Advanced Verilog Techniques Workshop

Clifford E. Cummings
Sunburst Design, Inc.
cliffc@sunburst-design.com
www.sunburst-design.com

Expert Verilog HDL & Verilog Synthesis Training

Agenda

• The two details most misunderstood about Verilog

• A couple of important Verilog tips

• Efficient Verilog FSM coding styles

• Verilog-2001 enhancements
  – Enhancements already implemented by ModelSim

• ModelSim quirks & tricks

Papers with more details about the Verilog Seminar topics can be downloaded from:
www.sunburst-design.com/papers
The Two Details Most Misunderstood About Verilog

Net-types vs. register-types
Blocking vs. nonblocking assignments

Introducing: the Verilog event queue

Use of Reg Types Vs. Net Types

- Register types are only assigned inside of procedural blocks
  
  ```verilog
  reg [8:0] sum;
  always @(a or b)
  sum = a + b;
  ```

- Net types are assigned or driven outside of procedural blocks
  
  ```verilog
  wire [8:0] sum;
  assign sum = a + b;
  ```

  ```verilog
  wire y;
  and g1(y, c, d);
  ```
Blocking & Nonblocking Assignments

- Verilog "race" conditions
- Blocking and nonblocking assignment fundamentals
- 7 Guidelines to help avoid Verilog race conditions
- Verilog's "Stratified Event Queue"
- Examples:
  - Sequential pipeline
  - Sequential LFSR (Linear Feedback Shift Register)
  - Combining blocking & nonblocking assignments
- Common misconceptions about NBAs

"Nonblocking" Is Not A Typo

- The word nonblocking is frequently misspelled as:
  - non-blocking
  - Misspelled with a "-"
  - Misspelled in most Verilog books
- The correct spelling as noted in the IEEE 1364-1995 Verilog Standard is:
  - nonblocking
  - Spell-checkers do not recognize Verilog keywords
  
IEEE Std 1364-1995
Section 5.6.4 Nonblocking assignment

From the IEEE Verilog Standard itself
Verilog Race Conditions

- The IEEE Verilog Standard defines:
  - which statements have a guaranteed order of execution
    ("Determinism", section 5.4.1)
  - which statements do not have a guaranteed order of execution
    ("Nondeterminism", section 5.4.2 & "Race conditions", section 5.5)

- A Verilog race condition occurs when two or more statements that are scheduled to execute in the same simulation time-step, would give different results when the order of statement execution is changed, as permitted by the Verilog Standard

- To avoid race conditions, it is important to understand the scheduling of Verilog blocking and nonblocking assignments

Blocking Assignments

- Blocking assignment operator: =

- Execution of blocking assignments is a one-step process:
  - Evaluate the RHS (right-hand side argument) and update the LHS (left-hand side argument) of the blocking assignment without interruption from any other Verilog statement

  "Blocks" trailing assignments in the same always block from occurring until after the current assignment has completed
Inter-Dependent Blocking Assignments

- Problem: Inter-dependent blocking assignments that execute in the same time step

```
module fbosc1 (y1, y2, clk, rst);
output y1, y2;
input  clk, rst;
reg    y1, y2;
always @(posedge clk or posedge rst)
if (rst) y1 = 0;  // reset
else     y1 = y2;
always @(posedge clk or posedge rst)
if (rst) y2 = 1;  // preset
else     y2 = y1;
endmodule
```

After rst, on next posedge clk,
y1=1 (y2) and y2=y1=1
-OR-
y2=0 (y1) and y1=y2=0

Execution order is not guaranteed!

Which simulation model???
Both are wrong!!

Nonblocking Assignments

- Nonblocking assignment operator: <=
- Execution of nonblocking assignments can be viewed as a two-step process:
  1. Evaluate the RHS of nonblocking statements at the beginning of the time step
  2. Update the LHS of nonblocking statements at the end of the time step
- Allows assignment scheduling without blocking evaluation and execution of other Verilog statements
- Only used in procedural blocks (not continuous assignments)

```
assign y <= a + b;
```

Syntax error! Illegal continuous assignment
Inter-Dependent Nonblocking Assignments

• No problem: Inter-dependent nonblocking assignments that execute in the same time step

```verilog
module fbosc2 (y1, y2, clk, rst);
  output y1, y2;
  input  clk, rst;
  reg    y1, y2;
  always @(posedge clk or posedge rst)
    if (rst) y1 <= 0; // reset
    else     y1 <= y2;
  always @(posedge clk or posedge rst)
    if (rst) y2 <= 1; // preset
    else     y2 <= y1;
endmodule
```

On rst, y1=0 and y2=1

Simulation & synthesis result

After rst, on next posedge clk, y1=1 (y2') and y2=0 (y1')

No race condition!

7 Coding Guidelines

• In general, following specific coding guidelines can eliminate Verilog race conditions:

  Guideline #1: Sequential logic and latches - use nonblocking assignments

  Guideline #2: Combinational logic in an always block - use blocking assignments

  Guideline #3: Mixed sequential and combinational logic in the same always block is sequential logic - use nonblocking assignments

  Guideline #4: In general, do not mix blocking and nonblocking assignments in the same always block

  Guideline #5: Do not make assignments to the same variable from more than one always block

  Guideline #6: Use $strobe to display values that have been assigned using nonblocking assignments

  Guideline #7: Do not make #0 procedural assignments

Follow these guidelines to remove 90-100% of all Verilog race conditions
### IEEE1364-1995 Verilog

**Stratified Event Queue**

- **Blocking assignments**: Evaluate RHS of nonblocking assignments
- **Continuous assignments**: $\$display$ command execution
- **Nonblocking assignments**: Update LHS of nonblocking assignments
- **Monitor events**: $\$monitor$ command execution
- **Strobe events**: $\$strobe$ command execution
- **Other specific PLI commands**: These events may be scheduled in any order

#### Guideline #7: do not use #0 delays

### Self-Triggering Oscillators?

#### Oscillator 1 (osc1)
```verilog
module osc1 (clk);
  output clk;
  reg clk;
  initial
    #10 clk = 0;
  always @(clk)
    #10 clk = ~clk;
endmodule
```

#### Oscillator 2 (osc2)
```verilog
module osc2 (clk);
  output clk;
  reg clk;
  initial
    #10 clk = 0;
  always @(clk)
    #10 clk <= ~clk;
endmodule
```

#### Self-Triggering Oscillators

<table>
<thead>
<tr>
<th>osc1</th>
<th>osc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: clk1=x clk2=x</td>
<td>0: clk1=0 clk2=0</td>
</tr>
<tr>
<td>10: clk1=0 clk2=0</td>
<td>20: clk1=1 clk2=1</td>
</tr>
<tr>
<td>30: clk1=1 clk2=0</td>
<td>40: clk1=1 clk2=1</td>
</tr>
<tr>
<td>50: clk1=1 clk2=0</td>
<td>60: clk1=1 clk2=1</td>
</tr>
<tr>
<td>70: clk1=1 clk2=0</td>
<td>80: clk1=1 clk2=1</td>
</tr>
<tr>
<td>90: clk1=1 clk2=0</td>
<td></td>
</tr>
</tbody>
</table>

**NOTE**: contrived examples - not the recommended coding style for clock oscillators!
Self-Triggering?
(not with blocking assignments)

osc2: nonblocking assignment

initial
#10 clk = 0;
always @(clk)
#10 clk <= ~clk;

Active Events
#10 clk=0 Blocking Assignment
@ (clk) Trigger always block
(RHS=1) Evaluate RHS of NBA @ (clk)
wait for clk transition
activate NB events
#10 clk=1 NBA assignment
(RHS=0) Evaluate RHS of NBA
(continues) activate NB events
@ (clk) Wait for clk transition
oscillating!
clk<=0 Update LHS of NBA
clk<=1 NBA assignment
(triggers @(clk)) activate NB events

osc1: blocking assignment

initial
#10 clk = 0;
always @(clk)
#10 clk = ~clk;

Active Events
#10 clk<=0 Blocking Assignment
@ (clk) Trigger always block
#10 clk=clk=1 Blocking Assignment
@ (clk) Wait for clk transition
stopped!

Pipeline
Blocking Assignments Style #1

module pipeb1 (q3, d, clk);
output [7:0] q3;
input  [7:0] d;
input        clk;
reg    [7:0] q3, q2, q1;
always @(posedge clk) begin
  q1 = d;
  q2 = q1; // d
  q3 = q2; // d
end
endmodule

q1 = d; Blocking assignment
q2 = q1; // d Blocking assignment
q3 = q2; // d Blocking assignment

NB Events

Monitor Events

Simulation result

Synthesis result!!

Three sequential blocking assignments but only one synthesized register
Pipeline
Blocking Assignments Style #2

module pipeb2 (q3, d, clk);
output [7:0] q3;
input [7:0] d;
input clk;
reg [7:0] q3, q2, q1;

always @(posedge clk) begin
    q3 = q2;
    q2 = q1;
    q1 = d;
end
endmodule

Simulation result

Synthesis result

Pipeline
Blocking Assignments Style #3

module pipeb3 (q3, d, clk);
output [7:0] q3;
input [7:0] d;
input clk;
reg [7:0] q3, q2, q1;

always @(posedge clk) begin
    q3 = q2;
    q2 = q1;
    q1 = d;
end
endmodule

Simulation result

Synthesis result
**Pipeline**

Blocking Assignments Style #4

- **Active Events**
  - `q2 = q1;`
  - `q3 = q2; // q1`
  - `q1 = d; // q1`

- **NB Events**

- **Monitor Events**

- **Executes in any order**

```verilog
module pipeb4 (q3, d, clk);
output [7:0] q3;
input [7:0] d;
input clk;
reg [7:0] q3, q2, q1;

always @(posedge clk) q2 = q1;
always @(posedge clk) q3 = q2; // q1
always @(posedge clk) q1 = d;
endmodule
```

**Simulation result**

**Synthesis result**

This ordering might not work

**Blocking Assignment Coding Styles (Pipelines)**

- **Pipeline Coding Styles**

<table>
<thead>
<tr>
<th>Coding Style</th>
<th>Simulates correctly</th>
<th>Synthesizes correctly</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blocking Assignments #1</td>
<td>NO!</td>
<td>NO!</td>
</tr>
<tr>
<td>Blocking Assignments #2</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Blocking Assignments #3</td>
<td>?</td>
<td>Yes</td>
</tr>
<tr>
<td>Blocking Assignments #4+</td>
<td>?</td>
<td>Yes</td>
</tr>
</tbody>
</table>

1 of 4 simulates correctly

3 of 4 synthesize correctly
Pipeline
Nonblocking Assignments Style #1

Active Events
(RHS=d')
(RHS=q1')
(RHS=q2')

NB Events
q1 <= d'
q2 <= q1'
q3 <= q2'

Monitor Events

Simulation result

Synthesis result

module pipen1 (q3, d, clk);
output [7:0] q3;
input  [7:0] d;
input        clk;
reg    [7:0] q3, q2, q1;
always @(posedge clk) begin
    q1 <= d;
    q2 <= q1;
    q3 <= q2;
end
endmodule

Pipeline
Nonblocking Assignments Style #2

Active Events
(RHS=q2')
(RHS=q1')
(RHS=d')

NB Events
q3 <= q2'
q2 <= q1'
q1 <= d'

Monitor Events

Simulation result

Synthesis result

module pipen2 (q3, d, clk);
output [7:0] q3;
input  [7:0] d;
input        clk;
reg    [7:0] q3, q2, q1;
always @(posedge clk) begin
    q3 <= q2;
    q2 <= q1;
    q1 <= d;
end
endmodule
Pipeline
Nonblocking Assignments Style #3

Active Events
(RHS=d') ← Evaluate RHS of NBA
(RHS=q1') ← Evaluate RHS of NBA
(RHS=q2') ← Evaluate RHS of NBA

NB Events
q1 <= d' ← Update LHS of NBA
q2 <= q1' ← Update LHS of NBA
q3 <= q2' ← Update LHS of NBA

Monitor Events

Simulation result

Synthesis result

This ordering will work

Pipeline
Nonblocking Assignments Style #4

Active Events
(RHS=q1') ← Evaluate RHS of NBA
(RHS=q2') ← Evaluate RHS of NBA
(RHS=d') ← Evaluate RHS of NBA

NB Events
q2 <= q1' ← Update LHS of NBA
q3 <= q2' ← Update LHS of NBA
q1 <= d' ← Update LHS of NBA

Monitor Events

Simulation result

Synthesis result

ANY NEW ordering will work
Nonblocking Assignment Coding Styles (Pipelines)

### Pipeline Coding Styles

<table>
<thead>
<tr>
<th>Coding Style</th>
<th>Simulates correctly</th>
<th>Synthesizes correctly</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blocking Assignments #1</td>
<td>NO!</td>
<td>NO!</td>
</tr>
<tr>
<td>Blocking Assignments #2</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Blocking Assignments #3</td>
<td>?</td>
<td>Yes</td>
</tr>
<tr>
<td>Blocking Assignments #4+</td>
<td>?</td>
<td>Yes</td>
</tr>
<tr>
<td>Nonblocking Assignments #1</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Nonblocking Assignments #2</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Nonblocking Assignments #3</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Nonblocking Assignments #4+</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

1 of 4 simulate correctly
3 of 4 synthesize correctly
4 of 4 simulate correctly
4 of 4 synthesize correctly

Careful Coding with Blocking Assignments

- Careful usage of blocking assignments can work
  - But why work this hard??

- Single flipflop modules will work with blocking assignments
  - Bad habit - better to consistently code sequential logic with nonblocking assignments

- Coding sequential logic with feedback loops
  - Additional problems when using blocking assignments
  - LFSR (Linear Feedback Shift Register) examples

```verilog
module dffb (q, d, clk, rst);
    output q;
    input  d, clk, rst;
    reg    q;
    always @(posedge clk)
      if (rst) q = 1'b0;
      else     q = d;
endmodule
```

This example is in most Verilog books (bad habit!)
Bad coding style!
LFSR
Blocking Assignments Style #1

module lfsrbl (q3, clk, pre_n);
output q3;
input clk, pre_n;
reg q3, q2, q1;
wire n1;
assign n1 = q1 ^ q3;
always @(posedge clk or negedge pre_n)
if (!pre_n) begin
  q3 = 1'b1;
  q2 = 1'b1;
  q1 = 1'b1;
end
else begin
  q3 = q2;
  q2 = n1;
  q1 = q3;
end
endmodule

Active Events
q3=1
q2=1
q1=1
n1=(1^1)=0
q3=(q2)=1
q2=(n1)=0
q1=(q3)=1
n1=(1^1)=0
assign n1=q1^q3
q3=q2
q2=n1
q1=q3

Wrong value! Should be "1"

LFSR
Blocking Assignments Style #2

module lfsrbl2 (q3, clk, pre_n);
output q3;
input clk, pre_n;
reg q3, q2, q1;
always @(posedge clk or negedge pre_n)
if (!pre_n) (q3, q2, q1) = 3'b111;
else (q3, q2, q1) = (q2, (q1^q3), q3);
endmodule

Active Events
q1,q2,q3=111
q1,q2,q3=101
q1,q2,q3=010
...
module lfsrn1 (q3, clk, pre_n);
output q3;
input clk, pre_n;
reg q3, q2, q1;
wire n1;
assign n1 = q1 ^ q3;
always @ (posedge clk or negedge pre_n)
    if (!pre_n) begin
        q3 <= 1'b1;
        q2 <= 1'b1;
        q1 <= 1'b1;
    end
    else begin
        q3 <= q2;
        q2 <= n1;
        q1 <= q3;
    end
endmodule
LFSR
Nonblocking Assignments Style #2

```verilog
module lfsrn2 (q3, clk, pre_n);
    output q3;
    input  clk, pre_n;
    reg    q3, q2, q1;

    always @(posedge clk or negedge pre_n)
        if (!pre_n) {q3,q2,q1} <= 3'b111;
        else        {q3,q2,q1} <= {q2,(q1^q3),q3};
endmodule
```

**Synthesizes and simulates correctly**

... but again, a cryptic coding style

---

Nonblocking Assignment Coding Styles (LFSRs)

<table>
<thead>
<tr>
<th>Coding Style</th>
<th>Simulates correctly</th>
<th>Synthesizes correctly</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blocking Assignments #1</td>
<td>NO!</td>
<td>Yes</td>
</tr>
<tr>
<td>Blocking Assignments #2</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Nonblocking Assignments #1</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Nonblocking Assignments #2</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Cryptic coding style

1 of 2 simulates correctly

2 of 2 synthesize correctly

2 of 2 simulate correctly

2 of 2 synthesize correctly
Nonblocking Assignment Guidelines #1 & #2

- **Guideline #1:** Use **nonblocking** assignments in always blocks that are written to generate **sequential and latching logic**

- **Guideline #2:** Use **blocking** assignments in always blocks that are written to generate **combinational logic**

```verilog
class nbex1 (q, a, b, clk, rst_n);
output q;
input clk, rst_n;
input a, b;
reg q, y;

always @(a or b)
y = a ^ b;

always @(posedge clk or negedge rst_n)
if (!rst_n) q <= 1'b0;
else q <= y;
endmodule
```

![Synthesized result!]

```
module nbex2 (q, a, b, clk, rst_n);
output q;
input clk, rst_n;
input a, b;
reg q;

always @(posedge clk or negedge rst_n)
if (!rst_n) q <= 1'b0;
else q <= a ^ b;
endmodule
```

Nonblocking Assignment Guideline #3

- **Guideline #3:** Use **nonblocking** assignments in always blocks that are written to generate **sequential and combinational logic** in the same always block

Advanced Verilog Techniques Workshop - 2001
Combinational Logic & Nonblocking Assignments?

- Using nonblocking assignments in purely combinational always blocks might be functionally wrong
  - Pre-synthesis simulation will be incorrect
  - Synthesizes equivalent combinational logic with warnings about the sensitivity list

```verilog
module ao4 (y, a, b, c, d);
    output y;
    input a, b, c, d;
    reg y, tmp1, tmp2;
    always @(a or b or c or d) begin
        tmp1 <= a & b;
        tmp2 <= c & d;
        y    <= tmp1 | tmp2;
    end
endmodule
```

Wrong!

Previous values tmp1 and tmp2! NOT the values calculated during the current pass through this always block

```verilog
module ao5 (y, a, b, c, d);
    output y;
    input  a, b, c, d;
    reg    y, tmp1, tmp2;
    always @(a or b or c or d or tmp1 or tmp2 ) begin
        tmp1 <= a & b;
        tmp2 <= c & d;
        y    <= tmp1 | tmp2;
    end
endmodule
```

Active Events

- (RHS=1) Evaluate RHS of NBA
- (RHS=0) Evaluate RHS of NBA
- (RHS=1) Evaluate RHS of NBA
- (RHS=0) Evaluate RHS of NBA
- tmp1<=1 NBA assignment
- tmp1<=1 NBA assignment
- tmp1<=1 NBA assignment

NB Events

- tmp1<=1 Update LHS of NBA
- tmp1<=1 Update LHS of NBA
- tmp1<=1 Update LHS of NBA
- tmp1<=1 Update LHS of NBA
- tmp1<=1 Update LHS of NBA
Combinational Logic Coding Efficiency

- The model with all blocking assignments simulates 7% - 40% faster than the model with all nonblocking assignments!

```
module ao5 (y, a, b, c, d);
    output y;
    input  a, b, c, d;
    reg    y, tmp1, tmp2;
    always @(a or b or c or d or tmp1  or tmp2 ) begin
        tmp1 <=  a & b;
        tmp2 <=  c & d;
        y    <=  tmp1 | tmp2;
    end
endmodule
```

```
module ao1 (y, a, b, c, d);
    output y;
    input  a, b, c, d;
    reg    y, tmp1, tmp2;
    always @(a or b or c or d) begin
        tmp1 =  a & b;
        tmp2 =  c & d;
        y    =  tmp1 | tmp2;
    end
endmodule
```

Nonblocking assignments to *q* (sequential)

```
module ba_nbal (q, a, b, clk, rst_n);
    output q;
    input  a, b, rst_n;
    input  clk;
    reg    q, tmp;
    always @(posedge clk or negedge rst_n)
        if (!rst_n) q  <=  1'b0;
        else begin
            tmp  =  a & b ;
            q  <=  tmp ;
        end
endmodule
```

Risky coding style!

... but it does work!

Combining Blocking & Nonblocking Assignments
Combining Blocking & Nonblocking Assignments

module ba_nba3 (q, a, b, clk, rst_n);
output q;
input a, b, rst_n;
input clk;
reg q;

always @(posedge clk or negedge rst_n)
if (!rst_n) q <= 1'b0;
else q <= a & b;
endmodule

module ba_nba4 (q, a, b, clk, rst_n);
output q;
input a, b, rst_n;
input clk;
reg q;
wire tmp;
assign tmp = a & b;
always @(posedge clk or negedge rst_n)
if (!rst_n) q <= 1'b0;
else q <= tmp;
endmodule

Recommended coding styles
Combine the combinational assignment with the sequential equation...
... or separate the combinational logic assignment into a separate continuous assignment and separate always block

Not recommended: coding style with mixed blocking and nonblocking assignments in the same always block

Guideline #4: Do not mix blocking and nonblocking assignments in the same always block

module ba_nba6 (q, a, b, clk, rst_n);
output q;
input a, b, rst_n;
input clk;
reg q, tmp;

always @(posedge clk or negedge rst_n)
if (!rst_n) q = 1'b0;
else begin
    tmp = a & b;
    q <= tmp;
end
endmodule

Verilog simulators permit this!
Blocking assignment to q
Nonblocking assignment to q

Error: A reg can only be assigned with all RTL assignments or all procedural assignments near symbol ";" on line 11 in file ba_nba6.v

Advanced Verilog Techniques Workshop - 2001
Nonblocking Assignment Guideline #5

• Guideline #5: Do not make assignments to the same variable from more than one always block

```verilog
module badcode1 (q, d1, d2, clk, rst_n);
  output q;
  input  d1, d2, clk, rst_n;
  reg    q;
  always @(posedge clk or negedge rst_n)
    if (!rst_n) q  <= 1'b0;
    else        q  <= d1;
  always @(posedge clk or negedge rst_n)
    if (!rst_n) q  <= 1'b0;
    else        q  <= d2;
endmodule
```

Warning: In design 'badcode1', there is 1 multiple-driver net with unknown wired-logic type.

Common Misconceptions About Nonblocking Assignments
$display & Nonblocking Assignments

- Myth: “Using the $display command with nonblocking assignments does not work”
- Truth: Nonblocking assignments are updated after all $display commands

```verilog
module display_cmds;
  reg a;
  initial $monitor("$monitor: a = %b\n", a);
  initial begin
    $strobe ("$strobe : a = %b\n", a);
    a = 0;
    a <= 1;
    $display ("$display: a = %b\n", a);
    #1 $finish;
  end
endmodule
```

#0 Assignments

- Myth: “#0 forces an assignment to the end of a time step”
- Truth: #0 forces an assignment to the "inactive events queue"
module nb_schedule1;
    reg a, b;

    initial begin
        a = 0;
        b = 1;
        a <= b;
        b <= a;

        $monitor( "%0dns: \$monitor: a=%b  b=%b", $stime, a, b);
        $display("%0dns: \$display: a=%b  b=%b", $stime, a, b);
        $strobe( "%0dns: \$strobe : a=%b  b=%b\n", $stime, a, b);

        #0 $display("%0dns: #0      : a=%b  b=%b", $stime, a, b);
        #1 $monitor("%0dns: \$monitor: a=%b  b=%b", $stime, a, b);
        $display("%0dns: \$display: a=%b  b=%b", $stime, a, b);
        $strobe("%0dns: \$strobe : a=%b  b=%b\n", $stime, a, b);

        #1 $finish;
    end
endmodule
Multiple Nonblocking Assignments

- Myth: “Making multiple nonblocking assignments to the same variable in the same always block is undefined”

- Truth: Making multiple nonblocking assignments to the same variable in the same always block is defined by the 1364 Verilog Standard
  - The last nonblocking assignment to the same variable wins!

Nonblocking Procedural Assignments

From the IEEE Std 1364-1995 [2], pg. 47, section 5.4.1 - Determinism

"Nonblocking assignments shall be performed in the order the statements were executed. Consider the following example:

```verbatim
initial begin
    a <= 0;
    a <= 1;
end
```

When this block is executed, there will be two events added to the nonblocking assign update queue. The previous rule requires that they be entered on the queue in source order; this rule requires that they be taken from the queue and performed in source order as well. Hence, at the end of time-step 1, the variable a will be assigned 0 and then 1."

Translation: “Last nonblocking assignment wins!”
Why are there problems at the beginning of the simulation?

- Problem: different simulators and simulation options cause the simulation to behave differently at the beginning of a simulation

- Reasons:
  - reset is asserted, but not recognized at time 0

  **Guideline:** set reset to 0 with a nonblocking assignment at time 0

  - the first clock is executing at time 0

  **Guideline:** set the clock to 0 for the first half-cycle

---

**Reset Race Condition**

```verilog
//define cycle 100
timescale 1ns / 1ns
module reset_time0;
reg d, clk, rst_n;
reg q;
initial begin
  clk = 0;
  forever #(`cycle/2) clk = ~clk;
end
initial begin
  $timeformat(-9,0," ns ",6);
  $monitor("%t: q=%b  d=%b  clk=%b  rst_n=%b", $time, q, d, clk, rst_n);
  rst_n = 0;
  $display("RESET at time 0");
end
always @(posedge clk or negedge rst_n)
  if (!rst_n) q <= 0;
  else q <= d;
initial @(negedge rst_n) $display("RESET at time 0");
endmodule
```

<table>
<thead>
<tr>
<th>Time</th>
<th>q</th>
<th>d</th>
<th>clk</th>
<th>rst_n</th>
</tr>
</thead>
<tbody>
<tr>
<td>0ns</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>50ns</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>100ns</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>150ns</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>200ns</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>250ns</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Missed negedge rst_n!

Procedural blocks can start up in any order

waiting for negedge rst_n

Might miss negedge rst_n

Advanced Verilog Techniques Workshop - 2001
No Reset Race Condition

Procedural blocks can start up in any order

Must assert rst_n at the end of time 0
No Race Condition!

RESET at time 0
0ns: q=0 d=1 clk=0 rst_n=0
50ns: q=0 d=1 clk=1 rst_n=0
100ns: q=0 d=1 clk=0 rst_n=1
150ns: q=1 d=1 clk=1 rst_n=1
200ns: q=0 d=0 clk=0 rst_n=1
250ns: q=0 d=0 clk=1 rst_n=1

Triggered negedge rst_n

Assert rst_n with nonblocking assignment

`define cycle 100
`timescale 1ns / 1ns
module reset_time0;
reg d, clk, rst_n;
reg q;
initial begin
clk = 0;
forever #(`cycle/2) clk = ~clk;
end
initial begin
$timeformat(-9,0,"ns",6);
$monitor("%t: q=%b  d=%b  clk=%b  rst_n=%b",
$stime, q, d, clk, rst_n);
  rst_n <= 0;
d = 1;
 @(negedge clk) rst_n = 1;
 @(negedge clk) d = ~d;
 @(negedge clk) $finish;
end
always @(negedge clk or negedge rst_n)
if (!rst_n) q <= 0;
else q <= d;
initial @(negedge rst_n)
$display("RESET at time 0");
endmodule

Clock Race Condition

Procedural blocks can start up in any order

0ns: q=x d=1 clk=1
50ns: q=x d=0 clk=0
CLK HIGH at time 0
100ns: q=0 d=0 clk=1
150ns: q=0 d=1 clk=0
200ns: q=1 d=1 clk=1

Missed posedge clk!

Assert rst_n with blocking assignment

clk assigned with blocking assignment

`define cycle 100
`timescale 1ns / 1ns
module clk_time0;
reg d, clk;
reg q;
initial begin
clk = 1;
forever #(`cycle/2) clk = ~clk;
end
initial begin
$timeformat(-9,0,"ns",6);
$monitor("%t: q=%b  d=%b  clk=%b",
$stime, q, d, clk);
d = 1;
 @(negedge clk) d = ~d;
 @(negedge clk) d = ~d;
 @(negedge clk) $finish;
end
always @(posedge clk)
q <= d;
initial @(posedge clk)
$display("CLK HIGH at time 0");
endmodule

Advanced Verilog Techniques Workshop - 2001
No Clock Race Condition

```
`define cycle 100  
`timescale 1ns / 1ns
module clk_time0;  
  reg d, clk;  
  reg q;  
initial begin  
  clk <= 1;  
  forever #(`cycle/2) clk = ~clk;  
end
initial begin  
  $timeformat(-9,0,"ns",6);  
  $monitor("%t: q=%b  d=%b  clk=%b",  
           $time, q, d, clk);  
  d <= 1;  
  @(negedge clk) d = ~d;  
  @(negedge clk) $finish;  
end  
always @(posedge clk)  
  q <= d;  
initial @(posedge clk)  
  $display("CLK HIGH at time 0");  
endmodule
```

**Procedural blocks can start up in any order**

**Assign clk with nonblocking assignment**

**Waiting for a posedge clk**

**Triggered posedge clk**

CLK HIGH at time 0
- 0ns: q=1  d=1  clk=1
- 50ns: q=1  d=0  clk=0
- 100ns: q=0  d=0  clk=1
- 150ns: q=0  d=1  clk=0
- 200ns: q=1  d=1  clk=1

Why don't I see the last printed value in my Simulation?

- Problem: the simulation ends before the last value is printed
- Reason: The $finish command was scheduled in the active events queue
$\text{finish Race Condition}$

```
module finish_print;
    integer cnt;
    reg clk;

    initial begin
        clk = 0;
        forever #(cycle/2) clk = ~clk;
        initial begin
            cnt = 1;
            repeat (7) begin
                @(posedge clk) cnt = cnt + 1;
                $strobe("STROBE: \text{cnt}=\%0d", cnt);
            end
        end
        initial wait(cnt == 8)
        $display("DISPLAY: \text{cnt}=\%0d", cnt);
    end
$\text{monitor}$
$\text{strobe}$
$\text{finish command}$

First use a wait command to test cnt and then $\text{display}$

Displayed values do not include cnt=8

```

$\text{No finish Race Condition}$

```
module finish_print;
    integer cnt;
    reg clk;

    initial begin
        clk = 0;
        forever #(cycle/2) clk = ~clk;
        initial begin
            cnt = 1;
            repeat (7) begin
                @(posedge clk) cnt = cnt + 1;
                $strobe("STROBE: \text{cnt}=\%0d", cnt);
            end
        end
        initial wait(cnt == 8)
        $display("DISPLAY: \text{cnt}=\%0d", cnt);
    end
$\text{monitor}$
$\text{strobe}$
$\text{finish command}$

First use a wait command to test cnt and then $\text{display}$

Displayed values DO include cnt=8

```

Advanced Verilog Techniques Workshop - 2001
$display and #0 Assignment Guidelines

• **Guideline #6**: Use $strobe to display values of variables that have been assigned by nonblocking assignments
  – Do not use $display to display variables assigned by nonblocking assignments

• **Guideline #7**: Do not make #0 procedural assignments

---

Summary of 7 Guidelines to Avoid Verilog Race Conditions

- Guideline #1: Sequential logic and latches - use nonblocking assignments
- Guideline #2: Combinational logic in an always block - use blocking assignments
- Guideline #3: Mixed sequential and combinational logic in the same always block is sequential logic - use nonblocking assignments
- Guideline #4: In general, do not mix blocking and nonblocking assignments in the same always block
- Guideline #5: Do not make assignments to the same variable from more than one always block
- Guideline #6: Use $strobe to display values that have been assigned using nonblocking assignments
- Guideline #7: Do not make #0 procedural assignments

Follow these guidelines to remove 90-100% of all Verilog race conditions
A Couple of Important Verilog Tips

File Naming Fundamental

• Verilog is case sensitive
• Make Verilog file names match the Verilog module names

Example:
file name: asicTop.v
module name: asicTop

• Advantages
  – Easier to compile all necessary files using the -y library directory command line switch
  – Easier to build synthesis scripts with a single list of designs that also serve as a list of file names
### Case Statement Definitions

- A common set of terms for case statements

```verilog
module casemux (y, a, b, sela);
output y;
input a, b, sela;
reg y;
always @(a or b or sela) begin
  case (sela)
    1'b0: y = b;
    1'b1: y = a;
    default: $display("%m: ERROR sela=%b", sela);
  endcase
end
endmodule
```

### Case Statement Types

#### `case` Statement

```verilog
module cl (y, a);
output [1:0] y;
input a;
reg [1:0] y;
always @(a) begin
  case (a)
    1'b0: y = 0;
    1'b1: y = 1;
    1'bz: y = 2;
    1'bx: y = 3;
  endcase
end
endmodule
```

#### `casez` Statement

```verilog
module czl (y, a);
output [1:0] y;
input a;
reg [1:0] y;
always @(a) begin
  casez (a)
    1'b0: y = 0;
    1'b1: y = 1;
    1'bz: y = 2;
    1'bx: y = 3;
  endcase
end
endmodule
```

#### `casex` Statement

```verilog
module cx1 (y, a);
output [1:0] y;
input a;
reg [1:0] y;
always @(a) begin
  casex (a)
    1'b0: y = 0;
    1'b1: y = 1;
    1'bz: y = 2;
    1'bx: y = 3;
  endcase
end
endmodule
```

<table>
<thead>
<tr>
<th>If <code>a</code> = ...</th>
<th><code>case</code></th>
<th><code>casez</code></th>
<th><code>casex</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>1'b0:</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1'b1:</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1'bz:</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1'bx:</td>
<td>3</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

When should `casez` & `casex` be used for synthesis?
module code6 (memce0, memcel, cs, en, addr);
output  memce0, memcel, cs;
input   en;
input [31:30] addr;
reg     memce0, memcel, cs;
always @(addr or en) begin
  {memce0, memcel, cs} = 3'b0;
  casex ({addr, en})
    case (2’d10)
        2'b10: memce0 = 1'b1;
        2'b11: memcel = 1'b1;
        2'b001: cs = 1'b1;
    endcase
end
endmodule
module code7 (memce0, memcel, cs, en, addr);

output memce0, memcel, cs;
input en;
input [31:30] addr;
reg memce0, memcel, cs;

always @(addr or en) begin
  {memce0, memcel, cs} = 3'b0;
  casez ({addr, en})
    3'b101: memce0 = 1'b1;
    3'b111: memcel = 1'b1;
    3'b011: cs = 1'b1;
  endcase
end
endmodule
Casez & Casex Guidelines

- Guideline: Do not use casex for synthesizable code

- Guideline: Exercise caution when coding synthesizable models using the Verilog casez statement

- Coding Style Guideline: When coding a case statement with "don't cares":
  - use a casez statement
  - use "?" characters instead of "z" characters in the case items to indicate "don't care" bits

"full_case parallel_case", the Evil Twins of Verilog Synthesis

- Guideline: In general, do not add a "full_case" or "parallel_case" directives to Verilog models
  - Coding "One-Hot" FSMs using the unique "case (1'b1)" coding style is the only exception

- The "full_case" and "parallel_case" directives are ignored by simulation but change the behavior of synthesis tools
  - These switches can make some designs smaller and faster
  - These switches can make some designs larger and slower
  - These switches can change the functionality of the design
  - THESE SWITCHES ARE ALWAYS MOST DANGEROUS WHEN THEY WORK!
State Machines

Abbreviated FSMs (Finite State Machines)

State Machine Classification

- Moore - outputs are only a function of the present state

- Mealy - one or more of the outputs are a function of the present state and one or more of the inputs

- For an “n”-state FSM (examples on next slide):
  - Binary (highly encoded) = Ceiling log₂n flip-flops
    5 states -> 3 FF’s (fewest number of FF’s)

  - One-Hot = n flip-flops
    5 states -> 5 FF’s (smaller combinational logic)
Binary & One-Hot FSMs

- Binary (Highly Encoded) FSM
  - IDLE
  - READ
  - B1
  - B2
  - B3

- One-Hot FSM
  - IDLE
  - READ
  - B1
  - B2
  - B3

Primary State Machine Blocks

- Next State (N.S.) combinational logic
- Clocked Preset State (P.S.) logic
- Output combinational logic

Advanced Verilog Techniques Workshop - 2001
State Machine Coding Styles

- Two-Always Block State Machine
  - Use three-always blocks to register outputs

- Efficient One-Hot State Machine - combinational outputs
  - Using a coding style that is unique to Verilog

General State Machine Guidelines

- Guideline: Make each state machine a separate Verilog module
  - Easier to accurately maintain the FSM code
  - Enables FSM tools to do optimization/manipulation

- Guideline: Make state assignments using parameters with symbolic state names
  - Generally better than `define state definitions
  (examples on next two slides)
State Definitions Using `define

```
`define IDLE 3'b000
`define B1   3'b001
`define B2   3'b010
`define B3   3'b101
`define READ 3'b100
```

module fsm1 ( ... );
...
endmodule

```
`define IDLE 2'b00
`define READ 2'b01
`define S2   2'b10
```

module fsm2 ( ... );
...
endmodule

"Re-definition" warnings!

Multiple FSMs with the same state names

Bad coding style!!

define creates a global definition

State Definitions Using parameter

```
module fsm1 ( ... );
...
parameter
  IDLE = 3'b000,
  B1   = 3'b001,
  B2   = 3'b010,
  B3   = 3'b101,
  READ = 3'b100;
...
endmodule
```

```
module fsm2 ( ... );
...
parameter
  IDLE = 2'b00,
  READ = 2'b01,
  S2   = 2'b10;
...
endmodule
```

No state definition problems!

Multiple FSMs with the same state names

Good coding style!!

parameters create local definitions
State Parameter Definitions

Parameter definitions for binary encoding

\[
\text{parameter IDLE} = 3\text{'b}000, \\
S1 = 3\text{'b}001, \\
S2 = 3\text{'b}010, \\
S3 = 3\text{'b}011, \\
\text{ERROR} = 3\text{'b}100;
\]

Parameter definitions for verbose one-hot encoding

\[
\text{parameter IDLE} = 5\text{'b}00001, \\
S1 = 5\text{'b}00010, \\
S2 = 5\text{'b}00100, \\
S3 = 5\text{'b}01000, \\
\text{ERROR} = 5\text{'b}10000;
\]

State register encoding

Inefficient way to code One-Hot FSMs

Two-Always Block State Machines

1. Code a sequential always block to represent the state-vector register

2. Code a combinational always block to represent the next-state combinational logic

3. Combinational outputs can be coded using either:
   - Continuous-assignment outputs
   - OR -
   - Include the output assignments in the combinational next-state always block
Moore Example #1A- Bubble Diagram

- State machine bubble diagram

```
module sm2a (rd, ds, go, ws, clk, rstN);
output rd, ds;
input  go, ws, clk, rstN;
parameter IDLE = 2'b00,
    READ = 2'b01,
    DLY  = 2'b10,
    DONE = 2'b11;
reg [1:0] state, next;

always @(posedge clk or negedge rstN)
    if (!rstN) state <= IDLE;
    else state <= next;
```

Two-Always Block Coding Style
(Symbolic Parameter Assignments - Sequential Always Block)
Two-Always Block Coding Style
(Combinational Always Block - Continuous Assignment Outputs)

always @(state or go or ws) begin
  next = 2'bx;
  case (state)
    IDLE : if (go) next = READ;
           else    next = IDLE;
    READ : begin rd = 1'b1;
            next = DLY;
           end
    DLY  : begin rd = 1'b1;
           if (!ws) next = DONE;
           else next = READ;
    DONE : begin ds = 1'b1;
           next = IDLE;
           end
  endcase
end

assign rd = ((state==READ) || (state==DLY));
assign ds = (state==DONE);
endmodule

Two-Always Block Coding Style
(Combinational Always Block - Always Block Outputs)

always @(state or go or ws) begin
  next = 2'bx; rd = 1'b0; ds = 1'b0;
  case (state)
    IDLE : if (go) next = READ;
           else    next = IDLE;
    READ : begin rd = 1'b1;
            next = DLY;
           end
    DLY  : begin rd = 1'b1;
           if (!ws) next = DONE;
           else next = READ;
    DONE : begin ds = 1'b1;
           next = IDLE;
           end
  endcase
end
endmodule
Next State ‘x’-Default Assignment

- Make an initial default state assignment
  \[ \text{next} = 2'\text{bx}; \]

- Simulation/synthesis trick!
  - x-assignments are treated as “don’t-cares” by synthesis
  - x-assignments are treated as unknowns in simulation
    A missing next-state assignment will become obvious during simulation
    (FSM debugging trick!)

Next State Fixed-Default Assignment

- Some models require unused/undefined states to transition to known states
  - Satellite applications (designs subject to radiation)
  - Medical applications (for legal reasons!)
  - Formal verification (no equivalency checking with next= 2’bx)
  - Diagnostic hardware that uses FSM FF’s in a scan chain

  \[ \text{next} = \text{IDLE}; \quad \text{OR} \quad \text{next} = \text{ERROR}; \quad \text{etc...} \]

- Make a determinant state assignment
Two-Always Block Coding Style Guidelines Review

• Symbolic state names - easy to identify and modify
  – parameter READ = 2'b01;
  – case (state) READ
  – next = READ;
  – assign <Moore output> = (state == READ) …

• Separate present state sequential block
  – Reset condition checked first
  – Nonblocking assignments

Two-Always Block Coding Style Guidelines Review (cont.)

• Separate next state combinational block
  – Sensitivity list sensitive to state and all inputs
  – Default state assignment (next = 2’bx;)
  – Blocking assignments

• Making output assignments
  – Separate output continuous assignments
    All conditions that affect an output are in one location and easily scanned
    -OR-
  – Place output assignments in the combinational always block
    Input conditions have already been decoded
Registering The "Next Outputs"

Except where noted, outputs "rd" and "ds" equal 0.

Registered Outputs
(Add a third (sequential) always block)

```verilog
always @(posedge clk or negedge rst_n)
    if (!rst_n) begin
        ds <= 1'b0;
        rd <= 1'b0;
    end
    else begin
        ds <= 1'b0;
        rd <= 1'b0;
        case (next) 
            READ: rd <= 1'b1;
            DLY: rd <= 1'b1;
            DONE: ds <= 1'b1;
        endcase
    end
endmodule
```

Easy to register outputs for Moore State Machines

The next registered output depends on the next state
Registered "Next Outputs" Example

State register, sequential always block

Next state, combinational always block

"Next output", sequential always block

State Machines - Mealy Outputs

- Mealy Machine - One or more outputs are a function of the Present State and one or more of the inputs

- How are Mealy outputs coded?
  - Continuous assignments
    \[
    \text{assign } \text{<out> } = (\text{state }== \text{READ}) \& \!\text{read_strobe_n};
    \]
  - OR-
    \[
    \text{READ : if } (!\text{read_strobe_n}) \text{<out> } = 1;
    \]
**Onehot Coding Style**

(Two Always Blocks - Symbolic Parameter Assignments - Sequential Always Block)

```verilog
module sm1htfpa (rd, ds, go, ws, clk, rstN);
output rd, ds;
input  go, ws, clk, rstN;
parameter IDLE = 0,
             READ = 1,
            DLY  = 2,
           DONE = 3;
reg [3:0] state, next;
always @(posedge clk or negedge rstN)
if (!rstN) begin
    state       <= 4'b0;
    state[IDLE] <= 1'b1;
end
else         state       <= next;
...

Same example - One-Hot coding style

These are index values into the state register

```verilog
module sm1htfpa (rd, ds, go, ws, clk, rstN);
output rd, ds;
input  go, ws, clk, rstN;
parameter IDLE = 0,
             READ = 1,
            DLY  = 2,
           DONE = 3;
reg [3:0] state, next;
always @(posedge clk or negedge rstN)
if (!rstN) begin
    state       <= 4'b0;
    state[IDLE] <= 1'b1;
end
else         state       <= next;
...

always @(state or go or ws) begin
    next = 4'b0;
    case (1'b1) // synopsys full_case parallel_case
        state[IDLE] : if (go) next[READ] = 1'b1;
                       else next[IDLE] = 1'b1;
        state[READ]  : next[DLY] = 1'b1;
        state[DLY]   : if (!ws) next[DONE] = 1'b1;
                       else next[READ] = 1'b1;
        state[DONE]  : next[IDLE] = 1'b1;
    endcase
end
assign rd = (state[READ] || state[DLY]);
assign ds = (state[DONE]);
endmodule
```

**Onehot Coding Style**

(Two Always Blocks - Continuous Assignment Outputs)

In general, full_case & parallel_case will make a size and speed difference

```verilog
... always @(state or go or ws) begin
    next = 4'b0;
    case (1'b1) // synopsys full_case parallel_case
        state[IDLE] : if (go) next[READ] = 1'b1;
                       else next[IDLE] = 1'b1;
        state[READ]  : next[DLY] = 1'b1;
        state[DLY]   : if (!ws) next[DONE] = 1'b1;
                       else next[READ] = 1'b1;
        state[DONE]  : next[IDLE] = 1'b1;
    endcase
end
assign rd = (state[READ] || state[DLY]);
assign ds = (state[DONE]);
endmodule
```
FSM Summary

• Many ways to infer state machines (many obscure ways!)

• One-Hot state machine using `case (1'b1)`
  – Style is unique to Verilog
  – Using `parallel_case` directive makes a positive difference

• Registered outputs are glitch-free and a recommended synthesis coding style

Verilog-2001 Behavioral and Synthesis Enhancements
Verilog Enhancement Strategy

- Why make Verilog enhancements?
  - increase design productivity
  - enhance synthesis capability
  - improve verification efficiency

- The Verilog "prime directive"
  - do not break existing designs
  - do not impact simulator performance
  - make the language more powerful and easier to use

To boldly go where no simulator has gone before!

Top-Five Verilog Enhancement Requests

- IVC-1996 birds of a feather
  - Session chaired by Kurt Baty of WSFDB

- #1 - Verilog generate statement
- #2 - Multi-dimensional arrays
- #3 - Better Verilog file I/O
- #4 - Re-entrant tasks
- #5 - Better configuration control

Look at this enhancement first

NOTE: Verilog-2001 in this section is not fully tested! (waiting for a V2001 simulator!)
module dualpipe (dout1, dout0, din, en, clk, rst_n);
output [1:0] dout1, dout0;
input [1:0] din, en;
input clk, rst_n;
wire [1:0] data0_0, data0_1, data0_2, data0_3, data1_0, data1_1, data1_2, data1_3;

assign data1_0 = din;
assign data0_0 = din;
dff u000 (.q(data0_1[0]), .d(data0_0[0]), .clk(clk), .en(en[0]), .rst_n(rst_n));
dff u010 (.q(data0_1[0]), .d(data0_2[0]), .clk(clk), .en(en[0]), .rst_n(rst_n));
dff u020 (.q(data0_1[0]), .d(data0_3[0]), .clk(clk), .en(en[0]), .rst_n(rst_n));
dff u001 (.q(data0_1[1]), .d(data0_0[1]), .clk(clk), .en(en[0]), .rst_n(rst_n));
dff u011 (.q(data0_1[1]), .d(data0_2[1]), .clk(clk), .en(en[0]), .rst_n(rst_n));
dff u021 (.q(data0_1[1]), .d(data0_3[1]), .clk(clk), .en(en[0]), .rst_n(rst_n));
dff u100 (.q(data1_1[0]), .d(data1_0[0]), .clk(clk), .en(en[1]), .rst_n(rst_n));
dff u110 (.q(data1_1[0]), .d(data1_2[0]), .clk(clk), .en(en[1]), .rst_n(rst_n));
dff u120 (.q(data1_1[0]), .d(data1_3[0]), .clk(clk), .en(en[1]), .rst_n(rst_n));
dff u101 (.q(data1_1[1]), .d(data1_0[1]), .clk(clk), .en(en[1]), .rst_n(rst_n));
dff u111 (.q(data1_1[1]), .d(data1_2[1]), .clk(clk), .en(en[1]), .rst_n(rst_n));
dff u121 (.q(data1_1[1]), .d(data1_3[1]), .clk(clk), .en(en[1]), .rst_n(rst_n));
assign dout1 = data1_3;
assign dout0 = data0_3;
endmodule
module dualpipe_v2k (dout1, dout0, din, en, clk, rst_n);

output [1:0] dout1, dout0;
input [1:0] din, en;
input clk, rst_n;

wire [1:0] data [1:0] [3:0];

assign data[1][0][0] = din;
assign data[0][0][0] = din;

dff u000 (.q(data[0][1][0]), .d(data[0][0][0]), .clk(clk), .en(en[0]), .rst_n(rst_n));
dff u010 (.q(data[0][2][0]), .d(data[0][1][0]), .clk(clk), .en(en[0]), .rst_n(rst_n));
dff u020 (.q(data[0][3][0]), .d(data[0][2][0]), .clk(clk), .en(en[0]), .rst_n(rst_n));
dff u001 (.q(data[0][1][1]), .d(data[0][0][1]), .clk(clk), .en(en[0]), .rst_n(rst_n));
dff u011 (.q(data[0][2][1]), .d(data[0][1][1]), .clk(clk), .en(en[0]), .rst_n(rst_n));
dff u021 (.q(data[0][3][1]), .d(data[0][2][1]), .clk(clk), .en(en[0]), .rst_n(rst_n));
dff u100 (.q(data[1][1][0]), .d(data[1][0][0]), .clk(clk), .en(en[1]), .rst_n(rst_n));
dff u110 (.q(data[1][2][0]), .d(data[1][1][0]), .clk(clk), .en(en[1]), .rst_n(rst_n));
dff u120 (.q(data[1][3][0]), .d(data[1][2][0]), .clk(clk), .en(en[1]), .rst_n(rst_n));
dff u101 (.q(data[1][1][1]), .d(data[1][0][1]), .clk(clk), .en(en[1]), .rst_n(rst_n));
dff u111 (.q(data[1][2][1]), .d(data[1][1][1]), .clk(clk), .en(en[1]), .rst_n(rst_n));
dff u121 (.q(data[1][3][1]), .d(data[1][2][1]), .clk(clk), .en(en[1]), .rst_n(rst_n));

assign dout1 = data[1][3];
assign dout0 = data[0][3];
endmodule
Generate Statement
Verilog-2001 Enhancement #1

module dualpipe (dout1, dout0, din, en, clk, rst_n);
output [1:0] dout1, dout0;
input [1:0] din, en;
input        clk,  rst_n;
genvar i, j, k;
wire   [1:0] data [1:0] [3:0];
assign data[1][0] = din;
assign data[0][0] = din;
generate  for (i =0; i <=1; i=i+1 ) begin: ff1
  generate  for ( j =0; j <=2; j=j+1 ) begin: ff2
    generate  for ( k =0; k <=1; k=k+1 ) begin: ff3
      dff  u1 (.q(data[i][j+1][k]), .d(data[i][j][k]), .clk(clk), .en(en[i]), .rst_n(rst_n));
    endgenerate
  endgenerate
endgenerate
assign dout1 = data[1][3];
assign dout0 = data[0][3];
endmodule

Generate Statement

genvar Index Variable

- New keyword: genvar
  - Better than imposing restrictions on use of existing integer data type

- Restrictions
  - Genvars shall be declared within the module where they are used
  - Genvars can be declared either inside or outside of a generate scope
  - Genvars are positive integers that are local to and only used within a generate loop that uses them as index variables
  - Genvars are only defined during the evaluation of the generate blocks
  - Genvars do not exist during simulation of a Verilog design
  - No nesting of generate loops that use the same genvar index variable
  - The value of a genvar can be referenced in any context where the value of a parameter could be referenced
Other Types of Verilog
Generate Statements

• if-else generate

```verilog
module ...
    generate if ((A_WIDTH < 8) || (B_WIDTH < 8))
        CLA_multiplier u1 (...);
    else
        WALLACE_multiplier u1 (...);
endgenerate
```

• case-generate

```verilog
module ...
    generate case (1'b1)
        (WIDTH<6)                : adder1 u1 (...);
        ((WIDTH>5) && (WIDTH<10)): adder2 u1 (...);
        (WIDTH>10)               : adder3 u1 (...);
endgenerate
```

Enhanced File I/O
Verilog-2001 Enhancement #3

• Functions and tasks that open and close files

Verilog-1995
Multi-Channel Descriptor

```verilog
integer mcd;
initial begin
    mcd = $fopen ("file_name");
    ...
    $fclose(mcd);
end
```

Verilog-2001
File Descriptor

```verilog
integer fd;
initial begin
    fd = $fopen ("file_name", type);
    ...
    $fclose(fd);
end
```

<table>
<thead>
<tr>
<th>Type</th>
<th>Verilog-2001 File Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>open for reading</td>
</tr>
<tr>
<td>w</td>
<td>truncate to zero length or create for writing</td>
</tr>
<tr>
<td>a</td>
<td>append; open for writing at end of file, or create for writing</td>
</tr>
<tr>
<td>r+</td>
<td>open for update (reading and writing)</td>
</tr>
<tr>
<td>w+</td>
<td>truncate or create for update</td>
</tr>
<tr>
<td>a+</td>
<td>append; open for create for update at end-of-file</td>
</tr>
</tbody>
</table>
Enhanced File I/O (cont.)
Verilog-2001 Enhancement #3

- Tasks that output values into files
- Tasks that output values into variables
- Tasks/functions that read from files and load into variables or memories

New Verilog-2001 file I/O commands

$ferror
$fgetc
$fgets
$fread
$fscanf
$fseek
$ftell
$fungetc
$fungetf
$rewind
$sformat
$sscanf
$swrite (also $fwriteb, $fwriteo, $fwriteh)
$ungetc

Verilog-2001 files and Verilog-1995 files can both be used in the same testbench

Re-Entrant Tasks & Functions
Verilog-2001 Enhancement #4

module ...
    task write1;
    ...
endtask
    task automatic write2;
    ...
endtask
    ...
endmodule

Verilog-1995 tasks use static variables
Verilog-2001 new keyword: automatic
Verilog-2001 tasks use stacked variables
Automatic tasks & functions makes recursion possible

Advanced Verilog Techniques Workshop - 2001
Verilog Configuration Files
Verilog-2001 Enhancement #5

Simulation Configurations
Verilog-2001 Enhancement #5
ANSI-C Style Ports

module dffarn (q, d, clk, rst_n);
output q;
input d, clk, rst_n;
reg q;
always @(posedge clk or negedge rst_n)
if (!rst_n) q <= 1'b0;
else q <= d;
endmodule

q - in the port list
q - port type is "output"
q - data type is "reg"

Verbose port declarations

Comma-Separated Sensitivity List

module ram1 (addr, data, en, rw_n);
parameter ADDR_SIZE = 10;
parameter DATA_SIZE = 8;
parameter MEMDEPTH = 1<<ADDR_SIZE;
output [DATA_SIZE-1:0] data;
input [ADDR_SIZE-1:0] addr;
input en, rw_n;
reg [DATA_SIZE-1:0] mem [0:MEN_DEPTH-1];
assign data = (rw_n & en) ? mem[addr] : (DATA_SIZE{1'b0});
always @(addr, data, rw_n, en)
if (rw_n & en) mem[addr] = data;
endmodule

MEM_DEPTH is automatically sized from the ADDR_SIZE
Comma-separated sensitivity list

Parameterized Verilog-2001 memory (ram1) model

Works with ModelSim 5.5!
module code1a (o, a, b);
output o;
input a, b;
reg o;
always @(a or b)
o = a & b;
endmodule

module code1b (o, a, b);
output o;
input a, b;
reg o;
always @(a)
o = a & b;
endmodule

module code1c (o, a, b);
output o;
input a, b;
reg o;
always o = a & b;
endmodule

"Warning: Variable 'b' ... does not occur in the timing control of the block ..."

"Warning: Variable 'a' ... does not occur in the timing control of the block ..."

Incomplete Sensitivity List

Full sensitivity list
Pre-synthesis simulation matches post-synthesis simulation
Pre-synthesis simulation does not match post-synthesis simulation

Incomplete sensitivity list

No sensitivity list

Pre-synthesis simulation hangs!

Synthesized result!
@* Combinational Sensitivity List

module code1d (o, a, b);
output o;
input  a, b;
reg    o;
always @*
  o = a & b;
endmodule

Pre-synthesis simulation matches post-synthesis simulation

Synthesized result!

ANSI-C Style Parameters

parameter(s) can be redefined when the model is instantiated

module ram1 #(parameter ADDR_SIZE = 10,
  parameter DATA_SIZE = 8,
  localparam MEM_DEPTH = 1<ADDR_SIZE);
(output [DATA_SIZE-1:0] data,
  input  [ADDR_SIZE-1:0] addr,
  input                  en,  rw_n);
reg    [DATA_SIZE-1:0] mem [0:MEM_DEPTH-1];
assign data = (rw_n & en) ? mem[addr] : {DATA_SIZE{1'bz}};
always @*
  if (rw_n & en) mem[addr] <= data;
endmodule

MEM_DEPTH is automatically sized from the ADDR_SIZE

localparam(s) cannot be redefined when the model is instantiated

Combinational
@* sensitivity list
Errata - @*

(module decoder (output reg [7:0] y, input [2:0] a, input en);
always @* begin
y = 8'hff;
y[a] = ! en;
end
endmodule

always @* begin
next = 4'b0;
case (1'b1) // synopsys parallel_case
state[IDLE] : if (go) next[READ] = 1'b1;
else next[IDLE] = 1'b1;
state[READ] : next[DLY] = 1'b1;
state[DLY] : if (! ws) next[DONE] = 1'b1;
else next[READ] = 1'b1;
state[DONE] : next[IDLE] = 1'b1;
endcase
end

always @* begin
next = 4'b0;
case (1'b1) // synopsys parallel_case
state[IDLE] : if (go) next[READ] = 1'b1;
else next[IDLE] = 1'b1;
state[READ] : next[DLY] = 1'b1;
state[DLY] : if (! ws) next[DONE] = 1'b1;
else next[READ] = 1'b1;
state[DONE] : next[IDLE] = 1'b1;
endcase
end

Index variables should be part of a combinational sensitivity list

Non-constant case-items should be part of a combinational sensitivity list

Passing Parameters to Instances

module regblk (q, d, ce, clk, rst_n);
parameter SIZE = 4;
output [SIZE-1:0] q;
input [SIZE-1:0] d;
input ce, clk, rst_n;
reg [SIZE-1:0] q;
always @(posedge clk or negedge rst_n)
if (!rst_n) q <= 0;
else if (ce) q <= d;
endmodule

Two instantiated regblk instances with different sizes

Parameterized regblk model

demuxreg model block diagram

Advanced Verilog Techniques Workshop - 2001
defparam Redefinition
Verilog-1995

module demuxreg (q, d, ce, clk, rst_n);
output [15:0] q;
input [7:0] d;
input ce, clk, rst_n;
wire [15:0] q;
wire [7:0] n1;
not u0 (ce_n, ce);
defparam u1.SIZE=8;
regblk u1 (.q(n1), .d(d), .ce(ce), .clk(clk), .rst_n(rst_n));
defparam u2.SIZE=16;
endmodule

A defparam can be before the instance
A defparam can be after the instance
A defparam can be in a whole separate file!!

- The Verilog Standards Group hopes that defparams eventually die!

12.2.1 defparam statement (Verilog-2001)
"When defparams are encountered in multiple source files, e.g., found by library searching, the defparam from which the parameter takes it’s value is undefined."

Positional Parameter Redefinition
Verilog-1995

module demuxreg (q, d, ce, clk, rst_n);
output [15:0] q;
input [7:0] d;
input ce, clk, rst_n;
wire [15:0] q;
wire [7:0] n1;
not u0 (ce_n, ce);
defparam u1 #(8) u1 (.q(n1), .d(d), .ce(ce), .clk(clk), .rst_n(rst_n));
defparam u2 #(16) u2 (.q(q), .d((d,n1)), .ce(ce_n), .clk(clk), .rst_n(rst_n));
endmodule

instance and parameter information all on the same line of code

Passing parameters by position

One serious disadvantage:
When passing parameters by position, all parameters up to and including the changed parameters must be listed, even if some parameter values do not change!!
Named Parameter Redefinition
Verilog-2001 Enhancement

module demuxreg (q, d, ce, clk, rst_n);
output [15:0] q;
input [7:0] d;
input ce, clk, rst_n;
wire [15:0] q;
wire [7:0] n1;

not u0 (ce_n, ce);
regblk #( .SIZE(8) ) u1 (.q(n1),.d (d), .ce(ce), .clk(clk),.rst_n(rst_n));
regblk #( .SIZE(16) ) u2 (.q (q),.d({d,n1}),.ce(ce_n),.clk(clk),.rst_n(rst_n));
endmodule

Instance and parameter information all on the same line of code
Parameter redefinition using named parameters
(same syntax as named port connections)
Only the parameters that change must be listed

Demuxreg Verilog-2001 Model

module demuxreg (q, d, ce, clk, rst_n);
output [15:0] q;
input [7:0] d;
input ce, clk, rst_n;
wire [15:0] q;
wire [7:0] n1;

not u0 (ce_n, ce);
regblk #( .SIZE(8) ) u1 (.q(n1),.d (d), .ce(ce), .clk(clk),.rst_n(rst_n));
regblk #( .SIZE(16) ) u2 (.q (q),.d({d,n1}),.ce(ce_n),.clk(clk),.rst_n(rst_n));
endmodule
Parameter Redefinition

- **Guideline:** Do not use defparam statements
  - Extra line of code
  - Can be anywhere and in any file
  - In-house tools are harder to implement

- **Guideline:** Use instantiated models with the new Verilog-2001 named parameter redefinition

Top-Level Module Instantiation

- **Top-level I/O pad instantiation**
- **Pre-Verilog-1995**

```verilog
module top_pads1 (pdata, paddr, pctl1, pctl2, pwr, pclk);
input [15:0] pdata;                     // pad data bus
input [31:0] paddr;                     // pad addr bus
input pctl1, pctl2, pwr, pclk;   // pad signals
wire [15:0] data;                      // data bus
wire [31:0] addr;                      // addr bus
main_blk ul (.data(data), .addr(addr),
    .sig1(ctl1), .sig2(ctl2), .sig3(we), .clk(clk));
IBUF c4 (.O( we), .pI( pwr));
IBUF c3 (.O(ctl2), .pI(pctl2));
IBUF c2 (.O(ctl1), .pI(pctl1));
IBUF c1 (.O( clk), .pI( pclk));

BIDIR b15 (.N2(data[15]), .pN1(pdata[15]), .WR(we));
BIDIR b14 (.N2(data[14]), .pN1(pdata[14]), .WR(we));
...
BIDIR b0 (.N2(data[ 0]), .pN1(pdata[ 0]), .WR(we));

IBUF i31 (.O(addr[31]), .pI(paddr[31]));
IBUF i30 (.O(addr[30]), .pI(paddr[30]));
IBUF i29 (.O(addr[29]), .pI(paddr[29]));
...
IBUF i2 (.O(addr[ 2]), .pI(paddr[ 2]));
IBUF i1 (.O(addr[ 1]), .pI(paddr[ 1]));
IBUF i0 (.O(addr[ 0]), .pI(paddr[ 0]));
endmodule
```
Top-Level Module Instantiation

Verilog Generate Method

- Top-level I/O pad instantiation using generate statements
- Verilog-2001

```
module top_pads2 (pdata, paddr, pctl1, pctl2, pwr, pclk);
  input [15:0] pdata;  // pad data bus
  input [31:0] paddr;  // pad addr bus
  input pctl1, pctl2, pwr, pclk;  // pad signals
  wire [15:0] data;  // data bus
  wire [31:0] addr;  // addr bus

  main_blk u1 (.data(data), .addr(addr), .ctl1(pctl1), .ctl2(pctl2), .wr(pwr), .clk(pclk));

  genvar i;
  IBUF c4 (.O(wr), .I(pwr));
  IBUF c3 (.O(ctl2), .I(pctl2));
  IBUF c2 (.O(ctl1), .I(pctl1));
  IBUF c1 (.O(clk), .I(pclk));

  generate for (i=0; i<16; i=i+1) begin: data
    BIDIR b1 (.N2(data[i]), .pN1(pdata[i]), .WR(wr));
  endgenerate

  generate for (i=0; i<32; i=i+1) begin: addr
    IBUF i1 (.O(addr[i]), .I(paddr[i]));
  endgenerate

endmodule
```

Top-Level Module Instantiation

Array of Instances

- Top-level I/O pad instantiation using arrays of instances
- Verilog-1995

```
module top_pads3 (pdata, paddr, pctl1, pctl2, pwr, pclk);
  input [15:0] pdata;  // pad data bus
  input [31:0] paddr;  // pad addr bus
  input pctl1, pctl2, pwr, pclk;  // pad signals
  wire [15:0] data;  // data bus
  wire [31:0] addr;  // addr bus

  main_blk u1 (.data(data), .addr(addr), .ctl1(pctl1), .ctl2(pctl2), .wr(pwr), .clk(pclk));

  IBUF c4 (.O(wr), .I(pwr));
  IBUF c3 (.O(ctl2), .I(pctl2));
  IBUF c2 (.O(ctl1), .I(pctl1));
  IBUF c1 (.O(clk), .I(pclk));

  BIDIR b15:0 (.N2(data), .pN1(pdata), .WR(wr));
  IBUF i31:0 (.O(addr), .I(paddr));

endmodule
```
V++ Enhancement Idea?
Verilog-2005 Enhancement? - What do you think?

- Automatic port matching where applicable
- Named port matching for exceptions

```
module top_pads3 (pdata, paddr, pclt1, pclt2, pwr, pclk);
  input [15:0] pdata; // pad data bus
  input [31:0] paddr; // pad addr bus
  input  pclt1, pclt2, pwr, pclk; // pad signals
  wire [15:0] data; // data bus
  wire [31:0] addr; // addr bus

  main_blk ul (@*, .sig1(pctl1), .sig2(pctl2), .sig3(pwr));
  IBUF c4 (.O(wr), .pI(pwr));
  IBUF c3 (.O(ctl2), .pI(pctl2));
  IBUF c2 (.O(ctl1), .pI(pctl1));
  IBUF c1 (.O(clk), .pI(pclk));

  BIDIR b[15:0] (.N2(data), .pN1(pdata), .WR(wr));
  IBUF i[31:0] (.O(addr), .pI(paddr));
endmodule
```

Attributes

- New tokens

```
(*  *)
VSG members call these "funny braces"

(* attribute_name = constant_expression *)
-or-
(* attribute_name *)

// rtl_synthesis full_case parallel_case
replace with
(* rtl_synthesis, full_case, parallel_case *)
```

Synthetic comments require that all comments are parsed

Put tool directives into attributes to eliminate the need to parse comments
Additional Verilog-2001 Enhancements

- Signed arithmetic
- Exponential operator

- Enhanced conditional compilation

- Removed: net declaration requirement for LHS of continuous assignment to an internal net
- Added: `default_nettype none

- Initialize variables in the declaration
- Indexed vector part selects

What Broke in Verilog-2001?
(1) Having Exactly 31 Open Files

- Two behavioral capabilities broke in Verilog-2001

- File I/O (one corner case)
  - Verilog-1995 permits 31 open files, each open file sets one integer bit
  - Verilog-2001 steals the integer-MSB for new file I/O file handles
    1,000's of open files are now possible
  - If a design uses exactly 31 open files, the new file I/O enhancement will cause a conflict
  - Work-around: replace one of the open files with a Verilog-2001 file
What Broke in Verilog-2001?
(2) Unsized `bz Assignments

• Unsized z-assignment
  – `bz assignments assign up to 32 Z’s and then pad the MSBs with all 0’s
  – Any design that counts on this behavior deserves to be broken!

```
module tribuf (y, a, en);
    parameter SIZE = 64;
    output [SIZE-1:0] y;
    input [SIZE-1:0] a;
    input en;
    assign y = en ? a : 'bz;
endmodule
```

Verilog-1995:  
y = 64'h0000_0000_zzzz_zzzz

Verilog-2001:  
y = 64'hzzzz_zzzz_zzzz_zzzz

Parameterized model:  
three-state driver  
(Works for both  
Verilog-1995 & Verilog-2001)

Errata
A Few Mistakes "Slipped" Into The LRM

• The Ceiling of the Log of Base 2 function has errors

```
function integer clogb2;
    input depth;
    integer i,result;
    begin
        for (i = 0; 2 ** i < depth; i = i + 1)
            result = i + 1;
        clogb2 = result;
    end
endfunction
```

Verilog-1995:
```
function integer clogb2;
    input [31:0] value;
    for (clogb2=0; value>0; clogb2=clogb2+1)
        value = value>>1;
endfunction
```

Verilog-2001:
```
function integer clogb2;
    input [31:0] value;
    for (clogb2=0; value>0; clogb2=clogb2+1)
        value = value>>1;
endfunction
```

... result is never initialized and ...
... clogb2 is assigned "X"
Final Example - Using Multiple Verilog-2001 Enhancements

- Build a parameterized Barrel Shifter model (Verilog-1995)
- Convert the model into a parameterized Verilog-2001 model
- Instantiate the model into a Barrel-Shift register

Barrel Shifter Example

module SHIFTER_BARREL (Rotate, A, Y);
  input [2:0] Rotate;
  input [5:0] A;
  output [5:0] Y;
  reg [5:0] Y;
  always @(Rotate or A)
    begin: COMB_BARREL_SHIFT
      case (Rotate)
        0: Y = A;
        1: Y = rol6(A, 1);
        2: Y = rol6(A, 2);
        3: Y = rol6(A, 3);
        4: Y = rol6(A, 4);
        5: Y = rol6(A, 5);
        default : Y = 6'bx;
      endcase
    end
endmodule

function [5:0] rol6;
  input [5:0] A;
  input [2:0] NoShifts;
  reg [6:0] _ExtendLeft;
  integer N;
  begin
    _ExtendLeft = {1'b0, A};
    for (N=1; N<= NoShifts; N=N+1)
      begin
        _ExtendLeft = _ExtendLeft << 1;
        _ExtendLeft[0] = _ExtendLeft[6];
      end;
    rol6 = _ExtendLeft[5:0];
  end
endfunction

No rol6 function necessary - use part-selects and concatenation

Advanced Verilog Techniques Workshop - 2001
Barrel Shifter
Better RTL Model

module SHIFTER_BARREL (Rotate, A, Y);
input [2:0] Rotate;
input [5:0] A;
output [5:0] Y;
reg [5:0] Y;
always @(Rotate or A)
begin: COMB_BARREL_SHIFT
  case (Rotate)
    0: Y = A;
    1: Y = {A[4:0], A[5]};
    3: Y = {A[2:0], A[5:3]};
    4: Y = {A[1:0], A[5:2]};
    5: Y = {A[0], A[5:1]};
    default: Y = 6'b0;
  endcase
end
endmodule

Barrel Shifter
Two-Parameter RTL Model

module barrel_shifter (y, a, rotate_cnt);
output [5:0] y;
input [2:0] rotate_cnt;
input [5:0] a;
wire [5:0] tmp;
assign {y, tmp} = {a, a} << rotate_cnt;
endmodule

module barrel_shifter (y, a, rotate_cnt);
parameter WIDTH = 6;
parameter CNT_SIZE = 3;
output [WIDTH-1:0] y;
input [CNT_SIZE-1:0] rotate_cnt;
input [WIDTH-1:0] a;
wire [WIDTH-1:0] tmp;
assign {y, tmp} = {a, a} << rotate_cnt;
endmodule

Let's turn this into a parameterized model

Two parameters,
WIDTH & CNT_SIZE

{y, tmp} = (a, a) << rotate_cnt
is equivalent to
{y, tmp} = (a[(WIDTH-1)-rotate_cnt:0], a[WIDTH-1:0], {rotate_cnt{1'b0}})

Advanced Verilog Techniques Workshop - 2001
module barrel_shifter #(
    parameter WIDTH = 6,
    localparam CNT_SIZE = $clogb2(WIDTH));
(output [WIDTH-1:0] y,
    input [CNT_SIZE-1:0] rotate_cnt,
    input [WIDTH-1:0] a);
wire [WIDTH-1:0] tmp;
assign {y, tmp} = {a, a} << rotate_cnt;
endmodule

module barrel_shifter (
    parameter = WIDTH = 6,
    parameter = CNT_SIZE = 3);
(output [WIDTH-1:0] y,
    input [CNT_SIZE-1:0] rotate_cnt,
    input [WIDTH-1:0] a);
wire [WIDTH-1:0] tmp;
assign {y, tmp} = {a, a} << rotate_cnt;
endmodule

module barrel_reg (
    output reg [7:0] q,
    input [7:0] din,
    input [2:0] rot,
    input clk, rst_n);
wire [7:0] bs;
always @(posedge clk, negedge rst_n)
if (!rst_n) q <= 8'b0;
else q <= bs;
barrel_shifter #( . WIDTH(8) ) u1 (. y(bs), . a(din), . rotate_cnt(rot));
endmodule

module barrel_reg (#
    parameter = WIDTH = 6,
    parameter = CNT_SIZE = 3);
(output [WIDTH-1:0] y,
    input [CNT_SIZE-1:0] rotate_cnt,
    input [WIDTH-1:0] a);
wire [WIDTH-1:0] tmp;
assign {y, tmp} = {a, a} << rotate_cnt;
endmodule

module barrel_shifter (#
    parameter WIDTH = 6,
    localparam CNT_SIZE = $clogb2(WIDTH));
(output [WIDTH-1:0] y,
    input [CNT_SIZE-1:0] rotate_cnt,
    input [WIDTH-1:0] a);
wire [WIDTH-1:0] tmp;
assign {y, tmp} = {a, a} << rotate_cnt;
function integer clogb2;
    input [31:0] value;
    for (clogb2=0; value>0; clogb2=clogb2+1)
        value = value>>1;
endfunction
endmodule

module barrel_shifter #(
    parameter WIDTH = 6,
    localparam CNT_SIZE = $clogb2(WIDTH));
(output [WIDTH-1:0] y,
    input [CNT_SIZE-1:0] rotate_cnt,
    input [WIDTH-1:0] a);
wire [WIDTH-1:0] tmp;
assign {y, tmp} = {a, a} << rotate_cnt;
function integer clogb2;
    input [31:0] value;
    for (clogb2=0; value>0; clogb2=clogb2+1)
        value = value>>1;
endfunction
endmodule
Registered Barrel Shifter
Verilog-2001 Version

- Build the whole model in one simple module

```verilog
module barrel_reg (
    output reg [7:0] q,
    input [7:0] din,
    input [2:0] rot,
    input clk, rst_n);
wire [7:0] bs, tmp;
always @(posedge clk, negedge rst_n)
    if (!rst_n) q <= 8'b0;
    else q <= bs;
assign {bs,tmp} = {din,din} << rot;
endmodule
```

Parameterizing the barrel_reg model is left as an exercise for seminar attendees!!

Verilog-2001 Advantages

- increased design productivity
- enhanced synthesis capability
- improved verification efficiency

Verilog-2001 will be more powerful and easier to use than Verilog-1995
ModelSim User Notes

Two Quirks & Three Tricks related to using the ModelSim Verilog simulator

The ModelSim PC License Server

Starting the PC license server is not well documented

Double-click the flexlm.cpl (control panel) file

<Start> the license server and select <OK>

Change to the Modeltech_5.5/win32 directory
Misleading Port Warning

Contrary to the warning, there is no problem here.

This should be a syntax error.

The problem is the extra comma before the closing ")”.

```
`timescale 1ns / 1ns
module tb;
    reg a, b;

    myand2 u1 (.y(y), .a(a), .b(b));

    initial begin
        $timeformat(-9,0,"ns",10);
        $monitor("%t: y=%b  a=%b  b=%b", $stime, y, a, b);
        a=0;  b=0;
        #10 a=1;  b=1;
        #1 $finish;
    end

endmodule

module myand2 (y, a, b);
    output y;
    input  a, b;

    and g1 (y, a, b);
endmodule
```

```
# vsim -do {run -all} -c tb
# Loading work.tb
# Loading work.myand2
# Region: /tb/u1
```

Trick: Running UNIX-Version of ModelSim without the GUI

• To run a simulation with the GUI
  – execute the command:
    
    vsim <top_level_design>

• To run a simulation without the GUI
  – create a vsimr alias
    
    alias vsmr 'vsim -c -do "run -all"'

    – execute the command:
      
      vsmr <top_level_design>
Trick: Running Faster ModelSim Simulations

- By default, ModelSim runs in debug-optimized mode
  - this is different from most simulators

- To run ModelSim in speed-optimized mode, add two vlog command line switches:

  ```vlog -fast -O5 <list of files or command-file>```

  - What's the difference?
    Module boundaries are flattened and loops are optimized so some levels of debugging hierarchy are eliminated
    Significantly faster simulation performance

How To Isolate Design Problems
(RTL & UDP dff Example)

D-flip-flop User Defined Primitive model (with a bug!)
Trick: Comparing Waveforms

- ModelSim 5.5 has a very useful waveform compare feature

Starting the vsim GUI

Compile

Open the wave window

RTL simulation

Run the simulation

Quit the simulation to create a valid vsim.wlf file

Comparing Waveforms

- The default file vsim.wlf will be re-written on the next simulation

Change this file name (example: rtl.wlf)

The "waves" window goes blank after executing the quit -sim command
Comparing Waveforms

- Run a second simulation to generate a second set of waveforms

Either quit the simulation (to create another valid vsim.wlf file) -OR- run comparison with the current data set (current simulation)

Comparing Waveforms

- Use the comparison wizard!!

Select one of the .wlf files (example: rtl.wlf)

Select a second .wlf file (example: udpbug.wlf) -OR- use the current simulation data set

At the end of each step, click on Next->
Comparing Waveforms

• "Waveform compare" highlights differences between signals

- Select Compute Differences Now ...

- Click on Next> to use the signals from the two selected data sets

- ... and click on Next>

- Select Finish to show the waveform display

- X marks the spot!

- Expand the different signals

- The differences are highlighted

- Find previous difference

- Find next difference

- Find first difference

- Find last difference

- The UDP output is going unknown!

Advanced Verilog Techniques Workshop - 2001
Waveform Comparison Helps Isolate Problems

```verilog
module dffudp (q, d, clk, rst_n);
  output q;
  input  d, clk, rst_n;

  `ifdef RTL
    reg  q;
    always @(posedge clk or negedge rst_n)
      if (!rst_n) q <= 1'b0;
      else        q <= d;
  `endif
  `ifdef UDP
    udpdff u1 (q, d, clk, rst_n);
  `endif
endmodule
```

```verilog
primitive udpdff (q, d, clk, rst_n);
  output q;
  input  d, clk, rst_n;
  reg    q;
  table
    //  d   clk  rst_n :  q  :  qnext
    ?    ?    0   :  ?  :  0 ;
    0    r    1   :  ?  :  0 ;
    1    r    1   :  ?  :  1 ;
    // ignore falling transitions
    // on clk
    ?    f    1   :  ?  :  - ;
    // ignore transitions on d
    // ignore transitions on rst_n
  endtable
endprimitive
```

A missing UDP entry for `rst_n` transitions caused the bug

Where to go to download the papers

- Some papers with more details can be found at the Sunburst Design web site:
  
  www.sunburst-design.com/papers

  (Bad web site that might improve some day!)

- Papers with more details about the Verilog Seminar topics:
  
  CummingsHDLCON2001_Verilog2001_rev1_1.pdf
  CummingsSNUG1999Boston_FullParallelCase_rev1_1.pdf
  CummingsSNUG1999SJ_SynthMismatch_rev1_1.pdf
  CummingsSNUG2000Boston_FSM_rev1_1.pdf
  CummingsSNUG2000SJ_NBA_rev1_2.pdf