Section 13
Clocking Domains

13.1 Introduction (informative)

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 inter-module communication is to be modeled.

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

SystemVerilog adds the **clocking** construct that identifies clock signals, and captures the timing and synchronization requirements of the blocks being modeled. A clocking domain assembles signals that are synchronous to a particular clock, and makes their timing explicit. The clocking domain is a key element in a cycle-based methodology, which enables users to write test-benches 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 test-bench may contain one or more clocking domains, each containing its own clock plus an arbitrary number signals.

EC-CH101

The clocking domain separates the timing and synchronization details from the structural, functional, and procedural elements of a test-bench. Thus, the timing for sampling and driving clocking domain signals is implicit and relative to the clocking-domain's clock. This enables a set of key operations to be written very succinctly, without explicitly using clocks or specifying timing. These operations are:

— Synchronous Events

— Input Sampling

— Synchronous Drives

13.2 Clocking domain declaration

The syntax for the **clocking** construct is:

EC-CH80

```
clocking_decl ::= [ default ] clocking [identifier] clocking_event ;
   { clocking_item }
endclocking
```

EC-CH80

EC-CH46

```
clocking_event ::= @ identifier
                | @ ( event_expression )
event_expression ::= // this item is already defined in the BNF

clocking_item := default default_skew;
| clocking_direction signal_or_assign_list ;

default_skew ::= input skew
                | output skew
                | input skew output skew

clocking_direction ::= input [ skew ]
                     | output [ skew ]
                     | input [ skew ] output [ skew ]
                     | inout

signal_or_assign_list ::= signal_or_assign { , signal_or_assign }
```

```
signal_or_assign ::= signal_identifier [ = hierarchical_expression ]
```

EC-CH80

```
skew ::= edge [ # delay_expression ]   // edge valid only if
     | # delay_expression             // clocking_event is simple edge

edge ::= posedge | negedge

delay_expression ::= unsigned_number | time_literal
```

Editor's Note: Update preceding BNF excerpt with new BNF, once available.

EC-CH80

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

The *identifier* specifies the name of the clocking domain being declared.

The *signal_identfier* identifies a port in the scope enclosing the clocking domain declaration, and declares the name of a signal in the clocking domain. Unless a *hierarchical_expression* is used, both the port and the interface signal will share the same name.

The *clocking_event* designates a particular event to act as the clock for the clocking domain. 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 domain are governed by the clocking event. All **input** or **inout** signals specified in the clocking domain are sampled when the corresponding clock event occurs. Likewise, all **output** or **inout** signals in the clocking domain are driven when the corresponding clock event occurs. Bidirectional signals (**inout**) are sampled as well as driven.

The *skew* parameters determine 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 section 13.3). When the clocking event specifies a simple edge, instead of a number, the skew may be specified as the opposite edge of the signal. A single *skew* may be specified for the entire domain by using a **default** clocking item.

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

Example:

EC-CH46

```
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 domain 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 domain will use a `10ns` input skew and a `2ns` output skew. The next line adds three input signals to the domain: `data`, `ready`, and `enable`; the last signal refers to the hierarchical signal `top.mem1.enable`. The fourth line adds the signal `ack` to the domain, 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.

EC-CH81

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 Section 17.6.. A **1step** input skew allows input signals to sample their steady-state values immediately before the clock event (i.e., at read-only-synchronize immediately before time advanced to the clock event). Unlike other time units, which represent physical units, a step cannot be used to set or modify the either the precision or the timeunit.

13.3 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 13-1 shows the basic sample/drive timing for a positive edge clock.

Figure 13-1—Sample and drive times including skew with respect to the positive edge of the clock.

Editor's Note: Figure still needs to be recreated.

EC-CH47

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
```

EC-CH48

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

EC-CH84

An input skew of **#0** forces a skew of zero. Inputs with zero skew are sampled at the same time as their corresponding clocking event, but to avoid races, they are sampled in the Observe region. Likewise, outputs with zero skew are driven at the same time as their specified clocking event, as nonblocking assignments (in the NBA region).

13.4 Hierarchical expressions

Any signal in a clocking domain can be associated with an arbitrary hierarchical expression. As described above, 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, concatenations, or combinations of signals in other scopes or in the current scope.

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

13.5 Signals in multiple clocking domains

EC-CH85

~~,~~The same signals—clock, inputs, inouts, or outputs—may appear in more than one clocking domain. Clocking domains that use the same clock (or clocking expression) will share the same synchronization event, in the same manner as several latches can be controlled by the same clock. Input semantics are described in section 13.13, and output semantics are described in section 13.14.

13.6 Clocking domain scope and lifetime

EC-CH86

A **clocking** construct 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 clock-domain name and the dot (**.**) operator:

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

EC-CH49

Clocking domains cannot be nested. They cannot be declared inside functions or tasks, or at the global (**$root**) level. Clocking domains can only be declared inside a module, interface or ~~a~~ program (see section 15).

EC-CH49

Clocking domains have static lifetime and scope local to their enclosing module, interface or program.

13.7 Multiple clocking domain example

In this example, a simple test module includes two clocking domains. The program construct used in this example is discussed in section 15.

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

   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

   // program begins here
   ...
   // user can access cd1.data , cd2.cmd , etc…
endprogram
```

The test module can be instantiated and connected to a device under test (`cpu` and `mem`).

```
module top;
   logic phi1, phi2;

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

13.8 Interfaces and clocking domains

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

For example, the previous example could be re-written using interfaces as follows:

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

interface bus_B (input clk);
      wire [8:1] cmd;
      wire 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

      // program begins here
      ...
      // user can access cd1.a.data , cd2.b.cmd , etc…
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, the clocking domain can be written using both interfaces and hierarchical expressions as:

```
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`,…).

13.9 Clocking domain events

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

For example.

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

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

```
@( dram );
```

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

EC-CH18

13.10 Cycle delay:

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

The syntax for the cycle delay statement is:

EC-CH51

```
## [ expression ];
```

Editor's Note: Update preceding syntax with BNF excerpt, once available.

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

EC-CH52 What constitutes a cycle is determined by the default clocking in effect (see section 13.11). If no default clocking has been specified for the current module, interface, or program then the compiler will issue an error.

Example:

EC-CH50
```
## [5];     // wait 5 cycles using the default clocking

## [j + 1]; // wait j+1 cycles using the default clocking
```

13.11 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:

```
default clocking_decl ;          // clocking declaration
```

or

```
default clocking clocking_name ; // existing clocking
```

Editor's Note: Update preceding syntax with BNF excerpt, once available.

The clocking_name must be the name of a clocking domain.

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.

EC-CH53 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 other instantiated modules or interfaces.

Example 1. Declaring a clocking as the default:

EC-CH54
```
program test( input bit clk, input reg [15:0] data )
   default clocking bus @(posedge clk);
      inout data;
   endclocking
   ## [5];
   if ( bus.data == 10 )
      ## [1];
   else
      ...
endprogram
```

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

EC-CH55

EC-CH54
```
clocking busA @(posedge clk1); ... endclocking
clocking busB @(negedge clk2); ... endclocking
module processor ...
   module cpu( interface y )
      default clocking busA ;
      initial begin
         ## [5]; // use busA => (posedge clk1)
         ...
      end
   endmodule
```



```
endmodule
```

EC-CH95

13.12 Synchronous events

EC-CH93

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 section 13.9).

EC-CH94

The syntax for the synchronization operator is given in Section 8,.9

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

These are some example synchronization statements:

— Wait for the next change of signal `ack_1` of clock domain `ram_bus`

```
@(ram_bus.ack_l);
```

— Wait for the next clocking event in clock-domain `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]`. Note that the index `a` is evaluated at runtime.

```
@(negedge dom.sign[a]);
```

— 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);
```

EC-CH58

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

EC-CH95

13.13 Input sampling

EC-CH60

All clocking domain inputs (input or inout) are sampled at the corresponding clocking event. If the input skew is non-zero 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 13-1 in section 13.3). If the input skew is zero then the value sampled corresponds to the signal value in the Observe region.

Samples happen immediately (the calling process does not block). 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.

EC-CH60

When the same signal is an input to multiple clocking domains, the semantics are straightforward; each clocking domain samples the corresponding signal with its own clocking event.

EC-CH95

13.14 Synchronous drives

EC-CH96

Clocking domain 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.

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

```
[ ## event_count ] clockvar_expression <= expression;
```

or

```
clockvar_expression <= [ ## event_count ] expression;
```

Editor's Note: Replace preceding syntax lines with BNF excerpt, once available.

The clockvar_expression is either a bit-select, slice, or the entire clocking domain 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 can be any valid expression that is assignment compatible with the type of the corresponding signal.

The event_count is an integral expression that optionally specifies the number of clocking events (i.e. cycles) that must pass before the statement executes. Specifying a non-zero event_count blocks the current process until the specified number of clocking events have elapsed, otherwise the statement executes at the current time. The event_count uses syntax similar to the cycle-delay operator (see section 13.10), however, the synchronous drive uses the clocking domain 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 statement, but the right-hand side expression is evaluated before the process blocks, instead of after.

Examples:

```
bus.data[3:0] <= 4'h5;  // drive in current cycle

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

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

bus.data <= ##2 r;      // sample r, wait 2 (bus) cycles, the drive
```

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.

EC-CH98

Synchronous signal drives are processed as non-blocking assignments

A key feature of **inout** clocking domain variables and synchronous drives is that a drive does not change the clock domain input. This is because reading the input always yields the last sampled value, and not the current driven value.

13.14.1 Drive value resolution

EC-CH99

When more than one synchronous drive is applied to the same clocking domain **output** (or **inout**) at the same simulation time, the driven values are checked for conflicts. When conflicting drives are detected a runtime 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;
```

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

EC-CH97

When the same variable is an output from multiple clocking domains, the last drive determines the value of the variable. This allows a single module to model multi-rate devices, such as a DDR memory, using a different clocking domain 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 j is an output to two clocking domains using different clocking events (posedge vs. negedge). When driven, the variable j will take on the value most recently assigned by either clocking domain.

Clock-domain outputs driving a net (i.e. through different ports) cause the net to be driven to its resolved signal value. When a clock-domain 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 clock-domain that is updated as a non-blocking assignment.