TauSim 4.0
User Guide

Tau Simulation Incorporated
46560 Fremont Blvd, Suite 115
Fremont, CA. 94538
(510) 623-0800

Contents

1. Introduction

TauSim is a Verilog simulator that provides extremely high performance verification for multi-million gate, synchronous integrated circuit designs. TauSim's algorithms are optimized for performance and its data structures are optimized for size. As a result of TauSim's compressed data structures, the simulation footprint for a multi-million gate design is very small. Consequently, the number of cache misses during simulation is reduced, resulting in additional performance benefits.

TauSim provides support for Verilog's 0, 1, X, and Z values. Due to TauSim's proprietary algorithms, X and Z are handled extremely efficiently. As a result, TauSim's 4-value and 2-value simulations achieve similar performance. However, 2-value mode is supported to provide compatibility with traditional cycle-based simulators.

TauSim is a multi-level Verilog simulator. It provides support for models at the RTL, gate, and transistor levels. Additionally, C models can be linked in through TauSim's support of Verilog's PLI capability. Any TauSim simulation can be run with models at different levels of abstraction to enable designers to optimize performance/accuracy tradeoffs.

TauSim provides PLI support for integrating testbenches and VCD support for integrating analysis tools.

TauSim is available for SPARC processors running UNIX and Pentium processors running LINUX. Compiling for TauSim is architecture independent. As a result, a design can be compiled on either architecture and the resultant image can be simulated on both.

TauSim supports a synthesizable subset of Verilog. The supported capabilities are those which are most likely to be used in synchronous designs and are described in detail in this manual. However, TauSim has been architected to be easily extensible and is continually being enhanced. If TauSim does not provide support for a capability required for your design, let us know.

2. Recommended Design Style

TauSim is ideally suited for fully synchronous designs. In particular, TauSim models a design with all delays set to zero and requires that all feedback paths within a design are clocked through flip-flops.

Additionally, the Verilog design description must use only the syntax that is supported within this release. The synthesizable subset of Verilog that is supported by TauSim is fully enumerated within this manual.

3. Data Types

3.1 Values

TauSim can be run in either 4-value or 2-value mode. In 4-value mode, TauSim supports the standard Verilog signal values: 0, 1, X, and Z. In 2-value mode, TauSim supports only the signal values 0 and 1.

2-value mode can simulate somewhat faster than 4-value mode. The difference in performance is based on the design of the circuit and the extent to which the circuit signal values are unknown. Normally there will not be a large performance difference between 4-value and 2-value modes. 2-value mode will also require less memory for modeling large memories. In 2-value mode, TauSim will use approximately one

bit of run-time memory to model each bit of a simulated memory. In 4-value mode, TauSim will use approximately 2 bits of run-time memory to model each bit of a simulated memory.

In 2-value mode, TauSim uses 0 or 1 to represent X and Z values, and properly models tri-state buses, with the exception that buses with no active drivers, or with multiple drivers outputting differing values will be assigned arbitrary values.

3.1.1. High Accuracy X Handling

TauSim uses X propagation rules that model hardware behavior more accurately than standard Verilog. In TauSim, **if**, **case**, and **casex** statements set all outputs to X if any significant bit in the select expression is X. For example, in Verilog, the following RTL inverter:

```
always @ (i)
        if (i)
                o = 0;
        else
                o = 1;
```

will set o = 1 if i = X. However, in TauSim, o = X if i = X, preventing the loss of the unknown state and ensuring that driven circuits do not receive misleading values.

In the following example:

```
always @ (a)
        casex (a[2:0])
                3'b1xx: b = 1;
                3'b001: b = 2;
                default: b = 3;
        endcase
```

The TauSim behavior will be:

If a[2] = 1, then b = 1, regardless of the value of a[1:0].

If a[2] = X, then b = X, regardless of the value of a[1:0].

If a[2:0] = 001, then b = 2.

If a[2:0] = 00X, 0X0, or 0XX, then b = X.

Otherwise, b = 3;

This makes the X behavior of RTL simulations match more closely with gate level simulations and hardware behavior. This enhanced level of accuracy allows more circuit initialization problems to be detected at the RTL level.

3.2 Strengths

TauSim provides support for the 8 strength levels available within Verilog simulations. These strength levels - supply, strong, pull, large, weak, medium, small, highz - are used to resolve conflicts between two or more drivers of different strengths on the same net.

TauSim provides support for the Verilog charge storage strengths - large, medium, small - for active use with resistive devices. However, TauSim does not support **trireg** nets or their use of the charge storage strengths. This capability will be added in a future release.

3.3 Nets

For nets, TauSim supports the **wire** and **tri** data types.

tri0, **tri1**, **triand**, **trior**, and **trireg** nets are not supported in the current release.Registers

3.4 Registers

TauSim provides support for the **reg** data type.

3.5 Vectors

TauSim provides support for vectors for nets and registers.

3.6 Memories

TauSim provides support for the Verilog standard of modeling memories as arrays of registers.

3.7 Integer, Real, and Time

TauSim supports integers only at compile time when they are used as **for** loop indexes.

TauSim does not support **real** or **time** registers. However, the t**ime** capability can be achieved with standard registers.

3.8 Parameter

TauSim supports a subset of the **parameter** statement. Parameters can be assigned and used within a module, where they are treated as constants, but they may not be modified with the **defparam** statement or through a module instance parameter value assignment. Subranges of parameters are not supported.

The following code **is** supported:

```
module plus(sum, a, b);
parameter size = 8;
output [size-1:0] sum;
input [size-1:0] a, b;
        always @ (a or b)
                sum = a + b;
endmodule
```

However, the following code **is not** supported:

```
module top;
wire [15:0] x, y, z;
wire [3:0] d, e, f;
        plus #(16) p0(x, y, z);
        plus p1(d, e, f);
        defparam top.p1.size = 4;
endmodule
```

A parameter may not be subscripted. The following code **is not** supported:

```
module width;
parameter bus1 = 16'b1101000101010101;
wire [3:0] g;
        assign g = bus1[11:8];
```

```
endmodule
```

3.9 Strings

TauSim provides support for strings.

4. For Loops

The **for** statement supports an integer index, incrementing or decrementing by a constant amount between a constant start and stop condition. TauSim expands **for** loops at compile time, treating the index as a constant within the loop. This supports nested **for** loops, with indexes of outer loops allowed as boundaries or increments of inner loops.

The following code **IS** supported:

```
for (i = 0; i < 64; i = i + 8)
        a[i] = b[i];
for (i = 5; i >= 2; i = i - 1)
        c[i] = d[i];
```

However, the following code **is not** supported:

```
for (i = 0; a[i] != 0; i = i + 1)
        a[i] = b[i];
for (i = 1; i < 16; i = i * 2)
        c[i] = d[i];
```

Since **for** loops are expanded at compile time, care must be taken to not use extremely large loops, for example to clear a large memory, to avoid a very large simulating design.

5. Keywords

5.1 Supported Keywords

TauSim supports the following Verilog keywords. If the support is not fully featured, the scope of the support provided is annotated in parentheses.

Table 1: Supported Keywords

always	**and**	**assign** (outside of procedural blocks)	**begin**
buf	**bufif0**	**bufif1**	**case**
casex	**casez**	**cmos**	**default**
else	**end**	**endcase**	**endfunction**
endmodule	**endspecify**	**endtask**	**for**
function	**if**	**initial**	**inout**
input	**integer**	**macromodule**	**module**
nand	**negedge**	**nmos**	**nor**
not	**notif0**	**notif1**	**or**
output	**parameter**	**pmos**	**posedge**
pulldown	**pullup**	**rcmos**	**reg**
rnmos	**rpmos**	**scalared**	**specify**
supply0 (as net type)	**supply1** (as net type)	**time**	**tri**
vectored	**wire**	**xnor**	**xor**

5.2 Unsupported Keywords

TauSim does not provide support for the following Verilog keywords. If partial support is provided, the type of support not provided is annotated in parentheses.

Table 2: Unsupported Keywords

assign (inside of procedural blocks)	**deassign**	**defparam**	**disable**
edge	**endprimitive**	**endtable**	**event**
force	**forever**	**fork**	**highz0**
highz1	**ifnone**	**join**	**large**
medium	**primitive**	**pull0**	**pull1**
real	**realtime**	**release**	**repeat**
rtran	**rtranif0**	**rtranif1**	**small**
specparam	**strong0**	**strong1**	**supply0** (as drive strength specifier)
supply1 (as drive strength specifier)	**table**	**tran**	**tranif0**
tranif1	**tri0**	**tri1**	**triand**
trior	**trireg**	**wait**	**wand**
weak0	**weak1**	**while**	**wor**

6. Operators

TauSim provides support for all Verilog operators. The operators, the operation performed, and the number of operands are listed below. For two operators, === and !==, TauSim provides limited support. Operator === performs the same function as operator == and operator !== performs the same function

as operator !=.

Type of Operator	Operator Symbol	Operation Performed	Number of Operands
Arithmetic	* / + - %	Multiplication Division Addition Subtraction Modulus	Two Two Two Two Two
Logical	! && ||	Logical negation Logical and Logical or	One Two Two
Relational	> < >= <=	Greater than Less than Greater than or equal to Less than or equal to	Two Two Two Two
Equality	== != === !==	Equality Inequality TauSim implements as equality TauSim implements as inequality	Two Two Two Two
Bitwise	~ & | ^ ^~ or ~^	Bitwise negation Bitwise and Bitwise or Bitwise xor Bitwise xnor	One Two Two Two Two
Reduction	& ~& | ~| ^ ^~ or ~^	Reduction and Reduction nand Reduction or Reduction nor Reduction xor Reduction xnor	One One One One One One
Shift	>> <<	Right shift Left shift	Two Two
Concatenation	{ }	Concatenation	Any number
Replication	{ { } }	Replication	Any number
Conditional	? :	Conditional	Three

7. Initial Blocks

TauSim supports 2 different types of **initial** blocks:

standard **initial** blocks

fast **initial** blocks

7.1 Standard Initial Blocks

Standard **initial** blocks support the same features as non-posedge procedural blocks, and can be used in the same places. Code is generated for these **initial** blocks. Delay statements can be used, but they must be unconditionally executed.

The following code **is** supported:

```
initial begin
        #10;
        if (a)
                b = 1;
        #10;
        c = 2;
end
```

However, the following code **is not** supported:

```
initial begin
        #10;
        if (a) begin
                b = 1;
                #10;
        end
        c = 2;
end
```

The following tasks are not supported in standard initial blocks:

$monitor

$dumpvars

$dumpfile

$dumpon

$dumpoff

7.2 Fast Initial Blocks

At most, one fast **initial** block is allowed in a design, and it must be located at the end of the top-level module. Code is not generated for the fast **initial** block. Instead, TauSim builds stimulus tables and interprets tasks such as **$dumpon** during simulation. This allows simulations with a large amount of input stimulus to run at higher speed than would be possible with a standard **initial** block. It also allows adding VCD file controls without modifying the simulating code, so simulations with and without VCD files will simulate identically, with the same order of evaluation.

If a fast **initial** block is used, it is distinguished from a standard **initial** block by the addition of the following line comment:

```
initial          // TauSim fast_initial
begin
                 a = 1'b1;
end
```

The following statements are supported in the fast **initial** block:

$monitor

$dumpvars

$dumpfile

$dumpon

$dumpoff

In addition, TauSim supports the following capabilities in the fast initial block:

- Assignment of a binary, matching-specified-width constant to a reg
- Delay statements

8. Clocking

8.1 Clocks

TauSim supports multiple clocks, as long as they are derived from the same oscillator.

TauSim provides a built-in oscillator primitive, which defines the base cycle of the simulation. The oscillator clock signal must be specified in the top-level module of the design using the "tausim_osc" special TauSim primitive as follows:

```
wire osc_clock;

tausim_osc optional_instance(osc_clock);
```

The oscillator clock signal must be of type **wire**, and must be undriven.

All other clocks are derived from positive edges of the oscillator clock with normal Verilog logic gates and flip-flops. Each clock rises and falls as a normal logic signal, but in a glitchless manner, and special propagation rules are used to prevent skew between clocks in complex designs.

8.1.1. Stages

The Verilog language is capable of specifying very complex clocking schemes, and TauSim defines a general-purpose synchronous subset which can be statically analyzed at build time and optimized for high-performance simulation.

All clocking in TauSim originates from the oscillator clock. Rising edges of the oscillator clock, either directly or indirectly, initiate transitions on user clocks, and therefore on user logic signals. TauSim counts the number of flip-flop clock-to-output paths between the oscillator clock and each user logic signal. Any flip-flop clocked directly by the oscillator clock is specified to be a "stage 1" flip-flop. Any logic signal generated using logic gates or combinatorial RTL code solely from "stage 1" flip-flop outputs is specified to be a "stage 1" logic signal. Any flip-flop clocked by a "stage **N**" logic signal is specified to be a "stage **N**+1" flip-flop.

All stage **N** flip-flops clock before any stage **N**+1 flip-flops, and all flip-flops within a stage clock simultaneously.

TauSim supports blocking and non-blocking assignments in **posedge** and **negedge** blocks for modeling flip-flops and memories. Procedural flip-flops should be modeled as follows:

```
always @ (posedge clock)
        c <= b;

always @ (posedge clock)
        b <= a;
```

In the example above, register "c" will always be loaded with the value of "b" from before the rising edge of the clock, and never with the value of "b" from after the rising edge of the clock.

When blocking assignments are used in **posedge** or **negedge** blocks, delays are not allowed. This is done to ensure that the TauSim results and Verilog timing simulation results will match. The ordering of reads and writes in blocking assignments between different **posedge** or **negedge** blocks within the same stage is not specified. The following code will not behave in a predictable manner, and should not

be used:

```
always @ (posedge clock)
        c = b;

always @ (posedge clock)
        b = a;
```

In the example above, register "c" could be loaded with either the value of "b" from before the rising edge of the clock, or the value of "b" from after the rising edge of the clock, which had been loaded from "a".

Operations within **posedge** and **negedge** blocks are performed in the following order:

- All logic signals generated from stage **N**-1 and below flip-flop outputs
- ---------- begin stage **N posedge** or **negedge** block ----------
- Blocking assignments and right-hand-side of non-blocking assignments,all in linear order
- Left-hand-side of non-blocking assignments
- ---------- end stage **N posedge** or **negedge** block ----------
- All logic signals generated from stage **N** and below flip-flop outputs

Very complex clocking schemes can be supported by TauSim using this approach, including ripple counters, clock gating, and multiple identical clocks. Flip-flops in a given stage, written with non-blocking assignments, will all appear to clock at the exact same time, without skew, regardless of the number of levels of combinatorial logic used to gate their clocks.

Structural flip-flops should be modeled using the special TauSim "tausim_d_flop" primitive as follows:

```
tausim_d_flop instance_name(q_output, clock, d_input);
```

It is recommended that the oscillator clock be used only to clock a state machine that generates user clocks, rather than clocking any user flip-flops directly. In a single-clock design, a simple toggle flip-flop can be clocked by the oscillator clock, and the output can be used to clock the user design.

8.1.2. Non-edge Always Blocks

TauSim supports blocking assignments within non-edge **always** blocks. Non-blocking assignments are not supported in non-edge **always** blocks. For non-edge **always** blocks, the sensitivity list is assumed to be complete. Every term read within the block which is not written in the block should be present in the

sensitivity list. Since TauSim requires feedback paths to be broken by a register, and since sensitivity list terms are treated as inputs to a block, no term which is present in the sensitivity list should be assigned to inside the block.

8.2 Asynchronous Loops

TauSim provides very high simulation performance by simulating only the logical behavior of a design, without maintaining any timing information within a cycle. For a design to simulate in TauSim, all feedback paths must pass through a flip-flop. If the output of a gate is used in determining any input to the gate in the same cycle, TauSim will consider this to be an asynchronous loop, and will print an error message. The following example shows an asynchronous loop:

```
nand n1(a, b, c);
nand n2(b, a, d);
```

Since “b” drives into “n1”, which produces output “a” which drives into “n2”, this is a loop and is not supported by TauSim.

The rules about asynchronous loops also apply to procedural blocks. For loop-detection purposes, all inputs to the block are considered to be used in determining the value of each output of the block. The following example shows a loop in procedural code:

```
always @ (a or b)
        c = a & b;

always @ (c or d)
        a = c & d;
```

Since “a” is used to determine “c”, in the first block, and “c” is used to determine “a” in the second block, a loop is formed.

Because all inputs to a procedural block are considered to be used to determine each output of the block, some loops can occur which are caused by the way the Verilog code is written, rather than by the design of the logic gates which would be used to implement the function.

The following example shows this type of loop:

```
always @ (a or b or d or e)
begin
```

```
                c = a & b;
                f = d & e;
end

always @ (f)
                a = f;
```

Since “f” is an output of the first block, and is used to determine “a” in the second block, and since “a” is an input to the first block, TauSim will report a loop. As the code is written, the first block needs to be executed once before the second block, to determine “f” from “d” and “e”, and once after the second block, to determine “c” from “a” and “b”. TauSim requires that each block be executed once per cycle, so it treats this as an error. The same design can be rewritten for TauSim as follows:

```
always @ (a or b)
                c = a & b;

always @ (d or e)
                f = d & e;

always @ (f)
                a = f;
```

As written above, each block can be executed once per cycle, and TauSim will not report an error.

8.3 Cycle Boundaries

TauSim simulates each base cycle as an uninterruptable operation. At the end of a base simulation cycle, TauSim has performed all assignments within stage-2-and-above **posedge** and **negedge** blocks and stage-2-and-above structural flip-flops. TauSim has also performed all blocking assignments in all stage-1 **posedge** and **negedge** blocks, but has not performed any non-blocking assignment in any stage-1 **posedge** or **negedge** block or stage-1 structural flip-flop.

At this point in the base cycle, TauSim will output VCD file values, sample signals for the **$monitor** statement, and perform any assignments specified in the fast **initial** block. Assignments specified in a standard Verilog **initial** block occur early in the base cycle, after the non-blocking assignments in stage-1 **posedge** and **negedge** blocks and stage-1 structural flip-flops are performed.

Because of this sequencing of **initial** block assignments, fast **initial** block assignments will not override

non-blocking assignments made in stage-1 **posedge** and **negedge** blocks and stage-1 structural flip-flops. However, standard Verilog **initial** block assignments will override non-blocking assignments in stage-1 **posedge** and **negedge** blocks and stage-1 structural flip-flops.

9. Path Names

TauSim provides support for path names which originate within the referencing module and may continue down into instantiated modules. Path names may extend downward through multiple levels of hierarchy. To access data types in other areas of the design hierarchy, explicit port references must be used.

10. PLI Support

TauSim supports PLI tasks and functions for integration of C models as well as for testbenches.

It is necessary to specify the binding of PLI task and function names as used in Verilog to the C routine names by using the "`pli_def`" function. One call to "`pli_def`" is needed for each Verilog-accessible PLI function and specifies the following information:

Verilog name

C name

Input vs. output specification for ports

During initialization, TauSim automatically executes the "pli_tab" function provided by the user, which contains the user "pli_def" calls.

Since TauSim statically levelizes the functions in the Verilog code, it needs to know which ports are inputs and which ports are outputs. For built-in Verilog functions, this is implicit, but for user PLI code, it must be specified by the user.

The third input to "pli_def" is a string containing the letters 'r' and 'w'. One character is used for each PLI function parameter, with the same ordering as the Verilog ports. An 'r' indicates the parameter will not be written (with tf_putp) by the PLI code. A 'w' indicates the parameter will be written. If the string has fewer characters than the number of Verilog ports, all remaining parameters are assumed to be of type 'r'.

If READONLY data is passed into a PLI task or function port that is specified as type 'w', a fatal error is reported. If a "tf_putp" write is done to a parameter of type 'r', the write proceeds normally, but the write will be ignored when levelizing is done, possibly causing a 1 cycle delay before the new data is seen at some or all places it is used.

For Verilog PLI calls made inside a **posedge** block, the I/O specification is not used, since the clocked nature of these blocks implies a specific ordering.

Example:

```
void pli_tab()

{
```

```
        pli_def("$here_i_am",pli_path,"");
        pli_def("$writeit",pli_write,"rw");
        pli_def("$complete",pli_finish,"");
        pli_def("$reset",pli_reset,"rrw");
}
```

Since sensitivity list information is processed statically, PLI functions and system tasks which are not in an **initial** block will be invoked every cycle, rather than only when the terms in the sensitivity list actually change. If these functions are not desired to be called at certain times, they should be placed inside Verilog **if** statements. Since TauSim does some synthesis, an X value going into an **if** statement selector may activate tasks slightly differently than in standard Verilog. Contact Tau Simulation if you have any questions.

A parameter is treated as READWRITE only if the entire structure is passed. A subscripted location of an array or a bit subrange is treated as READONLY.

TauSim supports the following PLI utility routines:

tf_nump()

tf_typep(nparam)

tf_mipname()

tf_gettime()

tf_dofinish()

tf_synchronize()

tf_putp(nparam, value)

tf_getp(nparam)

tf_strgetp(nparam, format_char)

11. System Tasks

11.1 Outside of Fast Initial Blocks

TauSim supports the following tasks outside of fast **initial** blocks:

$display - %[h,H,d,D,o,O,b,B,t,T] only

$displayb - %[h,H,d,D,o,O,b,B,t,T] only

$displayo - %[h,H,d,D,o,O,b,B,t,T] only

$displayh - %[h,H,d,D,o,O,b,B,t,T] only

$write - %[h,H,d,D,o,O,b,B,t,T] only

$writeb - %[h,H,d,D,o,O,b,B,t,T] only

$writeo - %[h,H,d,D,o,O,b,B,t,T] only

$writeh - %[h,H,d,D,o,O,b,B,t,T] only

$fdisplay - %[h,H,d,D,o,O,b,B,t,T] only

$fdisplayb - %[h,H,d,D,o,O,b,B,t,T] only

$fdisplayo - %[h,H,d,D,o,O,b,B,t,T] only

$fdisplayh - %[h,H,d,D,o,O,b,B,t,T] only

$fwrite - %[h,H,d,D,o,O,b,B,t,T] only

$fwriteb - %[h,H,d,D,o,O,b,B,t,T] only

$fwriteo - %[h,H,d,D,o,O,b,B,t,T] only

$fwriteh - %[h,H,d,D,o,O,b,B,t,T] only

$fopen

$fclose

$time

$finish

$readmemb - No addresses or starting offset

$readmemh - No addresses or starting offset

11.2 Inside of Fast Initial Blocks

TauSim supports the following tasks inside of fast **initial** blocks:

$dumpvars - Level and module specification only

$dumpfile

$dumpon

$dumpoff

$monitor - %[h,H,b,B] only

12. VCD Files

TauSim can generate VCD files to be used with waveform viewers or other analysis tools. Since TauSim

samples signal values only once per base cycle, the oscillator clock signal, which changes twice during each base cycle, will appear to have a constant value of 0.

13. Compiler Directives

13.1 Supported Directives

TauSim supports the following compiler directives:

‘default_nettype	(only wire and tri supported)
‘define	(formal arguments not supported)
‘else	
‘endif	
‘ifdef	
‘include	
‘timescale	(all ‘timescale in a design must be identical)
‘undef	

13.2 Unused Directives

TauSim accepts but does not use the following compiler directives:

‘celldefine

‘endcelldefine

‘nounconnected_drive

‘resetall

‘unconnected_drive

13.3 Timescale

TauSim recognizes the **‘timescale** compiler directive. If more than one **‘timescale** directive is present in a design, all **‘timescale** directives present must specify the same values. The period of the oscillator clock may be specified by appending a line comment containing “tausim clock_period” and a time specification to the line of the “tausim_osc” statement as follows:

```
tausim_osc dc(clock); // TauSim clock_period 1 ns
```

If this comment is present, and the **‘timescale** directive is also present, then TauSim will interpret **initial** block delays in the **‘timescale** units and will use the **‘timescale** units in the VCD file and for **$time** return values. Otherwise, TauSim will treat a delay of one unit as one clock cycle. Time values in TauSim are limited to 32 bits of precision.

Since TauSim applies stimulus only on cycle boundaries, all **initial** block delays must be an integer multiple of the clock period.The following code **is** supported:

```
‘timescale 1 ns / 1 ps

tausim_osc dc(clk); // TauSim clock_period 100 ns

initial begin
        a = 0;
        #100;
        a = 1;
        #300;
        a = 2;
end
```

The #100 will be a 1-cycle delay, and the #300 will be a 3-cycle delay.

The following code **is not** supported:

```
‘timescale 1 ns / 1 ps

tausim_osc dc(clk); // TauSim clock_period 100 ns

initial begin
        a = 0;
        #150;
        a = 1;
```

```
                #30;
                a = 2;
        end
```

The #150 represents a delay of 1.5 cycles, and the #30 represents a delay of 0.3 cycles, which are not supported.

14. Compilation and Execution

TauSim can be operated in three modes: “compile and run”, “precompile”, and “postcompile run”.

The “compile and run” TauSim reads the Verilog code, including an optional fast **initial** block, and runs the simulation for the number of cycles specified in the fast **initial** block. VCD files can be generated and **$monitor** can be used. This mode is intended for design debugging simulations.

The “precompile” TauSim reads the Verilog code and writes an intermediate design file. Any fast **initial** block is ignored. This is done because these functions currently require symbol table information, which is omitted from the intermediate file and subsequent simulations to save space. Future releases of TauSim will improve this.

The “postcompile run” TauSim reads the intermediate design file and simulates until either:

a tf_dofinish() is executed, or

a **$finish** is executed, or

1 billion cycles have passed

All control and I/O for the simulation must be done through PLI calls, system tasks, and standard **initial** blocks. This mode is intended for batch mode simulations.

The precompilation and postcompilation modes are processor independent, so both SPARC and Pentium architectures can be used interchangeably. PLI, **$display**, **$write**, etc., work the same way in pre/post mode as in “compile and run” mode. The PLI C code is linked into all 3 TauSim executables, but in precompile mode, only the “pli_tab” information is used. The “pli_tab” portion must be identical in the pre and post TauSim executables, since the intermediate file uses the ordering of the “pli_def” calls to bind Verilog and C names. The remaining PLI C code can be compiled and linked into a taupst executable without relinking taupre or rebuilding the intermediate design file. If Makefiles are used, it is advisable to put the “pli_tab” function into a separate source file from the other PLI C code.

Precompilation can be done for either 4-value or 2-value mode. A 4-value precompilation file can be used with either a 4-value or a 2-value postcompilation TauSim. A 2-value precompilation file can be used only with a 2-value postcompilation TauSim.

15. Running TauSim

The command line options to TauSim are slightly different depending on whether tausim, taupre, or

taupst is being run. If tausim, taupre, or taupst is run without arguments, it will print a brief description of the options. The following are some ways to invoke TauSim:

```
tausim file.v

tausimb -O0 file.v

tausim -s1=5000000 file.v

taupre -o=file.cmp file.v

taupst file.cmp

tausim top.v -f options.f +incdir+includes+

tausimb a.v b.v c.v -v lib1.v -v lib2.v

tausim xyz.v -y lib_dir +libext+.v+.hv+

tausimb vsrc.v +define+TAUSIM+
```

15.1 Optimization Control

TauSim normally performs performance optimizations during the compile process. The command line option “-O0” turns off these optimizations.

15.2 Command File Options

TauSim supports the “-f file” option for use of command files. The specified file is searched for TauSim options, which are processed as if they were part of the command line. The file may contain comments and the options may be separated by blank spaces, tabs, or newlines. Mutilple “-f” options may be used, and command files may use the “-f” option to access other command files.

15.3 Include File Options

TauSim supports the “+incdir+directory+” option for use of include files within the Verilog code. When a **‘include** compiler directive is encountered with an unrooted pathname, the working directory is searched for the specified file. If the file is not found, then each include directory is searched, in the order specified by the “+incdir+” options, until the file is found. If the include file is not present in any of these directories, a fatal error is reported. Multiple include directories may be specified with multiple “+incdir+” options. The final “+” is optional.

15.4 Library Options

TauSim supports the “-v file” option for accessing library files. The specified library file is read after the explicitly specified source files, and the library file is searched for any modules which have been refer-

enced but not yet defined. Multiple library files may be specified through the use of multiple “-v” options. If a module is defined multiple times, the first definition encountered is used.

The “-y directory” and “+libext+extension+” options are supported for accessing library directories.The “+libext+” option causes TauSim to search through the library directory most recently specified by the “-y” option for any files whose names end in the specified extension. Each matching file is used as a library file as specified above for the “-v” option. Multiple extensions may be specified through a combination of multiple “+libext+” options, and multiple extensions may be specified within an individual “+libext+ext1+ext2+ext3+” option. The final “+” is optional.

15.5 Define Options

TauSim supports the “+define+macro+” option for defining macros. The specified macro is set to have a defined value, which can be tested with the Verilog **‘ifdef** compiler directive. Multiple “+define+” options may be used. The final “+” is optional.

15.6 Miscellaneous Options

The “-s#=#” options are included to support very large designs. TauSim allocates some internal storage elements when it first starts running, and it is possible to overflow these if the design is very large. If an overflow occurs, the size option number and current setting is printed to stderr, along with an error message. As long as enough virtual memory is present, simply rerunning with the size set larger should solve the problem. These options are generally not required unless the design is larger than 2 million gates.

15.7 Unused Options

TauSim provides a warning message for any “`+option+`” options which are not supported.

16. Environments Supported

TauSim runs on SPARC processors running Solaris, and Pentium processors running Linux. Contact Tau Simulation if you have another type of machine you would like to use with TauSim. We are planning to port to other architectures and operating systems based on user demand.

17. Release 4.0.0

This release includes 3 object files:

tausim.o	- Compile and run - 4-value mode
tausimb.o	- Compile and run - 2-value mode

taupre.o - Precompile - 4-value mode

taupreb.o - Precompile - 2-value mode

taupst.o - Postcompile and run - 4-value mode

taupstb.o - Postcompile and run - 2-value mode

The “b” suffix indicates binary, 2-value mode.

A default PLI object file is included, and can be overwritten when user PLI code is used:

pli.o - Default-PLI object

In addition, the following files are included:

tausim - Default-PLI executable - 4-value mode

tausimb - Default-PLI executable - 2-value mode

pli.c - Default-PLI C code

pli.h - PLI C header

test.v - Sample Verilog code

test.out - Output from sample

An executable is built with the command:

```
cc tausimb.o pli.o
```

Usually, the “sim”, “pre”, or “pst” indicator is included in the executable name, so this one would be called “tausimb”. If there is no user PLI, the included “pli.o” file can be used.

18. License Files

TauSim uses a license file containing authorization keys matching specific computers. The license file will usually be delivered through email or a separate floppy disk. The license file must be installed and TauSim must be notified of the license file path name. TauSim accesses the license file by the UNIX environment variable:

```
$TAUSIM_LICENSE
```

Before running TauSim, this environment variable must be set to the path name of the license file. For the “C” shell, this can be done with “setenv”, something like the following command:

```
setenv TAUSIM_LICENSE ~tausim/key_file
```

Each line in the license file represents an authorization key for one computer. When additional keys are delivered, these new lines must be added to the license file.

Index