaneously when block starts
  - Assignment occurs at the end of the (optional) time-control

Procedural Synthesis
- Synthesis ignores all that timing stuff
- So, what does it mean to have blocking vs. non-blocking assignment for synthesis? How does this relate to reg?

```
begin
  A=B;
  B=A;
end
```

```
begin
  A=\( Y; \)
  B=A;
end
```

```
begin
  B=\( A; \)
  A=\( Y; \)
end
```

Synthesized Circuits
```
begin
  A = \( Y; \)
  B = A;
end
```

```
begin
  A <= \( Y; \)
  B <= A;
end
```

```
begin
  B = \( A; \)
  A = \( Y; \)
end
```

Assignments and Synthesis
- Note that different circuit structures result from different types of procedural assignments
  - Therefore you can’t mix assignment types in the same always block
  - Non-blocking is often a better model for hardware
    - Real hardware is often concurrent…
  - Blocking is often better for setting subsets of signals
    - Set them to defaults at beginning, then reset only the ones that change

Comparator Example
- Using continuous assignment
  - Concurrent execution of assignments

Module comp (a, b, Cgt, Clt, Cne);
parameter n = 4;
input [n-1:0] a, b;
output Cgt, Clt, Cne;
assign Cgt = (a > b);
assign Clt = (a < b);
assign Cne = (a != b);
endmodule
Comparator Example

- Using procedural assignment
  - Non-blocking assignment implies concurrent

```
Module comp (a, b, Cgt, Clt, Cne);
  parameter n = 4;
  input [n-1:0] a, b;
  output Cgt, Clt, Cne;
  reg Cgt, Clt, Cne;
  always @(a or b)
      begin
          Cgt  <= (a > b);
          Clt   <= (a < b);
          Cne <= (a != b);
      end
endmodule
```

Modeling a Flip Flop

- Use an always block to wait for clock edge

```
Module dff (clk, d, q);
  input clk, d;
  output q;
  reg q;
  always @(posedge clk)
      d = q;
endmodule
```

Synthesis: Always Statement

- This is a simple D Flip-Flop
  - `reg Q;` always @(posedge clk) Q <= D;`
  - `(posedge clk)` is the sensitivity list
  - The Q <= D; is the block part
  - The block part is always “entered” whenever the sensitivity list becomes true (positive edge of clk)
  - The LHS of the <= must be of data type `reg`
  - The RHS of the <= may use `reg` or `wire`

```
reg Q;
always @(posedge clk, posedge rst)
    if (rst) Q <= 'b0; else Q <= D;
```

- Notice , instead of or
  - Verilog 2001…
  - Positive reset

```
reg Q;
always @(posedge clk, posedge rst, posedge set)
    if (rst) Q <= 'b0;
    else if (set) Q <= 'b1;
    else Q <= D;
```

- What is this?
  - What is synthesized?

> beh2str foo.v foo_str.v UofU_Digital.db
Always Statement

reg Q;
always @(posedge clk, posedge rst, posedge set)
   if (rst) Q <= 'b0;
   else if (set) Q <= 'b1;
   else Q <= D;

What is this?

What is synthesized?

Inferred memory devices in process
in routine set line 5 in file
'/home/elb/IC_CAD/syn-f06/set.v'.

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Type</th>
<th>Width</th>
<th>Bus</th>
<th>MB</th>
<th>AR</th>
<th>AS</th>
<th>SR</th>
<th>SS</th>
<th>ST</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q_reg</td>
<td>Flip-flop</td>
<td>1</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

module foo ( clk, rst, set, D, Q );
input clk, rst, set, D;
output Q;
wire   N3, n2, n4;
dff Q_reg ( .D(N3), .G(clk), .CLR(n2), .Q(Q) );
tiehi U6 ( .Y(n2) );
nor2 U7 ( .A(rst), .B(n4), .Y(N3) );
nor2 U8 ( .A(D), .B(set), .Y(n4) );
endmodule
```verilog
module testme ( D, P, Q, R, clk );
output [3:0] R;
input D, clk;
output P, Q;
wire N0, N1, N2, N3, n1, n7, n8, n9;
dff Q_reg ( .D(D), .G(clk), .CLR(n1), .Q(Q) );
dff P_reg ( .D(Q), .G(clk), .CLR(n1), .Q(P) );
dff R_reg_0_ ( .D(N0), .G(clk), .CLR(n1), .Q(R[0]) );
dff R_reg_1_ ( .D(N1), .G(clk), .CLR(n1), .Q(R[1]) );
dff R_reg_2_ ( .D(N2), .G(clk), .CLR(n1), .Q(R[2]) );
dff R_reg_3_ ( .D(N3), .G(clk), .CLR(n1), .Q(R[3]) );
tiehi U9 ( .Y(n1) );
xor2 U10 ( .A(R[3]), .B(n7), .Y(N3) );
nor2 U11 ( .A(n8), .B(n9), .Y(n7) );
xor2 U12 ( .A(n8), .B(n9), .Y(N2) );
invX1 U13 ( .A(R[2]), .Y(n9) );
nand2 U14 ( .A(R[1]), .B(R[0]), .Y(n8) );
xor2 U15 ( .A(R[1]), .B(R[0]), .Y(N1) );
invX1 U16 ( .A(R[0]), .Y(N0) );
endmodule
```

### Constants
- `parameter` used to define constants
  - `parameter size = 16, foo = 8;`
  - `wire [size-1:0] bus; \\ \defines a 15:0 bus`
  - `externally modifiable`
  - `scope is local to module`
- `localparam` not externally modifiable
  - `localparam width = size * foo;`
- `define` macro definition
  - `define value 7'd53`
  - `assign a = (sel == \value) \& b;`
  - `scope is from here on out`

### Example: Counter
```verilog
module counter (clk, clr, load, in, count);
parameter width=8;
input clk, clr, load;
input [width-1 : 0] in;
output [width-1 : 0] count;
reg [width-1 : 0] tmp;
always @(posedge clk or negedge clr)
begin
if (!clr)
  tmp = 0;
else if (load)
  tmp = in;
else
  tmp = tmp + 1;
end
assign count = tmp;
endmodule
```

### Synthesis: Modules
- `// Verilog 1995 syntax`
  ```verilog
  module adder (e,f,g);
  parameter SIZE=2;
  input [SIZE-1:0] e, f;
  output [SIZE-1:0] g;
  assign g = e + f;
  endmodule
  ```
- `// Verilog 2001 syntax`
  ```verilog
  module subber #(parameter SIZE = 3)
  (input 
  [SIZE-1:0] c,d, output [SIZE-1:0]difference);
  assign difference = c - d;
  endmodule
  ```
module the_top (clk, rst, a, b, sel, result);
  parameter SIZE = 4;
  input clk, rst;
  input [SIZE-1:0] a,b;
  input [2:0] sel;
  output reg [SIZE-1:0] result;
  wire [SIZE-1:0] sum, dif, alu;

  adder #(SIZE) u0(a,b,sum);
  subber #(4) u1(.c(a), .d(b), .difference(dif));

  assign alu = {SIZE{sel == 'b000}} & sum | {SIZE{sel == 'b001}} & dif;

always @(posedge clk or posedge rst)
  if (rst) result <= 'h0;
  else  result <= alu;
endmodule

module priority (a, b, c, d, sel, z);
  input a,b,c,d;
  input [3:0] sel;
  output z;
  reg z;
  always @(a or b or c or d or sel)
    begin
      z = 0;
      if (sel[0]) z = a;
      if (sel[1]) z = b;
      if (sel[2]) z = c;
      if (sel[3]) z = d;
    end
endmodule

module parallel (a, b, c, d, sel, z);
  input a,b,c,d;
  input [3:0] sel;
  output z;
  reg z;
  always @(a or b or c or d or sel)
    begin
      z = 0;
      if (sel[3]) z = d;
      else if (sel[2]) z = c;
      else if (sel[1]) z = b;
      else if (sel[0]) z = a;
    end
endmodule
Case Statements

- Multi-way decision on a single expression

```verilog
case ( <expression> )
  <expression>: <statement>
  <expression>, <expression>: <statement>
  <expression>: <statement>
  default: <statement>
endcase
```

Case Example

```verilog
reg [1:0] sel;
reg [15:0] in0, in1, in2, in3, out;
case (sel)
  2'b00: out = in0;
  2'b01: out = in1;
  2'b10: out = in2;
  2'b11: out = in3;
endcase
```

Another Case Example

```verilog
// simple counter next-state logic
// one-hot state encoding...
parameter [2:0] s0=3'h1, s1=3'h2, s2=3'h4;
reg[2:0] state, next_state;
always @(input or state)
begin
  case (state)
    s0: if (input) next_state = s1;
    else next_state = s0;
    s1: next_state = s2;
    s2: next_state = s0;
  endcase
end
```

Weird Case Example

```verilog
Verilog allows you to put a value in the case slot, and test which variable currently has that value...

reg [ 2:0] curr_state, next_state;
parameter s1=3'b001, s2=3'b010, s3=3'b100
case (1)
  curr_state[0] : next_state = s2;
  curr_state[1] : next_state = s3;
  curr_state[2] : next_state = s1;
endcase
```

Latch Inference

- Incompletely specified `if` and `case` statements cause the synthesizer to infer latches

```verilog
always @(cond)
begin
  if (cond) data_out <= data_in;
end
```

- This infers a latch because it doesn’t specify what to do when `cond = 0`
  - Fix by adding an `else` if you want combinational logic
  - In a case, fix by including `default:

Full vs. Parallel

- Case statements check each case in sequence
- A `case` statement is `full` if all possible outcomes are accounted for
- A `case` statement is `parallel` if the stated alternatives are mutually exclusive
- These distinctions make a difference in how cases are translated to circuits...
  - Similar to the `if` statements previously described
Case full-par example

// full and parallel = combinational logic
module full-par (slct, a, b, c, d, out);
  input [1:0] slct;
  input a, b, c, d;
  output out;
  reg out; // optimized away in this example
  always @(slct or a or b or c or d)
    case (slct)
      2'b11 : out <= a;
      2'b10 : out <= b;
      2'b01 : out <= c;
      default : out <= d; // really 2'b10
    endcase
endmodule

Case notfull-par example

// a latch is synthesized because case is not full
module notfull-par (slct, a, b, c, d, out);
  input [1:0] slct;
  input a, b, c, d;
  output out;
  reg out; // NOT optimized away in this example
  always @(slct or a or b or c)
    case (slct)
      2'b11 : out <= a;
      2'b10 : out <= b;
      2'b01 : out <= c;
    endcase
endmodule

Case full-notpar example

// because case is not parallel - priority encoding
// but it is still full, so no latch...
// this uses a casez which treats ? as don't-care
module full-notpar (slct, a, b, c, out);
  always @(slct or a or b or c)
    casez (slct)
      2'b1? : out <= a;
      2'b?1 : out <= b;
      default : out <= c;
    endcase
endmodule

Synthesis Result

Note that full-par results in combinational logic

Synthesized Circuit

Because it's not full, a latch is inferred...

Synthesized Circuit

It's full, so it's combinational, but it's not parallel so it's a priority circuit instead of a “check all in parallel” circuit
Case not full - not parallel example

// because case is not parallel - priority encoding
// because case is not full - latch is inferred
// uses a casez which treats ? as don't-care
module full-notpar (slct, a, b, c, out);
...
always @(slct or a or b or c)
casez (slct)
2'b1? : out <= a;
2'b?1 : out <= b;
endcase
endmodule

Get off my Case

- Verification
  - CASE matches all (works like ===)
  - CASEX uses “z”, “x”, “?” as don't care
  - CASEZ uses “z”, “?” as don't care
  - Beware: Matches first valid case

Synthesis
- CASE works like ==
- CASEX uses “?” as don't care
- CASEZ uses “?” as don't care

FSM Description

- One simple way: break it up like a schematic
  - A combinatorial block for next_state generation
  - A combinatorial block for output generation
  - A sequential block to store the current state

Get off my Case

- We've only just begun
  - Driving add
  - CASEX. Opcode: add
  - Driving subtract
  - CASEX. Opcode: subtract
  - Driving multiply
  - CASEX. Opcode: multiply

Order Matters
// General view
module FSM (clk, in, out);
input clk, in;
output out;
reg out;
// state variables
reg [1:0] state;
// next state variable
reg [1:0] next_state;
always @posedge(clk) // state register
    state = next_state;
always @(state or in) // next-state logic
    // compute next state and output logic
    // make sure every local variable has an
    // assignment in this block
endmodule

module moore (clk, clr, insig, outsig);
input clk, clr, insig;
output outsig;
// define state encodings as
parameters
    parameter [1:0] s0 = 2'b00,
s1 = 2'b01, s2 = 2'b10, s3 = 2'b11;
// define reg vars for state register
// and next_state logic
    reg [1:0] state, next_state;
//define state register (with
//synchronous active-high clear)
always @(posedge clk)
begin
    if (clr) state = s0;
    else state = next_state;
end
// define combinational logic for
// next_state
always @(insig or state)
begin
    case (state)
        s0: if (insig) next_state = s1;
            else next_state = s0;
        s1: if (insig) next_state = s2;
            else next_state = s1;
        s2: if (insig) next_state = s3;
            else next_state = s2;
        s3: if (insig) next_state = s1;
            else next_state = s0;
    endcase
end
// now set the outsig. This could also be done in an always
// block... but in that case, outsig would have to be
// defined as a reg.
assign outsig = ((state == s1) || (state == s3));
endmodule

// define combinational logic for next_state
always @(insig or state)
begin
    case (state)
        s0: if (insig) next_state = s1;
            else next_state = s0;
        s1: if (insig) next_state = s2;
            else next_state = s1;
        s2: if (insig) next_state = s3;
            else next_state = s2;
        s3: if (insig) next_state = s1;
            else next_state = s0;
    endcase
end
Unsupported for Synthesis

- Delay (Synopsys will ignore #'s)
- initial blocks (use explicit resets)
- repeat
- wait
- fork
- event
- deassign
- force
- release

More Unsupported Stuff

- You cannot assign the same reg variable in more than one procedural block

// don't do this…
always @(posedge a)
out = in1;
always @(posedge b)
out = in2;

Combinational Always Blocks

- Be careful…
always @(sel) always @(sel or in1 or in2)
  if (sel == 1)
    out = in1;
  else out = in2;
- Which one is a good mux?

Sync vs. Async Register Reset

// synchronous reset (active-high reset)
always @(posedge clk)
  if (reset) state = s0;
  else state = s1;

// async reset (active-low reset)
always @(posedge clk or negedge reset)
  if (reset == 0) state = s0;
  else state = s1;

Finite State Machine

Four in a Row
Textbook FSM

module wave (clk, cin, csig, swav);
input clk, cin, csig;
output swav;

// define state encoding as parameters
parameter [2:0] s0 = 3'd000; // initial state, start at least 1 zero
parameter [2:0] s1 = 3'd001; // start
parameter [2:0] s2 = 3'd010; // start
parameter [2:0] w = 3'd011; // wave 2 ones
parameter [2:0] s3 = 3'd100; // wave 3 ones
parameter [2:0] s4 = 3'd101; // wave at least 4 ones
reg [2:0] state, next_state;
always @ (posedge clk orposedge csig) // state register
begin
  if (clknr != s0) state = s0;
else state = next_state;
end
always @ (posedge cin orposedge state) // next state logic
begin
  case (state)
    0: if (cin) next_state = s1;
    else next_state = s0;
    1: if (cin) next_state = s2;
    else next_state = s1;
    2: if (cin) next_state = s3;
    else next_state = s2;
    3: if (cin) next_state = s4;
    else next_state = s3;
    default: next_state = s0;
  endcase
end
assign swav = state == w1;
endmodule

Documented FSM

module wave (clk, cin, csig, swav);
input clk, cin, csig;
output swav;

// define state encoding as parameters
parameter [2:0] s0 = 3'd000; // initial state, start at least 1 zero
parameter [2:0] s1 = 3'd001; // start
parameter [2:0] s2 = 3'd010; // start
parameter [2:0] w = 3'd011; // wave 2 ones
parameter [2:0] s3 = 3'd100; // wave 3 ones
parameter [2:0] s4 = 3'd101; // wave at least 4 ones
reg [2:0] state, next_state;
always @ (posedge clk orposedge csig) // state register
begin
  if (clknr != s0) state = s0;
else state = next_state;
end
always @ (posedge cin orposedge state) // next state logic
begin
  case (state)
    0: if (cin) next_state = s1;
    else next_state = s0;
    1: if (cin) next_state = s2;
    else next_state = s1;
    2: if (cin) next_state = s3;
    else next_state = s2;
    3: if (cin) next_state = s4;
    else next_state = s3;
    default: next_state = s0;
  endcase
end
assign swav = state == w1;
endmodule

Waveform Test Bench

// Four in a row tester
// Allen Tennant

initial
begin
  clk = 1'b0;
  cin = 1'b0;
  csig = 1'b1;
  start = 1'b0;
  $monitor(33'h001_1000_1010_1111_0000_0111_1110_0000);
  $monitor(33'h001_1000_1010_1111_0000_0111_1110_0000);
  $finish;
end

always @ (posedge cin) // start
begin
  cin = 1'b1;
  $monitor(33'h001_1000_1010_1111_0000_0111_1110_0000);
end

task print_message;
  input [1:0] pattern;
  integer i;
begin
  for(i=0; i<2; i++) begin
    #10
    $display(“pattern[“+inttostr(i)+”]= “+inttostr(pattern[i]));
  end
endtask

endtask // print_message

endmodule

Waveform

Pay attention to first few cycles...
One-Hot FSM

module one-hot_fsm #(clk, clrst, en, sel) (state, out);
    input clk, clrst, en;
    input sel;
    output reg state;
    output [3:0] out;
    assign state = (sel)? 3'b000 : 3'b001 : 3'b100 : 3'b101;
    assign out = state;
endmodule

One-Hot FSM Counting

module one-hot_fsm_counting #(clk, clrst, en, sel) (state, out);
    input clk, clrst, en;
    input sel;
    output reg state;
    output [3:0] out;
    always @(posedge clk or posedge clrst) begin
        if (en) begin
            state <= 0;
            out <= 0;
        end
        else begin
            state <= (state + 1'b1) % 4;
            out <= state;
        end
    end
endmodule

Oops

module one-hot_fsm #(clk, clrst, en, sel) (state, out);
    input clk, clrst, en;
    input sel;
    output reg state;
    output [3:0] out;
endmodule

No Asynchronous Sets

module no_asynchronous_sets #(clk, clrst, en, sel) (state, out);
    input clk, clrst, en;
    input sel;
    output reg state;
    output [3:0] out;
    assign state = 0;
endmodule
That's better

That's better

Synchronous Clear

Synchronous Clear

ROM vs. Verilog

ROM vs. Verilog

Synchronous Clear

Synchronous Clear

Is asynchronous clear really asynchronous?

What about set-up & hold with respect to clock edge?

ROM vs. Verilog

ROM vs. Verilog