Sponsored by

Model Technology
A MENTOR GRAPHICS COMPANY

Sunburst Design

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

Sunburst Design

Sponsored by Model Technology

- 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

Sponsored by Model Technology

3

Sunburst Design

The Two Details Most Misunderstood About Verilog

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

Introducing: the Verilog event queue

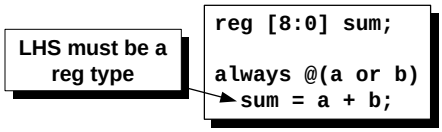
Use of Reg Types Vs. Net Types

4

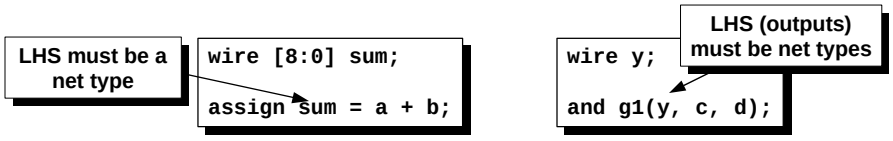
Sunburst Design

Sponsored by Model Technology

- Register types are only assigned inside of procedural blocks



- Net types are assigned or driven outside of procedural blocks



5

Blocking & Nonblocking Assignments

Sunburst Design

Sponsored by Model Technology

- 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

6

"Nonblocking" Is Not A Typo

Sunburst Design

Sponsored by Model Technology

- 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

7

Verilog Race Conditions

Sponsored by Model Technology

- 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

8

Blocking Assignments

Sponsored by Model Technology

- Blocking assignment operator: **=**
- Execution of blocking assignments is a one-step process:
  ① 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

Sunburst Design

Sponsored by Model Technology

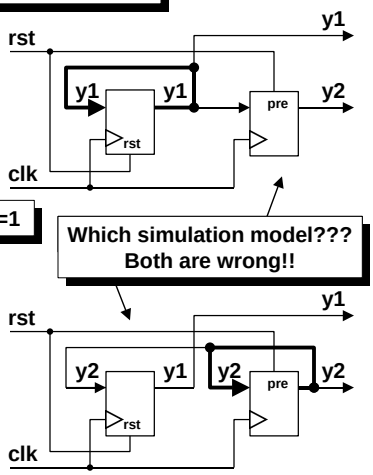
- Problem: Inter-dependent blocking assignments that execute in the same time step

Execution order is not guarantee!!

```
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
```

On rst, y1=0 and y2=1

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



Which simulation model??? Both are wrong!!

Nonblocking Assignments

Sunburst Design

Sponsored by Model Technology

- 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)

Syntax error! Illegal continuous assignment

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

Inter-Dependent Nonblocking Assignments

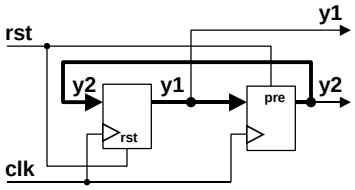
- No problem: Inter-dependent nonblocking assignments that execute in the same time step

```
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

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

Simulation & synthesis result



No race condition!

7 Coding Guidelines

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

- 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

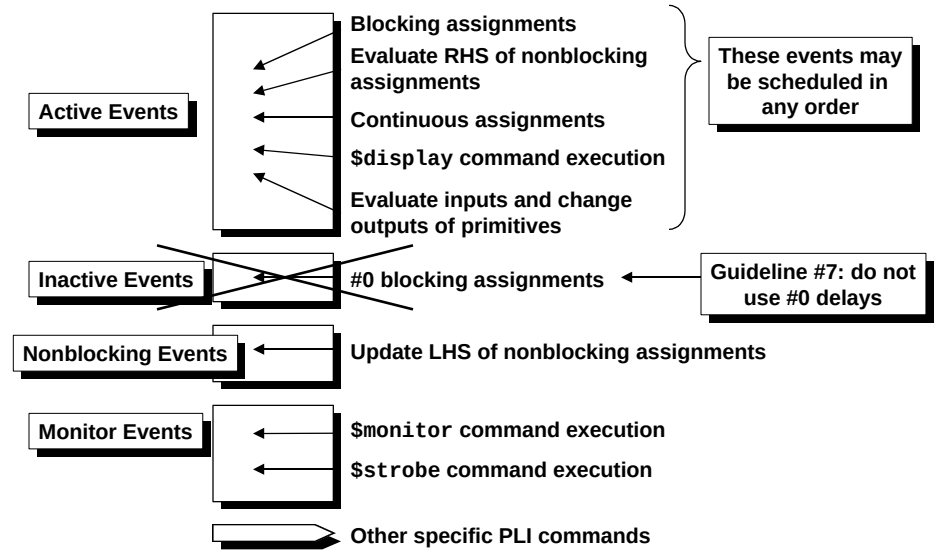
Guideline #7: Do not make #0 procedural assignments

13

IEEE1364-1995 Verilog Stratified Event Queue

Sunburst Design

Sponsored by Model Technology



14

Self-Triggering Oscillators?

Sunburst Design

NOTE: contrived examples - not the recommended coding style for clock oscillators!

```
module osc1 (clk);
  output clk;
  reg    clk;

  initial
    #10 clk = 0;

  always @(clk)
    #10 clk = ~clk;
endmodule
```

```
module osc2 (clk);
  output clk;
  reg    clk;

  initial
    #10 clk = 0;

  always @(clk)
    #10 clk <= ~clk;
endmodule
```

```
     osc1    osc2
    ------  ------
 0: clk1=x  clk2=x
10: clk1=0  clk2=0
20: clk1=1  clk2=1
30: clk1=1  clk2=0
40: clk1=1  clk2=1
50: clk1=1  clk2=0
60: clk1=1  clk2=1
70: clk1=1  clk2=0
80: clk1=1  clk2=1
90: clk1=1  clk2=0
```

```
module tb;
  osc1 x1 (.clk(clk1));
  osc2 x2 (.clk(clk2));

  initial begin
    $display("             osc1    osc2");
    $display("            ------  ------");
    $monitor("%d: clk1=%b  clk2=%b", $stime, clk1, clk2);
  end

  initial #100 $finish;
endmodule
```

osc1: blocking assignment

osc2: nonblocking assignment

osc1: no oscillation!

osc2: oscillating!

15

Self-Triggering?

(not with blocking assignments)

Sponsored by Model Technology

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
clk<=1	← NBA assignment >(triggers @(clk))
(RHS=0)	← Evaluate RHS of NBA
	← activate NB events
clk<=0	← NBA assignment >(triggers @(clk))
(continues)	→ oscillating!

NB Events

clk<=1	← Update LHS of NBA
clk<=0	← Update LHS of NBA

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!

16

Pipeline

Blocking Assignments Style #1

Sponsored by Model Technology

Active Events

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

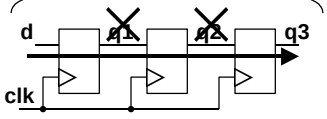
NB Events

Monitor Events

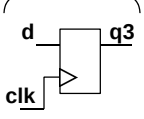
```
module pipeb1 (q3, d, clk);
  output [7:0] q3;
  input  [7:0] d;
  input        clk;
  reg    [7:0] q3, q2, q1;

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

Three sequential blocking assignments but only one synthesized register

Simulation result



Synthesis result!!

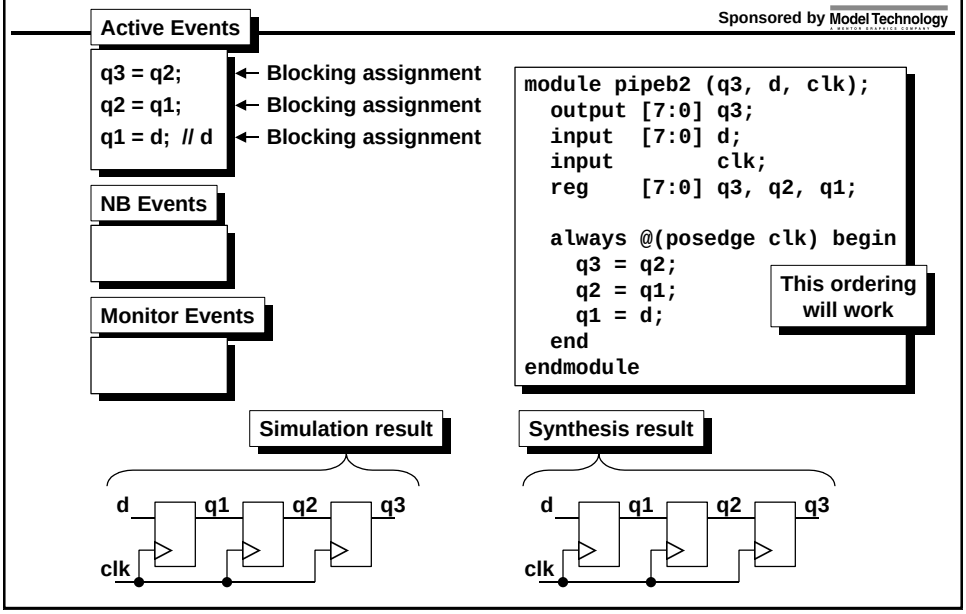


Pipeline

Blocking Assignments Style #2

Sunburst Design

Sponsored by Model Technology

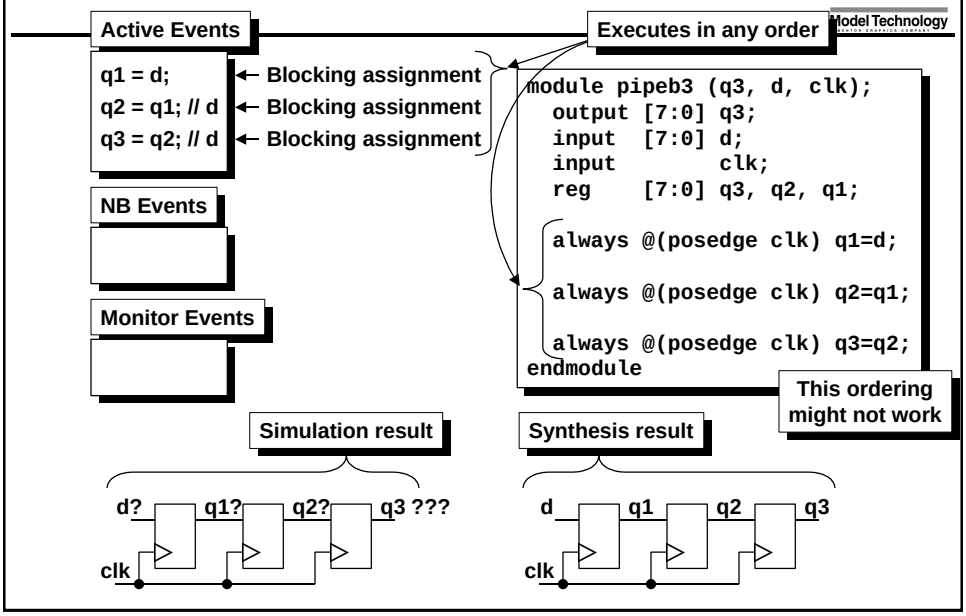


Pipeline

Blocking Assignments Style #3

Sunburst Design

Model Technology



Pipeline

Blocking Assignments Style #4

Sunburst Design

Model Technology

Active Events

```
q2 = q1;      <- Blocking assignment
q3 = q2; // q1 <- Blocking assignment
q1 = d;  // q1 <- Blocking assignment
```

Executes in any order

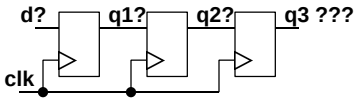
NB Events

Monitor Events

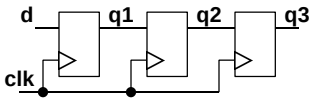
```
module pipeb4 (q3, d, clk);
  output [7:0] q3;
  input  [7:0] d;
  input        clk;
  reg    [7:0] q3, q2, q1;

  always @(posedge clk) q2=q1;

  always @(posedge clk) q3=q2;

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

This ordering might not work

Simulation result



Synthesis result



Blocking Assignment Coding Styles (Pipelines)

Sunburst Design

Sponsored by Model Technology

Pipeline Coding Styles

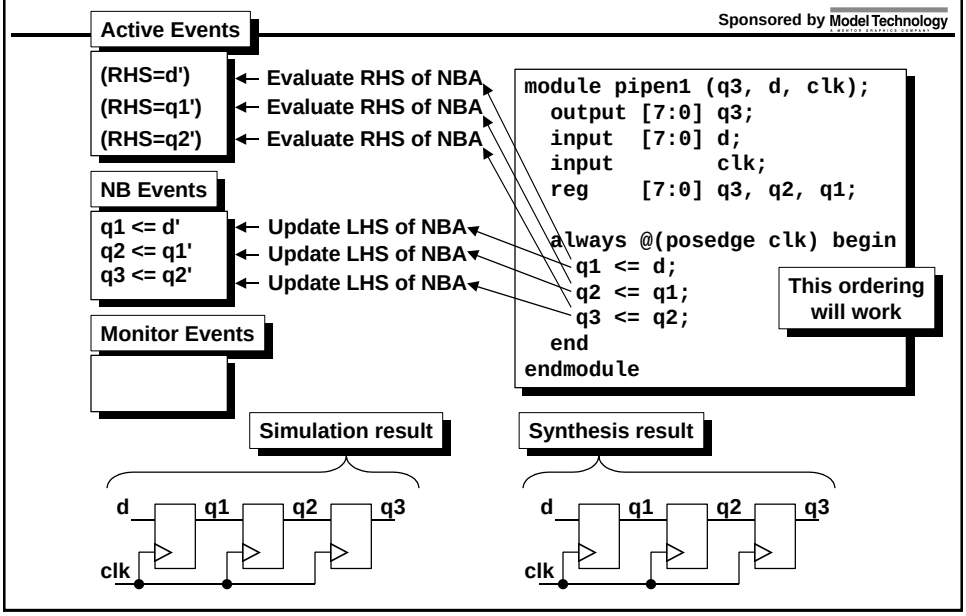
Coding Style	Simulates correctly	Synthesizes correctly
Blocking Assignments #1	NO!	NO!
Blocking Assignments #2	Yes	Yes
Blocking Assignments #3	?	Yes
Blocking Assignments #4 +	?	Yes

1 of 4 simulates correctly

3 of 4 synthesize correctly

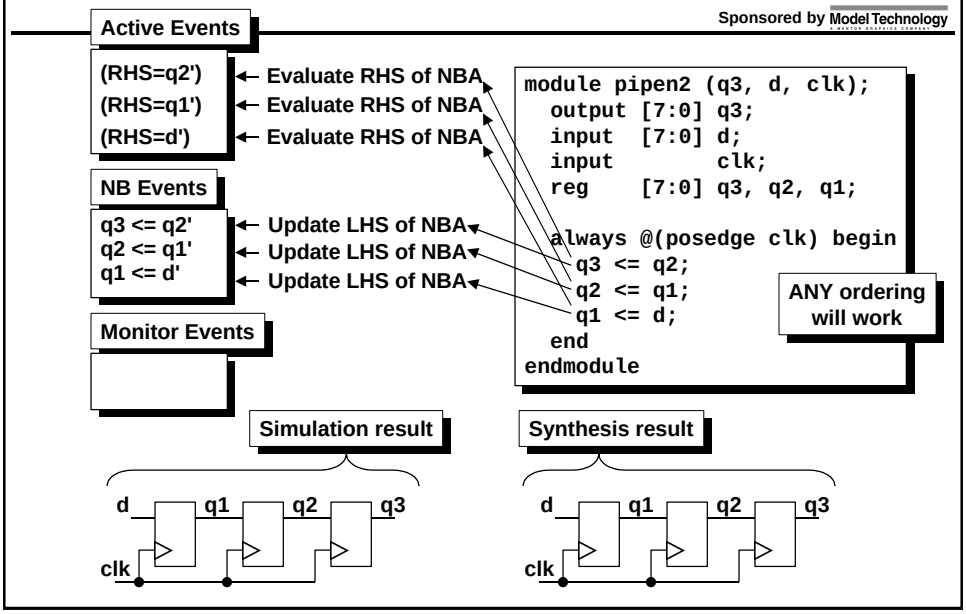
Pipeline

Nonblocking Assignments Style #1



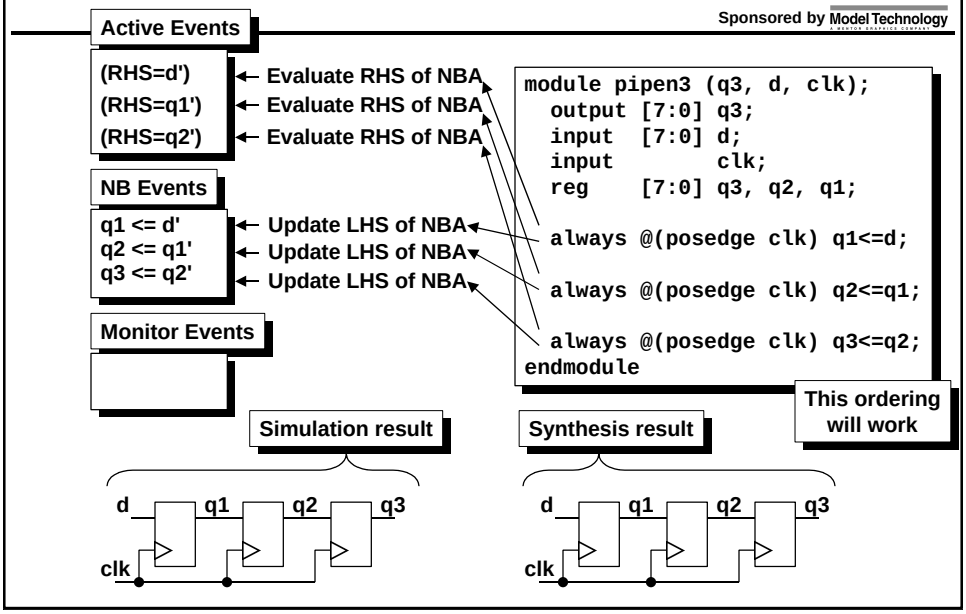
Pipeline

Nonblocking Assignments Style #2



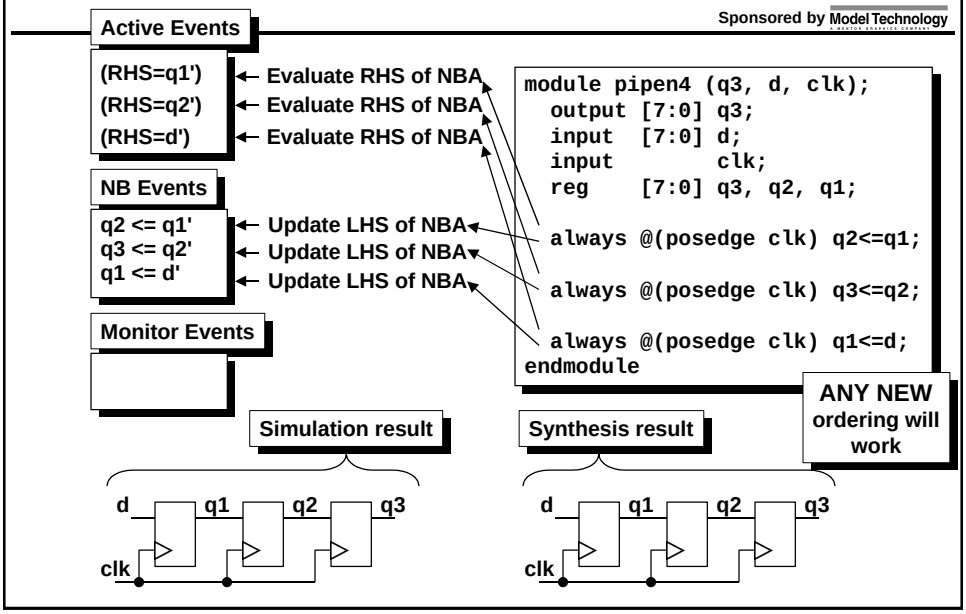
Pipeline

Nonblocking Assignments Style #3



Pipeline

Nonblocking Assignments Style #4



Nonblocking Assignment Coding Styles (Pipelines)

Pipeline Coding Styles

Coding Style	Simulates correctly	Synthesizes correctly
Blocking Assignments #1	NO!	NO!
Blocking Assignments #2	Yes	Yes
Blocking Assignments #3	?	Yes
Blocking Assignments #4 +	?	Yes
Nonblocking Assignments #1	Yes	Yes
Nonblocking Assignments #2	Yes	Yes
Nonblocking Assignments #3	Yes	Yes
Nonblocking Assignments #4 +	Yes	Yes

1 of 4 simulates 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

```
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

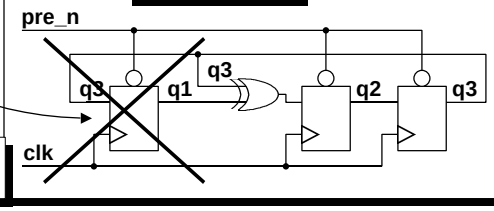

```
module lfsrb1 (q3, clk, pre_n);
  output q3;
  input  clk, pre_n;
  reg    q3, q2, q1;
  wire   n1;

  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	← q3=1'b1 (preset)
q2=1	← q2=1'b1
q1=1	← q1=1'b1
n1=(1^1)=0	← assign n1=q1^q3
q3=(q2)=1	← q3=q2 (clk)
q2=(n1)=0	← q2=n1
q1=(q3)=1	← q1=q3
n1=(1^1)=0	← assign n1=q1^q3
q3=(q2)=0	← q3=q2 (clk)
q2=(n1)=0	← q2=n1
q1=(q3)=0	← q1=q3

q1-q2-q3: 1 1 1 → 1 0 1 → 0 0 0

Wrong value! Should be "1"



Synthesizes, but does not simulate correctly

LFSR

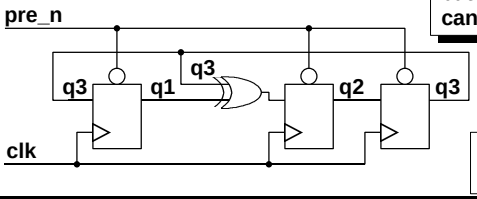
Blocking Assignments Style #2


```
module lfsrb2 (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	← {q3,q2,q1}=3'b111
q1,q2,q3=1(0)1	← {q3,q2,q1}={q2,(q1^q3),q3}
q1,q2,q3=1(0)0	← {q3,q2,q1}={q2,(q1^q3),q3}
q1,q2,q3=0(1)0	← {q3,q2,q1}={q2,(q1^q3),q3}
...	

This will work ... but the equations can be confusing



preset	q1-q2-q3: 1 1 1
clk	q1-q2-q3: 1 0 1
clk	q1-q2-q3: 1 0 0
clk	q1-q2-q3: 0 1 0

Synthesizes and simulates correctly

LFSR

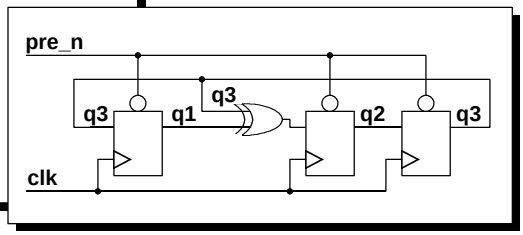
Nonblocking Assignments Style #1

Sunburst Design

Sponsored by Model Technology

```
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
```

Synthesizes and simulates correctly

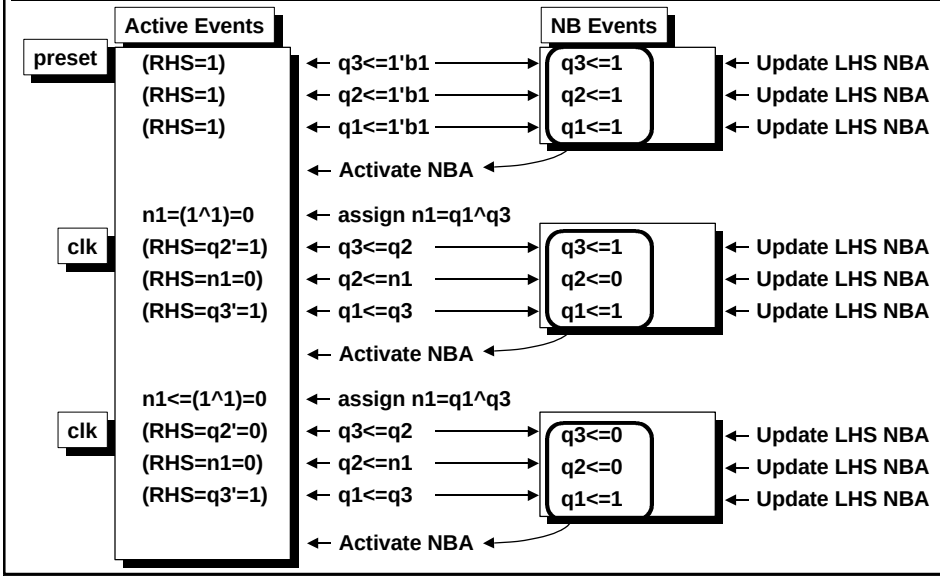


LFSR

Nonblocking Assignments Style #1

Sunburst Design

Sponsored by Model Technology



LFSR

Nonblocking Assignments Style #2

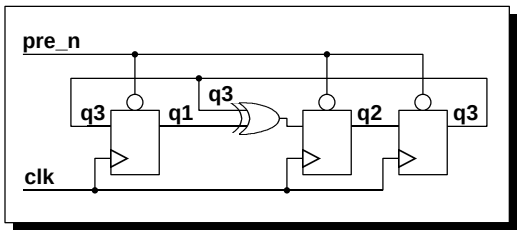
Sunburst Design

Sponsored by Model Technology

```
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)

Sunburst Design

Sponsored by Model Technology

LFSR Coding Styles

Coding Style	Simulates correctly	Synthesizes correctly
Blocking Assignments #1	NO!	Yes
Blocking Assignments #2	Yes	Yes
Nonblocking Assignments #1	Yes	Yes
Nonblocking Assignments #2	Yes	Yes

Cryptic coding style

Cryptic coding style

1 of 2 simulates correctly

2 of 2 synthesize correctly

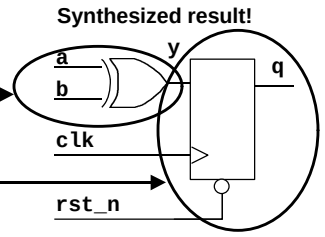
2 of 2 simulate correctly

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**

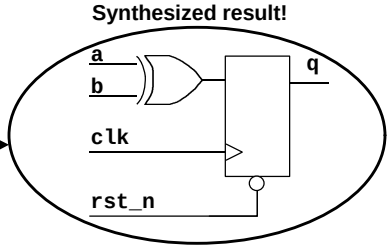
```
module nbex1 (q, a, b, clk, rst_n);
  output q;
  input  clk, rst_n;
  input  a, b;
  reg    q, y;

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

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



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

```
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
```



Combinational Logic & Nonblocking Assignments?

Sunburst Design

Sponsored by Model Technology

- 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

```
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!

a	b	c	e	tmp1	tmp2	y
0	1	0	0	0	0	0
1	1	0	0	1	0	0

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

Sunburst Design

a	b	c	d	tmp1	tmp2	y
0	1	0	0	0	0	0
1	1	0	0	1	0	0
1	1	0	0	1	0	1

```
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=0)	← Evaluate RHS of NBA
	← activate NB events
tmp1<=1	← NBA assignment
	→ Again - trigger always block
(RHS=1)	← Evaluate RHS of NBA
(RHS=0)	← Evaluate RHS of NBA
(RHS=1)	← Evaluate RHS of NBA
tmp2<=0	← NBA assignment
y<=0	← NBA assignment
	← activate NB events
tmp1<=1	← NBA assignment
tmp2<=0	← NBA assignment
y<=1	← NBA assignment

NB Events

tmp1<=1	← Update LHS of NBA
tmp2<=0	← Update LHS of NBA
y<=0	← Update LHS of NBA
tmp1<=1	← Update LHS of NBA
tmp2<=0	← Update LHS of NBA
y<=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 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
```

All blocking 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
```

All nonblocking assignments

Combining Blocking & Nonblocking Assignments


```
module ba_nba1 (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
```

Nonblocking assignments to q (sequential)

Blocking assignment to tmp (combinational)

Risky coding style! Potentially confusing event schedule ...

... but it does work!

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
```

Recommended coding styles

Combine the combinational assignment with the sequential equation . . .

```
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
```

. . . 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

Combining Blocking & Nonblocking Assignments

- **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!

Synthesis syntax error!

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
```

Nonblocking Assignment Guideline #5

Sunburst Design

Sponsored by Model Technology

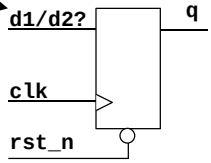
- **Guideline #5 :** Do not make assignments to the same variable from more than one always block

```
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
```

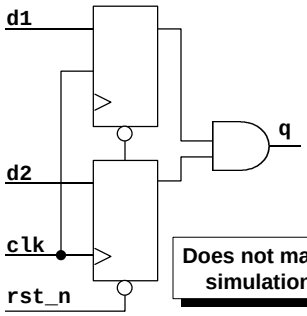
```
Warning: In design 'badcode1', there
         is 1 multiple-driver net with
         unknown wired-logic type.
```

Race condition!

Simulation result??



Synthesis result !!



Does not match simulation!

Sponsored by Model Technology

Sunburst Design

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

```
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
```

Actual output display

```
$display: a = 0

$monitor: a = 1

$strobe : a = 1
```


#0 Assignments

- Myth: “#0 forces an assignment to the end of a time step”
- Truth: #0 forces an assignment to the "inactive events queue"

Scheduling Example

Sunburst Design

```
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);
       $display ("%0dns: #0      : a=%b  b=%b", $stime, a, b);

    #1 $finish;
  end
endmodule
```

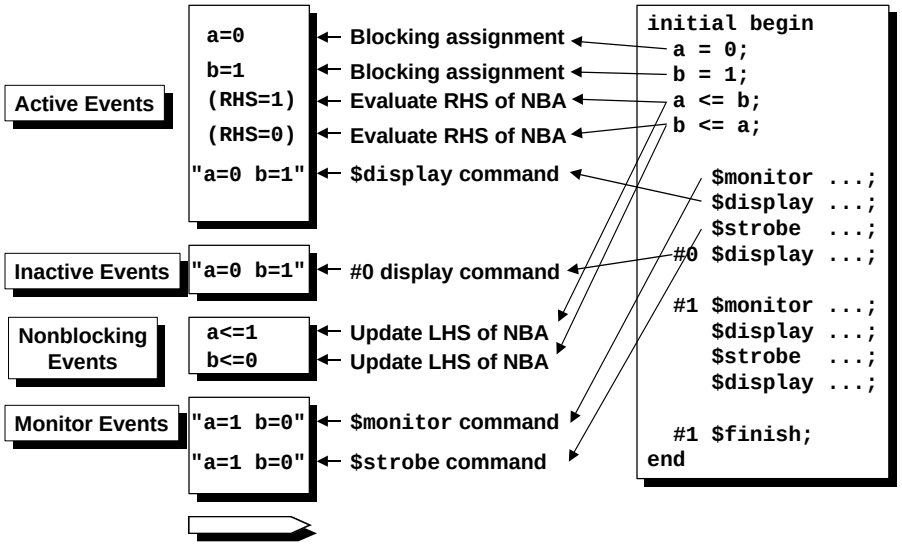
Displayed output!!

```
0ns: $display: a=0  b=1
0ns: #0      : a=0  b=1
0ns: $monitor: a=1  b=0
0ns: $strobe : a=1  b=0

1ns: $display: a=1  b=0
1ns: #0      : a=1  b=0
1ns: $monitor: a=1  b=0
1ns: $strobe : a=1  b=0
```


Scheduling Example
Queued Events

Sunburst Design

Sponsored by Model Technology



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:

```
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

```
`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 @(posedge clk or negedge rst_n)
    if (!rst_n) q <= 0;
    else        q <= d;

  initial @(negedge rst_n)
    $display("RESET at time 0");
endmodule
```

Procedural blocks can start up in any order

rst_n asserted with blocking assignment

Might miss negedge rst_n

Waiting for a negedge rst_n

```
   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=1  d=0  clk=0  rst_n=1
 250ns: q=0  d=0  clk=1  rst_n=1
```

Missed negedge rst_n!

No Reset Race Condition

```
`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 @(posedge clk or negedge rst_n)
    if (!rst_n) q <= 0;
    else        q <= d;

  initial @(negedge rst_n)
    $display("RESET at time 0");
endmodule
```

Assert rst_n with nonblocking assignment

Procedural blocks can start up in any order

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

Negedge rst_n will be triggered

Waiting for a negedge rst_n

```
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=1  d=0  clk=0  rst_n=1
 250ns: q=0  d=0  clk=1  rst_n=1
```

Triggered negedge rst_n

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",
             $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
```

clk assigned with blocking assignment

Procedural blocks can start up in any order

Waiting for a posedge clk

```
   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!

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",
             $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
```

Assign clk with nonblocking assignment

Procedural blocks can start up in any order

```
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
```

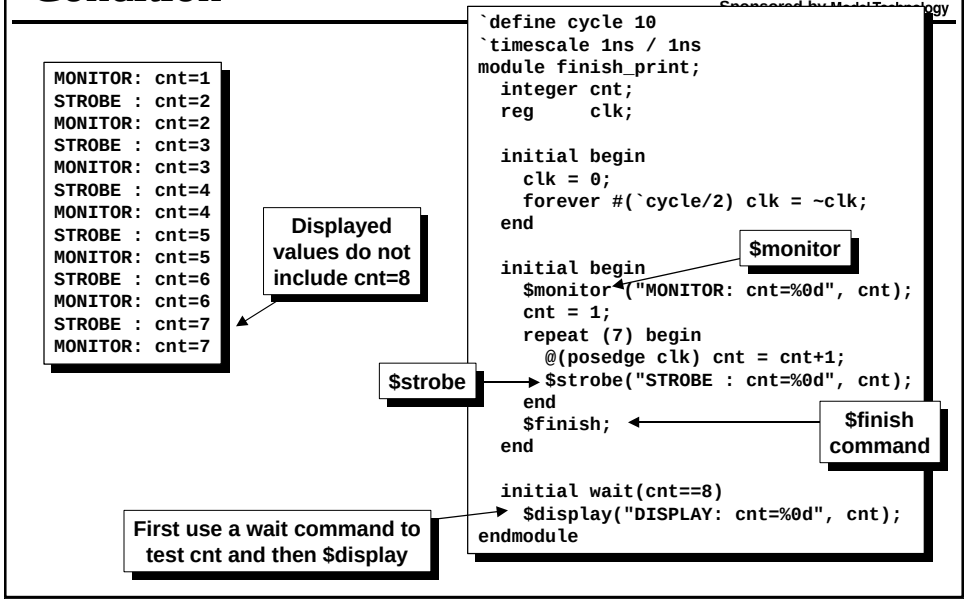
Waiting for a posedge clk

Triggered posedge clk

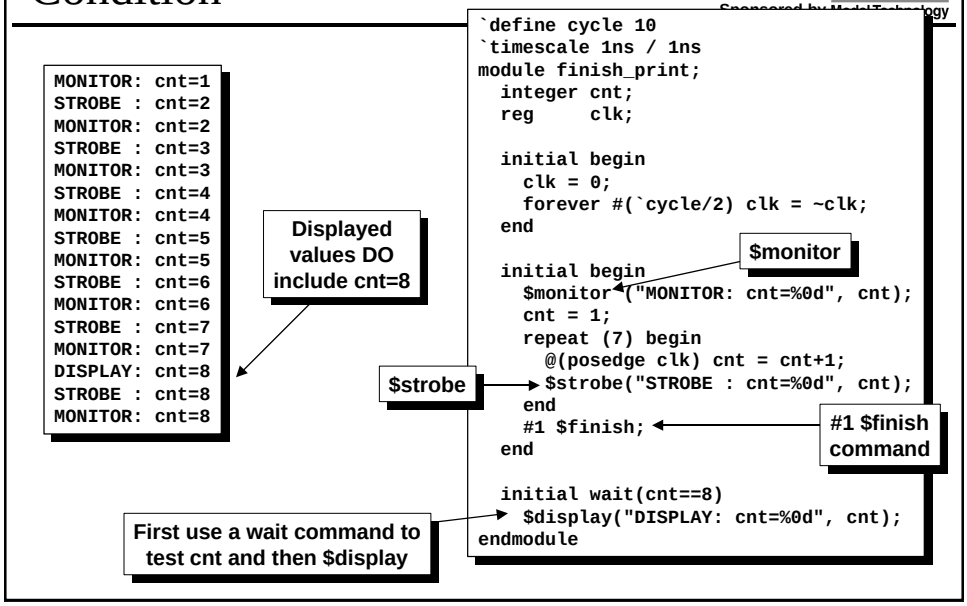
Why don't I see the last printed value in my Simulation?

- Problem: the simulation ends before the last value is printed
- Reason: The $finish command was scheduled in the active events queue

$finish Race Condition



No $finish Race Condition



$display and #0 Assignment Guidelines

Sponsored by Model Technology

- **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

Sponsored by Model Technology

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

Sponsored by Model Technology

Sunburst Design

A Couple of Important VerilogTips

File Naming Fundamental

Sponsored by Model Technology

Sunburst Design

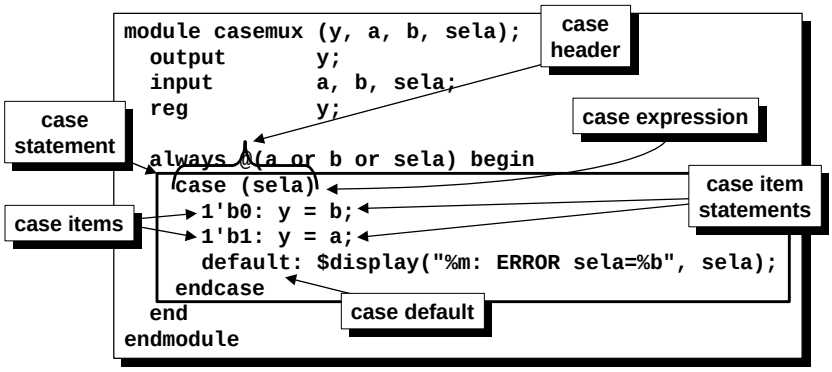
- Verilog is case sensitive
- Make Verilog file names match the Verilog module names

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



Case Statement Types


```
module c1 (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
```

case statement

```
module cz1 (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
```

casez statement

```
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
```

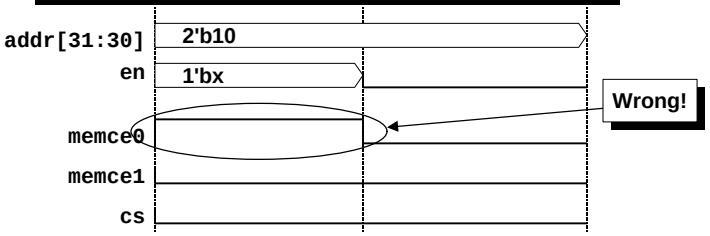
casex statement

If a = ...	case	casez	casex
1'b0:	0	0	0
1'b1:	1	1	1
1'bz:	2	0	0
1'bx:	3	3	0

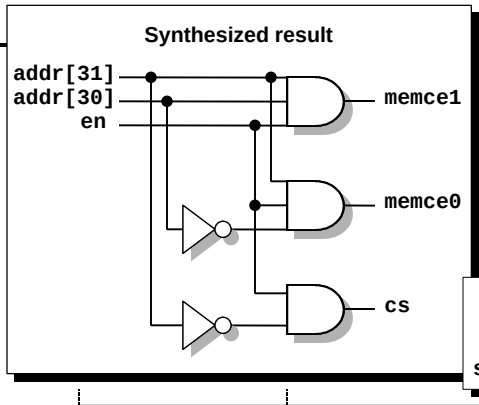
When should casez & casex be used for synthesis?

Casex

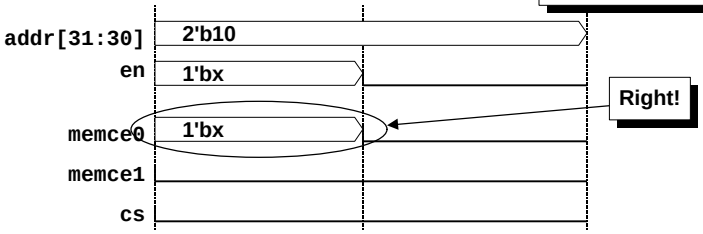
```
module code6 (memce0, memce1, cs, en, addr);
  output         memce0, memce1, cs;
  input          en;
  input  [31:30] addr;
  reg            memce0, memce1, cs;

  always @(addr or en) begin
    {memce0, memce1, cs} = 3'b0;
    casex ({addr, en})
      3'b101: memce0 = 1'b1;
      3'b111: memce1 = 1'b1;
      3'b0?1: cs     = 1'b1;
    endcase
  end
endmodule
```



Casex

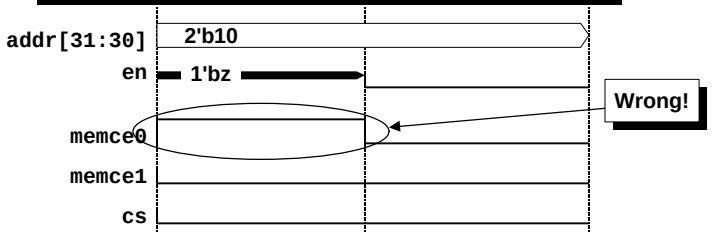


Pre-synthesis simulation does not match post-synthesis simulation

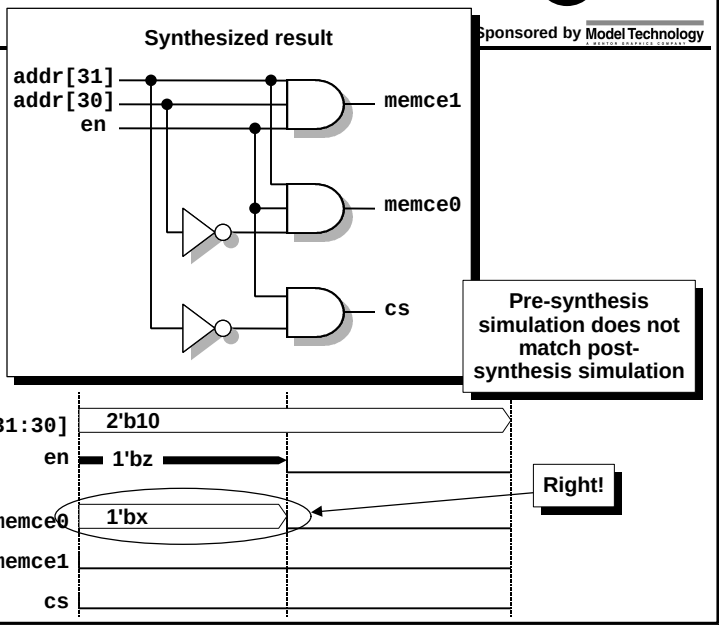


Casez

```
module code7 (memce0, memce1, cs, en, addr);
  output         memce0, memce1, cs;
  input          en;
  input  [31:30] addr;
  reg            memce0, memce1, cs;

  always @(addr or en) begin
    {memce0, memce1, cs} = 3'b0;
    casez ({addr, en})
      3'b101: memce0 = 1'b1;
      3'b111: memce1 = 1'b1;
      3'b0?1: cs     = 1'b1;
    endcase
  end
endmodule
```



Casez



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!

Sponsored by Model Technology

Sunburst Design

State Machines

Abbreviated FSMs (Finite State Machines)

State Machine Classification

Sunburst Design

Sponsored by Model Technology

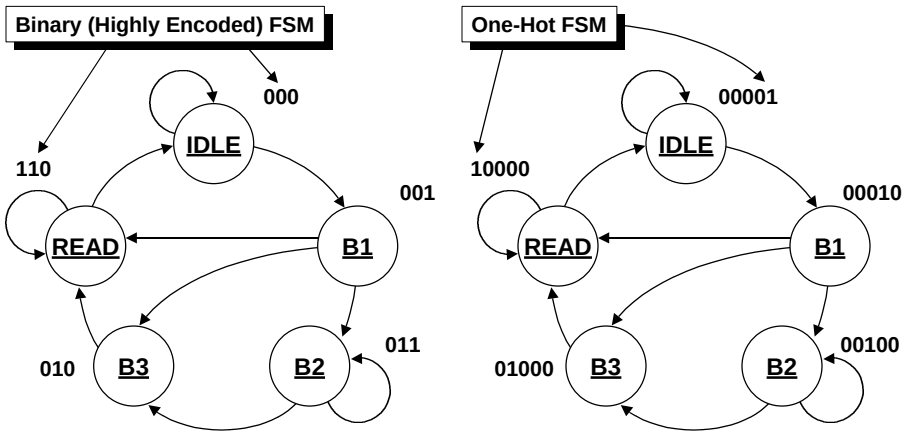
- Moore - outputs are only a function of the present state
- Mealy - one or more of the outputs are a function of the present state and one or more of the inputs
- For an “n”-state FSM (examples on next slide):
  - Binary (highly encoded) = Ceiling $\log_2 n$ flip-flops
    5 states -> 3 FF’s (fewest number of FF’s)
  - One-Hot = n flip-flops
    5 states -> 5 FF’s (smaller combinational logic)

71

Binary & One-Hot FSMs

Sunburst Design

Sponsored by Model Technology



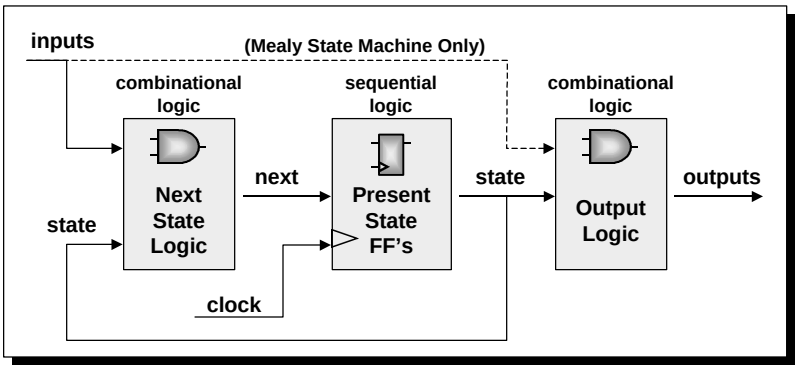
72

Primary State Machine Blocks

Sunburst Design

Sponsored by Model Technology

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



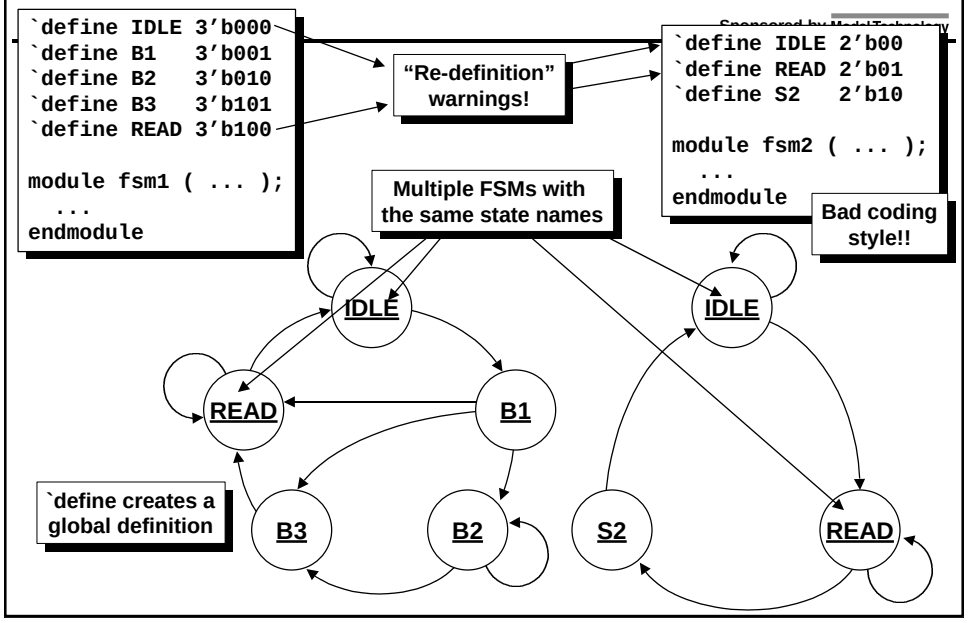
State Machine Coding Styles

- 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

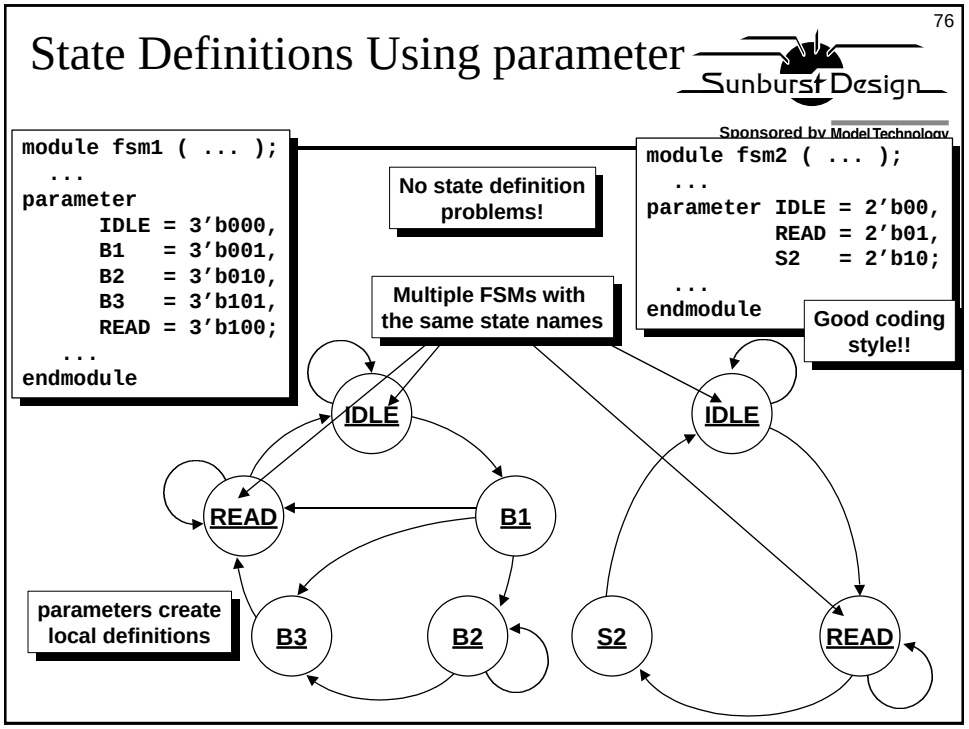
General State Machine Guidelines

- 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)

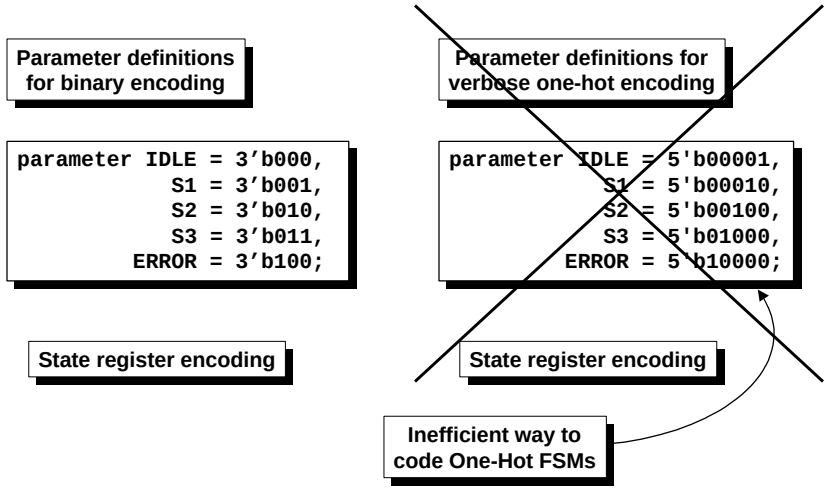
State Definitions Using `define



State Definitions Using parameter



State Parameter Definitions

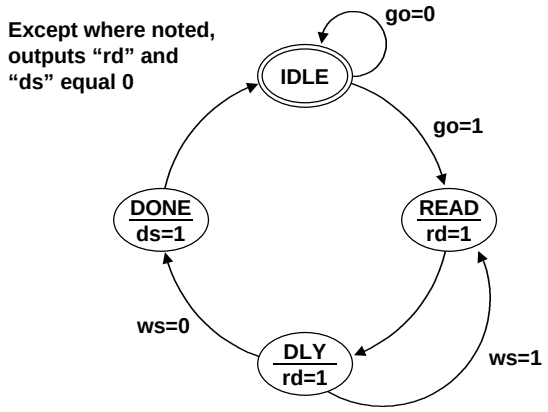


Two-Always Block State Machines

- (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



Two-Always Block Coding Style

(Symbolic Parameter Assignments - Sequential Always Block)


```
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

(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 :          next = DLY;
      DLY  :
             if (!ws) next = DONE;
             else     next = READ;
      DONE :          next = IDLE;
    endcase
  end

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

Output method #1 (continuous assignments)

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;
             end
      DONE : begin      ds   = 1'b1;
                        next = IDLE;
             end
    endcase
  end
endmodule
```

Initial default value assignment - initializes the outputs to a default state

Output method #2 (always-block assignments)

Next State 'x'-Default Assignment

- Make an initial default state assignment

```
next = 2'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

```
next = IDLE;
```
- OR -
```
next = ERROR;
```
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

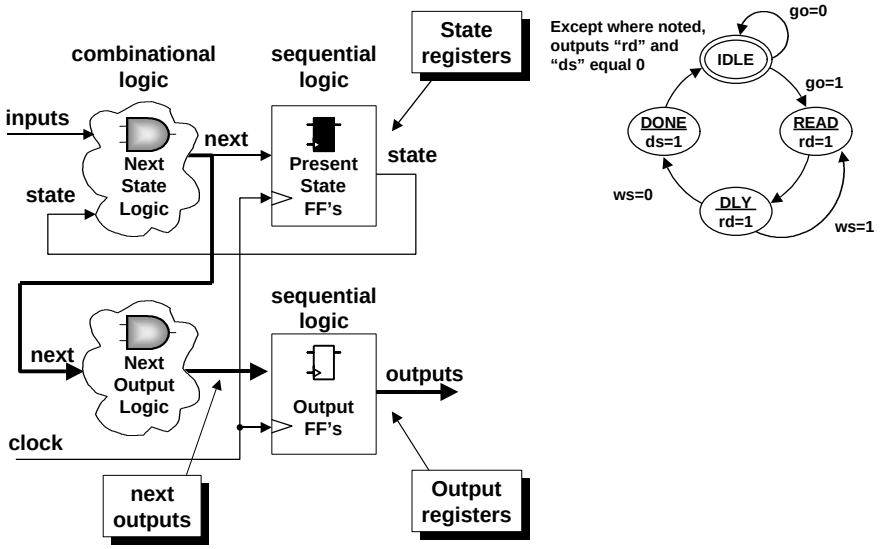
    -OR-
  - Place output assignments in the combinational always block

    Input conditions have already been decoded

Registering The "Next Outputs"

Sunburst Design

Sponsored by Model Technology



Registered Outputs

(Add a third (sequential) always block)

Sunburst Design

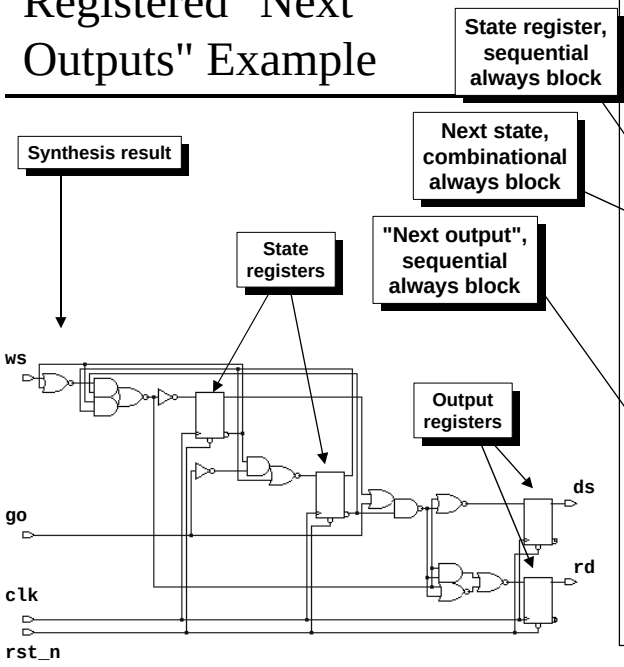
Sponsored by Model Technology

```
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




```
module fsm1b (ds, rd, go, ws, clk, rst_n);
  output ds, rd;
  input  go, ws;
  input  clk, rst_n;
  reg    ds, rd;

  parameter [1:0] IDLE = 2'b00,
                  READ = 2'b01,
                  DLY  = 2'b10,
                  DONE = 2'b11;

  reg [1:0] state, next;

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

  always @(state or go or ws) begin
    next = 2'bx;
    case (state)
      IDLE: if (go) next = READ;
            else    next = IDLE;

      READ:         next = DLY;

      DLY:  if (ws) next = READ;
            else    next = DONE;

      DONE:         next = IDLE;
    endcase
  end

  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
```


State Machines - Mealy Outputs

Sunburst Design

Sponsored by Model Technology

- 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

    `assign <out> = (state == READ) & !read_strobe_n;`

    -OR-

  - In the next state combinational always block

    `READ : if (!read_strobe_n) <out> = 1;`

Onehot Coding Style

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


```
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

Onehot Coding Style

(Two Always Blocks - Continuous Assignment Outputs)


```
...
  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
```

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

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

95

Verilog Enhancement Strategy

Sponsored by Model Technology

- 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!

96

Top-Five Verilog Enhancement Requests

Sponsored by Model Technology

- IVC-1996 birds of a feather
  - Session chaired by Kurt Baty of WSFDB
- #1 - Verilog generate statement
- #2 - Multi-dimensional arrays ← **Look at this enhancement first**
- #3 - Better Verilog file I/O
- #4 - Re-entrant tasks
- #5 - Better configuration control

NOTE: Verilog-2001 in this section is not fully tested! (waiting for a V2001 simulator!)

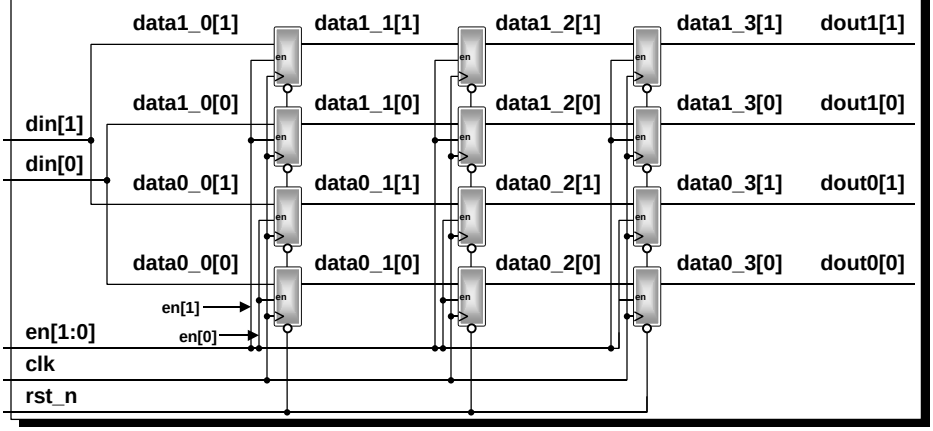
Structural Dual-Pipeline Model

Sunburst Design

Sponsored by Model Technology

2-bit wire-bus declarations

```
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;
```



Dual Pipeline Code

Sunburst Design

Sponsored by Model Technology

1-dimensional wire array declaration

```
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_2[0]), .d(data0_1[0]), .clk(clk), .en(en[0]), .rst_n(rst_n));
  dff u020 (.q(data0_3[0]), .d(data0_2[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_2[1]), .d(data0_1[1]), .clk(clk), .en(en[0]), .rst_n(rst_n));
  dff u021 (.q(data0_3[1]), .d(data0_2[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_2[0]), .d(data1_1[0]), .clk(clk), .en(en[1]), .rst_n(rst_n));
  dff u120 (.q(data1_3[0]), .d(data1_2[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_2[1]), .d(data1_1[1]), .clk(clk), .en(en[1]), .rst_n(rst_n));
  dff u121 (.q(data1_3[1]), .d(data1_2[1]), .clk(clk), .en(en[1]), .rst_n(rst_n));

  assign dout1 = data1_3;
  assign dout0 = data0_3;
endmodule
```

2-bit assignments

Bit reference listed last

Multi-Dimensional

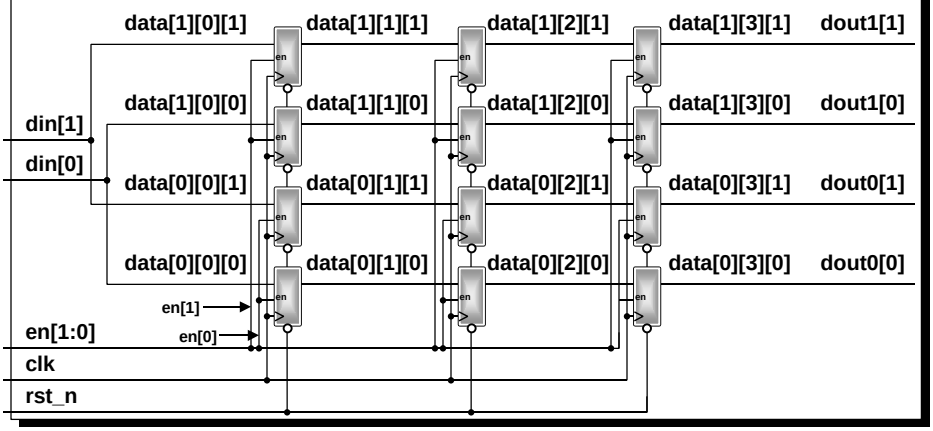
Verilog-2001 Enhancement #2

Sunburst Design

Synthesizable Soon!

3-dimensional wire array declaration

```
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];
```



Multi-Dimensional Array Code

Sunburst Design

Sponsored by Model Technology

3-dimensional wire array declaration

```
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] = din;
  assign data[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
```

Word assignment 2 index variables

Bit reference listed last

Bit assignment 3 index variables

Generate Statement

Verilog-2001 Enhancement #1

Sunburst Design

Synthesizable Soon!

```
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
```

Three genvar variables declared: `i, j, k`

3-dimensional wire array declaration

Named blocks are required

Three nested generate for-loops

First generated dff instance name: `ff1[0].ff2[0].ff3[0].u1`

Last generated dff instance name: `ff1[1].ff2[2].ff3[1].u1`

Generate Statement
genvar Index Variable

Sunburst Design

Sponsored by Model Technology

- 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

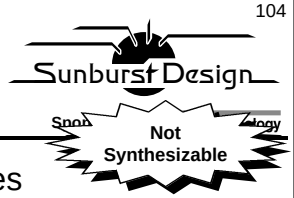
```
module ...
  generate if ((A_WIDTH < 8) || (B_WIDTH < 8))
    CLA_multiplier u1 (...);     // CLA multiplier
  else
    WALLACE_multiplier u1 (...); // Wallace-tree multiplier
  endgenerate
  ...
endmodule
```

- case-generate

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


Enhanced File I/O

Verilog-2001 Enhancement #3



- Functions and tasks that open and close files

Verilog-1995 Multi-Channel Descriptor

integer MSB="0"

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

Verilog-2001 File Descriptor

integer MSB="1"

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

Type			Verilog-2001 File Types
r	rb		open for reading
w	wb		truncate to zero length or create for writing
a	ab		append; open for writing at end of file, or create for writing
r+	r+b	rb+	open for update (reading and writing)
w+	w+b	wb+	truncate or create for update
a+	a+b	ab+	append; open or create for update at end-of-file

Enhanced File I/O (cont.)

Verilog-2001 Enhancement #3

Sunburst Design

Sponsored by Model Technology

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

```
$ferror
$fgetc
$fgets
$fread
$fscanf
$fseek
$ftell
$fungetc
$fungetf
$rewind
$sformat
$sscanf
$swrite   (also $swriteb, $swriteo, $swriteh)
$ungetc
```

New Verilog-2001 file I/O commands

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

Re-Entrant Tasks & Functions

Verilog-2001 Enhancement #4

Sunburst Design

Synthesizable Soon!

```
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

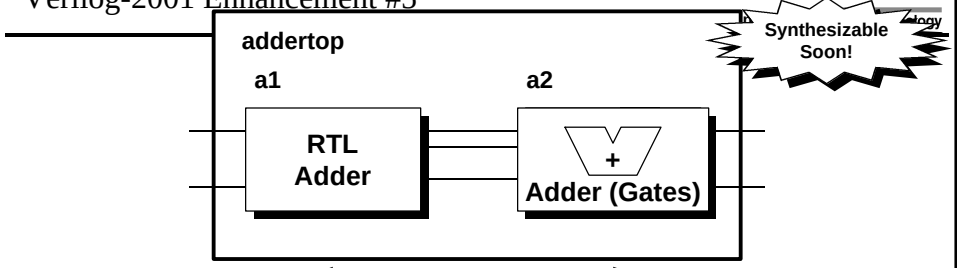
Automatic tasks & functions makes recursion possible

Verilog Configuration Files

Verilog-2001 Enhancement #5

Sunburst Design

Synthesizable Soon!



Simulation specification #1
addertop.a1 - RTL model
addertop.a2 - RTL model

file: `rtl.cfg`

```
config rtl;
  design   rtlLib.addertop;
  default  liblist rtlLib;
endconfig
```

Simulation specification #2
addertop.a1 - RTL model
addertop.a2 - gate-level model

file: `mixed.cfg`

```
config mixed;
  design   rtlLib.addertop;
  default  liblist rtlLib;
  instance addertop.a2 liblist gateLib;
endconfig
```

Libraries on the next slide

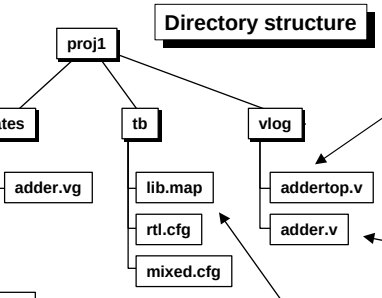
Simulation Configurations

Verilog-2001 Enhancement #5

Sunburst Design

Sponsored by Model Technology

Directory structure



file: `addertop.v`

```
module addertop();
  adder a1(...);
  adder a2(...);
endmodule
```

file: `adder.vg`

```
module adder();
  // gate-level adder
  // description
endmodule
```

file: `adder.v`

```
module adder();
  // RTL adder
  // description
endmodule
```

file: `lib.map`

```
library rtlLib ../proj1/vlog/*.v
library gateLib ../proj1/gates/*.vg
```

rtl.cfg & mixed.cfg on previous slide

ANSI-C Style Ports

Works with ModelSim 5.5!

Sunburst Design

Synthesizable Soon!

q - in the port list

```
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 - port type is "output"

q - data type is "reg"

Verbose port declarations

(port-list, port direction and port data type)

```
module dffarn (
  output reg q,
  input      d, clk, rst_n);

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


Comma-Separated Sensitivity List

Works with ModelSim 5.5!

Sunburst Design

Synthesizable Soon!

Parameterized Verilog-2001 memory (ram1) model

```
module ram1 (addr, data, en, rw_n);
  parameter ADDR_SIZE = 10;
  parameter DATA_SIZE = 8;
  parameter  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 @(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

Incomplete Sensitivity List

Sunburst Design

Sponsored by Model Technology

```
module code1a (o, a, b);
  output o;
  input  a, b;
  reg    o;

  always @(a or b)
    o = a & b;
endmodule
```

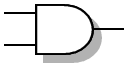
Full sensitivity list

```
module code1b (o, a, b);
  output o;
  input  a, b;
  reg    o;

  always @(a)
    o = a & b;
endmodule
```

Incomplete sensitivity list

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

Synthesized result!



```
module code1c (o, a, b);
  output o;
  input  a, b;
  reg    o;

  always
    o = a & b;
endmodule
```

No sensitivity list

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

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

Incomplete Sensitivity List

Sunburst Design

Sponsored by Model Technology

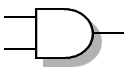
```
module code1a (o, a, b);
  output o;
  input  a, b;
  reg    o;

  always @(a or b)
    o = a & b;
endmodule
```

Pre-synthesis simulation matches post-synthesis simulation

```
module code1b (o, a, b);
  output o;
  input  a, b;
  reg    o;

  always @(a)
    o = a & b;
endmodule
```

Pre-synthesis simulation does not match post-synthesis simulation

Synthesized result!



```
module code1c (o, a, b);
  output o;
  input  a, b;
  reg    o;

  always
    o = a & b;
endmodule
```

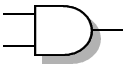
Pre-synthesis simulation hangs!

@* 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

Synthesizable Soon!

Parameterized Verilog-2001 memory (ram1) model

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

MEM_DEPTH is automatically sized from the ADDR_SIZE

localparam(s) cannot 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
```

Combinational @* sensitivity list

115

Errata - @*

(Information Missing from the Verilog-2001 Standard)

Sponsored by Model Technology

```
module decoder (
  output reg [7:0] y,
  input      [2:0] a,
  input            en);

  always @* begin
    y = 8'hff;
    y[a] = !en;
  end
endmodule
```

3-to-8 decoder with enable

@* is the same as @(a or en)

Code from a One-Hot State Machine design

@* is the same as @(state or go or ws)

Index variables should be part of a combinational sensitivity list

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

```
...
  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
...
```

116

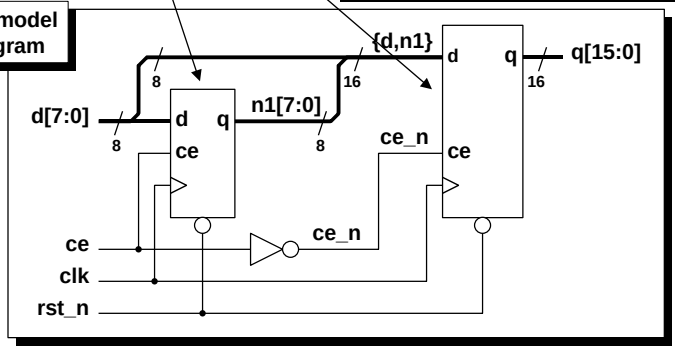
Passing Parameters to Instances

Parameterized regblk model

```
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

demuxreg model block diagram



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));
  regblk u2 (.q (q), .d({d,n1}), .ce(ce_n), .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!!*

```
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
```

- 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);
  regblk #( 8) u1 (.q(n1), .d (d),     .ce(ce),   .clk(clk), .rst_n(rst_n));
  regblk #(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!!*

```
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
```

Named Parameter Redefinition

Verilog-2001 Enhancement

Sunburst Design

Sponsored by Model Technology

```
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)

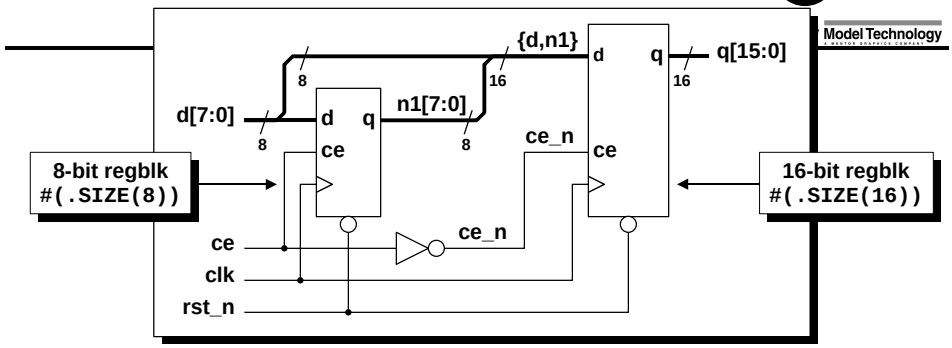
Only the parameters that change must be listed

```
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
```


Demuxreg Verilog-2001 Model

Sunburst Design

Model Technology




```
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

16 data BIDIR I/O pads

32 addr IBUF I/O pads

```
module top_pads1 (pdata, paddr, pctl1, pctl2, pwr, pclk);
  input  [15:0] pdata;                    // pad data bus
  inout  [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),
               .sig1(ctl1), .sig2(ctl2), .sig3(wr), .clk(clk));

  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 b15 (.N2(data[15]), .pN1(pdata[15]), .WR(wr));
  BIDIR b14 (.N2(data[14]), .pN1(pdata[14]), .WR(wr));
  ...
  BIDIR b1  (.N2(data[ 1]), .pN1(pdata[ 1]), .WR(wr));
  BIDIR b0  (.N2(data[ 0]), .pN1(pdata[ 0]), .WR(wr));

  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
```

123

Top-Level Module Instantiation

Verilog Generate Method

Sponsored by Model Technology

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

16 data BIDIR I/O pads

32 addr IBUF I/O pads

```
module top_pads2 (pdata, paddr, pctl1, pctl2, pwr, pclk);
  input  [15:0] pdata;                    // pad data bus
  inout  [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),
               .sig1(ctl1), .sig2(ctl2), .sig3(wr), .clk(clk));

  genvar i;

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

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

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

endmodule
```

124

Top-Level Module Instantiation

Array of Instances

Works with ModelSim 5.5!

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

16 data BIDIR I/O pads

32 addr IBUF I/O pads

```
module top_pads3 (pdata, paddr, pctl1, pctl2, pwr, pclk);
  input  [15:0] pdata;                    // pad data bus
  inout  [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),
               .sig1(ctl1), .sig2(ctl2), .sig3(wr), .clk(clk));

  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
```

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, pctl1, pctl2, pwr, pclk);
  input  [15:0] pdata;                    // pad data bus
  inout  [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 (@*, .sig1(ctl1), .sig2(ctl2), .sig3(wr));

  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
```

@* automatic port matching??

```
main_blk u1 (@*, .sig1(ctl1), .sig2(ctl2), .sig3(wr));
```


Attributes

- New tokens

```
(*   *)
```

VSG members call these "funny braces"

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

Synthetic comments require that all comments are parsed

```
   // rtl_synthesis full_case parallel_case

                replace with

(* rtl_synthesis, full_case, parallel_case *)
```

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
```

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

  assign y = en ? a : {SIZE{1'bz}};
endmodule
```

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

Verilog-1995: y = 64'h0000_0000_zzzz_zzzz

Verilog-2001: y = 64'hzzzz_zzzz_zzzz_zzzz

Errata

A Few Mistakes "Slipped" Into The LRM

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

No range??

depth is only 1-bit wide (oops!)

```
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
```

If depth = 0 the for-loop never executes ...

Simple 1-statement implementation

```
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

Barrel Shifter model from a popular synthesis book

```
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
```

```
  function [5:0] rol6;
    input [5:0] A;
    input [2:0] NoShifts;

    reg [6:0] A_ExtendLeft;
    integer N;

    begin
      A_ExtendLeft = {1'b0, A};
      for (N=1; N<=NoShifts; N=N+1)
         begin
           A_ExtendLeft = A_ExtendLeft << 1;
           A_ExtendLeft[0] = A_ExtendLeft[6];
         end
      rol6 = A_ExtendLeft[5:0];
    end
  endfunction
endmodule
```

No `rol6` function necessary - use part-selects and concatenation

Barrel Shifter

Better RTL Model

Sunburst Design

Sponsored by Model Technology

```
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 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]};
        2: Y = {A[3:0],A[5:4]};
        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'bx;
      endcase
    end
endmodule
```

No always block or case statement necessary, use a continuous assignment

```
{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}}}
```


Barrel Shifter

Two-Parameter RTL Model

Sunburst Design

Sponsored by Model Technology

```
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
```

Let's turn this into a parameterized model

```
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
```

Two parameters, WIDTH & CNT_SIZE

Barrel Shifter - Verilog-2001

One-Parameter RTL Model

```
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
```

Verilog-2001 parameter and port list

```
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
```

One parameter WIDTH

localparam and a constant function are used to set the CNT_SIZE

```
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

ANSI-C style parameters and ports

```
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
```

Local parameter

Constant function

Data type included in the port declaration

ANSI-C style ports

```
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
```

Comma-separated sensitivity list

Parameter redefinition using named parameters

Registered Barrel Shifter

Verilog-2001 Version

- Build the whole model in one simple module

```
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

Sponsored by

Model Technology
A MENTOR GRAPHICS COMPANY

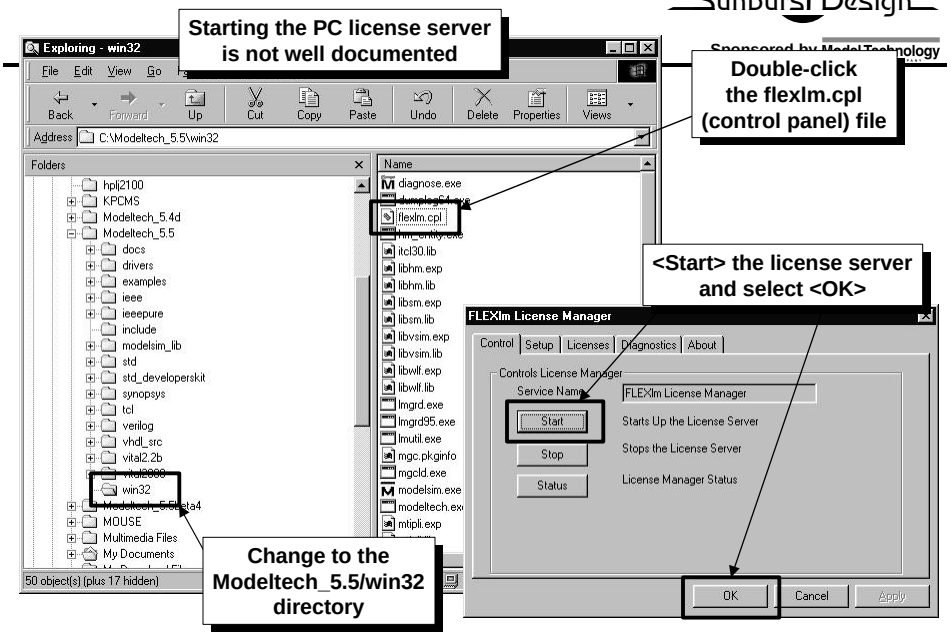
Sunburst Design

ModelSim User Notes

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

The ModelSim PC License Server

Sunburst Design



Misleading Port Warning


```
`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
```

Contrary to the warning, there is no problem here

This should be a syntax error

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

```
# vsim -do {run -all} -c tb
# Loading work.tb
# Loading work.myand2
# WARNING: myand2.v(5): [TFMPC] - Too few port connections.
#        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 vimr 'vsim -c -do "run -all"'
```

Add this UNIX alias to the .cshrc file

  - execute the command:
    vsimr <*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>
```

Very important for ModelSim users!

  - 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)


```
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
```

+define+RTL to run an RTL version of the d-flip-flop

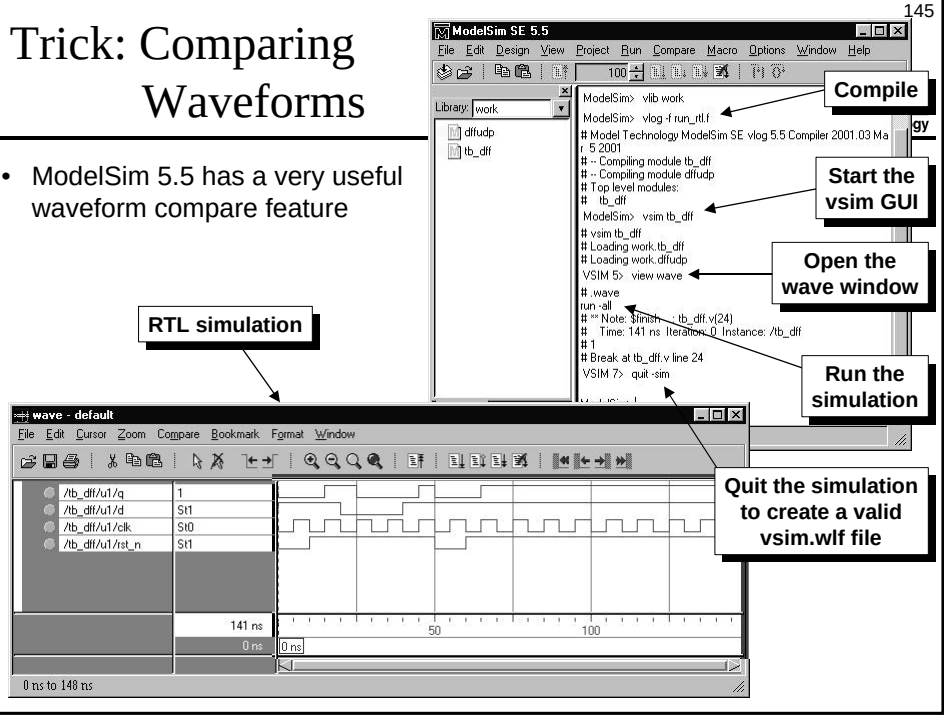
```
primitive udpdff (q, d, clk, rst_n);
  output q;
  input  d, clk, rst_n;
  reg    q;

  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
      *    ?    ?   :  ?  :  - ;
  endtable
endprimitive
```

+define+UDP to run a UDP version of the d-flip-flop

D-flip-flop User Defined Primitive model (with a bug!)

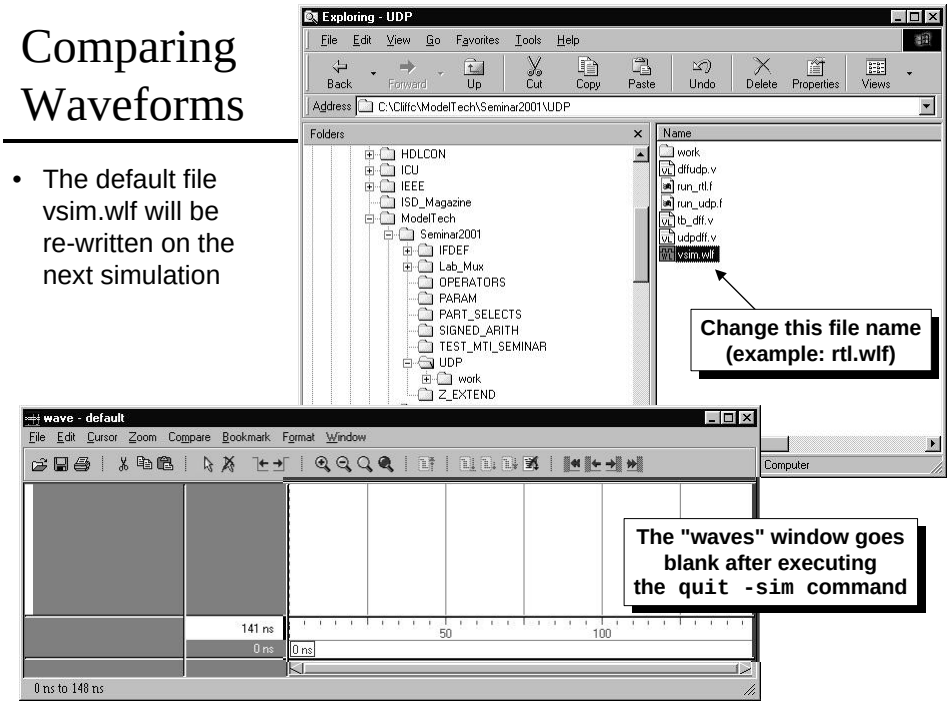
Trick: Comparing Waveforms

- ModelSim 5.5 has a very useful waveform compare feature



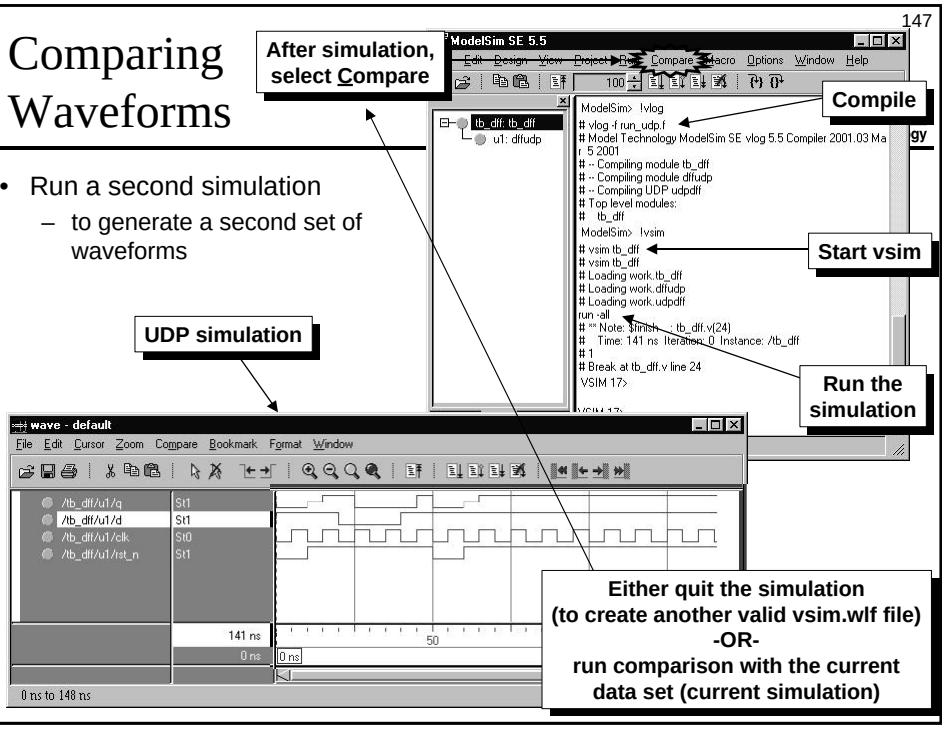
Comparing Waveforms

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



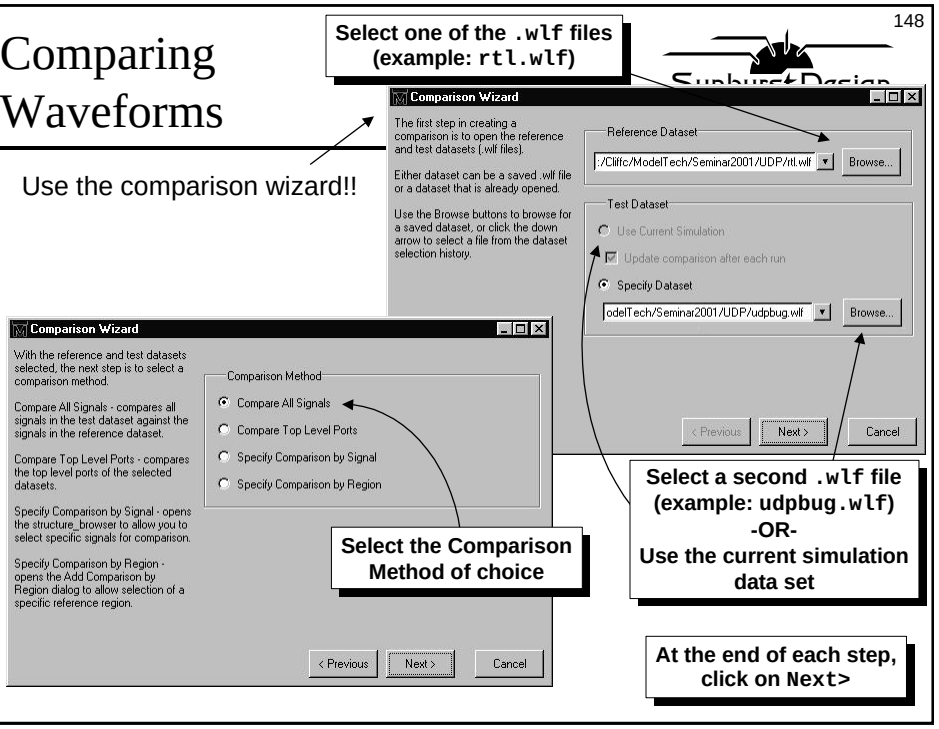
Comparing Waveforms

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

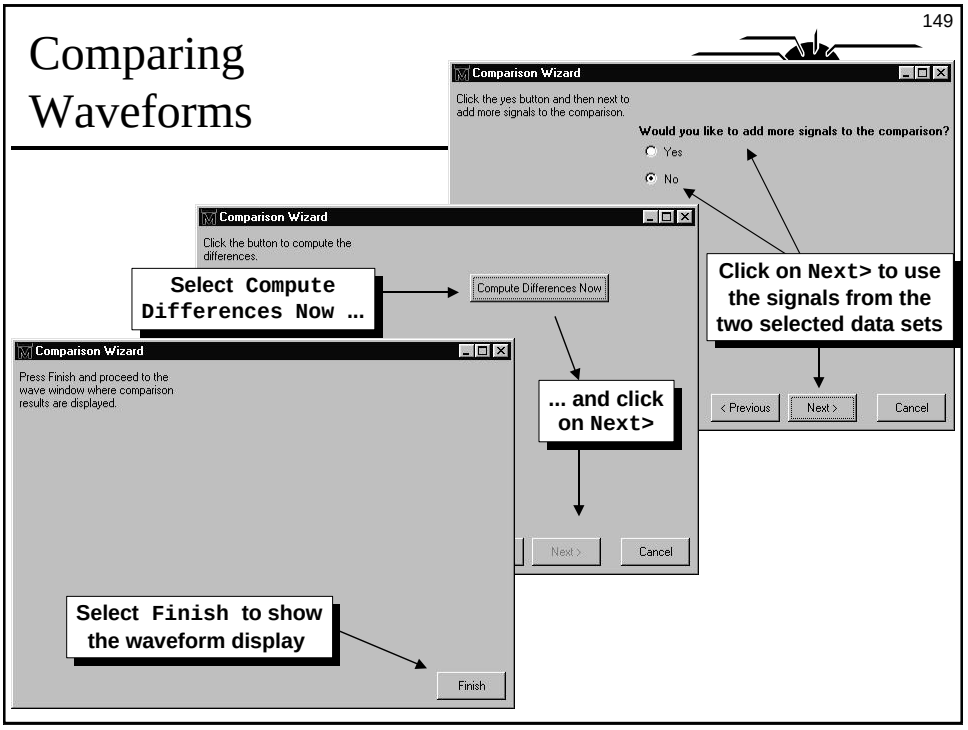


Comparing Waveforms

- Use the comparison wizard!!



Comparing Waveforms

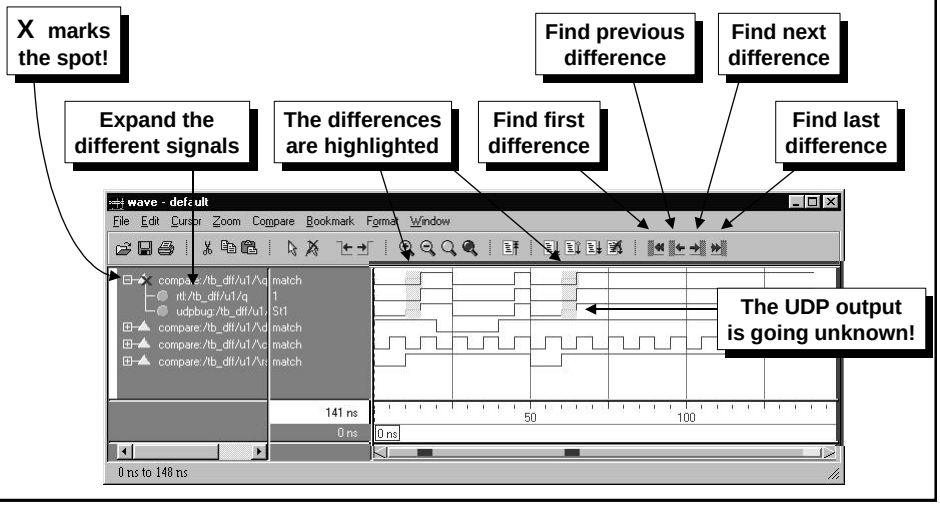


Comparing Waveforms

Sunburst Design

Sponsored by Model Technology

- "Waveform compare" highlights differences between signals



Waveform Comparison Helps Isolate Problems


```
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
```

A missing UDP entry for rst_n transitions caused the bug

```
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
```


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