Guidelines for Safe Simulation and Synthesis of Implicit Style Verilog

Mark G. Arnold
Computer Science Department
University of Wyoming
Laramie, Wyoming 82071
marnold@uwyo.edu

Neal J. Sample
Computer Science Department
University of Wyoming
Laramie, Wyoming 82071
nsample@uwyo.edu

James D. Shuler
Computer Science Department
SUNY College at Brockport
Brockport, New York 14420
jshuler@cs.brockport.edu

Abstract

We discuss the classes of machines for which implicit style design is appropriate, and give guidelines for safe simulation and synthesis of implicit style Verilog that ensure the results of cycle based simulation agree with the results of synthesis.

Introduction

Hardware systems of any complexity inevitably contain one or more state machines. Traditionally, such machines have been designed using the explicit style, where the designer focuses on implementing the transition from the present state to the next state. Although the definition of a state machine eventually involves such transitions, the designer's goal in specifying a state machine is seldom that it just make transitions. Rather, the designer wishes the machine to carry out various steps of some algorithm during a sequence of clock cycles. The real work of the designer (choosing an algorithm and deciding how its steps will be scheduled) occurs at a higher level of abstraction than can be expressed clearly in the explicit style. The explicit style requires the designer to describe too many details about the implementation of the state machine, and so the designer is not able to focus complete attention on choosing a good algorithm to solve the problem. Because of the investment of effort in existing explicit style designs, the designer of a complex system can become locked into a particular set of state transitions, instead of exploring the design space more thoroughly.

Recently, there has been growing interest [6, 2, 4, 3, 7] in an alternative approach, known as the implicit style. Like the explicit style, the implicit style allows the designer to choose the cycle by cycle schedule of algorithmic steps carried out by each state of the machine. Unlike the explicit style, the implicit style does not require the designer to describe how the transitions between those states occur. Hardware design with the implicit style is very similar to software design. The main distinction between software and implicit style hardware is the issue of scheduling computation relative to the clock. In contrast, hardware design with the explicit style is far more tedious than typical software design.

The purpose of this paper is to describe the "safe" subset of implicit style design that works for both simulation and synthesis, and how such Verilog should be coded so that it can be easily ported between tools. It is the authors' opinion that although this "safe" subset is somewhat limited, it is adequate for the design of a wide range of hardware systems.

Categories of machines

In hierarchical design, a machine is composed of smaller sub-machines. One can categorize the behavior of the entire machine, or of any of the sub-machines. One useful taxonomy of machines is given by Clair [5], who categorizes machines into five classes. These classes will allow us to define in a more formal way the set of machines for which the implicit style is appropriate.

Class 0

Class 0 consists only of combinational logic, and therefore has no memory. There are two common ways to describe a class 0 machine using behavioral Verilog: a continuous `assign` statement or an `always` block with the proper sensitivity list. (Every variable inside the `always` block that is not assigned a value occurs in the list; every variable inside the `always` block that is assigned a value does not occur in the list). The distinction between the `assign` and `always` approaches is whether the output is defined as a `reg` or a `wire`.

Class 1

Class 1 and above utilize a clock signal. To be successful with most synthesis tools, and to avoid avoid physical problems such as hazards and metastabilty, such a clock signal must be single phase (either `posedge` or `negedge`), and must be distributed to all clocked devices in the system. In this paper, we will assume the name of this global system clock signal is `sysclk` and that all clocked devices are triggered by `@(posedge sysclk)`.

Class 1, which is known as delay, has memory which only lasts for one clock cycle and therefore is of little practical value by itself. On the other hand, class 1 machines are vital to the implementation of the higher classes. The designer uses an `always` block with a single `@(posedge sysclk)` to define a class 1 machine in behavioral Verilog. (Asynchronous resetting is often also use, as in `@(posedge sysclk or posedge reset)`.) A variable assigned by a class 1 machine must be defined as `reg`.

The most widely used technique in behavioral Verilog, known as explicit style design, requires the designer to manually transform a higher class machine into a cooperating collection of class 0 and class 1 sub-machines. In other words, the Verilog source code consists of two separate parts: combinational logic (class 0) and registers (class 1). The combinational logic is defined by a series of continuous assignments and/or `always` blocks with proper sensitivity lists. The registers are defined by a series of `always` blocks with one `@(posedge sysclk)` each. There is a clean separation between the Verilog code for class 0 sub-machines, and class 1 sub-machines, such as:

```
assign class0 = f( ... );
always @(posedge sysclk)
  class1 = class0;
```

The difficulty with this perfectly correct explicit style is that it requires the designer to think at a much lower level than the level at which the customer's problem is stated. Most of the explicit style designer's effort focuses on the state machine implementation problem, and so little of the designer's time is spent actually solving the customer's problem. However comfortable one may feel with the explicit style, given today's ever increasing time to market pressures, reliance on explicit style alone becomes harder to justify.

Class 2

The next higher class in Clair's taxonomy, class 2, illustrates the advantage of abandoning explicit style in favor of implicit style. Clair defines class 2 as having direct state transitions. This means the machine has no input, and only outputs a cyclical sequence derived from a continually repeating set of state transitions. Class 2 machines are essentially free running counters that follow a prescribed sequence. Class 2 is the most elementary of the higher classes of machines which benefit from implicit style design. To define a class 2 machine with implicit style behavioral Verilog, the designer uses an `always` block immediately followed by `begin`, with multiple `@(posedge sysclk)`s inside the block. For example, a four bit gray code counter can be modeled as:

```
reg [1:0] c;
always  //1st implicit example
  begin
    @(posedge sysclk) `ENS;
      c <= `CLK 2'b00;
    @(posedge sysclk) `ENS;
      c <= `CLK 2'b01;
    @(posedge sysclk) `ENS;
      c <= `CLK 2'b11;
    @(posedge sysclk) `ENS;
      c <= `CLK 2'b10;
  end
```

The `@(posedge sysclk)`s indicate the four states of this machine. The `<=` is the non-blocking assignment statement, which is essential to the proper use of the implicit style. The effect of a non-blocking assignment will not be visible until the next rising edge of the clock. The reason for using `<=` instead of `=` will be explained later. We will also ignore the meaning of the macros `` `CLK `` and `` `ENS `` for the moment, as these will be explained together with the `<=`

This machine is a class 2 machine because it continues to produce the sequence ... 00, 01, 11, 11, 00, 01 ... regardless of what other machines may be doing. The experienced explicit style designer may wonder what is all this excitement about implicit style. After all, such a trivial class 2 gray code counter can be described as a combination of class 0 and class 1 sub-machines:

```
reg [1:0] c;
wire [1:0] new_c;
assign new_c = {c[0],~c[1]};
always @(posedge sysclk)
  c = new_c;
```

The above explicit style code is trivial because only constants are on the right of the assignments in the equivalent implicit style code. Most problems instead involve variables on the right of the assignments. The power of the implicit style is that the designer can include many assignment statements together in a single implicit style block, where each assignment involves arbitrarily complex expressions. In the implicit style, the designer puts these where they belong sequentially in the sequence of algorithm steps, making the code readable. An example of such an implicit style machine is:

```
reg [4:0] sum,odd;
always   //2nd implicit example
  begin
    @(posedge sysclk) `ENS;
```

```
      sum <= `CLK  4;
      odd <= `CLK  5;
    @(posedge sysclk) `ENS;
      sum <= `CLK  sum + odd;
      odd <= `CLK  odd + 2;
    @(posedge sysclk) `ENS;
      sum <= `CLK  sum + odd;
      odd <= `CLK  odd + 2;
    @(posedge sysclk) `ENS;
      sum <= `CLK  sum + odd;
      odd <= `CLK  odd + 2;
  end
```

This implicit style class 2 machine outputs some odd numbers (5, 7, 9, 11, 5, 7 ...) and the corresponding squares (4, 9, 16, 25, 4, 9 ...). It is important to note that adjacent the non-blocking assignments (i.e., without intervening `if`, `@(posedge sysclk)`, etc.) execute in parallel, and so their position in the source code could be reversed and still produce the same result. Note that because of this property of the non-blocking assignment, a machine identical to the above can be defined as:

```
always   //reversed 2nd example
  begin
    @(posedge sysclk) `ENS;
      odd <= `CLK  5;
      sum <= `CLK  4;
    @(posedge sysclk) `ENS;
      odd <= `CLK  odd + 2;
      sum <= `CLK  sum + odd;
    ...
  end
```

In the explicit style, the designer has to break such statements involving variables into separate parts of the code that execute in parallel to the code that caries out the state transitions. For a machine of any complexity, this makes the code hard to read and maintain. The part of the code that carries out the state transitions is know as the *controller*, and the part of the code that carries out the data manipulation is known as the *architecture*. The architecture is often referred to as the *datapath*. Assuming the explicit style class 2 gray code counter shown above (consisting of class 1 and class 0 sub-machines) acts as the controller of the odd/square, the architecture of the odd/square machine can be defined as two class 1 sub-machines and two class 0 sub-machines:

```
wire [4:0] new_sum,new_odd;
assign new_sum = (c==2'b00)?4:sum+odd;
assign new_odd = (c==2'b00)?5:odd+2;
always @(posedge sysclk)
  sum = new_sum;
always @(posedge sysclk)
  odd = new_odd;
```

The variable `c` is now a *command signal*, which tells the architecture what to do. The choice of the gray encoding for the controller was arbitrary. (We actually prefer a one-hot encoding, but an advantage of the implicit style technique is hiding the state encoding.)

Notice that we can categorize each sub-machine at various levels in the hierarchy. In this example, both the controller, which produces the command signal, and the complete system (consisting of the aggregate of the controller and architecture), which produces the squares, are class 2 machines because they do not accept inputs from the outside. The architecture, by itself, is actually a higher class machine, because it accepts the command signal as its input, but when it is connected to the controller, the aggregate of the controller and architecture acts as a class 2 machine.

From a theoretical standpoint, the implicit and explicit styles are equivalent because they produce the same outputs, but from a practical standpoint, there is a huge difference. The implicit style is easy for the designer. The explicit style makes the designer work much harder.

Synopsys, the synthesis vendor who pioneered the use of implicit style, recommends the use of the implicit style for a class 2 machine, which they describe as: "each state in the state machine can only be reached from one other state" [7].

Class 3

Clair defines a class 3 machine as having conditional state transitions and unconditional outputs. This means two things: First, the machine can make decisions based on inputs to determine which one of several possible next state(s) to proceed to. Second, the machine ignores the inputs for the purpose of producing outputs, and so the outputs depend only on the present state. Class 3 machines are commonly referred to as Moore machines.

Behavioral Verilog statements, such as `if`, `while`, and `forever` allow a designer to define an implicit style class 3 machine. For example, the following implicit style machine is in class 3 because it does not begin to proceed through the sequence of squares until an external `start` signal is recognized. Before the `start` signal occurs, `sum` and `odd` are reassigned their initial values.

```
always  //3rd implicit example
  begin
    @(posedge sysclk) `ENS;
      sum <= `CLK  4;
      odd <= `CLK  5;
      if (start)
        begin
          while (odd != 9)
```

```
              begin
                @(posedge sysclk) `ENS;
                  sum <= `CLK  sum + odd;
                  odd <= `CLK  odd + 2;
              end
          end
   end
```

Had the `if` statement been omitted, the aggregate machine would have been categorized as class 2 because its output sequence would not have depended on any external inputs. Regardless of whether the `if` statement is included in the implicit style source code, the equivalent explicit style code will require a class 3 controller. This is because the controller will need to receive the value of `odd` from the architecture to determine whether to stay inside the `while` loop, or whether to return to the top state where the variables will be re-initialized.

For any implicit style code, it is a requirement that `while` and `forever` loop bodies contain at least one `@(posedge sysclk)`. Omission of `@(posedge sysclk)` inside a `while` or `forever` loop body causes problems in both simulation and synthesis. In simulation, such omission may deadlock the simulator. In synthesis, such omission is prohibited because such code cannot be realized as synchronous logic.

In general, implicit style code that uses `if`, `while`, or `forever` needs at least a class 3 controller for its explicit implementation. If the first statement inside every nesting of `if`s and `while`s is not an assignment, the controller is guaranteed to be class 3.

Class 4

Clair defines a class 4 machine as having conditional state transitions and conditional outputs. This means two things: First, as with the class 3 machine, it can make decisions based on inputs to determine which one of several possible next state(s) to proceed to. Second, unlike the class 3 machine, it may produce an output that depends on an external input. Class 4 machines are commonly referred to as Mealy machines.

The gate level implementation of a Mealy machine exhibits a property that Moore machines cannot: if an input makes multiple changes during a clock cycle, the output makes the same number of changes during that cycle. The output is a combinational logic function of the input and the present state. This is why class 0 is not considered a subset of class 3 but class 0 is considered a subset of class 4.

Because of the event drive semantics of behavioral Verilog, it is not possible to write implicit style code that describes this input/output behavior of a class 4 machine. On the other hand, it is easy to write implicit style code that will use a class 4 controller. If the first statement inside any `if` or `while` is an assignment, the controller must be class 4. The registers in the architecture, which form the output of the aggregate system, will only change at the rising edge of the clock, and therefore the aggregate system is only a class 3 machine. When we refer to an implicit style Mealy machine, we are not referring oxymoronically to a class 4 aggregate (controller/architecture) system which would be impossible to implement in the implicit style, but rather to a class 4 controller inside a class 3 aggregate system, which can be described with the implicit style notation.

Here is an example of an implicit style Mealy machine, which is similar to the previous example, except for how `sum` is treated in the top state.

```
always  //4th implicit example
  begin
    @(posedge sysclk) `ENS;
       odd <= `CLK  5;
       if (start)
         begin
           sum <= `CLK  4;
           while (odd != 9)
             begin
               @(posedge sysclk) `ENS;
                 sum <= `CLK  sum + odd;
                 odd <= `CLK  odd + 2;
             end
         end
  end
```

If `start` is not asserted in the top state, the machine remains in that state and `sum` is left alone. This is desirable for keeping the last square computed after the machine has exited the `while` loop. If `start` is asserted in the top state, the class 4 controller commands the architecture to initialize `sum`. The controller is class 4 because the command to initialize `sum` occurs in the same clock cycle when the `start` signal is asserted, but the aggregate system is class 3 because the change to `sum` will not be visible until the next rising edge of the clock. The following table shows how the classification of the controller, architecture, and their aggregate in the three implicit style odd/square examples above effect the `sum` sequence output by each aggregate, assuming `start` is asserted only during the second clock cycle:

```
    class           result
    -----    ---------------------------
2nd 2 3 2    x  4  9 16 25  4  9 16 25...
3rd 3 3 3    x  4  4  9 16 25  4  4  4...
4th 4 3 3    x  x  4  9 16 25 25 25 25...
```

The above lists the class for the controller, the architecture and the aggregate in that order, followed by the result. Note that the architecture automatically synthesized for an implicit style design is guaranteed to be class 3 since it receives command inputs from the controller but it outputs only values from clocked registers. Since all of the outputs of the implicit style aggregate come from the architecture, the aggregate is guaranteed to be class 2 or 3.

It is our opinion that in the vast majority of situations where a Mealy approach is required, it is only required for the control of register transfers in the architecture, and therefore can be implemented in implicit style. For example, the nature of register transfers that occur in a RISC machine are conditioned on several factors, such as whether data forwarding occurs, but the time of such transfers is always synchronized to the clock.

Safe Implicit Style Design

Ideally, the effects of Verilog statements in an IEEE 1364 compliant simulator should be the same as the effects of hardware synthesized from those statements. Unfortunately, there is a semantic gap between synthesis tools and simulators. *Safe design* is when a designer is restricted to a subset of Verilog features whose effects are known to be identical in simulation and synthesis.

Explicit style design is guaranteed to be safe because the designer makes a clean separation of class 0 and class 1 machines in the source code. This separation insures the results of simulation using any IEEE 1364 compliant simulator will agree with synthesis, despite the event driven semantics of the simulator.

The unsafe =

Software languages like C typically only provide one kind of assignment. The timing of this assignment statement (typically denoted by =) is ambiguous. In Verilog simulation, = without intra– assignment delay is defined to be instantaneous and un-interruptable.

Not all synthesis tools accept = in implicit style code. For example, our VITO [1] preprocessor prohibits = inside implicit style code because this restriction simplifies the internal operation of VITO.

Even in commercial synthesis tools that do accept =, there are often restrictions on its use. For example, in Synopsys Design Compiler, if you use = with a particular variable, you cannot ever use <= with that variable. Further, if you use = with any variable in an implicit block, you cannot use <= with any of the other variables in that block [7]. In implicit style code that is destined to be synthesized, we view = like a contagious disease– if you use it all, it will spread throughout your whole design.

If =s were safe, their proliferation would not be a problem. Unfortunately, as we will see, =s are unsafe in implicit style code that is destined to be synthesized. Safety is an issue that only comes up when the same code is to be both simulated and synthesized. We often use =s in special ways for implicit style code that is only simulated, but our guidelines prohibit the use of = in code that is to be synthesized.

To understand why = is unsafe, consider an example of four adjacent =s:

```
  sum = 4;
  odd = 5;
  sum = sum + odd;
  odd = odd + 2;
```

To a simulator, these are indistinguishable from

```
  sum = 9;  //transformed =s
  odd = 7;
```

because the =s do not give the simulator an opportunity to execute another process that could observe the their intermediate values. Synthesis tools that accept =s transform them into the later form.

The synthesized hardware that contains values are D-type (or similar) flip flops, that do not change instantaneously. Therefore, the implicit style designer must specify the schedule using `@(posedge sysclk)` for both simulation and synthesis. For example, we might try the following:

```
@(posedge sysclk);
  sum = 4;  //unsafe
  odd = 5;
@(posedge sysclk);
  sum = sum + odd;
  odd = odd + 2;
...
```

The hardware generated by the synthesis tool for the above includes something equivalent to a mux (with 4 and `sum+odd` as its inputs) whose output feeds a D type register, which contains `sum`. Because, by definition, the output of each D-type flip flop is delayed by one clock cycle, `sum` does not contain 4 until the clock cycle *after* the cycle when the mux outputs the 4. This means `sum` is unknown during the first clock cycle.

Simulation of the above code shows the values of `sum` and `odd` being computed one clock cycle earlier than will actually happen on the synthesized hardware. This means the above code is unsafe. For example, the simulator will report erroneously that `sum` is 4 before the end of the first clock cycle.

An additional difficulty with the = in implicit style code is that adjacent assignments cannot be interchanged arbitrarily. For example, the following is not equivalent to the last example:

```
...
 @(posedge sysclk);
   odd = odd + 2;   //unsafe and
   sum = sum + odd; //backwards
```

because the synthesis tool transforms this into something the designer might not expect:

```
...
 @(posedge sysclk);
   sum = sum + odd + 2; //unsafe and
   odd = odd + 2;       //unexpected
```

Despite the synthesis tool's valiant attempt to deal with =, it is unsuccessful in concealing this semantic gap. The tool is forced to generate hardware whose actions will be different than the original code because the semantics of = are incompatible with a reasonable model of what synchronous hardware does.

Non-blocking assignment

As mentioned earlier, the solution to this semantic gap is to limit ourselves to the other kind of behavioral assignment statement that Verilog provides: the non-blocking assignment (<=). In order to make simulation agree with synthesis, we need to define two macros used earlier. These deal with Entering a New State (`ENS) and CLocKing the registers (`CLK). The reason for using a macro is that these two parts of the code will differ between simulation and synthesis. In synthesis, these two macros are simply empty:

```
`define ENS
`define CLK
```

In simulation, these two macros describe the the proper delay so that the simulation will agree with synthesis:

```
`define ENS #1
`define CLK @(posedge sysclk)
```

The `CLK must be intra—-assignment delay for simulation so that the change to the register does not happen until the next rising edge of the clock. `CLK must be empty for synthesis because some synthesis tools generate incorrect controllers unless the intra—-assignment delay is omitted.

Designers who are new to the implicit style may be confused by the need for #1 in simulation. The definition of <= says that the expression on the right is evaluated instantaneously, but the assignment on the left is scheduled to happen as the *last event* at the $time specified by the intra—assignment delay. In other words, the assignment happens after @(posedge sysclk) #0 but before @(posedge sysclk) #1.

To illustrate, if the implicit style code of the "2nd example" were written without the #1, the design would be unsafe. The first events at the $time of the rising edge would be to evaluate the expressions odd+2 and sum+odd. Later in sequence at that $time would be the storage of values computed in the previous clock cycle. This would produce a wrong sequence for odd (x, 5, x) and sum (x, 4, x).

Therefore, our guidelines for safe implicit style assignments is to use only <= with the `ENS and `CLK macros defined differently for synthesis and simulation.

Bottom testing loops

Although while loops are natural to describe many mathematical algorithms, there are also other kinds of loops. For instance, to interface with external physical equipment requiring prescribed delays, the while loop is not very convenient. Explicit style designers typically solve such problems by using a *bottom testing loop*, which is guaranteed to execute at least once. The advantage of a bottom testing loop is its hardware implementation tends to cost less than an equivalent while loop. Bottom testing loops are also common in software languages such as C (do ... while(cond)) and Pascal (repeat ... until cond).

Recommended Extension to IEEE 1364

Unfortunately, IEEE 1364 does not provide such a bottom testing control statement that can be used safely in both simulation and synthesis. Therefore, we recommend the Verilog standard be extended to include a bottom testing loop, whose syntax utilizes only the existing reserved words of Verilog:

```
repeat
  begin
    ...
  end
while (cond);
```

This can be distinguished from the existing use of repeat in IEEE 1364 by the absence of a repetition count.

To understand why designers often want a bottom testing loop, consider the "3rd implicit example" from section ???. This computes the correct sequence for sum and odd, provided that the start pulse occurs *after* the first clock cycle. If the start pulse occurs in the first clock cycle, the non-blocking assignment of 5 to odd has not yet taken effect. This means Verilog may skip the while loop altogether (if odd happens to be 9 on power up of the physical hardware or odd is 'bx as happens at $time 0 in simulation), and since we assume the start pulse only lasts for one cycle, the squares are never computed. If our extension to IEEE 1364 were adopted, we could generalize this code to handle a `start` pulse at any time as follows:

```
always  //proposed extension
  begin
    @(posedge sysclk) `ENS;
       sum <= `CLK  4;
       odd <= `CLK  5;
       if (start)
         begin
           repeat
             begin
               @(posedge sysclk) `ENS;
                 sum <= `CLK  sum + odd;
                 odd <= `CLK  odd + 2;
             end
           while (odd != 9);
         end
  end
```

This insures that the `odd != 9` test only occurs in the bottom state, and not in the top state as happened with the ordinary `while` loop.

We are realistic that adoption of this natural extension to Verilog may not be quick in coming, especially since its need is justified by implicit style design, which has only recently begun to attract much attention. Therefore, we will discuss three ways to cope with a situation that calls for a bottom testing loop, none of which is completely satisfactory.

Duplicate state(s) before `while`

The goal of the bottom testing loop is to make sure that the state(s) inside the loop execute at least once. One way to accomplish this with a `while` loop is to put an exact duplicate of the loop body code just prior to the `while` statement.

```
always  //5th implicit example
  begin
    @(posedge sysclk) `ENS;
       sum <= `CLK  4;
       odd <= `CLK  5;
       if (start)
         begin
           @(posedge sysclk) `ENS;  //duplicate
             sum <= `CLK  sum + odd;
             odd <= `CLK  odd + 2;
             while (odd != 9)
               begin
                 @(posedge sysclk) `ENS;
                   sum <= `CLK  sum + odd;
                   odd <= `CLK  odd + 2;
               end
         end
  end
```

This code is safe, and so simulation will agree with synthesis. The additional state in the middle is the first time when the `odd != 9` test occurs. Even if the start pulse occurs in the first clock cycle, `odd` will not be tested until the second cycle when it will contain the value 5.

For a simple loop, such as this, with only one state inside, the cost of duplicating the loop body code is reasonable. When there are multiple states inside a bottom testing loop becomes becomes high. The cost grows exponentially with the nesting level of bottom testing loops.

Use a flag variable

Another way to solve this problem is to introduce a flag variable that is set in the clock cycle *prior* to the first clock cycle when the `while` condition is tested. This flag must then be reset in that first clock cycle when that condition is tested. The `while` condition must include `|flag`.

```
always  //6th implicit example
  begin
       flag <= `CLK 1;
    @(posedge sysclk) `ENS;
       sum <= `CLK  4;
       odd <= `CLK  5;
       if (start)
         begin
           flag <= `CLK 0;
           while ((odd != 9)|flag)
             begin
               @(posedge sysclk) `ENS;
                 sum <= `CLK  sum + odd;
                 odd <= `CLK  odd + 2;
             end
         end
  end
```

This approach transforms the controller into class 4, regardless of whether the controller was originally class 3 or class 4. What the flag will be in the next clock cycle depends not only on which state the controller is in, but also on the `start` signal and whether `odd != 9`:

present state	start	odd!=9	next state	next flag
top	0	-	top	1
top	1	-	bottom	0
bottom	-	0	top	1
bottom	-	1	bottom	stays 0

The value of the flag in a particular clock cycle represents which of the two states the machine is in during that cycle. Although, in theory, such information is already available inside the controller, the only way to access this information in the implicit style is by introducing this extra variable.

In a problem where nesting occurs or where there are many states inside a bottom testing loop, the cost of this approach will be significant lower than duplicating the states. Also, in a problem with multiple non-nested bottom testing loops, the one bit flag variable can be reused by each loop, resulting in further cost savings. In general, the number of flag variables required is bounded by the maximum nesting level. Although this approach is safe and cost effective, designers new to the implicit style may find it somewhat confusing.

`disable` inside `forever`

In languages that provide a `goto` statement, implementing a bottom testing loop is trivial. Verilog provides the `disable` statement, which under certain circumstances synthesis tools treat like a `goto`. The problem with the `disable` statement is that its simulation semantics in the context of the non-blocking assignment are ambiguous. An interpretation of the semantics implemented by many simulation vendors produces a simulation result that differs from synthesis.

This technique puts a `disable` statement inside an `if` at the bottom of `forever`, rather than using a `while`:

```
always  //unsafe
  begin
    @(posedge sysclk) `ENS;
       sum <= `CLK  4;
       odd <= `CLK  5;
       if (start)
         begin
           begin : label
             forever
               begin
                 @(posedge sysclk) `ENS;
                   sum <= `CLK  sum + odd;
                   odd <= `CLK  odd + 2;
                   if (odd != 9)
                     disable label;
               end
             end
         end
  end
```

This technique is unsafe is because many simulators do not treat the `disable` just as an ordinary `goto`. Instead these simulators un-schedule any non-blocking assignments that executed during the same clock cycle in which the `disable` executed. Although the number of clock cycles associated with loop execution will be the same as in the physical hardware, the value of the variables will not change properly in the final clock cycle. In this machine, assuming a start pulse during the first clock cycle, synthesis produces the correct result for `odd` (x 5 7 9 11 5 ...) and `sum` (x 4 9 16 25 4 ...). On the other hand, simulation produces an incorrect result for `odd` (x 5 7 9 9 5 ...) and `sum` (x 4 9 16 16 4 ...) because the assignments of 11 to `odd` and 25 to `sum` are undone by the `disable`.

A modest revision to IEEE 1364

There is a more modest way in which IEEE 1364 could be revised to support bottom testing loops than our `repeat ... while` proposal: provide a command line option to Verilog simulators that causes the `disable` to leave non-blocking assignments alone. Since this would be an option, it would not disturb any existing Verilog code that depends on the ambiguous semantics of `disable`. Also, this proposal only effects simulation vendors, since synthesis vendors already agree with us about the semantics of `disable`.

Despite the greater likelihood of this more modest proposal being adopted, we feel it is important to raise the issue of Verilog's control structure deficency. In our view, none of the three approaches for bottom testing loops described above capture the designer's intent as well as our proposed `repeat ... while` syntax.

References

[1] M. G. Arnold, and J. D. Shuler, "A Synthesis Preprocessor that Converts Implicit Style Verilog into One-hot Designs," *Proceedings of the Sixth International Verilog HDL Conference,* Mar. 31–Apr. 3, 1997, Santa Clara, CA, pp. 38-45.

[2] M. G. Arnold, *Verilog Digital Computer Design: Algorithms into Hardware,* a textbook in preparation that makes extensive use of implicit style Verilog.

[3] M. G. Arnold, T. A. Bailey, J. R. Cowles, J. J. Cupal and F. N. Engineer, "Behavior to Structure: Using Verilog and In-Circuit Emulation to Teach how an Algorithm becomes Hardware," *Proceedings of the Fourth International Verilog HDL Conference,* Mar. 27–29, 1995, Santa Clara, CA.

[4] S-T Cheng and R. K. Brayton, "Synthesizing Multi-Phase HDL Programs," *Proceedings of the Fifth International Verilog HDL Conference,* Feb. 26–29, 1996, Santa Clara, CA, pp. 67–76.

[5] C. R. Clair, *Designing Logic Systems Using State Machines*, New York: McGraw-Hill, 1973.

[6] H. Howe, "Pre- and Postsynthesis Simulation Mismatches," *Proceedings of the Sixth International Verilog HDL Conference,* Mar. 31–Apr. 3, 1997, Santa Clara, CA, pp. 24-31.

[7] *HDL Compiler for Verilog Reference Manual,* Version 3.1a, March 1994, Synopsys, Inc., pp. 8–19 and 5–10.

[8] S. Palnitkar, *Verilog HDL: A Guide to Digital Design and Synthesis*, Prentice-Hall, 1996.