Chapter 2
SystemVerilog Language and Simulation Semantics Overview

The limits of my language mean the limits of my world.

— Ludwig Wittgenstein

SystemVerilog language evolved from Verilog with three main goals:

1. To add features for describing test benches such that the stimulus generation portion of verification can go hand in hand with the design portion, replacing troublesome ad-hoc means for generating stimuli. Testbenches are often written using Verification Programming Interface(VPI) [7] to connect to external means such as verification languages, C/C++ programs [53], and scripts.
2. To add features for checking the expected behavior in simulation and formal methods. These features are related to assertions.
3. To simplify expressing hardware designs by providing language constructs such as `struct`, `typedef`, and new variants of `always` procedure.

Our objective in this chapter is to introduce some of the important SystemVerilog features that are often needed for writing assertions, or used in conjunction with assertions to support other tasks. We are not describing assertion features as they are covered in the rest of the book, but only assume that the reader has already gained sufficient overview of the assertion features from Chap. 1, at least in concept.

Some of the new features, especially those for writing testbenches, have a close semblance to constructs that you may have seen in programming languages. These include data types such as `struct` and `typedef`, auto-increment operators, classes and programs. They are similar in concept, but deviate sufficiently to suggest studying them without presumptions. Other features such as clocking blocks for describing clocks and interfaces for describing complex interconnections are brand new in purpose as well as in style.

Following the overview of the new features, we present an overview of the simulation semantics that establishes a framework to place the evaluation and semantics of assertions. It forms the basis for understanding assertion features described in this book.

Finally, this chapter is not meant to prepare you for writing testbenches or synthesizable designs. What we cover in this overview should motivate you to look for detailed syntax and semantics in other books on topics of test bench writing,

E. Cerny et al., *The Power of Assertions in SystemVerilog*,
DOI 10.1007/978-1-4419-6600-1_2, © Springer Science+Business Media, LLC 2010

methodology and design [15, 31, 52]. The SystemVerilog Language Reference Manual [7] (referred in the rest of this book as LRM) is, of course, the most comprehensive, and is required as a reference.

In this chapter, we cover the following language features:

- Data type and variable declarations
- New expression operators
- Clocking block and event controls
- New procedures
- **`interface`**
- **`program`**
- `$unit` and `$root`
- **`package`**
- **`bind`** statement
- **`generate`** constructs

2.1 Data Types and Variable Declarations

First, we discuss integral data types which are said to represent logic values. The set of logic values in SystemVerilog have not changed from Verilog.

Integral Data Types

- 1 for true
- 0 for false
- X for unknown
- Z for high impedance

Some data types hold two values (0 and 1), while others hold all four values (0, 1, X and Z). Two-valued data types are:

- **`bit`**, unsigned, a single bit length
- **`byte`**, signed, 8-bit length
- **`shortint`**, signed, 16-bit length
- **`int`**, signed, 32-bit length
- **`longint`**, signed, 64-bit length

Four-valued data types are:

- **`logic`**, unsigned, a single bit length
- **`reg`**, unsigned, a single bit length
- **`integer`**, signed, 32-bit length
- **`time`**, unsigned, 64-bit length

Many other concepts about data types are adopted from common programming languages such as C++ [24, 53] and Java [32]. Some of the data types included in

this category are **enum**, **struct**, **union**, and **string**. Although similar in concept to their counterparts in programming languages, the detailed rules of interpretation have evolved to become incongruous to them. Many extensions are made to these concepts to suit the needs of hardware designers. A complete description of the rules and features can be found in the SystemVerilog LRM. We give a brief introduction to these types below.

enum Type

enum is a convenient way to name individual members of a group of integral constants. For example,

```
enum {on, off, disabled} SwitchStatus;
```

declares variable `SwitchStatus` to obtain one of three values: `on`, `off` and `disabled`. By default, constant values are of type **int**. The type of the constants can also be explicitly stated to be different than **int**. For example,

```
enum byte {on=10, off, disabled} SwitchStatus;
```

specifies the type of each constant to be **byte** and the starting value to be 10. The value of `on` is 10, `off` is 11 and `disabled` is 12.

struct Type

While an array is an aggregate of elements of the same type, **struct** is a set of named elements, where each element can be of a different type.

Example 2.1. A **struct** type declaration

```
bit clk;
struct {
  int id;
  enum {on, off, disabled}SwitchStatus;
  bit [7:0] op;
  bit [127:0] addr;
} trans;
a1: assert property (@(posedge clk) trans.id < 200);
a2: assert property (@(posedge clk)
  (trans.op < 16) |=> (trans.addr < 128));
```

□

Each named element of `trans` can be accessed individually as shown above.

union Type

A slight variation from type **struct** is type **union** which is provided to map one or more members to the same storage, but allows access to the storage by naming either one of them and interpreting the type accordingly.

Example 2.2. A **union** type declaration

```
struct {
  int id;
  enum {on, off, disabled} SwitchStatus;
  bit [7:0] op;
  union {
    bit [127:0] fullAddr;
    bit [3:0][31:0] aSlices;
  } addr;
} trans;
//...
a1: assert property (@(posedge clk) trans.id < 200);
a2: assert property (@(posedge clk)
  trans.op < 16 |=> trans.addr.fullAddr[31:0] < 128);
a3: assert property (@(posedge clk)
  trans.op < 16 |=> trans.addr.aSlices[0] < 128);
```

□

Here, `addr` can be accessed either as `fullAddr` or in 8 32-bit slices from `aSlices`.

The important aspect of **union** is that it saves storage for data that is needed to be referenced in a variety of ways. There is no need to duplicate data or reassign to another data for the sake of getting a part of it.

typedef Type

In Example 2.2, `trans` is a user-defined type. Users can name a type by using **typedef**. Once defined as **typedef**, the name can be used in any place where a type can be used and interpreted as the original type defined by the **typedef**. For readers well acquainted with programming languages will find this concept rather convenient.

Example 2.3. A **typedef** declaration

```
typedef struct {
  int id;
  enum {on, off, disabled} SwitchStatus;
  bit [7:0] op;
  union {
    bit [127:0] fullAddr;
    bit [3:0][31:0] aSlices;
  } addr;
} transT;
transT t1, t2;
```

□

struct `trans` from Example 2.2 is declared as a type name `transT`, which now allows convenient declaration of two variables `t1` and `t2` with the same type.

string Type

Another useful type added to SystemVerilog is **string**. It represents a dynamic array of characters, where each character is of type **byte**. As the representation is

dynamic, one can extend or shrink a variable of type **string**. Numerous other built-in operators and methods are provided such as equating two strings, concatenating multiple strings, and converting a string to other types.

Example 2.4. Use of **string** for messages

```
string urgentPkt = "Packet of high urgency";
string regularPkt = "Packet of regular urgency";
bit    urgent, endPkt;
string arrivalMsg = " arrived before repeat cycle.";
string logMsg;
// ...
always @(endPkt)
  if (urgent) logMsg = {urgentPkt, arrivalMsg};
  else logMsg = {regularPkt, arrivalMsg};
```

□

Strings `urgentPkt`, `regularPkt` and `arrivalMsg` are initialized with literal strings. At every end of a packet indicated by signal `endPkt`, `logMsg` is constructed by concatenating individual substrings. Strings are often useful in constructing and displaying suitable assertion messages of success and failure. Strings can be passed to a **checker**,[1] making it possible, for example, to customize messages from a library of assertions.

2.1.1 Packed and Unpacked Arrays

In SystemVerilog, arrays can be declared as *packed* or *unpacked*. Packed arrays represent contiguous bits, while unpacked arrays do not assume any specific layout of bits. Both forms are useful, and can be converted to either form using *bit-stream casting*.

Packed Array

A packed array is declared by specifying dimensions before the data identifier name, such as

```
bit [3:0][7:0] u1;
```

where `u1` is a two-dimensional array, with the higher level dimension as `[3:0]` and the lower level dimension as `[7:0]`. It can also be viewed as array `u1` consisting of four subarrays, where each subarray is 8-bits long.

A one-dimensional packed array is also referred to as a vector.

Because data types with predefined length are already vectors, they cannot be used in packet array declarations while single bit length data types (**bit**, **logic** and **reg**) can. For example,

```
bit [31:0] status_bits;
```

[1] Checkers are described in Chap. 21

declares `status_bits` as a vector of 32 bits. Another way to represent the same data is by declaring it as a multidimensional packed array. Because packed arrays represent contiguous bits, an equivalent declaration is

```
bit [3:0][7:0] status_bits;
```

More than one dimension for packed arrays is provided simply to slice the data representation in a more intuitive form, so that the slices can be accessed with indices into arrays. In fact, a multidimensional packed array is equivalent to a vector of bits. This makes the conversion between two different dimensioned packed arrays straightforward as there is a common equivalent form of a vector representation, as long as the total number of bits are the same for both arrays.

For example,

```
logic [31:0] f1;
logic [3:0][7:0] f2;
logic [1:0][15:0] f3;
assign f1 = f2;
assign f3 = f1;
```

is the same as,

```
logic [3:0][7:0] f2;
logic [1:0][15:0] f3;
assign f3 = f2;
```

Unpacked Array

For unpacked arrays, the dimensions are specified after the data identifier name, such as

```
bit p1 [3:0][7:0];
```

where `p1` is a two-dimensional array, with the higher level dimension as `[3:0]` and the lower level dimension as `[7:0]`. Similar to the specification of packed arrays, `p1` can be viewed as consisting of four sub-arrays of 8-bits length.

Bit-Stream Casting

Unpacked arrays do not intrinsically translate to a vector of contiguous bits, unless explicitly type casted using bit-stream casting. An example is shown below.

Example 2.5. Use of bit-stream casting

```
typedef  bit unpackedArray [3:0][31:0];
typedef bit [127:0] packedVector;
unpackedArray  aStatus;
packedVector vStatus;
always @(posedge clk)
  vStatus <= packedVector'(aStatus);
```

□

Bit-stream casting `packedVector'(aStatus)` is used in the assignment to `vStatus`, whereby array `aStatus` consisting a total of 128 elements of bits from the array are converted to a vector of 128 bits.

For a complete set of rules for bit-stream casting, the reader is advised to refer to the LRM.

2.1.2 Lifetime of Variables

Variables of any data type have lifetime associated with them. A variable can be **`static`** or **`automatic`**. A static variable can be accessed throughout the simulation, while access to an automatic variable is restricted during the time that its scope of declaration is active. In general, variables inherit the lifetime of their scopes.

Following are some of the important rules that govern whether the lifetime of a variable defaults to **`static`** or **`automatic`**. Variables are **`static`** by default when declared in

- Compilation unit scopes (scopes outside modules, programs, interfaces, and checkers)
- Module, interface, program, checker, package scopes (scopes outside tasks, functions and procedural blocks)
- Static tasks, functions or procedural blocks

Variables are by default **`automatic`** when declared in

- Automatic tasks, functions or procedural blocks
- For-loop initialization
- Classes

The following example illustrates the case of for-loop.

Example 2.6. Use of the for-loop construct

```
always @(posedge clk) begin
  shiftCtrl <= en && shiftCode[3];
  for (int i = 1; i < MAX_WIDTH, i++)
    shiftData[i-1] <= shiftData[i];
end
```

Index variable `i` is automatic by default. The code in Example 2.6 is equivalent to

```
always @(posedge clk) begin
  shiftCtrl <= en && shiftCode[3];
  begin
    automatic int i;
    for (i = 1; i < MAX_WIDTH,i++)
      shiftData[i-1] <= shiftData[i];
  end
end
```

□

To override its default lifetime, a variable can be declared with an explicit lifetime of **static** or **automatic** in tasks, functions, classes, and procedural blocks.

Example 2.7. Overriding default static lifetime

```
1 module m1 (input logic [3:0] in1, output logic [31:0] out1);
2   function [31:0] recur(input logic [3:0] in1);
3     automatic logic [3:0] r1 = in1;
4     if (r1 == 0) recur = 1;
5     else recur = r1 * recur(r1 - 1);
6   endfunction
7   always @(in1)
8     out1 = recur(in1);
9 endmodule
```

Example 2.7 illustrates how a variable in a static function can be overridden to be automatic. Function `recur` is static by default, but variable `r1` is overridden to be automatic by explicitly declaring it as **automatic**. An initial value equal to the value of `in1` is assigned in line 3 to `r1` which gets executed each time `recur` is entered.

Example 2.8. Overriding default automatic lifetime

```
module m2(input logic [3:0] in1, output integer out1,out2);
  function automatic integer recur1(input logic [3:0] in1,
                                    output integer out1);
    static integer nCount = 0;
    if (in1 == 0) begin
      recur1 = 1; out1 = nCount;
    end
    else begin
      nCount = nCount + 1;
      recur1 = in1 * recur1(in1 - 1, out1);
    end
  endfunction
  always @(*)
    out2 = recur1(in1, out1);
endmodule
```

□

In Example 2.8, function `recur1` is declared as **automatic**, so by default `nCount` is automatic. But, variable `nCount` is overridden to be static by explicitly declaring as **static**. Unlike automatic variables, static variables are guaranteed to be initialized only once, and it occurs at the beginning of the simulation. Variable `nCount` is initialized in

```
static integer nCount = 0;
```

2.2 New Expression Operators

In this section, we describe some of the new expression operators available in SystemVerilog. These operators are meant not only for assertions, but also for any code using expressions. The following new operators are discussed:

- **inside**
- Implication and equivalence

2.2.1 inside Operator

SystemVerilog introduced a notation for sets of singular values and set membership operator **inside** on these sets. **inside** denotes the test for set membership.

Example 2.9. **inside** operator

```
1 module m1(input logic [3 : 0] in1, input logic clk);
2   logic c1;
3   assign c1 = in1 inside {2, 4, 6, 8} ;
4   always @(posedge clk)
5     if  (c1==1'b1) $display ("Match for in1.");
6     else if (c1==1'b0) $display ("No match for in1.");
7     else $display ("This should never execute.");
8 endmodule
```

The **inside** operator compares values from the value set against an expression to determine whether there is a match. In Example 2.9, the value of expression `in1` is compared against each value in the set `{2,4,6,8}` in line 3 to compute the assignment value for `c1`.

A singular set consists of a list of individual values and ranges of values. For a range of values, each value in the range is considered as if it were specified as an individual value. This is shown in the example below.

```
module m1(input logic [3 : 0] in1, input logic clk);
  logic c2;
  assign c2 = in1 inside {[1:8], [12:15]};
  //...
endmodule
```

`in1` is compared against values 1–8 and 12–15.

Moreover, an unpacked array is allowed as a set member, in which case, the comparison is made over the constituent singular values of each element of the array as shown in the example below.

```
module m1(input logic [3 : 0] in1, input logic clk);
  logic c3;
  byte ar[4] = '{4,6,8,10};
  assign c3 = in1 inside {2,ar};
  //...
endmodule
```

Here, in1 is compared against the value 2 and each element of array ar, namely, 4, 6, 8, and 10.

The values in the value set do not need to be integral. For nonintegral values, the equality operator == is used, while for integral values the *wildcard equality* operator ==? is used for comparison. Keep in mind that the wildcard equality operator considers values X and Z for a bit as don't care values when they appear in the value set, resulting in a match for the bit regardless of the value of the corresponding bit in the expression.

Example 2.10. **inside** operator on a value set containing x or z

```
module m2(input logic [3:0] in2, input  logic clk);
   logic c4;
   assign c4 = in2 inside {4'b00X1, 4'b1100, 4'b100Z};
   always @(posedge clk)
     if  (c4 == 1'b1) $display("Match for in2.");
     else if (c4 == 1'b0) $display("No match for in2.");
     else $display("This should never execute.");
endmodule
```

□

Example 2.10 shows a value set with values X and Z. Due to X in 4'b00X1 and Z in 4'b100Z, the value set in line 3 is equivalent to

```
{4'b0001, 4'b0011, 4'b00X1, 4'b00Z1,
 4'b1100,
 4'b1000, 4'b1001,4'b100X, 4'b100Z}
```

After comparing the expression with a value from the value set bit by bit, the resulting comparison is 1'bX if the result of the comparison for any bit is 1'bX.

Example 2.11. **inside** operator on an expression containing X or Z

```
module m3(input logic[3:0] in3, input logic clk);
   logic c5;
   assign c5 = in3 inside {4'b00X1, 4'b1100, 4'b100Z};
   always @(posedge clk)
     if  (c5 == 1'b1) $display("Match for in3.");
     else if (c5 == 1'b0) $display("No match for in3.");
     else if (c5 === 1'bX) $display("x/z miscompare for in3.");
     else $display("This should never execute.");
endmodule
```

□

In Example 2.11, c5 will be 1'bX whenever there is a X or Z in in3 and there is no *wildcard* 4-value match with any value from the value set.

In summary, the result of **inside** is calculated after comparing the expression with each value in the value set as follows:

- When there is a match with a value, the result of the **inside** is 1'b1.
- When there is no match, but the result for a comparison is 1'bX, then the result of **inside** is 1'bX.
- Otherwise, the result of the **inside** operator is 1'b0.

2.2.2 Implication and Equivalence Operators

Two new logical operators have been added to SystemVerilog: implication written as `->`, and equivalence written as `<->`.

Example 2.12. Assertion using logical implication operator

```
a1: assert property (@(posedge clk) rdy -> !rst);
          else $error ("Reset must be low.");
```
□

Using the implication operator, the assert statement ensures that signal `rst` must be low when signal `rdy` is high. When `rdy` is low, `rst` can be low or high.

For the equivalence operator, both signals must be high or low at the same time.

Example 2.13. Assertion using logical equivalence operator

```
always @(posedge clk)
a22: assert property (@(posedge clk) b1 <-> b2));
          else $error("Mismatch between b1 and b2.");
```
□

In the equivalence operator used in `a2`, whenever `b1` is true, `b2` must be true, and vice versa. Similarly, if `b1` is false, `b2` must be false, and vice versa.

In essence, these operators are shortcuts for other logical expressions. The implication `(e1 -> e2)` is logically equivalent to `(!e1 || e2)`, and the equivalence `(e1 <-> e2)` is logically equivalent to `(e1 -> e2)&& (e2 -> e1)`.

2.3 Extensions to Event Controls

Before we discuss clocking blocks, let us review changes to the event control feature. The event control specification has been made easier with two new constructs to simplify writing clocking event expressions: an event operator **`edge`**, and a qualifier **`iff`**.

The event operator **`posedge`** detects a change when a value changes toward 1, while **`negedge`** detects a change when a value changes toward 0. The new event operator **`edge`** is a disjunctive operation of the two, detecting a change when the value changes either toward 1 or 0. For example, **`edge`** shortens the expression **`always`** `@(`**`posedge`** `clk` **`or`** **`negedge`** `clk)` to **`always`** `@(`**`edge`** `clk)`.

An event control can be made conditional by adding the new operator **`iff`** to the event control. This is useful for many cases, such as to express gated clocks, where the event occurs only under certain active signals.

```
always @(edge clk iff clkEnable)
```

Here, the clock is guarded by signal `clkEnable` and is effective only when signal `clkEnable` is high.

Using **`iff`**, the event control is able to express this directly in the above statement. A special form of a blocking statement is the **`wait`** statement, which delays execution until the **`wait`** condition becomes true. An example follows.

Example 2.14. **wait** condition

```
module levelControl(input logic [3:0] in1, in2,
                    input logic clk,
                    output logic [3:0] out1);
  bit [4:0] c1;
  bit [4:0] c2;
  assign c1 = in1;
  assign c2 = in2;
  initial begin
    wait (c1 == 4 && c2 == 5);
    if (c1 == 4) out1 = c1;
    else out1 = c1 + c2;
  end
endmodule
```

□

The difference between an event control and a **wait** statement is that the condition of the **wait** statement is level-sensitive. There is no event expressed with `@` in the condition `(c1 == 4 && c2 == 5)` of the **wait** statement. The condition is just an expression which is evaluated any time the value of any of the variables in the expression changes. When the condition becomes true, the statement is unblocked and the statement proceeds to the next procedural statement. Otherwise, it continues to delay the execution of the next statement.

2.4 Clocking Blocks

Clocking specification is rather crucial to synchronous designs as well as to concurrent assertions. This importance is reflected by the introduction of a new construct referred to as *clocking block*. The primary purpose of this construct is twofold.

One is to provide a flexible scheme for synchronizing and sampling of signals with respect to a design clock. Often the input signals are driven from a test bench to design units, while the test bench is modeled by a *program*. New values to signals for a test are set in the test bench with appropriate delays using a clocking block.

The other one is to enclose property and sequence definitions in a clocking block, using the common event control of the clocking block. This assists in grouping related properties and sequences as well as it provides the convenience of leaving out the explicit specification of a clock for each individual declaration of a property or sequence in the clocking block.

For use of clocking blocks in test benches, readers are advised to refer to the SystemVerilog LRM. A clocking block is declared with

- A name
- An event expression
- Variables sampled and driven by the clocking block
- A list of sequences and properties

The example below illustrates a simple use of a clocking block.

Example 2.15. A clocking block declaration

```
module mCheck;
  bit a,b,c;
  // ...
  clocking mCB @(posedge clk);
    input  a, b, c;
    sequence s1;
      b ##[1:24] c;
    endsequence
    sequence s2;
      a ##3 c;
    endsequence
    sequence s3;
      s1 within s2;
    endsequence
    property p1;
      s1 or s2;
    endproperty
  endclocking
  a1: assert property(mCB.p1) else $error("a1 violation");
  a2: assert property(mCB.s3) else $error("a2 violation");
endmodule
```

□

In Example 2.15, a clocking block named `mCB` is declared with the clocking event `@(posedge clk)`. Variables `a`, `b`, and `c` are used as inputs in the clocking block and they get sampled with the event expression. Sequences `s1`, `s2`, and `s3` are declared within the clocking block and they use the same event expression for the clock as the clocking block. Likewise, property `p1` is declared in the clocking block and uses sequences `s1` and `s2`. It should be noted that a clocking block creates a scope, and all items declared within it fall under that scope. Therefore, assertion `a1` refers to property `p1` as `mCB.p1`. There is one restriction of importance on the declarations of properties and sequences: no explicit clock is allowed in the declarations. Consequently, multiply clocked properties or sequences cannot be declared inside clocking blocks.

Default Clocking

An alternate way to accomplish the same convenience of a common clock, but without the restrictions of confining the sequences and properties to a single clock is by using the clocking block as the default for the module as shown next.

Example 2.16. Using a **`default clocking`**

```
module mCheck;
  bit a,b,c;
  default clocking mCB @(posedge clk); endclocking
  sequence s1;
    b ##[1:24] c;
```

```
  endsequence
  sequence s2;
    a ##3 c;
  endsequence
  sequence s3;
    s1 within s2;
  endsequence
  property p1;
    s1 or s2;
  endproperty
  a3: assert property(p1) $display("a3 succeeds");
              else $error("a3 fails");
endmodule
```

□

The clocking block declared as default becomes the implied clock for those declarations which omit the clock. If needed, a different clock can be explicitly specified. Also, now that we have a declared clocking block, it can be used as an event expression itself. Expression `@(mCB)` is an event triggered by `@(posedge clk)`.

2.5 New Procedures

In addition to **`always`** and **`initial`**, four new procedures are provided.

- `always_comb`
- **`always_latch`**
- **`always_ff`**
- **`final`**

`always_comb`, **`always_latch`**, and **`always_ff`** follow the same syntax as **`always`**.

`always_comb` Procedure

Like the **`always`** `@(*)` phrase, `always_comb` is used to express combinational logic and automatically infers the sensitivity list for the procedure.

Example 2.17. An `always_comb` procedure

```
module m2(logic c, d, clk);
  logic a, b;
  always_comb begin :blk
    a = c & d;
    b = c | d;
    a1: assert (a -> b);
  end
endmodule
```

□

The always_comb procedure in Example 2.17 infers the sensitivity list (c,d) and triggers each time c or d changes. There are some differences of significance between always_comb and **always** @(*) constructs as listed below.

- always_comb executes unconditionally at time 0, while **always** @(*) waits for a change in the value of an element from the sensitivity list.
- The sensitivity list of always_comb includes elements of functions used in the procedure, while **always** @(*) includes only the arguments of function calls.
- Variables on the left-hand side of an assignment, either directly in always_comb or in any invoked function within it, cannot be assigned from any other procedure, while **always** @(*) allows such assignments from multiple processes.
- Statements of always_comb cannot be nonblocking, while there is no such restriction in **always** @(*).

always_latch Procedure

always_latch is identical in behavior to always_comb. It is intended to directly state that the logic included in the procedure represents one or more hardware latches. Taking advantage of this statement, tools often check to ensure that the logic indeed synthesizes to latches only.

Example 2.18. An **always_latch** procedure

```
module latch(input logic [2:0] in1, in2,
             output logic [2:0] out1);
  always_latch
    if (in1 == in2) out1 <= in1;
    else if (in1 < in2) out1 <= ~in1 + in2;
endmodule
```

□

Conventionally, out1 is inferred as a latch by most tools (latching when in1 > in2), although its inference as such is not important for simulation.

always_ff Procedure

always_ff, like **always_latch**, also serves a similar purpose to directly represent synthesizable hardware. The logic contained within **always_ff** synthesizes to sequential logic, such as flip-flops and registers.

Example 2.19. An **always_ff** procedure

```
module always_ff_async5 (input logic clk, rst, set,
                         input logic [3:0] data,
                         output logic out_reg[3:0]);
  always_ff @(posedge clk)
    if (rst) out_reg <= 0;
    else if (set) out_reg <= data;
    else out_reg <= out_reg + 1;
endmodule
```

□

Some restriction must be followed in the procedure, as noted below.

- **always_ff** must contain one and only one event control.
- **always_ff** must not contain any blocking statement other than the one event control.
- Variables on the left-hand side of an assignment, either directly in **always_ff** or in any invoked function within it, cannot be assigned from any other procedure.

final Procedure

final procedure is the opposite of the **initial** procedure. It executes at the end of the simulation. Often it is used as a clean-up routine and for displaying or storing information such as simulation final results, statistics, and coverage data. The users can declare more than one final procedure, in which case, they are executed sequentially, but in an arbitrary order.[2] Effectively, the final procedures constitute a single process in which the procedures execute sequentially.

Example 2.20. A **final** procedure

```
int fCount = 0;
always @(posedge clk) rdy_fail:
assert (rdy -> !rst) else fCount++;
final begin : f1
  $display("Number of assertions rdy_fail failed: %d",fCount);
  $display("Last value of = %h", PC);
end
```
□

Assertion `rdy_fail` increments `fCount` each time it fails. At the end of simulation, **final** procedure `f1` prints the total number of assertion `rdy_fail` failures. A number of restrictions are placed on the statements contained in the final procedure. The restrictions emanate from the requirement that a final procedure must execute in zero time. In particular, the following restrictions should be noted.

- The statements are limited to the kind of statements allowed in functions.
- The final procedure executes only once.
- No events can be scheduled from the procedure as the simulation immediately terminates after completing all final procedures.

2.6 Interfaces

A number of enhancements have been made to improve the specification of connectivity between modules. Modules, as we know, often represent design units. As the designs are getting bigger, so are the number of connections between modules. One

[2] SystemVerilog LRM suggests that the order of execution for **final** procedures be deterministic for a tool.

of the enhancements is to provide automatic connection of module instance ports. The name and size of a module instance port are matched against the name and the size of variables contained in the module where the instance is specified. If both the name and the size match, then the connection is made. This shortcut is called *implicit* port connections and is syntactically denoted as `(.*)`.

Example 2.21. Using implicit port connections

```
module sum(input logic [3:0] data, ctrl,
           input logic [7:0] addr,
           input byte d1,d2,d3,
           output byte v1,v2,v3);
  always_comb begin
    case( ctrl )
    2 : v1 = data + d1;
    4 : v2 = data + d2;
    default: v3 = data + d3;
    endcase
  end
endmodule
module dataAdd();
  logic [7:0] addr;
  logic [3:0] data;
  logic [3:0] ctrl;
  byte d1, d2, d3;
  byte v1, v2, v3;
  sum sumi (.*);
  // ...
endmodule
```

□

Variables `data`, `ctrl`, `addr`, `d1`, `d2`, and `d3` declared in module `dataAdd` are connected to the input ports of instance `sumi` with the same name. Similarly, variables `v1`, `v2`, and `v3` are connected to the output ports of `sumi`.

For simpler cases, the implicit port connections method works very conveniently. Yet, one can easily foresee how this method loses its benefit when the signal names or sizes change. Besides, there is no way to encapsulate a group of related signals together and consider them as a single port.

A container construct **`interface`**, similar to **`module`**, defines signals and other entities. But unlike **`module`**, it can be passed via ports as a group. The above example is now modified by using **`interface`** to illustrate one of the basic motivations behind this feature.

Example 2.22. Using an **`interface`**

```
interface busSigs;
   logic [7:0] addr;
   logic [3:0] data;
   logic [3:0] ctrl;
   byte v1,v2,v3;
   byte d1,d2,d3;
   a1: assert #0 (v1 > v2 + v3);
endinterface
```

```
module sum(busSigs sig);
  always_comb begin
    case( sig.ctrl )
    2: sig.v1 = sig.data + sig.d1;
    4: sig.v2 = sig.data + sig.d2;
    default: sig.v3 = sig.data + sig.d3;
    endcase
  end
endmodule
module dataAdd();
  busSigs busI();
  sum sumi (busI);
  //...
endmodule
```
□

Now, all signals are bundled into interface `busSigs`, and simply the interface is connected to module `sum` by instance `busI`. Individual signals such as `ctrl` and `addr` from the interface are accessed directly from the interface instance.

One clear advantage of this encapsulation is that the individual signals are no longer tied to the module ports. Rather, related signals are syntactically represented as a group as well as passed as a group. The contents of an interface can change, such as the data width or even the number of control signals, without affecting the module port definitions. Surely, the final usage of the interface would need to be modified further in the design, but the changes do not percolate beyond where they are needed.

The contents of an interface are not limited to signals. In fact, it may contain most entities allowed in modules such as,

- Data types and variables
- Clocking blocks, functions and tasks
- **`initial`**, **`always`**, and **`final`** procedures to define additional behavior
- Sequences, properties, and assertions

The assertions in an interface can ensure that the behavior included in the interface is checked and this monitoring is encapsulated well within it. Assertion `s1` in **`interface`** `busSigs` is one example of this usage.

Another common use is to form a core of assertions to monitor the relationships within a group of related input signals of an interface. The interface is instantiated with selected design signals, forming connections to the input signals of the interface, thereby applying assertions to different sets of signals. In a similar way, sequences and properties defined in an interface can be accessed and reused to configure assertions in various portions of the design.

Example 2.23. Assertions in an **`interface`**

```
interface busCheck(input logic [7:0] addr,
                   input logic [3:0] data,
                   input logic [3:0] ctrl,
                   input logic clk, rst);
```

```
  a1: assert property (@(posedge clk) ctrl[0] -> !rst);
  a2: assert property (@(posedge clk) $onehot(ctrl));
  a3: assert property
      (@(posedge clk) ctrl[0] |=> $stable(data));
endinterface
module dataAdd();
  logic [1:0][7:0] addr;
  logic [1:0][3:0] data;
  logic [1:0][3:0] ctrl;
  logic clock, reset;
  busCheck bus1(addr[0], data[0], ctrl[0], clock, reset);
  busCheck bus2(addr[1], data[1], ctrl[1], clock, reset);
  //...
endmodule
```
□

Other important features of **interface** which we do not discuss here are:

- **modport** that controls the directions of signals and their visibility to the environment where the interface is instantiated
- Importing and exporting functions and tasks using **modport**
- Exporting clocking blocks for synchronous signals
- Parameterizing interfaces for customization of the interface contents as done for modules
- Virtual interfaces to select and configure interfaces and their connections at runtime

2.7 Programs

SystemVerilog provides features to clearly separate the testbench code from the design code. Design code is conventionally expressed in modules and the hierarchy is built up from modules and its instances. As a counterpart to modules, programs are used for describing testbenches with the primary purpose of generating and sending stimuli to the design signals and receiving responses to validate the design behavior. The code within **program** is referred to as *program block*.

A simple example of a program is shown below.

Example 2.24. A **program** declaration

```
program test (input logic clk, reset,
              output logic [7:0] addr, data, ctrl);
  integer lastD;
  integer lastA;
  initial begin
    string log = "Descriptor test";
    lastD = 0;
    lastA = 0;
    repeat (100) begin
      @clk; #1;
      if (!reset) begin
```

```
        ctrl =  $random % 256;
        lastD = lastD +2;
        data = lastD;
        addr = lastA++;
      end
    end
  end
endprogram
```

□

Program `test` sets values for `ctrl`, `data`, and `addr` for each clock cycle. The fundamental differentiating aspect of a program block is its order of execution in relation to the execution of code in a module. In the cycle of evaluation, the statements in modules are executed first (Active region), followed by the evaluation of assertions (Observed region), and finally the statements of a program block are executed. This is illustrated in Fig. 2.2 and explained in detail in Sect. 2.12.3.

By embodying a systematic scheme of ordered evaluation, the tightly coupled course of stimulus generation and response acknowledgment is made to work efficiently, and the hazards of races between the inputs and outputs of the program block are averted. Moreover, explicit delays and synchronization that were needed before are minimized because the order of execution is deterministic and predetermined.

The program construct resembles the module construct in its declaration of ports and body. A program can contain:

- Ports such as modules
- Data types, nets, and variables
- Class declarations
- Function and task declarations
- Sequences, properties, and assertions
- Covergroups
- Initial and final procedures
- Continuous assignments
- Generate statements

A notable exception is the exclusion of **`always`** procedures from this list.

2.8 $Unit and $Root

Generally, there are two phases of processing in building a simulation model from a given set of source files: compilation and elaboration. Compilation reads one or more source files, performs syntactic and semantic analysis, and stores an intermediate model ready for elaboration. The source files can be compiled all at once or divided into multiple sets of files so that each set can be compiled separately into what is known as a *compilation unit*.

To build the final simulation model, all compilation units together must go through the elaboration phase which binds all components by evaluating parameter

values, connecting instances, building hierarchies, and resolving references. For noninterpretive simulators, another step is needed to create object code from the elaborated model to create the final executable simulation model.

In SystemVerilog, each separately compiled unit can have declarations outside any scope to create a global scope for that compilation unit. This global scope is denoted as `$unit`. A declaration in `$unit` space is visible to all scopes within that compilation unit but not to scopes in other compilation units. An identifier in `$unit` scope can be referenced directly without a hierarchical path. This is illustrated in the following example.

Example 2.25. Declarations of a `$unit` and modules

```
wire [1:0] out; // $unit
wire [1:0] in; // $unit

module topM;
  //basic gate instantiations
  and g1(out[0], in[0], in[1]);
  xnor g2(out[1],in[0], in[1]);
  main m1 ();
endmodule

module main;
  initial  begin
    #0 in = 0;
    #20
    $finish();
  end
  always #5 in++;
endmodule
```

□

Signals `in` and `out` are referenced in modules `topM` and `main` without using a hierarchical path. These signals from `$unit` are directly visible to all modules.

Because `$unit` is a scope, objects in it can also be referenced with a hierarchical path using `$unit` as the top level scope. This is particularly useful when the name of an object from `$unit` scope is shadowed by another local object of the same name in another scope.

In Example 2.26, a local version of signal `in` is declared in module `main`, which shadows signal `in` from `$unit`. Using syntax `$unit::in`, signal `in` from `$unit` is explicitly referenced in the assign statement. Other references of signal `in` in `main` refer to the locally declared version of the signal. In module `topM`, signals `in` and `out` refer to signals from `$unit`.

Example 2.26. A declaration shadowed by another declaration

```
wire [1:0] out; // $unit
wire [1:0] in; // $unit
//basic gate instantiations
module topM;
  and    g1(out[0], in[0], in[1]);
  xnor   g2(out[1],in[0], in[1]);
```

```
  main m1 ();
endmodule
module main;
  reg [1:0] in;
  assign $unit::in = in;
  initial  begin
    #0 in = 0;
    #20 $finish();
  end
  always #5 in++;
endmodule
```
□

Thus, we have seen that each compilation unit has a special `$unit` scope. When compilation units are elaborated, the root of the design is denoted as `$root`. To explicitly specify a top-down hierarchical path starting from a top level module, the hierarchical path can be prefixed with `$root`, such as `$root.topMod.regA`. Ordinarily this is superfluous, but there are special situations where there is a need to distinguish between upward and downward hierarchical paths.

2.9 Package

The construct **`package`** is intended for reuse of common declarations across compilation units. One can use packages as libraries of useful declarations, such as functions, tasks, properties, sequences, and checkers.

Example 2.27. A package declaration and its usage

```
package p1;
  typedef struct {
    int i; int r;
  } IntPair;
endpackage
module top;
  p1::IntPair a;
  initial begin
    a.i= 10; #5 a.r = 5;
  end
endmodule
```
□

A declaration from a package can be used as shown in Example 2.27. **`package`** `p1` declares a type `IntPair`, which is used in **`module`** `top` to declare variable `a`. When the same declaration is needed multiple times, one can use the import feature which makes the declaration visible throughout the scope.

Example 2.28. Importing a declaration from a **`package`**

```
package p1;
  typedef struct {
    int i; int r;
  } IntPair;
```

```
endpackage
module top;
  import p1::Intpair;
  IntPair a, b;
  initial begin
    a.i= 10;
    b.i = 15;
    #5 a.r = 5;
    b.r = 20;
    end
endmodule
```

□

Once `IntPair` is imported into module `top`, it can be referenced without the syntax `::`. When many declarations are needed from a package, the wildcard import can be used to make all declarations of the package visible throughout the scope.

Example 2.29. Use of a wildcard import

```
package p1;
  typedef struct {
    int i;
    int r;
  } IntPair;
  function IntPair intPairAdd(IntPair a, IntPair b);
    intPairAdd.r = a.r + b.r;
    intPairAdd.i = a.i + b.i;
  endfunction
endpackage
module top;
  import p1::*;
  IntPair a, b, c;
  initial begin
    a.i = 10;
    b.i = 15;
    c = intPairAdd (a, b);
    #5 a.r = 5;
    b.r = 20;
    c = intPairAdd (a, b);
  end
endmodule
```

□

`p1::*` makes all declarations within package `p1` available to module `top`.

While the package import feature provides considerable convenience, the name visibility of a package identifier can create a conflict with an identifier in the scope where the package is imported. There are many rules to disambiguate such conflicts. Among these rules, there are two basic rules to remember.

- If an identifier is visible prior to the source point of the import then,
  - for the wildcard import, the identifier with the same name from the package is not made visible
  - for an explicit identifier import, it is illegal to import an identifier with the same name

- If an identifier is visible after the source point of the import,
  - for the wildcard import, the identifier with the same name from the package is made visible
  - for an explicit identifier import, it is illegal to import an identifier with the same name

Some of the commonly used declarations in a package are:

- Types and classes
- Nets and variables
- Functions and tasks
- Sequences, properties, let declarations, and checkers
- Covergroups

A unique aspect of package usage is that the declarations are truly shared. Accordingly, for example, a package creates only one instance of a declared variable or a net, regardless of how many times a package is imported, and the same variable is referenced in each import.

Example 2.30. Using shared net declarations from a package

```
package common;
  logic [1:0] out;
  logic [1:0] in;
endpackage
module topM;
  import common::*;
  and    g1(out[0], in[0], in[1]);
  xnor   g2(out[1],in[0], in[1]);
  main m1 ();
endmodule
module main;
  import common::*;
  initial  begin #0 in = 0;
    #20 $finish();
  end
  always #5 in++;
  always @(out) $display("out value:%b",out);
endmodule
```

□

Package `common` declares variables for shared use in the design. Same copy of variables `in` and `out` are accessed in both modules `topM` and `main`.

A package provides a natural means to store a library of useful sequence and property definitions. Wherever needed, it is simply imported and assertions are built from the library definitions to suit the application.

2.10 Bind Statement

Frequently, there is a need to bring subordinate code into the main design code. The following are some of the prevailing reasons for using such subordinate code:

- Assertions
- Procedural verification code

- Coverage related code
- A patch to fix a bug

The **bind** construct provides an orderly way to bring such code by instantiating a module, a program, an interface or a checker into the target code from outside the target. An example is shown below.

```
bind dataAdd busCheck firstCheck(.*);
```

An instance of `busCheck` named `firstCheck` is included in module `dataAdd`. The port connections are made according to the *implicit* connection rules in module `dataAdd`.

The purpose of the **bind** statement is to add code in a place without actually modifying that code where it is added. Customarily, once the design is implemented, the design code is maintained with rigor and discipline to minimize code changes. During the verification phase, either verification engineers or designers themselves add code for monitoring the behavior of the design. The **bind** statement is ideal for that purpose, as the monitoring code can be added or removed without affecting the design code. This method is nonintrusive and efficient.

Less frequently, code is also added to temporarily fix a bug in the design code, or work around a deficiency. Once proven to work correctly, the patch is removed and the design code is fixed by actually modifying the code to retain the modified behavior permanently.

The **bind** statement can appear either in a **module**, an **interface** or in a compilation-unit scope. It binds a source instance to one or more target instances. When the source instance is bound, it behaves as if it were instantiated in the target instance. Based on this premise, semantic correctness check of the bound instance is performed.

The source instance can be a module, interface, program, or a checker instance, while the target instance can be a module or an interface instance. More than one source instance can be bound to the same target instance.

The following example illustrates binding a checker to a module instance.

Example 2.31. Binding a **checker** instance to a **module** instance

```
module top;
  logic mclk, sig, snda, outa;
  logic sndb, outb;
  //...
  trans ta(mclk, sig, snda, outa);
  trans tb(mclk, sig, sndb, outb);
  bind trans: ta delayProps p1(req, out, clk);
  bind trans: tb delayProps p1(req, out, clk);
endmodule : top
module trans (input logic clk, req, in,
              output out);
  logic [7:0] tmp;
  always @(posedge clk) begin
    tmp[0] <= in;
```

```
    for (int i = 7; i > 0; i++)
      tmp[i] <= tmp[i - 1];
    out <= tmp[7];
  end
  //...
endmodule
checker delayProps(logic sig, dSig, clk);
  property sigWindow(s, so);
    s |-> ##[1:16] so;
  endproperty;
  a1: assert property (@(posedge clk)sigWindow(sig, dSig));
  //...
endchecker
```

□

The following **bind** statement

```
bind trans: ta delayProps p1(req, out, clk);
```

binds instance `p1` to instance `ta`. This is equivalent to instantiating `p1` in `ta` with the arguments `req`, `out`, `clk` of `p1` connected to `req`, `out`, and `clk` of `ta`, respectively.

Assertion `a1` checks property `sigWindow` between `req` and `out`, without actually modifying module `trans`.

The **bind** statement

```
bind trans: tb delayProps p1(req, out, clk);
```

performs the same binding for instance `tb`. Another variation of syntax is provided for convenience when the same binding is needed for all instances of a module. Rather than specifying instance by instance of the target module, one can just specify a module as the target, in which case the source instance is bound to all instances of that module or interface. In the above case, the two **bind** statements could be replaced by

```
bind trans delayProps p1(req, out, clk);
```

Now, all instances of `trans` are bound with instance `p1`, as if instance `p1` were syntactically placed in module `trans`. There is no limitation on how many **bind** statements can be specified. For example below, two **bind** statements bind two separate instances to the same target instance.

Example 2.32. Binding a **checker** instance to all instances of a **module**

```
checker sigProps(logic req, clk);
  property sigRep(s);
    s |-> ##[1:8] !s;
  endproperty;
  a1: assert property (@(posedge clk)sigRep(req));
  //...
endchecker: sigProps

module top;
  logic mclk, sig, snda, outa;
  logic sndb, outb;
  //...
  trans ta(mclk, sig, snda, outa);
```

```
  trans tb(mclk, sig, sndb, outb);
  bind trans delayProps p1(req, out, clk);
  bind trans sigProps p2(req, clk);
endmodule: top
```
□

In this example, another checker `sigProps` is bound to `trans` in addition to `delayProps` to check a different relationship.

2.11 Generate Constructs

A technique to replicate design descriptions is needed to alleviate mechanical duplication of code by hand.

Example 2.33. Replicated instances

```
module comp (input logic in1, in2, output logic out);
  always_comb
    out = (in1 > in2) ? in1 : in2;
endmodule

module top(input logic [3:0] in1, in2,
           output logic out);
  logic [3:0] ot;
  comp inst1(in1[0], in2[0], ot[0]);
  comp inst1(in1[1], in2[1], ot[1]);
  comp inst1(in1[2], in2[2], ot[2]);
  comp inst1(in1[3], in2[3], ot[3]);
  //...
endmodule
```
□

It is easy to see that the above code is not only time consuming to write, but it also suffers from the risk of inadvertently leading into typing and other mistakes. The generate looping constructs provide a clean way to write such repetitive code, as shown below.

Example 2.34. Using **`generate`** for-loop

```
module top(input logic [3:0] in1, in2,
           output logic out);
  reg [3:0] ot;
  for(genvar i = 0; i < 4; i++) begin: blk
    comp inst(in1[i], in2[i], ot[i]);
  end
  //...
endmodule
```
□

Four instances named `blk[0].inst`, `blk[1].inst`, `blk[2].inst`, and `blk[3].inst` are generated. Here, **`for`** is a generate loop construct which defines a generate scheme for repeating the body of the loop.

Construct **for** consists of

- Looping index declared as **genvar**.
- Looping control consisting of index initialization, looping end condition, looping assignment.
- Generate block representing the body of the loop.

The index used in the looping control must be an elaboration time constant, generally consisting of constant literals or parameters. Conventionally, the generate loop iterations are configured based on the elaboration-time parameters.

The looping scheme is processed during the elaboration phase to unroll the generate block as many times as the loop condition holds true. Each iteration of the loop creates an instance of the generate block. After processing, the generate loop is completely replaced by the unrolled code, leaving no looping code for run time execution.

The looping index must be declared as **genvar**. This declaration must exist prior to using the index and its use is confined within the body of the loop. Also, it is treated like a **localparam** and can only be used in places where a **localparam** can be used. A reference to the **genvar** index outside the loop is illegal.

Another generate scheme uses a conditional **if** clause. In this scheme, one of the alternative blocks gets selected, based on the condition.

Example 2.35. Using **generate** if-else statement

```
module mGen  #(width = 8) (input logic in1, in2, in3,
                           output logic y);
  if (width == 1)
    always_comb y = in1;
  else if (width==8)
    always_comb  y = in2;
  else
    always_comb y = in3;
endmodule
```

□

In Example 2.35, for width equal to 1 the statement

```
always_comb y = in1;
```

is used. Similarly, for width equal to 8, statement

```
always_comb  y = in2;
```

is used. Otherwise, statement

```
always_comb y = in3;
```

is used.

Again, the condition of **if** must consist of elaboration time constants. At the elaboration time, the condition is evaluated, resulting in the selection of one of the alternative blocks. After elaboration, only the body of the selected block remains.

For more than one alternative blocks, a generate case statement is often used as shown below. Example 2.36 suggests an alternative implementation for Example 2.35.

Example 2.36. Using **generate** case statement

```
module mGen (input logic in1, in2, in3,
             output logic y);
  parameter width = 8;

  case (width)
    1: always_comb y = in1;
    8: always_comb y = in2;
    default: always_comb y = in3;
  endcase
endmodule
```

□

The generate block in both conditional and looping case can be named or left unnamed. The use of naming the block is that the contents of the block can be identified and referenced with the block name. For the case of looping generate block, the name of the generate block instance is tagged with the index, like an array. In contrast, the contents of an unnamed generate block cannot be referenced from the outside.

In any case, a generate block creates a scope. The references, when legal, to the contents must be made as hierarchical references following the normal scoping rules.

For complex situations, generate constructs can be nested. There is no restriction on the depth of nesting.

The generate block can contain most of the description items, including assertions, properties, and sequences. Disallowed items are stand-alone blocks containing procedural statements.

Syntactically, the generate looping (**for**-loop) and conditional (**if-else** and **case**) constructs are similar to the corresponding procedural statements. What distinguishes one from the other is that a generate construct can only appear outside any procedural code, while a procedural statement can only appear inside procedural code.

Optionally, the generate code can be enclosed within **generate** - **endgenerate** keywords for clear demarcation from the rest of the code. An example of using **generate** - **endgenerate** is shown below.

Example 2.37. Enclosing **generate** by generate-endgenerate keywords

```
module mGen (input logic in1, in2, in3,
             output logic y);
  parameter width = 8;
  generate
    case (width)
      1: always_comb y = in1;
      8: always_comb y = in2;
      default: always_comb y = in3;
    endcase
  endgenerate
endmodule
```

□

The generate feature can be used in modules, programs, interfaces, and checkers. They cannot appear inside any procedural code.

2.12 Simulation Semantics Overview

This section provides an overview of SystemVerilog simulation semantics. SystemVerilog supplies many constructs to fulfill designer and testbench writer requirements for reducing the time to write tests and catching bugs early on in the design cycle. These constructs do not exist in isolation. While one can see the operations of individual constructs, understanding the interactions between the constructs is rather complex. In particular, the order in which the activities must take place is the crux of event semantics underlying the operational semantics of the language constructs.

We can see the necessity of ordering events from the following simple example.

Example 2.38. An example showing the order of execution of statements

```
1  module procReq(logic req, preGrant, grant, clk);
2    logic allow;
3    wire ctr, proceed;
4    assign ctr = allow && preGrant;
5    assign proceed = ctr && grant;
6    always @(posedge clk) begin
7       allow <= req;
8    end
9    always @(posedge proceed)  begin
10      processTh();
11   end
12 endmodule :procReq
```

Assuming that input `clk` transitions from 0 to 1 and no other input changes at that time, the correct order of evaluation is:

1. assignment to `allow` in line 7
2. assignment to `ctr` in line 4
3. assignment to `proceed` in line 5
4. evaluation of `processTh` if `proceed` becomes true in line 10

As we will see in this section, this order of evaluation is obtained by creating events, scheduling events, and performing the computation directed by the scheduled events, all carried out in the order established by the semantic framework to obtain the intended result. The parallelism between the continuous statements and always statements in this example is broken down into ordered discrete events. Thus, in this case, the parallelism is unrolled into a sequential order as directed by the occurrence of events. In other cases, true parallelism may exist between statements, allowing indeterminate order of statement execution and values of variables.

In this section, we review the structure of event ordering, the event regions, the interactions between the execution of statements, and finally, the progression of time through this orderly management of events. Our interest is on the evaluation of assertion constructs as it unfolds over the event semantic structure by stepping through the regions, with often its own order within the regions. Nonetheless, assertion evaluation is tightly integrated with the rest of the language semantics, which dictates a broader discussion of the semantic framework to cover full semantics of assertions. We cover simulation semantics of other language features as needed for a frame of reference to complete the discussion on semantics of assertions.We predominantly make use of queues to explain scheduling and ordering.

Another important facet of SystemVerilog is Programming Language Interface (PLI) (or its newer version VPI) which provides an interface from the evaluation of language constructs to the external environment using other programming languages or scripts. The interface is used to inspect values, change values or get callbacks. There are certain points in the semantic structure where specific groups of VPI functions are allowed to take place. We, however, do not delve into the details of that allotment. The rest of the semantics are largely unaffected by its exclusion.

2.12.1 The Simulation Engine

There are two types of principal activities that help explain the event-driven simulation engine: *update event* and *evaluation event*. We call them *semantic events* to distinguish them from the **`event`** construct in SystemVerilog which is a data type used to name and trigger events.

An update event occurs whenever there is a change in the value of a variable. In Example 2.38, the nonblocking assignment

```
allow <= req;
```

causes an update event if the value of variable `allow` changes as a result of the assignment. The update event may trigger other activities and events dependent on the change in value.

There are many language constructs whose execution is tied to the occurrence of update events. The continuous assign statement in Example 2.38

```
assign ctr = allow && preGrant;
```

is may execute only when either the update event on `allow` or `preGrant` occurs. Similarly, the **`always`** statement in Example 2.38

```
always @(posedge clk) begin
  allow <= req;
end
```

is executed when the event `(posedge clk)` occurs.

However, the execution of a statement may not materialize immediately, but is scheduled as an evaluation event in a queue within a region for execution, based on the type of the statement and its surrounding context. By scheduling evaluation

events in various queues and by executing them later from the queues, the intended order between statements is accomplished. The execution of an evaluation event can result in further update events or evaluation events which are again scheduled. For instance, when the evaluation event for the **always** statement

```
always @(posedge clk) begin
  allow <= req;
end
```

is executed, an evaluation event for the nonblocking assignment

```
allow <= req;
```

emerges and gets scheduled. When this evaluation event for the nonblocking assignment is executed, an update event for variable `allow` is issued, assuming the value of `allow` changes as a result of the assignment. The creation and execution of these semantic events, together with their scheduling in queues is what keeps the simulation engine running. As long as there are scheduling events left to process, the engine keeps executing statements and progresses through time. The simulation ends only when there are no more semantic events left in the queues.

2.12.2 Bringing Order to Events

Now, we can see the important role of queues in the assembly of discordant events into a predictable simulation execution model.[3] First, we focus on the execution of statements specified in the context of design code, rather than assertions or programs that represent test bench code. Semantic events issued from the design code are grouped in a *region* called the Active region. We elaborate upon the notion of a region, including the Active region and other regions, in the next section. For now, we limit our discussion to the activities within the queues of the Active region.

The queues represented in Fig. 2.1 belong to the Active region. There are three principal queues in this region: Active queue, Inactive queue and NBA queue.

As the name suggests, the Active queue contains semantic events pending for immediate execution. Events in this queue may be executed in any order, implying parallelism between events. The code must be written carefully as the indeterminate order of execution can cause unintended effects if the code contains interdependency of the variable value changes in the Active queue. After an event from the queue is executed, such as updating a variable value, it is removed from the queue. Initially at time 0, the initial processes are scheduled in the Active queue. If a statement is encountered with `#0` as the delay control, an evaluation event for the statement is

[3] Here, we deviate from the terminology used in the SystemVerilog LRM. In the LRM, a time slot is divided into regions and some regions are grouped to form a *region set*. We prefer to call regions within a region set as queues, and collapse a region set as simply a region. For example, we call the Active region set as the Active region, while its subregions Active region, Inactive region and NBA region are called Active queue, Inactive queue and NBA queue, respectively.

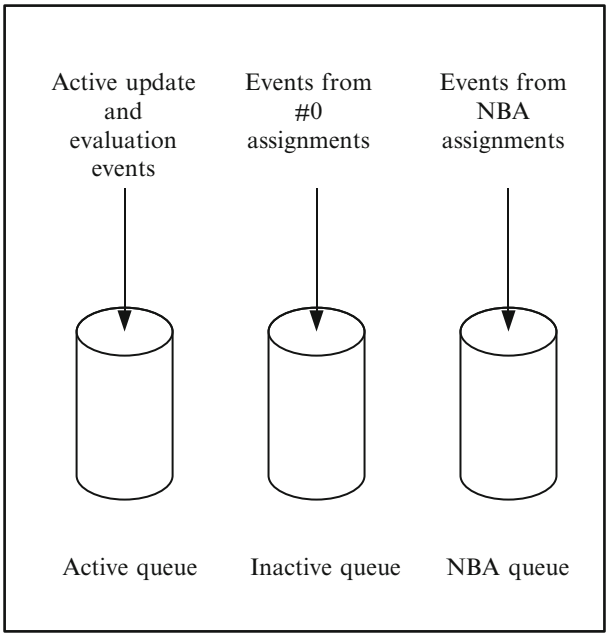

Fig. 2.1 The Active region

entered into the Inactive queue. When all events from the Active queue are executed with the queue being empty, the events from the Inactive queue are transferred to the Active queue, resulting in an empty Inactive queue. The execution of the events from the Active queue resumes once again.

Finally, if the execution encounters a nonblocking assignment, the expression on the right-hand side of the assignment is evaluated and an update event for the variable on the left-hand side with the computed value is appended to the NBA queue. The events from this queue start execution only when the Active and Inactive queues are empty. Unlike the Active queue, the events in the NBA queue are executed in the same order as they are entered in the queue. Therefore, the sequential order of nonblocking assignments in a procedure is replicated in the queue and those statements are executed in order.

Ordinarily, non-blocking assignments model register transfer statements triggered by the clock derived from its enclosing `always` procedure. As a result, the register stores the new value and propagates the value. If there is a combinational logic driven by the register, its value is further propagated by the new update event entered in the Active queue.

Algorithmically, the following steps are taken.

1. Execute semantic events from the Active queue. New semantic events may be issued and entered in the appropriate queues. Execute until all events are consumed.

2. Transfer events from the Inactive queue to the Active queue and return to step 1. Skip this step if there are no events in the Inactive queue.
3. Execute events from the NBA queue. New update events issued from this execution are entered in the Active queue. Return to step 1.

The above algorithm is iterated until there are no more events left in any queue in the current time.

2.12.3 Carving Safe Regions

The event queues Active, Inactive and NBA are all confined to the Active region. The execution iterates over the queues in a region until all events scheduled in any of its queues are executed. Certain events, however, may be scheduled in the other regions.

The processing of events is sequentially ordered into distinct regions, where each region schedules, manages and executes events that are scheduled in the region. The processing proceeds from one region to the next and can iterate until no further processing is needed in any region for the given time-step.

We discuss the role of the following regions. Other regions are not important for the understanding of assertion semantics.

1. Preponed region
2. Active region
3. Observed region
4. Reactive region
5. Postponed region

The processing of regions also takes place in the order as shown above.

One region differs from another because of the kind of events that are handled by it. As we saw in the previous section, the Active region handles events from the design code. A procedural assertion is scheduled in the Observed region when the point of execution reaches that statement in the Active region. An update event on the port connection to a program schedules an update event in the Reactive region. Nevertheless, events scheduled in other regions cannot be executed from a region, which is currently being processed. This makes a region safe from execution interference of statements that belong to a semantically different region.

The Preponed region is a precursor to the time slot, where only the sampling of data values takes place. No value changes or events occur in this region. On the contrary, the postponed region is the tail end of the time slot meant for finishing simulation tasks that do not include value changes or events. Both of these regions are entered only once. Effectively, sampled values of signals do not change through the time slot.

There are two more important regions in the simulation engine to support SystemVerilog features: the Observed region and the Reactive region.

The Observed region is meant for the evaluation of sequences, properties, assertions, and checker code. Values that are not local to a property or a sequence remain constant during the Observed region. The evaluation mechanism and the queues that reside in this region are quite different than in Active region. Nevertheless, events do get scheduled into the Active and Reactive regions.

The Reactive region executes statements from programs. Programs are intended for writing testbenches, as external environments for designs, feeding stimulus, observing results from the stimulus and building tests to exercise the design. This region is a mirror image of the Active region, with similar events, queues and statement execution to the Active region. The corresponding queues are called Reactive queue, Re-Inactive queue, and Re-NBA queue. This region can also schedule events to the Active region, but does not execute events in the Active region.

The simulation engine processes one region at a time, and transitions from one region to the next only after exhausting events and evaluations in a region. The order of the movement between the regions is fixed as follows: Active region, Observed region, and Reactive region. The iterative motion between the regions continues until there are no more events or evaluation tasks left in any region. The regions and their order are depicted in Fig. 2.2.

The simulation engine iterates over the Active, Observed, and Reactive regions.

The example below illustrates how three regions are involved in executing statements.

Example 2.39. Simulation using the three regions

```
module procReq();
  wire req, preGrant, grant, clk;
  logic allow;
  wire ctr,proceed;
  assign ctr = allow && preGrant;
  assign proceed = ctr && grant;
  always @(posedge clk) begin: blk1
    allow <= req;
  end
  always @(posedge proceed)  begin: blk2
    processTh();
  end
  a1: assert property (@(posedge clk) (grant -> preGrant));
  test t1(req, preGrant, grant, clk);
endmodule : procReq
program test(output logic rW, pgW, gW, clkW);
  logic rR, pgR, gR, clkR;
  assign rW = rR;
  assign pgW = pgR;
  assign gW = gR;
  assign clkW = clkR;
  initial begin
```

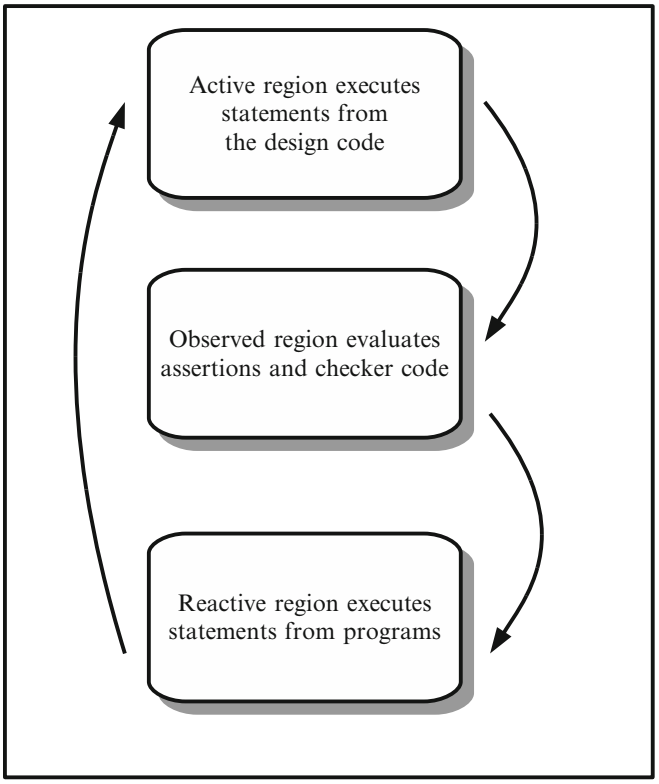

Fig. 2.2 Three main regions

```
    rR = 1;
    #10;
    pgR = 1;
    #10;
    clkR = 1; gR = 1;
  end
endprogram
```

□

At time 0, the continuous assignments are evaluated once to propagate the initial values by scheduling update events in the Active region. The default values of `rR`, `pgR`, `gR`, and `clkR` are assigned to `rW`, `pgW`, `gW`, and `clkW` respectively. These values then propagate via ports to `req`, `preGrant`, `grant`, and `clk`. The clock `clk`, however, does not trigger, so there is no activity in the Observed region. In the Reactive region, `rR` gets assigned to 1 which causes an update event to assign `rR` to `rW`. Likewise, `rW` causes an update event for the port connection to `req`. There are no other events due to program statements. Signal `req` does not cause any event in module `procReq`.

At time 10, the state of the Active and Observed regions is unchanged. `pgR` gets assigned to 1 in the Reactive region, issuing an update event to assign to `pgW` and

its port connection `preGrant`. The new value of `preGrant` does not change the resulting value of `allow && preGrant`, so no new update events are scheduled in the Active region.

At time 20, `clkR` is assigned, followed by an assignment to `clkW` as an update event in the Reactive queue, and correspondingly its port connection to `clk`. This update event to `clk` results in scheduling an evaluation event for **`always`** procedure `blk1` in the Active queue of the Active region. Similarly, assignment `gR` propagates to `grant` via an update event. As a result of the clock `clk` change, assertion `a1` gets scheduled in the Observed region.

After the completion of the Reactive region, the simulation enters the Active region. Here, the execution of `blk1` from the Active queue sets an update event to set the value of `allow` to 1 in the NBA queue. After iterating over the NBA queue, the Active queue is iterated again due to the continuous assignment to `proceed` to 1. Block `blk2` gets executed, invoking task `processTh`. The Active region queues at this time are empty, so the control moves to the Observed region, where assertion `a1` gets evaluated.

Simulation keeps running until there are no more semantic events to execute.

2.12.4 A Time Slot and The Movement of Time

In Example 2.39, we noted that the events get scheduled at different times, such as at time 10 and time 20. Each event is associated with a simulation time, which is the time maintained by the simulator to account for the delays in the design. Without the delays in the design, the simulation time will not advance and all events will occur at time 0.

The time delays in the system are specified with a scale and a precision. Nonetheless, time is discrete and there exists a global time precision which is the smallest unit of time in the system being simulated. *1step* denotes the smallest time precision.

Example 2.40.

```
module m1( );
   timeunit 1ns;
   timeprecision 10ps;
   //...
endmodule
module m2( );
   timeunit 1ns;
   timeprecision 1ps;
   //...
endmodule
```

□

In this example, the smallest unit of time(one step) is `1ps`. The same time unit and precision can also be specified by the timescale compiler directive.

For the purpose of event semantics, a single step is referred to as a *time slot*. We have seen how the event queues and the regions establish the order of processing within a time slot. The time within a time slot remains constant, and thus, all events scheduled within a time slot refer to the same time. When all events are processed for a time slot, the simulation control moves to the nearest time slot containing scheduled events.

> The step denotes the smallest time-step. A single step is referred to as a time slot.

Let us review how an event is scheduled for a time slot associated with a future time. The program portion from Example 2.39 is shown below.

```
program test(output logic rW, pgW, gW, clkW);
    logic rR, pgR, gR, clkR;
    assign rW = rR;
    assign pgW = pgR;
    assign gW = gR;
    assign clkW = clkR;
    initial begin
        rR = 1;
        #10;
        pgR = 1;
        #10;
        clkR = 1; gR = 1;
    end
endprogram
```

When the execution reaches the statement,

```
#10;
```

an evaluation event is scheduled for a future time slot. Let us assume that the time for the time slot is t. The rest of the initial block gets scheduled at time t_1 equal to t plus 10 in the Reactive queue of the Reactive region. The statements scheduled for time t_1 are as follows.

```
pgR = 1;
#10;
clkR = 1;
gR = 1;
```

When the simulation control transitions to time t_1, the scheduled event from the Reactive queue is executed. After executing statement,

```
pgR = 1;
```

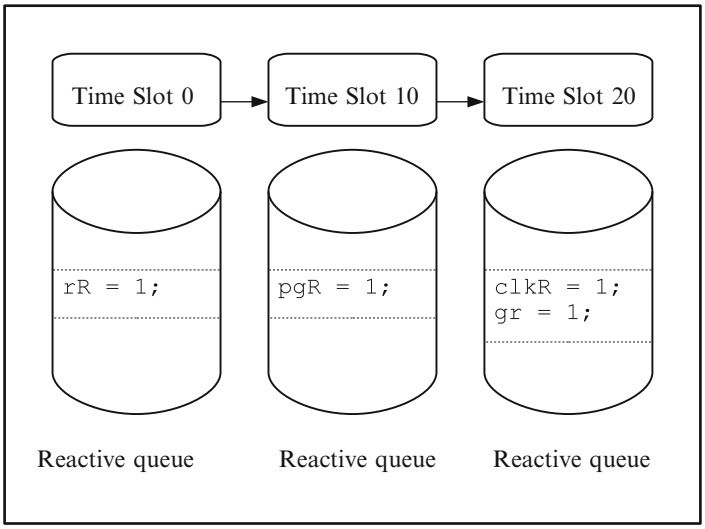

Fig. 2.3 Scheduling for future time slots

another future evaluation event is scheduled in the Reactive queue of the Reactive region at time $t_2 = t_1 + 10$, with the following statements.

```
clkR = 1;
gR = 1;
```

In effect, Fig. 2.3 illustrates the initial state of scheduled events in the queues at the beginning of each time slot.

The events in the Reactive queue, as they execute, give rise to other events, scheduled in the Active queue of the Active region, thereby creating an order in accordance with the semantics of the statements.

The time advances only when the events of the current time slot are exhausted.

We briefly review the impact of assignments on scheduling events.

- A continuous assignment schedules an update event in the Active queue to update the value of its left-hand side, whenever there is a change in the right-hand side expression value.
- A blocking assignment without a delay executes immediately, and issues update events for statements dependent on the new value of the left-hand side. An assignment with 0 intra-assignment delay computes the right-hand side and schedules an evaluation event in the Inactive queue to make the assignment, issue other update events if necessary, and continue the sequential execution from that statement. For a greater delay, it schedules like for 0 delay in the Active queue for the future time.

- A nonblocking assignment schedules an update event in the NBA queue to update the left-hand side based on the current value of the right-hand side. For a delayed statement, it schedules the event for a future time in the NBA queue based on the delay.

These rules apply to the Active and Reactive regions in their corresponding queues. In the Observed region, only certain special statements are executed. Largely, the evaluation of the assertion is carried out without the need of the queues used in other regions. Chapters 3 and 14 discuss assertion simulation semantics in more detail.