Clarifications in clauses 15 and 16, clocking blocks and programs

In 15.2, MODIFY the text as follows:

15.2 Clocking Block Declaration

…

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 read, and an **input** signal cannot be driven.

In 15.10, ADD the following blue text:

15.10 Cycle delay

…

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:

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

~~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.~~

The cycle delay statement shall wait for the specified number of clocking events. If a ##1 statement is executed at a simulation time that is not coincident with the associated clocking event, the calling process shall be delayed a fraction of the associated clock cycle.

Assuming the following declaration: **`default clocking @(posedge clk);`**

These semantics guarantees
1) The following two statements are equivalent: `## N;` ⇔ **`repeat`** `N @(`**`posedge`**` clk);`
2) Symmetry between ## statements, clocking-block drives and samples.

In 15.12, MODIFY the text as follows:

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. In this case, the newly sampled values shall be available for reading at the end of the Observe region processing.

~~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, when the clocking event occurs, the sampled value shall be updated and available for reading the instant the clocking event takes place and before any other statements are executed. 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 input's value and the execution of program code that reads that value.~~

The first 2 sentences are unrelated to the rest of this paragraph, which is not specific to #0 sampling.

Requiring clocking block updates to execute before *other* statements implies that the event regions are ordered. This is a big deviation from the existing scheduling semantics and in any event does not solve the general problem. The last paragraph below offers a general solution.

Finally, the last sentence is informative, not normative so a note is appropriate.

NOTE — When the clocking block event is triggered by the execution of a program, there is a potential race between the update of a clocking-block input value and programs that read that value. This race does not exist when the clocking block event is triggered from within a module.

Upon processing its specified clocking event, a clocking block shall update its sampled values before triggering the event associated with the clocking block name. Thus, a process that waits for the clocking block itself is guaranteed to read the updated sampled values, regardless of the scheduling region in which either the waiting or the triggering processes execute. For example:

```
clocking cb @(negedge clk);
   input  v;
endclocking

always @(cb) $disdplay( cb.v );

always @(negedge clk); $disdplay( cb.v );
```

The first **always** block above is guaranteed to display the updated sampled value of signal v. In contrast, the second **always** exhibits a potential race, and may display the old or the newly updated sampled value.

In 15.14, ADD the following blue text and DELETE the following red strikethrough text:

15.14 Synchronous drives

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

For zero skew clocking block outputs with no cycle delay, synchronous drives shall schedule new values in the NBA region of the current time ~~unit~~ slot. This ~~has the effect of causing the big~~ causes the loop in Figure 9-1 to iterate from the ~~r~~Reactive/~~r~~Re-inactive regions back ~~in~~to the NBA region of the current time ~~unit~~ slot. 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~~ slot.

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

Examples:

```
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 for a drive statement to execute ~~asynchronously~~ at a time that ~~does not correspond to its associated~~ is not coincident with its clocking event. Such drive statements shall be processed as if they had executed at the time of the next clocking event. ~~Any~~ The values ~~read~~ on the right hand side of the drive statement ~~are~~ shall be read immediately, but the processing of the ~~statement~~ value drive 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~~.

~~Note: The synchronous drive syntax does not allow intra-assignment delays like a regular procedural assignment does.~~

Note – Unlike a regular procedural assignment, the synchronous drive syntax does not allow intra-assignment delays.

…

15.14.1 Drives and nonblocking assignments

~~Synchronous signal drives are processed as nonblocking assignments.~~

~~Note: While the non-blocking assignment operator is used in the synchronous drive syntax, these assignments are different than non-blocking variable assignments. The intention of using this operator is to remind readers of certain similarities shared by synchronous drives and non-blocking assignments. One main similarity is that 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.~~

The above is not strictly true. The reason for re-using the NBA syntax was to remind users that at any given timestep, the value read is not the same as the value driven. The fact that they both use the NBA region is coincidental.
Also, referring to clocking block variables as signals is inaccurate; signal is typically used to refer to a variable or a wire, and in this case variable is more accurate.

The paragraph below makes the text normative and less speculative with respect to intent.

Although synchronous drives use the same operator syntax as nonblocking variable assignments, they are not the same. However, they do share certain characteristics. One similarity is that the update of variables and wires connected to clocking block outputs (and inouts) are scheduled in the NBA region. Also, like nonblocking variable assignments, a key feature of inout clocking variables 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.

~~One difference between synchronous drives and classic 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.~~

If intra-assignment delays with synchronous drives are disallowed (as mentioned above) then it doesn't make much sense to dwell on the differences between synchronous drives and regular variable NBAs with regard to transport delay. Finally, the parenthetical expression is self-contradictory.

15.14.2 Drive value resolution

…
The driven value of `nibble` is `4'b0xx1`, regardless of whether `nibble` is a **`reg`** or a **`wire`**.

If a given clocking output is driven by more than one assignment in the same time ~~unit~~ slot, 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, and t~~T~~he drives shall be applied ~~with classic Verilog~~ using ~~NBA~~ transport delay semantics.

~~If a given clocking output is driven asynchronously~~ When multiple drives are applied to a given clocking output at different times ~~units~~ within the same clock cycle (between clocking events) then drive value resolution is performed as if all such ~~assignments~~ drives were made at the same time ~~unit~~ in which the next clocking event occurs.

When the same ~~signal~~ variable is an output from multiple clocking blocks, the last drive determines the value of the variable ~~signal~~. This allows a single module to model multi-rate 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~~ variable 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.
It is possible to use a procedural assignment to assign to a signal which is associated with a **clocking** block output. When the associated signal is a variable, the 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 procedural assignment). It shall be illegal to drive a ~~signal~~ variable with an explicit continuous assignment or a primitive when that signal is associated with a **clocking** block output.

The use of the term “variable” above is appropriate. The declaration “register j” is definitely a variable. Since the text is referring explicitly to variables, using “signal” seems to confuse the issue.

In 16.2, MODIFY the text as follows:

16.2 The program construct

…

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 variables declared as ports, shall only be assigned using blocking assignments, continuous assignments, or as output arguments of tasks or functions. Non-program variables can only be assigned using non-blocking assignments. Using non-blocking 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.

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.

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 ~~a~~Active region, any continuous assignment that uses that input variable on its right hand side shall be scheduled for evaluation in the ~~r~~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 ~~r~~Reactive region. However, any design constructs that are sensitive to changes on the wire shall be scheduled for evaluation in the ~~a~~Active region. Any program constructs that are sensitive to changes on the wire shall be scheduled in the current ~~r~~Reactive or ~~r~~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~~ over the entire loop in Figure 9-1. This is because the **assign** statement in **program** p shall not be executed until the ~~r~~Reactive region. And, ~~then~~ when it executes and triggers activity on the **always** construct in **module** m, that **always** construct is not ~~allowed to~~ executed until the ~~a~~Active region in the next iteration of the ~~big~~ relaxation loop.

Allowing *blocking* continuous assignments between a program and the design does create the potential for races in programs that read the design values directly. While there's no problem with the above proposal, perhaps we should consider the addition of the nonblocking continuous assignment. The only difference between the continuous and the nonblocking assignment is that the update of the LHS would be scheduled in the NBA region. Then the program example above could be written as:

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

This simple enhancement would allow program ports that do not use clocking blocks the same isolation.