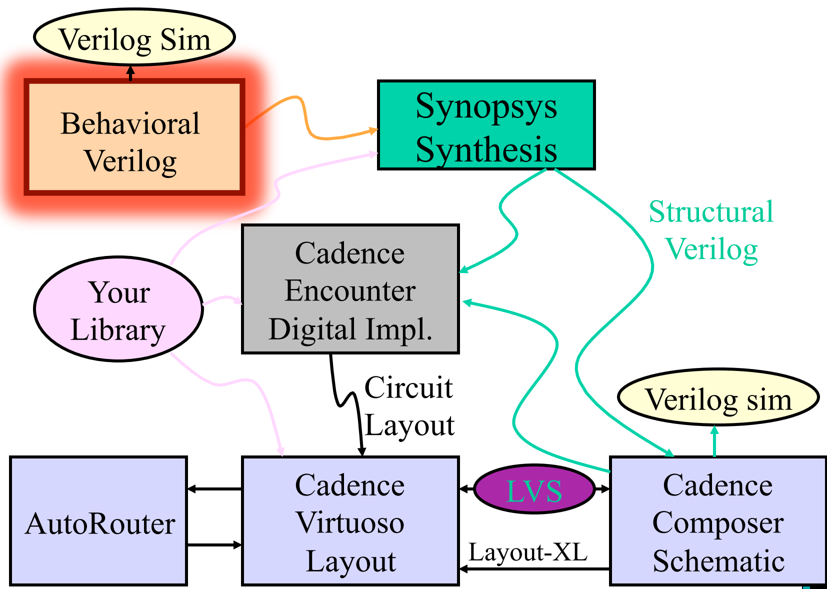
# CS6710 Tool Suite




# 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.

## Synthesis

This lecture is only about synthesis...

## Quick Review

```
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
     <primitive instantiations>
     <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
  - *Note: no X in synthesis…*
- Digital Hardware: The domain of Verilog
  - Either logic (gates)
    - outputs represented by wire variables
  - Or storage (registers & latches)
    - outputs represented by reg variables

## 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 operator 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;
```

## 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

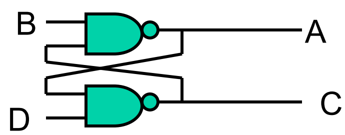
- The assign statement is sufficient to create all combinational logic
- What about this:

```
assign a = ~(b & c);
assign c = ~(d & a);
```

## 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 Behavioral Module

```
// Behavioral model of NAND gate
module NAND (out, in1, in2);
    output out;
    input in1, in2;
    // Uses Verilog builtin nand function
    // syntax is func id (args);
    nand i0(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);
    AND2X1 u1(w1, in1, in2);
    INVX1 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...
     AND2X1 u1(.Q(w1), .A(in1), .B(in2));
     INVX1 u2(.A(w1), .Q(out));
endmodule
```

## Primitive Gates

- Multiple input gates
  - <gatename> [delay] [id] (out, in1, in2, in3...);
  - and, or, nand, nor, xor, xnor
- Multiple output gates
  - <gatename> [delay] [id] (out1, out2, ... outn, in);
  - buf, not
- Tristate gates
  - <gatename> [delay] [id] (out, in, ctrl);
  - bufif1, bufif0, notif1, notif0

## 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, 3:2:5)

## Primitive Gates

- and (out, a, b);
- nand i0 (out a b c d e f g);
- xor #(2,3) (out a b c);
- buf (Y A);
- buf #(2:3:4, 3:4;5) _i1 (y, a);
- bufif1 (out, in, ctl);
- notif0 #(1, 2, 3) (Y, A, S);

## Primitive Gates

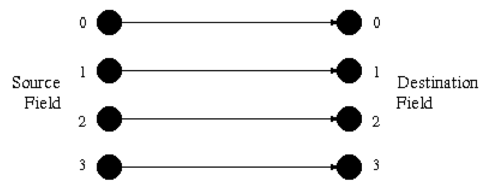
- OR – you can skip the delays on each gate, and use a specify block for the whole module
  - Specifies from module input to module outputs
  - Outputs must be driven by a primitive gate
  - The syntax defines the delay for each path from input to output

## Simple Behavioral Module

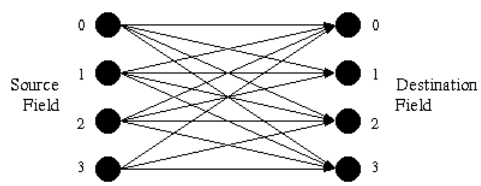
```
// Behavioral model of NAND gate
module NAND (out, in1, in2);
     output out;
     input in1, in2;

     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
```

# Specify Block Types

- Parallel Connection (one to one)




- Full Connection (one to many)




# Parallel Specify

```
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 form 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
```

# Full Specify

```
// a[63:0] is a 64 bit input register and
// q[7:0] is an 8 bit output register
// this would require 64 x 8 = 512 parallel
// connections, but only 1 full

specify
  ( a *> q ) = 8;   // eqivalent to 512 parallel connections
endspecify
```

## 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
```

## 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
```

## 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

## 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 (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
```

# 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 (2001 syntax)
  - 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

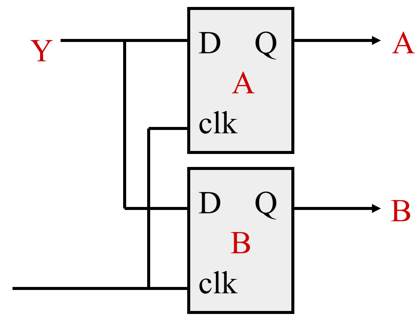
- Synthesis ignores all that timing stuff
- So, what does it mean to have blocking vs. non-blocking assignment for synthesis?
  - ```
    begin            ?           begin
      A=B;     <----------->       A<=B;
      B=A;                         B<=A;
    end                          end
    ```

---

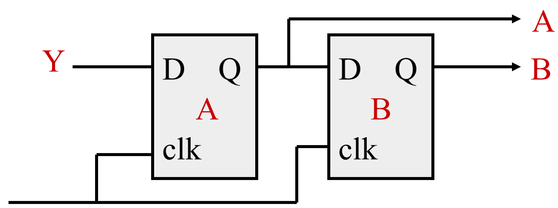
  - ```
                     ?
    begin      <----------->     begin
      A=Y;                         A<=Y;
      B=A;                         B<=A;
    end                          end
    ```

# Synthesized Circuits

- begin
  A = Y;
  B = A;
  end

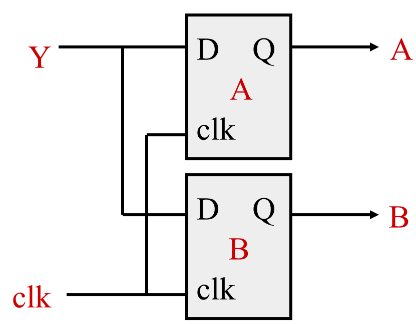


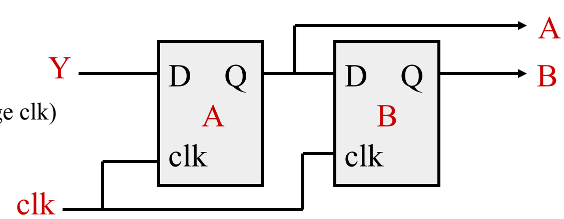

- begin
  A <= Y;
  B <= A;
  end
- begin
  B = A;
  A = Y;
  end




# 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

always @(posedge clk)
  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…
  - Non-blocking is often better for setting subsets of signals
    - Set them to defaults at beginning, then reset only the ones that change

## Non-blocking example

```
always @(*)
  begin  // set all outputs to zero, then assert only the appropriate ones
     irwrite <= 4'b0000;
     pcwrite <= 0; pcwritecond <= 0; regwrite <= 0; regdst <= 0;
     memread <= 0; memwrite <= 0; alusrca <= 0; alusrcb <= 2'b00;
     aluop <= 2'b00; pcsource <= 2'b00; iord <= 0; memtoreg <= 0;
     case(state)
          FETCH1:
            begin
              memread <= 1;
              irwrite <= 4'b0001; // change to reflect new memory
              alusrcb <= 2'b01;   // get the IR bits in the right spots
              pcwrite <= 1;       // FETCH 2,3,4 also changed...
             end
          FETCH2:
            begin
              memread <= 1;
              irwrite <= 4'b0010;
              alusrcb <= 2'b01;
              pcwrite <= 1;
              end
// etc…
```

## 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 > a);
          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

## 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

## 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?

> 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_CAD/synopsys/nt.v'.
===============================================================================
|    Register Name    |   Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
===============================================================================
|      rout_reg       | Flip-flop |   1   |  N  | N  | Y  | Y  | N  | N  | N  |
===============================================================================
Presto compilation completed successfully.
Current design is now '/home/atanner/IC_CAD/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 U6 ( .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), .Q(rout) );
  TIELO U9 ( .Y(n2) );
  NOR2 U10 ( .A(rst), .B(n5), .Y(n4) );
  INV U11 ( .A(set), .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

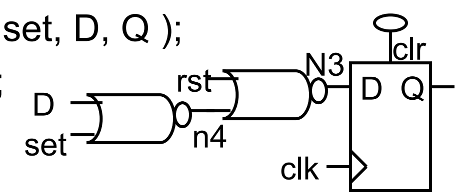
```
reg Q;
always @(posedge clk)
  if (rst) Q <= 'b0;
    else if (set) Q <= 'b1;
      else Q <= D;
```

- 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
```


```
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
```


| A B | Out |
|---|---|
| 0 0 | 1 |
| 0 1 | 0 |
| 1 0 | 0 |
| 1 1 | 0 |

## 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;
   assign g = e + f;
endmodule

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

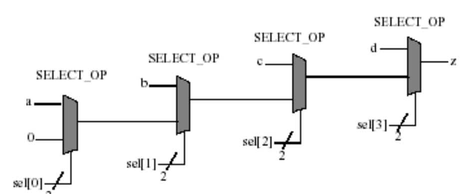
# Multi-Way Decisions

- Standard if-else-if syntax

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

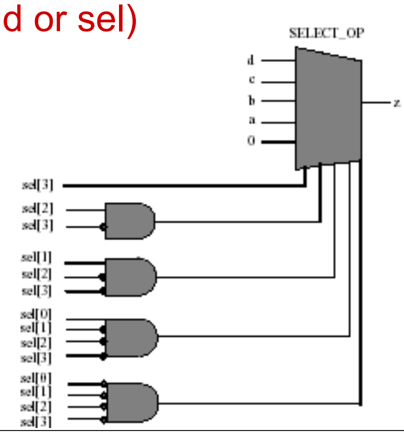
# Priority vs. Parallel Choice (if)

```
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
```



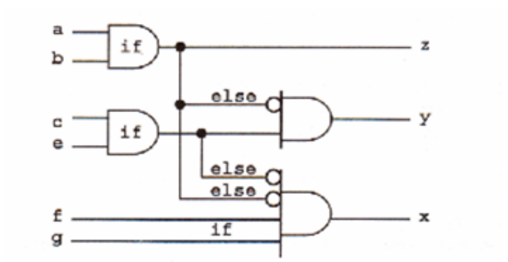
# Priority vs. Parallel Choice

```
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
```



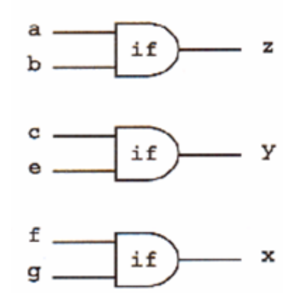

# Priority Encoders

```
//
// 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

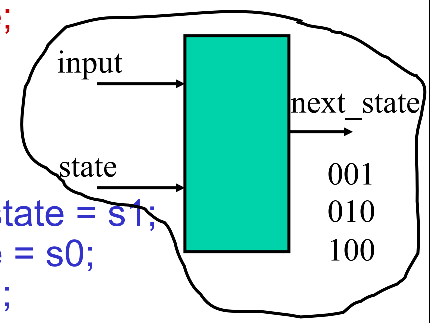
case ( *<expresion>* )
   *<expression>*: *<statement>*
   *<expression>*, *<expression>*: *<statement>*
   *<expression>*: *<statement>*
   default: *<statement>*
endcase

# 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
```



## 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

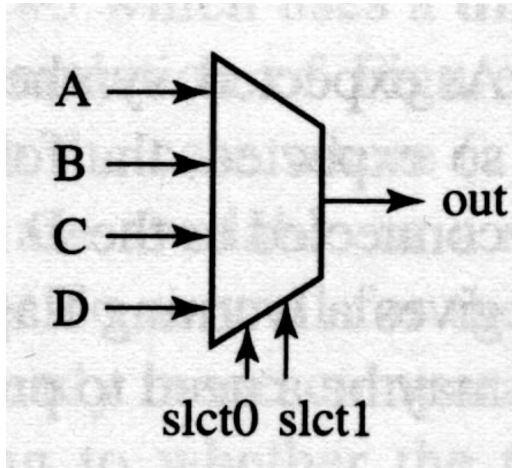
- Case statements check each case in sequence
- A case statement is full if all possible outcomes are accounted for
- A case statement is parallel if the stated alternatives are mutually exclusive
- These distinctions make a difference in how cases are translated to circuits…
  - Similar to the if statements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

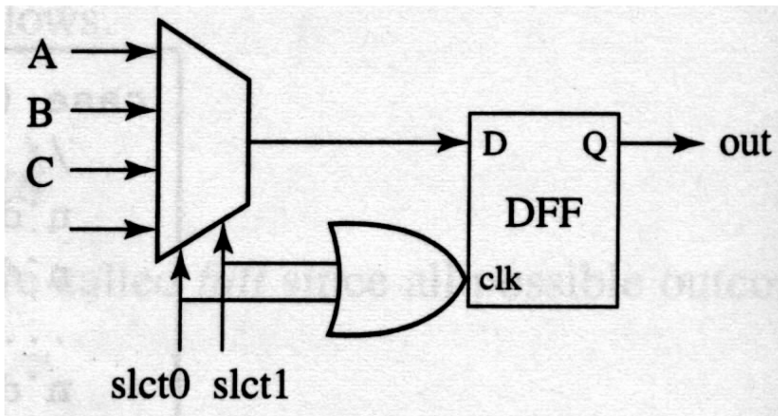



# Case notfull-par example

```
// 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…

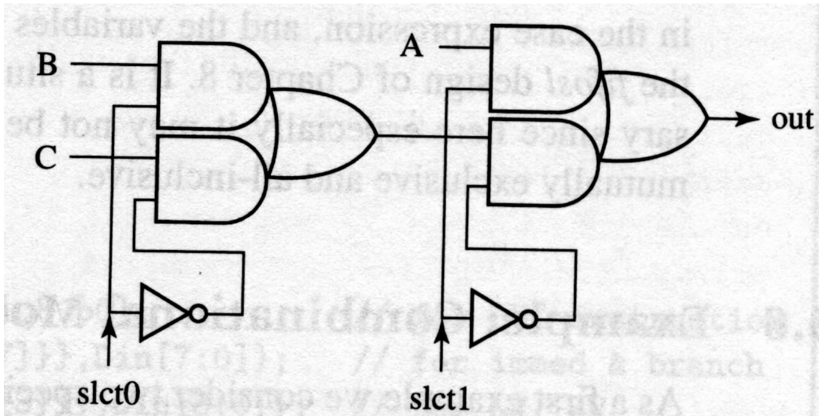



# Case full-notpar example

```
// 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

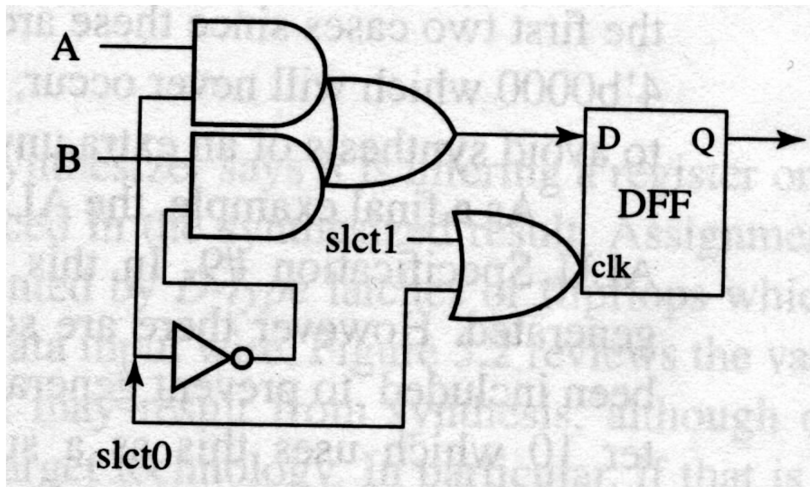



## Case notfull-notpar example

```
// 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 ("\n       Driving 0");
   sel = 0;
   #1 $display ("\n       Driving 1");
   sel = 1;
   #1 $display ("\n       Driving x");
   sel = 1'bx;
   #1 $display ("\n       Driving z");
   sel = 1'bz;
   #1 $finish;
end

always @ (sel)
case (sel)
   1'b0 : $display("CASE    : Logic 0 on s
   1'b1 : $display("CASE    : Logic 1 on s
   1'bx : $display("CASE    : Logic x on s
   1'bz : $display("CASE    : Logic z on s
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");
   1'bz : $display("CASEX   : Logic z 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");
   1'bz : $display("CASEZ   : Logic z on sel");
endcase

always @ (sel)
casez (sel)
   1'b1 : $display("CASEZa : Logic 1 on sel");
   1'b0 : $display("CASEZa : Logic 0 on sel");
   1'bx : $display("CASEZa : Logic x on sel");
   1'bz : $display("CASEZa : Logic z on sel");
endcase
```

```
We've only just begun

      Driving 0
CASEZa : Logic 0 on sel
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
CASEZa : Logic 1 on sel

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

      Driving z
CASE   : Logic z on sel
CASEX  : Logic 0 on sel
CASEZ  : Logic 0 on sel
CASEZa : Logic 1 on sel
L20 "testfixture.new": $fi
```

**Order Matters**

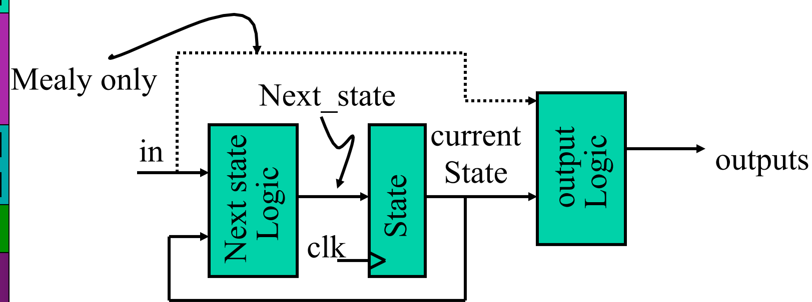
## Get off my Case

```
//
// 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 mutiply");
   opcode = 16'b0010_0010_1010_1111;
   #1 $finish;
end

always @ (opcode)
casex (opcode)
  16'b1xz?_xxxx_xxxx_xxxx : $display("CASEX. Opcode: add");
  16'bx1xx_xxxx_xxxx_xxxx : $display("CASEX. Opcode: subtract");
  16'bxx1x_xxxx_xxxx_xxxx : $display("CASEX. Opcode: multiply");
endcase

always @ (opcode)
casez (opcode)
  16'b1???_zzzz_zzzz_zzzz : $display("CASEZ. Opcode: add");
  16'b?1??_zzzz_zzzz_zzzz : $display("CASEZ. Opcode: subtract");
  16'b??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 mutiply
CASEZ. Opcode: multiply
CASEX. Opcode: multiply
L18 "testfixture.new": $fir
```

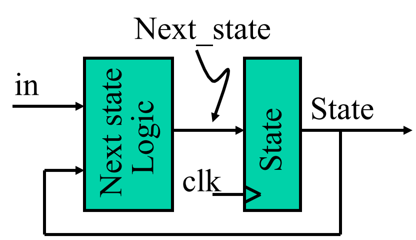
# FSM Description

- 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

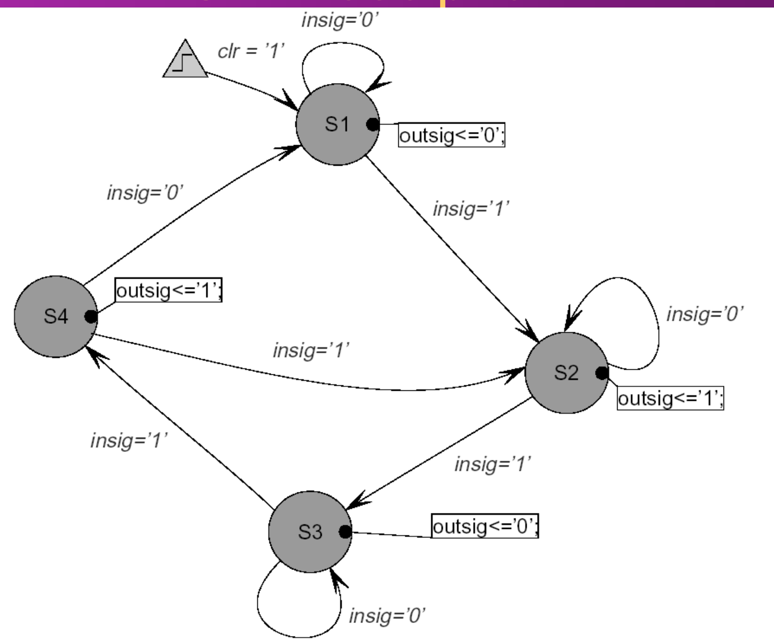



# Modeling State Machines

```
// 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 Desciption




# 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
// 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
// assign outsig as continuous assign
assign outsig =
    ((state == s1) || (state == s3));
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
```

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

```
always @(sel or in1 or in2)
   if (sel == 1)
      out = in1;
   else out = in2;
```

- Which one is a good mux?

## Combinational Always Blocks

- Be careful…

```
always @(sel)
   if (sel == 1)
      out = in1;
   else out = in2;
```

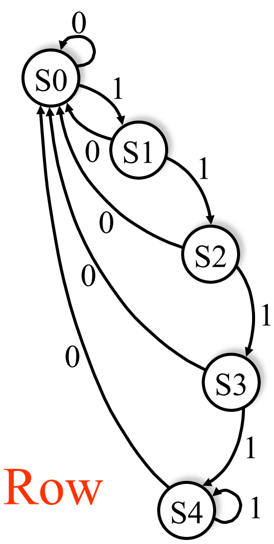
```
always @(sel or in1 or in2)
   if (sel == 1)
      out = in1;
   else out = in2;
```

- Which one is a good mux?
- Always @*
  if (sel == 1) out = in1; else out = in2;

## 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", "see4" "behavioral"
// Four in a row detector - Allen Tanner

module see4 (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", "see4" "behavioral"
// Four in a row detector - Allen Tanner

module see4 (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?

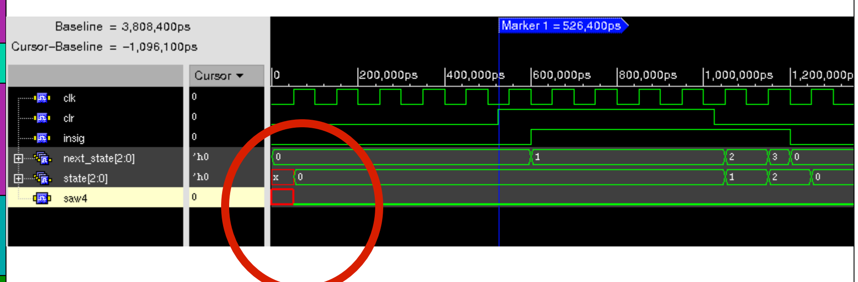
# Documented FSM

```
// Verilog HDL for "Ax", "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 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
```

# Waveform




Pay attention to first few cycles...

## 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 == 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;

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

    assign saw4 = state == s4;

endmodule
```

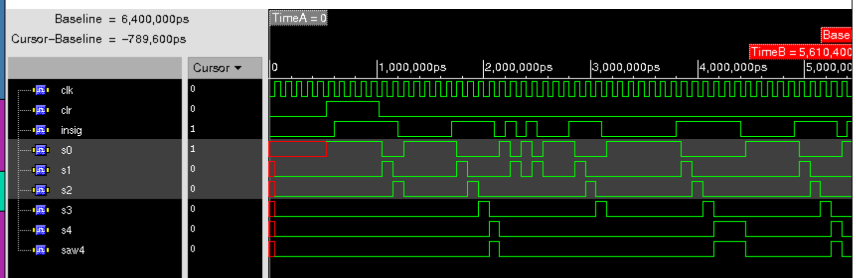
# FSM

```
// 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 & 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(n10) );
  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 !nso (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
```

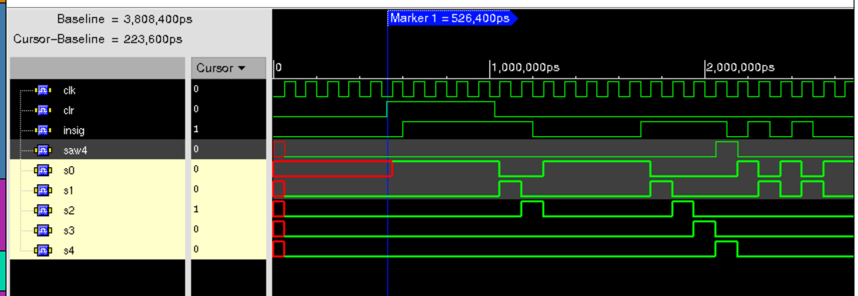
## Synchronous Clear

```verilog
// 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

```verilog
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(net9), .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
```

# Synchronous Clear

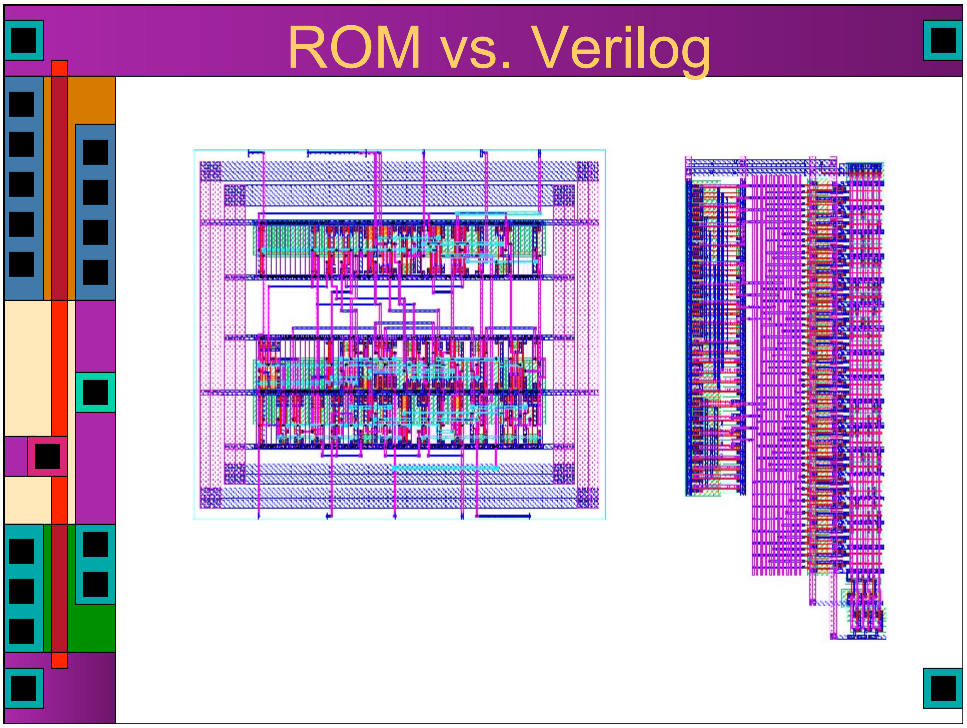


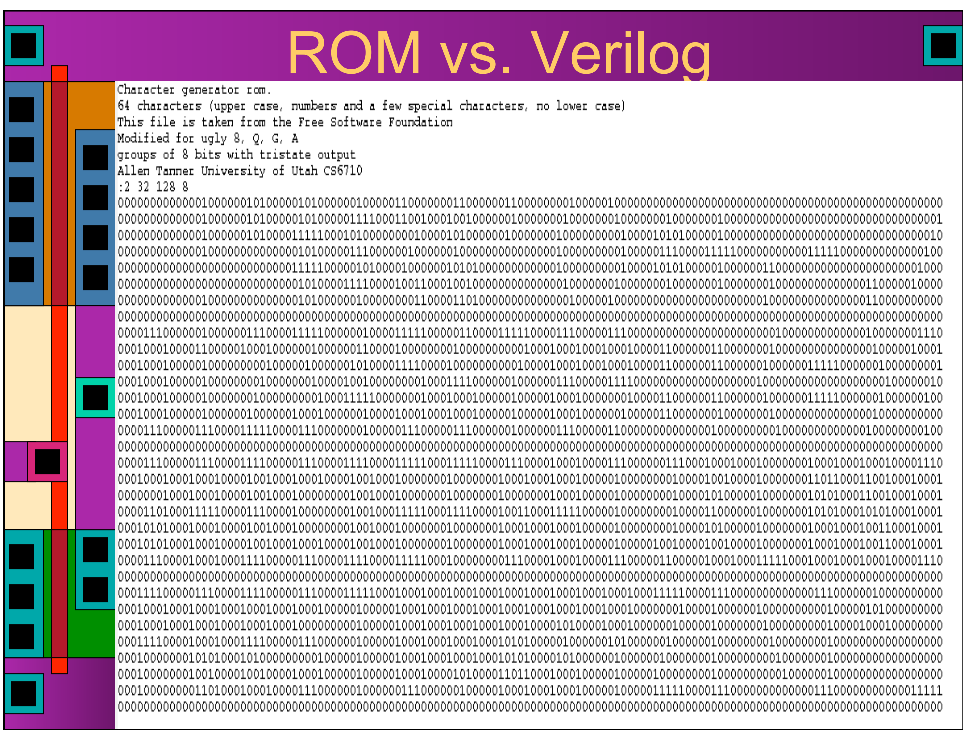

- Is asynchronous clear really asynchronous?
- What about set-up & hold with respect to clock edge?

# ROM vs. Verilog

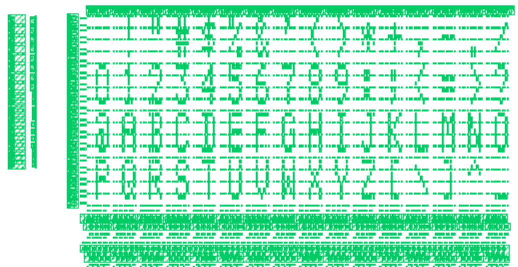
```
name rom
32 characters of six bits each
:16 32 6
00 ;
00 ;
28 ; H
25 ; E
2C ; L
2C ; L
2F ; O
00 ;
37 ; W
2F ; O
32 ; R
2C ; L
24 ; D
00 ;
00 ;
00 ;

00 ;
00 ;
33 ; S
25 ; E
34 ; T
00 ;
2D ; M
25 ; E
00 ;
26 ; F
32 ; R
25 ; E
25 ; E
00 ;
00 ;
00 ;
```

```
module mywords(addr, char);
   input [4:0] addr;
   output reg [5:0] char;

always @(addr)
 begin
  case(addr)
    'h00 : char = 'h00  ;  //
    'h01 : char = 'h00  ;  //
    'h02 : char = 'h28  ;  // H
    'h03 : char = 'h25  ;  // E
    'h04 : char = 'h2C  ;  // L
    'h05 : char = 'h2C  ;  // L
    'h06 : char = 'h2F  ;  // O
    'h07 : char = 'h00  ;  //
    'h08 : char = 'h37  ;  // W
    'h09 : char = 'h2F  ;  // O
    'h0A : char = 'h32  ;  // R
    'h0B : char = 'h2C  ;  // L
    'h0C : char = 'h24  ;  // D
    'h0D : char = 'h00  ;  //
    'h0E : char = 'h00  ;  //
    'h0F : char = 'h00  ;  //

    'h10 : char = 'h00  ;  //
    'h11 : char = 'h00  ;  //
    'h12 : char = 'h33  ;  // S
    'h13 : char = 'h25  ;  // E
    'h14 : char = 'h34  ;  // T
    'h15 : char = 'h00  ;  //
    'h16 : char = 'h2D  ;  // M
    'h17 : char = 'h25  ;  // E
    'h18 : char = 'h00  ;  //
    'h19 : char = 'h26  ;  // F
    'h1A : char = 'h32  ;  // R
    'h1B : char = 'h25  ;  // E
    'h1C : char = 'h25  ;  // E
    'h1D : char = 'h00  ;  //
    'h1E : char = 'h00  ;  //
    'h1F : char = 'h00  ;  //
  endcase
 end
endmodule // mywords
```


ROM vs. Verilog


ROM vs. Verilog
Character generator rom.
64 characters (upper case, numbers and a few special characters, no lower case)
This file is taken from the Free Software Foundation
Modified for ugly 8, Q, G, A
groups of 8 bits with tristate output
Allen Tanner University of Utah CS6710
:2 32 128 8

# ROM vs. Verilog



# ROM vs. Verilog

```
// Verilog HDL for "Ax", "char10" "behavioral"
//2Character generator rom.
//64 characters (upper case, numbers and a few special characters, no lower case)
//This file is taken from the Free Software Foundation
//Modified for ugly 8, Q, G, A
//groups of 8 bits with tristate output
//Allen Tanner University of Utah CS6710

module char10(ROM_row, nOE0, nOE8, nOE16, nOE24, nOE32, nOE40, nOE48, nOE56, nOE64, nOE72, nOE80, nOE88, nOE96, nOE104, nOE112, nOE120, T);
   input [4:0]ROM_row;
   input      nOE0, nOE8, nOE16, nOE24, nOE32, nOE40, nOE48, nOE56, nOE64, nOE72, nOE80, nOE88, nOE96, nOE104, nOE112, nOE120;
   output [7:0] T;

   wire [7:0]   twist;
   reg [127:0] ROW;

   always @(ROM_row)
     case(ROM_row)

       5'h00 : ROW = 128'b00000000000000100000010100000101000000100000110000000110000001100000000100000100000000000000000000000000000000000000000000000000
       5'h01 : ROW = 128'b00000000000000100000010100000101000001111000110010001001000000100000001000000010000000100000001000000010000000000000000000000000001
       5'h02 : ROW = 128'b0000000000000010000001010000111110001010000000001000010100000010000000100000000010000101010000010000000000000000000000000000000010
       5'h03 : ROW = 128'b0000000000000010000000000000010100000111000000100000010000000000000001000000000100000111000011110000000000011110000000000000100
       5'h04 : ROW = 128'b0000000000000000000000000000111110000010100001000000101010000000000000100000000010000101010000010000001100000000000000000000001000
       5'h05 : ROW = 128'b0000000000000000000000000000101000011110000100110001001000000000000000100000001000000010000000100000001000000000000000011000001000
       5'h06 : ROW = 128'b0000000000000010000000000000010100000010000000011000011010000000000000010000010000000000000000000000010000000000000000011000000000
       5'h07 : ROW = 128'b00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
       5'h08 : ROW = 128'b000011100000001000000111000011111000000100001111100000110000111110001110000011100000000000000000000000010000000000000010000000111
       5'h09 : ROW = 128'b000100010000110000001000100000010000001100001000000001000000000100010001000100010000110000001100000001000000000000000001000001000
       5'h0A : ROW = 128'b000100010000001000000000100000100000010100001111000010000000000100001000100010001000011000000110000001000000111100000010000000001
       5'h0B : ROW = 128'b00010001000000100000000010000000100001001000000000100011110000000100000011100000111100000000000000000000100000000000000000010000001
       5'h0C : ROW = 128'b00010001000000100000000100000000010001111100000000100010001000001000000100010000000010000110000001100000010000001111000000010000000100
       5'h0D : ROW = 128'b00010001000000100000001000000010001000000100001000100010001000001000000100010000000100000110000000100000001000000000000000010000000000
       5'h0E : ROW = 128'b0000111000000111000011111000011100000000100000111000001110000000100000011100000110000000000000001000000000100000000000000100000000100
       5'h0F : ROW = 128'b00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
       5'h10 : ROW = 128'b000011100000111000011110000011100001111000011111000111110000111000010001000011100000001110001000100010000000100010001000100001110
       5'h11 : ROW = 128'b000100010001000100001001000100010000100100010000000010000000100010001000100000100000000010000100100001000000011011000110010001001
       5'h12 : ROW = 128'b000000010001000100001001000100000000100100010000000010000000100000001000100000100000000010000101000001000000010101000110010001001
       5'h13 : ROW = 128'b000011010001111100001110000100000000100100011110000111100001001100011111000001000000000100001100000010000000101010001010100010001
       5'h14 : ROW = 128'b0001010100010001000010010001000000001001000100000000100000001000100010001000001000000000100001010000010000000100010001001100010001
       5'h15 : ROW = 128'b0001010100010001000010010001000100001001000100000000100000001000100010001000001000001001000010010000100000001000100010011000100001
       5'h16 : ROW = 128'b0000111000010001000111100000111000011110000111110001000000001110000100010000111000001100000100010001111100010001000100010000111
       5'h17 : ROW = 128'b00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
       5'h18 : ROW = 128'b0001111000001110000111100000111000011111000100010001000100010001000100010001000100011111000011100000000000001110000001000000000
       5'h19 : ROW = 128'b0001000100010001000100010001000100010000000100010001000100010001000100010001000100010000000100000001000000000010000010100000000
       5'h1A : ROW = 128'b00010001000100010001000100010000000001000000100010001000100010001000100001010000100010000000100000010000000100000000010001000100000000
       5'h1B : ROW = 128'b00011110000100010001111000001110000000100000100010001000100010101000000100000010100000010000000100000000100000000010000000000000000
       5'h1C : ROW = 128'b00010000000101010001010000000001000000100000010001000100010001010100001010000001000000100000001000000000100000000100000000000000000
       5'h1D : ROW = 128'b0001000000001001000010010000100010000001000000100010000101000011011000100010000001000000100000000010000000000100000010000000000000000
       5'h1E : ROW = 128'b00010000000011010001000100001110000000100000011100000010000010001000100010000010000011111000011100000000000001110000000000000111111
       5'h1F : ROW = 128'b00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
     endcase
```

# ROM vs. Verilog

```
assign      twist = {8{  nOE0}} & ROW[  7:  0]
                  | {8{  nOE8}} & ROW[ 15:  8]
                  | {8{ nOE16}} & ROW[ 23: 16]
                  | {8{ nOE24}} & ROW[ 31: 24]
                  | {8{ nOE32}} & ROW[ 39: 32]
                  | {8{ nOE40}} & ROW[ 47: 40]
                  | {8{ nOE48}} & ROW[ 55: 48]
                  | {8{ nOE56}} & ROW[ 63: 56]
                  | {8{ nOE64}} & ROW[ 71: 64]
                  | {8{ nOE72}} & ROW[ 79: 72]
                  | {8{ nOE80}} & ROW[ 87: 80]
                  | {8{ nOE88}} & ROW[ 95: 88]
                  | {8{ nOE96}} & ROW[103: 96]
                  | {8{nOE104}} & ROW[111:104]
                  | {8{nOE112}} & ROW[119:112]
                  | {8{nOE120}} & ROW[127:120];

assign      T = {twist[0],twist[1],twist[2],twist[3],twist[4],twist[5],twist[6],twist[7]};

endmodule // char10
```

# ROM vs. Verilog