Verilog is the Key Tool

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

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

Verilog as HDL

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

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
endmodule
  // 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

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)

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

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

```plaintext
code
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;
```
Funky Conditional

```v VHDL
cond_exp ? true_expr : false_expr
```

```vhdl
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
- `&`, `~&`, `|`, `~|`, `^`, `~^`, `^~`

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

Synthesis: Assign Statement

The assign statement is sufficient to create all combinational logic

What about this:

```vhdl
assign a = ~(b & c);
assign c = ~(d & a);
```
The assign statement is sufficient to create all combinational logic.

What about this:

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

// 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
// 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

---

**Simple Structural Module**

**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);`
- `bufi1 (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

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

    nand _i0(out, in1, in2);

    // include specify block for timing
    specify
        (in1 => out) = (1.0, 1.0);
        (in2 => out) = (1.0, 1.0);
    endspecify
endmodule
```
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
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

// 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;  // equivalent to 512 parallel connections
endspecify
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

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
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;
  ```

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

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

Procedural NAND gate

```plaintext
// 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
  A=B;
  B=A;
end

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

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

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

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

begin
B = A;
A = Y;
end

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; alusrc <= 0; alusrpb <= 2'b00;
  aluop <= 2'b00; pcsoure <= 2'b00; iord <= 0; memtoreg <= 0;
  case(state)
  FETCH1: 
    begin
      memread <= 1;
      irwrite <= 4'b0001; // change to reflect new memory
      alusrpb <= 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;
      alusrpb <= 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
This is an asynchronous clear D Flip-Flop

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

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

What is this?
What is synthesized?

> beh2str foo.v foo_str.v UofU_Digital.db
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?

```vhd
module nt (clk, rst, set, a, b, c, d, rout);
input clk, rst, set, a, b;
output c, d, rout;
wire [n2, n6, n8];

NAND2  (X, (A(b), B(b), Y(b))); 
XNOR2 (Y, Xn(b));
if (rst) Y <= 'b0;
else if (set) Y <= 'b1;
else Y <= D;
endmodule
```
Sythesis: Always Statement

```vhdl
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'.

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Type</th>
<th>Width</th>
<th>Bus</th>
<th>MB</th>
<th>AR</th>
<th>AS</th>
<th>SR</th>
<th>SS</th>
<th>ST</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q_reg</td>
<td>Flip-flop</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>
module foo ( clk, rst, set, D, Q );
  input clk, rst, set, D;
  output Q;
  wire   N3, n2, n4;

  dff Q_reg ( .D(N3), .G(clk), .CLR(n2), .Q(Q) );
  tiehi U6 ( .Y(n2) );
  nor2 U7 ( .A(rst), .B(n4), .Y(N3) );
  nor2 U8 ( .A(D), .B(set), .Y(n4) );
endmodule
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?

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

```verilog
module counter (clk, clr, load, in, count);
parameter width=8;
input clk, clr, load;
input [width-1 : 0] in;
output [width-1 : 0] count;
reg [width-1 : 0] tmp;

always @(posedge clk or negedge clr)
begin
if (!clr)
  tmp = 0;
else if (load)
  tmp = in;
else
  tmp = tmp + 1;
end
assign count = tmp;
endmodule
```

Synthesis: Modules

```verilog
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
```
// 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

module the_top (clk, rst, a, b, sel, result);
    parameter SIZE = 4;
    input clk, rst;
    input [SIZE-1:0] a,b;
    input [2:0] sel;
    output reg [SIZE-1:0] result;
    wire [SIZE-1:0] sum, dif, alu;

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

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

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

- Standard if-else-if syntax

\[
\text{If ( } <expression> \text{ )}
\]
\[
<statement>
\]
\[
\text{else if ( } <expression> \text{ )}
\]
\[
<statement>
\]
\[
\text{else if ( } <expression> \text{ )}
\]
\[
<statement>
\]
\[
\text{else } <statement>
\]

Priority vs. Parallel Choice (if)

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

// 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'h0;
    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

```verilog
module prior_enc(x, y, z, a, b, c, d, e, f);
  output seq x, y, z;
  input a, b, c, d, e, f;
  always@{a, b, c, d, e, f}
    begin
      x = y = ~c;
      if (a==1) && (b==1) s = 1;
      if (c==1) && (d==1) y = 1;
      if (a==1) && (i==1) x = 1;
    end
endmodule // prior_enc
```

Case Statements

- Multi-way decision on a single expression

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

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

Another Case Example

```verilog
// simple counter next-state logic
// one-hot state encoding...
parameter [2:0] s0=3'h1, s1=3'h2, s2=3'h4;
reg[2:0] state, next_state;
always @(input or state)
begin
  case (state)
    s0: if (input) next_state = s1;
      else next_state = s0;
    s1: next_state = s2;
    s2: next_state = s0;
  endcase
endcase
```
Verilog allows you to put a value in the case slot, and test which variable currently has that value…

```verilog
reg [2:0] curr_state, next_state;
parameter s1=3'b001, s2=3'b010, s3=3'b100

case (1)
  curr_state[0] : next_state = s2;
  curr_state[1] : next_state = s3;
  curr_state[2] : next_state = s1;
endcase
```

**Latch Inference**

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

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

- This infers a latch because it doesn’t specify what to do when `cond = 0`
  - Fix by adding an `else`
  - 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
Note that full-par results in combinational logic.

**Synthesis Result**

- A latch is synthesized because case is not full.

```verilog
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
```
Because it’s not full, a latch is inferred…

```verilog
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
```

// 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
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

```verilog
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

We’ve only just begun

Driving add
CASEZ. Opcode: add
CASEX. Opcode: add

Driving subtract
CASEZ. Opcode: subtract
CASEX. Opcode: subtract

Driving multiply
CASEZ. Opcode: multiply
CASEX. Opcode: multiply

L18 "testfixture.new": $fin

Get off my Case

We’ve only just begun

Driving 0
CASEEa: Logic 0 on sel
CASEER: Logic 0 on sel
CASEX: Logic 0 on sel
CASE: Logic 0 on sel

Driving 1
CASE: Logic 1 on sel
CASEER: Logic 1 on sel
CASEX: Logic 1 on sel
CASEEa: Logic 1 on sel

Driving x
CASEEa: Logic x on sel
CASEER: Logic x on sel
CASEX: Logic x on sel
CASE: Logic x on sel

Order Matters
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

Mealy only

Next state Logic

State

outputs

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
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 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
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
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

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

Combinational Always Blocks

- Be careful…

```verilog
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)         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?
- 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

![Finite State Machine Diagram]

Textbook FSM

```verbatim
// Verilog HDL for "Ax", "exef" "behavioral"
// Post in a row detector - Allen Tew

module post (clk, cke, inmsg, reset);
input clk, cke, inmsg;
output reset;

// 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 var 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 regwrite clk)
begin
  if (regwrite == 1) state = state;
  else state = next_state;
end

// define combinational logic for next state
always @(regwrite clk)
begin
  case (state)
    s0: if (inmsg) next_state = s1;
        else next_state = s0;
    s1: if (inmsg) next_state = s2;
        else next_state = s0;
    s2: if (inmsg) next_state = s3;
        else next_state = s2;
    s3: if (inmsg) next_state = s4;
        else next_state = s3;
    s4: if (inmsg) next_state = s0;
        else next_state = s4;
    default: next_state = s0;
  endcase
end

// now cut the reset. This could also be done in an always
// block...but in that case, reg4 would have to be
// defined as a reg.
assign reg4 = state == s0;
endmodule
```
### Textbook FSM

```plaintext
// Verilog HDL for "Aam", "read", "behavioral"
// "Four in a Row Detector" - Allen Tannen
module aam4 (clk, clr, insig, saw4);
    input clk, clr, insig;
    output saw4;
    // define state encodings as parameters
    parameter [10:0] s0 = 'b0000, s1 = 'b0010, s2 = 'b0100, s3 = 'b1000, s4 = 'b1001;
    // define reg values 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) 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;
                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
```

### Documented FSM

```plaintext
// Verilog HDL for "Aam", "read", "behavioral"
// "Four in a Row Detector" - Allen Tannen
module aam4 (clk, clr, insig, saw4);
    input clk, clr, insig;
    output saw4;
    parameter [2:0] s0 = 'b0000, // initial state: saw at least 1 one
                    s1 = 'b0010, // saw 1 one
                    s2 = 'b0100, // saw 2 ones
                    s3 = 'b0110, // saw 3 ones
                    s4 = 'b1001; // saw at least 4 ones
    reg [2:0] state, next_state;
    always @(posedge clk or negedge 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;
                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
```
Waveform Test Bench

```verilog
// Four ones in a row detector.
// Test bench
// Allen Tannen

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

    send_message(32'h0011_1000_1010_1111_0000_0111_1110_0000);
    send_message(32'h0011_1000_0110_1111_0000_0111_1110_0000);
    $finish;
end
always @ (posedge clk or negedge clt)
begin
    $nc0 clk = ~clk;
end

initial
begin
    #500 clt = 1'b1;
    #500 clt = 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
```

Waveform

Pay attention to first few cycles...
module meet (clk, clr, insig, saw4);
  input clk, clr, insig;
  output saw4;
  reg [2:0] state, next_state;
  assign saw4 = state == 4;
endmodule
// Verilog HDL for "Ax", "seq" "Behavioral"
// Four in a row detector - Allen Turner
module seq4 (clk, clr, inseq, saw4);
  input clk, clr, inseq;
  output saw4;
parameter [1:0] s0 = 3'b000; // initial state, saw at least 1 zero
parameter [1:0] s1 = 3'b001; // saw 1 one
parameter [1:0] s2 = 3'b010; // saw 2 ones
parameter [1:0] s3 = 3'b011; // saw 3 ones
parameter [1:0] s4 = 3'b100; // saw at least, 4 ones
reg [3:0] state;
always @(posedge clk or posedge clr) // state register
begin
  if (clr)
    state <= s0;
  else state <= (state == s0 & inseq) & s0
    | (state == s1 & inseq) & s1
    | (state == s2 & inseq) & s2
    | (state == s3 & inseq) & s3
    | (state == s4 & inseq) & s4;
end
assign saw4 = state == s4;
endmodule

// Verilog HDL for "Ax", "seq" "behavioral"
// Four in a row detector - Allen Turner
module one_hot_seq4 (clk, clr, inseq, saw4);
  input clk, clr, inseq;
  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)
    s0 <= 1'b1;
  else begin
    s1 <= s1 & inseq;
    s2 <= s2 & inseq;
    s3 <= s3 & inseq;
    s4 <= s4 & inseq;
    assign saw4 = s4;
  end
endmodule
One-Hot FSM Counting

```verilog
module sees4 (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;

  \*\*abella*  s0_reg ( .next_state[n14], .clocked_on(clk), .force_00(n11),
      .force_10(clk), .force_11(n11), .Q(s0) );
  ODD s1_reg ( .D(N1), .CLK(clk), .nCLR(n15), .Q(s1) );
  ODD s2_reg ( .D(N2), .CLK(clk), .nCLR(n15), .Q(s2) );
  ODD s4_reg ( .D(N3), .CLK(clk), .nCLR(n15), .Q(saw4), .Q(saw9) );
  ODD s8_reg ( .D(N4), .CLK(clk), .nCLR(n15), .Q(saw16), .Q(sa21) );
  INV U20 ( .Y(clk) );
  INV U21 ( .A(clk) );
  AND U22 ( .A(n15), .B(n17), .C(insig), .Y(n14) );
  INV U23 ( .A(s2), .B(s1), .Y(n17) );
  INV U24 ( .A(m0), .B(m15), .Y(n16) );
  INV U25 ( .A(m9), .Y(m5) );
  NAND U26 ( .A(n18), .B(insig), .Y(n19) );
  NAND U27 ( .A(n9), .B(n10), .Y(n18) );
  INV U28 ( .A(n20), .Y(n24) );
  NAND U29 ( .A(insig), .B(n3), .Y(n20) );
  INV U30 ( .A(n22), .Y(m3) );
  NAND U31 ( .A(insig), .B(n1), .Y(n21) );
  INV U32 ( .A(n21), .Y(n2) );
  NAND U33 ( .A(insig), .B(s0), .Y(n22) );
endmodule
```
No Asynchronous Sets

```verilog
module saw4 (clk, cl2, in4, saw4);
  input clk, cl2, in4;
  output saw4;
  reg ms0; // 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 ms0; // alias for !in4 (ms0 used to avoid YFGEN in behstr)
  assign ms0 = !ms4;
  always @ (posedge clk or posedge cl2) // state register
  begin
    if (cl2)
      begin
        ms0 <= 1'b0;
        s1 <= 1'b0;
        s2 <= 1'b0;
        s3 <= 1'b0;
        s4 <= 1'b0;
        end
    else
      begin
        ms0 <= ~((s0 | s1 | s2 | s3) | ms4) & ~in4;
        s1 <= s0 & in4;
        s2 <= s1 & in4;
        s3 <= s2 & in4;
        s4 <= s3 & in4;
        end
  end
  assign saw4 = s4;
endmodule
```

That's better

```verilog
module saw4 (clk, cl2, in4, saw4);
  input clk, cl2, in4;
  output saw4;
  wire ms0, N0, X1, N2, N4, N5, N6, n9, n10, n11, n12, n13, n14, n15;
  reg s1_reg = .D(N5), .CLK(clk), .xCLR(n1), .Q(saw4), .QB(n9) ;
  reg s2_reg = .D(N2), .CLK(clk), .xCLR(n1), .QB(n12) ;
  reg s3_reg = .D(N1), .CLK(clk), .xCLR(n1), .QB(n10) ;
  reg s4_reg = .D(N0), .CLK(clk), .xCLR(n1), .QB(n11) ;
  reg ms0_reg = .D(N4), .CLK(clk), .xCLR(n1), .Q(ms4) ;
  INV U12 (.A(clir), .Y(n11) ) ;
  AGI U13 (.A(n9), .B(n12), .C(n13), .Y(N5) ) ;
  GAL U14 (.A(n14), .B(n15), .C(n13), .Y(N4) ) ;
  NAND2 U15 (.A(ms0), .B(n12), .Y(n15) ) ;
  NAND3 U16 (.A(n10), .B(n9), .C(n11), .Y(n14) ) ;
  XOR2 U17 (.A(n13), .B(n10), .Y(N2) ) ;
  XOR2 U18 (.A(n13), .B(n11), .Y(N1) ) ;
  XOR2 U19 (.A(n5), .B(n13), .Y(N0) ) ;
  INV U20 (.A(in4), .Y(n13) ) ;
endmodule
```
Synchronous Clear

module s4 (clk, clr, insig, saw4);
    input clk, clr, insig;
    output saw4;
    reg s0; // initial state, saw at least 1 sec
    reg s1; // saw 1 sec
    reg s2; // saw 2 sec
    reg s3; // saw 3 sec
    reg s4; // saw at least 4 sec
    always @posedge clk // state register with synchronous clear
        begin
            if (clr)
                s0 <= 1'b0;
                s1 <= 1'b0;
                s2 <= 1'b0;
                s3 <= 1'b0;
                s4 <= 1'b0;
            else begin
                s0 <= (s0 | s1 | s2 | s3 | s4) & !insig;
                s1 <= s0 & !insig;
                s2 <= s1 & !insig;
                s3 <= s2 & !insig;
                s4 <= s3 & !insig;
            end
        end
    assign saw4 = s4;
endmodule
Is asynchronous clear really asynchronous?

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

ROM vs. Verilog
### ROM vs. Verilog

The image contains a table comparing ROM and Verilog. The table includes rows and columns with hexadecimal values, indicating a comparison of memory addresses and values.

#### ROM

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>0000</td>
</tr>
<tr>
<td>00001</td>
<td>0001</td>
</tr>
<tr>
<td>00010</td>
<td>0002</td>
</tr>
<tr>
<td>00011</td>
<td>0003</td>
</tr>
<tr>
<td>00020</td>
<td>0004</td>
</tr>
<tr>
<td>00021</td>
<td>0005</td>
</tr>
<tr>
<td>00030</td>
<td>0006</td>
</tr>
<tr>
<td>00031</td>
<td>0007</td>
</tr>
</tbody>
</table>

#### Verilog

```verilog
module ROM (input [7:0] address, output [7:0] data);
    always @(*) begin
        case(address)
            8'h00: data <= 8'h00;
            8'h01: data <= 8'h01;
            8'h02: data <= 8'h02;
            8'h03: data <= 8'h03;
            8'h04: data <= 8'h04;
            8'h05: data <= 8'h05;
            8'h06: data <= 8'h06;
            8'h07: data <= 8'h07;
            default: data <= 8'hFF;
        endcase
    end
endmodule
```
ROM vs. Verilog

```verilog
assign twist = 
(0) 0000 & ROM: 0
(1) 0001 & ROM: 1
(2) 0010 & ROM: 10
(3) 0011 & ROM: 11
(4) 0100 & ROM: 10
(5) 0101 & ROM: 11
(6) 0110 & ROM: 10
(7) 0111 & ROM: 11
(8) 1000 & ROM: 10
(9) 1001 & ROM: 11
(10) 1010 & ROM: 10
(11) 1011 & ROM: 11
(12) 1100 & ROM: 10
(13) 1101 & ROM: 11
(14) 1110 & ROM: 10
(15) 1111 & ROM: 11

assign T = [twist:0..twist:7],

module echo // char10
```