

- ### Verilog is the Key Tool
- ▶ Behavioral Verilog is synthesized into Structural Verilog
  - ▶ Structural Verilog represents net-lists
    - ▶ From Behavioral
    - ▶ From Schematics
    - ▶ From makeMem
    - ▶ High-level (Synthesizer will flatten these)
  - ▶ Verilog-XL is used for testing all designs
    - ▶ Behavioral & Structural & Schematic & High-level

- ### Verilog has a Split Personality
- ▶ Hardware Description Language (HDL)
    - ▶ Reliably & Readably
      - ▶ Create hardware
      - ▶ Document hardware
    - ▶ The wire-list function fits into HDL
  - ▶ Testbench creation language
    - ▶ Create external test environment
      - ▶ Time & Voltage
      - ▶ Files & messages
  - ▶ Are these two tasks
    - ▶ Related?
    - ▶ Compatible?

- ### Verilog as HDL (AHT)
- ▶ “C-like hardware description language.”
    - ▶ But what does C have to do with hardware?
    - ▶ Marketing hype cast into vital tools
  - ▶ Verilog is ill-suited to its use.
    - ▶ Verbose
    - ▶ Feels to me like I am “tricking it”
  - ▶ Good engineers
    - ▶ Use only a subset of the language.
    - ▶ Keep Learning.
    - ▶ Try before they buy.
    - ▶ Demo today.

### Synthesis

This lecture is only about synthesis...

### Quick Review

```

Module name (args...);
begin
  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;`
  - ▶ Result 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

## 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>
  
```

## 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:
  - ▶ **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

## Synthesis: Data Types

- ▶ Register declarations
  - ▶ `reg a;` \\ a scalar register
  - ▶ `reg [3:0] b;` \\ a 4-bit vector register
  - ▶ `output g;` \\ an output can be a reg reg g;
  - ▶ `output reg g;` \\ Verilog 2001 syntax
- ▶ Wire declarations
  - ▶ `wire d;` \\ a scalar wire
  - ▶ `wire [3:0] e;` \\ a 4-bit vector wire
  - ▶ `output f;` \\ an output can be a wire

## 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
  - ▶ `~` (not), `&` (and), `|` (or), `^` (xor), `^~` or `~^` (xnor)
- ▶ Simple Arithmetic Operators
  - ▶ Binary: `+`, `-`
  - ▶ Unary: `-`
  - ▶ Negative numbers stored as 2's complement
- ▶ Relational Operators
  - ▶ `<`, `>`, `<=`, `>=`, `==`, `!=`
- ▶ Logical Operators
  - ▶ `!` (not), `&&` (and), `||` (or)
  - `assign a = (b > 'b0110) && (c <= 4'd5);`
  - `assign a = (b > 'b0110) && !(c > 4'd5);`

## Synthesis: Operand Length

- ▶ When operands are of unequal bit length, the shorter operand is zero-filled in the most significant bit position
  - `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{f}} | 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
  - `assign a = b >> 2; // shift right 2, division by 4`
  - `assign a = b << 1; // shift left 1, multiply by 2`
- ▶ Arithmetic
  - `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: Operand Length

- ▶ Operator length is set to the longest member (both RHS & LHS are considered). Be careful.
  - `wire [3:0] sum, a, b; wire cin, cout, d, e, f, g;`  
`wire[4:0]sum1;`
  - `assign {cout,sum} = a + b + cin;`  
`assign {cout,sum} = a + b + {4'b0,cin};`
  - `assign sum1 = a + b;`  
`assign sum = (a + b) >> 1; // what is wrong?`

## Synthesis: Extra Operators

- ▶ Funky Conditional
  - ▶ `cond_exp ? true_expr : false_expr`
  - `wire [3:0] a,b,c; wire d;`
  - `assign a = (b == c) ? (c + 'd1): 'o5; // good luck`
- ▶ Reduction Logical
  - ▶ Named for impact on your recreational time
  - ▶ Unary operators that perform bit-wise operations on a single operand, reduce it to one bit
  - ▶ `&, ~&, |, ~|, ^, ~^, ^^`
  - `assign d = &a || ~^b ^^~c;`

## Synthesis: Assign Statement

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

## Simple Behavioral Module

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

## Simple Structural Module

```
// Structural Module for NAND gate
module NAND (out, in1, in2);
  output out;
  input in1, in2;
  wire w1;
  // call existing modules by name
  // module-name ID (signal-list);
  AND2 u1(w1, in1, in2);
  INV u2(out,w1);
endmodule
```

## Simple Structural Module

```
// Structural Module for NAND gate
module NAND (out, in1, in2);
  output out;
  input in1, in2;
  wire w1;
  // call existing modules by name
  // module-name ID (signal-list);
  // can connect ports by name...
  AND2 u1(.Q(w1), .A(in1), .B(in2));
  INV u2(.A(w1), .Q(out));
endmodule
```

## Procedural Assignment

- ▶ Assigns values to **register** types
- ▶ They do not have a duration
  - ▶ The register holds the value until the next procedural assignment to that variable
- ▶ They occur only within procedural blocks
  - ▶ **initial** and **always**
  - ▶ **initial is NOT supported for synthesis!**
- ▶ They are triggered when the flow of execution reaches them

## 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 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
```

## 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
```

*Is out combinational?*

## Synthesis: 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
  - ▶ Can then use the always goodies
  - ▶ Is this a good coding style?

## Procedural Assignments

- ▶ Assigns values to reg types
  - ▶ Only useable 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?

```

▶ begin
  A=B;
  B=A;
end
    
```

?

```

▶ begin
  A<=B;
  B<=A;
end
    
```

---

```

▶ begin
  A=Y;
  B=A;
end
    
```

?

```

▶ begin
  A<=Y;
  B<=A;
end
    
```

## Synthesized Circuits

```

▶ begin
  A = Y;
  B = A;
end
    
```

```

▶ begin
  A <= Y;
  B <= A;
end
    
```

## Synthesized Circuits

```

always @(posedge clk)
begin
  A = Y;
  B = A;
end
    
```

```

always @(posedge clk)
begin
  B = A;
  A = Y;
end
    
```

```

always @(posedge clk)
begin
  A <= Y;
  B <= A;
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
- ▶ And you can't use different assignment types to assign the same register either
- ▶ Non-blocking is often a better model for hardware
  - ▶ Real hardware is often concurrent...

## 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)
  Cgt <= (a > b);
  Clt <= (a < b);
  Cne <= (a != b);
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`

### Synthesis: Always Statement

- ▶ This is an asynchronous clear D Flip-Flop

```

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

```

- ▶ Notice `,` instead of `or`
- ▶ Verilog 2001...
- ▶ Positive reset (how does the edge play?)

### Synthesis: 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?

```

syn-f06> beh2str foo.v foo_str.v UofU_Digital.db

```

### Synthesis: 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?

```

/home/atanner/IC_CRUI/synopsys/nt.v'.
-----
| Register Name | Type | Width | Bus | MB | AR | AS | SR | SS | ST |
-----
| rout_reg     | Flip-flop | 1 | IN IN IN IY IY IN IN IN |
-----
Presto compilation completed successfully.
Current design is now "/home/atanner/IC_CRUI/synopsys/nt.db:nt"

```

### Synthesis: 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?

```

module nt ( clk, rst, set, a, b, c, d, rout );
  input clk, rst, set, a, b;
  output c, d, rout;
  wire n2, n4, n5;

  NAND2 U5 ( ,A(c), ,B(b), ,Y(d) );
  \**FFGEN** rout_reg ( ,next_state(a), ,clocked_on(clk), ,force_00(n2),
    ,force_01(rst), ,force_10(n4), ,force_11(n2), ,0(rout) );
  TIELO U8 ( ,Y(n2) );
  NOR2 U10 ( ,A(rst), ,B(n5), ,Y(n4) );
  INV U11 ( ,A(n5), ,Y(n5) );
  NAND2 U12 ( ,A(d), ,B(a), ,Y(c) );
endmodule

```

### Synthesis: Always Statement

```

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

```

► What is this?

### Synthesis: Always Statement

```

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

```

► What is this?

```

Inferred memory devices in process
in routine set line 5 in file
'/home/elb/IC_CAD/syn-f06/set.v'.
-----
| Register Name | Type | Width | Bus | MB | AR | AS | SR | SS | ST |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| Q_reg         | Flip-flop | 1 | N | N | N | N | N | N | N |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|

```

```

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

```

### Synthesis: Always Statement

```

reg P,Q;
reg [3:0] R;
always @(posedge clk)
begin
  Q <= D;
  P <= Q;
  R <= R + 'h1;
end

```

► What is this?  
► Will it synthesize? Simulate?

### Synthesis: Always Statement

```

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

```

### Synthesis: Always Statement

► This is a simple D Flip-Flop

```

reg Q;
always @(posedge clk) Q <= D;

```

► So is this

```

reg Q;
always @(posedge clk) Q = D;

```

► = is for blocking assignments  
► <= is for nonblocking assignments



## 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

```

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

```

module the_top (clk, rst, a, b, sel, result);
  input clk, rst;
  input [3:0] a,b; input [2:0] sel;
  output reg [3:0] result;
  wire[3:0] sum, dif, alu;

  adder u0(a,b,sum);
  subber u1(.subtrahend(a), .subtractor(b), .difference(dif));

  assign alu = {4{(sel == 'b000)}} & sum
    | {4{(sel == 'b001)}} & dif;

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

endmodule

```

## Synthesis: Modules

```

// Verilog 1995 syntax
module adder (e,f,g);
  parameter SIZE=2;
  input [SIZE-1:0] e, f;
  output [SIZE-1:0] g;
  g = e + f;
endmodule

// Verilog 2001 syntax
module subber #(parameter SIZE = 3)
  (input [SIZE-1:0] c,d, output [SIZE-1:0]difference);
  difference = c - d;
endmodule

```

## Synthesis: Modules

```

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(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

```

## Multi-Way Decisions

- ▶ Standard if-else-if syntax

```

if ( <expression> )
  <statement>
else if ( <expression> )
  <statement>
else if ( <expression> )
  <statement>
else <statement>

```

### Priority vs. Parallel Choice (if)

```

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

```

### Priority vs. Parallel Choice

```

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

```

### Priority Encoders

```

// Priority encoders
// Allen Tanner
module prior_enc(x,y,z, a,b,c,d,e,f);
output reg x,y,z;
input a,b,c,d,e,f;
always@(a,b,c,d,e,f,g)
begin
    {x,y,z} = 3'b0;
    if ((a==1) && (b==1)) z = 1;
    else if ((c==1) && (d==1)) y = 1;
    else if ((e==1) && (f==1)) x = 1;
end
endmodule // prior_enc

```

### Priority Encoders

```

// Priority encoders
// Allen Tanner
module prior_enc(x,y,z, a,b,c,d,e,f);
output reg x,y,z;
input a,b,c,d,e,f;
always@(a,b,c,d,e,f,g)
begin
    {x,y,z} = 3'b0;
    if ((a==1) && (b==1)) z = 1;
    if ((c==1) && (d==1)) y = 1;
    if ((e==1) && (f==1)) x = 1;
end
endmodule // prior_enc

```

### Case Statements

► Multi-way decision on a single expression

```

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

```

### Case Example

```

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

```
// 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
```

001
010
100

### Weird Case Example

► 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

```
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**
- 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
```

### Synthesis Result

► Note that full-par results in combinational logic

### 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

```

### Synthesized Circuit

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

### 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

```

### 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 notfull-notpar 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

```

### Synthesized Circuit

► Not full and not parallel, infer a latch

## 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

## Get off my Case

```

Case tests
Allen Tanner
reg sel;
initial begin
  $display("We've only just begun");
  #1 $display (" Driving 0");
  sel = 0;
  #1 $display (" Driving 1");
  sel = 1;
  #1 $display (" Driving x");
  sel = 1'bx;
  #1 $display (" Driving s");
  sel = 1'b0s;
end
always @ (sel)
case (sel)
1'b0 : $display("CASE : Logic 0 on sel");
1'b1 : $display("CASE : Logic 1 on sel");
1'bx : $display("CASE : Logic x on sel");
endcase
always @ (sel)
casez (sel)
1'b0 : $display("CASEZ : Logic 0 on sel");
1'b1 : $display("CASEZ : Logic 1 on sel");
1'bx : $display("CASEZ : Logic x on sel");
endcase
always @ (sel)
casex (sel)
1'b0 : $display("CASEX : Logic 0 on sel");
1'b1 : $display("CASEX : Logic 1 on sel");
1'bx : $display("CASEX : Logic x on sel");
endcase
always @ (sel)
casexz (sel)
1'b0 : $display("CASEXZ : Logic 0 on sel");
1'b1 : $display("CASEXZ : Logic 1 on sel");
1'bx : $display("CASEXZ : Logic x on sel");
endcase

```

We've only just begun

Driving 0  
CASEZ : Logic 0 on sel  
CASEX : Logic 0 on sel  
CASE : Logic 0 on sel

Driving 1  
CASE : Logic 1 on sel  
CASEX : Logic 1 on sel  
CASEZ : Logic 1 on sel  
CASE : Logic 1 on sel

Driving x  
CASEZ : Logic x on sel  
CASEX : Logic x on sel  
CASE : Logic x on sel

Driving s  
CASEZ : Logic s on sel  
CASEX : Logic s on sel  
CASE : Logic s on sel

L18 "testfixture.new": \$f

Order Matters

## Get off my Case

```

Case tests
Allen Tanner
reg [15:0] opcode;
initial begin
  $display("We've only just begun");
  #1 $display (" Driving add");
  opcode = 16'b1000_1000_1010_1111;
  #1 $display (" Driving subtract");
  opcode = 16'b0100_0100_1010_1111;
  #1 $display (" Driving multiply");
  opcode = 16'b0010_0010_1010_1111;
  #1 $finish;
end
always @ (opcode)
casex (opcode)
16'b1xx?_xxxx_xxxx_xxxx : $display("CASEX. Opcode: add");
16'b1x1x_xxxx_xxxx_xxxx : $display("CASEX. Opcode: subtract");
16'b0xx1x_xxxx_xxxx_xxxx : $display("CASEX. Opcode: multiply");
endcase
always @ (opcode)
casez (opcode)
16'b1?1?_zzzz_zzzz_zzzz : $display("CASEZ. Opcode: add");
16'b1?1?_zzzz_zzzz_zzzz : $display("CASEZ. Opcode: subtract");
16'b?1?1?_zzzz_zzzz_zzzz : $display("CASEZ. Opcode: multiply");
endcase

```

We've only just begun

Driving add  
CASEZ. Opcode: add  
CASEX. Opcode: add  
Driving subtract  
CASEX. Opcode: subtract  
CASEZ. Opcode: subtract  
Driving multiply  
CASEZ. Opcode: multiply  
CASEX. Opcode: multiply  
L18 "testfixture.new": \$f

[Link](#)

## FSM Description

- ▶ One simple way: break it up like a schematic
  - ▶ A combinational block for next\_state generation
  - ▶ A combinational block for output generation
  - ▶ A sequential block to store the current state

## Modeling State Machines

```

// 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

```

## FSM Description

### Verilog Version

```

module moore (clk, clr, insig, outsig); // define combinational logic for
input clk, clr, insig; // next_state
output outsig; // always @(insig or state)
begin
// define state encodings as parameters
case (state)
parameter [1:0] s0 = 2'b00, // s0: if (insig) next_state = s1;
s1 = 2'b01, s2 = 2'b10, s3 = 2'b11; // else next_state = s0;
// define reg vars for state register // s1: if (insig) next_state = s2;
// and next_state logic // else next_state = s1;
reg [1:0] state, next_state; // s2: if (insig) next_state = s3;
//define state register (with // else next_state = s2;
//synchronous active-high clear) // s3: if (insig) next_state = s1;
always @(posedge clk) // else next_state = s0;
begin // endcase
end // end
if (clr) state = s0; // assign outsig as continuous assign
else state = next_state; // assign outsig =
// ((state == s1) || (state == s3));
end
endmodule

```

### Verilog Version

```

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
end

```

### Verilog Version Continued...

```

// 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

```

### Verilog Version Continued...

```

// 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

```

- ### 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)        if (sel == 1)
    out = in1;          out = in1;
  else out = in2;      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

```

// Verilog HDL for "Ax", "next" "behavioral"
// Four in a row detector - Allen Tanner

module saw4 (clk, clr, insig, saw4);
  input clk, clr, insig;
  output saw4;

  // Define state encodings as parameters
  parameter [2:0] s0 = 3'b000, s1 = 3'b001, s2 = 3'b010, s3 = 3'b011, s4 = 3'b100;

  // Define reg vars for state register and next_state logic
  reg [2:0] state, next_state;

  // Define state register (with asynchronous active-low clear)
  always @(posedge clk or negedge clr)
    begin
      if (clr==0) 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 = s0;
        s2: if (insig) next_state = s3;
            else next_state = s0;
        s3: if (insig) next_state = s4;
            else next_state = s0;
        s4: if (insig) next_state = s4;
            else next_state = s0;
        default: next_state = s0;
      endcase
    end

  // now set the saw4. This could also be done in an always
  // block... but in that case, saw4 would have to be
  // defined as a reg
  assign saw4 = state == s4;
endmodule

```

## Textbook FSM

```

// Verilog HDL for "Ax", "next" "behavioral"
// Four in a row detector - Allen Tanner

module saw4 (clk, clr, insig, saw4);
  input clk, clr, insig;
  output saw4;

  // Define state encodings as parameters
  parameter [2:0] s0 = 3'b000, s1 = 3'b001, s2 = 3'b010, s3 = 3'b011, s4 = 3'b100;

  // Define reg vars for state register and next_state logic
  reg [2:0] state, next_state;

  // Define state register (with asynchronous active-low clear)
  always @(posedge clk or negedge clr)
    begin
      if (clr==0) 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 = s0;
        s2: if (insig) next_state = s3;
            else next_state = s0;
        s3: if (insig) next_state = s4;
            else next_state = s0;
        s4: if (insig) next_state = s4;
            else next_state = s0;
        default: next_state = s0;
      endcase
    end

  // now set the saw4. This could also be done in an always
  // block... but in that case, saw4 would have to be
  // defined as a reg
  assign saw4 = state == s4;
endmodule

```

Comments

Polarity?

Always use <= for FF

## Documented FSM

```

// Verilog HDL for "Ax", "next" "behavioral"
// Four in a row detector - Allen Tanner

module saw4 (clk, clr, insig, saw4);
  input clk, clr, insig;
  output saw4;

  parameter [2:0] s0 = 3'b000; // initial state, saw at least 1 zero
  parameter [2:0] s1 = 3'b001; // saw 1 one
  parameter [2:0] s2 = 3'b010; // saw 2 ones
  parameter [2:0] s3 = 3'b011; // saw 3 ones
  parameter [2:0] s4 = 3'b100; // saw at least, 4 ones

  reg [2:0] state, next_state;

  always @(posedge clk or posedge clr) // state register
    begin
      if (clr) state <= s0;
      else state <= next_state;
    end

  always @(insig or state) // next state logic
    begin
      case (state)
        s0: if (insig) next_state = s1;
            else next_state = s0;
        s1: if (insig) next_state = s2;
            else next_state = s0;
        s2: if (insig) next_state = s3;
            else next_state = s0;
        s3: if (insig) next_state = s4;
            else next_state = s0;
        s4: if (insig) next_state = s4;
            else next_state = s0;
        default: next_state = s0;
      endcase
    end

  assign saw4 = state == s4;
endmodule

```

```

Waveform Test Bench
// Four ones in a row detector.
// Test bench
// Allen Tanner

initial
begin
  clk = 1'b0;
  clr = 1'b0;
  insig = 1'b0;

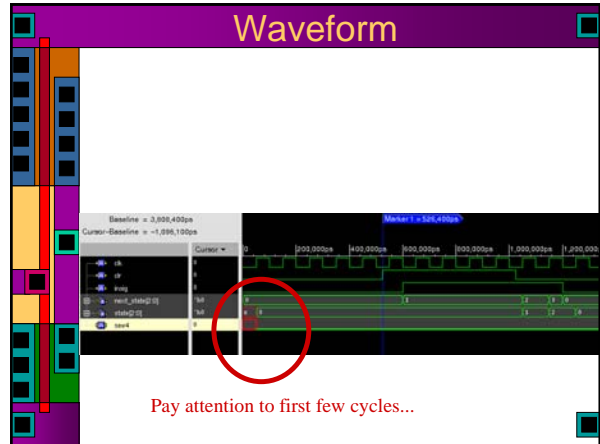
  send_message(32'b0011_1000_1010_1111_0000_0111_1110_0000);
  send_message(32'b0011_1000_1010_1111_0000_0111_1110_0000);
  $finish;
end

always #50 clk = ~clk;

initial
begin
  #525
  clr = 1'b1;
  #500
  clr = 1'b0;
end

task send_message;
  input [31:0] pattern;
  integer i;
  begin
    for(i=0;i<32; i=i+1)
      @(negedge clk) insig = pattern[i];
    end
endtask // send_message

```



```

FSM
// Verilog HDL for "Ax", "see4" behavioral
// Four in a row detector - Allen Tanner

module see4 (clk, clr, insig, saw4);
  input clk, clr, insig;
  output saw4;

  parameter [2:0] s0 = 3'b000; // initial state, saw at least 1 zero
  parameter [2:0] s1 = 3'b001; // saw 1 one
  parameter [2:0] s2 = 3'b010; // saw 2 ones
  parameter [2:0] s3 = 3'b011; // saw 3 ones
  parameter [2:0] s4 = 3'b100; // saw at least, 4 ones

  reg [2:0] state, next_state;

  always @(posedge clk or posedge clr) // state register
  begin
    if (clr) state <= s0;
    else
      case (state)
        s0: if (insig) state <= s1;
            else state <= s0;
        s1: if (insig) state <= s2;
            else state <= s0;
        s2: if (insig) state <= s3;
            else state <= s0;
        s3: if (insig) state <= s4;
            else state <= s0;
        s4: if (insig) state <= s4;
            else state <= s0;
        default: state <= s0;
      endcase // case(state)
    end
  assign saw4 = state == s4;
endmodule

```

```

FSM
// Verilog HDL for "Ax", "see4" behavioral
// Four in a row detector - Allen Tanner

module see4 (clk, clr, insig, saw4);
  input clk, clr, insig;
  output saw4;

  parameter [2:0] s0 = 3'b000; // initial state, saw at least 1 zero
  parameter [2:0] s1 = 3'b001; // saw 1 one
  parameter [2:0] s2 = 3'b010; // saw 2 ones
  parameter [2:0] s3 = 3'b011; // saw 3 ones
  parameter [2:0] s4 = 3'b100; // saw at least, 4 ones

  reg [2:0] state;
  wire [2:0] next_state;

  assign next_state = {3{(state == s0) && !insig} & s0,
                     {3{(state == s1) && !insig} & s1,
                     {3{(state == s1) && insig} & s2,
                     {3{(state == s2) && !insig} & s0,
                     {3{(state == s2) && insig} & s3,
                     {3{(state == s3) && !insig} & s0,
                     {3{(state == s3) && insig} & s4,
                     {3{(state == s4) && !insig} & s0,
                     {3{(state == s4) && insig} & s4;

  always @(posedge clk or posedge clr) // state register
  begin
    if (clr) state <= s0;
    else state <= next_state;
  end
  assign saw4 = state == s4;
endmodule

```

```

FSM
// Verilog HDL for "Ax", "see4" behavioral
// Four in a row detector - Allen Tanner

module see4 (clk, clr, insig, saw4);
  input clk, clr, insig;
  output saw4;

  parameter [2:0] s0 = 3'b000; // initial state, saw at least 1 zero
  parameter [2:0] s1 = 3'b001; // saw 1 one
  parameter [2:0] s2 = 3'b010; // saw 2 ones
  parameter [2:0] s3 = 3'b011; // saw 3 ones
  parameter [2:0] s4 = 3'b100; // saw at least, 4 ones

  reg [2:0] state;

  always @(posedge clk or posedge clr) // state register
  begin
    if (clr) state <= s0;
    else state <= {3{(state == s0) && !insig} & s0,
                  {3{(state == s0) && insig} & s1,
                  {3{(state == s1) && !insig} & s0,
                  {3{(state == s1) && insig} & s2,
                  {3{(state == s2) && !insig} & s0,
                  {3{(state == s2) && insig} & s3,
                  {3{(state == s3) && !insig} & s0,
                  {3{(state == s3) && insig} & s4,
                  {3{(state == s4) && !insig} & s0,
                  {3{(state == s4) && insig} & s4;
    end
  assign saw4 = state == s4;
endmodule

```

```

One-Hot FSM
// Verilog HDL for "Ax", "see4" behavioral
// Four in a row detector - Allen Tanner

module see4 (clk, clr, insig, saw4);
  input clk, clr, insig;
  output saw4;

  reg s0; // initial state, saw at least 1 zero
  reg s1; // saw 1 one
  reg s2; // saw 2 ones
  reg s3; // saw 3 ones
  reg s4; // saw at least, 4 ones

  always @(posedge clk or posedge clr) // state register
  begin
    if (clr)
      begin
        s0 <= 1'b1;
        s1 <= 1'b0;
        s2 <= 1'b0;
        s3 <= 1'b0;
        s4 <= 1'b0;
      end
    else
      begin
        s0 <= (s0 | s1 | s2 | s3 | s4) & !insig;
        s1 <= s0 & insig;
        s2 <= s1 & insig;
        s3 <= s2 & insig;
        s4 <= s3 & insig;
        s4 <= s4 & insig;
      end
    end
  assign saw4 = s4;
endmodule

```



## One-Hot FSM Counting

## Oops

```

module see4 ( clk, clr, insig, saw4 );
input clk, clr, insig;
output saw4;
wire s0, s1, s2, N2, N3, N4, N5, n9, n10, n11, n14, n15, n16, n17, n18,
n19, n20, n21, n22;

//FFGEN
s0_reg (.next_state(n14), .clocked_on(clk), .force_00(n11),
.force_01(n11), .force_10(clr), .force_11(n11), .Q(s0) );
DFF s2_reg (.D(N3), .CLK(clk), .nCLR(n15), .Q(s2) );
DFF s1_reg (.D(N2), .CLK(clk), .nCLR(n15), .Q(s1) );
DFF s4_reg (.D(N5), .CLK(clk), .nCLR(n15), .Q(saw4), .QB(n9) );
DFF s3_reg (.D(N4), .CLK(clk), .nCLR(n15), .QB(saw4) );
TIELO U20 (.Y(n11) );
INV U21 (.A(clr), .Y(n15) );
AOI U22 (.A(n16), .B(n17), .C(insig), .Y(n14) );
NOR2 U23 (.A(s2), .B(s1), .Y(n17) );
NOR2 U24 (.A(s0), .B(n18), .Y(n16) );
INV U25 (.A(n19), .Y(n5) );
NAND2 U26 (.A(n18), .B(insig), .Y(n19) );
NAND2 U27 (.A(n9), .B(n10), .Y(n18) );
INV U28 (.A(n20), .Y(n4) );
NAND2 U29 (.A(insig), .B(s2), .Y(n20) );
INV U30 (.A(n21), .Y(n3) );
NAND2 U31 (.A(insig), .B(s1), .Y(n21) );
INV U32 (.A(n22), .Y(n2) );
NAND2 U33 (.A(insig), .B(s0), .Y(n22) );
endmodule

```

## No Asynchronous Sets

```

// Verilog HDL for "Ax", "see4" behavioral
// Four in a row detector - Allen Tanner
module see4 (clk, clr, insig, saw4);
input clk, clr, insig;
output saw4;

reg ns0; // initial state, saw at least 1 zero
reg s1; // saw 1 one
reg s2; // saw 2 ones
reg s3; // saw 3 ones
reg s4; // saw at least, 4 ones

wire s0; // alias for ns0 (ns0 used to avoid FFGEN in beh2str)
assign s0 = ns0;

always @(posedge clk or posedge clr) // state register
begin
if (clr)
begin
ns0 <= 1'b0;
s1 <= 1'b0;
s2 <= 1'b0;
s3 <= 1'b0;
s4 <= 1'b0;
end
else
begin
ns0 <= ~(s0 | s1 | s2 | s3 | s4) & insig;
s1 <= s0 & insig;
s2 <= s1 & insig;
s3 <= s2 & insig;
s4 <= s3 & insig
| s4 & insig;
end
end

assign saw4 = s4;

```

## That's better

```

module see4 ( clk, clr, insig, saw4 );
input clk, clr, insig;
output saw4;
wire ns0, N0, N1, N2, N4, N5, n1, n9, n10, n11, n12, n13, n14, n15;

DFF s4_reg (.D(N5), .CLK(clk), .nCLR(n1), .Q(saw4), .QB(n9) );
DFF s3_reg (.D(N2), .CLK(clk), .nCLR(n1), .QB(n12) );
DFF s2_reg (.D(N1), .CLK(clk), .nCLR(n1), .QB(n10) );
DFF s1_reg (.D(N0), .CLK(clk), .nCLR(n1), .QB(n11) );
DFF ns0_reg (.D(N4), .CLK(clk), .nCLR(n1), .Q(ns0) );
INV U12 (.A(clr), .Y(n1) );
AOI U13 (.A(n9), .B(n12), .C(n13), .Y(n5) );
OAI U14 (.A(n14), .B(n15), .C(n13), .Y(n4) );
NAND2 U15 (.A(ns0), .B(n12), .Y(n15) );
NAND3 U16 (.A(n10), .B(n9), .C(n11), .Y(n14) );
NOR2 U17 (.A(n13), .B(n10), .Y(n2) );
NOR2 U18 (.A(n13), .B(n11), .Y(n1) );
NOR2 U19 (.A(ns0), .B(n13), .Y(n0) );
INV U20 (.A(insig), .Y(n13) );
endmodule

```

## Synchronous Clear

```

// Verilog HDL for "Ax", "see4" behavioral
// Synchronous clear - Allen Tanner
module see4 (clk, clr, insig, saw4);
input clk, clr, insig;
output saw4;

reg s0; // initial state, saw at least 1 zero
reg s1; // saw 1 one
reg s2; // saw 2 ones
reg s3; // saw 3 ones
reg s4; // saw at least, 4 ones

always @(posedge clk) // state register with synchronous clear
begin
if (clr)
begin
s0 <= 1'b1;
s1 <= 1'b0;
s2 <= 1'b0;
s3 <= 1'b0;
s4 <= 1'b0;
end
else
begin
s0 <= (s0 | s1 | s2 | s3 | s4) & insig;
s1 <= s0 & insig;
s2 <= s1 & insig;
s3 <= s2 & insig;
s4 <= s3 & insig
| s4 & insig;
end
end

assign saw4 = s4;
endmodule

```

## Synchronous Clear

```

module see4 ( clk, clr, insig, saw4 );
input clk, clr, insig;
output saw4;
wire N9, N10, N11, N12, N13, n2, n18, n19, n20, n21, n22, n23, n24, n25,
n26, n27, n28, net12, net11, net9, net8;

DFF s4_reg (.D(N13), .CLK(clk), .nCLR(n2), .Q(saw4), .QB(net8) );
DFF s3_reg (.D(N12), .CLK(clk), .nCLR(n2), .QB(net9) );
DFF s2_reg (.D(N11), .CLK(clk), .nCLR(n2), .Q(net11), .QB(n19) );
DFF s1_reg (.D(N10), .CLK(clk), .nCLR(n2), .QB(n18) );
DFF s0_reg (.D(N9), .CLK(clk), .nCLR(n2), .Q(net12), .QB(n20) );
TIEHI U21 (.Y(n2) );
NAND2 U22 (.A(n21), .B(n22), .Y(n9) );
NAND2 U23 (.A(n23), .B(n24), .Y(n22) );
INV U24 (.A(insig), .Y(n24) );
NAND3 U25 (.A(n25), .B(n18), .C(n26), .Y(n23) );
NOR2 U26 (.A(net12), .B(net11), .Y(n26) );
NOR2 U27 (.A(n25), .B(n27), .Y(n13) );
INV U28 (.A(n28), .Y(n25) );
NAND2 U29 (.A(net8), .B(n29), .Y(n28) );
NOR2 U30 (.A(n27), .B(n19), .Y(n12) );
NOR2 U31 (.A(n27), .B(n18), .Y(n11) );
NOR2 U32 (.A(n27), .B(n20), .Y(n10) );
NAND2 U33 (.A(insig), .B(n21), .Y(n27) );
INV U34 (.A(clr), .Y(n21) );
endmodule

```



### ROM vs. Verilog

```
assign twant = {  
    {mc00} & ROME 7: 0}  
    {mc01} & ROME 15: 8}  
    {mc02} & ROME 23: 16}  
    {mc03} & ROME 31: 24}  
    {mc04} & ROME 39: 32}  
    {mc05} & ROME 47: 40}  
    {mc06} & ROME 55: 48}  
    {mc07} & ROME 63: 56}  
    {mc08} & ROME 71: 64}  
    {mc09} & ROME 79: 72}  
    {mc10} & ROME 87: 80}  
    {mc11} & ROME 95: 88}  
    {mc12} & ROME103: 96}  
    {mc13} & ROME111: 104}  
    {mc14} & ROME119: 112}  
    {mc15} & ROME127: 120};  
  
assign T = {twant[0],twant[1],twant[2],twant[3],twant[4],twant[5],twant[6],twant[7]};  
module // char10
```

