Verilog According to Tom

1.0 What's Verilog?

The name Verilog refers to both a language and a simulator which are used to functionally specify and model digital systems. This document describes Verilog in the context of producing RTL models of hardware, especially hardware which will subsequently be implemented.

Hopefully this document along with some example Verilog code provide what most students need to master Verilog syntax, semantics, and good coding practice, leaving the Verilog reference manuals to be reference manuals.

1.1 The Verilog Language

Verilog HDL (Hardware Description Language) was concocted by Gateway Design Automation and later put in the public domain by Cadence Design Systems in order to promote the language as a standard.

Verilog models look like programs. Descriptions are partitioned into Verilog *modules*. Modules resemble subroutines in that you can write one description and use (instantiate) it in multiple places.

You can assemble modules hierarchically. Lower-level modules will have inputs and outputs which syntactically look like procedure parameters. The higher-level module instantiates them and connects their input and output ports with Verilog "wires" in a syntax that looks like a procedure call.

The lowest modules in the hierarchy, and possibly others, will have descriptions of functionality. Both declarative and procedural descriptions look like C-language statements with C-like expression operators, but with different meaning for the "variables".

```
// Verilog Example, an SR-latch made from two nand gates
// This description has no delays, so it won't actually work,
// but it shows how modules are put together.

module nand(in1, in2, out);
input in1, in2;
output out;

assign out = ~(in1 & in2);

endmodule
```

```
// This module instantiates and “hooks up” two “nand” modules

module srlatch(s, r, q, q_b);
input s,r;
output q, q_b;

nand nand1(s, q_b, q);
nand nand2(r, q, q_b);

endmodule
```

1.2 The Verilog Simulator

Cadence Design Systems sells *Verilog-XL*, a simulator for the *Verilog HDL* language. *Verilog-XL* compiles and runs a system’s modules either interactively or in batch mode. Special waveform and state displays are available. Section 5.0 contains details and hints for running *Verilog-XL*.

If the simulated system spans several files, *Verilog-XL* can assemble it regardless of the order the files are specified in. *Verilog-XL* compiles the entire system on each invocation, so there are no intermediate “object” files nor is there an explicit link phase. The compilation step is quite fast, not at all in the way of getting things done.

Cadence restricts our use of *Verilog-XL*, so it will only run on particular machines by the grace of various key files. Contact the authorities for details.

1.3 Of What Use is Verilog?

This document considers Verilog a hardware modeling tool. Producing a Verilog model should be one part of your design process. If your code is an end in itself, you’re reading the wrong thing, and somebody else will have to tell you what Verilog is worth!

1.3.1 Exploring Options

At best, your Verilog model will be your design’s most easily manipulated abstraction among those that both correspond to hardware and are executable. It makes sense to use Verilog to explore design options at a high level while keeping in sight the approximate hardware implications of your choices. Synthesis tools can vastly improve the quality of these approximations.

1.3.2 Simulating and Verifying Functionality

You can execute your Verilog model, so you can gain confidence that your system’s design is functionally correct. Functional simulation ferrets out many errors in complex systems.

1.3.3 Generating Functional Test Vectors

The Verilog system should be the earliest executable model of your design, so it can save tedious work by automatically producing functional test vectors for subsequent simulations at lower abstractions or even the completed hardware.

1.3.4 Input for Downstream Tools

By “working”, the Verilog model becomes a good specification. Verilog models can specify functionality to synthesis tools like Synopsys, connectivity to routers, and so on.

2.0 An RTL HW-Modeling Verilog Subset

The Verilog language and simulator have a lot of features, but a small number of constructs are sufficient for RTL hardware modeling. Many exotic Verilog features like tasks, functions, and named blocks obscure the connection between the functional model and its implementation in hardware. Others, like primitive gates and support for a complete MOS switch-level system are too low-level to manipulate easily.

2.1 Values

Disregarding the signal-strength system that supports switch-level MOS, all nodes in a Verilog model will have one of four values: 1, 0, X (undefined/error), or Z (high impedance). If the node has multiple drivers, they combine as in FIGURE 1..

	0	1	X	Z
0	0	X	X	0
1	X	1	X	1
X	X	X	X	X
Z	0	1	X	Z

FIGURE 1. Value Combinations

Verilog defaults to decimal constants. Binary, octal, decimal, and hexadecimal constants can be expressed, but only binary constants may contain X or Z. The most general constant syntax is a string with a size in bits, an apostrophe, a radix (b, o, d, or h), and the constant. Examples:

```
100      // Decimal 100, 32 bits by default
6'h3a    // Binary 111010
1'bx     // One-bit X
32'bz    // 32 bits of High-Z
```

Code targeted for synthesis by Synopsys should have sizes on all constants. The sizes will prevent Synopsys from generating lots of warnings and from synthesizing unnecessary hardware by assuming constants are 32 bits.

2.2 Objects

2.2.1 Nodes

The verilog *wire* data type corresponds to a circuit node. The *wire* keyword itself appears in declarations. Verilog assumes that undeclared identifiers are names for one-bit wires.

Verilog has other data types similar to wire which indicate wired logic. They're declared *wand*, *wor* and *tri*, you shouldn't need them. Synopsys will occasionally output *tri*, but it's a synonym for *wire*.

2.2.2 Ports

The inputs, outputs, and bidirectional ports of modules are either *wire* or *reg* objects. They can be vectors, and are declared with the *input*, *output*, and *inout* keywords.

2.2.3 Reg

Reg data types are referenced like wires, but a *reg*'s value is whatever was most recently procedurally assigned to it. You create state in Verilog modules by selectively assigning to *reg* objects.

A *Reg* can be a vector, and can also have an integer subscript.

2.3 Hierarchy (Modules)

Modules are the basic units in Verilog models. They contain functional descriptions and have input, output, and bidirectional ports for interfaces. A module can contain instantiations of other modules. The instantiation syntax looks like a procedure call, complete with parameter renaming for port connections.

The following are true of module interfaces:

- An *input* or *inout* port is a *wire* within its module.
- An *output* port must be a *wire* if it is generated by a submodule.
- An *output* port must be a *wire* if it is generated declaratively.
- An *output* port must be a *reg* if it is assigned to procedurally.

Bidirectional ports cannot be assigned to procedurally. A method for working around this limitation is described in section 3.2.3.

Modules tend to partition into leaf modules and connection modules, where leaf modules contain procedural (*always*) and declarative (*assign*) functionality, and

higher modules connect them together by instantiating them and connecting their ports with *wires*.

2.4 Expressions

Expressions in a C-like syntax do the combinational logic in Verilog. The operators are C-like logical and arithmetic operators, and the operands are *wires*, *regs*, or constants.

2.4.1 Logic

Bitwise and logical operators look like their C counterparts. There are also logical reduction operators.

Comparisons can have two different semantics in the presence of X values. The == and != operators return X if either argument is X, while the === and !== operators can compare to the value X itself.

Verilog includes the C conditional expression (question mark - colon) construct.

2.4.2 Arithmetic

All arithmetic is signed. The carry can be recovered if the result is one bit larger than the largest operand, see the example in section 2.4.3.

Keep in mind that arithmetic can be expensive! Grouping of terms in expressions can affect how well synthesis tools can share resources. Synopsys will not even synthesize divide or modulus operations.

2.4.3 Concatenation

Verilog has a unique syntax for vector concatenation. The fields are comma-separated between braces. An integer preceding the brace repeats the result. Examples:

```
word[31:0] = {short1[15:0], short2[15:0]};
{cout, result[7:0]} = byte1[7:0] + byte2[7:0] + cin;
   // OK, OK, it's not really an expression ...
word[31:0] = {24{byte[7]}, byte[7:0]};
   // sign-extend "byte"
```

2.5 Declarative Descriptions (assign)

The *assign* construct expresses functionality declaratively. Syntactically, the keyword *assign* precedes one or more assignments. The left side of each is a wire, and the right hand of each is an expression.

In Verilog nomenclature, the result is a *continuous assignment*. The *assign* asserts that the assigned-to wire is driven with the value of the right-hand expression at all times, with a delay if any is specified.

Assign constructs always express combinational logic. Examples:

```
assign sum[4:0] = a[3:0] + b[3:0];
assign a = x & y, o = x | y;
```

2.6 Procedural Descriptions (always)

The *initial* and *always* constructs express functionality procedurally. Both cause the execution of a Verilog statement, but *initial* executes the statement only once while *always* executes the statement repeatedly.

A Verilog statement can be compound, with multiple statements in a *begin-end* block. Simple statements are assignments to *reg* variables, flow control, (*if*, *case*), or system tasks. You won't have many system tasks in your code unless you're debugging.

2.6.1 Timing Control

All of the *always* blocks in a system can be thought of as separate, concurrent processes. Each has a "program counter" which starts at the beginning of the block, advances to the end, and restarts at the beginning. Multiple statements in a block are executed one at a time, so that the side-effects of statements are available to subsequent ones, but simulated time advances only at timing-control points.

Any statement can be proceeded by a timing-control construct. When Verilog encounters one, execution of statements in that *always* block is suspended until the event control is satisfied. Simulated time will only advance when event control is blocking execution of the *always*. This example has no event control at all, which causes the simulator to hang:

```
always
   begin
      sum = a + b;
      diff = a - b;
   end
```

There are two common types of timing control, delay controls and event controls. Delay controls suspend execution in their *always* blocks for the specified number of time units. Event controls suspend execution until one of the listed signals changes. Most of your *always* blocks will have single, initial event control lists as their only timing control.

```
// 50% duty cycle clock, 20 time unit period
always
   begin
      clock = 0;
      #10   // Delay control - 10 units
      clock = 1;
      #10
      clock = 0;
   end

// set x to be a xor b; combinational logic
always @(a or b)// Event control - only continue
            // when a changes or b changes
   begin
      x = a | b;
      if (a & b)
         x = 0;
   end
```

2.6.2 Flow Control

Verilog provides *if-else* and *case* for flow control. The *case* statement also comes in variants *casex* and *casez*, which treat X and Z specially.

The *if* condition can be a single-bit *reg* or *wire*, or an expression. X or Z will cause the *else* arm of the *if* to execute.

Example?

2.6.3 State

Assignments in *always* blocks are made to *reg* variables. The *reg* retains the last value it was last assigned, so if an *always* block conditionally assigns to a *reg* between timing controls, state is implied. Section 3.2.1, for example, shows how to build latches and flip-flops with this mechanism.

The interaction of timing control and flow control indirectly implies state, so the most general use of timing control can hopelessly obfuscate the functionality of your model. The coding guidelines of section 4.5 are highly recommended in order to avoid deadly, subtle errors.

3.0 How to Model Stuff

3.1 Assign vs. Always

A functional object that needs state requires specification via *always*, and driving an *inout* port requires an *assign*, but a lot of combinational logic could be specified with either mechanism. Often the choice is a matter of taste, but each method does have advantages.

Assign is less error prone. Section 4.5 explains in detail how innocent-looking *always* blocks can have unintended meaning.

Always is often less tedious to work with. Logic can be embedded in code that already must exist to create latches and flops. Sequential assignments and/or case statements express some logic more cleanly than conditional expressions can.

```
wire d;
reg p;
// The following two lines each accomplish the
// same thing
assign d = a & b;
always @(a or b) p = a & b;
```

3.2 Common Primitive Objects

3.2.1 Flops and Latches

Model your flip-flops and flow latches with *always* blocks. A reg variable holds the state, and flow and timing controls regulate updates.

For flow latches, be sure to model transparency. The latch inputs should appear on the event control list.

You can combine several latches or flops in the same *always* block, but flops must all have the same clock.

```
reg Qff, Ql;
reg [31:0] a, b;

// Positive edge-triggered D flip flop
always @(posedge clk)
   Qff = D;

// D flow latch
always @(clk or D)
   if (clk) Ql = D;

// Two 32-bit latches
always @(phi1 or x or y)
   if (phi1)
      begin
         a = x;
         b = y;
      end
```

3.2.2 Muxes

Conditional expressions naturally form 2-input muxes. For a higher number of inputs, muxes will be easier to express procedurally. If the selects aren't fully

decoded and you intend to synthesize, be careful of implied priority as described in section 4.5.

```
assign muxout = select ? input1 : input2;
```

3.2.3 Buses

Any Verilog wire can have multiple drivers. You can specify a “high-impedance” output for an object driving a bus by driving the Verilog value Z. Expressions in both continuous assignments and *always* blocks can generate the value Z, but remember that module *inout* port wires must be driven by a continuous assignment. If it’s easier to compute the value you want to drive procedurally, you will need an intermediate reg and a continuous assignment to drive it. All three of these approaches accomplish the same thing:

```
module adder(phi1, bus, loada, loadb, drive);
input phi1, load, drive;
inout[7:0] bus;

reg [7:0] opa, opb;

// Two load-qualified latches to hold operands
always @(phi1 or bus or loada or loadb)
   if (phi1)
      begin
         if (loada) opa = bus;
         if (loadb) opb = bus;
      end

// You can do it this way -

assign bus = drive ? opa + opb : 8’bz;

// Or this way -

reg [7:0] sum;
assign bus = drive ? sum : 8’bz;
always @(opa or opb) sum = opa + opb;

// Or this way -

reg [7:0] busout;
assign bus = busout;
always @(opa or opb or drive)
   busout = drive ? opa + opb : 8’bz;

endmodule
```

Remember that even for buses, the module port can be an *output* rather than an *inout* if the value isn’t used as an input in that module because pure *outputs* can also be driven with a Z.

3.3 State Machines

There are a lot of ways to make state machines. This section describes a methodical way of writing state machine descriptions. The resulting code and hardware are hopefully easy to understand and modify.

First, decide whether you want a Moore or a Mealy machine. The organization of the code will differ for the two machine types.

FIGURE 2. illustrates which type of machine is which. A Moore machine's outputs

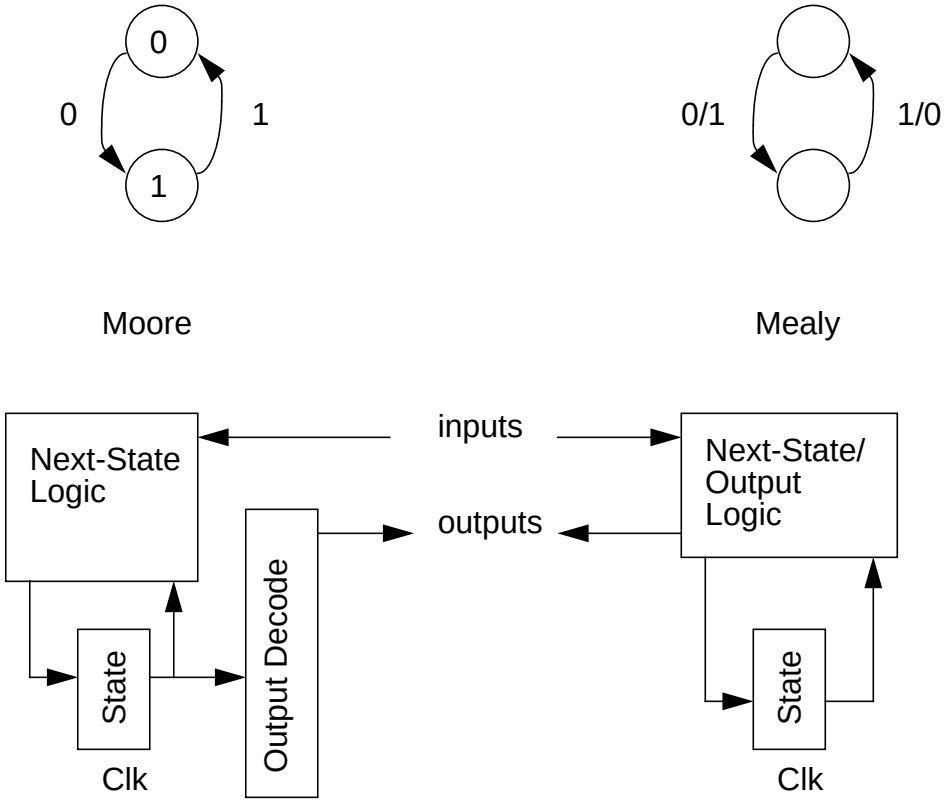

FIGURE 2. Moore and Mealy Machines

depend only on its state, while a Mealy machine's outputs depend on its state and inputs. A Mealy machine typically has fewer states than a Moore machine for the same function. System timing constraints are less obvious for systems containing Mealy machines.

Represent the state with one register, either a flop or a master-slave latch pair. Use the preprocessor or parameters to give the states symbolic names. Case statements in *always* blocks work well for specifying next-state logic and decode logic. If you want to make a Moore machine, it may be a good idea to keep the next-state logic and the output decode logic in separate *always* blocks. The following example is deliberately long-winded for what it accomplishes, in order to illustrate the desired coding style.

```
// Mickey-mouse parity machine
// (Master-slave latches for state)

reg state_s1, state_s2;

// Slave state latch
always @(phi1 or state_s1)
   if (phi1) state_s2 = state_s1;

// Master state latch & next-state logic
always @(phi2 or in or state_s2)
   case (state_s2)
      'ODD:
         if (in)
            state_s1 = EVEN;
         else
            state_s1 = ODD;
      'EVEN:
         if (in)
            state_s1 = ODD;
         else
            state_s1 = EVEN;
   endcase

// Output logic
always @(state_s2)
   case (state_s2)
      'EVEN: out = 0;
      'ODD: out = 1;
      // We want combinational logic here, so:
      default: out = 1'bx;
   endcase
```

3.3.1 Synopsys State-Machine targeting

If you want a single edge-triggered state machine, (you probably don't,) there is a way to specify the state encoding with the Verilog *parameter* mechanism and some pseudo-comment directives for the benefit of the FSM Compiler in Synopsys. The ordinary Synopsys Design Compiler will synthesize state machines as they are described, but the FSM Compiler can manipulate state encodings in order to get better answers.

4.0 Coding Style

A uniform coding style may seem unnecessarily restrictive, but there are many advantages. Other people will understand your code more easily, a big plus if you are part of a team or even if you ever need some help. Certain good habits can protect you from entire classes of errors. Finally, tools other than the Verilog simulator that accept Verilog as input may restrict what the code can look like.

4.1 General Guidelines

- Write one module per file. You can find stuff easily, multiple people can work on one system, the design can be sent piecemeal to other tools, and it's easy to swap subsystems.
- Don't write modules as primitive as "latch", "mux", and "driver", they are as easy to express in Verilog when needed as they are to instantiate.
- Beware of describing hardware that cannot be started. Initial blocks have no hardware counterpart, nor do comparisons to X (===, !==, ??????casex)

4.2 Documentation

Documentation - you know, comments! (Ugh!)

- Write a large comment near the header of each module that describes its context. People will be more able to find their way around your design hierarchy.
- Use vertical or horizontal whitespace to keep code and comments apart.
- As usual, point out subtleties

4.3 Targeting Synopsys

Synopsys accepts a subset of Verilog. Aside from following the guidelines in this section:

- Bit field indices cannot be variables, Synopsys won't synthesize the required shifter nor can it determine the width of the result.

If you are synthesizing more than random/control logic:

- Synopsys cannot synthesize memories or dividers.
- You will save hardware by manually grouping common subexpressions.

4.4 Naming Conventions

- Have one module per file, and name the file and the module the same thing.
- Often, a module has exactly one instantiation in a system. If so, name the module and the instantiation the same thing.
- Formal/Actual name mapping can take place in every module instantiation. Make every effort to use the same name for a given circuit node everywhere in the system's hierarchy. Unless modules have multiple instantiations, you will be able to do this.

4.4.1 Identifier Suffixes

Choose a convention for labeling every node identifier with a suffix which describes assertion level, clock timing-type, and other good information (pipeline stage names, for example) both to avoid mistakes and to aid documentation.

When all identifiers have suffixes, inspection or automatic tools can help to find errors and inconsistencies.

4.4.2 Abbreviations

When you name something, avoid abbreviating words to less than three letters. Everybody may know what an ALU is, but there are a lot of common words that want the same one or two-letter abbreviations. What abbreviations you use, use uniformly.

Suggestions:

Register	reg
Address	addr (not add or adr)
Drive	drv
Result	res
Bus	bus
Immediate	imm
Load	load (not ld)
Store	store (not st)

4.5 Procedural Pitfalls

Don’t mix delay timing control with event timing control, the meaning of your code will quickly evade you. Each *always* block should have a single event control list at its beginning.

The statements in an *always* block execute in sequence. The sequencing often implies priority that you didn’t intend, but synthesis tools will build unnecessary hardware, not knowing any better.

Procedural Verilog semantics allow for the possibility of some very subtle problems. Incomplete event control lists and unintentional implication of latches cause a lot of bugs. The real dangers of both of these common errors are that both lead to models which appear to simulate correctly but synthesize incorrectly and that both can be very difficult to casually notice.

4.5.1 Incomplete Event Control Lists

Always blocks that specify combinational logic and/or flow latches should execute any time that any input changes. If an input is missing from the event control list, that input will have unwanted state behavior, as if it were connected through some strange sort of flop or latch.

Every value referenced in a combinational or flow-latch *always* block should be included in the event control list. Referenced values include:

- Operands in expressions
- *If* and *case* arguments
- *Reg* subscripts on the left-hand side of an assignment

A subscripted *reg* should be in the event control list with the same subscript it is referenced with. If it is used with multiple subscripts, they should all be on the list. Example:

```
always @(foo or memaddr or reg[aspec] or
            reg[bspec] or aspec or bspec)
   begin
      if (foo)
         mem[memaddr] = reg[aspec];
      else
         mem[memaddr] = reg[bspec];
      cspec = aspec | bspec;
```

4.5.2 Unwanted Implicit State

A *reg* keeps the value of its last assignment until it is assigned to again. If a *reg* is assigned to on some path of execution through an always block but not on all paths, it behaves as a latch. Make sure that non-latch *reg* variables are assigned to through every path - both arms of *ifs*, and all arms of *cases*. A *case* should have a default even if all possible inputs match some label. You may want to assign a default value (perhaps X) to variables early in the always block.

5.0 Running the Simulator

The Verilog simulator runs on the L&IR Sparcstations (the elaines) in Sweet Hall.

Simply invoke Verilog with all of your source files as command-line arguments, and they will be compiled and run. If you have specified a waveform analyzer, it will appear. You may have to send a Ctrl-C to make the simulator pause the execution of your model and give you a prompt.

From the prompt, you can send a Ctrl-D to exit the simulator, or begin using Verilog interactively as described in section 5.2. With execution paused, you can also use the analyzer.

5.1 System Tasks

You can insert calls to special system tasks in blocks of procedural code. All system task names begin with the character $.

Some system tasks “execute” fully as they are encountered. *$display* and *$stop* are examples of such tasks, which you would use in *always* blocks. Others like *$monitor* and *$gr_waves* instantiate processes which continue to run after they are executed. You would probably put calls to such tasks in *initial* blocks.

- **$display**
  $display corresponds to printf in C. The first, optional argument is a format string, like C except that “%h” is hexadecimal, not “%x”, and there are formats “%t” for times and “%m” for hierarchical names.

- **$stop**
  When Verilog encounters a *$stop*, it pauses as if you sent a Ctrl-C.
- **$finish**
  Verilog exits as if you sent Ctrl-D when it encounters *$finish*.
- **$monitor**
  Verilog instantiates a process which, for all subsequent time, prints its arguments according to the optional format string as in *$display* whenever one of those arguments changes in the course of simulation.
- **$time**
  $time is not a system task, but a system variable which can be especially useful as an argument to *$display* or *$monitor*.
- **$gr_waves**, **$gr_addwaves (see**
  Both of these tasks instantiate processes which set up waveform analyzer traces and update them during subsequent simulation. Section 5.3 has more on the analyzer.

5.2 Debugging in Interactive Mode

The command-line argument “-s” will cause Verilog to begin in interactive mode rather than immediately starting execution. You may want just that if your model is screwing up early on.

5.2.1 Seeing Out-of-Scope Nodes

Verilog has a mechanism and syntax for accessing values from lower scopes than the current module. Such access would be a bad practice for modeling, but it can be very useful for instrumenting and debugging, especially with *$gr_waves*.

5.2.2 Interactive Mode

Things you type at the interactive prompt execute as if they were in a sort of *initial* block. There are several Verilog constructs that are handy in interactive mode, and there are some system tasks intended specifically to be used interactively.

- **$scope**
  You can set the current default scope, analogously to “cd” in Unix, with the *$scope* command.
- **$showscopes**
  $showscopes prints all of the modules, tasks, and functions in the current scope.
- **$showvars**
  $showvars displays the status of Verilog “variables”. *$showvars* is useful for debugging multi-driver busses, because it is the one way to see what the individual drivers are *trying* to put on the wire, rather than the overall “X” you usually see.

- **$list**
  $list prints the source text for the current scope, annotated with the current values of variables.
- **,**
  A comma will step the execution through a single statement.
- **.**
  A period will resume execution to the next *$stop* or Ctrl-C.
- **$db_help**
  Shows a long list of debugger options that allows you to set breakpoints much like the C-debugger. For instance,

  ```
  $db_breakonposedge(clock);
  ```

  will cause a break on every cycle of clock.

Verilog interactive mode allows the *forever* construct, which is like an *always* construct. The usual timing controls are available, so you can set breakpoints in many ways:

```
#10000 $stop;                 // Stop after 10,000 time units
forever @(posedge clk) $stop; // stop at each clock cycle
forever @(opcode)
   if (opcode==72) $stop;     // stop when “opcode” becomes 72
```

Break conditions could possibly be quite complex.

5.2.3 Viewing the waveform

During the interactive mode, the waveform of the signals can still be viewed. The next section describes two methods of viewing the waveform.

Gr_waves can be applied through the command line. Existing *gr_wave* commands already in the code will display the appropriate signals as you step through.

Simwave, however, is a bit more difficult to use. Since *simwave* loads the data through the “dump” file, you must first ensure that the latest information is entirely dumped to file. This can be forced by typing on the command line:

```
$dumpflush;
```

To load the results with the new information, the *File->Setup->Restore* option is very handy once you save the setup.

5.3 The Waveform Analyzer

There are several methods of displaying a waveform in Verilog. The first section describes *gr_waves*. For our Verilog environment, *gr_waves* is not available. However a more user friendly tool can be used through Cadence called *simwave*. The second portion of this section describes how to use *simwave*. A third option is to use a run-time debugger. This feature is not currently supported.

5.3.1 gr_waves

If Verilog executes a *$gr_waves* task, it opens a graphics window and keeps it updated with the history of the values of the variables you want to monitor.

History prior to the *$gr_waves* call does not get recorded, so *$gr_waves* calls are almost always in initial blocks.

Only one *$gr_waves* call is allowed. More signals can be added to the analyzer with the variation *$gr_addwaves*. *$gr_addwaves* might be useful interactively, if old values are not of interest.

You will often want to access out-of-scope values with the analyzer, so it makes sense to put all of your $gr_waves and $gr_addwaves calls in an initial block in the topmost module in your system, where all nodes are accessible.

```
// Somewhere in a top-level module...
initial
   begin
      $gr_waves(“Phi1”, phi1, “Phi2”, phi2);
      $gr_addwaves(“T”, datapath.tbus, “S”, datapath.sbus);
      $gr_addwaves(“Reset”, reset);
      $gr_addwaves(“bogus signal”, interface.controller.foo);
   end
```

When the simulator is not executing, you can use a mouse to manipulate the waveform display. The first and second buttons move and swap the two cursors, the third moves the time axis. A menu of pushbuttons allows zooming, among other things.

5.3.2 simwave

Simwave is a post-processing tool that can read the dumped output of Verilog. You can launch simwave by typing at the command prompt:

```
>> simwave
```

Simwave will open a display window. The menus are fairly self explanatory. The *File->Database->Load* allows you to load a new Verilog dumped output which is the *vcd* format. Verilog can also generate the *shm* format but it is not as convenient to use. *File->Setup->Save/Restore* allows you to save a particular setup so that you can reload it next time you startup *simwave*. *Tools->Browser* allows you to open a browser for selecting the signals to display. Play around or use their Help to get familiar with the tool.

To generate the dump output from Verilog, there are several commands that are useful within the Verilog code.

```
$dumpfile(<filename>);
$dumplimit(<filesize>);
$dumpflush;
$dumpall;
```

The first instantiates a file to dump. Otherwise, it defaults to *verilog.dump*. The second limits the dump filesize to avoid runaways. The third makes sure all data in the buffer are stored in file before proceeding. The last gives a checkpoint at the instance in time it is called.

```
$dumpon;
```

```
$dumpoff;
```

Stops or resumes the recording of data values.

```
$dumpvars(<level>,<module|variable>....);
```

```
$dumpvars(2,count,clkmod, trafmod);
```

IMPORTANT, <level> refers to the depth that it can probe if you give it a module name. Every register/wire inside that heirarchical depth is logged. Otherwise, to save space you can specify a specific variable. Typically, I use the modules to save time. As an example,

```
// Somewhere in a top-level module...
initial
   begin
      $dumpfile("outfile");
      $dumpvars(2, datapath, datapath.sbus);
   end
```