Cliff-notes - 20060924

One of the big problems with this section is that lack of complete examples that show the clocking block used in a recommended context. As written, readers are left to imagine the proper use of clocking blocks in a reasonable context. Some the proposed changes show the clocking block used in a more complete context.

If we keep the very last paragraph of clause 15, it needs to be described much sooner in the clause:

The **`clocking`** block outputs driving a net (i.e., through different ports) cause the net to be driven to its resolved signal value. When a **`clocking`** block output corresponds to a wire, a driver for that wire is created that is updated as if by a continuous assignment from a register inside the **`clocking`** block that is updated as a nonblocking assignment.

To me, it seems like this paragraph actually describes two different conditions. I believe the first part refers to output or inout ports of modules, programs and interfaces and when a module, program or interface port drives the same net as another module, program or interface port, that there is resolution (this would not be surprising). The reason I believe this was the intent is because of the "(i.e., through different ports)" statement, as opposed to referring to a net within the same scope where the clocking block is declared (such as a net within a module or program).

An interpretation that a signal declared as an output in a clocking block that drives a wire inside the same program, module or interface as another driver in the same program, module or interface, and that there should be resolution from this circumstance would be surprising and non-intuitive the way this paragraph is written.

In contrast, the second half of the paragraph seems to describe making procedural assignments to a net INSIDE of a program, module or interface and that this is treated as a continuous assignment that schedules the result into the NBA region. I did not even remember reading this until I read Chris Spear's new SystemVerilog Verification book and he pointed this out. I thought he was wrong until I found the above, very last sentence in clause 15. This should be described much sooner if this is indeed the intent. This behavior is unlike any other behavior in Verilog (procedural assignment to a net that creates another driver on that net).

Until I read this last sentence, I was trying to figure out how a program could drive a bi-directional inout port. Verilog users are used to inout ports NOT being assigned by a procedural block (inout ports must be nets and the LHS of procedural assignments must be variables).

I think there are some real problems with clocking block and program block restrictions and event scheduling. These are my issues:

(1) I believe program block ports that connect to wires in a module must be considered a design port and as such, currently requires a nonblocking assignment (so I almost completely disagree with Mentor's proposed section 16.2.1 for program blocks (end of this proposal)).

(2) It is not clear, except by example in the clocking block section, that programs can make continuous assignments to design ports and that these assignments are made with blocking assignments but that the events are scheduled into the NBA region. Not well explained and non-intuitive, so I could be completely wrong on this point.

(3) Until you get to the last paragraph of clause 15, the reader does not know that they can make procedural assignments to nets and that they behave like continuous assignment drivers to design signals that are again scheduled into the NBA region. Completely non-intuitive and therefore I might again be wrong.

(4) If one program port communicates with another program port, the communication happens across a module wire and so the event should be scheduled into the NBA region(??).This seems like a simple explanation that isolates testbench activity from design activity, a primary goal for the new testbench features. I believe this could be considered a clarification (and a simplification). If a program wants to communicate with another program in the Reactive region, perhaps this requires cross-program hierarchical assignments(??)

15. Clocking blocks

15.1 Introduction

NOTE—In Verilog, the communication between blocks is specified using module ports. SystemVerilog adds the interface, a key construct that encapsulates the communication between blocks, thereby enabling users to easily change the level of abstraction at which the intermodule communication is to be modeled.

An interface can specify the signals or nets through which a testbench communicates with a device under test (DUT). However, an interface does not explicitly specify any timing disciplines, synchronization requirements, or clocking paradigms.

SystemVerilog adds the **`clocking`** block that identifies clock signals and captures the timing and synchronization requirements of the blocks being modeled. A **`clocking`** block assembles signals that are synchronous to a particular clock and makes their timing explicit. The **`clocking`** block is a key element in a cycle-based methodology, which enables users to write testbenches at a higher level of abstraction. Rather than focusing on signals and transitions in time, the test can be defined in terms of cycles and transactions. Depending on the environment, a testbench can contain one or more **`clocking`** blocks, each containing its own clock plus an arbitrary number of signals.

<update>
The **`clocking`** block separates the timing and synchronization details from the structural, functional, and procedural elements of a testbench. Thus, the timing for sampling and driving **`clocking`** block signals ~~is~~ can be implicit and relative to the **`clocking`** block's clock. This enables a set of key operations to be written very succinctly, without explicitly using clocks or specifying timing. These operations are as follows:

— Synchronous events
— Input sampling
— Synchronous drives

15.2 Clocking block declaration

The syntax for the **`clocking`** block is as follows:

Syntax 15-1—Clocking block syntax (excerpt from Annex A)

The *delay_control* must be either a time literal or a constant expression that evaluates to a positive integer value.

The *clocking_identifier* specifies the name of the **`clocking`** block being declared.

The *signal_identfier* identifies a signal in the scope enclosing the **`clocking`** block declaration and declares the name of a signal in the **`clocking`** block. Unless a *hierarchical_expression* is used, both the signal and the *clocking_item* names shall be the same.

<update>
The *clocking_event* designates a particular event to act as the clock for the **`clocking`** block. Typically, this expression is either the **`posedge`** or **`negedge`** of a clocking signal. The timing of all the other signals specified in a given **`clocking`** block is governed by the clocking event. All **`input`** or **`inout`** signals specified in the **`clocking`** block are sampled when the corresponding clock event occurs. Likewise, all **`output`** or **`inout`** signals in the **`clocking`** block are driven when the corresponding clock event occurs. See 15.12 and 15.14 for details on the precise timing semantics of sampling and driving clocking signals. Bidirectional signals (**`inout`**) are sampled as well as driven. An **`output`** signal cannot be sampled ~~read~~, and an **`input`** signal cannot be driven.

I believe this means that outputs cannot have the cb-prefix if the output is on the RHS of an equation, and inputs cannot have the cb-prefix on the LHS of an equation (in general, you shouldn't make assignments to inputs anyway)

The *clocking_skew* determines how many time units away from the clock event a signal is to be sampled or driven. Input skews are implicitly negative, that is, they always refer to a time before the clock, whereas output skews always refer to a time after the clock (see 15.3). When the clocking event specifies a simple edge, instead of a number, the skew can be specified as the specific edge of the signal. A single skew can be specified for the entire block by using a **`default`** clocking item.

```
clocking ck1 @(posedge clk);
  default input #1step output negedge; // legal
  // outputs driven on the negedge clk
  input ... ;
  output ... ;
endclocking

clocking ck2 @(clk); // no edge specified!
  default input #1step output negedge; // legal
  input ... ;
  output ... ;
endclocking
```

The *hierarchical_identifier* specifies that, instead of a local port, the signal to be associated with the **`clocking`** block is specified by its hierarchical name (cross-module reference).

Example:

```
clocking bus @(posedge clock1);
  default input #10ns output #2ns;
  input data, ready, enable = top.mem1.enable;
  output negedge ack;
  input #1step addr;
endclocking
```

In the above example, the first line declares a **`clocking`** block called `bus` that is to be clocked on the positive edge of the signal `clock1`. The second line specifies that by default all signals in the **`clocking`** block shall use a `10ns` input skew and a `2ns` output skew. The next line adds three input signals to the **`clocking`** block: `data`, `ready`, and `enable`; the last signal refers to the hierarchical signal `top.mem1.enable`. The fourth line adds the signal `ack` to the **`clocking`** block and overrides the default output skew so that `ack` is driven on the negative edge of the clock. The last line adds the signal `addr` and overrides the default input skew so that `addr` is sampled one step before the positive edge of the clock.

Unless otherwise specified, the default **`input`** skew is `1step` and the default **`output`** skew is **`0`**. A step is a special time unit whose value is defined in 19.10. A `1step` input skew allows input signals to sample their steady-state values in the time step immediately before the clock event (i.e., in the preceding Postponed region). Unlike other time units, which represent physical units, a step cannot be used to set or modify either the precision or the time unit.

15.3 Input and output skews

Input (or inout) signals are sampled at the designated clock event. If an input skew is specified, then the signal is sampled at *skew* time units *before* the clock event. Similarly, output (or inout) signals are driven *skew* simulation time units *after* the corresponding clock event. Figure 15-1 shows the basic sample and drive timing for a positive edge clock.

Figure 15-1—Sample and drive times including skew with respect to the positive edge of the clock

A skew must be a constant expression and can be specified as a parameter. If the skew does not specify a time unit, the current time unit is used. If a number is used, the skew is interpreted using the timescale of the current scope.

```
clocking dram @(clk);
  input #1ps address;
  input #5 output #6 data;
endclocking
```

An input skew of `1step` indicates that the signal is to be sampled at the end of the previous time step. In other words, the value sampled is always the signal's last value immediately before the corresponding clock edge.

NOTE—A **`clocking`** block does not eliminate potential races when an event control outside of a program block is sensitive to the same clock as the **`clocking`** block and a statement after the event control attempts to read a member of the **`clocking`** block. The race is between reading the old sampled value and the new sampled value.

Inputs with explicit #0 skew shall be sampled at the same time as their corresponding clocking event, but to avoid races, they are sampled in the Observed region. Likewise, **`clocking`** block outputs with no skew (or explicit #0 skew) shall be driven at the same time as their specified clocking event, as nonblocking assignments (in the NBA region).

Skews are declarative constructs; thus, they are semantically very different from the syntactically similar procedural delay statement. In particular, an explicit #0 skew does not suspend any process, nor does it execute or sample values in the Inactive region.

15.4 Hierarchical expressions

Any signal in a **`clocking`** block can be associated with an arbitrary hierarchical expression. As described in 15.2, a hierarchical expression is introduced by appending an equal sign (`=`) followed by the hierarchical expression:

```
clocking cd1 @(posedge phi1);
  input #1step state = top.cpu.state;
endclocking
```

However, hierarchical expressions are not limited to simple names or signals in other scopes. They can be used to declare slices and concatenations (or combinations thereof) of signals in other scopes or in the current scope.

```
clocking mem @(clock);
  input instruction = { opcode, regA, regB[3:1] };
endclocking
```

In a **`clocking`** block, any expression assigned to a signal in its declaration shall be an expression that would be legal in a port connection to a port of any of the directions specified in the declaration. For example, it would be illegal to assign an **`inout`** signal an expression in its declaration that would be illegal in a port connection to an **`inout`** port.

15.5 Clocking block and non-clocking block timing of the same signals

Any signal declared as a module / program / interface signal or port that is also declared in a clocking block can be assigned with both a non-clocking normal Verilog delay and clocking block cycle-based delay. The absence of the clocking block identifier indicates normal timing while inclusion of the clocking block identifier indicates clocking block timing.

Example:

```
`timescale 1ns / 1ns
module clk_blk1a;
```

```
  logic d, clk;

  initial begin
    clk <= '0;
    forever #5 clk = ~clk;
  end

  blk1a t1 (.*);
endmodule

program blk1a (output logic d,
               input        clk);

  // program clocking block
  clocking cb1 @(posedge clk);
    output #2 d;
  endclocking

  initial begin
               d <= '1; //  0ns: clk=0  d=1
    @(cb1)     d <= '0; //  5ns: clk=1  d=0
                        // 10ns: clk=0
    @(cb1)     d <= '1; // 15ns: clk=1  d=1
                        // 20ns: clk=0
    @(cb1) cb1.d <= '0; // 25ns: clk=1
                        // 27ns:        d=0
                        // 30ns: clk=0
    @(cb1) cb1.d <= '1; // 35ns: clk=1
                        // 37ns:        d=1
                        // 40ns: clk=0
    @(cb1) $finish;     // 45ns: clk=1
  end
endprogram
```

In the preceding example, all occurrences of **`@(cb1)`** are equivalent to **`@(posedge clk)`**. All assignments to the variable **`d`** happen immediately after the preceding event or time delay, just like any other assignment in Verilog. The assignments in this example proceed as follows:

— The first assignment to the **`d`** variable happens at time **`0`** because there are no specified delays, events or clocking block events and the **`d`**-variable is not prefixed with a clocking block specifier.

— The second assignment to the **`d`** variable happens after **`@(cb1)`** (posedge clk at time 5ns), again because the **`d`**-variable is not prefixed with a clocking block specifier.

— The third assignment to the d variable happens after **`@(cb1)`** (posedge clk at time 15ns), again because the **`d`**-variable is not prefixed with a clocking block specifier.

— The fourth assignment to the **`d`** variable happens after **`@(cb1)`** (posedge clk at time 25ns) plus an additional delay of 2ns (time 27ns) because the **`d`**-variable is prefixed with a clocking block specifier as noted by the assignment **`cb1.d <= '0;`**. In this case, the assignment is equivalent to: **`@(posedge clk) d <= #2 '0;`**

— The last assignment to the d variable happens after **`@(cb1)`** (posedge clk at time 35ns) plus an additional delay of 2ns (time 37ns) because the **`d`**-variable is again prefixed with a clocking block specifier as noted by the assignment **`cb1.d <= '1;`**. **`;`**. In this case, the assignment is equivalent to: **`@(posedge clk) d <= #2 '1;`**

Note: Once a clocking block is defined and clocking block timing is used, mixing of clocking block-timed assignments with non-clocking block assignments to the same variables is generally not recommended.

Making time-0 non-clocking block assignments to variables to initialize their values might be a reasonable exception to this recommendation.

(increment all section numbers if section 15.5 is added)

15.5 Signals in multiple clocking blocks

The same signals—clock, inputs, inouts, or outputs—can appear in more than one **`clocking`** block. When **`clocking`** blocks use the same clock (or clocking expression), they shall share the same synchronization event, in the same manner as several latches can be controlled by the same clock. Input semantics is described in 15.12, and output semantics is described in 15.14.

15.6 Clocking block scope and lifetime

A **`clocking`** block is both a declaration and an instance of that declaration. A separate instantiation step is not necessary. Instead, one copy is created for each instance of the block containing the declaration (like an **`always`** block). Once declared, the clocking signals are available via the **`clocking`** block name and the dot (**`.`**) operator:

```
dom.sig // signal sig in clocking dom
```

Multiple **`clocking`** blocks cannot be nested. They cannot be declared inside functions, tasks, or packages or outside all declarations in a compilation unit. A **`clocking`** block can only be declared inside a module, interface, or program (see Clause 16).

A **`clocking`** block has static lifetime and scope local to its enclosing module, interface, or program.

15.7 Multiple clocking blocks example

In this example, a simple test program includes two **`clocking`** blocks. The program construct used in this example is discussed in Clause 16.

```
program test( input phi1, input [15:0] data, output logic write,
              input phi2, inout [8:1] cmd, input enable );

  reg [8:1] cmd_reg;

  clocking cd1 @(posedge phi1);
    input data;
    output write;
    input state = top.cpu.state;
  endclocking

  clocking cd2 @(posedge phi2);
    input #2 output #4ps cmd;
    input enable;
  endclocking

  initial begin
    // program begins here
  ...
    // user can access cd1.data , cd2.cmd , etc…
  end
  assign cmd = enable ? cmd_reg: 'x;
endprogram
```

The test program can be instantiated and connected to a DUT (`cpu` and `mem`).

```
module top;
  logic phi1, phi2;
  wire [8:1] cmd; // cannot be logic (two bidirectional drivers)
  logic [15:0] data;

  test main( phi1, data, write, phi2, cmd, enable );
  cpu cpu1( phi1, data, write );
  mem mem1( phi2, cmd, enable );
endmodule
```

15.8 Interfaces and clocking blocks

A **`clocking`** encapsulates a set of signals that share a common clock; therefore, specifying a **`clocking`** block using a SystemVerilog **`interface`** can significantly reduce the amount of code needed to connect the testbench. Furthermore, because the signal directions in the **`clocking`** block within the testbench are with respect to the testbench and not the design under test, a **`modport`** declaration can appropriately describe either direction. A testbench program can be contained within a program, and its ports can be interfaces that correspond to the signals declared in each **`clocking`** block. The interface's wires shall have the same direction as specified in the **`clocking`** block when viewed from the testbench side (i.e., **`modport`** `test`) and reversed when viewed from the DUT (i.e., **`modport`** `dut`).

For example, the previous example could be rewritten using interfaces as follows:

```
interface bus_A (input clk);
  logic [15:0] data;
  logic write;
  modport test (input data, output write);
  modport dut (output data, input write);
endinterface

interface bus_B (input clk);
  logic [8:1] cmd;
  logic enable;
  modport test (input enable);
  modport dut (output enable);
endinterface

program test( bus_A.test a, bus_B.test b );
  clocking cd1 @(posedge a.clk);
    input a.data;
    output a.write;
    inout state = top.cpu.state;
  endclocking

  clocking cd2 @(posedge b.clk);
    input #2 output #4ps b.cmd;
    input b.enable;
  endclocking

  initial begin
    // program begins here
    ...
    // user can access cd1.a.data , cd2.b.cmd , etc…
  end
endprogram
```

The test module can be instantiated and connected as before:

```
module top;
  logic phi1, phi2;

  bus_A a(phi1);
  bus_B b(phi2);
  test main( a, b );
  cpu cpu1( a );
  mem mem1( b );
endmodule
```

Alternatively, in the program test above, the **clocking** block can be written using both interfaces and hierarchical expressions as follows:

```
clocking cd1 @(posedge a.clk);
  input data = a.data;
  output write = a.write;
  inout state = top.cpu.state;
endclocking

clocking cd2 @(posedge b.clk);
  input #2 output #4ps cmd = b.cmd;
  input enable = b.enable;
endclocking
```

This would allow using the shorter names (`cd1.data`, `cd2.cmd`, …) instead of the longer interface syntax (`cd1.a.data`, `cd2.b.cmd`, …).

15.9 Clocking block events

The clocking event of a **clocking** block is available directly by using the **clocking** block name, regardless of the actual clocking event used to declare the **clocking** block.

For example.

```
clocking dram @(posedge phi1);
  inout data;
  output negedge #1 address;
endclocking
```

The clocking event of the `dram` **clocking** block can be used to wait for that particular event:

```
@( dram );
```

The above statement is equivalent to `@(`**posedge** `phi1).`

15.10 Cycle delay:

WAS: The `##` operator can be used to delay execution by a specified number of clocking events or clock cycles.

PROPOSED: The `##` operator can be used to delay execution by a specified number of default clocking events or default clock cycles (see 15.11 for the definition of default clocking).

REASON: *The reader needs to be notified immediately that ## delays cannot be generally used unless a default clocking block has been defined. The current LRM does not note this important restriction until well after the syntax table.*

The syntax for the cycle delay statement is as follows:

Syntax 15-2—Cycle delay syntax (excerpt from Annex A)

The *expression* can be any SystemVerilog expression that evaluates to a positive integer value.

What constitutes a cycle is determined by the default clocking in effect (see 15.11). If no default clocking has been specified for the current module, interface, or program, then the compiler shall issue an error.

Example:

Modified example and explanatory text PROPOSAL:

```
...
// default clocking block
default clocking cb1 @(posedge clk);
endclocking

initial begin
  ...
  ## 5; // wait 5 cycles (clocking events) using the default clocking
  ...
  ## (j + 1); // wait j+1 cycles (clocking events) using the default
clocking
  ...
end
```

In the above example, `##5;` is equivalent to `repeat(5) @(posedge clk);` and `##(j+1);` is equivalent to `repeat (j+1) @(posedge clk);`

SVDB #890 PROPOSAL: If a ## cycle delay operator is executed at a simulation time that does not correspond to a default clocking event (perhaps due to the use of a # delay control or an asynchronous @ event control), ~~the processing of the cycle delay is postponed until the time of the next default clocking event. Thus a ##1 cycle delay shall always be guaranteed to wait at least one full clock cycle.~~

Modified PROPOSAL: If a ## cycle delay operator is executed at a simulation time that does not correspond to a default clocking event (perhaps due to the use of a # delay control or an asynchronous @ event control), then a ##1 delay will delay until the next active default clock edge, which will be less than one full default clock cycle.

In fact, given the following default clocking block, the two initial block delay statements are equivalent:

```
default clocking cb1 @( my_clocking_event );
endclocking

initial begin
  ##(n);
  repeat (n) @( my_clocking_event );
end
```

Below is a full example with top-level module that includes a clock oscillator, an instantiated program that includes a default clocking block, and initial block assignments using default clocking delays and one asynchronous delay. At the end of the program is a second copy of the same program initial block with annotated delays for all of the events that would occur during simulation of this example.

```
`timescale 1ns / 1ns
module async_delay;
  logic d, clk;

  initial begin
    clk <= '0;
    forever #5 clk = ~clk;
  end
  tst t1 (.*);
endmodule

program tst (output logic d,
             input        clk);
  int j;     // program variable

  // default clocking block
  default clocking cb1 @(posedge clk);
  endclocking

  initial begin
             j  = 2;
             d <= '1;
    ##4;
             d <= '0;
    ##(j+1);          // delay is ##3
    ##1      d <= '1;
    #3ns     d <= '0; // asynchronous 3ns delay
    ##1      d <= '1;
    ##1      $finish;
  end
endprogram
```

The scheduled timing of the events from the preceding program initial block is shown below

```
  ...
  initial begin
             j  = 2;
             d <= '1; //  0ns: clk=0  d=1
    ##4;              //  5ns: clk=1 (1st clk event)
                      // 10ns: clk=0
                      // 15ns: clk=1 (2nd clk event)
                      // 20ns: clk=0
                      // 25ns: clk=1 (3rd clk event)
                      // 30ns: clk=0
                      // 35ns: clk=1 (4th clk event)
             d <= '0; // 35ns:        d=0
                      // 40ns: clk=0
    ##(j+1); // ##3   // 45ns: clk=1 (1st clk event)
                      // 50ns: clk=0
                      // 55ns: clk=1 (2nd clk event)
                      // 60ns: clk=0
                      // 65ns: clk=1 (3rd clk event)
                      // 70ns: clk=0
    ##1      d <= '1; // 75ns: clk=1  d=1
    #3ns     d <= '0; // 78ns: clk=1  d=0 (async 3ns)
                      // 80ns: clk=0
    ##1      d <= '1; // 85ns: clk=1  d=1
                      // 90ns: clk=0
    ##1      $finish; // 95ns: clk=1  $finish
  end
  ...
```

In summary, ## delays are only valid when used with default clocking blocks and they provide a useful shorthand to specify repetitions of default clocking event delays.

15.11 Default clocking

One **`clocking`** can be specified as the default for all cycle delay operations within a given module, interface, or program.

The syntax for the default cycle specification statement is as follows:

Syntax 15-3—Default clocking syntax (excerpt from Annex A)

The *clocking_identifier* must be the name of a **`clocking`** block.

Only one default clocking can be specified in a program, module, or interface. Specifying a default clocking more than once in the same program or module shall result in a compiler error.

A default clocking is valid only within the scope containing the default clocking specification. This scope includes the module, interface, or program that contains the declaration as well as any nested modules or interfaces. It does not include instantiated modules or interfaces.

Example 1: Declaring a clocking as the default:

```
program test( input bit clk, input reg [15:0] data );
  default clocking bus @(posedge clk);
    inout data;
  endclocking

  initial begin
    ## 5;
    if ( bus.data == 10 )
      ## 1;
    else
    ...
  end
endprogram
```

Example 2: Assigning an existing clocking to be the default:

```
module processor ...
  clocking busA @(posedge clk1); ... endclocking
  clocking busB @(negedge clk2); ... endclocking
  module cpu( interface y );
    default clocking busA ;
    initial begin
      ## 5; // use busA => (posedge clk1)
      ...
    end
  endmodule
endmodule
```

Example 3: Making clocking and non-clocking assignments in a program with a default clocking block.

```
`timescale 1ns / 1ns
module clk_blk1a;
  logic d, clk;

  initial begin
    clk <= '0;
    forever #5 clk = ~clk;
  end
  blk1a t1 (.*);
endmodule

program blk1a (output logic d,
              input        clk);

  // program clocking block
  default clocking cb1 @(posedge clk);
    output #2 d;
  endclocking

  initial begin
               d <= '1; //  0ns: clk=0  d=1
    ##1        d <= '0; //  5ns: clk=1  d=0
                        // 10ns: clk=0
    @(cb1)     d <= '1; // 15ns: clk=1  d=1
                        // 20ns: clk=0
    ##1    cb1.d <= '0; // 25ns: clk=1
                        // 27ns:        d=0
                        // 30ns: clk=0
    @(cb1) cb1.d <= '1; // 35ns: clk=1
                        // 37ns:        d=1
                        // 40ns: clk=0
    ##1 $finish;        // 45ns: clk=1
  end
endprogram
```

In the example above, all occurrences of **`##1`** are equivalent to **`@(posedge clk)`**. Similarly, all occurrences of **`@(cb1)`** are equivalent to **`@(posedge clk)`**. All assignments to the variable **`d`** happen immediately after the preceding event or time delay, just like any other assignment in Verilog. The assignments in this example proceed as follows:

— The first assignment to the **`d`** variable happens at time **`0`** because there are no specified delays, events or clocking block events and the **`d`**-variable is not prefixed with a clocking block specifier.

— The second assignment to the **`d`** variable happens after **`##1`** (posedge clk at time 5ns), again because the **`d`**-variable is not prefixed with a clocking block specifier.

— The third assignment to the d variable happens after **`@(cb1)`** (posedge clk at time 15ns), again because the **`d`**-variable is not prefixed with a clocking block specifier.

— The fourth assignment to the **`d`** variable happens after **`##1`** (posedge clk at time 25ns) plus an additional delay of 2ns (time 27ns) because the **`d`**-variable is prefixed with a clocking block specifier as noted by the assignment **`cb1.d <= '0;`**.

— The last assignment to the d variable happens after **`@(cb1)`** (posedge clk at time 35ns) plus an additional delay of 2ns (time 37ns) because the **`d`**-variable is again prefixed with a clocking block specifier as noted by the assignment **`cb1.d <= '1;`**.

Note: Once a clocking block is defined and clocking block timing is used, mixing of clocking block-timed assignments with non-clocking block assignments to the same variables is generally not recommended. Making time-0 non-clocking block assignments to variables to initialize their values might be a reasonable exception to this recommendation.

15.12 Input sampling

All **`clocking`** block inputs (input or inout) are sampled at the corresponding clocking event. If the input skew is not an explicit #0, then the value sampled corresponds to the signal value at the Postponed region of the time step skew time units prior to the clocking event (see Figure 15-1 in 15.3). ~~If the input skew is an explicit #0, then the value sampled corresponds to the signal value in the Observed region.~~

If the input skew is an explicit #0, several additional considerations shall govern input sampling. First, the value sampled corresponds to the signal value in the Observed region. Next, if the clocking event occurs due to activity on a design object (net or variable), the sampled value shall be updated and available for reading before any program code starts execution in the reactive region. Finally, if the clocking event occurs due to activity on a program object, there is a race condition between the update of the clocking block's input value and the execution of program code that reads that value.

??? (I do not understand the point of this paragraph. Each of these additional considerations needs to be accompanied by an example before we can reasonably vote to approve this change)

When a signal appears in an expression, it is replaced by the signal's sampled value, that is, the value that was sampled at the last sampling point.

When the same signal is an input to multiple **`clocking`** blocks, the semantics is straightforward; each **`clocking`** block samples the corresponding signal with its own clocking event.

15.13 Synchronous events

Explicit synchronization is done via the event control operator, `@`, which allows a process to wait for a particular signal value change or a clocking event (see 15.9).

The syntax for the synchronization operator is given in 10.10.

The expression used with the event control can denote **`clocking`** block input (**`input`** or **`inout`**) or a slice thereof. Slices can include dynamic indices, which are evaluated once, when the `@` expression executes.

These are some examples of synchronization statements:

— Wait for the next change of signal `ack_1` of **`clocking`** block `ram_bus`
```
@(ram_bus.ack_l);
```

— Wait for the next clocking event in **`clocking`** block `ram_bus`
```
@(ram_bus);
```

— Wait for the positive edge of the signal `ram_bus.enable`
```
@(posedge ram_bus.enable);
```

— Wait for the falling edge of the specified 1-bit slice `dom.sign[a]`
```
@(negedge dom.sign[a]);
```
NOTE—The index `a` is evaluated at run time.

— Wait for either the next positive edge of `dom.sig1` or the next change of `dom.sig2`, whichever happens first
```
@(posedge dom.sig1 or dom.sig2);
```

— Wait for the either the negative edge of `dom.sig1` or the positive edge of `dom.sig2`, whichever happens first
```
@(negedge dom.sig1 or posedge dom.sig2);
```

The values used by the synchronization event control are the synchronous values, that is, the values sampled at the corresponding clocking event.

15.14 Synchronous drives

The **clocking** block outputs (**output** or **inout**) are used to drive values onto their corresponding signals, but at a specified time. In other words, the corresponding signal changes value at the indicated clocking event as modified by the output skew.

Clocking block outputs are scheduled to propagate back into the design after the reactive and re-inactive regions of a given time unit have completed their iterations and contain no more events. (See Figure 9-1) After these regions have been processed, all possible synchronous drives will have executed. For zero skew clocking block outputs with no cycle delay, the new values will be scheduled in the NBA region of the current time unit. For clocking block outputs with non-zero skew or non-zero cycle delay, the corresponding signal shall be scheduled to change value in the NBA region of a future time unit.

I believe the above wording is a deviation from the LRM description and intended behavior, but I like some of the ideas expressed in the text and I believe it would solve other problems with clocking blocks.

Cliff's interpretation of intended behavior:
NBA assignments are required in a program block to force stimulus events to be scheduled into the NBA region after all RTL NBA events have been activated and executed in the same timestep. Placing the program stimulus events into the NBA region ensures that they cannot trigger additional RTL events until after the program block has completed all reactive and re-inactive events, at which time all events scheduled in the NBA region will again be activated as a group and possibly force additional RTL events to trigger. The fact that only program variables can be assigned using blocking assignments means that program variables are executed immediately in the Reactive Region but since they do not touch any RTL signals, they do not cause any RTL events and hence we avoid race conditions between the program execution and the RTL execution. In essence, I believe we have re-used the NBA region to store up stimulus events that will be activated as a block and not interleave with RTL signals that might re-trigger due to updates to RTL signals driven from the program block. To me, this explains why it was so important that program signals that drive RTL signals be assigned using nonblocking assignments.

On the other hand, if blocking assignments to RTL signals were scheduled into the Active Region, and if the Active Region was not activated until the current pass through the Reactive/Re-inactive regions, I believe we could reuse the active region for stimulus scheduling and not suffer any race conditions with RTL signals (not 100% sure of this).

What we want to avoid is blocking assignments to RTL signals that could immediately re-trigger RTL events in the Active Region that would inter-mingle with Reactive and Re-inactive events, yielding potential Design-testbench race conditions.

That having been said, we can't make NBA assignments from a continuous assignment which makes it somewhat difficult to deal with bi-directional signals.

The syntax to specify a synchronous drive is similar to an assignment:

Syntax 15-4—Synchronous drive syntax (excerpt from Annex A)

The *clockvar_expression* is either a bit-select, slice, or the entire **clocking** block output whose corresponding signal is to be driven (concatenation is not allowed):

```
dom.sig // entire clockvar

dom.sig[2] // bit-select

dom.sig[8:2] // slice
```

The *expression* (in the *clocking_drive* production) can be any valid expression that is assignment compatible with the type of the corresponding signal.

The *event_count* refers to the *expression* after the ## in the *cycle_delay* production and is an integral *expression* that optionally specifies the number of clocking events (i.e., cycles) that must pass before the statement executes. Specifying a nonzero `event_count` blocks the current process until the specified number of clocking events has elapsed; otherwise, the statement executes at the current time. The `event_count` uses syntax similar to the cycle delay operator (see 15.10); however, the synchronous drive uses the **`clocking`** block of the signal being driven and not the default clocking.

The second form of the synchronous drive uses the intra-assignment syntax. An intra-assignment `event_count` specification also delays execution of the assignment. In this case, the process does not block, and the right-hand expression is evaluated when the statement executes.

Examples:

Add the following default clocking block to this example to make the comments correct. If the default clocking block included the statements: **`default output 2ns; output data;`** ***then the comments would either be wrong or incomplete.***

```
default clocking bus @(bus_cycle);
endclocking

bus.data[3:0] <= 4'h5; // drive data in the NBA region of the current cycle

##1 bus.data <= 8'hz;  // wait 1 (bus) cycle and then drive data

##2; bus.data <= 2;    // wait 2 default clocking cycles, then drive data

bus.data <= ##2 r;     // remember the value of r and then drive
                       // data 2 (bus) cycles later
```

Regardless of when the drive statement executes (due to `event_count` delays), the driven value is assigned to the corresponding signal only at the time specified by the output skew.

It is possible (though not conventional) for a ***cb-prefixed*** drive statement to execute asynchronously at a time that does not correspond to its associated clocking event. Such drive statements shall be processed as if they had executed at the time of the next clocking event. Any values read on the right hand side of the drive statement are read immediately, but the processing of the statement is delayed until the time of the next clocking event. This has implications on synchronous drive resolution (See 15.14.2) and ## cycle delay scheduling.

It shall be an error if a ***cb-prefixed*** synchronous drive uses the intra-assignment NBA syntax form with the conventional # delay control operator.

15.14.1 Drives and nonblocking assignments

Synchronous signal drives are processed as nonblocking assignments.

While the NBA operator is used in the synchronous drive syntax, it is worth noting that these assignments are different than classic Verilog NBA variable assignments. The intention of using this operator is to remind readers of certain similarities shared by synchronous drives and nonblocking assignments. One main similarity is that design variables and wires connected to clocking block outputs and inouts are driven in the NBA region.

Another key NBA-like feature of **`inout clocking`** block ~~variables~~ signals and synchronous drives is that a drive does not change the **`clocking`** block input. This is because reading the input always yields the last sampled value, and not the driven value.

A key difference between synchronous drives and class NBA assignments is that transport delay is not performed by synchronous drives (except in the presence of the intra-assignment cycle delay operator). Another key difference is drive value resolution, discussed in the next section.

15.14.2 Drive value resolution

When more than one synchronous drive is applied to the same **`clocking`** block output (or inout) at the same simulation time, the driven values are checked for conflicts. When conflicting drives are detected, a run-time error is issued, and each conflicting bit is driven to `X` (or 0 for a 2-state port).

For example:

```
clocking pe @(posedge clk);
  output nibble; // four bit output
endclocking

pe.nibble <= 4'b0101;
pe.nibble <= 4'b0011;
```

<update>
The driven value of `nibble` is `4'b0xx1`, regardless of whether `nibble` is a variable ~~**`reg`**~~ or a net ~~**`wire`**~~.

If a given clocking output is driven by more than one assignment in the same time unit, but the assignments are scheduled to mature at different future times due to the use of cycle delay, then no drive value resolution shall be performed. The drives shall be applied with classic Verilog NBA transport delay semantics in this case.

~~If a given clocking output is driven asynchronously at different time units within the same clock cycle, then drive value resolution is performed as if all such assignments were made at the same time unit in which the next clocking event occurs.~~

When the same ~~variable~~ signal is an output from multiple **`clocking`** blocks, the last drive determines the value of the ~~variable~~ signal. This allows a single module to model multirate devices, such as a DDR memory, using a different **`clocking`** block to model each active edge.

For example:

```
reg j;

clocking pe @(posedge clk);
  output j;
endclocking

clocking ne @(negedge clk);
  output j;
endclocking
```

~~The variable~~ signal `j` is an output ~~to~~ from two **clocking** blocks using different clocking events (**posedge** versus **negedge**). When driven, the ~~variable~~ signal `j` shall take on the value most recently assigned by either **clocking** block. A clocking block output only assigns a value to its associated signal in clock cycles where a synchronous drive occurs.

~~The~~ Multiple **clocking** block outputs driving a net (i.e., through different ports) cause the net to be driven to its resolved signal value. When a **clocking** block output corresponds to a wire, a driver for that wire is created that is updated as if by a continuous assignment from a ~~register~~ variable inside the **clocking** block that is updated as a nonblocking assignment. This semantic model applies to each clocking **block** output that drives the net.

<modified>

It is possible to use a non cb-prefixed procedural assignment to assign to a signal which is associated with a **clocking** block output. When the associated signal is a variable, the non cb-prefixed procedural assignment simply assigns a new value to the variable, and the variable shall hold that value until another assignment occurs (either from a **clocking** block output or another non cb-prefixed procedural assignment). It shall be illegal to drive a signal with an explicit continuous assignment or a primitive when that signal is associated with a **clocking** block output.

In 16.2, add the following blue text:

16.2 The program construct

…

<modified>

Type and data declarations within the program are local to the program scope and have static lifetime. Variables declared within the scope of a program are called program variables. Program variables, including variable ports that do not connect to other module, primitive or interface ports, can only be assigned using blocking assignments, continuous assignments, or as output arguments of tasks or functions. Non-program variables can only be assigned using nonblocking assignments. Using nonblocking assignments with program variables or blocking assignments with design (non-program) variables shall be an error. References to program variables from outside any program block shall be an error.

16.2.1 Operation of program port connections in the absence of clocking blocks

The interaction of clocking blocks with program ports is described in Clause 15. Clocking blocks are an important component in establishing race-free behavior between designs and testbenches. However, it is possible to construct a program that contains no clocking blocks. ~~Such programs are more prone to races when interacting with design code.~~ This subclause defines the interaction of program ports with design code in the absence of clocking blocks.

I do not believe the original proposal was the intended behavior

~~Program ports are program-scope objects. As such, program variable ports are subject to the blocking assignment restrictions described in 16.2.~~ Another property of program ports is that they are always connected to design objects (wires and variables), since programs can only be instantiated in design scopes.

Again, I do not believe this proposal was ever the intended behavior. I tend to believe that program ports should be considered design ports. I could be wrong (after all, I have been wrong more times than most due to my extreme seniority :-))

~~In the absence of clocking blocks, the concept of NBA-like drives across the program/design boundary no longer applies (except in the case of an NBA assignment via hierarchical reference). Rather, variables on the other side of a port connection may be updated right away, in the current scheduling region. The same applies to the driving and resolution of wires on the other side of a port connection.~~

~~Constructs that are sensitive to such cross-region updates and drives, however, shall be scheduled in the scheduling region that corresponds to their declarative scope. Thus if a program input variable port is updated in the active region, any continuous assignment that uses that input variable on its right hand side shall be scheduled for evaluation in the reactive region. Similarly, if program code drives an inout wire port with a new value, the change on that port is immediately driven into the connected design scope. The wire is fully resolved and propagated in the reactive region. However, any design constructs that are sensitive to changes on the wire shall be scheduled for evaluation in the active region. Any program constructs that are sensitive to changes on the wire shall be scheduled in the current reactive or re-inactive regions.~~

~~Consider the following example design, which contains both design constructs and program constructs:~~

```
  ~~module m;~~
    ~~reg r;~~
    ~~wire dw1, dw2;~~

    ~~initial begin~~
      ~~r = 0;~~
      ~~#10 r = 1;~~
    ~~end~~

    ~~assign dw1 = r;~~

    ~~p p_i(dw2, dw1);~~

    ~~always @(dw2)~~
      ~~$display("dw2 is %b", dw2);~~
  ~~endmodule~~

  ~~program p(output pw2, input pw1);~~
    ~~assign pw2 = pw1;~~
  ~~endprogram~~
```

~~In this design, the flow of data originates in reg r and terminates in the execution of the always construct. Due to the presence of program p, it is necessary for simulators to perform multiple iterations of the big scheduling loop in Figure 9-1. This is because the assign statement in program p shall not be executed until the reactive region. And then when it executes and triggers activity on the always construct in module m, that always construct is not allowed to execute until the active region in the next iteration of the big loop.~~