

# 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](http://www.sunburst-design.com/papers)

Sponsored by

**Model Technology**  
A MENTOR GRAPHICS COMPANY



3

## 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



Sponsored by Model Technology

4

- Register types are only assigned inside of procedural blocks

**LHS must be a reg type**

```
reg [8:0] sum;  
always @(a or b)  
    sum = a + b;
```

- Net types are assigned or driven outside of procedural blocks

**LHS must be a net type**

```
wire [8:0] sum;  
assign sum = a + b;
```

**LHS (outputs) must be net types**

```
wire y;  
and g1(y, c, d);
```

## Blocking & Nonblocking Assignments



5

- 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



6

- 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



7

- 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



8

- 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 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 **Execution order is not guarantee!!**

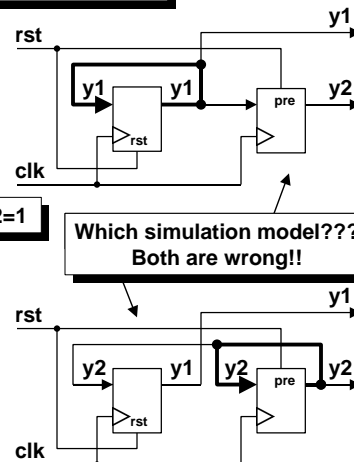
```
module fbosc1 (y1, y2, clk, rst);
    output y1, y2;
    input  clk, rst;
    reg   y1, y2;

    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
```

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

On rst, y1=0 and y2=1



## Nonblocking Assignments

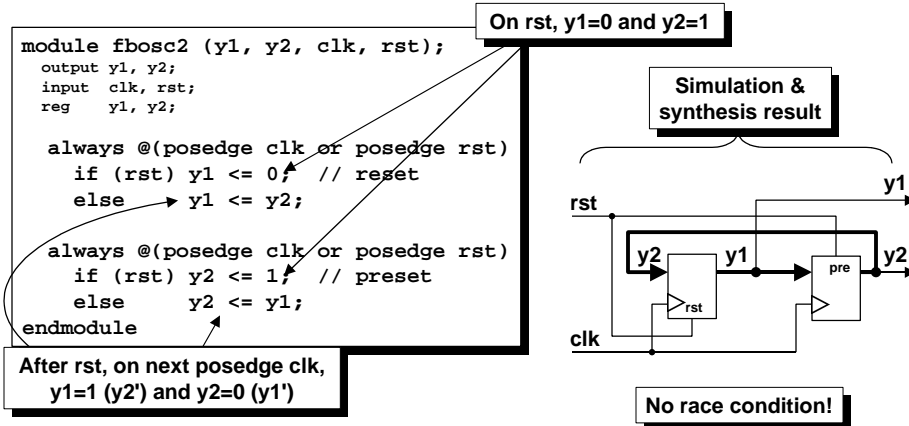
- Nonblocking assignment operator: **<=**
- Execution of nonblocking assignments can be viewed as a two-step process:
  - ① Evaluate the RHS of nonblocking statements at the beginning of the time step
  - ② Update the LHS of nonblocking statements at the end of the time step
- Allows assignment scheduling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

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



## 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

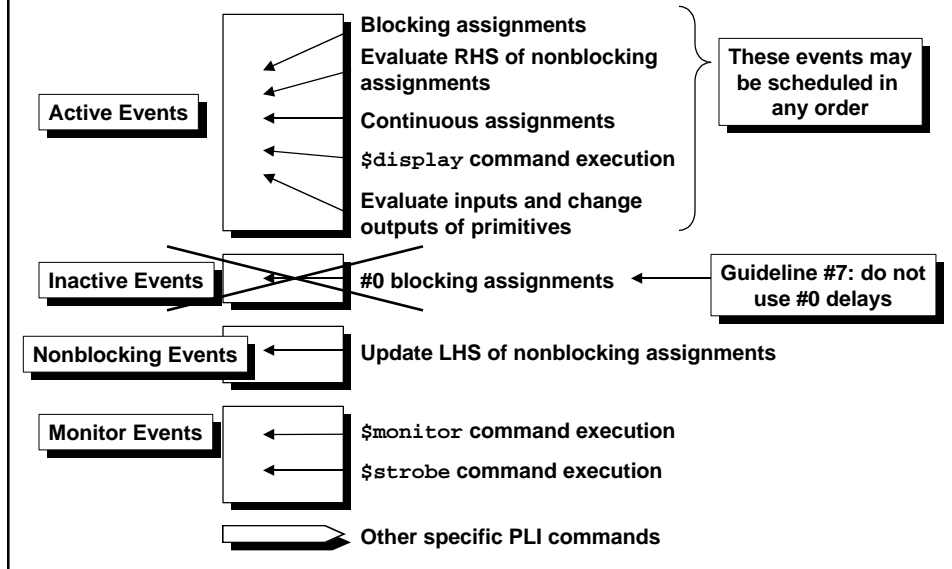
**Guideline #4:** In general, do not mix blocking and nonblocking assignments in the same always block

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

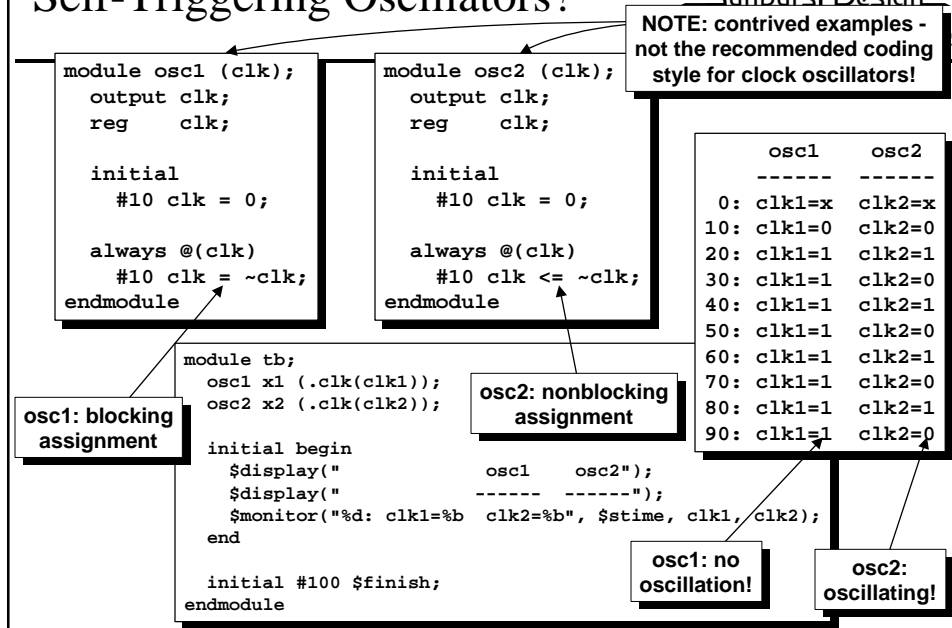
**Guideline #6:** Use \$strobe to display values that have been assigned using nonblocking assignments

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

# IEEE1364-1995 Verilog Stratified Event Queue



## Self-Triggering Oscillators?



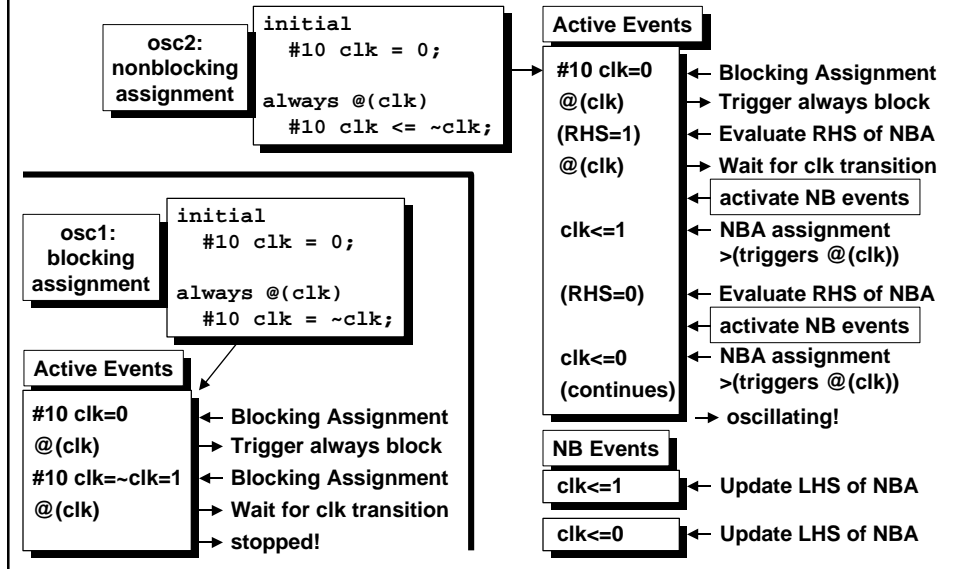
# Self-Triggering?

(not with blocking assignments)



Sponsored by Model Technology

15



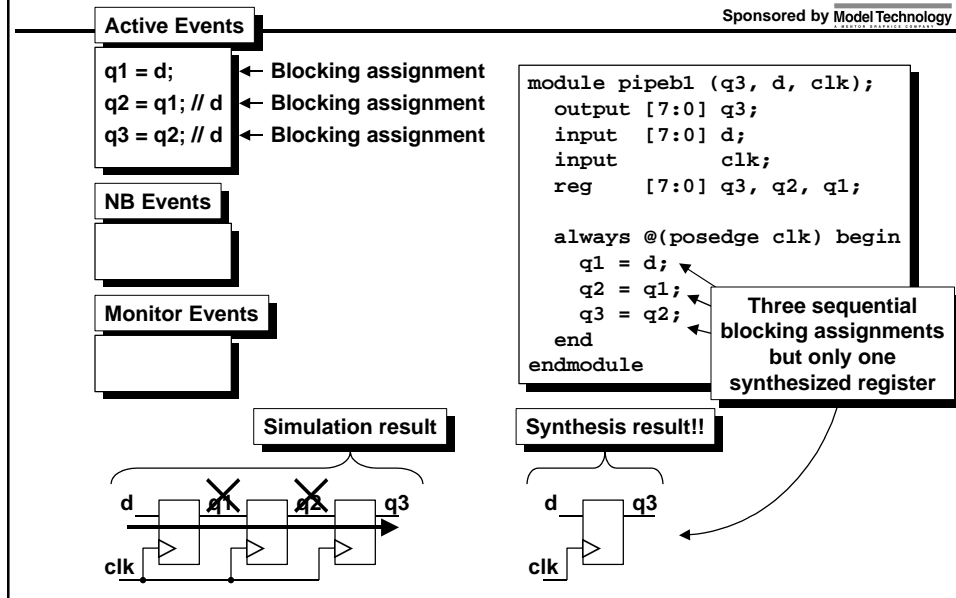
# Pipeline

Blocking Assignments Style #1



Sponsored by Model Technology

16





# Pipeline

## Blocking Assignments Style #2



Sponsored by Model Technology

17

### Active Events

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

### NB Events

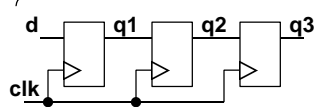
### Monitor Events

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

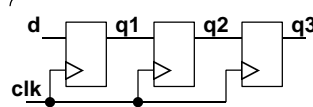
  always @(posedge clk) begin
    q3 = q2;
    q2 = q1;
    q1 = d;
  end
endmodule
```

This ordering  
will work

### Simulation result



### Synthesis result



# Pipeline

## Blocking Assignments Style #3



18

### Active Events

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

### NB Events

### Monitor Events

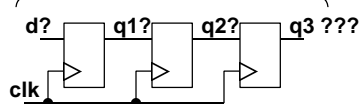
### Executes in any order

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

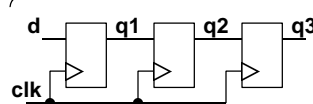
  always @(posedge clk) q1=d;
  always @(posedge clk) q2=q1;
  always @(posedge clk) q3=q2;
endmodule
```

This ordering  
might not work

### Simulation result



### Synthesis result

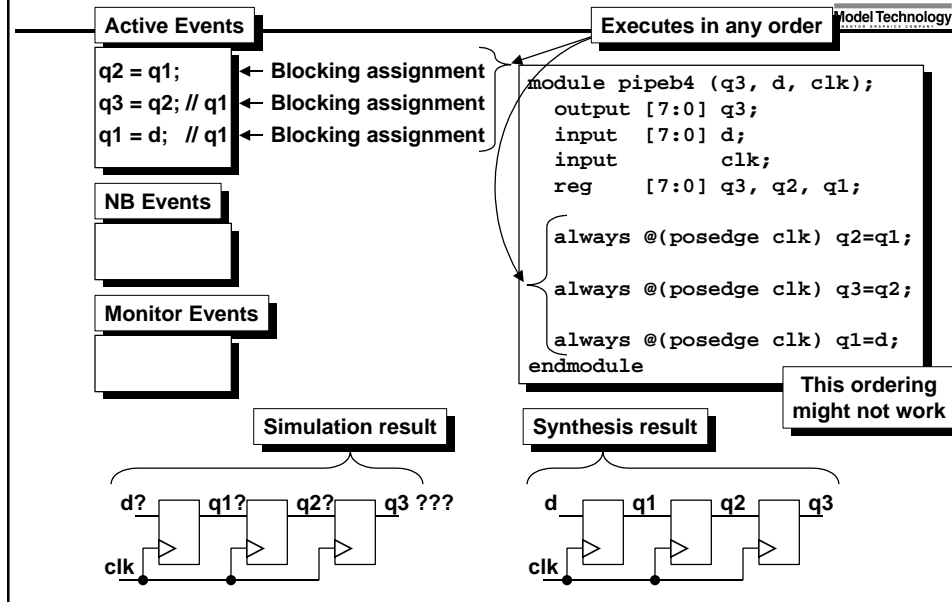


# Pipeline

Blocking Assignments Style #4



19



## Blocking Assignment Coding Styles (Pipelines)



20

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



Sponsored by Model Technology

### Active Events

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

← Evaluate RHS of NBA  
← Evaluate RHS of NBA  
← Evaluate RHS of NBA

### NB Events

q1 <= d'  
q2 <= q1'  
q3 <= q2'

← Update LHS of NBA  
← Update LHS of NBA  
← Update LHS of NBA

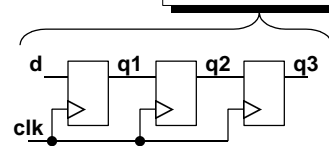
### Monitor Events

```
module pipen1 (q3, d, clk);
  output [7:0] q3;
  input  [7:0] d;
  input   clk;
  reg    [7:0] q3, q2, q1;

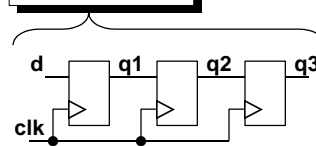
  always @(posedge clk) begin
    q1 <= d;
    q2 <= q1;
    q3 <= q2;
  end
endmodule
```

This ordering  
will work

### Simulation result



### Synthesis result



# Pipeline

## Nonblocking Assignments Style #2



Sponsored by Model Technology

### Active Events

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

← Evaluate RHS of NBA  
← Evaluate RHS of NBA  
← Evaluate RHS of NBA

### NB Events

q3 <= q2'  
q2 <= q1'  
q1 <= d'

← Update LHS of NBA  
← Update LHS of NBA  
← Update LHS of NBA

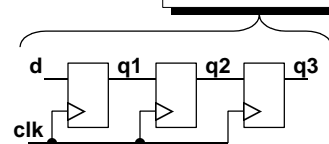
### Monitor Events

```
module pipen2 (q3, d, clk);
  output [7:0] q3;
  input  [7:0] d;
  input   clk;
  reg    [7:0] q3, q2, q1;

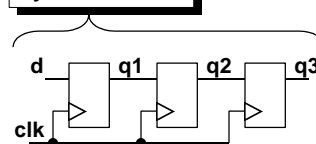
  always @(posedge clk) begin
    q3 <= q2;
    q2 <= q1;
    q1 <= d;
  end
endmodule
```

ANY ordering  
will work

### Simulation result



### Synthesis result



# Pipeline

## Nonblocking Assignments Style #3



Sponsored by Model Technology

23

### Active Events

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

← Evaluate RHS of NBA  
← Evaluate RHS of NBA  
← Evaluate RHS of NBA

### NB Events

q1 <= d'  
q2 <= q1'  
q3 <= q2'

← Update LHS of NBA  
← Update LHS of NBA  
← Update LHS of NBA

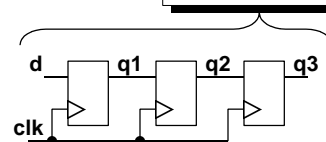
### Monitor Events

```
module pipen3 (q3, d, clk);
    output [7:0] q3;
    input  [7:0] d;
    input   clk;
    reg    [7:0] q3, q2, q1;

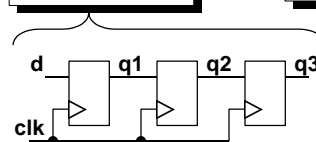
    always @(posedge clk) q1<=d;
    always @(posedge clk) q2<=q1;
    always @(posedge clk) q3<=q2;
endmodule
```

This ordering will work

### Simulation result



### Synthesis result



# Pipeline

## Nonblocking Assignments Style #4



Sponsored by Model Technology

24

### Active Events

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

← Evaluate RHS of NBA  
← Evaluate RHS of NBA  
← Evaluate RHS of NBA

### NB Events

q2 <= q1'  
q3 <= q2'  
q1 <= d'

← Update LHS of NBA  
← Update LHS of NBA  
← Update LHS of NBA

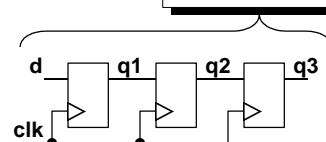
### Monitor Events

```
module pipen4 (q3, d, clk);
    output [7:0] q3;
    input  [7:0] d;
    input   clk;
    reg    [7:0] q3, q2, q1;

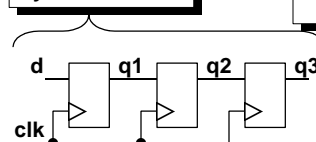
    always @(posedge clk) q2<=q1;
    always @(posedge clk) q3<=q2;
    always @(posedge clk) q1<=d;
endmodule
```

ANY NEW ordering will work

### Simulation result



### Synthesis result



# 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

27



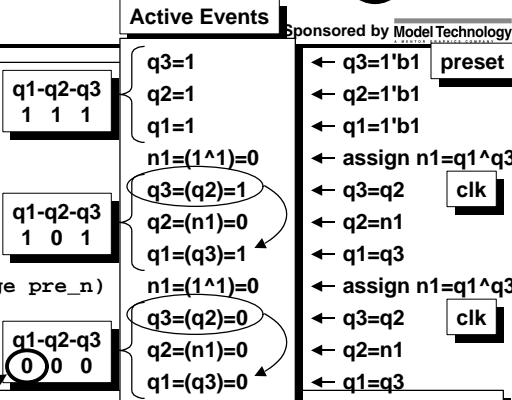
Sponsored by Model Technology

```

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

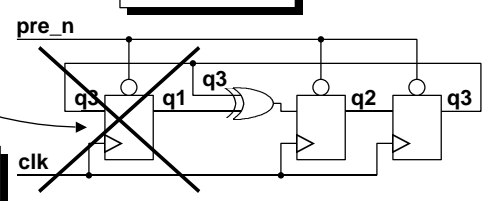
    assign n1 = q1 ^ q3;

    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
    
```



Wrong value!  
Should be "1"

Synthesizes, but does  
not simulate correctly



# LFSR

## Blocking Assignments Style #2

28



```

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

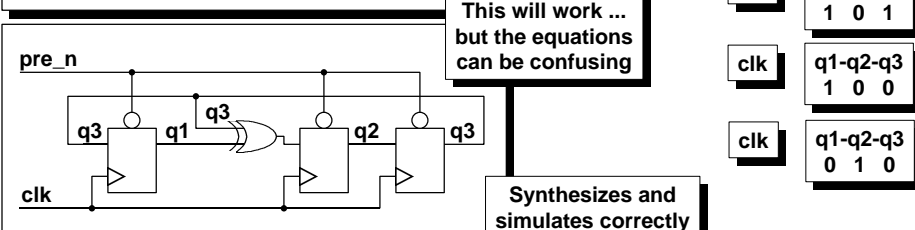
q1,q2,q3=111	← {q3,q2,q1}=3'b111
q1,q2,q3=1(0)1	← {q3,q2,q1}={q2,(q1^q3),q3}
q1,q2,q3=1(0)0	← {q3,q2,q1}={q2,(q1^q3),q3}
q1,q2,q3=0(1)0	← {q3,q2,q1}={q2,(q1^q3),q3}
...	

```

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

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

Synthesizes and  
simulates correctly



# LFSR

Nonblocking Assignments Style #1



Sponsored by Model Technology

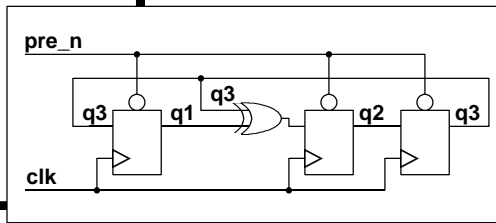
29

```
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
    end
endmodule
```

Synthesizes and  
simulates correctly



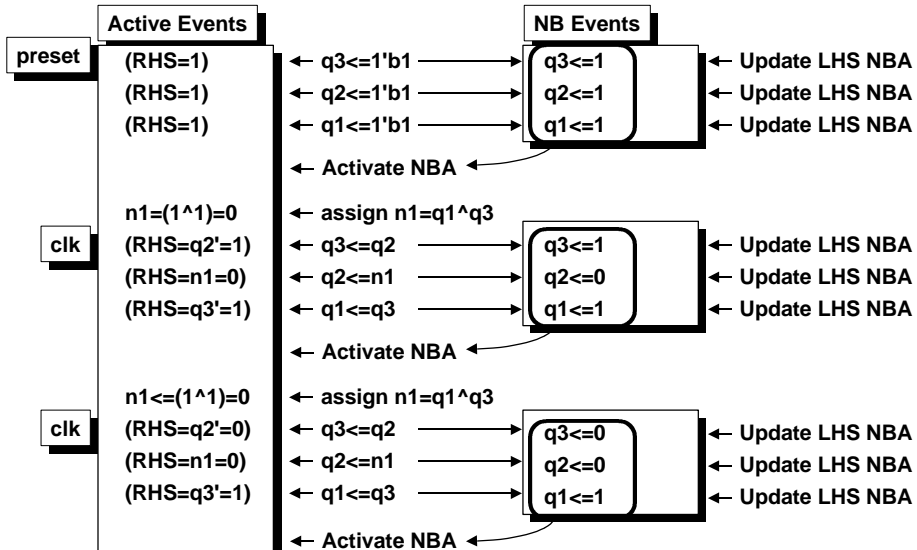
# LFSR

Nonblocking Assignments Style #1



Sponsored by Model Technology

30



# LFSR

## Nonblocking Assignments Style #2



Sponsored by Model Technology

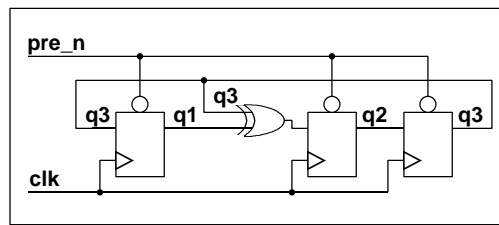
31

```
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)



Sponsored by Model Technology

32

## LFSR Coding Styles

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

1 of 2  
simulates  
correctly

2 of 2  
synthesize  
correctly

2 of 2  
simulate  
correctly

2 of 2  
synthesize  
correctly

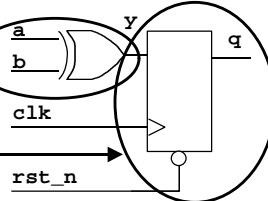


## Nonblocking Assignment Guidelines #1 & #2

- **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
```

Synthesized result!

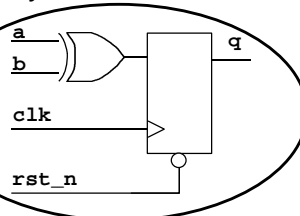


## Nonblocking Assignment Guideline #3

- **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
```

Synthesized result!



# 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

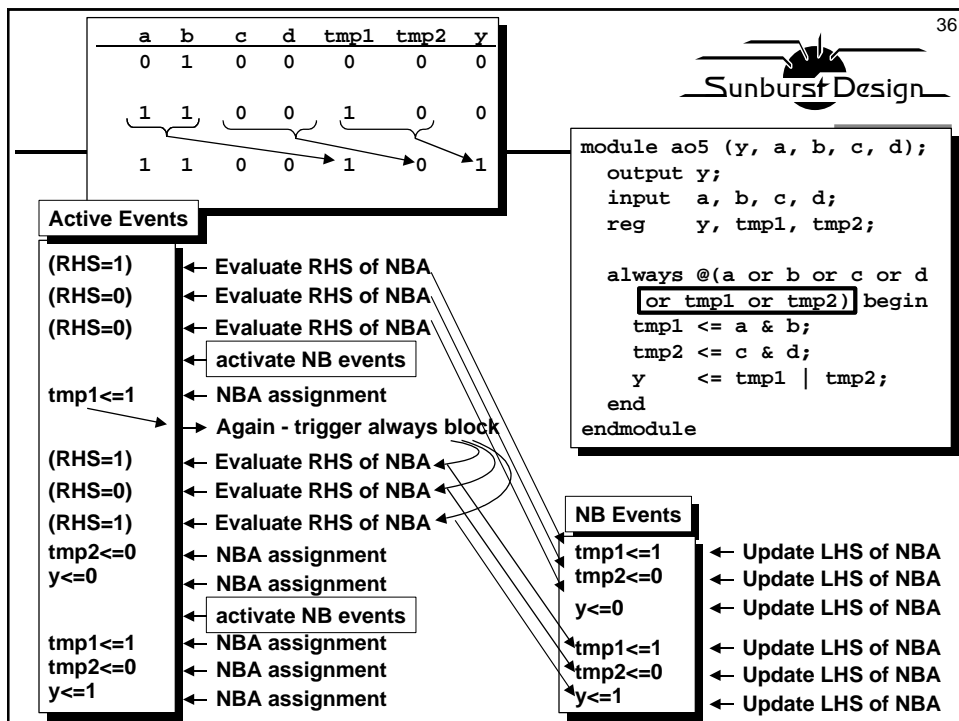
```
module ao4 (y, a, b, c, d);
    output y;
    input a, b, c, d;
    reg y, tmp1, tmp2;

    always @(a or b or c or d) begin
        tmp1 <= a & b;
        tmp2 <= c & d;
        y <= tmp1 | tmp2;
    end
endmodule
```

**WRONG !**

a	b	c	d	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



## Combinational Logic Coding Efficiency



Sponsored by Model Technology

37

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

**All blocking 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
```

```
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



Sponsored by Model Technology

38

```
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
```

**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

41



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;

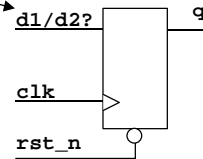
    always @(posedge clk or negedge rst_n)
        if (!rst_n) q <= 1'b0;
        else       q <= d1;

    always @(posedge clk or negedge rst_n)
        if (!rst_n) q <= 1'b0;
        else       q <= d2;
endmodule
```

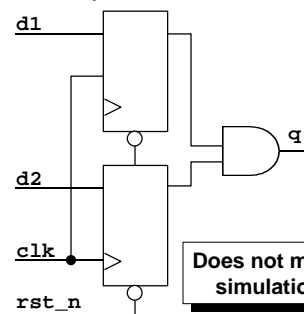
Warning: In design 'badcode1', there is 1 multiple-driver net with unknown wired-logic type.

Race  
condition!

Simulation result??



Synthesis result !!



Sponsored by

**Model Technology**  
A MENTOR GRAPHICS COMPANY

42



## Common Misconceptions About Nonblocking Assignments

## \$display & Nonblocking Assignments

43



Sponsored by Model Technology

- 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

44



Sponsored by Model Technology

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

## Scheduling Example

45



```
module nb_schedule1;
  reg a, b;

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

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
```

```
    $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
```

## Scheduling Example Queued Events

46



Sponsored by Model Technology

Active Events

```
a=0
b=1
(RHS=1)
(RHS=0)
"a=0 b=1"
```

Inactive Events

```
"a=0 b=1"
```

Nonblocking  
Events

```
a<=1
b<=0
```

Monitor Events

```
"a=1 b=0"
"a=1 b=0"
```

Blocking assignment  
Blocking assignment  
Evaluate RHS of NBA  
Evaluate RHS of NBA  
\$display command  
#0 display command  
Update LHS of NBA  
Update LHS of NBA  
\$monitor command  
\$strobe command

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

  $monitor ...;
  $display ...;
  $strobe ...;
  #0 $display ...;

  #1 $monitor ...;
  $display ...;
  $strobe ...;
  $display ...;

  #1 $finish;
end
```

## 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

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

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!**

```

`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;
    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
  
```

**rst\_n asserted with blocking assignment**

**Might miss negedge rst\_n**

**Waiting for a negedge rst\_n**

## No Reset Race Condition

51

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

**Must assert rst\_n at the end of time 0  
No Race Condition!**

**Assert rst\_n with nonblocking assignment**

**Negedge rst\_n will be triggered**

**Waiting for a negedge rst\_n**

**Triggered negedge rst\_n**

```

`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

```

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

## Clock Race Condition

52

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

**clk assigned with blocking assignment**

**Waiting for a posedge clk**

**Missed posedge clk!**

```

`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 HIGH at time 0			
0ns:	q=x	d=1	clk=1
50ns:	q=x	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

## No Clock Race Condition

53

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

```

`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**

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

**Triggered posedge clk**

**Waiting for a posedge clk**

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

54



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

## \$finish Race Condition

55



```
MONITOR: cnt=1
STROBE : cnt=2
MONITOR: cnt=2
STROBE : cnt=3
MONITOR: cnt=3
STROBE : cnt=4
MONITOR: cnt=4
STROBE : cnt=5
MONITOR: cnt=5
STROBE : cnt=6
MONITOR: cnt=6
STROBE : cnt=7
MONITOR: cnt=7
```

Displayed values do not include cnt=8

\$strobe

\$monitor

\$finish command

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

```
`define cycle 10
`timescale 1ns / 1ns
module finish_print;
integer cnt;
reg clk;

initial begin
clk = 0;
forever #(`cycle/2) clk = ~clk;
end

initial begin
$monitor("MONITOR: cnt=%0d", cnt);
cnt = 1;
repeat (7) begin
@(posedge clk) cnt = cnt+1;
$strobe("STROBE : cnt=%0d", cnt);
end
$finish;

initial wait(cnt==8)
$display("DISPLAY: cnt=%0d", cnt);
endmodule
```

## No \$finish Race Condition

56



```
MONITOR: cnt=1
STROBE : cnt=2
MONITOR: cnt=2
STROBE : cnt=3
MONITOR: cnt=3
STROBE : cnt=4
MONITOR: cnt=4
STROBE : cnt=5
MONITOR: cnt=5
STROBE : cnt=6
MONITOR: cnt=6
STROBE : cnt=7
MONITOR: cnt=7
DISPLAY: cnt=8
STROBE : cnt=8
MONITOR: cnt=8
```

Displayed values DO include cnt=8

\$strobe

\$monitor

#1 \$finish command

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

```
`define cycle 10
`timescale 1ns / 1ns
module finish_print;
integer cnt;
reg clk;

initial begin
clk = 0;
forever #(`cycle/2) clk = ~clk;
end

initial begin
$monitor("MONITOR: cnt=%0d", cnt);
cnt = 1;
repeat (7) begin
@(posedge clk) cnt = cnt+1;
$strobe("STROBE : cnt=%0d", cnt);
end
#1 $finish;

initial wait(cnt==8)
$display("DISPLAY: cnt=%0d", cnt);
endmodule
```

## \$display and #0 Assignment Guidelines



57

- **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



58

- 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**  
A MENTOR GRAPHICS COMPANY



59

## A Couple of Important Verilog Tips

## File Naming Fundamental



60

Sponsored by **Model Technology**  
A MENTOR GRAPHICS COMPANY

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

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

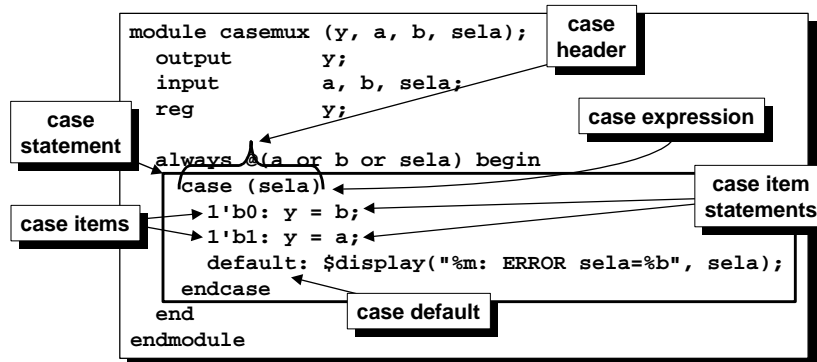
- Advantages
  - Easier to compile all necessary files using the -y library directory command line switch
  - Easier to build synthesis scripts with a single list of designs that also serve as a list of file names

## Case Statement Definitions

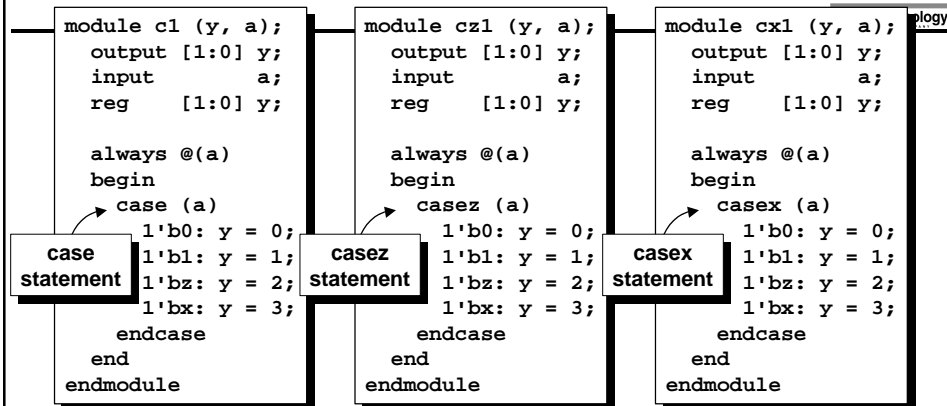


Sponsored by Model Technology

- A common set of terms for case statements



## Case Statement Types



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?

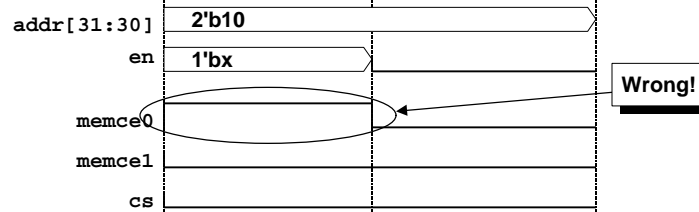
## Casex

```

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

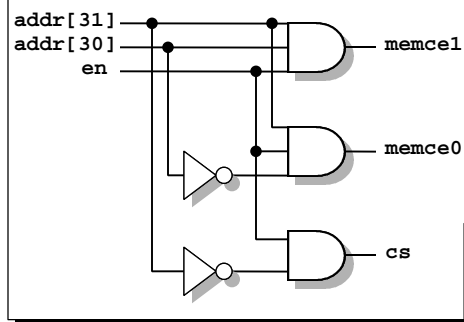
  always @(addr or en) begin
    {memce0, memce1, cs} = 3'b0;
    casex ({addr, en})
      3'b101: memce0 = 1'b1;
      3'b111: memce1 = 1'b1;
      3'b0?1: cs     = 1'b1;
    endcase
  end
endmodule

```

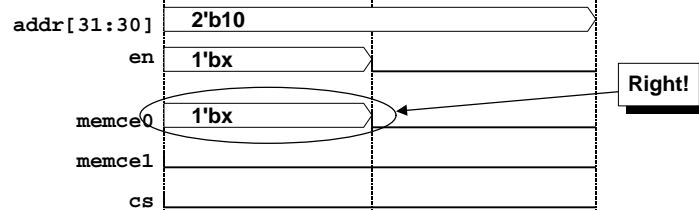


## Casex

### Synthesized result



Pre-synthesis  
simulation does not  
match post-  
synthesis simulation





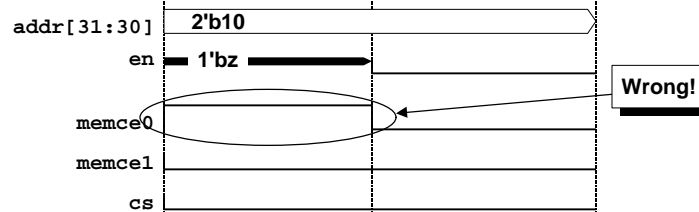
## Casez

```

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

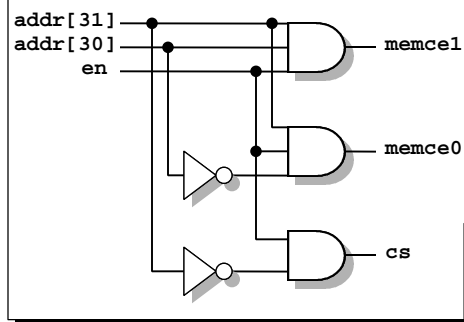
  always @(addr or en) begin
    {memce0, memce1, cs} = 3'b0;
    casez ({addr, en})
      3'b101: memce0 = 1'b1;
      3'b111: memce1 = 1'b1;
      3'b0?1: cs     = 1'b1;
    endcase
  end
endmodule

```

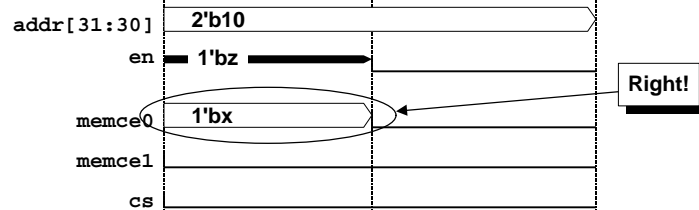


## Casez

## Synthesized result



Pre-synthesis  
simulation does not  
match post-  
synthesis simulation



## Casez & Casex Guidelines



Sponsored by Model Technology

- 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



Sponsored by Model Technology

- 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**  
A MENTOR GRAPHICS COMPANY



69

# State Machines

Abbreviated FSMs (Finite State Machines)

## State Machine Classification

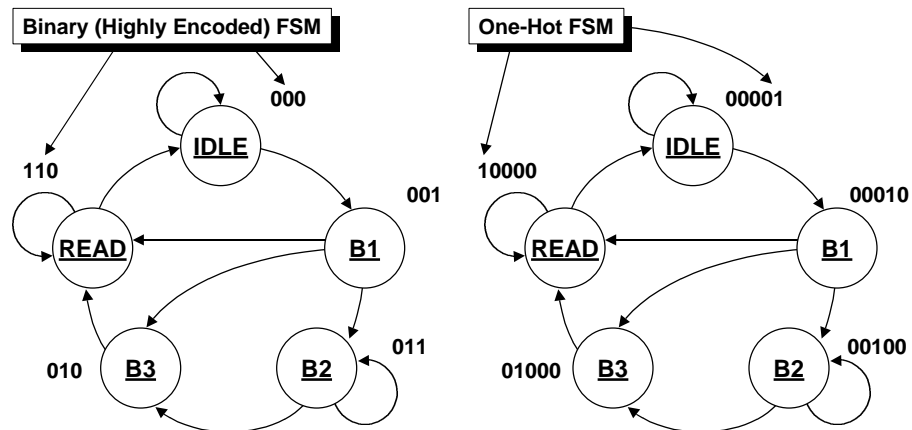


70

Sponsored by **Model Technology**  
A MENTOR GRAPHICS COMPANY

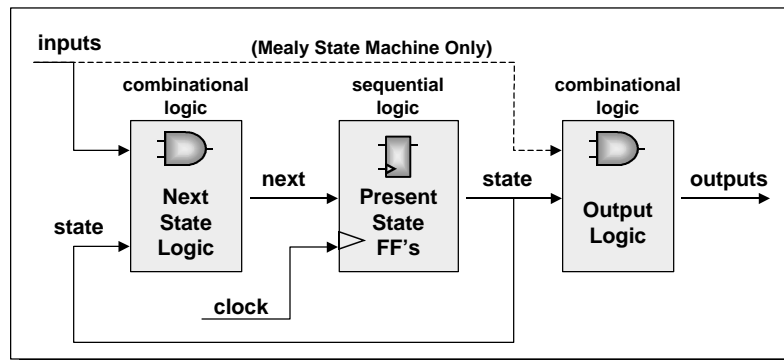
- Moore - outputs are only a function of the present state
- Mealy - one or more of the outputs are a function of the present state and one or more of the inputs
- For an “n”-state FSM (examples on next slide):
  - Binary (highly encoded) = Ceiling  $\log_2 n$  flip-flops  
5 states -> 3 FF's (fewest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

## Binary & One-Hot FSMs



## Primary State Machine Blocks

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



## State Machine Coding Styles



73

- 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



74

- 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

75



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

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

"Re-definition"  
warnings!

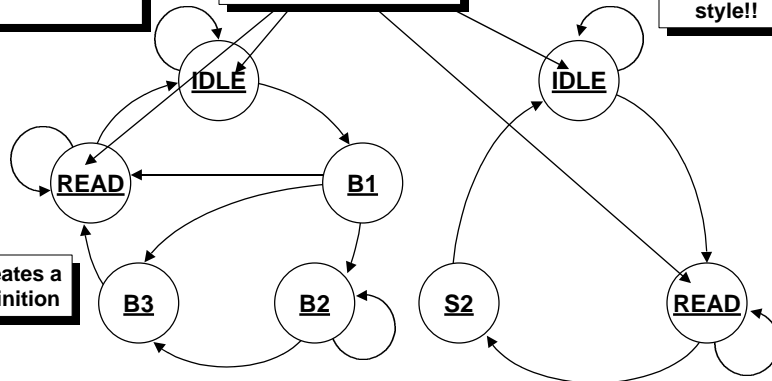
Multiple FSMs with  
the same state names

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

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

Bad coding  
style!!

`define creates a  
global definition



## State Definitions Using parameter

76



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

No state definition  
problems!

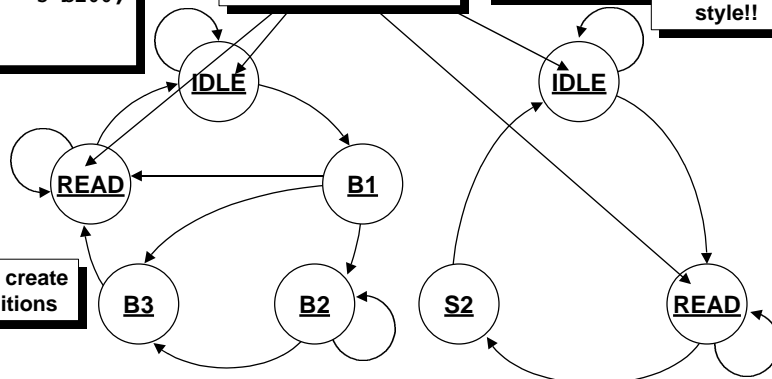
Multiple FSMs with  
the same state names

Sponsored by ModelTechnology

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

Good coding  
style!!

parameters create  
local definitions



## State Parameter Definitions



Sponsored by Model Technology

77

### Parameter definitions for binary encoding

```
parameter IDLE = 3'b000,  
          S1 = 3'b001,  
          S2 = 3'b010,  
          S3 = 3'b011,  
          ERROR = 3'b100;
```

### State register encoding

### Parameter definitions for verbose one-hot encoding

```
parameter IDLE = 5'b00001,  
          S1 = 5'b00010,  
          S2 = 5'b00100,  
          S3 = 5'b01000,  
          ERROR = 5'b10000;
```

### State register encoding

Inefficient way to  
code One-Hot FSMs

## Two-Always Block State Machines



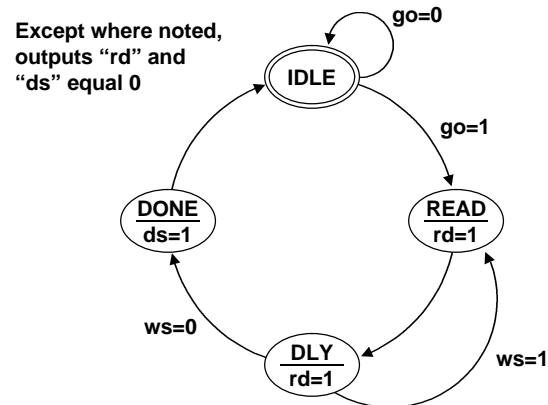
Sponsored by Model Technology

78

- (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



85

- 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.)



86

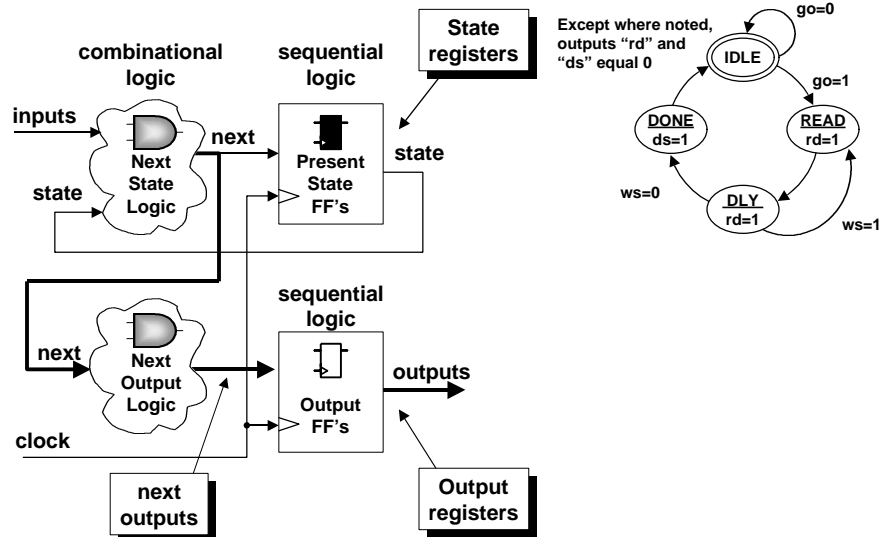
- Separate next state combinational block
  - Sensitivity list sensitive to state and all inputs
  - Default state assignment (next = 2'bx;)
  - Blocking assignments
- Making output assignments
  - Separate output continuous assignments
    - All conditions that affect an output are in one location and easily scanned
  - OR–
  - Place output assignments in the combinational always block
    - Input conditions have already been decoded

## Registering The "Next Outputs"

87



Sponsored by Model Technology



## Registered Outputs

(Add a third (sequential) always block)

88



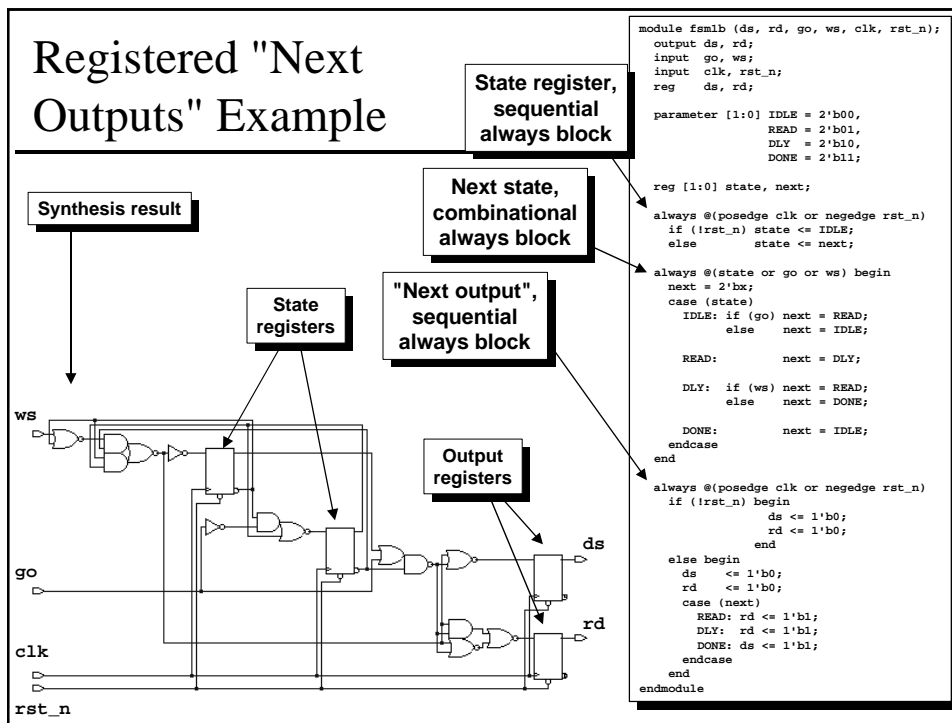
Sponsored by Model Technology

```
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



## State Machines - Mealy Outputs



90

- 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)



Sponsored by Model Technology

91

```
module smlhtfpa (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)



Sponsored by Model Technology

92

```
...
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



Sponsored by Model Technology

93

- 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

Sponsored by

**Model Technology**  
A MENTOR GRAPHICS COMPANY



94

## Verilog-2001 Behavioral and Synthesis Enhancements

## Verilog Enhancement Strategy



95

- 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



96

- 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!)**



# Structural Dual-Pipeline Model



Sponsored by Model Technology

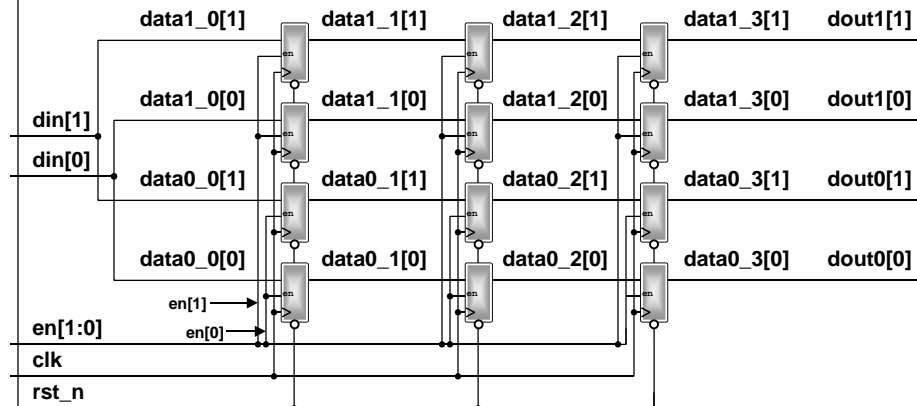
```

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;

```

2-bit wire-bus  
declarations



# Dual Pipeline Code



Sponsored by Model Technology

```

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

```

1-dimensional wire  
array declaration

2-bit assignments

Bit reference  
listed last

## Multi-Dimensional Verilog-2001 Enhancement #2

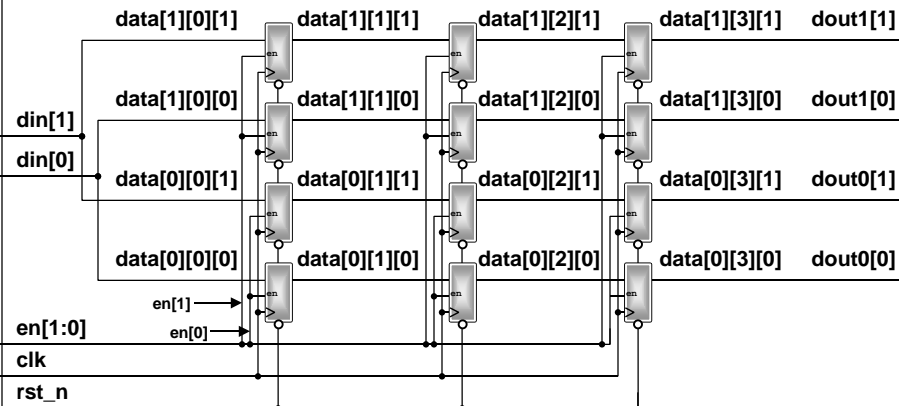
3-dimensional wire array declaration

Sunburst Design

Soon Synthesizable Soon!

```
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

3-dimensional wire array declaration

Sunburst Design

Sponsored by Model Technology

```
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



```

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



- 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

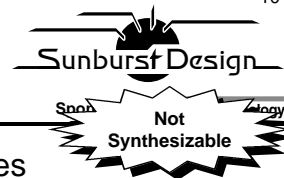
```
module ...
  generate if ((A_WIDTH < 8) || (B_WIDTH < 8))
    CLA_multiplier u1 (...);    // CLA multiplier
  else
    WALLACE_multiplier u1 (...); // Wallace-tree multiplier
  endgenerate
  ...
endmodule
```

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

integer MSB="0"

#### Verilog-2001 File Descriptor

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

integer MSB="1"

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



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

**New Verilog-2001 file I/O commands**

```

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

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

## Re-Entrant Tasks & Functions

Verilog-2001 Enhancement #4



Soon Synthesizable Soon!

```

module ...
    task writel;
    ...
    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

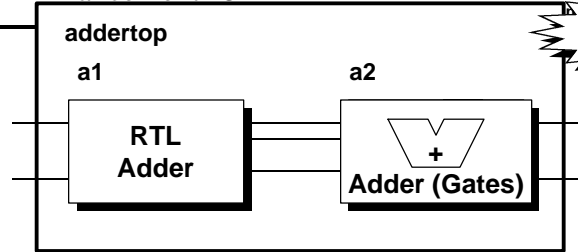
# Verilog Configuration Files

Verilog-2001 Enhancement #5

107



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

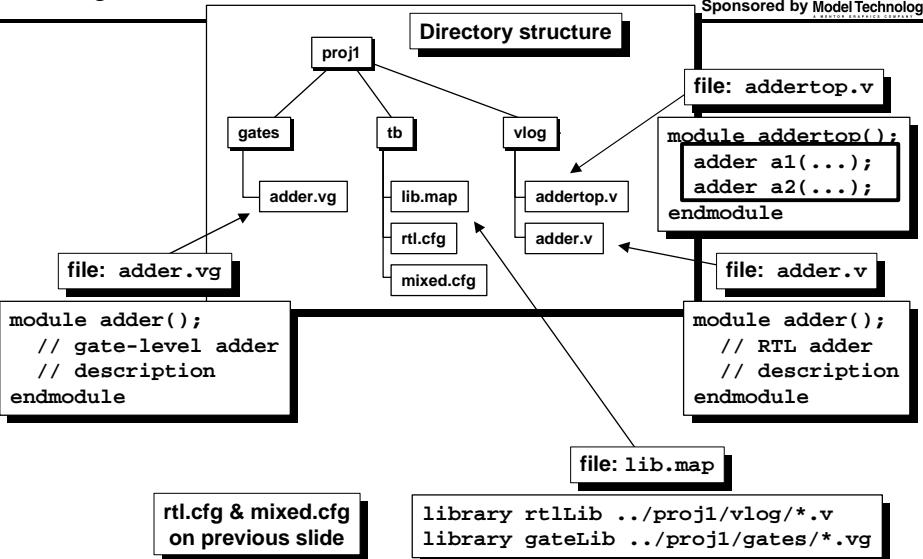
# Simulation Configurations

Verilog-2001 Enhancement #5

108



Sponsored by Model Technology



## ANSI-C Style Ports

Works with  
ModelSim 5.5!

Sunburst Design

109

Soon Synthesizable Soon!

```
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

( 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

110

Soon 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



Sponsored by Model Technology

111

```
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 ..."

```
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 ..."

Synthesized result!



# Incomplete Sensitivity List



Sponsored by Model Technology

112

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

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

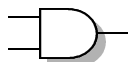
Pre-synthesis simulation matches post-synthesis simulation

```
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

113



Sponsored by Model Technology

```
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

114



Soon Synthesizable Soon!

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

Parameterized Verilog-2001  
memory (ram1) model

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

## 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
...
```

## Passing Parameters to Instances



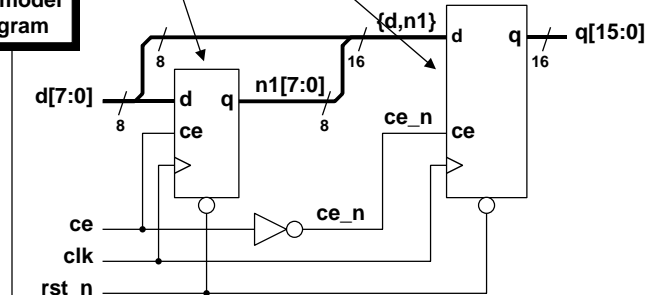
Parameterized  
regblk model

Two instantiated regblk instances  
with different sizes

demuxreg model  
block diagram

```
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
```



## defparam Redefinition

### Verilog-1995



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);
  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!!*

- 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."

```

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

```

## Positional Parameter Redefinition

### Verilog-1995



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 #( 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



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;

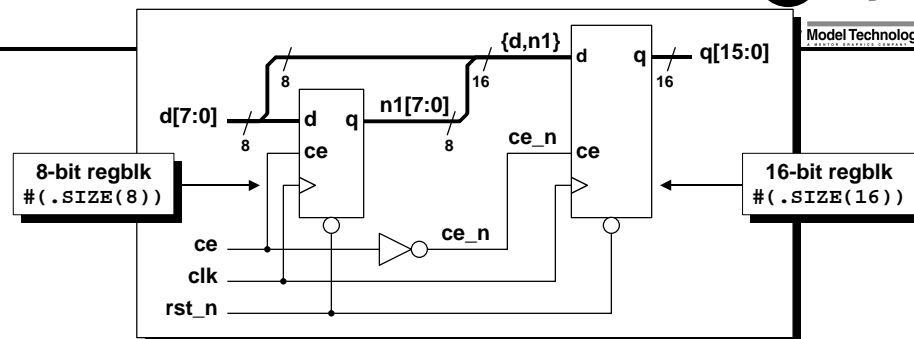
  always @(posedge clk or negedge rst_n)
    if (!rst_n) q <= 0;
    else if (ce)  q <= d;
endmodule

```

## Demuxreg Verilog-2001 Model



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



Sponsored by Model Technology

- 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

```

# Top-Level Module Instantiation

## Verilog Generate Method



Sponsored by Model Technology

- 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
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
```

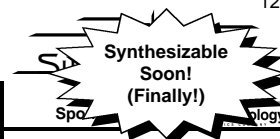
16 data BIDIR I/O pads

32 addr IBUF I/O pads

# Top-Level Module Instantiation

## Array of Instances

Works with  
ModelSim 5.5!



- 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
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
```

16 data BIDIR I/O pads

32 addr IBUF I/O pads

## V++ Enhancement Idea?

Verilog-2005 Enhancement? - What do you think?



Sponsored by Model Technology

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

@\* automatic port matching??

```
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
```

main\_blk u1 (@\*, .sig1(ctl1), .sig2(ctl2), .sig3(wr));

## Attributes



Sponsored by Model Technology

- 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

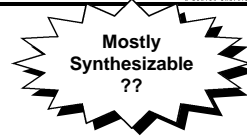
## Additional Verilog-2001 Enhancements

127



Sponsored by Model Technology

- 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?

128



Sponsored by Model Technology

### (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



Sponsored by Model Technology

- 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

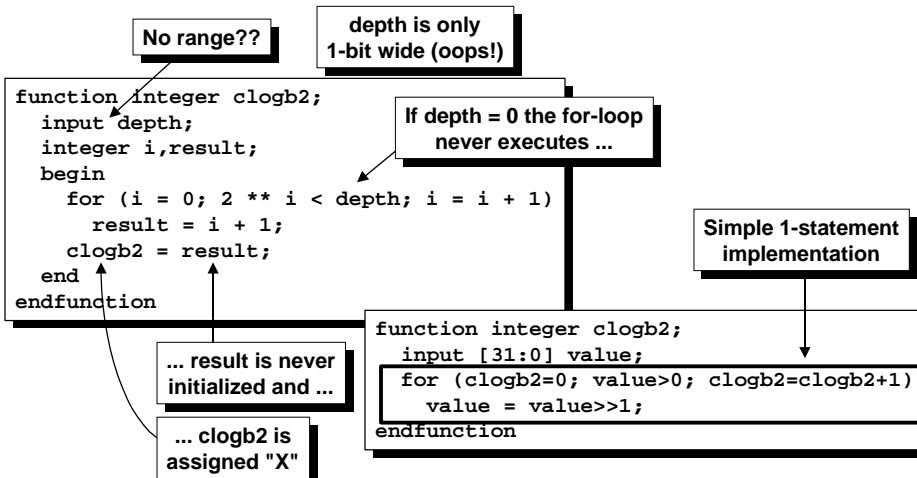
# Errata

## A Few Mistakes "Slipped" Into The LRM



Sponsored by Model Technology

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



## 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
```

No rol6 function necessary -  
use part-selects and concatenation

```
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
```

# Barrel Shifter

## Better 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
```

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

Sponsored by Model Technology

Sunburst Design

```
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'bxx;
        endcase
    end
endmodule
```

{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}}}

133

# 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
```

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

134

# 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

```
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
```

Data type included in  
the port declaration

Local  
parameter

ANSI-C  
style 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
```

ANSI-C  
style parameters  
and ports

Comma-separated  
sensitivity list

Constant  
function

Parameter redefinition using named parameters

## Registered Barrel Shifter

Verilog-2001 Version



137

Sponsored by Model Technology

- 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



138

Sponsored by Model Technology

- 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

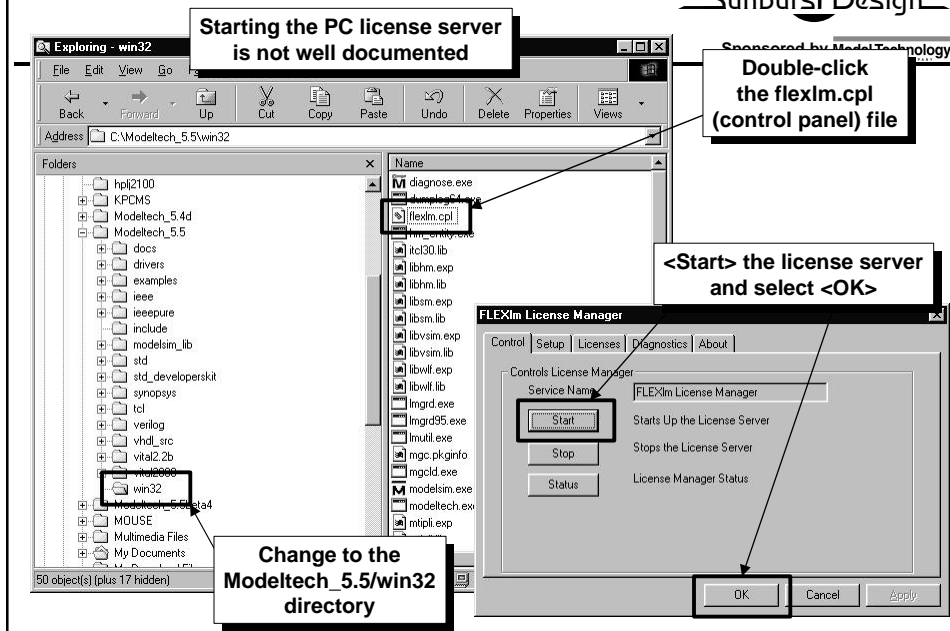
## The ModelSim PC License Server

**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**



The screenshot shows a Windows Explorer window with the address bar set to 'C:\Modeltech\_5.5\win32'. The file list includes 'flexlm.cpl', which is highlighted. A callout box points to this file with the text 'Double-click the flexlm.cpl (control panel) file'. Another callout box points to the 'win32' folder with the text 'Change to the Modeltech\_5.5/win32 directory'. The FLEXlm License Manager dialog box is open, showing the 'Control' tab. The 'Service Name' is 'FLEXlm License Manager'. The 'Start' button is highlighted, with a callout box pointing to it that says '<Start> the license server and select <OK>'. The 'OK' button at the bottom of the dialog is also highlighted.

## Misleading Port Warning



141

Sponsored by Model Technology

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

  myand2 ul (.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/ul
...
```

## Trick: Running UNIX-Version of ModelSim without the GUI



142

Sponsored by Model Technology

- 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

143



Sponsored by Model Technology

- 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)

144



Sponsored by Model Technology

```
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 ul (q, d, clk, rst_n);
  `endif
endmodule
```

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

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

**D-flip-flop User Defined Primitive model (with a 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
    *   ?   ?   : ? : - ;
  endtable
endprimitive
```



## Trick: Comparing Waveforms

- ModelSim 5.5 has a very useful waveform compare feature

**RTL simulation**

**Quit the simulation to create a valid vsim.wlf file**

**wave - default**

Signal	Value
/tb_dff/ui/q	1
/tb_dff/ui/d	S11
/tb_dff/ui/clk	S10
/tb_dff/ui/res_n	S11

0 ns to 148 ns

## Comparing Waveforms

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

**Exploring - UDP**

Address: C:\Mtc\ModelTech\Seminar2001\UDP

Change this file name (example: rtl.wlf)

**wave - default**

Computer

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

## Comparing Waveforms

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

After simulation, select **Compare**

UDP simulation

ModelSim SE 5.5

Compile

Start vsim

Run the simulation

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

147

## Comparing Waveforms

- Use the comparison wizard!!

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

Comparison Wizard

Select the Comparison Method of choice

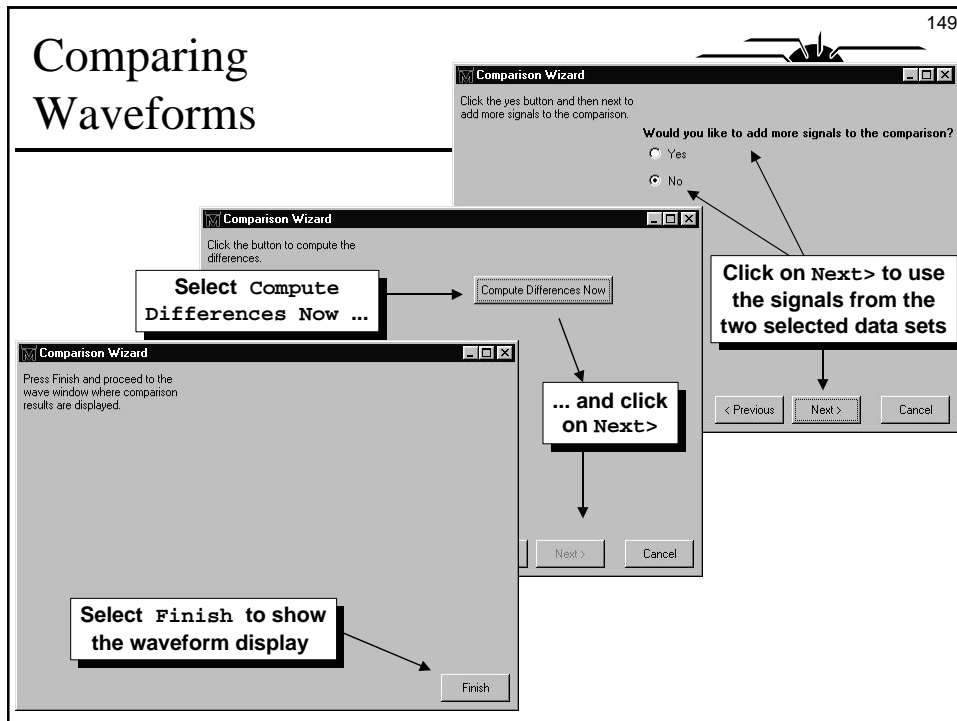
Select a second .wlf file (example: udpbug.wlf) -OR- Use the current simulation data set

At the end of each step, click on Next >

148

## Comparing Waveforms

149

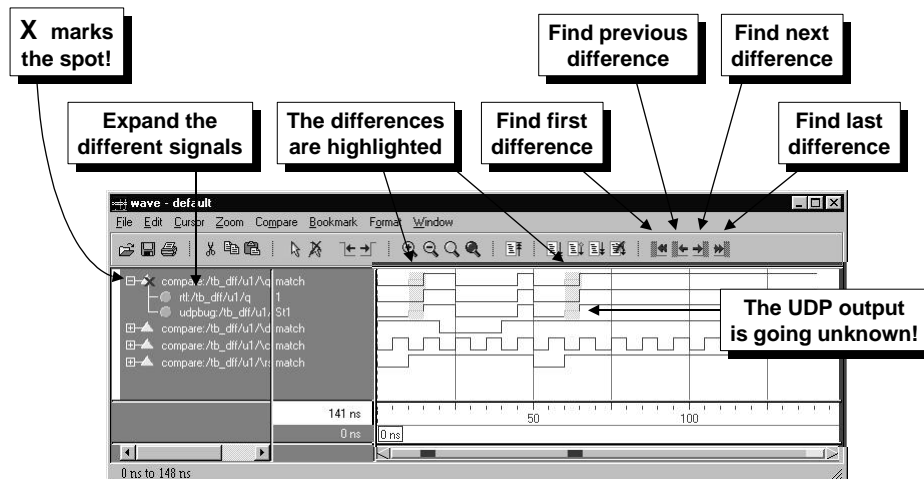


## Comparing Waveforms

150



- "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;

    `ifndef RTL
        reg q;
        always @(posedge clk or negedge rst_n)
            if (!rst_n) q <= 1'b0;
            else q <= d;
    `endif
    `ifndef 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](http://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