Verilog Nonblocking Assignments with Delays - Myths & Mysteries

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

- IEEE 1364 reference model & event queue
- Review 8 Guidelines to avoid "death by Verilog!"
- 0-delay models - Nonblocking assignments happen first
- Inertial & transport delays
- Delay line modeling with transport delays
- Benchmark VCS simulations with and without #1 delays
- VCS switches: +nbaopt and +rad
- Multiple common clocks - are there race conditions?
- Mixing blocking & nonblocking assignments
- Mixed RTL & gate simulations
- Miscellaneous SDF notes & testbench tricks
- Flawed guidelines - better guidelines - Conclusions
while (there are events) {
  if (no active events) {
    if (there are inactive events) {
      activate all inactive events;
    } else if (there are nonblocking assign update events) {
      activate all nonblocking assign update events;
    } else if (there are monitor events) {
      activate all monitor events;
    } else {
      advance T to the next event time;
      activate all inactive events for time T;
    }
  } else if (there are events) {
    E = any active event;
    if (E is an update event) {
      update the modified object;
      add evaluation events for sensitive processes to event queue;
    } else { /* shall be an evaluation event */
      evaluate the process;
      add update events to the event queue;
    }
  }
}
while (there are events) {
    if (there are active events) {
        E = any active event;
        if (E is an update event) {
            update the modified object;
            add evaluation events for sensitive processes to event queue;
        }
        else { // this is an evaluation event, so ... 
            evaluate the process;
            add update events to the event queue;
        }
    }
    else if (there are nonblocking update events) {
        activate all nonblocking update events;
    }
    execute $monitor and $strobe commands before advancing time
}
else {
    advance T to the next event time;
    activate all inactive events for time T;
}
... update LHS of nonblocking assignments

First: set T=0 / set nets=HiZ / set variables=X / activate always blocks / activate initial blocks
Execute all active events
schedule newly triggered events
... update LHS of nonblocking assignments
Advance to the next simulation time and start over
IEEE1364-1995 Verilog
Stratified Event Queue

Active Events
- Blocking assignments
  - Evaluate RHS of nonblocking assignments
- Continuous assignments
  - $display command execution
- $monitor command execution
  - $strobe command execution
  - Other specific PLI commands

Inactive Events
- #0 blocking assignments

Nonblocking Events
- Update LHS of nonblocking assignments

Monitor Events

These events may be scheduled in any order

*Guideline #8: do not use #0 delays

* Guidelines on slide 7
IEEE1364-1995 Verilog Stratified Event Queue

Active Events

Nonblocking Events

Can trigger nested events in the same time step

Blocking assignments
Evaluate RHS of NBAs
Continuous assignments

These events may be scheduled in any order

Update LHS of nonblocking assignments

Active Events

Nonblocking Events

Blocking assignments
Evaluate RHS of NBAs
Continuous assignments

These events may be scheduled in any order

Update LHS of nonblocking assignments

$monitor command execution
$strobe command execution
8 Guidelines to avoid Coding Styles that Kill!

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

  Guideline #1: Sequential logic - use **nonblocking assignments**

  Guideline #2: Latches - use **nonblocking assignments**

  Guideline #3: Combinational logic in an always block - use **blocking assignments**

  Guideline #4: Mixed sequential and combinational logic in the same always block - use **nonblocking assignments**

  Guideline #5: Do not mix blocking and nonblocking assignments in the same always block

  Guideline #6: Do not make assignments to the same variable from more than one always block

  Guideline #7: Use $strobe to display values that have been assigned using nonblocking assignments

  Guideline #8: Do not make #0 procedural assignments

Follow these guidelines and remove 90-100% of the Verilog race conditions.
Are There Exceptions to the Guidelines?

- **Probably!**

  - How to judge valid exceptions:
    - Does the exception make the simulation significantly faster?
    - Does the exception make the code more understandable?
    - Does the exception make the coding effort much easier?

- Faster! ... More understandable! ... Easier!

- If not, the exception is probably not a good exception
For 0-Delay RTL Models
Nonblocking Assignments finish first!

• ??? - The Verilog event queue schedules blocking assignments before nonblocking assignments
• Using clk-edge-based simulation techniques ...

All combinational outputs settle out immediately after the posedge clk
module sblk1 (  
    output reg q2,  
    input a, b, clk, rst_n);  
reg q1, d1, d2;

always @(a or b or q1) begin  
d1 = a & b;  
d2 = d1 | q1;  
end

always @(posedge clk or negedge rst_n)  
if (!rst_n) begin  
q2 <= 0;  
q1 <= 0;  
end  
else begin  
q2 <= d2;  
q1 <= d1;  
end
endmodule
sblk1 Input Stimulus & Input Combinational Logic Timing

External combinational inputs typically change on negedge clks

Diagram of a combinational logic circuit with inputs a, b, d1, d2, q1, q2, clk, and reset (rst_n) with timing waveforms for each input and output.
sblk1 Input Stimulus & Input
Combinational Logic Event Scheduling

- **Active Events**
  - Blocking assignment (clk = ~ clk; // testbench negedge clk)
  - Triggers testbench stimulus commands @negedge clk
  - Triggers *combinational* inputs on the device under test

- **Nonblocking Events**
  - Empty (no nonblocking assignments to update)

- **Monitor Events**
  - `$monitor` command execution (if any)
  - `$strobe` command execution (if any)

**Advance to next event**
(should be a posedge clk blocking assignment)

(Booking assignment in the testbench clock oscillator)
sblk1 Sequential (Clocked) Logic Timing

Sequential logic outputs change on posedge clks
Internal combinational outputs change on posedge clks (after sequential logic)
sblk1 Sequential & Combinational Logic Event Scheduling

Active Events

Blocking assignment (clk = ~ clk; // testbench posedge clk)
Triggers evaluation of RHS of sequential logic NBAs

Nonblocking Events

Update LHS of sequential logic nonblocking assignments

Active Events

Activate and execute NBAs events
Triggers combinational logic blocking assignments (after the NBAs in the same time step)

Nonblocking Events

Empty (no additional nonblocking assignments to update)

Monitor Events

$monitor command execution (if any)
$strobe command execution (if any)

Advance to next event

negedge clk - triggers stimulus input events
posedge clk - triggers sequential logic events
• Command line switches for gate-level simulations
  – Reject pulses shorter than \( x\% \) of the propagation delay
    \[ +\text{pulse}_r/x \]
  – Display unknowns (errors) for pulses greater than \( x\% \) of the propagation delay but shorter than \( y\% \) of the propagation delay
    \[ +\text{pulse}_e/y \]
  – Enable transport delays for gate-level simulation
    \[ +\text{transport\_path\_delays} \]
Simple Test Buffer with 5ns propagation delay

Simple delay buffer model (delaybuf.v)

```
`timescale 1ns/1ns
module delaybuf
  (output y, input a);
  buf u1 (y, a);
  specify
    (a*>y) = 5;
endspecify
endmodule
```

Verilog buffer primitive

5ns specify block delay from a-to-y

```
module tb;
  reg     a;
  integer i;
  delaybuf i1 (.y(y), .a(a));
  initial begin
    a=0;
    #10 a=~a;
    for (i=1;i<7;i=i+1)
      #(i) a=~a;
    #20 $finish;
  end
endmodule
```

Simple testbench (tb.v)

Stimulus instance

5ns

a ------- y
Inertial & Transport Delays
commands & displays

vcs -RI +v2k tb.v delaybuf.v +pulse_r/100 +pulse_e/100

Pure inertial delays

vcs -RI +v2k tb.v delaybuf.v +pulse_r/0 +pulse_e/0 +transport_path_delays

Pure transport delays
Error & Mixed Delays
commands & displays

```
vcs -RI +v2k tb.v delaybuf.v +pulse_r/0 +pulse_e/100
```

**Pure unknown (error) delays**

```
vcs -RI +v2k tb.v delaybuf.v +pulse_r/40 +pulse_e/80
```

**Mixed delays (r/40 & e/80)**

Pulses shorter than 40% of 5ns are filtered out
Pulses between 40% & 80% of 5ns are passed as X's
Pulses greater than 80% of 5ns are passed
Delay Line Model
Transport Delays

Both models use Verilog-2001 enhanced coding style

Parameterized delay line model with two output taps

```verilog
`timescale 1ns / 1ns
module DL2
    #(parameter TAP1 = 25, TAP2 = 40)
    (output reg y1, y2,
     input in);

    always @(in) begin
        y1 <= #TAP1 in;
        y2 <= #TAP2 in;
    end

endmodule
```

Delay line model with two output taps

```verilog
`timescale 1ns / 1ns
module DL2
    (output reg y1, y2,
     input in);

    always @(in) begin
        y1 <= #25 in;
        y2 <= #40 in;
    end

endmodule
```

RHS nonblocking delays are transport delays

These events are scheduled into future nonblocking assign update event queues

NOTE: Synthesis tools ignore delays

Cannot synthesize delay lines
Benchmark Circuit #1
20K bits of Sequential Logic

20 x 1000-bit registers
Benchmark Circuit #2
20K bits Sequential / 40K bits Combinational

20 x 1000-bit registers with inverted-inputs and inverted-outputs
Benchmark Models
(with inverters and without inverters)

module dff (q, d, clk, rst_n);
  parameter SIZE=100;
  output [SIZE-1:0] q;
  input [SIZE-1:0] d;
  input clk, rst_n;
  reg [SIZE-1:0] q;

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

DFF models with no inverters

DFF models with inverters

module dffi (q, d, clk, rst_n);
  parameter SIZE=100;
  output [SIZE-1:0] q;
  input [SIZE-1:0] d;
  input clk, rst_n;
  reg [SIZE-1:0] qq;
  wire [SIZE-1:0] dd;

  assign q = ~qq;
  assign dd = ~d;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) qq <= 0;
    else        qq <= dd;
endmodule

Invert inputs to flip-flops and invert outputs from flip-flops
Benchmark RTL Code
With and Without Delays

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

always @(posedge clk or negedge rst_n)
    if (!rst_n) q <= #1 0;
    else q <= #1 d;

always @(posedge clk or negedge rst_n)
    if (!rst_n) q = #1 0;
    else q = #1 d;

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

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

#1 Nonblocking assignments with no delays

#2 Nonblocking assignments with #1 delays

#3 Blocking assignments with #1 delays (BAD)

#4 Nonblocking assignments with macro-added #0 delays

#5 Nonblocking assignments with macro-added no delays
module dffpipe (q, d, clk, rst_n);
    parameter SIZE=1000;
    output [SIZE-1:0] q;
    input  [SIZE-1:0] d;
    input             clk,  rst_n;

    wire [SIZE-1:0] qq1,  qq2,  qq3,  qq4,  qq5,  qq6,  qq7,  qq8,  qq9;
    wire [SIZE-1:0] qq10, qq11, qq12, qq13, qq14, qq15, qq16, qq17, qq18, qq19;

    `DFF  #(SIZE)  u1 (.q( qq1), .d(   d), .clk(clk), . rst_n( rst_n));
    `DFF  #(SIZE)  u2 (.q( qq2), .d( qq1), .clk(clk), .rst_n( rst_n));
    `DFF  #(SIZE)  u3 (.q( qq3), .d( qq2), .clk(clk), .rst_n( rst_n));
    `DFF  #(SIZE)  u4 (.q( qq4), .d( qq3), .clk(clk), .rst_n( rst_n));

    `DFF  #(SIZE)  u17 (.q(qq17), .d(qq16), .clk(clk), .rst_n( rst_n));
    `DFF  #(SIZE)  u18 (.q(qq18), .d(qq17), .clk(clk), .rst_n( rst_n));
    `DFF  #(SIZE)  u19 (.q(qq19), .d(qq18), .clk(clk), .rst_n( rst_n));
    `DFF  #(SIZE)  u20 (.q(   q), .d(qq19), .clk(clk), .rst_n( rst_n));
endmodule

Command line options to select different DFF models with or without delays

+define+DFF="dff"
+define+DFF="dff1"
+define+DFF="dff1b"
+define+DFF="dff0"
+define+DFF="dff__"

Large Pipeline Benchmark Circuit
(top-level model)

20 registers
1000 bits each
# Benchmark Results - Circuit #1

Pipeline with no inverters

<table>
<thead>
<tr>
<th></th>
<th>CPU Time (seconds)</th>
<th>Speed compared to no-delay model</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFF pipeline (no inverters)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>No delays</td>
<td>292.920</td>
<td>Baseline no-delay model</td>
</tr>
<tr>
<td>Nonblocking #1 delays ( &lt;= #1 )</td>
<td>376.460</td>
<td>29% slower</td>
</tr>
<tr>
<td>Blocking #1 delays ( = #1 NOT RECOMMENDED)</td>
<td>358.240</td>
<td>22% slower</td>
</tr>
<tr>
<td>Nonblocking #0 delays ( &lt;= <code>D and </code>define D #0 )</td>
<td>307.630</td>
<td>5% slower</td>
</tr>
<tr>
<td>Nonblocking blank delays ( &lt;= <code>D and </code>define D &lt;no_value&gt; )</td>
<td>292.880</td>
<td>~same speed</td>
</tr>
</tbody>
</table>

IBM ThinkPad T21, Pentium III-850MHz, 384MB RAM, Redhat Linux 6.2
VCS Version 6.2 - Simulation ended at Time: 800002150 ns
### Benchmark Results - Circuit #1

Pipeline with no inverters

<table>
<thead>
<tr>
<th>Description</th>
<th>CPU Time (seconds)</th>
<th>Speed compared to no-delay model</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFF pipeline (no inverters)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>No delays</td>
<td>438.090</td>
<td>Baseline no-delay model</td>
</tr>
<tr>
<td>Nonblocking #1 delays ((\leq #1))</td>
<td>839.270</td>
<td>92% slower</td>
</tr>
<tr>
<td>Blocking #1 delays ((= #1) NOT RECOMMENDED)</td>
<td>548.110</td>
<td>25% slower</td>
</tr>
<tr>
<td>Nonblocking #0 delays ((\leq \text{<code>D and </code>define D #0}))</td>
<td>447.770</td>
<td>2% slower</td>
</tr>
<tr>
<td>Nonblocking blank delays ((\leq \text{<code>D and </code>define D &lt;no_value&gt;}))</td>
<td>437.960</td>
<td>~same speed</td>
</tr>
</tbody>
</table>

SUN Ultra 80, UltraSPARC-II 450MHz, 1GB RAM, Solaris 8
VCS Version 6.2 - Simulation ended at Time: 800002150 ns

SUN Workstation
## Benchmark Results - Circuit #2

### Pipeline with inverters

<table>
<thead>
<tr>
<th>Description</th>
<th>CPU Time (seconds)</th>
<th>Speed compared to no-delay model</th>
</tr>
</thead>
<tbody>
<tr>
<td>No delays</td>
<td>390.140</td>
<td>Baseline no-delay model</td>
</tr>
<tr>
<td>Nonblocking #1 delays</td>
<td>462.230</td>
<td>18% slower</td>
</tr>
<tr>
<td>( &lt;= #1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Blocking #1 delays</td>
<td>458.750</td>
<td>18% slower</td>
</tr>
<tr>
<td>( = #1 NOT RECOMMENDED)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nonblocking #0 delays</td>
<td>390.320</td>
<td>~same speed</td>
</tr>
<tr>
<td>( &lt;= <code>D and </code>define D #0 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nonblocking blank delays</td>
<td>390.630</td>
<td>~same speed</td>
</tr>
<tr>
<td>( &lt;= <code>D and </code>define D &lt;no_value&gt; )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**IBM ThinkPad T21, Pentium III-850MHz, 384MB RAM, Redhat Linux 6.2**

**VCS Version 6.2** - Simulation ended at Time: 800002150 ns

**Linux Laptop**
### Benchmark Results - Circuit #2
Pipeline with inverters

<table>
<thead>
<tr>
<th>Configuration</th>
<th>CPU Time (seconds)</th>
<th>Speed compared to no-delay model</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFF pipeline with inverters</td>
<td></td>
<td></td>
</tr>
<tr>
<td>No delays</td>
<td>668.170</td>
<td>Baseline no-delay model</td>
</tr>
<tr>
<td>Nonblocking #1 delays (&lt;= #1)</td>
<td>1,112.130</td>
<td>66% slower</td>
</tr>
<tr>
<td>Blocking #1 delays (= #1 NOT RECOMMENDED)</td>
<td>777.440</td>
<td>16% slower</td>
</tr>
<tr>
<td>Nonblocking #0 delays (&lt;= <code>D and </code>define D #0)</td>
<td>744.160</td>
<td>11% slower</td>
</tr>
<tr>
<td>Nonblocking blank delays (&lt;= <code>D and </code>define D &lt;no_value&gt; )</td>
<td>673.95</td>
<td>1% slower</td>
</tr>
</tbody>
</table>

SUN Ultra 80, UltraSPARC-II 450MHz, 1GB RAM, Solaris 8
VCS Version 6.2 - Simulation ended at Time: 800002150 ns

SUN Workstation
Command Line Switches

+\texttt{nbaopt} and +\texttt{rad}

- VCS command line switch to remove delays from the RHS of nonblocking assignments

  \texttt{+nbaopt} can be used to remove #1 delays

- VCS command line switch to improve overall simulation performance

  \texttt{+rad} is actually a family of optimizations that will make improvements to non-timing designs

  \texttt{+rad} does not affect delay scheduling

  \texttt{+rad} is not just for cycle-based simulations

  Speeds up logic and event propagation

Synopsys reports some designs achieve large speedups with +\texttt{rad} (typically the uglier the code, the larger the speedup)
### Benchmark Results - Circuit #1
With `+nbaopt` Command Switch

<table>
<thead>
<tr>
<th>System</th>
<th>DFF pipeline (no inverters)</th>
<th>CPU Time (seconds)</th>
<th>Speed compared to no-delay model</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IBM ThinkPad T21</strong></td>
<td>No delays</td>
<td>293.770</td>
<td>Baseline no-delay model</td>
</tr>
<tr>
<td><strong>SUN Ultra 80</strong></td>
<td>Nonblocking #1 delays</td>
<td>311.070</td>
<td>6% slower</td>
</tr>
<tr>
<td><strong>Linux Laptop</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VCS Version 6.2 - including the +nbaopt command switch</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>SUN Workstation</strong></td>
<td>No delays</td>
<td>439.000</td>
<td>Baseline no-delay model</td>
</tr>
<tr>
<td><strong>IBM ThinkPad T21</strong></td>
<td>Nonblocking #1 delays</td>
<td>448.630</td>
<td>2% slower</td>
</tr>
<tr>
<td><strong>SUN Ultra 80</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VCS Version 6.2 - including the +nbaopt command switch</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Benchmark Results - Circuit #1

**With `+rad` Command Switch**

<table>
<thead>
<tr>
<th>Configuration</th>
<th>CPU Time (seconds)</th>
<th>Speed compared to no-delay model</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFF pipeline (no inverters)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>No delays</td>
<td>233.540</td>
<td>Baseline no-delay model</td>
</tr>
<tr>
<td>Nonblocking #1 delays (<code>&lt;= #1</code>)</td>
<td>293.250</td>
<td>26% slower</td>
</tr>
<tr>
<td>Blocking #1 delays (<code>= #1</code> NOT RECOMMENDED)</td>
<td>289.940</td>
<td>24% slower</td>
</tr>
<tr>
<td>Nonblocking #0 delays (<code>&lt;= </code>D and <code>define D #0</code>)</td>
<td>229.290</td>
<td>2% faster</td>
</tr>
<tr>
<td>Nonblocking blank delays (<code>&lt;= </code>D and <code>define D &lt;no_value&gt;</code>)</td>
<td>233.100</td>
<td>~same speed</td>
</tr>
</tbody>
</table>

**IBM ThinkPad T21, Pentium III-850MHz, 384MB RAM, Redhat Linux 6.2**

**VCS Version 6.2 - (using `+rad` switch)**

**Linux Laptop**
## Benchmark Results - Circuit #2

**With `+rad` Command Switch**

<table>
<thead>
<tr>
<th>DFF pipeline with inverters</th>
<th>CPU Time (seconds)</th>
<th>Speed compared to no-delay model</th>
</tr>
</thead>
<tbody>
<tr>
<td>No delays</td>
<td>233.710</td>
<td>Baseline no-delay model</td>
</tr>
<tr>
<td>Nonblocking #1 delays</td>
<td>294.480</td>
<td>25% slower</td>
</tr>
<tr>
<td>( &lt;= #1 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Blocking #1 delays</td>
<td>288.910</td>
<td>23% slower</td>
</tr>
<tr>
<td>( = #1 NOT RECOMMENDED)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nonblocking #0 delays</td>
<td>228.410</td>
<td>3% faster</td>
</tr>
<tr>
<td>( &lt;= <code>D and </code>define D #0 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nonblocking blank delays</td>
<td>234.510</td>
<td>2% faster</td>
</tr>
<tr>
<td>( &lt;= <code>D and </code>define D &lt;no_value&gt; )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**IBM ThinkPad T21, Pentium III-850MHz, 384MB RAM, Redhat Linux 6.2**

**VCS Version 6.2 - (using `+rad` switch)**

**Linux Laptop**
module blk2_2 (  
    output reg q2,  
    input a, b,  
    input clk1a, clk1b, rst_n);  
reg q1;  
wire d1 = a & b;  
wire d2 = q1 | d1;  
always @(posedge clk1a or negedge rst_n)  
    if (!rst_n) q1 <= 0;  
    else q1 <= d1;  
always @(posedge clk1b or negedge rst_n)  
    if (!rst_n) q2 <= 0;  
    else q2 <= d2;  
endmodule
Multiple Common Clocks

If there is no skew between `clk1a` and `clk1b` and ...

```verilog
always @(clk1a)
    clk1b <= clk1a;
```

No race conditions!

Sequential logic outputs change on posedge clks

```
<table>
<thead>
<tr>
<th></th>
<th>rst_n</th>
<th>clk1a</th>
<th>clk1b</th>
<th>a</th>
<th>b</th>
<th>d1</th>
<th>q1</th>
<th>d2</th>
<th>q2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

![Waveform diagram]
Do Not Mix Assignments!
(This guideline is often challenged)

Guideline #5: Do not mix blocking and nonblocking assignments in the same always block

- This is a VHDL-like coding style
- No simulation advantage in Verilog

- Reasons to avoid this coding style
  - Understanding event scheduling can be confusing
  - Mis-ordering statements -or- multiple NBAs will cause problems
  - Inputs, outputs and clocks all change simultaneously in a waves display
module blk1a (  
    output reg q,  
    output y,  
    input a, b, c,  
    input clk, rst_n);  

    always @(posedge clk or negedge rst_n) begin: logic  
        reg d;  
        if (!rst_n) q <= 0;  
        else begin  
            d = a & b;  
            q <= d;  
        end  
    end  

    assign y = q & c;  
endmodule
Mixed Assignment Example #1
(synthesis result)

Synthesizes okay
Yucky Waveform Display!

The combinational d-signal does not update when the a and b inputs go high.

... d input changes
... the q output changes

at the same time !!
module blk1a (
    output reg q,
    output     y,
    input      a, b, c,
    input      clk, rst_n);

always @(posedge clk or negedge rst_n) begin: logic
    reg d;
    if (!rst_n) q <= 0;
    else begin
        d = a & b;
        q <= d;
        d = 1'bx;
    end
end

assign y = q & c;
endmodule

Named block to permit local declarations
Combinational intermediate signal
Extra X-assignment to avoid waveform confusion (???)
begin: logic
  reg d;
  if (!rst_n) q <= 0;
  else begin
    d = a & b;
    q <= d;
    // d = 1'bx;
  end
end

Is This Really Much Better??
module blk2a (
    output reg q, q2,
    output     y,
    input      a, b, c,
    input      clk, rst_n);
    reg        d;
always @(posedge clk or negedge rst_n)
    if (!rst_n) q <= 0;
    else begin
        d = a & b;
        q <= d;
    end
assign y = q & c;
always @(d) q2 = d;
endmodule
Mixed Assignment Example #2
(synthesis result)

Oops! \( q_2 \) is now a registered output
• Are there any problems with mixed RTL and gate-level simulations?
always @ (posedge clk or negedge rst_n)
  if (!rst_n) ...; // reset regs
  else begin
    a1a <= a1;
    a2a <= a2;
    a1b <= ...;
    a2b <= ...;
    b1 <= #1 ...;
    b2 <= #1 ...;
  end

Guideline: Add a #1 (or more) delay to RTL statements that drive gate-level models

Tsetup = 1.3ns   Thold = 0.6ns

mod2.vg
(gates model)

mod1.v
(RTL model)

mod3.v
(RTL model)
Multiple clock drivers
(no nonblocking assignments and no skew)

```verilog
mod1.v

a1a <= a1;
a2a <= a2;
a1b <= ...;
a2b <= ...;
b1 <= ...;
b2 <= ...

mod2.v

b1a <= b1;
b2a <= b2;
b1b <= ...;
b2b <= ...;
c1 <= ...;
c2 <= ...

mod3.v

c1a <= c1;
c2a <= c2;
c1b <= ...;
c2b <= ...;
d1 <= ...;
d2 <= ...
```

No simulation problems
Instantiated clock driver (PLL) with skewed clock outputs

Must add delays to RTL outputs
Gate-Level Simulations
(With Instantiated PLL Clock Source)

Instantiated clock driver (PLL) with skewed clock outputs

Gate-level models have intrinsic delays (no problems)
Problem -
Vendors That Use Blocking Assignments

• What happens when an incorrectly-coded vendor model interacts with a correctly-coded RTL design?
  – 1st Examine: Bad-vendor1 model driving a good RTL model
  – 2nd Examine: Good RTL model driving a bad vendor2 model

![Diagram with nodes and edges representing blocking and nonblocking assignments between vendor models and RTL models.](image-url)
module vendor1_b0 (  
    output reg b,  
    input     a, clk, rst_n);  
  
  always @(posedge clk or negedge rst_n)  
    if (!rst_n) b = 0;  
    else        b = a;  
endmodule

module vendor1_b1 (  
    output reg b,  
    input     a, clk, rst_n);  
  
  always @(posedge clk or negedge rst_n)  
    if (!rst_n) b = #1 0;  
    else        b = #1 a;  
endmodule

Error in vendor1 coding style
module myrtl_nb0 (  
    output reg c,  
    input    b, clk, rst_n);  
  
    always @(posedge clk or negedge rst_n)  
      if (!rst_n) c <= 0;  
      else        c <= b;  
endmodule

module myrtl_nb1 (  
    output reg c,  
    input    b, clk, rst_n);  
  
    always @(posedge clk or negedge rst_n)  
      if (!rst_n) c <= #1 0;  
      else        c <= #1 b;  
endmodule
module vendor2_b0 (  
    output reg d,  
    input    c, clk, rst_n);  

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

module vendor2_b1 (  
    output reg d,  
    input    c, clk, rst_n);  

    always @(posedge clk or negedge rst_n)  
        if (!rst_n) d = #1 0;  
        else        d = #1 c;  
endmodule

Error in vendor2 coding style
Bad-Vendor1 Driving Good-RTL
(Equivalent code after compiled & flattened designs)

Scenario #1 & Scenario #2 both have potential race conditions

**Scenario #1**

```verilog
always @(posedge clk ...)
  ... begin
  b = a;
  c <= b;
  ...
```

**Vendor1_b0**

```
b = a
```

**Myrtl_nb0**

```
c <= b
```

**Passes**

**Scenario #2**

```verilog
always @(posedge clk ...)
  ... begin
  b = #1 a;
  c <= b;
  ...
```

**Vendor1_b1**

```
b = #1 a
```

**Myrtl_nb0**

```
c <= b
```

**Fails**
Bad-Vendor1 Driving Good-RTL
(Equivalent code after compiled & flattened designs)

Scenario #3 & Scenario #4 also both have potential race conditions

always @(posedge clk ...)
... begin
b = a;
c <= #1 b; ...

Fails

always @(posedge clk ...)
... begin
c <= #1 b;
b = a; ...

Passes

always @(posedge clk ...)
... begin
b = #1 a;
c <= #1 b; ...

Fails

always @(posedge clk ...)
... begin
c <= #1 b;
b = #1 a; ...

Passes
Good-RTL Driving Bad-Vendor2
(Equivalent code after compiled & flattened designs)

Scenario #5 & Scenario #6 both simulate with no race conditions

Scenario #5

```
always @(posedge clk ...)
... begin
  c <= b;
  d = c;
...
```
Passes

Scenario #6

```
always @(posedge clk ...)
... begin
  c <= b;
  d = #1 c;
...
```
Passes
Good-RTL Driving Bad-Vendor2

(Equivalent code after compiled & flattened designs)

Scenario #7 & Scenario #8 both simulate with no race conditions

always @(posedge clk ...) ... begin
  c <= #1 b;
  d = c; ... Passes

always @(posedge clk ...) ... begin
  d = c;
  c <= #1 b; ... Passes

Scenario #7

myrtl_nb1

b

clk

rst_n

c <= #1 b

d = c

d

c <= #1 b

d = #1 c

Scenario #8

vendor1_b0

myrtl_nb1

b

clk

rst_n

c <= #1 b

d = #1 c

d

c <= #1 b

d = #1 c

Passes

vendor1_b1

clk

rst_n

c <= #1 b

d = #1 c

d

c <= #1 b

d = #1 c

Passes

Passes
• "Adding #1 to my nonblocking assignments makes up for vendor coding problems"

<table>
<thead>
<tr>
<th>Vendor1 Model</th>
<th>RTL Model</th>
<th>Vendor2 Model</th>
<th>Race Condition?</th>
</tr>
</thead>
<tbody>
<tr>
<td>b = a</td>
<td>c &lt;= b</td>
<td></td>
<td>potential race condition</td>
</tr>
<tr>
<td>b = #1 a</td>
<td>c &lt;= #1 b</td>
<td></td>
<td>potential race condition</td>
</tr>
<tr>
<td>b = a</td>
<td>c &lt;= #1 b</td>
<td></td>
<td>potential race condition</td>
</tr>
<tr>
<td>b = #1 a</td>
<td>c &lt;= b</td>
<td>d = c</td>
<td>NO race condition</td>
</tr>
<tr>
<td></td>
<td></td>
<td>d = #1 c</td>
<td>NO race condition</td>
</tr>
<tr>
<td></td>
<td></td>
<td>d = c</td>
<td>NO race condition</td>
</tr>
<tr>
<td></td>
<td></td>
<td>d = #1 c</td>
<td>NO race condition</td>
</tr>
</tbody>
</table>

Nope! ... Sorry!
Benchmark Circuit #3

Only add delays to I/O registers

- Added #1 delays to io#f models
- 20 x 1000-bit registers
- Added #1 delays to io#f models

No delays on dff models

Input: clk, rst_n

Outputs: d, q, qq1, d, q, qq2, ..., qq18, d, q, qq19, d, q, q
Benchmark Results - Circuit #3
With delays only on the I/O flip-flops

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Time (s)</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM ThinkPad T21, Pentium III-850MHz, 384MB RAM, Redhat Linux 6.2, VCS</td>
<td>292.920</td>
<td>Baseline no-delay model</td>
</tr>
<tr>
<td>Version 6.2 - #1 delays only added to the 2,000 I/O flip-flops</td>
<td></td>
<td></td>
</tr>
<tr>
<td>No delays</td>
<td>292.920</td>
<td>Baseline no-delay model</td>
</tr>
<tr>
<td>Nonblocking #1 delays ( &lt;= #1 )</td>
<td>376.460</td>
<td>29% slower</td>
</tr>
<tr>
<td>Nonblocking #1 delays only on the 2,000 I/O flip-flops</td>
<td>375.710</td>
<td>28% slower</td>
</tr>
<tr>
<td>SUN Ultra 80, UltraSPARC-II 450MHz, 1GB RAM, Solaris 8, VCS Version 6.2</td>
<td>438.090</td>
<td>Baseline no-delay model</td>
</tr>
<tr>
<td>- #1 delays only added to the 2,000 I/O flip-flops</td>
<td></td>
<td></td>
</tr>
<tr>
<td>No delays</td>
<td>438.090</td>
<td>Baseline no-delay model</td>
</tr>
<tr>
<td>Nonblocking #1 delays ( &lt;= #1 )</td>
<td>839.270</td>
<td>92% slower</td>
</tr>
<tr>
<td>Nonblocking #1 delays only on the 2,000 I/O flip-flops</td>
<td>833.720</td>
<td>90% slower</td>
</tr>
</tbody>
</table>
• Why run gate-level simulations with SDF timing delays?

Isn't is good enough to do
(1) functional simulations,
(2) static timing analysis (STA), and
(3) equivalence check the gates model to the RTL model?

– Full system simulation
– Equivalence checking software costs money
– Final regression simulations with SDF timing delays verifies STA and equivalence checked models
Resets

initial begin
    rst_n = 0;
    ...
end

always @(posedge clk or negedge rst_n)
    ...

Race condition

initial begin
    rst_n <= 0;
    ...
end

always @(posedge clk or negedge rst_n)
    ...

No race condition
Common clock oscillator
(clk=0 at time 0)

clk=1 at time 0
No race condition

```
`define cycle 10
...
initial begin
    clk = 0;
    forever #(\cycle/2) clk = ~clk);
end
```

```
`define cycle 10
...
initial begin
    clk <= 1;
    forever #(\cycle/2) clk = ~clk);
end
```
Testbench Tricks
Change stimulus on clock edges

module tb;
    reg a, b, clk, rst_n;

    initial begin
        clk = 0;
        forever #10 clk = ~clk;
    end

    sblk1 u1 (.q2(q2), .a(a), .b(b),
        .clk(clk), .rst_n(rst_n));

    initial begin
        a = 0; b = 0;
        rst_n <= 0;
        @(posedge clk);
        @(negedge clk) rst_n = 1;
        a = 1; b = 1;
        @(negedge clk) a = 0;
        @(negedge clk) b = 0;
        @(negedge clk) $finish;
    end
endmodule

If the clock period changes, the testbench still stays the same

Simple testbench

Free-running clock oscillator

Initialize \( a \) & \( b \)

Reset at time 0 for one clock cycle

(\( \text{negedge} \ \text{clk} \)) release reset set \( a \) and \( b \)

(\( \text{negedge} \ \text{clk} \)) change \( a \)

(\( \text{negedge} \ \text{clk} \)) change \( b \)

(\( \text{negedge} \ \text{clk} \)) finish simulation
The Bergeron Guidelines
Four flawed guidelines

- Four "Guidelines for Avoiding Race Conditions:"
  - If a register is declared outside of the always or initial block, assign to it using a nonblocking assignment. Reserve the blocking assignment for registers local to the block

Better

Guideline #5: Do not mix blocking and nonblocking assignments in the same always block

- Assign to a register from a single always or initial block

Better

Guideline #6: Do not make assignments to the same variable from more than one always block
• Four "Guidelines for Avoiding Race Conditions:" (cont.)

  – Use continuous assignments to drive inout pins only. Do not use them to model internal combinational functions. Prefer sequential code instead.

  Procedural blocks are more prone to Verilog race conditions

  For simple Boolean expressions, use continuous assignments

  To group assignments or to include case, if-else, for-loops, use procedural blocks

  – Do not assign any value at time 0

  Initialize everything at time 0

  Use testbench nonblocking assignment tricks to avoid race conditions
8 Important Guidelines

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

Guideline #1: Sequential logic - use nonblocking assignments

Guideline #2: Latches - use nonblocking assignments

Guideline #3: Combinational logic in an always block - use blocking assignments

Guideline #4: Mixed sequential and combinational logic in the same always block - use nonblocking assignments

Guideline #5: Do not mix blocking and nonblocking assignments in the same always block

Guideline #6: Do not make assignments to the same variable from more than one always block

Guideline #7: Use $strobe to display values that have been assigned using nonblocking assignments

Guideline #8: Do not make #0 procedural assignments
Conclusions

• Follow the 8 important coding guidelines
• Do not mix blocking and nonblocking assignments
• Either code NBAs with no delays or with macro #1 delays

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

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

- or -

```
`define D #1
```

No delays

Simulations run up to 100% faster without #1 delays

• Mixed RTL & gates simulations require RTL-output delays
• Remember +nbaopt and +rad for faster VCS simulation

• Request: please give us a +nbal1 switch!