VERILOG CODING STYLES FOR IMPROVED SIMULATION EFFICIENCY

Clifford E. Cummings
cliffc@sunburst-design.com / www.sunburst-design.com

Sunburst Design, Inc.
14314 SW Allen Blvd.
PMB 501
Beaverton, OR 97005

INTERNATIONAL CADENCE USER GROUP CONFERENCE
OCTOBER 5-9, 1997
SAN DIEGO, CALIFORNIA

Verilog Coding Styles for Improved Simulation Efficiency

Clifford E. Cummings
Sunburst Design, Inc.
14314 SW Allen Blvd., PMB 501, Beaverton, OR 97007
Phone: 503-641-8446 Email: cliffc@sunburst-design.com.com

Abstract

This paper details different coding styles and their impact on Verilog-XL simulation efficiency.

1. Introduction

What are some of the more optimal ways to code Verilog models and testbenches to shorten simulation times? This paper is a collection of interesting coding style comparisons that have been run on Verilog-XL.

2. Verilog Efficiency Testing

The intent of this paper is to help identify which Verilog coding styles are more efficient than others; thereby, increasing design and simulation efficiency.

All of the Verilog benchmarks in this paper are intended to show the efficiency of how specific coding styles run on the same simulator. This paper is not intended to be a benchmark comparison between different simulators.

All of the benchmarks were run on a Sparc 5, running Solaris 2.5, with 32MB of memory and 500MB of swap space. Verilog-XL, version 2.21, compiled with Undertow version 5.3.3, was used for these benchmarks.

Note: over time, measured efficiency will likely change as newer versions of the same simulator are introduced and as simulators become more efficient.

3. Case Statements Vs. Large If/Else-If Structures

Question: Is there a simulation efficiency difference in coding large case statements as equivalent if/else-if statements?

The testcase for this benchmark is a synthesizable 8-to-1 multiplexer written as both a case statement and as a large if/else-if construct. Figure 1 shows the code for the case-statement multiplexer, Figure 2 shows the code for the equivalent if/else-if-statement multiplexer.

3.1 Case Vs. If Efficiency Summary

The results in Table 1 show that, using Verilog-XL, this 8-item case structure took about the same amount of memory to implement as the if/else-if structure but the case statement was about 6% faster.

```
module CaseMux8 (y, i, sel);
    output       y;
    input  [7:0] i;
    input  [2:0] sel;

    reg          y;
    wire   [7:0] i;
    wire   [2:0] sel;

    always @(i or sel)
      case (sel)
        3'd0: y = i[0];
        3'd1: y = i[1];
        3'd2: y = i[2];
        3'd3: y = i[3];
        3'd4: y = i[4];
        3'd5: y = i[5];
        3'd6: y = i[6];
        3'd7: y = i[7];
      endcase
  endmodule
```

Figure 1

```
module IfMux8 (y, i, sel);
    output       y;
    input  [7:0] i;
    input  [2:0] sel;

    reg          y;
    wire   [7:0] i;
    wire   [2:0] sel;

    always @(i or sel)
      if      (sel == 3'd0) y = i[0];
      else if (sel == 3'd1) y = i[1];
      else if (sel == 3'd2) y = i[2];
      else if (sel == 3'd3) y = i[3];
      else if (sel == 3'd4) y = i[4];
      else if (sel == 3'd5) y = i[5];
      else if (sel == 3'd6) y = i[6];
      else if (sel == 3'd7) y = i[7];
  endmodule
```

Figure 2

Sparc5/32MB RAM 491MB Swap Solaris version 2.5 Verilog-XL 2.2.1 - Undertow 5.3.3	Data Structure (bytes of memory)	Memory Usage Percentages	Simulation CPU Time (in seconds)	CPU Simulation Percentages
CaseMux8	92252	100.00%	1209.3	100.00%
IfMux8	92912	100.72%	1282.0	106.01%

Table 1

4. Begin-End Statements

Question: Is there a simulation efficiency difference when extra begin-end pairs are added to Verilog models?

The testcase for this benchmark is a synthesizable D flip-flop with asynchronous reset. In Figure 3 the synthesizable flip-flop was written with no begin-end statements in the always block (they are not needed for this model). In Figure 4, the same flip-flop was written with three unnecessary begin-end statements. 1000 flip-flops were then instantiated into a testbench and simulated.

```
// Removed unneeded begin-end pairs
module dff (q, d, clk, rst);
  output q;
  input  d, clk, rst;
  reg    q;

always @(posedge clk or posedge rst)
      if (rst == 1) q = 0;
      else          q = d;

endmodule
```

Figure 3

4.1 Begin-End Efficiency Summary

The results in Table 2 show that, using Verilog-XL, the flip-flop with three extra begin-end statements took about 6% more memory to implement and about 6% more time to simulate as the flip-flop with no begin-end statements.

```
// Includes unneeded begin-end pairs
module dff (q, d, clk, rst);
  output q;
  input  d, clk, rst;
  reg    q;

    always @(posedge clk or posedge rst)
    begin
      if (rst == 1) begin
        q = 0;
      end
      else begin
        q = d;
      end
    end

endmodule
```

Figure 4

Sparc5/32MB RAM 491MB Swap Solaris version 2.5 Verilog-XL 2.2.1 Undertow 5.3.3	Data Structure (bytes of memory)	Memory Usage Percentages	Simulation CPU Time (in seconds)	CPU Simulation Percentages
nobegin	4677676	100%	9403.2	100%
begin	4953264	105.89%	9960.4	105.93%

Table 2

5. `define Vs. Parameters

Question: What simulation efficiency difference is there between using `define macros and parameters? How is parameter efficiency affected by redefining parameters using parameter redefinition in the model instantiation and by using defparam statements?
The testcases for this benchmark were models with nine parameter delays (Figure 5) and nine `define-macro delays (Figure 6).

Testbenches were created with defined-macros (Figure 7), defparam statements (Figure 8) and #-parameter redefinition (Figure 9).

5.1 Define & Parameter Efficiency Summary

The results in Table 3 show that, parameter redefinition and defparams occupy about 25% - 35% more memory than do `define statements.

```
module paramchk (y, i, en);
  output [9:0] y;
  input        i, en;
  parameter Tr_min = 2;
  parameter Tr_typ = 5;
  parameter Tr_max = 8;
  parameter Tf_min = 1;
  parameter Tf_typ = 4;
  parameter Tf_max = 7;
  parameter Tz_min = 3;
  parameter Tz_typ = 6;
  parameter Tz_max = 9;

  bufif1 #(Tr_min:Tr_typ:Tr_max,Tf_min:Tf_typ:Tf_max,Tz_min:Tz_typ:Tz_max) b9 (y[9],i,en);
  bufif1 #(Tr_min:Tr_typ:Tr_max,Tf_min:Tf_typ:Tf_max,Tz_min:Tz_typ:Tz_max) b8 (y[8],i,en);
  bufif1 #(Tr_min:Tr_typ:Tr_max,Tf_min:Tf_typ:Tf_max,Tz_min:Tz_typ:Tz_max) b7 (y[7],i,en);
  bufif1 #(Tr_min:Tr_typ:Tr_max,Tf_min:Tf_typ:Tf_max,Tz_min:Tz_typ:Tz_max) b6 (y[6],i,en);
  bufif1 #(Tr_min:Tr_typ:Tr_max,Tf_min:Tf_typ:Tf_max,Tz_min:Tz_typ:Tz_max) b5 (y[5],i,en);
  bufif1 #(Tr_min:Tr_typ:Tr_max,Tf_min:Tf_typ:Tf_max,Tz_min:Tz_typ:Tz_max) b4 (y[4],i,en);
  bufif1 #(Tr_min:Tr_typ:Tr_max,Tf_min:Tf_typ:Tf_max,Tz_min:Tz_typ:Tz_max) b3 (y[3],i,en);
  bufif1 #(Tr_min:Tr_typ:Tr_max,Tf_min:Tf_typ:Tf_max,Tz_min:Tz_typ:Tz_max) b2 (y[2],i,en);
  bufif1 #(Tr_min:Tr_typ:Tr_max,Tf_min:Tf_typ:Tf_max,Tz_min:Tz_typ:Tz_max) b1 (y[1],i,en);
  bufif1 #(Tr_min:Tr_typ:Tr_max,Tf_min:Tf_typ:Tf_max,Tz_min:Tz_typ:Tz_max) b0 (y[0],i,en);
endmodule
```

Figure 5

```
module definechk (y, i, en);
  output [9:0] y;
  input        i, en;

  bufif1 #(`Tr_min:`Tr_typ:`Tr_max,`Tf_min:`Tf_typ:`Tf_max,`Tz_min:`Tz_typ:`Tz_max) b9 (y[9], i, en);
  bufif1 #(`Tr_min:`Tr_typ:`Tr_max,`Tf_min:`Tf_typ:`Tf_max,`Tz_min:`Tz_typ:`Tz_max) b8 (y[8], i, en);
  bufif1 #(`Tr_min:`Tr_typ:`Tr_max,`Tf_min:`Tf_typ:`Tf_max,`Tz_min:`Tz_typ:`Tz_max) b7 (y[7], i, en);
  bufif1 #(`Tr_min:`Tr_typ:`Tr_max,`Tf_min:`Tf_typ:`Tf_max,`Tz_min:`Tz_typ:`Tz_max) b6 (y[6], i, en);
  bufif1 #(`Tr_min:`Tr_typ:`Tr_max,`Tf_min:`Tf_typ:`Tf_max,`Tz_min:`Tz_typ:`Tz_max) b5 (y[5], i, en);
  bufif1 #(`Tr_min:`Tr_typ:`Tr_max,`Tf_min:`Tf_typ:`Tf_max,`Tz_min:`Tz_typ:`Tz_max) b4 (y[4], i, en);
  bufif1 #(`Tr_min:`Tr_typ:`Tr_max,`Tf_min:`Tf_typ:`Tf_max,`Tz_min:`Tz_typ:`Tz_max) b3 (y[3], i, en);
  bufif1 #(`Tr_min:`Tr_typ:`Tr_max,`Tf_min:`Tf_typ:`Tf_max,`Tz_min:`Tz_typ:`Tz_max) b2 (y[2], i, en);
  bufif1 #(`Tr_min:`Tr_typ:`Tr_max,`Tf_min:`Tf_typ:`Tf_max,`Tz_min:`Tz_typ:`Tz_max) b1 (y[1], i, en);
  bufif1 #(`Tr_min:`Tr_typ:`Tr_max,`Tf_min:`Tf_typ:`Tf_max,`Tz_min:`Tz_typ:`Tz_max) b0 (y[0], i, en);
endmodule
```

Figure 6

```
`define Tr_min 2
`define Tr_typ 5
`define Tr_max 8
`define Tf_min 1
`define Tf_typ 4
`define Tf_max 7
`define Tz_min 3
`define Tz_typ 6
`define Tz_max 9

`define CNT 1000000
`define cycle 20
`timescale 1ns / 1ns

module tb_define;
  wire [9:0] y9, y8, y7, y6, y5, y4, y3, y2, y1, y0;
  reg        i, en;
  reg        clk;

  definechk i9 (y9, i, en);
  definechk i8 (y8, i, en);
  definechk i7 (y7, i, en);
  definechk i6 (y6, i, en);
  definechk i5 (y5, i, en);
  definechk i4 (y4, i, en);
  definechk i3 (y3, i, en);
  definechk i2 (y2, i, en);
  definechk i1 (y1, i, en);
  definechk i0 (y0, i, en);

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

  initial begin
    i = 0; en = 1;
    repeat (`CNT) begin
      @(negedge clk) i = ~i;
    end
    @(negedge clk) i = ~i;
    repeat (`CNT) begin
      @(negedge clk) en = ~en;
    end
    @(negedge clk) i = ~i;
    repeat (`CNT) begin
      @(negedge clk) en = ~en;
    end
  `ifdef RUN
    @(negedge clk) $finish(2);
  `else
    @(negedge clk) $stop(2);
`endif
  end
endmodule
```

Figure 7

```
`define CNT 1000000
`define cycle 20
`timescale 1ns / 1ns

module tb_defparam;
  wire [9:0] y9, y8, y7, y6, y5, y4, y3, y2, y1, y0;
  reg        i, en, clk;

  paramchk i9 (y9, i, en);
  defparam i9.Tr_min=0;  defparam i9.Tr_typ=1;  defparam i9.Tr_max=2;
  defparam i9.Tf_min=3;  defparam i9.Tf_typ=4;  defparam i9.Tf_max=5;
  defparam i9.Tz_min=6;  defparam i9.Tz_typ=7;  defparam i9.Tz_max=8;
  paramchk i8 (y8, i, en);
  defparam i8.Tr_min=0;  defparam i8.Tr_typ=1;  defparam i8.Tr_max=2;
  defparam i8.Tf_min=3;  defparam i8.Tf_typ=4;  defparam i8.Tf_max=5;
  defparam i8.Tz_min=6;  defparam i8.Tz_typ=7;  defparam i8.Tz_max=8;
  paramchk i7 (y7, i, en);
  defparam i7.Tr_min=0;  defparam i7.Tr_typ=1;  defparam i7.Tr_max=2;
  defparam i7.Tf_min=3;  defparam i7.Tf_typ=4;  defparam i7.Tf_max=5;
  defparam i7.Tz_min=6;  defparam i7.Tz_typ=7;  defparam i7.Tz_max=8;
  paramchk i6 (y6, i, en);
  defparam i6.Tr_min=0;  defparam i6.Tr_typ=1;  defparam i6.Tr_max=2;
  defparam i6.Tf_min=3;  defparam i6.Tf_typ=4;  defparam i6.Tf_max=5;
  defparam i6.Tz_min=6;  defparam i6.Tz_typ=7;  defparam i6.Tz_max=8;
  paramchk i5 (y5, i, en);
  defparam i5.Tr_min=0;  defparam i5.Tr_typ=1;  defparam i5.Tr_max=2;
  defparam i5.Tf_min=3;  defparam i5.Tf_typ=4;  defparam i5.Tf_max=5;
  defparam i5.Tz_min=6;  defparam i5.Tz_typ=7;  defparam i5.Tz_max=8;
  paramchk i4 (y4, i, en);
  defparam i4.Tr_min=0;  defparam i4.Tr_typ=1;  defparam i4.Tr_max=2;
  defparam i4.Tf_min=3;  defparam i4.Tf_typ=4;  defparam i4.Tf_max=5;
  defparam i4.Tz_min=6;  defparam i4.Tz_typ=7;  defparam i4.Tz_max=8;
  paramchk i3 (y3, i, en);
  defparam i3.Tr_min=0;  defparam i3.Tr_typ=1;  defparam i3.Tr_max=2;
  defparam i3.Tf_min=3;  defparam i3.Tf_typ=4;  defparam i3.Tf_max=5;
  defparam i3.Tz_min=6;  defparam i3.Tz_typ=7;  defparam i3.Tz_max=8;
  paramchk i2 (y2, i, en);
  defparam i2.Tr_min=0;  defparam i2.Tr_typ=1;  defparam i2.Tr_max=2;
  defparam i2.Tf_min=3;  defparam i2.Tf_typ=4;  defparam i2.Tf_max=5;
  defparam i2.Tz_min=6;  defparam i2.Tz_typ=7;  defparam i2.Tz_max=8;
  paramchk i1 (y1, i, en);
  defparam i1.Tr_min=0;  defparam i1.Tr_typ=1;  defparam i1.Tr_max=2;
  defparam i1.Tf_min=3;  defparam i1.Tf_typ=4;  defparam i1.Tf_max=5;
  defparam i1.Tz_min=6;  defparam i1.Tz_typ=7;  defparam i1.Tz_max=8;
  paramchk i0 (y0, i, en);
  defparam i0.Tr_min=0;  defparam i0.Tr_typ=1;  defparam i0.Tr_max=2;
  defparam i0.Tf_min=3;  defparam i0.Tf_typ=4;  defparam i0.Tf_max=5;
  defparam i0.Tz_min=6;  defparam i0.Tz_typ=7;  defparam i0.Tz_max=8;

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

  initial begin
    i = 0; en = 1;
    repeat (`CNT) @(negedge clk) i  = ~i;
                  @(negedge clk) i  = ~i;
    repeat (`CNT) @(negedge clk) en = ~en;
                  @(negedge clk) i  = ~i;
    repeat (`CNT) @(negedge clk) en = ~en;
  `ifdef RUN  @(negedge clk) $finish(2);
  `else       @(negedge clk) $stop(2);
  `endif
  end
endmodule
```

Figure 8

```
`define CNT 1000000
`define cycle 20
`timescale 1ns / 1ns

module tb_param;
  wire [9:0] y9, y8, y7, y6, y5, y4, y3, y2, y1, y0;
  reg        i, en;
  reg        clk;

  paramchk #(0,1,2,3,4,5,6,7,8) i9 (y9, i, en);
  paramchk #(0,1,2,3,4,5,6,7,8) i8 (y8, i, en);
  paramchk #(0,1,2,3,4,5,6,7,8) i7 (y7, i, en);
  paramchk #(0,1,2,3,4,5,6,7,8) i6 (y6, i, en);
  paramchk #(0,1,2,3,4,5,6,7,8) i5 (y5, i, en);
  paramchk #(0,1,2,3,4,5,6,7,8) i4 (y4, i, en);
  paramchk #(0,1,2,3,4,5,6,7,8) i3 (y3, i, en);
  paramchk #(0,1,2,3,4,5,6,7,8) i2 (y2, i, en);
  paramchk #(0,1,2,3,4,5,6,7,8) i1 (y1, i, en);
  paramchk #(0,1,2,3,4,5,6,7,8) i0 (y0, i, en);

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

  initial begin
    i = 0; en = 1;
    repeat (`CNT) begin
      @(negedge clk) i = ~i;
    end
    @(negedge clk) i = ~i;
    repeat (`CNT) begin
      @(negedge clk) en = ~en;
    end
    @(negedge clk) i = ~i;
    repeat (`CNT) begin
      @(negedge clk) en = ~en;
    end
  `ifdef RUN
    @(negedge clk) $finish(2);
  `else
    @(negedge clk) $stop(2);
  `endif
  end
endmodule
```

Figure 9

Sparc5/32MB RAM 491MB Swap Solaris version 2.5 Verilog-XL 2.2.1 - Undertow 5.3.3	Data Structure (bytes of memory)	Memory Usage Percentages	Simulation CPU Time (in seconds)	CPU Simulation Percentages
define	162448	100.00%	1436.9	100.00%
paramredef	201236	123.88%	1427.2	99.32%
defparam	220568	135.78%	1420.0	98.82%

Table 3

6. Grouping of Assignments Within Always Blocks

Question: Is there a penalty for splitting assignments into multiple always blocks as opposed to grouping the same assignments into fewer always blocks?

The testcase for this benchmark is a conditionally complied set of four assignments in four separate always blocks, or the same four assignments in a single always block (Figure 10).

6.1 Always Block Grouping Efficiency Summary

The results in Table 4 show that, four assignments in four separate always blocks is about 34% slower than placing the same assignments in a single always block

```
`define ICNT 10000000
`define cycle 100
`timescale 1ns / 100ps

module AlwaysGroup;
  reg       clk;
  reg [7:0] a, b, c, d, e;

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

  initial begin
    a = 8'haa;
    forever @(negedge clk) a = ~ a;
  end

  initial begin
    repeat(`ICNT) @(posedge clk);
  `ifdef RUN @(posedge clk) $finish(2);
  `else      @(posedge clk) $stop(2);
  `endif
  end

  `ifdef GROUP4 // Group of four always blocks
    always @(posedge clk) begin
      b <= a;
    end
    always @(posedge clk) begin
      c <= b;
    end
    always @(posedge clk) begin
      d <= c;
    end
    always @(posedge clk) begin
      e <= d;
    end

  `else // Four assignments grouped into a
        // single always block
    always @(posedge clk) begin
      b <= a;
      c <= b;
      d <= c;
      e <= d;
    end
  `endif

endmodule
```

Figure 10

Sparc5/32MB RAM 491MB Swap Solaris version 2.5 Verilog-XL 2.2.1 - Undertow 5.3.3	Data Structure (bytes of memory)	Memory Usage Percentages	Simulation CPU Time (in seconds)	CPU Simulation Percentages
group1	85712	100.00%	1196.6	100.00%
group4	87544	102.14%	1607.2	134.31%

Table 4

7. Port Connections

Question: Is there a penalty for passing data over module ports as opposed to passing data by hierarchical reference?

The idea for this benchmark came from a paper presented by Martin Gravenstein[1] of Ford Microelectronics at the 1994 International Verilog Conference.

The testcase for this benchmark is a pair of flip-flops. The first flip-flop has no ports and testbench communication with this model was conducted by hierarchical reference. The second flip-flop is a model with normal port communication with a testbench (Figure 11).

7.1 Ports/No Ports Efficiency Summary

The results in Table 5 show that, communicating with a four-port model as opposed to referencing the ports hierarchically required about 46% more simulation time

Model port usage and communication is still recommended; however, passing monitor data over ports would be simulation-time expensive. It is better to reference monitored state and bus data hierarchically

These results also suggest that models with extra levels of hierarchy will significantly slow down a simulation

```
`ifdef NOPORTS
  module PortModels;
    reg    [15:0] q;
    reg    [15:0] d;
    reg           clk, rstN;

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

  endmodule
`else
  module PortModels (q, d, clk, rstN);
    output [15:0] q;
    input  [15:0] d;
    input         clk, rstN;
    reg    [15:0] q;

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

  endmodule
`endif
```

Figure 11

Sparc5/32MB RAM 491MB Swap Solaris version 2.5 Verilog-XL 2.2.1 - Undertow 5.3.3	Data Structure (bytes of memory)	Memory Usage Percentages	Simulation CPU Time (in seconds)	CPU Simulation Percentages
NoPorts	90948	100.00%	1343.7	100.00%
Ports	89436	98.34%	1967.4	146.42%

Table 5

8. `timescale Efficiencies

Question: Are there any simulator performance penalties for using a higher precision `timescale during simulation?

The testcase for this benchmark is a buffer and inverter with propagation delays that are simulated with a variety of `timescales (Figure 12).

8.1 `timescale Efficiency Summary

The results in Table 6 show that, using a `timescale of 1ns/1ps requires about 156% more memory and about 99% more time to simulate than the same model using a timescale of 1ns/1ns.

9. Displaying $time Values

Question: Are there any simulator performance penalties for displaying different $time values to STDOUT during simulation? Do the $time display format specifiers affect simulation performance?

The idea for this benchmark came from a paper presented by Jay Lawrence[2] of Cadence at the 1995 ICU. The testcase for this benchmark is a set of different display commands of time values (Figure 13).

9.1 Display $time Efficiency Summary

```
`define ICNT 10000000
`define cycle 10

`ifdef Time_1ns      `timescale 1ns / 1ns
`endif
`ifdef Time_100ps    `timescale 1ns / 100ps
`endif
`ifdef Time_10ps     `timescale 1ns / 10ps
`endif
`ifdef Time_1ps      `timescale 1ns / 1ps
`endif

module TimeModel;
  reg         i;
  wire [1:2]  y;

  initial begin
    i = 0;
    forever #(`cycle) i = ~i;
  end

  initial begin
    repeat(`ICNT) @(posedge i);
  `ifdef RUN  @(posedge i) $finish(2);
  `else       @(posedge i) $stop(2);
  `endif
  end

  buf #(2.201, 3.667) i1 (y[1], i);
  not #(4.633, 7.499) i2 (y[2], i);

endmodule
```

Figure 12

The results in Table 7 show that, needless display of $time values is very costly in simulation time

Sparc5/32MB RAM 491MB Swap Solaris version 2.5 Verilog-XL 2.2.1 - Undertow 5.3.3	Data Structure (bytes of memory)	Memory Usage Percentages	Simulation CPU Time (in seconds)	CPU Simulation Percentages
Time_1ns	83916	100.00%	1395.0	100.00%
Time_100ps	83920	100.00%	1459.4	104.62%
Time_10ps	92620	110.37%	1718.1	123.16%
Time_1ps	214476	255.58%	2777.8	199.13%

Table 6

Sparc5/32MB RAM 491MB Swap Solaris version 2.5 Verilog-XL 2.2.1 - Undertow 5.3.3	Data Structure (bytes of memory)	Memory Usage Percentages	Simulation CPU Time (in seconds)	CPU Simulation Percentages
No_Display	84784	100.00%	906.6	100.00%
One_Display	84924	100.17%	1297.3	143.10%
Two_Display	85060	100.33%	1646.1	181.57%
Two_Display0	85064	100.33%	1624.5	179.19%
Two_Format0	85024	100.28%	1348.4	148.73%
Two_RtimeFormat0	85832	101.24%	1328.1	146.49%

Table 7

```
`define hcycle 12.5
`define ICNT 1000000
`timescale 1ns / 100ps

module DisplayTime;
  reg     clk;
  time    rClkTime,  fClkTime;
  real   rRealTime, fRealTime;

  initial begin
    clk = 0;
    forever #(`hcycle) clk = ~clk;
  end

  initial begin
    repeat(`ICNT) @(posedge clk);
  `ifdef RUN  @(posedge clk) $finish(2);
  `else       @(posedge clk) $stop(2);
  `endif
  end

  `ifdef NoDisplay // display with no time values
    always @(negedge clk) begin
      fClkTime = $time;
      $display ("Negedge Clk");
    end
  `endif

  `ifdef OneDisplay // display with one time value
    always @(negedge clk) begin
      fClkTime = $time;
      $display ("Negedge Clk at %d", fClkTime);
    end
  `endif

  `ifdef TwoDisplay // display with two time values
    always @(negedge clk) begin
      fClkTime = $time;
      $display ("Posedge Clk at %d - Negedge Clk at %d", rClkTime, fClkTime);
    end
  `endif

  `ifdef TwoDisplay0 // display with two time values
    always @(negedge clk) begin
      fClkTime = $time;
      $display ("Posedge Clk at %0d - Negedge Clk at %0d", rClkTime, fClkTime);
    end
  `endif

  `ifdef TwoFormat0 // display with two time values
    always @(negedge clk) begin
      fClkTime = $time;
      $display ("Posedge Clk at %0t - Negedge Clk at %0t", rClkTime, fClkTime);
    end
  `endif

  `ifdef TwoRtimeFormat0 // display with two time values
    initial $timeformat(-9,2,"ns",15);

    always @(negedge clk) begin
      fRealTime = $realtime;
      $display ("Posedge Clk at %0t - Negedge Clk at %0t", rRealTime,fRealTime);
    end
  `endif

  `ifdef TwoRtimeFormat0  always @(posedge clk) rRealTime = $realtime;
  `else                   always @(posedge clk) rClkTime = $time;
  `endif

endmodule
```

Figure 13

10. Clock Oscillators

Question: Are there any significant simulator performance advantages to implementing a clock oscillator using an always block, forever loop or Verilog gate primitives?

The testcase for this benchmark are three conditionally compiled clock oscillator implementations, clocking a flip-flop model (Figure 14).

10.1 Clock Oscillator Efficiency Summary

The results in Table 8 show that, gate clock oscillators were about %85% slower than behavioral clock oscillators.

11. Efficiency - Final Word

Simulation efficiency should not be the only Verilog coding criteria.

Code readability, simulation accuracy and displaying timely simulation and diagnostic information might actually increase design productivity. However, reckless use of inefficient coding styles when a more efficient alternative exists is detrimental to simulation productivity.

Again, these benchmarks were only run on Verilog-XL. Mileage may vary on other simulators.

12. References

[1] M. Gravenstein, "Modeling Techniques to Support System Level Simulation and a Top-Down Development Methodology," *International Verilog HDL Conference Proceedings 1994*, pp 43-50

[2] J. Lawrence & C. Ussery, "INCA: A Next-Generation Architecture for Simulation," *International Cadence Users Group Conference Proceedings 1995*, pp 105-109


```
`define ICNT 100_000
`define cycle 100
`timescale 1ns / 1ns

module Clocks;
  reg d, rstN;

  `ifdef ALWAYS
    reg clk;     // driven by a procedural block
    initial clk = 0;
    always        // free running behave clk #1
      #(`cycle/2) clk = ~clk;
  `endif

  `ifdef FOREVER
    reg clk;     // driven by a procedural block
    initial begin
      clk = 0;
      forever #(`cycle/2) clk = ~clk;
    end
  `endif

  `ifdef GATE  // free running clk #3 (gate)
    reg  start;
    wire clk;    // driven by a gate
    initial begin
      start = 0; #(`cycle/2) start = 1;
    end
    nand #(`cycle/2) (clk, clk, start);
  `endif

  dff d1 (q, d, clk, rstN);

  initial begin
    rstN = 0; d = 1;
    @(negedge clk) rstN = 1;
    repeat(`ICNT) @(posedge clk);
  `ifdef RUN   @(posedge clk) $finish(2);
  `else        @(posedge clk) $stop(2);
  `endif
  end

// Veritools Undertow-dumpfile option
`ifdef UT
  initial begin
    $dumpfile(dump.ut);  $vtDumpvars;
  end
`endif
endmodule
```

Figure 14

Sparc5/32MB RAM 491MB Swap Solaris version 2.5 Verilog-XL 2.2.1 - Undertow 5.3.3	Data Structure (bytes of memory)	Memory Usage Percentages	Simulation CPU Time (in seconds)	CPU Simulation Percentages
Always	88104	100.00%	1359.2	100.00%
Forever	88168	100.07%	1371.0	100.87%
Gate	88588	100.55%	2562.7	188.54%
AlwaysUT	90368	102.57%	2446.5	180.00%
ForeverUT	90432	102.64%	2481.5	182.57%
GateUT	90696	102.94%	3448.8	253.74%

Table 8

Author & Contact Information

Cliff Cummings, President of Sunburst Design, Inc., is an independent EDA consultant and trainer with 20 years of ASIC, FPGA and system design experience and ten years of Verilog, synthesis and methodology training experience.

Mr. Cummings, a member of the IEEE 1364 Verilog Standards Group (VSG) since 1994, chaired the VSG Behavioral Task Force, which was charged with proposing enhancements to the Verilog language. Mr. Cummings is also a member of the IEEE Verilog Synthesis Interoperability Working Group and the Accellera SystemVerilog Standardization Group

Mr. Cummings holds a BSEE from Brigham Young University and an MSEE from Oregon State University.

E-mail Address: cliffc@sunburst-design.com

This paper can be downloaded from the web site: www.sunburst-design.com/papers

(Data accurate as of January 4th, 2002)