2024 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION JAPAN

A Subjective Review on IEEE Std 1800-2023

Kazuya Shinozuka
Artgraphics

accellera SYSTEMS INITIATIVE

IEEE Std 1800-2023

- A new revision of SystemVerilog was published on February 28 this year as IEEE Std 1800-2023.
- The new revision includes corrections and clarifications in the aspect of the language definition in IEEE Std 1800-2017. It has also introduced enhancements that ease design and improve verification effort.
- As for corrections, checking for errata will suffice. Further, many of important clarifications were presented thoroughly at DVCon last year by Dave Rich. Thus, we feel that reviewing of both corrections and clarifications can be safely skipped in this tutorial.
- Thus, this tutorial intends to summarize the enhanced language features that will bring various benefits to users.

What are the enhancements?

- Tripple-quoted strings allow you to insert quotes and new lines without using backslashes, which is convenient for long string literals.
- Soft packed union, which is less restrictive than hard packed one, is added.
- Parameterized class type can be represented with type(this).
- Array method syntax is extended so that index iterator can be specified.
- The array map() method is added to transform one array to another with concise description on the conversion process.
- Specifiers are added to class and class methods to avoid accidental overrides. Similar specifiers are also added to constraint.
- Subclass constructor can call superclass one with super.new(default).
- Variable of real type can be made random.
- It is now possible to extend covergroup in subclass.
- There are some others. Key language enhancements are listed in Table 1 on next page.

Summary of Enhancements

- Enhanced features in this revision are listed below, each of which will be reviewed shortly.

Table 1 List of Enhanced Features

1	triple-quoted string	10	[expr +/- expr], [expr +%- expr]
2	type(this)	11	rand real
3	soft packed union	12	constraint with extends, initial, and final
4	index_argument is added to array manipulation	13	dist_item with default :/ expression
5	array mapping method	14	covergroup can be extended
6	class with final	15	$timeunit
7	class constructor with default	16	$timeprecision
8	class method with extends, initial, and final	17	$stacktrace
9	weak_reference#(T)	18	compiler directives added &&, ||, etc.

Subtle Differences

- As stated elsewhere, the new revision added clarifications on language features that might have misled users. In addition, rigorous descriptions on operation results are also presented. A typical example is shown below.

Table 2 Differences in Operation Result between the Two Revisions

operators	IEEE Std 1800-2017	IEEE Std 1800-2023
relational (<, >, <=, >=)	1, 0, 1'bx	1'b1, 1'b0, 1'bx
equality(==, !=)	1, 0, 1'bx	1'b1, 1'b0, 1'bx
equality (===, !==)	1, 0	1'b1, 1'b0

- Since not all the changes are necessarily covered in this tutorial, a quick perusal of LRM is recommended if concerns about operation result arise.

Triple-quoted string ("""...""")

- In string literals, quotes and new lines must be escaped with backslashes. With triple-quoted string, backslashes can be omitted.
- An example using triple-quoted strings is shown below ([1]).

Tripple-quoted string.

Quotes are not escaped.

A new line is here but invisible.

```
$display("""Humpty Dumpty sat on a "wall".
Humpty Dumpty had a great fall.""");
```

```
Humpty Dumpty sat on a "wall".
Humpty Dumpty had a great fall.
```

Text strings will be printed as shown left.

type(this)

- type(this) represents the type of enclosing class. For example, in the following code, datatype of m_inst is registry#(T).

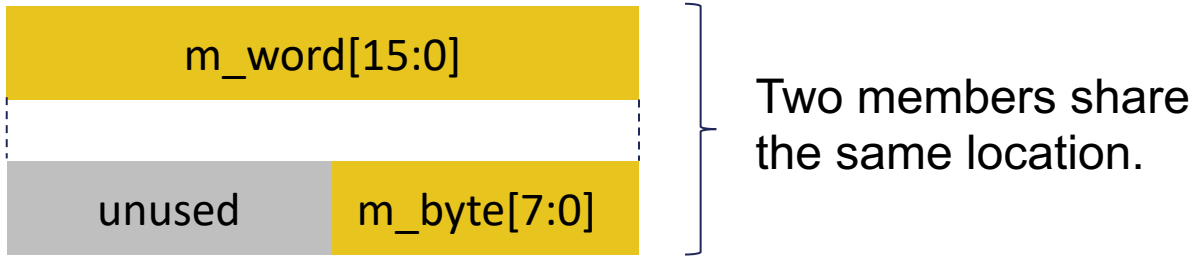
```
class registry #(type T=int);
static type(this)   m_inst;
...
endclass
```

- By using type(this), use of typedef can be avoided, which is one of benefits of this enhancement.
- In addition, even if class parameters change, declaration of m_inst will not be affected at all, which is substantial benefits brought by this feature.

Soft packed union

- In packed untagged union, all members must be integral and have the same size. The soft qualifier is added to alleviate the restriction. In soft packed union, members do not have to be of the same size. The size of soft packed union is determined by maximum size of members. The bits of each member are right-adjusted.
- An example of soft packed union is shown below. When assigning value to m_byte, all MSBs beyond m_byte will stay intact. In other words, m_word[15:8] will not be affected at all.

```
typedef union soft {
        logic [15:0]  m_word;
        logic [7:0]   m_byte;
} word_u;
```



Array manipulation method

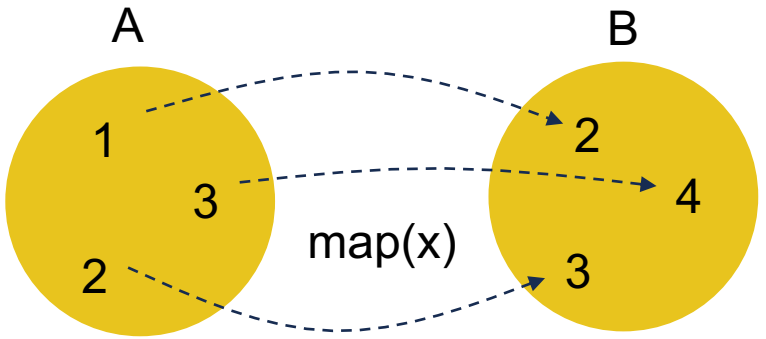
- Previously, array index iterator had fixed name called “index“, which causes conflicts to occur with member names of item. With this enhancement, however, users can specify the name of index iterator to avoid conflicts.
- The following code fragment illustrates use of index iterator. In this example, item has a member called “index”, therefore default index iterator will not work. In this example “iter_index“ is used instead of "index" as shown below.

```
typedef struct {int index; ...} idx_type;
idx_type      arr[], q[$];
...
q = arr.find(item,iter_index) with (item.index != item.iter_index);
```

Array mapping method

- Array mapping method is similar to mathematical map function $y = f(x)$, where both x and y serve as array iterator. Index iterator is also available to users.
- Typically, assignments and foreach are used to copy arrays, but with this enhancement copy can be done in a mathematical way.
- For example, the following code defines $y = f(x) = x + 1$.

```
byte   A[] = '{ 1, 2, 3 },
       B[];
B = A.map(x) with (x+1);   // B = { 2, 3, 4 }
```



Class and final specifier

- Adding the final specifier keeps a class from being extended.

process is a class but cannot be extended by users.

```
class :final process;
      typedef enum {FINISHED, RUNNING, WAITING, SUSPENDED, KILLED} state;
      static function process self();
      function state status();
      function void kill();
      task await();
      function void suspend();
      function void resume();
      function void srandom(int seed);
      function string get_randstate();
      function void set_randstate(string state);
endclass
```

Class constructor with default keyword

- The argument list of a subclass constructor may include default keyword that represents the whole argument list of superclass constructor. With it, superclass constructor can be called as simple as super.new(default).
- As shown below, subclass constructor must have default keyword in the argument list.

```
class sub_t extends base_t;
...
function new(default,byte v);
        super.new(default);
        ...
endfunction
endclass
```

Subclass constructor must have default keyword in its argument list.

If this statement is absent, it will be automatically generated.

Class method with extends, initial, and final

- In order to avoid accidental mistakes, specifiers extends, initial and final are added in method declaration. These specifiers must be placed after class keyword and preceded by a colon(:).
- Method qualified with extends must be a virtual override, which means that corresponding method must be defined as virtual in base class.
- Conversely, method qualified with initial may not be a virtual override, which implies that the method shall not be defined as virtual in base class.
- Method may be qualified with final to indicate that no further overrides of the method shall be allowed.

An example using class method specifiers

- Methods build_phase() and run_phase() are supposedly virtual overrides, and each one is attached the extends specifier.
- Conversely, get_and_drive() and drive_dut() are not virtual overrides. Thus, they are assigned initial.

```
class simple_driver_t extends uvm_driver #(simple_item_t);
virtual simple_if   vif;
`uvm_component_utils(simple_driver_t)
function new(default);
       super.new(default);
endfunction
extern function :extends void build_phase(uvm_phase phase);
extern task :extends run_phase(uvm_phase phase);
extern task :initial get_and_drive();
extern task :initial drive_dut(input simple_item_t item);
endclass
```

[expr +/- expr], [expr +%- expr]

- Operator +/- is called absolute tolerance and +%- relative tolerance, each of which is defined as follows.

```
[A +/- B] ::= [A-type(A)'(B) : A+type(A)'(B)]
[A +%- B] ::= [A-type(A)'(A*B/100.0) : A+type(A)'(A*B/100.0)]
```

- In relative tolerance, when real value is converted to integer, it will be truncated.

```
case for real type: A=7.0

[A+%-25]
=[A-real'(A*25/100.0):A+real'(A*25/100.0)]
=[7.0-1.75:7.0+1.75]
=[5.25:8.75]
```

```
case for integral type: A=7

[A+%-25]
=[A-int'(A*25/100.0):A+int'(A*25/100.0)]
=[7-1:7+1]
=[6:8]
```

rand real

- Previously, only integral type is allowed for random variables. Now, real type variable can be made random.
- Typical example is shown below.

```
class sample_t;
rand logic [7:0]     a;
rand real            r;
constraint C1 {
       a inside {[0:7]};
       r > 0.0 && r < 2.0;
}
endclass
```

constraint with extends, initial, and final

- Similar to class method, constraint can also be qualified with extends, initial and final.
- A constraint with initial specifier shall not be an override, which implies that it will be an error if it is defined in base class.
- A constraint with extends specifier shall be an override, which means that it must be defined in base class as well.
- A constraint with final indicates that no further override is possible in any subclass.
- Note that initial and extends are mutually exclusive.

Examples of constraint with specifiers

- Shown below are examples of typical usage.

```
class base_t;
constraint C1 {}
constraint :final FC {}
endclass
```

```
class sub2_t extends base_t;
constraint :initial C1 {}  // error
constraint :extends C2 {}  // error
constraint FC {}           // error
endclass
```

```
class sub1_t extends base_t;
constraint :extends C1 {}
constraint :initial C2 {}
constraint :final C3 {}
endclass
```

```
class sample_t;
constraint :extends C {}   // error
endclass
```

dist_item with default :/ expression

- When dist is used, value outside the designated ranges makes the constraint false. Adding default :/ into the constraint makes it true even if value is outside the ranges. In other words, default implies “otherwise” similar to default used in case statement.
- In the following example, any value of x is accepted. However, if "defaut:/1" is omitted, only value in ranges [0:3] and [8:10] are accepted.

```
constraint C1 { x dist { [0:3]:= 3, [8:10]:/2, default:/1}; }
```

Covergroup can be extended

- If a covergroup is defined in a class, it can be extended in subclasses.
- Shown below is a typical way to extend base covergroup.

```
class base_t;
rand bit [2:0]       a;
covergroup cg(int low,int high);
  coverpoint a {
    ignore_bins value ={ [low:high] };
  }
endgroup
function new(int low,int high);
       cg = new(low,high);
endfunction
endclass
```

```
class sub_t extends base_t;
rand logic [3:0]     value;
covergroup extends cg;
  coverpoint value {
    bins value_bins[4] = {[0:15]};
  }
endgroup
function new(default);
       super.new(default);
endfunction
endclass
```

$timeunit, $timeprecision

- Both system functions return value in the range [-15:2] according to time unit and time precision setups.
- Return value n indicates 10^n seconds. Thus, -15 means 1fs, -14 is 10fs, -13 is 100fs, and so on.

```
function string get_timeunit(int v);
string  unit;
int     n;
        case (v) inside
        [-15:-13]: begin unit="fs"; n=v+15; end
        [-12:-10]: begin unit="ps"; n=v+12; end
        [-9:-7]:   begin unit="ns"; n=v+9; end
        [-6:-4]:   begin unit="us"; n=v+6; end
        [-3:-1]:   begin unit="ms"; n=v+3; end
        [0:2]:     begin unit="s";  n=v; end
        endcase
        get_timeunit = {"1",{n{"0"}},unit};
endfunction
```

Given a value v in the range [-15:2], this function returns a string indicating time units. For instance, if v==2, then the function returns “100s“.

$stacktrace

- This system task can be used to retrieve the call stack. Although the definition of “call stack“ is not readily available in LRM, presumably it implies traces of subroutine calls and process invocations.
- It can be called as either a task or a function.
- However, information to be collected is highly implementation dependent. Please consult your tool manual for details.

A possible example of $stacktrace

- Below is information retrieved from a tool.

```
module test;
initial begin
        run();
end
task run();
        $stacktrace;
        fork
                proc1(1);
                proc2("sum");
        join_none
endtask
task proc1(int v);
        $stacktrace;
endtask
task proc2(string name);
        #20;
        $stacktrace;
endtask
endmodule
```

```
STACKTRACE StackTrace_N003.sv(10) @0:
  (1) StackTrace_N003.sv(5) test::initial
  (2) StackTrace_N003.sv(9) test::run()
STACKTRACE StackTrace_N003.sv(18) @0:
  (1) StackTrace_N003.sv(5) test::initial
  (2) StackTrace_N003.sv(9) test::run()
  (3) StackTrace_N003.sv(17) test::proc1(input int signed v)
STACKTRACE StackTrace_N003.sv(23) @20:
  (1) StackTrace_N003.sv(5) test::initial
  (2) StackTrace_N003.sv(9) test::run()
  (3) StackTrace_N003.sv(21) test::proc2(input string name)
```

Compiler directives

- `ifdef, `ifndef, `elsif macros added the following logical operator.

```
binary_logical_operator ::= && | || | -> | <->
```

- For example, operators can be used as shown below.

```
`define       macro1_m
module test;
initial begin
       `ifdef (macro1_m || macro2_m)
              $display("`ifdef (macro1_m || macro2_m)");
       `endif
       `ifdef (macro1_m && !macro2_m)
              $display("`ifdef (macro1_m && !macro2_m)");
       `endif
end
endmodule
```

weak_reference#(T)

- weak_reference#(T), where parameter T is a class type, is a parameterized class that allows access to an object while not preventing garbage collection (GC). It can be redefined by user code in any other scope.
- When a class handle is assigned object, it becomes strong reference. The object will not be deallocated as long as strong references exist.
- A weak reference is a sort of handle but indirect and ignored by GC. That is, when strong references no longer exist, GC starts reclaiming space used by unreferenced objects by clearing up weak references.
- Strong reference can be created at any time from weak reference's get() method unless it is cleared yet. Thus, effective use of weak references helps GC work efficiently.

Weak reference's methods

- The weak reference class provides the following methods:
  - Create a new weak reference: new()
  - Query the referent of the weak reference: get()
  - Clear the referent of the weak reference: clear()
  - Query the identification value for an object: get_id()

```
function new(T referent);
function T get();
function void clear();
static function longint get_id(T obj);
```

```
obj strong_obj = new();
weak_reference#(obj) weak_obj = new(strong_obj);
obj get_obj = weak_obj.get();     // get_obj is a strong reference!
```

An example usage of weak reference

- The following is a typical use of weak reference. However, functionality highly depends on implementation. Please consult your tool manual.

```
class sample_t;
...
endclass
```

```
module test;
sample_t                        sample;
weak_reference#(sample_t)       weak_obj;
initial begin
        sample = new;                           // strong reference
        weak_obj = new(sample);                 // weak reference
        wait( weak_obj.get() == null )
                $display("@%0t: weak_obj is cleared",$time);
end
initial begin
        #70;
        sample = null;
end
endmodule
```

Weak reference is useful as long as it's not cleared. At $time==70 or later, wait will be unblocked whenever GC gets ready.

At $time==70, strong reference count becomes 0, which causes GC to clear weak references and start reclaiming the object.

References

[1] IEEE Std 1800-2023: IEEE Standard for SystemVerilog – Unified Hardware Design, Specification and Verification Language.

[2] IEEE Std 1800-2017: IEEE Standard for SystemVerilog – Unified Hardware Design, Specification and Verification Language.

[3] Dave Rich, What's Next for SystemVerilog in the Upcoming IEEE 1800 standard, DVCon 2023.

Thank you

- Any question?