A Synthesis Preprocessor that Converts Implicit Style Verilog Into One-Hot Designs

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

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

Abstract

Different synthesis vendors support different subsets of Verilog. One such subset is the implicit style state machine (multiple uses of edge triggered events within an `always` *block). With this style, one can obtain working silicon in less time because the implicit style is more like software design. Unfortunately, most synthesis vendors do not support the implicit style. To make the implicit style more accessible, a freely available synthesis preprocessor is described here that converts implicit style* `always` *blocks into other equivalent Verilog, which can then be synthesized by most commercial synthesis tools. This paper discusses advantages of the implicit style, how the preprocessor translates implicit style code into a one-hot design, and why the language subset (non-blocking assignment) was chosen so that the semantics of the synthesized Verilog can agree with the simulation semantics defined by IEEE 1364. More information about our preprocessor can be found at* `http://plum.uwyo.edu/~vito` *or* `http://cs.brockport.edu/~jshuler/vito/`.

1 Introduction

Various subsets of Verilog are used in different ways by different groups of designers. The main stylistic distinction is whether the Verilog code is intended for simulation or for synthesis. Although IEEE 1364 has done a good job to standardize the semantics of Verilog simulators [7, 10, 11], the situation with commercial Verilog synthesis tools is much less straightforward. Each synthesis vendor has chosen some subset of Verilog syntax, and associated with it a semantics that is sometimes at odds with the simulation semantics defined by IEEE 1364. Furthermore, different synthesis vendors support different subsets of Verilog.

Synopsys Design Compiler [6] is probably the most advanced synthesis tool on the market, allowing the designer to work at a higher level of abstraction than many of its competitors. This higher level, which Synopsys refers to as the *implicit style*, consists of behavioral Verilog with multiple uses of edge triggered events within a block to describe state transitions driven by one single phase clock signal. Recent experimental work [3] has shown how this capability can be extended for multiphase state machines, however we restrict our discussion below to single phase state machines.

The subset of behavioral Verilog that Synopsys Design Compiler processes is quite large and powerful. Provided that every possible loop contains at least one edge triggered event, Design Compiler can extract the state transitions implied by arbitrarily nested control structures (including `while`, `forever` and `disable`). To our knowledge, no other commercially available Verilog synthesis tool has this capability. Within the Verilog community, this powerful feature is not as widely appreciated as we believe it should be. There are perhaps three reasons for this under-appreciation of implicit style state machines. First, the implicit style is much closer to modern software design practices than to the traditional hardware design practices that require a designer to specify explicit state transition (herein referred to as *explicit style*). Second, Synopsys has not promoted implicit style state machines. In fact, Synopsys only recommends their use for trivial cases “with a single flow of control ... where each state ... can only be reached from one other state” [6]. Finally, since to our knowledge this style of synthesis is currently only available through one rather expensive tool, not all designers have access to it.

It is our goal to overcome these objections to the implicit style. The first and second objections are to some extent subjective, and our answers to these points are discussed in the following sections. The final objection is quite concrete—if designers do not have access to a synthesis tool that supports the implicit style, they will be forced to translate the implicit style manually into the more common explicit style which is supported by all synthesis tools. We find the implicit style so desirable that we have used it this way for several years as a

Proceedings of the 1997 IEEE International Verilog HDL Conference (IVC '97)
1085-9403/97 $10.00 © 1997 IEEE

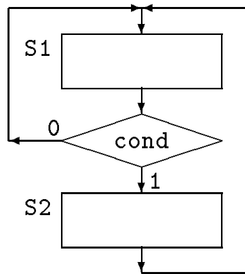

Figure 1: ASM example.

simulation-only style with a three stage manual process to translate to synthesizable Verilog [1, 2]. (In [2], the implicit style is referred to as "pure behavioral" and the explicit style is referred to as "pure structural".) However, for designers without access to a tool that supports the implicit style, it is understandable but regrettable that most designers prefer to skip over the implicit style, and directly code their designs in the explicit style.

To make the implicit style more accessible to all Verilog designers, we have developed a synthesis preprocessor that converts certain kinds of implicit style `always` blocks into equivalent Verilog that can be synthesized by most commercial synthesis tools. It is called `VITO` for Verilog Implicit To One-hot. This tool was developed jointly at the University of Wyoming and the SUNY College at Brockport. It is freely available at `http://plum.uwyo.edu/~vito/` or `http://cs.brockport.edu/~jshuler/vito/`.

2 Graphic descriptions of state machines

The Algorithmic State Machine (ASM) chart is a flowchart-like notation used to describe state machines invented by T. E. Osborne at Hewlett Packard in the early 1960s, and popularized by [4]. Figure 1 shows how a simple state machine can be described as an ASM chart.

This trivial example does not fully illustrate why the ASM chart notation is vastly superior to the bubble chart notation. One main advantage is that the ASM chart describes decisions separately (as diamonds), analogously to how Verilog describes decisions separately (as, for example, `if` statements). In the ASM chart, the testing described by diamonds occurs in parallel with the states to which the diamonds belong, just as nested `if` statements in Verilog execute at the same `$time`. In this example, the one diamond belongs to state S1, and so the testing of `cond` occurs in parallel to whatever other actions (not shown in this example) that are associated with state S1.

3 Disadvantages of explicit style

An alternative to a graphical description is an explicit state transition table. By tracing every possible path through the ASM chart, the designer can determine the state transitions:

present_state	cond	next_state
S1	0	S1
S1	1	S2
S2	0	S1
S2	1	S1

The above table describes the combinational logic that outputs the next state, given the `present_state` and `cond` inputs. Since we are restricted to synchronous design, the `next_state` is loaded into the `present_state` flip flop(s) at the clock edge.

The explicit style of describing a state machine in Verilog requires the designer choose a particular coding for these states. To aid in readability, the codes chosen by the designer can be indicated with `` `defines``. For example, to use a one-hot encoding each state corresponds to a constant that has only a single bit that is 1:

```
`define S1 2'b01
`define S2 2'b10
```

There are many ways that this machine could be coded in explicit style Verilog, but typically the designer has to declare a function including a `case` statement to describe the combinational logic that computes the `next_state`. The `case` statement uses the `present_state` to determine how the next state is computed:

```
function [1:0] state_gen;
  input ps,cond;
  reg [1:0] ns;
  begin
    case (ps)
      `S1: if (cond) //1st state
             ns = `S2;
           else
             ns = `S1;
      `S2: ns = `S1; //2nd state
      default: ns = `S1;
    endcase
    state_gen = ns;
  end
endfunction
```

The explicit style also requires the designer declare the `present_state` and `next_state`, and use an `always` block that will synthesize into edge triggered flip flops:

```
wire [1:0] next_state =
  state_gen(present_state,cond);
reg [1:0] present_state;

always @(posedge sysclk or posedge reset)
```

```
if (reset)
  present_state = `S1;
else
  present_state = next_state;
```

In summary, the explicit style places several burdens on the designer: First, the designer has to worry about the explicit state encodings. Second, the designer has to explicitly declare the `present_state` and `next_state` to be large enough to hold the state encodings. Third, the designer has to describe explicitly how to `goto` the `next_state` under every possible circumstance, and the designer has to declare a function (or other Verilog feature) that will synthesize into the combinational logic required to compute this `next_state`. Finally, the designer has to define an `always` block that will synthesize into the `present_state` register.

The resulting explicit style Verilog code has several disadvantages in contrast to the ASM chart notation. First, the relationship between the ASM chart and the `case` statement is often unclear on casual inspection. Even though this is a trivial example, if one were only shown the explicit style Verilog, the simplest way to comprehend what it does is to draw an equivalent ASM chart, rather than trying to trace through the `case` statement itself.

Second, the `case` statement approach is verbose. An understandable ASM chart that describes a moderately complex design, such as a RISC microprocessor, can fit neatly on one page, but the same design will require an overwhelming `case` statement with hundreds of lines of Verilog [1].

Third, the `case` statement is hard to maintain. Insertion of additional states into the algorithm may require re-declaring everything to match the new size of the state encoding. Also, changing one decision may require making more than one change within the `case` statement.

Fourth, the explicit style `case` statement approach forces the designer to work at a lower level of abstraction than is desirable. Although the `case` statement is behavioral code, it does not truly describe the behavior desired by the designer, but rather it describes the behavior of a lower level interpreter. By analogy, the ASM chart description of hardware is to the explicit style description of hardware as C++ (or similar high level) software is to machine language software.

Finally, the `case` statement approach violates one of most widely accepted tenets of modern software design [5]: one should avoid the use of unnecessary `goto`s. In effect, each state transition is an explicit `goto` to some other state.

4 Advantages of the implicit style

The implicit style for describing a state machine in Verilog directly corresponds to the ASM chart notation. Assuming a rising edge triggered `sysclk`, every rectangle corresponds to a particular `@(posedge sysclk)` within the implicit style code. The actions listed inside each rectangle, if any, correspond to behavioral Verilog statements that follow the `@(posedge sysclk)`. Diamonds correspond to Verilog statements, such as `if` and `while`. For example, the state machine described in the previous sections can be written in the implicit style as:

```
always
  begin
    @(posedge sysclk)
      // 1st state
      if (cond)
        begin
          @(posedge sysclk)
            // 2nd state
        end
  end
```

The relationship between the ASM chart and the implicit style Verilog is obvious. There is a one to one mapping between the ASM chart and the implicit style Verilog. Note that the semantics of Verilog insure that the decision occurs in parallel with whatever other actions are desired in the "1st state". Clearly, the implicit style code is shorter and more understandable than the equivalent explicit style.

The implicit style is easy to maintain. For example, to modify the code so that it loops in the "2nd state" as long as `cond` is true only requires changing the word `if` to `while`, but in the explicit style this change would require total replacement of the code for the "2nd state". If a further change were required such that the machine loops in that state while another condition is true, only one place in the implicit style code would have to be modified. On the other hand, two places in the explicit style code would have to be changed to accomplish the same thing, since the change effects the `goto`s from both states. It is reasonable to expect that a designer might forget to make such changes consistently. An advantage of the implicit style of Verilog and the ASM chart notation is that a decision, such as in a `while` loop, can be shared with several states, without having to denote the condition more than once. In contrast, the explicit style of Verilog and the bubble diagram both have the deficiency that the conditional `goto`s from each state must be redundantly recorded in a state machine that uses a `while` loop.

A related advantage is that inserting an additional state into the implicit style code only requires inserting an additional `@(posedge sysclk)`, whereas to make similar modification to the explicit style code requires

Proceedings of the 1997 IEEE International Verilog HDL Conference (IVC '97)
1085-9403/97 $10.00 © 1997 IEEE

changing several size declarations as well as editing all states that `goto` the new state.

5 Subset of Verilog supported

VITO accepts as its input a single valid Verilog module that supposedly contains one or more implicit style state machines, optionally with other Verilog features that our tool leaves alone. The output of VITO is equivalent Verilog that can be synthesized by any tool. We make no claim that the resulting netlist is as efficient as would be produced by Synopsys. We have only implemented a subset of the capabilities that exist in the Synopsis Design Compiler. The subset was chosen to be powerful enough for realistic implicit style designs, but simple enough to make implementation of VITO straightforward.

Since synthesis is being performed by a commercial tool, our strategy is to pass most of the source code through unmodified. The primary transformation of the source code occurs to `always` blocks that the tool suspects of being implicit style machines. For simplicity, we define `always` blocks that are immediately followed by `begin` as being implicit style state machines. There can be at most 998 statements (as described below) within all such implicit style blocks of the module.

The behavioral keywords that VITO allows in an implicit style block are: `begin`, `end`, `if`, `else`, `while`, and `forever`. The `case` (and similar) statements are not allowed inside an implicit style block, but this limitation can be circumvented in many situations by defining a function called from the implicit style block. Although functions may be in implicit style blocks, tasks are ignored. Only non-blocking assignment (`<=`) is allowed.

VITO is configurable in several ways. The name of the clock, and whether it is `posedge` or `negedge`, can be chosen globally for the entire module. However, having chosen these settings, the designer may only use one style of `@` within implicit style blocks. In this paper, we will assume the designer has chosen `@(posedge sysclk)`. Similarly, the designer may choose the name of the reset signal, and whether it is active-high or active-low. We will assume the designer has chosen an active-high `reset`. Whatever names the designer chooses, it is a requirement that the chosen clock signal (`sysclk`), and chosen reset signal (`reset`) occur within the port list of the single module being translated.

Features disallowed by Synopsis, such as `->`, `wait`, `fork`, and `deassign` are either disallowed, or passed through unmodified to the synthesis tool (which will probably produce an error as a consequence.)

6 Operation of the preprocessor

VITO converts an implicit style description of a synchronous machine into equivalent continuous `assigns` and `always` with one `@` and one `reg`. The new description is based on the one-hot controller methodology [9], and consists of two parts—a controller and an architecture. The *controller* portion is based on the flow of control in the ASM while the *architecture* portion is based on the actions performed in each state of the ASM. The generation of the controller is discussed in the following subsections while the generation of the architecture is discussed in a later section.

6.1 Pass one

VITO uses three passes. The first pass parses the source module separating it into two files. The first file contains the portions of the module that VITO will leave unmodified. The second file contains the remaining portions of the module that our tool identifies as being implicit style blocks. The first file is used in pass three. The second is used in pass two.

Pass one is implemented using *lex* and *yacc*. The result of pass one breaks each implicit style block apart into what we refer to as statements. These do not entirely correspond to a statement in the syntax, but also include isolated keywords, such as `begin` and `else` that play a pivotal role in the fabrication of the one-hot controller. Pass one will insert such statements into the design as necessary. Pass one numbers these statements sequentially starting at 1, up to a maximum of 998.

6.2 Pass two

Pass two works only with the implicit style blocks identified during pass 1. Pass two creates a directed graph representation that describes the control flow within each block. This graph is different than either the ASM chart or the bubble chart, because the graph describes where statements "*come from*" rather than where they "go to". In the subset of Verilog supported, for any given statement, there are at most two other statements that the machine could possibly be coming from. For example, immediately following an `if`, control can only "come from" the "then" or the "else" parts. This graph is similar to the ASM chart (but different than the bubble chart) in that it shows connections between statements that occur in parallel (i.e., like the connection between the rectangle and the diamond.)

The graph created by VITO is represented with an n by 2 matrix of integers. In this matrix, zero means no connection, positive means an unconditional connection or false conditional connection (as in skipping over the body of an `if`), and negative means true conditional connection (as in executing the body of an `if`). The following illustrates the matrix created for the implicit style code shown earlier that corresponds to Figure 1:

```
1: 999  10 always
2:   1   0 begin
3:   2   0 @(posedge sysclk)
```

Proceedings of the 1997 IEEE International Verilog HDL Conference (IVC '97)
1085-9403/97 $10.00 © 1997 IEEE

```
 4:   3   0 // 1st state
 5:   4   0 if(cond)
 6:  -5   0 begin
 7:   6   0 @(posedge sysclk)
 8:   7   0 // 2nd state
 9:   8   5 end
10:   9   0 end
```

The number to the left of the colon is the statement number (the row number of the matrix). The two numbers to the right of the colon are the contents of that row of the matrix. Statement number 999 is reserved for the initialization of the machine when the `reset` signal is active.

From the matrix, it is clear statements 2 through 5 only have one possible "come from" path. For example, on row 2 of the matrix, the only place from which control can "come from" is row 1, since the zero in the second column indicates there is not another statement that goes to statement 2.

Row 6 is different because it has a negative number. This corresponds to "coming from" an `if` statement when the condition (`cond`) is true. Rows 7 and 8 have only one possible path.

Row 9 shows two possible paths that can reach the `end` statement. One possible path is the path from 8 to 9 that occurs when exiting from the `if` body (`cond` is true). The other is the path from 5 to 9 that occurs when skipping over the `if` body (`cond` is false).

Row 10 shows there is only one path from 9 to 10. In order to see where 10 goes to, one needs to look back at row 1. In addition to 999 (which only occurs at `reset`), row 1 can "come from" 10. This is what describes the looping behavior of the `always`.

6.3 Pass three

Pass three uses the matrix representation produced in pass two to generate Verilog code that represents a one-hot controller whose state transitions are identical to those of the original implicit style block. During pass three, VITO produces Verilog code that uses several `wire`s and `reg`s not found in the original implicit style block. The names of these `wire`s and `reg`s consist of a prefix string concatenated to the statement number with which the `wire` or `reg` is associated. The prefix strings are configurable so that a designer may insure no conflict exists between the names in the original implicit style block and the names generated by VITO.

At the beginning of pass three, a `wire` is declared to correspond to each statement. Although many of these `wire`s might be synonymous (when certain statements execute in parallel to each other), generating a declaration for all of them simplifies the design of VITO. The name of each `wire` begins with a configurable prefix (`s_` in this example) followed by the number of the corresponding row in the matrix. Because of the possibility of arbitrary loops and decisions within an implicit style block, both forward and backwards references to these `wire`s might occur. Therefore, it is necessary to declare them prior to using them. (The other `wire`s may be declared at the point in the one-hot controller that they are needed, but the `s_ wire`s must be declared prior to use.) In the example given earlier, the first line that VITO outputs during pass three is:

```
wire s_1,s_2,s_3,s_4,s_5,s_6,s_7,s_8,s_9,s_10;
```

Every one-hot controller needs to begin with only one flip flop "hot" after the machine is reset. One way to accomplish this is to introduce a hidden "state", known as `s_999`, that is one when the machine is reset asynchronously, and zero after the first clock edge arrives. This, in essence, is a flip flop with the D input tied to one and the $\overline{\mathrm{Q}}$ output tied to `s_999`. Regardless of the specifics of a particular design, VITO outputs the following seven lines to describe `ff_999`:

```
reg ff_999;
always @(posedge sysclk or posedge reset)
  if (reset)
    ff_999 = 0;
  else
    ff_999 = 1;
wire s_999 = ~ff_999;
```

VITO next transforms each statement extracted by pass one in light of the information from the matrix. If the statement is neither `@(posedge sysclk)` nor a decision statement, default processing occurs.

The *default processing* in pass three simply generates a continuous assignment to define the `s_ wire` corresponding to that statement. The right hand side of the continuous assignment depends on the corresponding row of the matrix. If there are two non-zero values in that row, the right hand side is an OR of the corresponding `s_ wire`s. If the first value on the row is non-zero, but the second value on the row is zero, the right hand side is only the `s_ wire` that corresponds to the first value. If both are zero, the right hand side is `1'b0`, which indicates an unreachable statement (such as a statement that follows a `forever` block.)

In the example given earlier, `s_1` corresponds to `always`. There is no decision or state transition associated with this line, therefore, default processing occurs. Row 1 of the matrix contains two non-zero values (999 and 10), and so VITO generates:

```
assign s_1 = s_999|s_10;
```

Similarly, `s_2` corresponds to `begin`, which causes default processing. However, in this case, row 2 of the matrix (1 and 0) contains only one non-zero value, and so VITO generates:

Proceedings of the 1997 IEEE International Verilog HDL Conference (IVC '97)
1085-9403/97 $10.00 © 1997 IEEE

```
assign s_2 = s_1;
```

The next line, s_3, corresponds to @(posedge sysclk), which causes something other than default processing. The @(posedge sysclk) indicates the boundary between previous state(s) and the current state. Just as in the earlier statements, there may be two, one or zero ways of reaching this point, based on the matrix row, however now the wire that represents this is called tmp_3, rather than s_3. The intention is that tmp_3 is the D input to a flip flop whose Q output will connect to s_3. To synthesize this, ff_3 must be declared as a reg. Next, an always block describes a flip flop with an asynchronous reset. Finally, the flip flop's output is assigned to s_3 using a continuous assign. (This is necessary since s_3 was declared to be a wire earlier, but ff_3 must be a reg to synthesize as a flip flop.)

```
wire tmp_3 = s_2;
reg ff_3;
always @(posedge sysclk or posedge reset)
  if (reset)
    ff_3 = 0;
  else
    ff_3 = tmp_3;
assign s_3 = ff_3;
```

The next line (s_4) is simply another example of default processing of a statement that can be reached through only one path:

```
assign s_4 = s_3;
```

There is an if statement corresponding to s_5, which causes non-default processing during pass three of VITO. Again, there may be two, one or zero ways of reaching this point, based on the matrix row, but the wire that represents this is called tmp_5, rather than s_5. The condition that determines the outcome of the if statement is assigned to qual_5. There are two wires that must be computed here: s_5 which corresponds to the 5 elsewhere in the matrix, and sT_5, which corresponds to the -5 elsewhere in the matrix. The s_5 wire is true when the one-hot controller has reached this decision point (tmp_5) and the if condition is false (~qual_5). The sT_5 wire is true when the one-hot controller has reached this decision point (tmp_5) and the if condition is true (qual_5):

```
wire tmp_5 = s_4;
wire qual_5 = (cond);
assign s_5 = tmp_5&(~qual_5);
wire  sT_5 = tmp_5&qual_5;
```

The next line (s_6) is an another example of default processing, however this time the -5 on row 6 of the matrix refers to the wire sT_5, which was defined above. Therefore, s_6 is conditional:

```
assign s_6 = sT_5;
```

The next line (s_7) is another @(posedge sysclk). This generates Verilog code that synthesizes as another flip flop (ff_7):

```
wire tmp_7 = s_6;
reg ff_7;
always @(posedge sysclk or posedge reset)
  if (reset)
    ff_7 = 0;
  else
    ff_7 = tmp_7;
assign s_7 = ff_7;
```

Note that flip flops are only generated for each @(posedge sysclk) in the original implicit style block (ff_3 and ff_7) plus one flip flop for resetting the machine (ff_999). Such one to one correspondence between the original implicit style block and the synthesized one-hot controller makes the operation of VITO relatively simple.

The next line (s_8) is another example of default processing:

```
assign s_8 = s_7;
```

The following line (s_9) corresponds to the end statement that terminates the if block. Row 9 of the matrix (8 5) indicates two paths lead to this point. One path "comes from" the preceding statement (s_8) and the other path "comes from" the false if condition (s_5). Because there are two paths, s_8 and s_5 are ORed together:

```
assign s_9 = s_8|s_5;
```

Finally, s_10 is the last example of default processing:

```
assign s_10 = s_9;
```

At this stage, pass three processing is done, and the one-hot controller is complete.

7 Use of <=

The preceding example only considered control statements, such as if. Although control statements are essential to describe complex machines in the implicit style, by themselves, they are not capable of doing any useful work. Some form of procedural assignment statement is necessary so the machine can do computations and produce outputs. Verilog provides a rich selection of procedural assignment statements that can be used in simulation. However, we have a goal for VITO that limits which kind of assignment it can translate. Our goal is that a simulation of the original implicit style source code and a simulation of the code generated by VITO give identical results. To achieve our goal, we have chosen to limit assignment within an implicit style block to the non-blocking procedural assignment (<=).

Proceedings of the 1997 IEEE International Verilog HDL Conference (IVC '97)
1085-9403/97 $10.00 © 1997 IEEE

8 Generating the architecture

In order to perform the data manipulations described by <=, hardware registers and combinational logic must eventually be synthesized capable of carrying out these operations. We refer to this group of hardware elements as the *architecture* [9], also known as the *datapath* [8]. Since our goal is not to actually synthesize the machine, but instead produce Verilog statements that can be synthesized by a commercial tool, we do not attempt to instantiate hardware modules to implement the architecture, as described in [2]. Instead, to generate the architecture, VITO uses the same subset of Verilog also used for the one-hot controller (continuous `assign`s and `always` with one `@` and one `reg`).

For each `reg` that VITO observes during pass one, VITO generates a similar `wire` declaration of the same width. The names of these `wire`s begin with a configurable prefix (`new_` in the following example.) These `wire`s allow continuous `assign` statements to describe the combinational logic that computes what the values of each corresponding `reg` will become at the next clock edge.

During pass three, VITO makes a table of each non-blocking assignment statement. For each statement, the table contains three items: the `s_`number corresponding to that statement in the original implicit style block, the left hand value (*lval*) being assigned, and the right hand expression (*rval*). At the end of pass three, this table is sorted by lval. Each group of lines in the sorted table that have the same lval are used to generate one continuous `assign` statement. The `wire` assigned is the `new_ wire` corresponding to the lval. Each entry in the group generates a conditional operator (`? :`). The `? :` says if the corresponding `s_` number is active, the value of this `new_ wire` will be the corresponding rval. The default value returned from the continuous assign is the lval `reg`. This default value applies when none of the listed `s_` numbers are active.

As an example of how VITO generates the architecture, consider the following machine, which uses a very simplistic data transfer protocol:

```
module oc_mach(cond,in,out,reset,sysclk);
 input cond,reset,sysclk;
 input [3:0] in;
 output [3:0] out;
 wire cond,reset,sysclk;
 wire [3:0] in;
 reg [3:0] out,t;
 always
  begin
   @(posedge sysclk) #1;
    t <= @(posedge sysclk) in;
    if (cond)
     begin
      @(posedge sysclk) #1;
       out <= @(posedge sysclk) ~t;
     end
  end
endmodule
```

The `#1`s and the `@(posedge sysclk)`s inside each `<=` are explained later. Besides the required `sysclk` and `reset`, this machine has two inputs: `in` and `cond`. The machine also has one output, `out`. When `cond` is not asserted, the machine stays in the first state, where it schedules the loading (at the next clock edge) of a local temporary register, `t`, with the current value of `in`. The machine leaves `out` alone in the first state. In the clock cycle after `cond` is asserted, the machine visits the second state, where the one's complement of `t` is scheduled to be loaded into `out`. Therefore, `out` becomes the one's complement two clock cycles after the time when `cond` is asserted and the corresponding data is presented on `in`.

The controller is identical to the one described earlier for Figure 1. The only new thing here is the introduction of <=, which requires the generation of the architecture. During the parsing, VITO passes through the `reg` declaration unmodified:

```
reg [3:0]out,t;
```

At the same time, VITO also generates a related declaration that did not exist in the original source code:

```
wire [3:0]new_out,new_t;
```

During pass three, VITO notes that `t` is assigned when `s_4` is active and `out` is assigned when `s_8` is active. Since the table of this information is sorted, the first portion of the architecture generated is for `out`:

```
 assign new_out =
   s_8 ?  ~t  :
          out;
 always @(posedge sysclk)
   out = new_out;
```

When `s_8` is not asserted, `new_out`, the D input to the register, simply remains as the current value of `out`. Together with the associated `always` block, this corresponds to what is commonly referred to as a synchronous enabled D type register. VITO generates similar code for `t`:

```
 assign new_t =
   s_4 ?  in  :
          t;
 always @(posedge sysclk)
   t = new_t;
```

Consider a different machine which also uses the same controller as Figure 1. It sequences through the first eight odd integers, `r[3:0]`, and their sums, `r[9:4]` (which constitute the first eight squares), after `cond` causes the machine to initialize `r` to the beginning of the sequence:

1085-9403/97 $10.00 © 1997 IEEE

```
reg [9:0] r;
always
 begin
  @(posedge sysclk) #1;
   r <= @(posedge sysclk)
     {r[9:4]+r[3:0],r[3:0]+4'h2};
   if (cond)
    begin
     @(posedge sysclk) #1;
      r <= @(posedge sysclk) 1;
    end
 end
```

Assuming appropriate declarations, the architecture generated is:

```
wire [9:0]new_r;
assign new_r =
 s_4 ? {r[9:4]+r[3:0],r[3:0]+4'h2} :
 s_8 ? 1 :
       r;
 always @(posedge sysclk)
   r = new_r;
```

9 IEEE 1364 <= versus RTL <=

The preceding examples use `@` inside `<=` and `#1`s at the beginning of each state to insure that IEEE 1364 simulation agrees with post-VITO simulation and synthesis. On the other hand, Synopsys Design Compiler would not synthesize the correct hardware given the above code. To synthesize these machines correctly with Design Compiler, the `@`s inside `<=` and the `#1`s would be omitted, in what Synopsys calls *RTL* style. For example, in Synopsys the second state of the last example needs to be:

```
 @(posedge sysclk)
   r <= 1;
```

Unfortunately, as Synopsys admits, "because Design Compiler ignores delay information, synthesis may not agree with simulation [6]." With such RTL code, one cannot rely on pre-synthesis simulation. VITO treats `r <= 1;` the same as `r <= @(posedge sysclk) 1;` and so VITO is compatible with both IEEE 1364 and Synopsys RTL. However, we suggest using the IEEE 1364 style because this insures simulation agrees with synthesis.

10 Conclusion

We have developed a Verilog synthesis preprocessor that translates a highly useful subset of behavioral implicit style Verilog into continuous `assign`s and simple `always` blocks that we believe can be synthesized by any reasonable commercial tool. The subset of Verilog discussed here was chosen so that cycle based simulation of the original implicit style block can be relied upon to predict accurately the behavior of the synthesized machine. Our goal is not to provide a commercial quality tool capable of producing as efficient a netlist as Synopsys does, but rather our goal is to provide a tool that will allow designers to experiment with the implicit style. It is then our hope that designers who become convinced, like us, of the superiority of the implicit style will urge their vendors to support it in a way that is consistent with IEEE 1364.

Acknowledgment

The authors would like to thank Freddy Engineer of MINC, Inc. for help with Synopsys Design Compiler.

References

[1] M. G. Arnold, *Verilog Digital Computer Design: Algorithms into Hardware,* a textbook to be published by Prentice Hall, PTR in 1998, that makes extensive use of implicit style Verilog.

[2] 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.

[3] 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.

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

[5] E. W. Dijkstra, "Goto Statement Considered Harmful," *Communications of the ACM,* vol. 11, no. 3, pp. 147–148, Mar. 1968.

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

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

[8] N. Pappas, *Digital Design,* West, 1994.

[9] F. P. Prosser and D. W. Winkel, *The Art of Digital Design,* Prentice-Hall, 1987.

[10] D. E. Thomas and P. R. Moorby, *The Verilog Hardware Description Language, Third Edition,* Kluwer, 1996.

[11] *Verilog Hardware Description Language Reference Manual,* Version 2.0a, Los Gatos, CA, Open Verilog International, March 1993.

1085-9403/97 $10.00 © 1997 IEEE