Revised version of an invited paper to be published in the proceedings of the *Tenth Annual IEEE Symposium on Logic in Computer Science (LICS'95)*, **June 26-29, 1995, San Diego, California.**

The Semantic Challenge of Verilog HDL

Mike Gordon
Computer Laboratory
University of Cambridge
Cambridge, CB2 3QG, U.K.

April 11, 1996

Abstract

The Verilog hardware description language (HDL) is widely used to model the structure and behaviour of digital systems ranging from simple hardware building blocks to complete systems. Its semantics is based on the scheduling of events and the propagation of changes. Different Verilog models of the same device are used during the design process and it is important that these be 'equivalent'; formal methods for ensuring this could be commercially significant. Unfortunately, there is very little theory available to help.

This self-contained tutorial paper explains the semantics of Verilog informally and poses a number of logical and semantic problems that are intended to provoke further research. Any theory developed to support Verilog is likely to be useful for the analysis of the similar (but more complex) language VHDL.

Contents

1 Introduction

Modern hardware description languages enable the designer to mix different levels of design abstraction. The lowest level is a connection of gates (netlists), which may be generated manually or automatically as the output of synthesisers. The next level contains structures such as counters, multipliers etc. The highest level is 'behavioural' and uses programming constructs such as assignments, conditionals and `while`-loops. A common approach is to first build and test a prototype using behavioural constructs. As the design matures, modules which were first specified behaviourally are recoded in a subset of the HDL from which hardware can be synthesised automatically.

VHDL and Verilog are the two most widely used languages in industry. In the academic formal methods community VHDL is much better known and many people (e.g. me until quite recently) have barely heard of Verilog, even though it has been estimated that there are 25,000 Verilog designers today, with 5,000 additional students trained in Verilog graduating each year [4]. Verilog is employed by designers in numerous companies including Sun Microsystems, Apple and Hewlett-Packard. An industry survey recently found that in 1995 Verilog was getting 66 per cent of business and VHDL 34 per cent [3]. As a language Verilog has much in common with VHDL, however its programming constructs are based on C, whilst those of VHDL are based on Ada.

Verilog is taught to second year computer science undergraduates studying at Cambridge University as part of their hardware laboratory work. It is hoped (suitable theory permitting) that it will eventually come to underlie a third year course on the specification and formal verification of hardware.

I have chosen to work with Verilog primarily because of its role in teaching at Cambridge, but also because it is simpler (though less general) than VHDL.

The structure of the rest of the paper is as follows. Section 2 is an introduction to Verilog aimed at readers with a logical and semantic background. Some of the behavioural subtleties of the language that would need to be handled by a formal semantics are discussed. Section 3 specifies a subset of Verilog intended as a vehicle for semantic experiments. Section 4 is an informal semantics of the selected subset. This is intended to provide a self-contained reference for future formalization. Finally, Section 5 describes some problems that are of theoretical interest and practical utility.

2 Overview of Verilog

A specification in Verilog consists of one or more *modules*. The *top level module* specifies a closed system containing both test data and hardware models. It is what is executed by Verilog simulators. Component modules will normally have input and output *ports*. Events on the input ports cause events on the outputs. Events can either be changes in the values of *wires* or *registers*, or can be explicitly generated abstract events. Modules can represent bits of hardware ranging from simple gates to complete systems (e.g. microprocessors), they can either be specified *behaviourally* or *structurally* (or a combination of the two). A behavioural specification defines the behaviour of a module using programming language constructs. A structural specification expresses a module as a hierarchical interconnection of submodules. At the bottom of the hierarchy the components must either be *primitives* or specified behaviourally. Verilog's library of predefined primitives will not be discussed here.

2.1 Simple combinational examples

Here is a behavioural specification of a module `NAND`: the value output on port `o` is the negation of the conjunction of the value input on ports `i1` and `i2`.

```
module NAND (i1,i2,o);
 input i1, i2; output o;
 assign o = ~(i1 & i2);
endmodule
```

The ports `i1`, `i2` and `o` are wires. The symbols `~` and `&` denote negation and conjunction, respectively. The *continuous assignment* `assign o = ~(i1 & i2)` continuously watches for changes to variables in its right hand side (`i1` and `i2` in this example) and whenever a change happens the right hand side is re-evaluated and the result immediately propagated to the left hand side (`o` in the example).

Here is the structural specification of a module `AND_IMP` obtained by connecting the output of one `NAND` to both inputs of another one.

```
module AND_IMP (i1,i2,o);
 input i1,i2; output o; wire w;
 NAND NAND1(i1,i2,w);
 NAND NAND2(w,w,o);
endmodule
```

This structure has two instances of `NAND` (called `NAND1`, and `NAND2`), connected together by an internal wire `w`. The behaviour implied by this structure is expressed directly in the definition of the module `AND`:

```
module AND (i1,i2,o);
 input i1, i2; output o;
 assign o = i1 & i2;
endmodule
```

Verilog is used not only to specify hardware devices but also to specify test data. The module `AND_TEST_DATA` generates the inputs `i1=0` and `i2=0`, `i1=0` and `i2=1`, `i1=1` and `i2=0`, `i1=1` and `i2=1` at successive times.

```
module AND_TEST_DATA (i1,i2);
 output i1,i2; reg i1,i2;
 initial begin    i1 = 0;    i2 = 0;
               #1            i2 = 1;
               #1 i1 = 1;    i2 = 0;
               #1            i2 = 1;
         end
endmodule
```

The module `AND_TEST_DATA` has no inputs and two outputs `i1` and `i2`. Inside the module definition, the outputs are declared to be registers. Registers are variables that 'remember' the last value that was *procedurally assigned* to them (just like variables in imperative programming languages). Wires are the default kind of variable; they have no storage capacity. They can be continuously driven (e.g. with a continuous assignment or by the output of a module) or left unconnected, in which case they get a special value `x` that represents 'unknown'. Continuous assignments use the keyword `assign`, whereas procedural assignments just have the form v = e, where v is a register and e an *expression*.

The body of `AND_TEST_DATA` has the form `initial` s, where s is a *statement*. This means that statement s is to be executed once at the start of the simulation. In the example here, the statement to be executed is a *sequential block* consisting of a sequence of procedural assignments, some of which are *delayed*. When control reaches a statement of the form `#`n s, there is a delay of n units of *simulation time* before execution is continued at s. The effect of executing `AND_TEST_DATA` is thus to immediately assign `0` to both `i1` and `i2`, then to delay one unit of time, then to assign `1` to `i2`, then to delay another unit of time, then to assign `1` to `i1` and `0` to `i2`, then to delay another unit of time and finally to assign `1` to `i1`.

To apply the test data specified in `AND_TEST_DATA` to the modules `AND` and `AND_IMP` the following module is defined.

```
module AND_TEST ();
 wire i1,i2,o1,o2;

 AND_TEST_DATA M1(i1,i2);
 AND           M2(i1,i2,o1);
 AND_IMP       M3(i1,i2,o2);

 initial
  $monitor
   ("Time = %0d, i1 = %b, i2 = %b, o1 = %b, o2 = %b",
    $time, i1, i2, o1, o2);

endmodule
```

`AND_TEST` connects the outputs of `AND_TEST_DATA` to the inputs of `AND` and `AND_IMP` using wires. Separate output wires `o1` and `o2` are used for `AND` and `AND_IMP`, so that the outputs can be compared. It is a rule of Verilog that wires must be used to connect modules. Thus although inside the definition of `AND_TEST_DATA` the two outputs are registers, when the module is instantiated in `AND_TEST` the outputs are wires.

The statement `$monitor(`$\cdots$`)` is a directive to the simulator to print out the values on the wires whenever they change. Such extra-language constructs (which include statements prefixed by `$`, macros and embedded comments) are historically not part of the Verilog language, though the IEEE 1364 Draft document includes some of them. With real hardware prototyping, a device is built and then connected to oscilloscopes, logic analysers etc to observe its operation. Using Verilog, a model can be programmed and then controlled and observed using 'software probes'. The simulator and compiler directives provide a kind of metalanguage for manipulating the execution of the Verilog object language. This is different from VHDL, which contains both modelling and monitoring constructs within a single language.

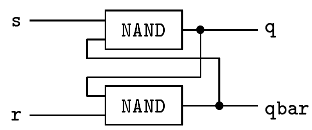
Simulating the module `AND_TEST` results in the following output.

```
Time = 0, i1 = 0, i2 = 0, o1 = 0, o2 = 0
Time = 1, i1 = 0, i2 = 1, o1 = 0, o2 = 0
Time = 2, i1 = 1, i2 = 0, o1 = 0, o2 = 0
Time = 3, i1 = 1, i2 = 1, o1 = 1, o2 = 1
```

This verifies that for all possible inputs, `AND` and `AND_IMP` produce the same output.

2.2 Feedback and memory

An SR flipflop is device with memory. It is built using two `NAND` gates:



This has two stable states. The value `1` can be stored by simultaneously driving `s=0` and `r=1`, which will cause `q=1` and `qbar=0`. The value `0` can be stored by driving `s=1` and `r=0`, which will cause `q=0` and `qbar=1`. If both `s` and `r` are then driven with `1`, the stored value will be maintained in the feedback loops and is available on output `q`. Driving both `s` and `r` with `0` is illegal (the subsequent behaviour will be unpredictable).

An SR flipflop is represented in Verilog by:

```
module SRFF (s,r,q,qbar);
 input s,r; output q,qbar;

 NAND NAND1(s,qbar,q);
 NAND NAND2(q,r,qbar);

endmodule
```

The operation of this can be tested with the module:

```
module TEST ();
 reg s,r; wire q,qbar;

 initial begin    s = 0; r = 1;
              #5 s = 1;
              #5 r = 0;
              #5 r = 1;
              #5 s = 0;
              #5 r = 0;
              #5 s = 1; r = 1;
         end

 SRFF M(s,r,q,qbar);

 initial
  $monitor
   ("Time = %0d, s = %b, r = %b, q = %b, qbar = %b",
    $time, s, r, q, qbar);

endmodule
```

which generates the following output:

```
Time = 0,  s = 0, r = 1, q = 1, qbar = 0
Time = 1,  s = 1, r = 1, q = 1, qbar = 0
Time = 5,  s = 1, r = 0, q = 0, qbar = 1
Time = 6,  s = 1, r = 1, q = 0, qbar = 1
Time = 10, s = 0, r = 0, q = 1, qbar = 1
Time = 15, s = 1, r = 1, q = 0, qbar = 1
```

This shows that `1` is loaded at time `0` and then stored in the flipflop until time `5` when `0` is loaded and then stored. At time `10` the flipflop is driven with the illegal input `s=0` and `r=0` causing both `q` and `qbar` to hold the value `1`. At time `15`, `s` is driven with `1` causing `q` to be `0` and then `r` is driven with `1` causing in `qbar` to be `1`. If `s` and `r` had been driven in the reverse order (i.e. `r = 1; s = 1`) at time `15` then first `qbar` would become `0` and then `q` would become `1`. Another possible behaviour at time `15` would be oscillation with both `q` and `qbar` switching between `1` and `0`.. This effect can be obtained by subtly changing the behaviour of the `NAND` by using a procedural assignment rather than a continuous assignment.

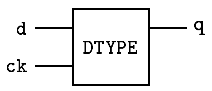
Consider the module `NANDP` ("`P`" for procedural):

```
module NANDP (i1,i2,o);
 input i1, i2; output o; reg o;
 always @(i1 or i2) o = ~(i1 & i2);
endmodule
```

The body of `NANDP` is of the form `always` s, which means that s should be repeated forever. In this example s is the procedural assignment `o = ~(i1 & i2)`, with the *timing control* `@(i1 or i2)` that waits for a change of value to either `i1` or `i2`. Whenever `i1` or `i2` changes, the `NAND` of their values is scheduled to be assigned to `o` 'at the end of the current time slot' (technically after a 'zero delay' – see Section 2.7). This means that if `NANDP` is used instead of `NAND` in the SR flipflop, the simulation will go into an infinite loop at time `15` (which represents an oscillation).

The explanation is that when, at time `15`, `s` is driven with `1` then `q` is scheduled to get value `0`, but it does not get this value immediately. First `r` is driven with `1` which, in turn, schedules `qbar` to get `0`. Only after the values of `s` and `r` have been changed to `1` are the values of `q` and `qbar` changed to `0`. Since `q` and `qbar` are fed back into the `NAND`s, the `@(s or qbar)` and `@(q or r)` fire and `q` and `qbar` are then both scheduled to change back to `1` again. An infinite loop ensues.

Another standard memory element is an edge-triggered Dtype register.

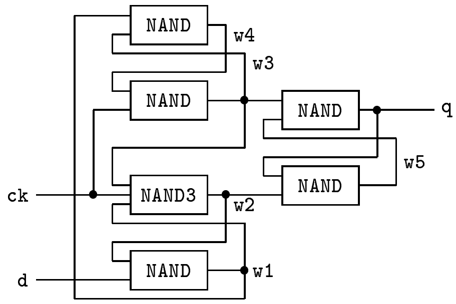


Whenever there is a positive edge on the input `ck` (i.e. a change from `0` to `1`) the value being input on `d` is stored and then output on `q`. A behavioural specification is:

```
module DTYPE (ck,d,q);
 input ck,d; output q; reg q;
 always @(posedge ck) q = d;
endmodule
```

The body of `DTYPE` is of the form `always @(posedge ck)` s, where s is the procedural assignment `q = d`. The timing control `@(posedge ck)` waits for a positive edge on `ck`. The behaviour this module is that whenever `ck` changes to `1` (i.e. a positive edge) the value being input on `d` is assigned to the register `q`.

A standard implementation of a Dtype is the circuit below. This will not be explained here, but can be understood from either a physical [11, Chapter 7] or logical [7] perspective.



The three-input NAND-gate `NAND3` is specified by:

```
module NAND3 (i1,i2,i3,o);
 input i1, i2, i3; output o;
 assign o = ~(i1 & i2 & i3);
endmodule
```

The Dtype implementation above is represented by the Verilog module:

```
module DTYPE_IMP (ck,d,q);
 input ck,d; output q; wire w1,w2,w3,w4,w5;
 NAND  M1(w2,d,w1);
 NAND3 M2(w3,ck,w1,w2);
 NAND  M3(w4,ck,w3);
 NAND  M4(w1,w3,w4);
 NAND  M5(w3,w5,q);
 NAND  M6(q,w2,w5);
endmodule
```

The module `DTYPE_TEST_DATA` specifies some test signals (everything following `//` on a line is a comment):

```
module DTYPE_TEST_DATA (ck,d);
 output ck,d; reg ck,d;
 initial begin     ck = 0; // time 0
               #5  d  = 1; // time 5
               #5  ck = 1; // posedge ck at time 10
               #10 ck = 0; // negedge ck at time 20
               #5  d  = 0; // time 25
               #5  ck = 1; // posedge ck at time 30
               #5  d  = 1; // time 35
         end
endmodule
```

A test harness to compare the behavioural specification and implementation with this test data is the module `DTYPE_TEST`

```
module DTYPE_TEST ();
 wire ck,d,q1,q2;
 DTYPE_TEST_DATA M1(ck,d);
 DTYPE           M2(ck,d,q1);
 DTYPE_IMP       M3(ck,d,q2);
 initial
  $monitor
   ("Time = %0d, ck = %b, d = %b, q1 = %b, q2 = %b",
    $time, ck, d, q1, q2);
endmodule
```

Running this results in:

```
Time = 0,  ck = 0, d = x, q1 = x, q2 = x
Time = 5,  ck = 0, d = 1, q1 = x, q2 = x
Time = 10, ck = 1, d = 1, q1 = 1, q2 = 1
Time = 20, ck = 0, d = 1, q1 = 1, q2 = 1
Time = 25, ck = 0, d = 0, q1 = 1, q2 = 1
Time = 30, ck = 1, d = 0, q1 = 0, q2 = 0
Time = 35, ck = 1, d = 1, q1 = 0, q2 = 0
```

Only times at which `ck`, `d`, `q1` or `q2` change are shown. The value `x`, representing 'unknown', is assigned to all variables (wires and registers) at the beginning of the simulation. This output shows that at time `10` there is the first positive edge of `ck` and at that time the value `1` being input on `d` is 'latched' by both `DTYPE` and `DTYPE_IMP`. The values on the outputs `q1` and `q2` remain stable at `1` until the next positive edge, which is at time `30`, when the input `0` on `d` is latched.

This test shows that `DTYPE` and `DTYPE_IMP` are equivalent for the test data in `DTYPE_TEST_DATA`, however one would like to be able to formally prove from the semantics of Verilog that they are equivalent for all possible inputs (if indeed they are). This is a semantic challenge (see 5.6). Some models used for verification by formal proof (e.g. the relational model – see 5.2) cannot predict that feedback loops in zero-delay combinational circuits will exhibit memory; however, the way Verilog's simulation semantics propagates signal changes enables this to be predicted.

2.3 Inertial and transport delay

The examples in the previous section are unrealistic because the components have no delay.

Verilog supports continuous assignments with delay. These have the form `assign #`n w `=` e and specify that whenever the value of e changes, w is scheduled to be driven with its new value after a delay of n time units. Verilog's semantics specifies that at most one change to a given wire can be scheduled at any one time, so if a change is scheduled before a previously scheduled one has been carried out, then the earlier one is *cancelled*. This rather subtle behaviour is called *inertial delay*. It has the effect that if two changes to e's value happen within n time units, then the effect of the first change is cancelled. To illustrate this, consider a simple unit-delay element.

```
module DEL (i,o);
 input i; output o;
 assign #1 o = i;
endmodule
```

Two unit-delays in series is:

```
module DEL_DEL (i,o);
 input i; output o; wire w;
 DEL M1(i,w);
 DEL M2(w,o);
endmodule
```

This can be compared with an inertial delay of 2.

```
module DEL2 (i,o);
 input i; output o;
 assign #2 o = i;
endmodule
```

The following trace shows an example of `DEL_DEL`'s and `DEL2`'s outputs for a particular sequence of inputs.

Time	Input	Output from DEL_DEL	Output from DEL2
0	x	x	x
1	x	x	x
2	x	x	x
3	x	x	x
4	x	x	x
5	0	x	x
6	0	x	x
7	0	0	0
8	0	0	0
9	0	0	0
10	1	0	0
11	0	0	0
12	0	1	0
13	0	0	0
14	0	0	0
15	1	0	0
16	1	0	0
17	0	1	1
18	0	1	1
19	0	0	0

When input `i` changed to `1` at time `10` the change `w=1` is scheduled inside `DEL_DEL` for time `11`, and the change `o=1` is scheduled by `DEL2` at time `12`. At time `11` the change `w=1` happens inside `DEL_DEL`, which causes `o=1` to be scheduled at time `12`. When `i` changes to `0` at time `11`, the continuous assignment in `DEL2` schedules `o=0` at time `13`, which cancels the `o=1` scheduled for time `12`. Thus at time `12`, the change to `o` in `DEL_DEL` happens, but the change to `o` in `DEL2` has been cancelled. At time `13`, `o` in `DEL2` is `0` so the previously scheduled `o=0` has no effect. When `i` changes from `0` to `1` at time `15`, `DEL2` schedules `o=1` for time `17`. Since `i` doesn't change at time `16`, `o=1` is not cancelled and `DEL2`'s output changes at time `17`.

`DEL_DEL` exhibits *transport delay*: all changes to its input are propagated to its output with a delay of `2`. `DEL2` exhibits inertial delay: only changes that persist for at least two time units are propagated. Note that `DEL_DEL` implements transport delay only because its internal delays are unit delays. Two `DEL2`s in series would exhibit a mixture of transport and inertial delay.

2.4 Blocking & non-blocking assignments

Verilog's *non-blocking assignment* enables transport delays to be expressed behaviourally. A non-blocking assignment has the form v `<= #`n e, where v is a register. Such an assignment causes no delay in the execution of the current module, but schedules the current value of e to be assigned to v after a delay of `n`. In contrast to continuous assignments, non-blocking assignment allows multiple changes to be scheduled to the same variable: no cancelling happens. Thus the following module has the same transport delay behaviour as DEL_DEL.

```
module TRANS_DEL2 (i,o);
 input i; output o; reg o;
 always @(i) o <= #2 i;
endmodule
```

This generates an infinite loop (`always`) with the behaviour that whenever `i` changes (`@(i)`) the new value of `i` is scheduled to be assigned to `o` after a delay of `2`.

Blocking assignments have the form v `= #`n e. When such an assignment is reached, the value of e is computed, execution is delayed ('blocked') for n time units and then the previously computed value of e is assigned to v. For example, consider:

```
begin x = 1; y = 2; x = #5 y; y = #5 x; end
```

When control reaches `x = #5 y`, the variable `x` has value `1` and `y` value `2`. The computation is delayed for `5` time units and then `x` is assigned the value `2`. The computation is then delayed another `5` time units and then `y` gets the value `2` that `x` had just been assigned. Thus executing this sequential block takes `10` time units and results in `x` and `y` both having the value `2`.

Consider now:

```
begin x = 1; y = 2; x <= #5 y; y <= #5 x; end
```

When control reaches `x <= #5 y`, the variable `x` has value `1` and `y` the value `2`, as before. The effect of `x <= #5 y` is to schedule `x=2` for `5` time units in the future; `x` is not changed until then. The non-blocking assignment itself takes no time and control immediately proceeds to `y <= #5 x` which itself takes no time, but schedules `y=1` also for `5` units of time in the future. The execution of the sequential block is now finished. `5` time units later `x` gets value `2` and `y` gets value `1`. Thus executing this sequential block takes zero time and results in `x` being scheduled to have the value `2` and `y` the value `1` after `5` time units have passed.

A blocking assignment v `= #`n e differs from a delayed assignment `#`n v `=` e because the former evaluates e before the delay occurs, but the latter evaluates it after the delay has taken place.

2.5 Datatypes

Verilog allows variables to be declared to carry arbitrary bitstrings (called *vectors*), signed integers, times (which are unsigned) and reals. For example, the declaration `reg[3:0] v` declares `v` to be a `4`-bit vector register (`reg` is a reserved word). Its components are accessed by expressions (from most to least significant) `v[3]`, `v[2]`, `v[1]` and `v[0]`.

Vectors of vectors, called *memories*, can also be declared. For example `reg[7:0] mem[0:255]` declares a memory `mem` consisting of `256` eight-bit registers. The details of Verilog's datatypes are not considered here.

2.6 Imperative programming constructs

Verilog provides a selection of familiar programming constructs including conditionals, case switches, `while`-statements, `for`-statements, sequential and parallel blocks. For simplicity, only a subset of these will be considered.

In the example that follows, the programming constructs are used to specify a behavioural model of a divider (DIVIDE) and also provide some test data for it (DIVIDE_TEST_DATA).

Whenever either `x` or `y` changes (`always @(x or y)`), the module DIVIDE computes, by repeated subtraction, the quotient `q` and remainder `r` of dividing `x` by `y` (the arithmetic operators `+` and `-` apply to vectors interpreted as natural numbers using modular arithmetic).

```
module DIVIDE(x,y,q,r);
 input [1:0] x,y; output q,r; reg [1:0] q,r;
 always @(x or y)
  begin
   q = 0;
   r = x;
   while (y<=r) begin
                 r = #1 r-y;
                 q = #1 q+1;
                end
   $display
    ("Time = %0d, x = %0d, y = %0d, q = %0d, r = %0d",
     $time, x, y, q, r);
  end
endmodule
```

The initialisation assignments of `q` and `r` are modelled as taking zero delay, but each assignment in the body of the `while`-loop is given unit delay.

The statement `$display(···)` prints out the time and the values of `x`, `y`, `q` and `r` (in decimal notation) when control reaches it (which is just after the `while`-loop has terminated).

The module DIVIDE_TEST_DATA generates all non-zero combinations of `x` and `y` in sequence, changing the values each `10` time units.

```
module DIVIDE_TEST_DATA (x,y);
 output x,y; reg [1:0] x,y;
 initial
  begin x=1; y=1;
        while (x<=3) begin
                      while (y<3) #10 y=y+1;
                      #10 x = x+1; y=1;
                     end
  end
endmodule
```

Notice that the outer `while`-statement is an infinite loop because addition on values of size `[1:0]` is modulo four.

The test harness DIVIDE_TEST feeds the data generated by DIVIDE_TEST_DATA to DIVIDE. It also sets up a separate thread that waits for `100` time units and then halts the simulation (`$finish`). The infinite loop (and the need for the separate thread) could be avoided by replacing `x<=3` by `x>0` in DIVIDE_TEST_DATA.

```
module DIVIDE_TEST ();
 wire [1:0] x,y,q,r;
 DIVIDE_TEST_DATA M1(x,y);
 DIVIDE           M2(x,y,q,r);
 initial #100 $finish;
endmodule
```

Simulating this example results in the following (printed by the `$display` statement in `DIVIDE`).

```
Time = 2,  x = 1, y = 1, q = 1, r = 0
Time = 10, x = 1, y = 2, q = 0, r = 1
Time = 20, x = 1, y = 3, q = 0, r = 1
Time = 34, x = 2, y = 1, q = 2, r = 0
Time = 42, x = 2, y = 2, q = 1, r = 0
Time = 50, x = 2, y = 3, q = 0, r = 2
Time = 66, x = 3, y = 1, q = 3, r = 0
Time = 72, x = 3, y = 2, q = 1, r = 1
Time = 82, x = 3, y = 3, q = 1, r = 0
Time = 90, x = 0, y = 1, q = 0, r = 0
```

At the start of the simulation `x` and `y` are initialised to the 'unknown' value. At time `0` they are both assigned value `1` by `DIVIDE_TEST_DATA`, which triggers `@(x or y)` in `DIVIDE` causing the `while`-loop to be executed to compute `q` and `r`. This only takes one iteration, which takes two time units (one for each assignment in the body of the `while`), thus the `$display(···)` is first reached at time `2` generating the first line of output. The next change to `x` and `y` happens at time `10`, when `DIVIDE_TEST_DATA` increments `y`. This triggers `@(x or y)` in `DIVIDE` again, but this time the test of the `while`-loop is false, so no iterations are done and `$display(···)` is reached for the second time at time `10`.

2.7 Concurrent threads

The general form of a module specification is:

module *name* ($port_1, \ldots, port_m$);
  declarations;
  item₁
  ⋮
  itemₙ
endmodule

Each item is executed in parallel in a separate thread of computation. The main module items are continuous assignments, instances of other modules, `initial`-statements and `always`-statements. Shared variables can lead to non-determinism. For example, consider:

```
module INTERLEAVE ();
integer x;
initial begin x=0; x=x+2; end
initial x=1;
endmodule
```

This is an example of a *race condition*: the semantics does non uniquely determine the result. If the first `initial`-statement is completed before the second one is started, then `x` is set to `1`. This behaviour can be forced by putting a zero delay before the second `initial`-statement:

```
initial #0 x=1
```

If the first `initial`-statement is started after the second one is completed, then `x` is set to `2`. This can be forced by putting a zero delay before the first `initial`-statement:

```
initial #0 begin x=0; x=x+2; end
```

If the second `initial`-statement is executed after the `x=0` in the first `initial`-statement, but before the `x=x+2`, then `x` is set to `3`. This can be forced by:

```
initial begin x=0; #0 #0 x=x+2; end
initial #0 x=1;
```

The use of explicit zero delays to force determinacy is considered a bad programming style by some. The exact semantics of delays is explained in section 4.

The use of non-blocking assignment can lead to further subtlety. For example, consider:

```
module NONBLOCK_INTERLEAVE ();
integer x;
initial begin x=0; x<=x+2; end
initial x=1;
endmodule
```

With the Viper/free simulator from interHDL [16] the result is that `x` is set to `1`, but with the Veriwell simulator from Wellspring Solutions Inc. [15], `x` is set to `2`. According to my reading of the official IEEE scheduling semantics [8], `x` should never end up set to `1` since *non-blocking assign update* events are scheduled for the very end of the simulation cycle, and the only way `x` could end up with `1` is if the assignment `x=1` is scheduled after the update created by `x<=x+2`. However, I may have misread the IEEE document!

Concurrent threads can also be generated using parallel blocks (`fork`–`join`), but these will not be considered here.

3 V: a simple version of Verilog

This section specifies a language called V that is proposed as a vehicle for experiments in contructing and using a formal semantics of Verilog.

V is close to being a subset of Verilog, but contains two constructs not in it. The first of these are assignments of the form $v \lhd\!\!- \texttt{\#}n\ e$ that are like delayed non-blocking assignments, but with inertial delay. Having these simplifies the description of the simulation cycle, by enabling continuous assignments to be translated into `always`-statements. The second construct in V, but not Verilog, is a timing control Δ that is very similar to `@`, but without the zero-delay discussed in connection with `NANDP` in Section 2.2. This timing control is also used for modelling continuous assignments.

The syntax of V will be specified in a BNF style, using metavariables to range over the various constructs.

Occurrences of metavariables may be distinguished by decorating them with subscripts, superscripts or primes.

3.1 Modules

A specification in V consists of a set of *modules*, one of which is singled out as the *top level module*.

Modules in V have a name, port list (which may be empty), set of declarations and a set of module items. Each item is either a continuous assignment, an `initial`-statement, an `always`-statement or an instance of another module. Details of datatypes and declarations are avoided here, as the main goal is to describe the simulation cycle.

3.2 Expressions

Expressions are composed out of variables, constants (ranged over by n), unary operators (ranged over by u) and binary operators (ranged over by b). For simplicity, V assumes expressions are evaluated to yield either a non-negative number (which can be thought of as a bitstring), or the special value x.

The syntax of expressions is specified by:

e ::= v | n | $u\ e$ | $e_1\ b\ e_2$ | e ? e_1 : e_2 | (e)

Thus an expression e is either a variable v, or a constant n, or a unary operator u applied to an expression e, or an infixed binary operator b applied to two expressions e_1 and e_2, or a conditional e ? e_1 : e_2 meaning "if e then e_1 else e_2", or parenthesised.

The value of an expression in V is a natural number or x (the unknown value). Unary and binary operators are assumed 'strict' (i.e. if an argument is x then the result is x). Conditionals are strict in their first argument. In the test expressions occurring in conditionals and `while`-statements, a non-zero result represents *true* and zero represents *false*.

Verilog supports various automatic coercions on bitwidths, which can make it tricky to handle arithmetic overflows. However, the details of these are orthogonal to the simulation semantics and are not treated here.

3.3 Timing controls

Timing controls (ranged over by c) are used for scheduling. They are sequences of *atomic timing controls* (ranged over by ϕ), which are either delays (#e) or *guards* (ranged over by g). Guards are either *edge sensitive* ($\Delta(\eta)$ or @(η)) or *level sensitive* (wait e).

η ::= v | posedge v | negedge v | η_1 or $\cdots$ or η_n

g ::= $\Delta(\eta)$ | @(η) | wait e

ϕ ::= #e | g

c ::= ϕ | $\phi\ c$

3.4 Statements

The syntax of statements is given by:

s ::= v = e	(assignment)
| v = c e	(delayed assignment)
| v <= c e	(non-blocking assignment)
| v $\lhd\!-$ #n e	(inertial assignment)
| c s	(timing controlled statement)
| if (e) s	(one-armed conditional)
| if (e) s_1 else s_2	(two-armed conditional)
| begin s_1; $\cdots$;s_n end	(sequential block)
| while (e) s	(`while`-statement)
| forever s	(`forever`-statement)

4 Semantics of V

The semantics of V is described by explaining how the top-level module is simulated. The first stage is to extract a collection of statements to be executed concurrently.

The top level module is 'flattened' by (i) renaming all local variables in instances of modules to avoid clashes, (ii) replacing module instances by the appropriately instantiated sequence of items they contain and (iii) declaring all local variables at top level. The result of this is a module that only contains continuous assignments and statements (i.e. no module instances).

After this flattening, the only way that a wire can be driven is by a continuous assignment (in unflattened modules they can be driven by module instances). For simplicity, it is assumed that each wire is driven by at most one continuous assignment (in Verilog, wires can be multiply driven and rules are given for computing the resultant value).

After flattening, the top-level module is further transformed so that the only module items it contains are `initial`-statements.

Let v_1, ... , v_n be the variables occurring in e, then all continuous assignments assign v = e are replaced by always $\Delta(v_1$ or $\cdots$ or $v_n)$ v = e and all delayed continuous assignments assign #n v = e are replaced by always $\Delta(v_1$ or $\cdots$ or $v_n)$ v $\lhd\!-$ #n e. Note that the wires driven by continuous assignments become registers.

All `always`-statements always s are replaced by initial forever s.

The resulting flattened and transformed module has the form:

```
module name ( port_1,..., port_m);
  declarations;
  initial s_1
     ⋮
  initial s_n
endmodule
```

The statements s_1, …, s_n are executed concurrrently. Each s_i gives rise to a separate thread of execution.

This flattening and transforming process is called *normalisation*. A more formal account will not be given here, but the following example of the result of normalising `AND_TEST` (see above) should illustrate the process (the `$monitor` statement is omitted).

```
module FLAT_AND_TEST ();
 reg i1,i2,o1,o2,w;
 initial begin    i1 = 0; i2 = 0;
              #1          i2 = 1;
              #1 i1 = 1; i2 = 0;
              #1          i2 = 1;
         end
 initial forever Δ(i1 or i2) o1 = i1 & i2;
 initial forever Δ(i1 or i2) w = ~(i1 & i2);
 initial forever Δ(w) o2 = ~(w & w);
endmodule
```

Observe how normalisation has converted all wires to registers.

4.1 The global state

The *global state* of a simulation consists of the simulation time, the values of registers and the set of threads.

Each thread consists of a statement (the code being executed) and a local state specifying:

1. an *execution point*, which indicates where to continue from the next time the thread is executed;
2. a *status*, which can be
   - (a) *enabled*: the thread can be executed immediately;
   - (b) *delayed until* t: execution is scheduled for a later simulation time t;
   - (c) *guarded by* g: the thread is waiting to be triggered by a change to a variable in g;
   - (d) *finished*: a thread is finished when there are no more statements to execute;
3. a possible *pending assignment* (only present if the thread was delayed within n past time units by a blocking assignment v = c e).

Threads are classified into (i) *statement threads*, which are the executions of statements extracted from the normalised module and (ii) *updates*, which are generated by non-blocking assignments.

4.2 The simulation cycle

The execution of a program is initialized by setting the simulation time to `0`, setting the values of all variables to `x`, creating an enabled statement thread (with no pending assignments) for each statement extracted from the normalised top-level module with the execution points at the beginning of each statement.

Thereafter, the following *simulation cycle* is repeated. Let t denote the current simulation time.

1 If there are any enabled statement threads then choose one and go to **2**, else if there are any threads delayed to t (the current simulation time), then enable all such threads and go to **1**, else if there are any enabled updates, then choose one, perform it, delete it and go to **3**, else if there are any threads delayed to t', where $t' > t$, then go to **4**.

2 If there is a pending assignment then perform it, delete it from the state. Go to **3**.

If the thread has no pending assignment, then make one step along it (see 4.2.1). Go to **3**.

3 If the value of a register has changed, then enable all guarded threads whose guards fire (see 4.2.2). Go to **1**.

4 Advance simulation time the minimum amount (which must be non-zero) needed to reach a time at which at least one thread is scheduled to restart. Enable all threads scheduled to restart at this time. Go to **1**.

The simulation terminates when all threads are finished.

4.2.1 Stepping along a thread

If the execution point is at the end of a thread, then stepping along the thread causes it to finish. If there is a statement following the execution point, then for each kind of statement the effect of taking a step is described below.

$\boxed{v \texttt{ = } e}$ The expression e is evaluated and the resulting value assigned to the register v in the global state. If there is a next statement in the thread it is enabled, otherwise the thread is finished.

$\boxed{v \texttt{ = } c\ e}$ The expression e is evaluated to get a number, n say, and then the status of the thread is set according to c (see 4.2.2), the assignment $v = n$ is made pending and the execution point is moved to the end of the assignment.

$\boxed{v \texttt{ <= } c\ e}$ The expression e is evaluated to get a number, n say, and then a new update thread is created consisting of just the assignment $v = n$ with the execution point at the beginning, the status of the thread is set according to c (see 4.2.2) and no pending assignment. If there is a next statement in the original thread it is enabled, otherwise the thread is finished.

$\boxed{v \lhd\!\!- \texttt{\#}n\ e}$ The expression e is evaluated to get a number, m say, and then a new statement thread is created consisting of just the assignment $v = m$ with the execution point at the beginning, the status delayed according to #n (see 4.2.2) and no pending assignment. All other delayed threads of the form $v = m'$ which are scheduled earlier than the one just created are deleted. If there is a next statement in the original thread it is enabled, otherwise the thread is finished.

$\boxed{c\ s}$ The execution point is moved to just before s and the status of the thread is set according to c (see 4.2.2).

$\boxed{\texttt{if (}e\texttt{) }s}$ The expression e is evaluated. If the result is true then the execution point moves to s and the thread remains enabled. If e is false and there is a next statement in the thread, then it is enabled, otherwise the thread is finished.

$\boxed{\texttt{if (}e\texttt{) }s_1\texttt{ else }s_2}$ The expression e is evaluated. If the result is true then the execution point moves to s_1, otherwise it moves to s_2. In both cases the thread remains enabled.

$\boxed{\texttt{begin }s_1\texttt{;}\cdots\texttt{;}s_n\texttt{ end}}$ Control moves to the first statement s_1 and the thread remains enabled.

$\boxed{\texttt{while (}e\texttt{) }s}$ The thread is replaced by the statement `if (`e`) begin` s`; while (`e`)` s `end` with the execution point at the beginning and the thread enabled.

$\boxed{\texttt{forever }s}$ This is equivalent to `while (1)` s.

4.2.2 Setting up a delay or guard

In general, a timing control is a non-empty sequence $\phi_1\,\phi_2\,\cdots\,\phi_n$ of atomic timing controls. Such a sequence is evaluated by considering ϕ_1 as below and prefixing $\phi_2\,\cdots\,\phi_n$ (which might be empty) to the statement following the execution point of the thread.

$\boxed{\texttt{\#}e}$ The value of e is added to the current simulation time to get a future time, t' say, and the status of the thread becomes delayed until t'. Note that a delayed thread is not enabled, so the effect of a zero delay `#0` is to schedule the rest of the thread for the current time, but after all currently enabled statement threads and before all currently enabled updates (see **1** above).

$\boxed{\texttt{@(}\eta\texttt{)}}$ This is equivalent to the sequence Δ`(`η`) #0`.

$\boxed{\Delta\texttt{(}v\texttt{)}}$ The thread becomes guarded with a guard that will fire whenever v is changed.

$\boxed{\Delta\texttt{(posedge }v\texttt{)}}$ The thread becomes guarded with a guard that will fire whenever v changes to `1`.

$\boxed{\Delta\texttt{(negedge }v\texttt{)}}$ The thread becomes guarded with a guard that will fire whenever v changes to `0`.

$\boxed{\Delta\texttt{(}\eta_1\texttt{ or }\cdots\texttt{ or }\eta_n\texttt{)}}$ A guard is created that fires when any of $\eta_1, \ldots, \eta_n$ fire.

$\boxed{\texttt{wait }e}$ If e is true then the thread remains enabled; if it is false then a guard is created that will fire whenever e becomes true.

4.3 Warning!

The semantics of V is intended to be a prototype for a semantics of Verilog. It is based on a careful reading of various sources [8, 12, 13, 14] and experiments with the Veriwell [15] and Viper/free [16] simulators. I hope to validate the semantics with a combination of review by Verilog experts (I am not one) and formalisation experiments, but until this is done the reader is warned not to place too much trust in the details. Already several errors in an earlier version of the simulation semantics have been corrected.

5 Semantic challenges

The semantic challenges in this section are intended to combine theoretical interest with practical utility. Many of them are instantiations to the world of Verilog of general topics in logic and semantics for which considerable abstract theory already exists.

5.1 Formal semantics of Verilog

The first challenge is just to get a formal semantics of Verilog, starting with the subset V, that is both accurate to the spirit of the language and mathematically tractable. Many attempts to give a formal semantics of VHDL [9, 5, 10] are in progress. These use a variety of techniques including stream processing (Fuchs & Mendler), functional programming (Breuer et al), labelled transition systems (Van Tassel), evolving algebras (Börger et al.), Petri nets (Olcoz), finite state automata (Döhmen & Herrmann), flow graphs (Reetz & Kropf), denotational semantics (Davis) and the state-delta temporal logic formalism (Filippenko). The semantics of other kinds of event simulation languages are also being studied [1, 2].

5.2 Validity of simplified semantics

Any semantics that reflects the spirit of the language (i.e. formalises the simulation cycle) is likely to be hard to work with and may well not be syntax directed (compositional). A second challenge is to develop simpler and more tractable semantics for subsets of the language and to prove that these agree with the general semantics on the subset.

One standard approach is to model hardware devices as a relation between sequences of values (the sequences representing successive values on a wire) [6]. There is considerable experience in using this model and it would be particularly useful if it could be related to Verilog's semantics.

5.3 A minimal simulation calculus

V is a first attempt to distill the essence of the Verilog simulation semantics into a simple setting. However, it is still relatively large, ad hoc and redundant. A challenge is to devise a minimal discrete event simulation calculus that would form a canonical basis for theoretical analysis. This calculus would be to Verilog/VHDL roughly as the λ-calculus is to functional programming.

5.4 Correctness of synthesisers

Current synthesisers can generate hardware implementations from substantial subsets of Verilog. For example, the `CV` Verilog Compiler implemented by David Greaves and used at Cambridge University can synthesise hardware implementations of modules that contain continuous assignments (without delays) and behavioural statements with `@(posedge` v`)` timing controls, non-blocking assignments, conditionals and sequential blocks.

A challenge is to formalise real-world synthesis algorithms and show that the hardware structures generated are equivalent to the behavioural source. There has been quite a lot of work on verifying synthesisers in the past, but none (that I know of) for synthesis from modern event-based HDLs.

5.5 Definition of equivalences

For many purposes it is important to ensure that pairs of specifications are 'equivalent'. However, exactly what equivalence means is subtle. The strongest equivalence would be that two specifications were indistinguishable by the simulator. However, in practice this is likely to be too strong: one may only need equivalence with respect to certain classes of test data. For example, an implementation using inertial delay might be equivalent to a behavioural specification with transport delay, under the condition that signals change slowly.

A challenge is to develop a general theory of behavioural equivalence for Verilog, together with 'laws' for using the theory.

5.6 Conditions for equivalence

Ensuring that behavioural specifications are equivalent to structural implementations is an important practical problem. Hardware components are often given several different models at different levels of abstraction and much time and expense can be wasted if there are undocumented differences. A challenge (that we hope to address at Cambridge) is to develop 'verification conditions' that are sufficient to ensure that two specifications are equivalent.

5.7 Relation to timed process calculi

Timed process calculi (e.g. timed CCS, and timed CSP) provide a standard compositional paradigm for representing timed behaviour. Is it possible to translate Verilog into such a calculus and prove the translation sound? The various theories of equivalence and refinement for process algebra might suggest useful notions for Verilog (e.g. kinds of equivalence).

5.8 Programming logic

Verilog's imperative programming constructs (assignment, sequencing, conditionals, `while`-loops etc.) should satisfy proof rules like those for Hoare logic, with suitable restrictions. Under what conditions can existing methods for reasoning about sequential and parallel programs be applied to subsets of Verilog? How can these methods be proved sound with respect to simulation semantics?

5.9 Checkable properties

The most successful applications of formal methods to hardware design have been the use of decision procedures and model checkers (usually based on binary decision diagrams – BDDs) to automatically verify properties. A challenge is to discover classes of properties of Verilog programs that can be automatically checked using such existing methods. To do this properly requires that metatheorems be proved establishing that the properties are equivalent to standard decision or model checking problems.

6 Summary and conclusions

Verilog is a relatively simple real-world language in need of theoretical support. It poses a variety of interesting semantic and logical challenges ranging from routine applications of standard techniques (e.g. formalizing the simulation cycle) to hard theoretical problems (e.g. developing a theory of behavioural congruence).

Acknowledgements

I became interested in Verilog through conversations with David Greaves. He helped me learn the language, supplied some of the examples and text used here and answered numerous questions concerning Verilog's semantics. At Cambridge, Richard Boulton, David Greaves and John Herbert read a first draft of this

paper and made many suggestions for its improvement. Following a post to `comp.lang.verilog` I received further helpful suggestions from Henry G. Cox, Peet James and John Sanguinetti. Also, Yatin Trivedi emailed me the section on "Scheduling semantics" from the the IEEE 1364 Draft document (*Draft Standard Verilog HDL*), which showed that my original treatment of non-blocking assignment was wrong.

References

[1] G. Birtwistle and C. Tofts. "Operational Semantics for Process-Based Simulation Languages. Part 1: πDemos", *Transactions of The Society for Computer Simulation*, Vol. 10(4), pp. 299–333, 1993.

[2] G. Birtwistle and C. Tofts. "Operational Semantics for Process-Based Simulation Languages. Part 2: μDemos", *Transactions of The Society for Computer Simulation*, Vol. 11(4), pp. 303–336, 1994.

[3] Newsgroup posting by John Cooley, see: `http://www.cl.cam.ac.uk/users/mjcg/Verilog/VHDL-Verilog.html`.

[4] Corporate Background Information on *Design Acceleration, Inc*; available on the WWW at the URL `http://www.designacc.com/`.

[5] K.C. Davis, "A Denotational Definition of the VHDL Simulation Kernel", Electrical and Computer Engineering Department, University of Cincinnati, Cincinnata, OH 45221-0030, email: `karen.davis@uc.edu`.

[6] M.J.C. Gordon, "Why Higher-Order Logic is a Good Formalism for Specifying and Verifying Hardware", *Formal Aspects of VLSI Design*, G. Milne and P.A. Subrahmanyam (Eds.), North Holland, 153–177, 1986.

[7] F.K. Hanna & N. Daeche, "Specification and Verification using Higher-Order Logic: A Case Study", *Formal Aspects of VLSI Design*, G. Milne and P.A. Subrahmanyam (Eds.), North Holland, pp. 179–213, 1986.

[8] "Scheduling semantics", Section 5 of IEEE 1364 Draft document *Draft Standard Verilog HDL*, Draft 3.1, March 1995.

[9] C. Delgado Kloos & P.T. Breuer (editors), *Formal Semantics for VHDL*, Kluwer Academic Publishers, March 1995.

[10] I.V. Filippenko, "VHDL Verification in the State Delta Verification System (SDVS)", *1991 International Workshop on Formal Methods in VLSI Design*, Miami, Florida, 1991.

[11] C. Mead & L. Conway, *Introduction to VLSI Systems*, Addison-Wesley, 1980.

[12] Open Verilog International (OVI), *Verilog Hardware Description Language Reference Manual*, Version 1.0, Open Verilog International, 15466 Los Gatos Blvd., Suite 109-071 Los Gatos, CA 95032, email: `ovi@netcom.com`.

[13] E. Sternheim, R. Singh, Y. Trivedi, R. Madhavan & W. Stapleton, *Digital Design And Synthesis with Verilog HDL*, Automata Publishing Company, 1072 S. Saratoga-Sunnyvale Rd., San Jose, CA 95129, ISBN 0-9627488-2-X, email: `help@apco.com`.

[14] D.E. Thomas & P.R. Moorby, *The Verilog Hardware Description Language (second edition)*, Kluwer Academic Publishers, 1995.

[15] The Veriwell simulator is available for downloading from Wellspring Solutions via ftp: `ftp://iii.net:/pub/pub-site/wellspring/`.

[16] The Viper/free simulator is available free from interHDL, Inc., 4984 El Camino Real, Suite 210, Los Altos, CA. 94022-1433, email: `info@interhdl.com`.