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
  - The wire-list function fits into HDL
  - Testbench creation language
    - Create external test environment
      - Time & Voltage
      - Files & messages
  - Are these two tasks
    - Related?
    - Compatible?

Verilog as HDL

- Want high level modeling
  - Unification at all levels
  - From fast functional simulation, accurate device simulation
  - Support simulation and formal 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
  <continuous assignments>
  <always blocks>
endmodule
Quick Review

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

Verilog Description Styles

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

Synthesis: Data Types

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

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;

Parameters

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

```verilog
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

```verilog
wire [3:0] sum, a, b; wire cin, cout, d, e, f, g;
assign sum = f & a;
assign sum = f | a;
assign sum = (4‘b1) | b;
assign sum = (4’b1 == g) & (a + b);
assign sum[0] = g & a[2];
assign sum[2:0] = (3’b1) & 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
  ```verilog
  assign a = b >> 2; // shift right 2, division by 4
  assign a = b << 1; // shift left 1, multiply by 2
  ```
- Arithmetic
  ```verilog
  assign a = b * c; // multiply b times c
  assign a = b * 'd2; // multiply b times constant (=2)
  assign a = b / 'b10; // divide by 2 (constant only)
  assign a = b % 'h3; // b modulo 3 (constant only)
  ```

Synthesis: Extra Operators

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

Synthesis: Assign Statement

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

What about this:

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

### Simple Behavioral Module

// Behavioral model of NAND gate
module NAND (out, in1, in2);
  output out;
  input in1, in2;
  // Uses Verilog built-in nand function
  // Syntax is func id (args);
  nand i0(out, in1, in2);
endmodule

### 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(.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)

- 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);

**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 ) = 6; // delay from c to q
    ( d => q ) = 6; // delay from d to q
  endspecify
  // module definition
  or o1( e, a, b );
  or o2( f, c, d );
  exor ex1( q, e, f );
endmodule
Full Specify

```verilog
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 0( e, a, b );
or_or( f, c, d );
exor ex1( q, e, f);
endmodule
```

Specify needs a gate!

```verilog
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

```verilog
// 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 Assignments**

- Assigns values to reg types
  - Only useable inside a procedural block Usually synthesizes to a register
  - But, under the right conditions, can also result in combinational circuits
- Blocking procedural assignment
  - LHS = timing-control exp  \( a = \#10 \ 1; \)
  - Must be executed before any assignments that follow (timing control is optional)
  - Assignments proceed in order even if no timing is given
- Non-Blocking procedural assignment
  - LHS <= timing-control exp  \( b <= 2; \)
  - Evaluated simultaneously when block starts
  - Assignment occurs at the end of the (optional) time-control

**Procedural Synthesis**

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

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

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

**Synthesized Circuits**

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

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

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

**Assignments and Synthesis**

- Note that different circuit structures result from different types of procedural assignments
  - Therefore you can't mix assignment types in the same always block
  - And you can't use different assignment types to assign the same register either
  - Non-blocking is often a better model for hardware
    - Real hardware is often concurrent...

**Comparator Example**

- Using 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
  - 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
```

- What is this?
- What is synthesized?

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

---

```
{}\n```

---

```
{}\n```

---

```
{}\n```

---

```
{}\n```

---

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

What is this?
What is synthesized?

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

D
clk
Q
n4
N3
set
rst
clr

A B Out
0 0 1
0 1 0
1 0 0
1 1 0

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 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

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

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

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

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

Example: Counter

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

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: 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
**Synthesis: Modules**

```verilog
module the_top (clk, rst, a, b, sel, result);
    parameter SIZE = 4;
    input clk, rst;
    input [SIZE-1:0] a,b;
    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**

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

**Priority vs. Parallel Choice (if)**

```verilog
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**

```verilog
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**

```verilog
module priority_enc(xs, y, z, a, b, c, d, e, f);
    output reg m, y, z;
    always @ (xs or y or z or a or b or c or d or e or f)
    begin
        Ce = y = z = 'b0;
        if (xs) {x=1} & {y=1} & {z=1} & {m=1};
        else if ((xs) & (c=1) & (d=1) & {f=1}) m = 1;
        else if ((xs) & (c=1) & {f=1} & {e=1}) m = 1;
    end
endmodule // priority_enc
```

```verilog
module priority_enc(xs, y, z, a, b, c, d, e, f);
    output reg m, y, z;
    always @ (xs or y or z or a or b or c or d or e or f)
    begin
        Ce = y = z = 'b0;
        if (xs) {x=1} & {y=1} & {z=1} & {m=1};
        else if ((xs) & (c=1) & (d=1) & {f=1}) m = 1;
        else if ((xs) & (c=1) & {f=1} & {e=1}) m = 1;
        else if ((xs) & (c=1) & {f=1} & {e=1}) m = 1;
    end
endmodule // priority_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
end
```

Weird Case Example

```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

```verilog
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

```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
```

Synthesized Circuit

- Because it's not full, a latch is inferred…

Case full-notpar example

```verilog
module full-notpar (slct, a, b, c, out);
input [1:0] slct;
input a, b, c;
output out;
reg out;
always @(slct or a or b or c)
  casez (slct)
    2'b1? : out <= a;
    2'b0? : 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

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

We've only just begun

Driving add
CASEX. Opcode: add
CASEX. Opcode: subtract
CASEX. Opcode: multiply
CASEX. Opcode: multiply

Get off my Case

Order Matters

Mealy only
Modeling State Machines

```verilog
// 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;
// state register
always @posedge (clk)
state = next_state;
// next-state logic
always @(state or in)
begin
case (state)
    s0: if (in) next_state = s1;
        else next_state = s0;
    s1: if (in) next_state = s2;
        else next_state = s1;
    s2: if (in) next_state = s3;
        else next_state = s2;
    s3: if (in) next_state = s1;
        else next_state = s0;
endcase
end
endmodule
```

FSM Description

1. **Module FSM**: The module `FSM` is defined with inputs `clk`, `in`, and output `out`. The state registers `state` and `next_state` are declared as `reg [1:0]`.
2. **State Register**: The state register is updated using the `posedge` of `clk`.
3. **Next-State Logic**: The next-state logic is determined by the current state and the input `in`. The case statement evaluates each state and assigns the next state accordingly.

Verilog Version

```verilog
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
reg [1:0] state, next_state;
//define state register (with asynchronous 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 Continued...

```verilog
// 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...

```verilog
// 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

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

Combinational Always Blocks

- Be careful…

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

- Which one is a good mux?

Sync vs. Async Register Reset

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

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

Finite State Machine

Textbook FSM

```plaintext
// state diagram from: "Textbook: Computer Organization and Design"

module fsm (clk, rst, in0, in1);
  input clk, rst, in0, in1;
  output reg state;
  reg [3:0] state;
  always @(posedge clk or negedge rst)
    if (rst == 1) state = 0;
    else state = next_state;

  assign next_state = sel_state;

  always_comb (
    sel_state = state;
    if (in0 == 1) state = s0;
    else state = s1;
  )

  initial begin
    state = s1;
  end

endmodule
```

```
```
Textbook FSM

```verilog
module textbook FSM #(clk, rst, in, en, out);
    input clk, rst, in, en;
    output out;
    reg [3:0] state;
    always @(posedge clk or posedge rst) begin
        if (rst) #10 state <= 0;
        else if (en) state <= state + in;
    endmodule
```

---

Documented FSM

```verilog
module documented FSM #(clk, rst, in, en, out);
    input clk, rst, in, en;
    output out;
    reg [3:0] state;
    always @(posedge clk or posedge rst) begin
        if (rst) #10 state <= 0;
        else if (en) state <= state + in;
    endmodule
```

---

Waveform Test Bench

```verilog
// Test bench for FSM
module test FSM #(clk, rst, in, en, out);
    input clk, rst, in, en;
    output out;
    reg [3:0] state;
    always @(posedge clk or posedge rst) begin
        if (rst) #10 state <= 0;
        else if (en) state <= state + in;
    endmodule
```

---

Waveform

Pay attention to first few cycles...

---

FSM

```verilog
// FSM
module FSM #(clk, rst, in, en, out);
    input clk, rst, in, en;
    output out;
    reg [3:0] state;
    always @(posedge clk or posedge rst) begin
        if (rst) #10 state <= 0;
        else if (en) state <= state + in;
    endmodule
```
Is asynchronous clear really asynchronous?

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