SystemVerilog

SystemVerilog is a Hardware Description and Verification Language based on Verilog. Although it has some features to assist with design, the thrust of the language is in verification of electronic designs. The bulk of the verification functionality is based on the OpenVera language donated by Synopsys. SystemVerilog has just become IEEE standard P1800-2005.

SystemVerilog is an extension of Verilog-2001; all features of that language are available in SystemVerilog. The remainder of this article discusses the features of SystemVerilog not present in Verilog-2001.

Contents

Design features

New data types

Multidimensional packed arrays unify and extend Verilog's notion of "registers" and "memories":

```
reg [1:0][2:0] my_var[32];
```

Classical Verilog permitted only one dimension to be declared to the left of the variable name. SystemVerilog permits any number of such "packed" dimensions. A variable of packed array type maps 1:1 onto an integer arithmetic quantity. In the example above, each element of `my_var` may be used in expressions as a six-bit integer. The dimensions to the right of the name (32 in this case) are referred to as "unpacked" dimensions. As in Verilog-2001, any number of unpacked dimensions is permitted.

Enumerated data types allow numeric quantities to be assigned meaningful names. Variables declared to be of enumerated type cannot be assigned to variables of a different enumerated type without casting. This is not true of parameters, which were the preferred implementation technique for enumerated quantities in Verilog-2001:

```
typedef enum reg [2:0] {
   RED, GREEN, BLUE, CYAN, MAGENTA, YELLOW
} color_t;

color_t   my_color = GREEN;
```

As shown above, the designer can specify an underlying arithmetic type (`reg [2:0]` in this case) which is used to represent the enumeration value. The meta-values X and Z can be used here, possibly to represent illegal states.

New integer types: SystemVerilog defines `byte`, `shortint`, `int` and `longint` as two-state integral types having 8, 16, 32 and 64 bits respectively. A `bit` type is a variable-width two-state type that works much like `reg`. Two-state types lack the X and Z metavalues of classical Verilog; working with these types may result in faster simulation.

Structures and **unions** work much like they do in the C programming language. A SystemVerilog enhancement is the **packed** attribute, which causes the structure or union to be mapped 1:1 onto a packed array of bits:

```
typedef struct packed {
    bit [10:0]  expo;
    bit         sign;
    bit [51:0]  mant;
} FP;

FP     zero = 64'b0;
```

Unique/priority if/case

The **unique** attribute on a cascaded `if` or `case` statement indicates that exactly one branch or case item must execute; it is an error otherwise. The **priority** attribute on an `if` or `case` statement indicates that the choices are to be considered in order, but some branch must execute. These features enforce at simulation time the properties often declared by the de-facto `synopsys full_case parallel_case` annotations for use during synthesis.

Procedural blocks

In addition to Verilog's `always` block, SystemVerilog offers new procedural blocks that better convey the intended design structure. EDA tools can verify that the behavior described is really that which was intended.

An `always_comb` block creates combinational logic. The simulator infers the sensitivity list from the contained statements:

```
always_comb begin
    tmp = b * b - 4 * a * c;
    no_root = (tmp < 0);
end
```

An `always_ff` block is meant to infer synchronous logic:

```
always_ff @(posedge clk)
    count <= count + 1;
```

An `always_latch` block is meant to infer a level-sensitive latch. Again, the sensitivity list is inferred from the code:

```
always_latch
    if (en) q <= d;
```

Verification features

The following verification features are typically not synthesizable. Instead, they assist in the creation of extensible, flexible test benches.

New data types

The `string` data type represents a variable-length text string.

In addition to the static array used in design, SystemVerilog offers dynamic arrays, associative arrays and queues:

```
int da[];       // dynamic array
int da[string]; // associative array, indexed by string
int da[$];      // queue

initial begin
    da = new[16]; // Create 16 elements
end
```

A dynamic array works much like an unpacked array, but it must be dynamically created as shown above. The array can be resized if needed. An associative array can be thought of as a binary search tree with a user-specified key type and data type. The key implies an ordering; the elements of an associative array can be read out in lexicographic order. Finally, a queue provides much of the functionality of the C++ STL deque type: elements can be added and removed from either end efficiently. These primitives allow the creation of complex data structures required for scoreboarding a large design.

Classes

SystemVerilog provides an object-oriented programming model.

SystemVerilog classes support a single-inheritance model. There is no facility that permits conformance of a class to multiple functional interfaces, such as the `interface` feature of Java. SystemVerilog classes can be type-parameterized, providing the basic function of C++ templates. However, function templates and template specialization are not supported.

The polymorphism features are similar to those of C++: the programmer may specify write a `virtual` function to have a derived class gain control of the function.

Encapsulation and data hiding is accomplished using the `local` and `protected` keywords, which must be applied to any item that is to be hidden. By default, all class properties are public.

SystemVerilog class instances are created with the `new` keyword. A constructor denoted by `function new` can be defined. SystemVerilog supports garbage collection, so there is no facility to explicitly destroy class instances.

Example:

```
virtual class Memory;
    virtual function bit [31:0] read(bit [31:0] addr); endfunction
    virtual function void write(bit [31:0] addr, bit [31:0] data);
endfunction
endclass

class SRAM #(parameter AWIDTH=10) extends Memory;
    bit [31:0] mem [1<<AWIDTH];

    virtual function bit [31:0] read(bit [31:0] addr);
        return mem[addr];
    endfunction

    virtual function void write(bit [31:0] addr, bit [31:0] data);
        mem[addr] = data;
    endfunction
endclass
```

Constrained random generation

Integer quantities, defined either in a class definition or as stand-alone variables in some lexical scope, can be assigned random values based on a set of constraints. This feature is useful for creating randomized scenarios for verification.

Within class definitions, the `rand` and `randc` modifiers signal variables that are to undergo randomization. `randc` specifies permutation-based randomization, where a variable will take on all possible values once before any value is repeated. Variables without modifiers are not randomized.

```
class eth_frame;
    rand bit [47:0] dest;
    rand bit [47:0] src;
    rand bit [15:0] type;
    rand byte       payload[];
    bit [31:0]      fcs;
    rand bit [31:0] fcs_corrupt;

    constraint basic {
        payload.size inside {[46:1500]};
    }

    constraint good_fr {
        fcs_corrupt == 0;
    }
endclass
```

In this example, the `fcs` field is not randomized; in practice it will be computed with a CRC generator, and the `fcs_corrupt` field used to corrupt it to inject FCS errors. The two constraints shown are applicable to conforming Ethernet frames. Constraints may be selectively enabled; this feature would be required in the example above to generate corrupt frames. Constraints may be arbitrarily complex, involving interrelationships among variables, implications, and iteration. The SystemVerilog constraint solver is required to find a solution if one exists, but makes no guarantees as to the time it will require to do so.

Assertions

SystemVerilog has its own assertion specification language, similar to Property Specification Language. Assertions are useful for verifying properties of a design that manifest themselves over time.

SystemVerilog assertions are built from **sequences** and **properties**. Properties are a superset of sequences; any sequence may be used as if it were a property, although this is not typically useful.

Sequences consist of boolean expressions augmented with temporal operators. The simplest temporal operator is the ## operator which performs a concatenation:

```
sequence S1;
@(posedge clk)
req ##1 gnt;
endsequence
```

This sequence matches if the `gnt` signal goes high one clock cycle after `req` goes high. Note that all sequence operations are synchronous to a clock.

Other sequential operators include repetition operators, as well as various conjunctions. These operators allow the designer to express complex relationships among design components.

An assertion works by continually attempting to evaluate a sequence or property. An assertion fails if the property fails. The sequence above will fail whenever `req` is low. To accurately express the requirement that `gnt` follow `req` a property is required:

```
property req_gnt;
@(posedge clk)
req |=> gnt;
endproperty

assert_req_gnt: assert property (req_gnt) else $error("req not followed
by gnt.");
```

This example shows an **implication** operator `|=>`. The clause to the left of the implication is called the **antecedent** and the clause to the right is called the **consequent**. Evaluation of an implication starts through repeated attempts to evaluate the antecedent. When the antecedent succeeds, the consequent is attempted, and the success of the assertion depends on the success of the consequent. In this example, the consequent won't be attempted until `req` goes high, after which the property will fail if `gnt` is not high on the following clock.

In addition to assertions, SystemVerilog supports assumptions and coverage of properties. An assumption establishes a condition that a formal logic proving tool must assume to be true. An assertion specifies a property that must be proven true. In simulation, both assertions and assumptions are verified against test stimulus. Property coverage allows the verification engineer to verify that assertions are accurately monitoring the design.

Coverage

Coverage as applied to hardware verification languages refers to the collection of statistics based on sampling events within the simulation. Coverage is used to determine when the device under test (DUT) has been exposed to a sufficient variety of stimuli that there is a high confidence that the DUT is functioning correctly. Note that this differs from code coverage which instruments the design code to ensure that all lines of code in the design have been executed. Functional coverage ensures that all desired corner cases in the design space have been explored.

A SystemVerilog coverage group creates a database of "bins" that store a histogram of values of an associated variable. Cross coverage can also be defined, which creates a histogram representing the Cartesian cross-product of multiple variables.

A sampling event controls when a sample is taken. The sampling event can be a Verilog event, the entry or exit of a block of code, or a call to the `sample` method of the coverage group. Care is required to ensure that data is sampled only when meaningful.

e.g.:

```
class eth_frame;
   // Definitions as above
   covergroup cov;
      coverpoint dest {
          bins bcast[1] = {48'hFFFFFFFFFFFF};
          bins ucast[1] = default;
      }
      coverpoint type {
          bins length[16] = { [0:1535] };
          bins typed[16] = { [1536:32767] };
          bins other[1] = default;
      }
      psize: coverpoint payload.size {
          bins size[] = { 46, [47:63], 64, [65:511], [512:1023],
[1024:1499], 1500 };
      }

      sz_x_t: cross type, psize;
   endgroup
endclass
```

In this example, the verification engineer is interested in the distribution of broadcast and unicast frames, the size/type field and the payload size. The ranges in the payload size coverpoint reflect the interesting corner cases, including minimum and maximum size frames.

Synchronization

A complex test environment consists of reusable verification components that must communicate with one another. SystemVerilog offers two primitives for communication and synchronization: the **mailbox** and the **mutex**. The mutex is modeled as a counting semaphore. The mailbox is modeled as a FIFO. Optionally, the FIFO can be type-parameterized so that only objects of the specified type may be passed through it. Typically, objects are class instances representing **transactions**: elementary operations (e.g. sending a frame) that are executed by the verification components.

External links

- SystemVerilog Tutorial and Information
- SystemVerilog.org

- SystemVerilog LRM (IEEE standards shop)
- Accellera homepage
- SystemVerilog resources and tutorial
- SystemVerilog Tutorial
- SystemVerilog Central