Open Verilog International

Verilog HDL Conference
March 14, 1994

Writing Verilog Models for Simulation Performance, Synthesis, and other CAE Tools

Writing Verilog Models for Simulation Performance, Synthesis, and other CAE Tools

Open Verilog International
Verilog HDL Conference
March 14, 1994

prepared and presented by

Stuart Sutherland

Senior Systems Consultant
Intergraph Corporation
Boulder, Colorado

John Cooley

President and Moderator
E-mail Synopsys Users Group
Holliston, Massachusetts

Who Are We?

Stuart Sutherland

Mr. Sutherland has more than 5 years of experience using Verilog with a variety of software tools. He holds a BS degree in Computer Science, with an emphasis on Electronic Engineering, and has worked as a design engineer in the defense industry, and as an Applications Engineer for Gateway Design Automation (the originator of Verilog) and Cadence Design Systems. Currently, Stuart is employed by Intergraph, developing the "VeriBest" Verilog simulator. He has also taught Verilog at the University of California, Santa Cruz (San Jose extension). Mr. Sutherland has authored a Verilog 2.0 Quick Reference Guide and a commercially available Verilog HDL training course. For information on these books, contact "stuart@sutherland.com" or call/fax (303) 682-8864.

John Cooley

Mr. Cooley is the Founder and Moderator of the 2400 member E-mail Synopsys Users Group (ESNUG), a user driven grassroots clearing house for user discovered bugs, their workarounds and user opinion. ESNUG is completely independent of Synopsys, Inc., but Synopsys employees frequently participate in solutions and answers provided through ESNUG. John has over seven years design experience using Synopsys, Verilog, VHDL, ESDA, and static timing software before becoming an ASIC design consultant and EDA teacher. ESNUG sends out moderated e-mail bulletins on a weekly basis. To join ESNUG, e-mail "jcooley@world.std.com" (preferred) or phone (508) 429-4357.

Who Are You?

- Verilog HDL Users?
- Synthesis Users?
- “Clone” Simulator Users?
- Other Verilog Tools?

Tutorial versus Lecture

This is a tutorial

The Question

- Is it possible to write a Verilog model that:
  - Will simulate correctly?
    - Closely represents how real hardware will function
    - Approximates real hardware timing
  - and ...

The Question (continued...)

- Is it possible to write a Verilog model that:
  - Will simulate correctly?
  - And simulate efficiently?
    - Simulation can be compute intensive
    - Large models require long simulation run times
    - Large models require large amounts of system memory for simulate
  - and ...

The Question (continued...)

- Is it possible to write a Verilog model that:
  - Will simulate correctly?
  - And simulate efficiently?
  - And is synthesizable?
    - Synthesis is a key part of the top-down design paradigm
    - Synthesis tools support an “RTL subset” of the Verilog HDL
  - and ...

The Question (continued...)

- Is it possible to write a Verilog model that:
  - Will simulate correctly?
  - And simulate efficiently?
  - And is synthesizable?
  - And will synthesize correctly?
    - Synthesis output should function the same as the original RTL model
    - The structural Verilog netlist generated by Synthesis must be able to be realized in actual hardware
    - Modeling style can significantly impact the size and speed of the structural design generated by synthesis
  - and ...

The Question (continued...)

- Is it possible to write a Verilog model that:
  - Will simulate correctly?
  - And simulate efficiently?
  - And is synthesizable?
  - And will synthesize correctly?

  - And simulates with various “clone” simulators?
    - Many companies use (or will use) Verilog simulators from more that one CAE vendor
    - Models should compile with multiple simulators
    - Models should generate consistent simulation results with multiple simulators

The Answer

- Please select the most correct answer:
  - Yes
  - No
  - Maybe
  - Who Cares
  - To Be Determined

Topics to be Discussed in this Tutorial

- Simulation
  - Modeling requirements for optimum simulation performance
- Synthesis
  - Modeling requirements for accurate synthesis
  - Modeling requirements for optimum synthesis performance
- Clone simulators and other CAE tools
  - Modeling within the OVI language standards
  - Avoiding “features” peculiar to a specific CAE tool
- Getting the most out of multiple tools
  - Maintaining simulation performance with a synthesis subset
  - Other considerations

Part One

Modeling for Simulation Performance

What is Simulation Performance?

- Memory usage
  - Total virtual memory size
  - Working set size (physical memory required)
- Simulation run time
  - CPU time
  - Wall clock time

Factors Affecting Memory Usage

- Number and size of variables and signals
- Data types of signals
- Number of behavioral statements
- Number of test patterns
- Style of test patterns
- Number of hierarchical structures
- Number of gates/switches
- Number of user defined primitives
- Number of path delays
- Number of timing checks

Factors Affecting Run Time

- Number of simulation events
- Calculation of delays
- Calculation of timing checks (setup, hold, ...)
- Number of system memory lookups
- Number of system memory page faults
- Processing of internal simulation time wheels
- Usage of slower internal simulation algorithms
- Switching between internal simulation algorithms

When Should You Worry About Modeling for Best Performance?

- When you write the models
  - Every model should be written with performance in mind so that performance is never a problem later on
- Not until performance is a problem
  - Performance is not a problem most of the time
  - Modeling for performance takes too long
  - It's faster to find and fix performance bottlenecks if and when performance is a problem
- All of the above
  - Decide in advance what models will cause performance problems and model those for optimum performance
  - Many models use a relatively small percentage of the total simulation events, and so have little impact on performance

Improving Simulation Performance After Models have been Written

- Finding performance problems after models are written usually involves running simulations and analyzing the performance using software tools
  - Verilog-XL's "Behavioral Profiler" and other built-in tasks
  - Third party PLI programs
  - Your own PLI programs
- Code reviews can be used to look for performance bottlenecks without running simulations

Caution: The Simulation General has determined that fixing performance problems in working models can be hazardous to your development schedule !

Focusing on Models Likely to Cause Performance Problems

- Focus on modeling for performance with the types of models that you spend the most time simulating

_____ % of my simulations are developing and debugging behavioral models before synthesis

_____ % of my simulations are debugging structural models after synthesis

_____ % of my simulations are debugging structural models after delay calculation or back annotation

Focusing on Models Likely to Cause Performance Problems

- Focus on modeling for performance with the models will have high percentages of total simulation events
  - Clock generators and oscillators
  - Circuits that trigger on every clock cycle
  - Caches
  - Stimulus

Tips and Examples

- Most examples have been extracted from real models
  - All examples have been abridged to emphasize key points (and to fit on the overheads)
  - Names have been changed to protect the innocent
- Examples and tips in this section of the tutorial focus on simulation performance
  - Synthesis and other factors will be discussed later in this tutorial

Please point out any modeling issues when you think of them; We want lots of discussion !

Improving Performance by Reducing the Data Structure Size

- The size of the simulation data structure:
  - Determines how large of a design can be simulated on a system
  - Affects performance when the operating system must page parts of the simulation out of physical memory and into virtual memory
- Most designers do not consider memory requirements when writing models
- A reduction of 20% to 80% (up to 5x) can be achieved on typical designs
  - Is reducing the system memory requirements by a factor of 5 worth the effort?

Improving the Data Structure Size by Limiting Source File Text

- Some simulators keep source file text in the simulation data structure to aid in debugging routines ($showvars, $list, $trace ...)
- Source text stored in the data structure may be reduced by:
  - Using Verilog defaults instead of explicit statements
    - wire a; // no declaration needed
    - nand #(2,2) (y, a, b); // use #2 for delay
    - xor (strong0, strong1) (y, a, b);// no strength needed
  - Using a single statement for similar declarations
    - input a; // use input a, b, c;
      input b;
      input c;

Improving the Data Structure Size by Omitting Source File Text

- Some simulators provide methods to omit some source text from the data structure
  - `remove_gatenames will omit primitive instance names in the simulation data structure
    - Primitive instance names are not required for simulation
  - `remove_netnames will omit net identifier names in the simulation data structure
    - Net names are required if explicit port connections are used
    - Net names may be required for debugging designs
    - Net names may be required for PLI and SDF utilities

Improving the Data Structure Size by Removing Unnecessary Hierarchy

- Hierarchy formed by modules and module instances makes modeling easier, but is not always required for simulation
- The macromodule keyword allows hierarchy to be used for writing an maintaining models
  - Macromodules are flattened in the simulation data structure, removing the memory consumed by the hierarchy
  - Macromodules do not support all Verilog HDL constructs

Inefficient Model

```
module state_machine (clk,...);
  input  clk, ...;
  output ...;
  reg [7:0] state;

  always @ (posedge clk)
    case (opcode)
      ...
endmodule
```

Efficient Model

```
macromodule state_machine
(clk,...);
  input  clk, ...;
  output ...;
  reg [7:0] state;

  always @ (posedge clk)
    case (opcode)
      ...
endmodule
```

Improving the Data Structure Size by Removing Unnecessary Signals

- Sometimes engineers create signal bundles in order to make debugging easier
  - The signal bundles have no functional purpose in the design
- Use `define to create aliases to refer to several signals instead of creating a non-functional signal

Inefficient Model

```
wire [117:0]
critical_signals;
assign critical_signals =
    {data_bus, cntrl1, cntrl2,
     address_bus, enable,...};

initial
    $monitor("...",
        critical_signals);
```

Efficient Model

```
`define critical_signals
    {data_bus, cntrl1, cntrl2,
     address_bus, enable,...}


initial
    $monitor("...",
        `critical_signals);
```

Improving the Data Structure Size by Reducing Memory Arrays

- Memories are easy to model in Verilog using register arrays
  - Simulation stores the entire memory in the data structure
  - Example: Simulator "Brand X" uses 2 bits of data structure for each bit of memory model

    a 64 meg memory model = 128 meg of data structure

- Memory may be modeled so that only the active portion of the memory is stored in the data structure
  - Using PLI
  - Using the $damem dynamic memory task in Verilog-XL
  - Using mapping schemes with standard Verilog HDL to maintain only a subsection of the entire memory

    An 8 meg subsection of a 64 meg memory = 16 megabyte data structure; an 8x reduction in data structure size

Improving the Data Structure Size by Using Behavioral Stimulus or PLI

- Test stimulus may be modeled in several ways
  - As vectors in a long procedural block
    - Adequate performance
    - Poor data structure efficiency
  - As vectors in a file loaded into an array
    - Adequate performance
    - Poor data structure efficiency
  - As vectors in a file read with PLI
    - Poor performance
    - Good data structure efficiency
  - As HDL behavioral statements
    - Good performance
    - Good data structure efficiency

```
initial  begin
 #10 in = 8'b00000000;
 #10 in = 8'b00000001;
 ...
```

```
reg [8:1] V [1:`N];
initial  begin
 $readmemb("test", V);
 for (i=1; i<=`N;
i=i+1)
   #10 in = V[i];
```

```
initial
 for (i=1; i<=`N; i=i+1)
  #10
$my_pli("test",in);
```

```
initial
 for (i=1; i<=`N; i=i+1)
  #10 in = i;
```

Improving the Data Structure Size by Omitting Unnecessary Data

- Use `ifdef to only compile the data needed for each simulation

Efficient Stimulus

```
`timescale 1ns/1ns
module test;
  ...
  `ifdef debug
    task monitor;
      begin
        $monitor (...);
        $vcd_dump
(....);
        ...
    endtask
    task debug_alu;
      ...
    endtask
  `endif
  ...
endmodule
```

Efficient Model

```
`ifdef full_timing
  `timescale 1ns/10ps
`else
  `timescale 1ns/1ns
  `delay_mode_unit
  `no_timing_checks
`endif
module my_chip (....);
  ...
  `ifdef full_timing
  specify
    (in *> out) = ...
    `ifdef delay_calc
      specparam ...
    `endif
  endspecify
  `endif
endmodule
```

Improving the Data Structure Size by Reducing the Gate Count

- The functionality of several gate primitives can often be represented as one or two User Defined Primitives
  - Very useful in library cell models
  - May be useful with full custom models

- Example:
  - Simulator “Brand X” requires 300 bytes-per-gate
  - A D-Flip-Flop with Scan is modeled with 10 discrete gates
  - A complex ASIC design uses 5,000 of the Flip-Flops

5,000 X 10 X 300 bytes = 15 megabytes

  - The same Flip-Flop is modeled with 2 UDPs

5,000 X 2 X 300 bytes = 3 megabytes

Typical Data Structure Size Reductions

	Typical data structure reduction		
◆ Limiting Source File Text	1%	to	5%
◆ Omitting Source File Text	2%	to	5%
◆ Removing Unnecessary Hierarchy	5%	to	20%
◆ Removing Unnecessary Signals	1%	to	5%
◆ Reducing memory Arrays	5%	to	10%
◆ Using Behavioral Stimulus or PLI	0%	to	10%
◆ Omitting Unnecessary Data	1%	to	5%
◆ Reducing the Gate Count	5%	to	20%
Total:	20%	to	80%

- Typical memory reduction is 1.25x to 5x
  - A design requiring 64 megabytes of memory for simulation can typically be reduced to 12 to 48 megabytes

Improving Simulation Run Times

- Modeling style can have a significant effect on simulation run times
- A performance increase of 1.5x to 5x is possible on typical models

Reducing Compile and Link Times

- Keep compilation and linking in physical memory
  - Page faulting kills compile and link performance
- Minimize the size of the data structure
  - Less physical memory required
  - Less compilation and linking required
- Avoid compiling unnecessary data by using conditional compilation (`ifdef)
- Some simulators offer a choice between “interpretive” mode and “compiled” mode
  - Interpretive mode will compile and link faster (only needs to build pseudo-code and link lists)
  - Interpretive mode will simulate slower

Avoid Virtual Memory Paging During Simulation

- Page faulting kills simulation run-time performance
  - If a simulation begins to page fault, abort it !
- Minimize the size of the data structure
  - Less physical memory required
- Only part of the total data structure is required during actual simulation
  - A portion of the data structure is used for during interactive debugging, etc.
  - Example: If simulator “Brand X” uses 66% of the data structure during simulation, a 48 megabyte data structure will simulate in 32 megabytes of RAM without page faulting

Improve Simulation Run Times by Eliminating Unnecessary Events

- Simulation models often contain events that do not affect the model outputs
- The following examples show several ways to:
  - reduce the number of simulation events
  - reduce the processing overhead required to process simulation events

Improve Simulation Run Times by Only Executing Statements If Needed

- In behavioral models, statements should only be executed when there will be a change to the result
  - Example 1: The model on the left evaluates "Q" at each clock cycle, even if "data" has not changed

Inefficient Model

```
always @(posedge clock)
  Q = data;
```

Efficient Model

```
always @(data)
  @(posedge clock)
    Q = data;
```

  - Example 2: The model on the left triggers on every clock cycle, even if "reset" is asserted

Inefficient Model

```
always @(posedge clock)
  if ( !reset )
    state = next_state;
```

Efficient Model

```
always wait ( !reset )
  @(posedge clock)
    if ( !reset )
      state = next_state;
```

Improve Simulation Run Times by Combining Multiple Operations

- The results of a procedure should be determined with as few statements as possible
  - In the example below, if the operation required rotating 24 bits, the model on the left would execute 97 operations, whereas the model on the right would execute 1 operation

Inefficient Model

```
function [31:0] rotate;
 input ...
 begin
  for (i=1; i<=N; i=i+1)
   begin
    tmp = data[0];
    data[30:0] = data[31:1];
    data[31] = tmp;
   end
  rotate = data;
 end
endfunction
```

Efficient Model

```
function [31:0] rotate;
 input ...
 rotate =
   {data[N-1:0],
data[31:N]};
endfunction
```

Improve Simulation Run Times by Avoiding Null Events

- Each " ; " used as a no-op causes an event which does nothing
- Each "begin" and "end" will cause an event in some simulators
  - The model on the left executes 3 null events each time a fetch operation is performed, the model on the right has no null events

Inefficient Model

```
always @(posedge fetch)
  begin
    do_my_task;
    #100 ;
  end
```

Efficient Model

```
initial
  begin
    @(posedge fetch) do_my_task;
    #100 forever @(posedge
fetch)
      #100 do_my_task;
  end
```

Improve Simulation Run Times by Reducing Wait State Events

- Wait states and other cycle based delays should be modeled with as few events as possible
  - The model on the left processes several unnecessary events for each wait state

Inefficient Model

```
if ( cache_miss == 1 )
  begin: wait_state
    count = 0;
    while ( count <= 2 )
      begin
        @(posedge clk)
          count = count +
1;
      end
  end   // wait_state
data = ...
```

Efficient Model

```
if ( cache_miss )
  repeat (3) @(posedge clk) ;
data = data_bus;
```

or

```
if ( cache_miss )
  @(posedge clk) #(`cycle * 2)
    data = data_bus;
```

Improve Simulation Run Times by Disabling Inactive Logic

- Logic that is not currently selected is still producing events unless the logic is “disabled”
  - In the example below, the model on the left continuously performs the ALU operations, even when the ALU output is not able to change

Inefficient Model

```
always @(opA, opB, opcode)
  alu_out =
alu_function(...);

always @(idle)
 if (idle == 1’b1)
  assign alu_out = 64’b0;
 else
  deassign alu_out;
```

Efficient Model

```
always wait ( !idle )
 begin: alu_operation
  forever @(opA, opB, opcode)
   alu_out = alu_function(...);
 end

always @(posedge idle)
 begin
   disable alu_operation;
   alu_out = 64’b0;
  end
```

Improve Simulation Run Times by Only Evaluating Statements When Required

- Models that combine sequential logic and combinational logic (such as transparent latches) should only evaluate the statements required for the mode of the model
  - In the example below, The model on the left must evaluate the decision on the sequential state of the latch every time data changes

Inefficient Model

```
/* 32-bit transparent latch */

always @(gate or data)
 if (!gate) //transparent mode
  q = data;
 else       //latched mode
  q = q;
```

Efficient Model

```
/* 32-bit transparent latch */

always @(gate)
 if (!gate) //transparent mode
  assign q = data;
 else       //latched mode
  deassign q;
```

Improve Simulation Run Times by Reducing Unnecessary Operations

- Minimize the number of operations to evaluate an expression
  - Example 1: The model on the left performs an unnecessary equality comparison (plus other inefficiencies)

Inefficient Model

```
parameter TRUE = 1;
always @(posedge clk)
  if (eval_flag == TRUE)
    ...
```

Efficient Model

```
always wait (eval_flag)
  @(posedge clk)
    if (eval_flag)
      ...
```

  - Example 2: The model on the left evaluates the debug concatenation even when the result will not be displayed

Inefficient Model

```
assign debug_signals =
 {data_bus,cntrl1,cntrl2,...};

initial (if DEBUG == 1)
 $monitor("...",debug_signals);
```

Efficient Model

```
`define debug_signals
  {data_bus,cntrl1,cntrl2,...}
`ifdef DEBUG
  initial

$monitor("...",`debug_signals);
```

Improve Simulation Run Times by Using Blocking Assignments

- The Verilog HDL provides 4 types of procedural assignments
  - blocking assignment `#5 a = b + c;`
  - intra-delay blocking assignment `a = #5 b + c;`
  - non-blocking assignment `a <= b + c;`
  - intra-delay non-blocking assignment `a <= #5 b + c;`
- The blocking assignment executes in 1 step – it is evaluated and assigned at the same delta time
- The non-blocking and intra-delay assignments are executed in 2 steps, the assignment is evaluated immediately, and assigned after the delay
  - The 2-step assignments require additional simulation overhead

Improve Simulation Run Times by Reducing Data Structure Accesses

- Processing time is lost each time a variable is referenced or altered
  - Use literal values instead of variables where the value will not change during a simulation

Inefficient Models

```
integer I;
parameter N = 1024,
          INC = 4,
          Delay = 1.2;
always @(negedge clk)
 for (I=0; I<N; I=I+INC)
  #Delay ...
```

```
parameter ADD = 4'h0;
parameter SUB = 4'h1;
parameter ...
always @(opcode)
 case (opcode)
  ADD : ...
```

Efficient Models

```
reg [10:0] I;
`define N 1024
`define INC 4
`define Delay 1.2
always @(negedge clk)
 for (I=0; I<`N; I=I+`INC)
  #`Delay ...
```

```
`define ADD  4'h0
`define SUB  4'h1
`define ...
always @(opcode)
 case (opcode)
  `ADD : ...
```

Improve Simulation Run Times by Eliminating Unnecessary Output

- Monitoring simulation logic values requires considerable processing overhead
  - Use prudence in selecting the signals to be captured for output
  - Use conditional compilation (`ifdef) to only compile debug routines when required
  - Use tasks that are invoked interactively to only turn on debug routines when needed

Inefficient Model

```
initial
  $monitor("...", a, b, c);
```

Efficient Model

```
`ifdef DEBUG
task monitor;
 forever @(a or b or c)
  $display("...", a, b, c);
endtask
task no_monitor;
  disable monitor;
endtask
`endif
```

Improve Simulation Run Times by Reducing Gate Level Events

- The number of gates through which signals propagate may be reduced by using User Defined Primitives
  - One UDP can represent several gates
- Simulation of structural cell-based or full-custom designs will speed up substantially if UDP's are used in the models
  - In a test design with 10,000 Flip-Flops, simulation speed was increased by a factor of 15x by modeling the Flip-Flops as UDP's

Improve Simulation Performance by Reducing Time Wheel Overhead

- Some simulators use a time wheel to process event delays
  - Accessing the time wheel is processing overhead that increases simulation run times
  - Models should be written to achieve a high number of events for each tick in the time wheel
    - Large delays and high timescale precision result in fewer events per time wheel tick

Inefficient Model

```
`timescale 1ns/1ps
module my_chip (....);
  ...
```

Efficient Model

```
`ifdef full_timing
  `timescale 1ns/1ps
`else
  `timescale 1ns/1ns
  `delay_mode_unit
  `no_timing_checks
`endif
module my_chip (....);
  ...
```

Improve Simulation Run Times by Eliminating Timing Checks

- The evaluation of timing checks is significant simulation overhead

Inefficient Model

```
`timescale 1ns/10ps
module my_chip (....);
  ...
  specify
    (in *> out) = ...
       specparam ...
    `endif
  endspecify
endmodule
```

Efficient Model

```
`ifdef full_timing
  `timescale 1ns/10ps
`else
  `timescale 1ns/1ns
  `delay_mode_unit
  `no_timing_checks
`endif
module my_chip (....);
  ...
  `ifdef full_timing
  specify
    (in *> out) = ...
    `ifdef delay_calc
       specparam ...
    `endif
  endspecify
  `endif
endmodule
```

Acceleration Algorithms

- A simulator algorithm will process some HDL constructs more efficiently than other constructs
  - Each and every Verilog simulator has its own unique strengths and weaknesses in regards to performance
- Some simulators have special “acceleration” algorithms for certain modeling constructs
  - Acceleration algorithms often have restrictions on how constructs may be used in order to be accelerated
  - Constructs that do not meet the acceleration restrictions are simulated by a slower algorithm
- For optimum performance, models should *usually* use the most efficient algorithm

Improve Simulation Run Times by Using the Fastest Algorithm

- The Cadence "XL" algorithm "accelerates" unidirectional primitives, but it cannot accelerate primitives that:
  - Have an expression on an input
  - Have more than one output
  - Have a bit-select of a register data type on an input
  - Have an input that has been forced
  - Have a non-constant delay

Inefficient Model

```
reg  [31:0] a_reg, b_reg;
xor (y[0], a_reg[0], b_reg[0]);
```

Efficient Model

```
reg  [31:0] a_reg, b_reg;
wire [31:0] a_wire, b_wire;
assign a_wire = a_reg;
assign b_wire = b_reg;
xor (y[0], a_wire[0], b_wire[0]);
```

Note: Though this example is specific to the Cadence Verilog-XL simulator, the concept of using the most optimum algorithm applies to all simulators

Improve Simulation Run Times by Reducing Algorithm Communication

- Simulators that have multiple algorithms must switch between algorithms when events propagate across them
  - Switching between algorithms is processing overhead that increases simulation run times
  - Models should be written to use the same algorithm as much as possible
  - Example: Use a behavioral clock oscillator for behavioral models and a gate clock oscillator for gate level models

Inefficient Model

```
//behavioral clock only
always #10 clk = ~clk;


dff ( clk, d, rst, q );

always @(posedge clk)
 ...
```

Efficient Model

```
//behavioral and gate clocks
always #10 Bclk = ~Bclk;
nand #10 (Gclk, Gclk, rst);

dff ( Gclk, d, rst, q );

always @(posedge Bclk)
 ...
```

Part Two

Modeling for Synthesis

(John Cooley presented 12 pages at this point, using overhead transparencies)

Part Three

Modeling for Verilog Clone Simulators and Other CAE Tools

Modeling for Clone simulators

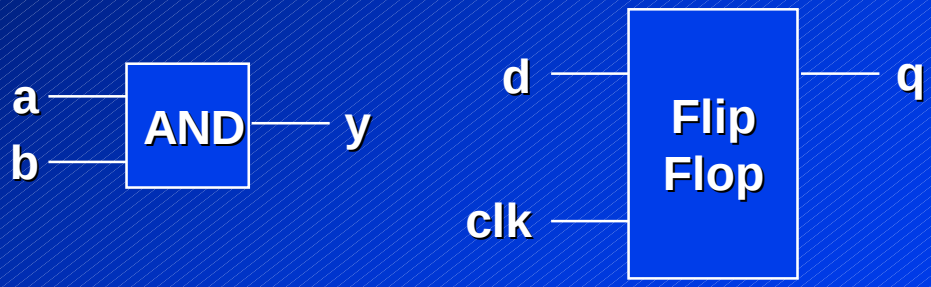
- Most (all?) existing models were written for the Cadence Verilog-XL simulator
  - Verilog-XL supports a superset of the OVI Verilog HDL standards
  - Verilog-XL has a few ~~bugs~~ "features"
- Clone simulators and other CAE tools:
  - Use the OVI Verilog HDL standards (the Language Reference Manual)
  - Have their own "features"
- To use the same model with multiple CAE tools requires:
  - Modeling within the OVI language standards
  - Avoiding "features" peculiar to a specific CAE tool

The Prime Objectives when Modeling for Multiple Simulators

- *Ideally*, the same model is syntactically correct with all simulators
  - *Reality:*
    - 100% achievable by adhering to OVI syntax
- *Ideally*, the same stimulus will work with all simulators
  - *Reality:*
    - Mostly achievable by adhering to OVI syntax
    - Each simulator will have unique commands (such as for waveform displays)
- *Ideally*, the simulation results are the same with all simulators
  - *Reality:*
    - Some clones emphasize cloning Verilog-XL results
    - Other clones emphasize results that the CAE vendor feels are more accurate than Verilog-XL
    - Modeling style must have deterministic results

Modeling for Deterministic Results

- What happens when two inputs change at the same time?




```
always @(posedge clk) d2 = d1;

always @(posedge clk) d3 = d2;
```

- Simulators can not really do simultaneous input changes
  - Changes are evaluated one-at-a-time, without advancing time
  - Different simulators will use different a order of evaluation
- Simultaneous input changes may be:
  - **Deterministic** – the result is the same for any evaluation order
  - **Non-deterministic** – the result is different with different evaluation orders
- To model for consistent results with multiple simulators
  - Use deterministic modeling styles wherever possible
  - Use timing checks on non-deterministic models

Unspecified Limits

- What is the limit for:
  - Maximum vector size?
  - Maximum port size?
  - Maximum identifier (name) lengths?
  - Maximum string lengths?
- The OVI LRM does not specify the maximum limits
  - Each CAE vendor must establish their own limits
  - Verilog-XL has very high limits
    - 1 million bit vectors and ports
    - 1024 character names
    - Unlimited strings (but must not be broken by a carriage return)
- To model for multiple CAE tools
  - Determine the limits of each tool you plan to use and stay within the smallest limit

Unspecified Sizes

- How many bits should the following numbers have?

```
integer I;    is ____ bits
time    J;    is ____ bits
`b101         is ____ bits
```

- The OVI Language Reference Manual does not specify the size for:
  - The integer and time data types
    - Verilog-XL uses 32 bits and 64 bits respectively
      - Note: The OVI LRM permits the integer and time data types to be specified with a range (i.e.: `integer [15:0] K;`)
      - Verilog-XL does not allow ranges for integer and time
  - Unsized integers
    - Verilog-XL uses the host machine word size
    - Other simulators may choose a different size
- To model for multiple CAE tools
  - Explicitly declare the size of all signals and values

Time Scaling

- The `timescale compiler directive allows software tools to:
  - Apply a time base to delays
  - Apply a precision to delays
- The OVI LRM does not specify a default time base if there is no timescale
  - Verilog-XL does not use any time base
- The OVI LRM specifies that delays should be rounded off to the precision
  - Verilog-XL rounds off using a non-conventional algorithm
- To model for multiple CAE tools
  - Always specify time scales
  - Use a time scale precision that matches the delay values

Event Timing Control Sensitivity

- Will this block trigger if both A and B are 0, then A changes to 1?

```
always @ ( A & B )
  $display("AND block triggered");
```

- Will this block trigger if both A and B are Strong1, then A changes to Weak1 ?

```
always @ ( {A,B} )
  $display("Concatenation block triggered");
```

- OVI syntax does not specify whether event sensitivity should trigger on changes to the operands of an expression or changes to the result
  - Verilog-XL triggers on changes to operands

- To model for use with multiple simulators
  - Do not use operators in event timing controls

Bit and Part Selects of Variables

- Are the following bit selects legal? Which bit is bit zero?

```
reg [0:15] A;        integer B;        parameter C = “Hello”;
Y = A[0];            Y = B[0];         Y = C[0];
```

- The OVI Language Reference Manual does not specify:
  - If bit/part selects of integer, time, and parameter data types are legal
    - Verilog-XL accepts the syntax
  - What the bit ordering should be for integer, time, and parameters
    - Verilog-XL uses little endian
- To use models with multiple CAE tools
  - Use the reg data type if bit/part selects are required

Operator Combinations

- The OVI LRM defines the XNOR operator as either ~^ or ^~
- The OVI LRM does not define a NAND operator pair.
  - Are ~& and &~ both legal operator pairs?
    - Verilog-XL will only accept ~&
    - Some clone simulators will accept both operator pairs
- The OVI LRM does not define a NOR operator pair.
  - Are ~| and |~ both legal operator pairs?
    - Verilog-XL will only accept ~|
    - Some clone simulators will accept both operator pairs
- To model for use with Multiple CAE tools
  - Only use operator pairs ~& and ~| for NAND and NOR operations

Delay Propagation on Strength Changes

- The OVI LRM provides for 8 strength levels for logic 0, 1, and X
- The LRM does not specify if a strength change should be delayed by gate delays
  - Verilog-XL applies gate delays to strength changes
- The LRM does not specify if a strength change should be delayed by path delays
  - Verilog-XL does not apply path delays to strength changes
- To model for use with multiple CAE tools
  - Use gate delays if strength changes need propagation delays

Non OVI System Tasks and Functions

- The OVI LRM provides the $<keyword> syntax which allows CAE tool vendors to add system tasks and functions specific to their tool
  - OVI specifies about 50 “$” system tasks and functions
    - $display(), $monitor(), $dumpvars(), $setup(), ...
  - Verilog-XL supports over 200 “$” built-in tasks and functions
    - $displayb(), $displayo(), $displayh(), $monitorb(), ...
    - $gr_waves(), $random(), $memory(), $damem(), ...
  - Most clone simulators support the OVI standard “$” tasks and functions plus a subset of the Verilog-XL tasks and functions
- To model for multiple CAE tools
  - Use the system tasks and functions defined by OVI
  - Determine which tasks are available in *all* tools you plan to use

Non OVI Compliant Syntax "Features"

- The Cadence Verilog-XL simulator permits a few syntactical constructs that are in violation of the OVI LRM
  - Verilog-XL permits multiple arguments to parameter assignment

```
parameter delay  = (1,2,3);
specparam foobar = (1.2, 2.2, 3.1);
```

  - Verilog-XL permits path delay assignments with no parentheses

```
(in *> out) = 1.8, 2.5, 4.1, 0.0, 1.0, 1.0 ;
```

  - Verilog-XL permits duplicate names in port lists

```
module dff (q, clk, clk, d);
  output q;
  input  clk, d;
```

- To use models with multiple CAE tools
  - Avoid using any non OVI compliant syntax

Special Functionality Not Specified by OVI

- The Cadence Verilog-XL simulator supports a super-set of the OVI Verilog HDL language constructs
  - The Verilog-XL “switch-xl” algorithm adds additional functionality, such as 255 strength levels
  - The Verilog-XL glitch (pulse) control alters how input glitches affect simulation outputs
- To model for use with multiple CAE tools
  - Avoid using constructs proprietary to one product

Non OVI Compliant PLI procedural calls

- The OVI PLI Reference Manual specifies standard procedural calls to interface with Verilog simulations
- PLI version 1.0 is what Cadence released to the public domain
  - Cadence did not release the complete set of procedural calls that were supported by Verilog-XL at that time
  - Cadence has added procedural calls to Verilog-XL since releasing PLI 1.0
- PLI 2.0 is the revised procedural calls specified by OVI in 1993
  - Extensive changes from PLI 1.0
  - Not widely adopted by most CAE tool vendors at this time
- To use PLI programs with multiple CAE tools
  - Avoid using any non OVI compliant PLI 1.0 procedural calls
  - Avoid using PLI 2.0 procedural calls until more tools support 2.0

Part Four

Modeling for Simulation Performance,
Synthesis, and other CAE tools;
All in the same model

Trade-offs when Modeling for Data Structure Efficiency

	Run Time?	Synthesizabe?	Portable?
◆ Limit or omit source file text using defaults	improves	yes	yes
◆ Remove hierarchy by using macromodules	improves	yes	yes
◆ Remove unnecessary signals using compiler directives	improves	yes	yes
◆ Omitting data using `ifdef	improves	yes	yes
◆ Reduce memory size using compiler directives	improves	no	???
◆ Using PLI for test vectors	degrades	yes	???
◆ Using behavioral stimulus	improves	yes	yes
◆ Reduce gate count with UDP's	improves	???	yes

Trade-offs when Modeling for Run-time Performance

	Synthesizabe?	Portable?
◆ Removing null events	no	yes
◆ Only execute procedures when the output will change	no	yes
◆ Combine multiple operations to the same output	no	yes
◆ Reduce number of operations for wait states	no	yes
◆ Reduce number of events with disable	???	yes
◆ Only evaluate signals when they change	no	yes
◆ Reduce number of access to data structure	yes	yes

Trade-offs when Modeling for Run-time Performance (continued)

	Synthesizabe?	Portable?
◆ Use non-blocking procedural assignments	no	no
◆ Minimizing simulation monitoring	yes	yes
◆ Use fastest algorithm	???	yes
◆ Reduce communications between algorithms	???	yes
◆ Reduce time-wheel overhead	yes	yes

Trade-offs when Modeling for Synthesis

	Performance?	Portable?
◆ Avoid non RTL constructs	degrades	yes
◆ Separate combinational and sequential logic	degrades	yes
◆ Partition logic into functional blocks (use macromodules for performance)	???	yes
◆ Only change outputs from 1 procedure	degrades	yes
◆ Specify all branches of decision statements	degrades	yes
◆ Specify details of architecture (i.e.: loops for barrel shifter instead of concatenation)	degrades	yes

Trade-offs when Modeling for Portability

	Performance?	Synthesizable?
◆ Model for deterministic results	degrades	yes
◆ Model within limits of all tools	no affect	yes
◆ Explicitly declare variable sizes	no affect	yes
◆ Avoid operators in sensitivity lists	degrades	yes
◆ Avoid bit selects of variables	no affect	yes
◆ Avoid illegal operator combinations	no affect	yes
◆ Avoid non OVI syntax and constructs	degrades	yes

The Question (again)

Is it possible to write a Verilog model that:

- Will simulate correctly?
- And simulate efficiently?
- And is synthesizable?
- And will synthesize correctly?
- And simulates with various “clone” simulators?

The Answer

- Please select the most correct answer:
  - ❑ Yes
  - ❑ No
  - ☑ Maybe
  - ❑ Who Cares
  - ❑ To Be Determined

- Improving performance by reducing data structure size:
  - ☺ Is generally compatible with synthesis
  - ☺ Is generally compatible with model portability

- Improving performance by reducing simulation events:
  - ☹ Is not compatible with synthesis
  - ☺ Is generally compatible with model portability