Blocking and Non-blocking Assignments in Explicit and Implicit Style Verilog Synthesis

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

Jerry J. Cupal
Electrical Engineering Dept.
University of Wyoming
Laramie, Wyoming 82071
jcupal@uwyo.edu

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

Abstract

We show transformations that convert blocking assignments into non-blocking assignments. Such transformations are useful because the parallel-processing nature of hardware is more easily conceptualized and mapped to technology with non-blocking assignment. To validate our theory, we synthesized using Synopsys FPGA Express, which supports both blocking and non-blocking assignments.

Some of these transformations apply to both explicit- (IEEE P1364.1 proposed RTL synthesis standard) and implicit- (multiple clocking events in an `always` *block) style Verilog. The more complicated transformations apply only to implicit-style code. We also notice there is a one-clock-cycle mismatch between implicit-style synthesis and simulation, which is, in general, solvable only for non-blocking assignment. We avoid this mismatch by modifying our testbench. We feel the complexity of transforming blocking to non-blocking assignment as well as testbench mismatch justify our decision to implement only non-blocking assignment in our Verilog Implicit to One-hot (VITO) preprocessor.*

Explicit versus Implicit

There are two styles of behavioral code that can be synthesized: implicit and explicit. Explicit-style code is more restrictive than implicit-style code in two main respects: explicit-style code prohibits the use of `while` or `forever` loops, and explicit-style code allows only one clocking-event expression per `always` block (which must occur at the top of the block). Explicit-style code is accepted by all synthesis vendors and is the basis of the proposed IEEE P1364.1 standard [5] for the so-called "Register Transfer Level Subset" of Verilog.

Implicit-style code [1, 6] relaxes these restrictions so that multiple clocking events as well as `while` and `forever` loops are allowed, provided that the clocking events are identical (shown as `@(posedge sysclk)` in the following examples) and each loop contains at least one clocking event. Several vendors, including Synopsys and Cadence, accept implicit-style code for synthesis. Also, a freely available preprocessor known as VITO [3] can translate a very usable (and growing[1]) subset of implicit-style Verilog into explicit-style Verilog compliant with IEEE P1364.1. Thus, synthesis design flows for implicit-style Verilog are available to all Verilog designers who wish to use this style.

Blocking versus Non-blocking

Verilog provides two kinds of behavioral assignment: the blocking assignment (`=`), which is similar to software assignment statements found in most conventional programming languages, and the non-blocking assignment (`<=`), which is the more natural assignment statement to describe many hardware systems, especially for synthesis. Cummings [4] expresses the view of many experienced Verilog designers that blocking assignments should be used only in a few situations, such as for modeling combinational logic, for defining functions, or for implementing testbench algorithms.

All IEEE P1364.1 compliant synthesis tools are required to support both blocking and non-blocking assignments in explicit-style code, with the restriction that each variable and each block may use only one or the other kind of assignment. Synopsys [7] supports both blocking and non-blocking assignments in implicit-style code with the same restrictions.

Parallel Processing

When we implemented VITO three years ago, we restricted implicit-style assignments to only non-blocking assignments for ease of implementation of our preprocessor. Although we have a philosophical bias against the blocking assignment, we would have implemented the blocking assignment if we could have devised an easy

[1] VITO 1.3 (`www.verilog.vito.com`) now supports `disable`, `case`, `wait`, `fork` and memories.

way to to do so. As this paper will show, there is no easy way. VITO is essentially a context-free preprocessor that translates each assignment statement of an implicit-style block independently into a portion of a one-hot state machine. The one-hot technique we used with VITO was independently discovered by Niklaus Wirth and coworkers at the Swiss Federal Institute of Technology [8], and so we feel quite confident that the VITO approach is sound. (Wirth is the well-known creator of the Pascal, Modula-2 and Oberon software languages.)

The state machine created by VITO controls a mux for each left-hand variable in the original implicit-style code. The inputs to the mux are the right-hand expressions assigned to the corresponding variable. The value of each variable is stored in a synchronous D-type register. This translation can be done without much context (and without a symbol table) because of the order-independent semantics of the non-blocking assignment statement. (The meaning of `a<=b` is the same regardless of earlier or later statements.) Also, by adhering to some simple restrictions [2], one can guarantee that implicit-style simulation matches post-VITO synthesis (using the criteria given in IEEE P1364.1 section 3.1.2).

A recent thread in `comp.lang.verilog` expressed the view that non-blocking assignment is more primitive and backwards than blocking assignment and that non-blocking assignment is for designers who are stuck with an antiquated view of thinking in terms of flip-flops rather than in terms of the problem being solved. A similar view was expressed during questions at a presentation of an earlier paper about VITO. We disagree with such a narrow view. We feel that non-blocking assignment is the proper notation for modeling hardware, not just because it has a straightforward mapping to technology, but rather we feel the mapping is more straightforward because non-blocking assignment captures the essential aspect of hardware: parallel processing. If there is no need for parallel processing in a problem, then there is usually no need to be designing new hardware. (Software running on an off-the-shelf microprocessor would suffice.)

To illustrate our point, consider the work Wirth has done with hardware compilation. To prototype hardware compilation, Wirth incorporated many of the concepts of implicit-style Verilog (with different syntax) into a behavioral HDL based on his Oberon language. This can then be translated into hardware using a one-hot technique similar to VITO. In Wirth's language, a semicolon acts like a clocking event in Verilog and a comma acts like a semicolon in Verilog. All of the assignments in Wirth's language are non-blocking. Wirth's conclusion [8] is:

> The strength of hardware lies in the potential for subcircuits to operate concurrently. Although this is also a topic in software design, we must be aware that genuine concurrency is only possible if we have concurrently operating circuits to support the software concept. Thus much of the work on parallel computing in software actually ends up implementing only quasiconcurrency–pretended concurrent execution–conveniently hiding the underlying sequentiality. This leads me to contend that any scheme of direct hardware compilation ... must include the facility to specify concurrent, parallel statements. Such a hardware programming language may indeed be the best way to let programmers specify parallel statements, which we call fine-grained parallelism.

Clocking Events

In both explicit- and implicit-style Verilog, assignment statements are preceded and followed by a clocking event. In the explicit style of IEEE P1364.1, those clocking events are the same `@(posedge sysclk)` at the top of the `always`. In the implicit style, those clocking events may relate to distinct states. Therefore, in either implicit- or explicit-style code, changes occur only at clock edges, forming what is commonly referred to as a Moore machine.

Assignment Transformations

Every non-blocking assignment can be translated into two blocking assignments: one that evaluates the right-hand side and saves the result in a temporary variable, and the other that later stores the temporary value into the variable in question [1].

It is also true that every blocking assignment can be translated into a series of non-blocking assignments, but the transformation is context sensitive, and sometimes quite counterintuitive. The purpose of this paper is to illustrate some of these transformations, which apply to both synthesis and simulation. The first examples apply equally to both explicit- and implicit-style code. The more intricate examples shown later apply only to implicit-style code

Single Assignment

The first rule in translating from blocking to non-blocking is that there is no difference between the two when there is only one statement between clocking events. For example, the explicit-style code with blocking assignment

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

is identical to a similar non-blocking assignment

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

when considered in isolation (assuming the `$strobe` task is used to observe `a` after the non-blocking assignment has occurred). Interaction of several such `always` may create other problems for blocking assignments [4].

Two Assignments

When there are two blocking assignment statements, the translation to non-blocking will differ based on whether there are dependencies. In the following subsections, the blocking assignments on the left are equivalent to the non-blocking assignments on the right given that `a`, `b`, `c`, `d` and `e` are unique variables, and `f()`, `g()` and `h()` are functions that involve only the argument shown and possibly variables other than `a`, `b`, `c`, `d` and `e`.

When there are no dependencies between two blocking assignments, there is a one-to-one translation to corresponding non-blocking assignments:

```
a=f(b);        a<=f(b);
c=g(d);        c<=g(d);
```

On the other hand, should the evaluation of the right-hand side of the second statement depend on the completion of the first blocking assignment, the right-hand side of the second non-blocking assignment must be re-written to reflect the composite computation (`g(f(b))`):

```
a=f(b);        a<=f(b);
c=g(a);        c<=g(f(b));
```

Sometimes, output dependencies of blocking assignments create dead code, which should be eliminated in the non-blocking equivalent:

```
a=f(b);        a<=g(d);
a=g(d);
```

If both kinds of dependencies are present, the two blocking assignments describe a single composite computation that modifies a single variable during a single clock cycle:

```
a=f(b);        a<=g(f(b));
a=g(a);
```

Assignments and One Decision

Both explicit and implicit styles allow decisions (`if` and `case` statements). The possible forms are too numerous to show here, but a simple example using `if` illustrates the kind of transformations needed. More complex `case` and `if` statements can be re-written in terms of composition of simple `if` statements.

Again, when there are no dependencies, the translation is one-to-one:

```
a=f(b);        a<=f(b);
if(h(e))       if(h(e))
  c=g(d);        c<=g(d);
```

Should the condition for the `if` depend on the first blocking assignment, it will need to be re-written:

```
a=f(b);        a<=f(b);
if(h(a))       if(h(f(b)))
  c=g(d);        c<=g(d);
```

as will the second assignment if it has a dependency on the first assignment:

```
a=f(b);        a<=f(b);
if(h(e))       if(h(e))
  c=g(a);        c<=g(f(b))
```

Both the condition and the second assignment might have a dependency:

```
a=f(b);        a<=f(b);
if(h(a))       if(h(f(b)))
  c=g(a);        c<=g(f(b))
```

The transformation becomes much more problematic when the second assignment (which is conditional) has a dependency on the first assignment (which is unconditional):

```
a=f(b);        if(h(e))
if(h(e))         a<=g(d);
  a=g(d);      else
                 a<=f(b);
```

There is no longer a one-to-one mapping, because only one non-blocking assignment actually happens per clock cycle, and so the condition must be tested before one or the other assignment occurs.

The code is further complicated when the condition has a dependency on the first assignment:

```
a=f(b);        if(h(f(b)))
if(h(a))         a<=g(d);
  a=g(d);      else
                 a<=f(b);
```

A similar scrambling of the original code occurs when there are both input and output dependencies between the first and second blocking assignments:

```
a=f(b);        if(h(e))
if(h(e))         a<=g(f(b));
  a=g(a);      else
                 a<=f(b);
```

and when the condition and both blocking assignments have dependencies:

```
a=f(b);          if(h(f(b)))
if(h(a))           a<=g(f(b));
  a=g(a);        else
                   a<=f(b);
```

To be useful, explicit-style code often needs complex nesting of such decisions. If the assignment in such code is blocking, the resulting hardware will often have little resemblance to the designer's intent. Also, the substitution of composite functions may create lengthy delay paths and make the resulting circuit larger than is optimal.

`while` loops

IEEE P1364.1 synthesis tools must be capable of handling both blocking and non-blocking assignments in explicit-style code. Although the substitutions required to obtain a realizable synchronous circuit are tedious, they result from simple recursive applications of the above rules until all the dependencies have been resolved. The IEEE P1364.1 requirement of a single clocking event in an explicit-style block limits the nature of dependencies to ones like those shown above. The lack of `while` and `forever` loops in IEEE P1364.1 guarantees that the dependencies that exist in explicit-style code are only those of textually later statements upon textually earlier statements.

In contrast, the dependencies that might exist in implicit-style code can be quite overwhelming. Let us consider an example of an innocent looking `while` loop with a blocking assignment statement:

```
module block2(a,b,waiting,sysclk);
 output [1:0] a;
 input [1:0] b;
 input waiting,sysclk;
 reg [1:0] a;
 wire [1:0] b;

 always
  begin
    while(waiting)
     @(posedge sysclk);
    a=b;
    while (a!=2)    //test current output
     begin
      @(posedge sysclk);
       a=gray(a);
     end
   @(posedge sysclk);
  end
...
endmodule
```

In the above, `gray()` is a function that computes the next value in a two-bit gray-code sequence (0, 1, 3, 2). The purpose of the module is to delay until `waiting` is 0, and then go through the gray-code sequence, starting at `b` and terminating at 2.

A thorough testbench that instantiates this needs to conduct four tests–one for each possible value of `b` (2, 3, 1 and 0). These tests correspond to calling the function zero to three times, respectively. In a correctly functioning system, the output will be the gray-code subsequence for one to four clock cycles, respectively.

The correct translation of the above blocking assignments into non-blocking assignments with a `while` loop requires the introduction of an extra state and a decision that were not present in the original blocking code:

```
always
 begin
   while(waiting)
    @(posedge sysclk);
   a<=b;
   if (b!=2)            //test input
    begin
     @(posedge sysclk); //extra state
      a<=gray(a);
      while(gray(a)!=2) //test next output
       begin
        @(posedge sysclk);
         a<=gray(a);
       end
    end
  @(posedge sysclk);
 end
```

When the input is 2, the machine needs to avoid calling the function at all, but since the non-blocking assignment will not have changed the output until later, the `if` must test the input, rather than the output.

The extra state handles the case when the input is 3. Since `gray(3)` is 2, the `while` loop is not entered in this case. This extra state also makes the first call to the function for the situation when the `while` loop is to be entered. Unlike the blocking code, the first call had to be unrolled from the `while` loop.

The transformed `while` loop is entered only if the function is to be called at least twice (the input is something besides 2 or 3). The condition that stops the loop needs to look ahead to the next gray value (rather than the current output used in the blocking example) because the non-blocking assignment will not have taken effect until after the decision has to be made.

Another Example

Consider a minor change in the blocking code (interchanging the clocking event and the function call):

```
always
 begin
   while(waiting)
    @(posedge sysclk);
   a=b;
   while (a!=2)
    begin
      a=gray(a);
     @(posedge sysclk);
    end
  @(posedge sysclk);
 end
```

This makes the value of `a` be `gray(b)` in the first cycle. For example, the machine outputs 1, 3, 2 when `b==0`. This seemingly minor variation causes a major difference in the equivalent non-blocking code:

```
always
 begin
   while(waiting)
    @(posedge sysclk);
   if (b!=2)
    begin
     a<=gray(b);
     if (gray(b)!=2)
      begin
       @(posedge sysclk); //1st extra
        a<=gray(a);
        while(gray(a)!=2)
         begin
          @(posedge sysclk);
           a<=gray(a);
         end
      end
     @(posedge sysclk);   //2nd extra
    end
   else
    a<=b;
  @(posedge sysclk);
 end
```

There are now two extra states and an `else`. The `else` is needed because two dependent blocking assignments happen in the first clock cycle, except when the input is 2. In that case, there is only one assignment (of the input to the output). As discussed earlier, equivalent non-blocking code requires an `if else`.

The first extra state is needed for the same reason as in the last example. Thus, the decision whether to call the function the first time is based on the input `b`, rather than `a` as was the situation in the blocking code. The second extra state, which is an empty state with just the clocking event, is needed because the blocking code has the clocking event at the bottom of the loop, but this particular non-blocking code needs the clocking event at the top of the loop. The extra state makes the cycle-by-cycle activity the same in the blocking and non-blocking code.

Bottom-Testing Loop

The reason the extra states are needed in the non-blocking versions above is because the current version of Verilog lacks a bottom-testing loop. For instance, the last non-blocking example can be simplified by using our proposed `repeat <statement> while (<condition>)` construct [2]:

```
always
 begin
   while(waiting)
    @(posedge sysclk);    //state 00
   if (b!=2)
    begin
     a<=gray(b);
     repeat
      begin
       @(posedge sysclk); //state 01
        if (a!=2)
         a<=gray(a);
      end
     while(a!=2);
    end
   else
    a<=b;
  @(posedge sysclk);      //state 10
 end
```

The advantages of the bottom-testing loop here are two-fold: it allows the transformed state machine to test `a==2` directly rather than `gray(a)==2`, and it has the same number of states as the original blocking version. The bottom-testing loop works here precisely because it ignores `a` during the clock cycle when the loop is entered.

Synthesis

A synthesis tool is not constrained to generate a state machine expressible in the behavioral constructs of the current Verilog standard; thus the tool is free to optimize to a bottom-testing-loop construct. For example, the last blocking code (with `a=gray(a)` before the clocking event) was synthesized using Synopsys FPGA Express, which supports blocking assignment in implicit-style code. The result of this synthesis produces a state machine equivalent to the bottom-testing code shown

above. The comments in that code show the state assignment chosen by FPGA Express.

It took us hours to decipher the netlist output from FPGA Express for this tiny three-state machine and prove it was isomorphic to what our theory predicts. Synopsys creates a present-state register (`multiple_wait_state`, which we will refer to as `s` for brevity) and combinational logic to compute the next state (`{N21,N22}`) for the controller and the load signal (`n102`) for the clock-enabled register, `a`. When synthesizing to a Xilinx FPGA, FPGA Express generates modules that compute the following expressions:

```
C52    = (s==2'b01)
syn151 = ( (s==2'b01)?(a[0]==0):(b[0]==0))
syn154 = syn151&
          ((s==2'b01)?(a[1]==1):(b[1]==1))
       = ((s==2'b01)?(a==2):(b==2))
N21    = syn154&((s==2'b01)|~waiting)
       = ((s==2'b01)?(a==2):((b==2)&~waiting))
N22    = ~syn154&((s==2'b01)|~waiting)
       = ((s==2'b01)?(a!=2):((b!=2)&~waiting))
n102   = C52&syn154|~C52&~waiting
       = ((s==2'b01)?(a!=2):~waiting)
```

Tabulating the truth table for this:

s	waiting	a==2	b==2	{N21,N22}	n102
00	0	x	0	01	1
00	0	x	1	10	1
00	1	x	x	00	0
01	x	0	x	01	1
01	x	1	x	10	0
10	0	x	0	01	1
10	0	x	1	10	1
10	1	x	x	00	0

we discover that the state machine Synopsys synthesized from the blocking code is identical to the non-blocking version using `repeat ... while`.

Simulation/Synthesis Mismatch

Both blocking and non-blocking assignments suffer from a semantic gap between synthesis and simulation when the assignments do not have an intra-assignment clocking event. In other words, the code in this paper has used `a<=b` rather than `a<= @(posedge sysclk)b`. In implicit-style code that uses only the non-blocking assignment there is a solution to overcome this mismatch[1]: define macros `` `ENS `` and `` `CLK `` differently for simulation and synthesis. These are used as:

```
@(posedge sysclk)`ENS;
  a<=`CLK gray(a);
```

For simulation, they substitute the appropriate time control. For synthesis, they are simply empty macros.

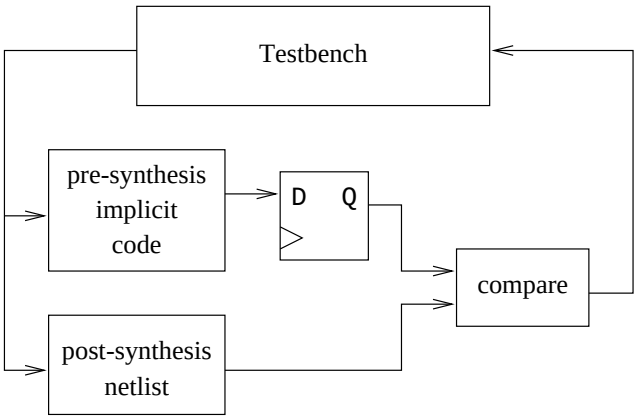

Figure 1: Testbench configuration.

Although using these macros with non-blocking assignment only is our recommendation [2] for "safe" implicit-style simulation and synthesis, we will not follow these recommendations here because they are inapplicable to blocking assignment.

Instead, we will modify the testbench. For blocking assignment in implicit-style code, there is no way to avoid such simulation mismatches. If we do not use the macros, the non-blocking code creates identical simulation mismatches. To compensate for this, we added an extra clock cycle of delay (outside of the implicit-style code) from the pre-synthesis simulation before comparing it to post-simulation results, as shown in Figure 1.

IEEE P1364.1 is designed to avoid this semantic gap–it arises only in implicit-style code. IEEE P1364.1 is stateless–any state machines must be declared using explicit `reg`s, which provide the necessary one-clock-cycle delay so that simulation matches synthesis.

Conclusions

We have given some examples of the transformations that convert blocking assignments into non-blocking assignments. Some of these transformations apply to both explicit- and implicit-style Verilog, but others apply only to implicit-style code. The goal of the transformations is to eliminate dependencies often found in blocking assignment so that we can use the parallel-processing nature of the non-blocking assignment to achieve the same cycle-by-cycle effect.

We synthesized using Synopsys FPGA Express, and found that it had transformed the blocking assignments into non-blocking assignments in precisely the way our theory predicted. We overcame the one-clock-cycle mismatch between implicit-style synthesis and simulation by adapting our testbench. Including a bottom-testing-

loop construct in Verilog would help conceptualize the fancy transformations a sophisticated tool like FPGA Express performs.

In general, the simulation/synthesis mismatch is solvable by using macros only for non-blocking assignment. We feel that non-blocking assignment is the natural way to express the parallel-processing power of hardware. Blocking assignment, while appropriate for software and mathematical functions (combinational logic), is clumsy and cumbersome for realistic hardware (sequential logic). The involved and counterintuitive transformations shown here from blocking to non-blocking assignment illustrate that many designers using blocking assignment may not understand how their synthesized design actually works. Implicit style with non-blocking assignment is the right level of abstraction. It is not as tied to implementation details as explicit style, but, as our examples have shown, non-blocking assignment emphasizes what the hardware does in parallel, unlike blocking assignment. (If one wishes to use blocking assignment, one might as well use a synthesis tool that translates directly from C and that automatically schedules the clocking events. A designer using blocking assignment in Verilog probably has no better idea of the cycle-by-cycle activity of the synthesized netlist than one using clockless C code.)

These results justify our decision to implement only non-blocking assignment in our VITO preprocessor. They also give us great appreciation for the skill that went into designing tools, like FPGA Express, that support blocking assignment in implicit-style Verilog.

References

[1] M. G. Arnold, *Verilog Digital Computer Design: Algorithms to Hardware,* Upper Saddle River, NJ: PTR Prentice Hall, 1999, pp. 96-109. Typos and example files are available at `strawberry.uwyo.edu/~verilog`.

[2] M. G. Arnold, N. J. Sample and J. D. Shuler, "Guidelines for Safe Simulation and Synthesis of Implicit Style Verilog," *Proceedings of the Seventh International Verilog HDL Conference,* Santa Clara, CA: Mar. 16–19, 1998.

[3] M. G. Arnold and J. D. Shuler, "A Synthesis Preprocessor that Converts Implicit Style Verilog into a One-hot Design," *Proceedings of the Sixth International Verilog HDL Conference,* Santa Clara, CA: Mar. 31–Apr. 3 1997, pp. 38-45. Source code for the latest version of VITO is available at `www.verilog.vito.com`. Mirrors of this site are available at: `strawberry.uwyo.edu/~verilog` and `www.cs.brockport.edu/~jshuler/vito`.

[4] C. E. Cummings, "Verilog Nonblocking Assignment Demystified," *Proceedings of the Seventh International Verilog HDL Conference,* Santa Clara, CA: Mar. 16–19, 1998, pp. 67–69.

[5] *DRAFT Standard Register Transfer Level Subset Based on the Verilog Hardware Description Language*, P1364.1, New York: IEEE, June 3, 1998. Available at `www.eda.org/vlog-synth`.

[6] J. M. Lee, *Verilog Quickstart,* Norwell, MA: Kluwer, 1997.

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

[8] N. Wirth, "Hardware Compilation: Translating Programs into Circuits," *Computer,* IEEE Computer Society, June, 1998, pp. 25–31.