SystemVerilog

Ming-Hwa Wang, Ph.D.
COEN 207 SoC (System-on-Chip) Verification
Department of Computer Engineering
Santa Clara University

Introduction

SystemVerilog is a standard (IEEE std 1800-2005) unified hardware design, specification, and verification language, which provides a set of extensions to the IEEE 1364 Verilog HDL:

- design specification method for both abstract and detailed specifications
- embedded assertions language and application programming interface (API) for coverage and assertions
- testbench language based on manual and automatic methodologies
- direct programming interface (DPI)

Purpose: provide a standard which improves productivity, readability, and reusability of Verilog-based code, extends for higher level of abstraction for system modeling and verification, provides extensive support for directed and constrained-random testbench development, coverage-driven verification, and formal assertion-based verification

Extensions to Verilog

- extended data types
  - C data types: int, typedef, struct, union, enum
  - other data types: bounded queues, logic (0, 1, X, Z) and bit (0, 1), tagged unions
  - dynamic data types: string, class, dynamic queues, dynamic arrays, associated arrays including automatic memory management
  - dynamic casting and bit-stream casting
  - automatic/static specification on per-variable-instance basis
- extended operators
  - wild equality and inequality
  - built-in methods to extend the language
  - operator overloading
  - streaming operators
  - set membership
- extended procedural statements
  - pattern matching on selection statements
  - loop statements
  - C-like jump statements: return, break, continue
  - final blocks that execute at the end of simulation (inverse of initial)
  - extended event control and sequence events
- extended process control
  - extensions to always blocks to include synthesis consistent simulation semantics
  - extensions to fork … join to model pipelines
  - fine-gram process control
- extended tasks and functions
  - C-like void functions
  - pass by reference
  - default arguments
  - argument binding by name
  - optional arguments
  - import/export function for DPI
- classes: object-oriented mechanism (abstraction, encapsulation, safe pointers)
- automatic testbench support with random constraints
- interprocess communication synchronization
  - semaphores
  - mailboxes
  - event extension, event variables, and event sequencing
- clarification and extension of the scheduling semantics
- cycle-based functionality: clocking blocks and cycle-based attributes
  - cycle-based signal drives and samples
  - synchronous samples
  - race-free program context
- assertion mechanism
  - property and sequence declarations
  - assertions and coverage statements with action blocks
- extended hierarchy support
  - packages for declaration encapsulation with import for controlled access
  - compilation-unit scope nested modules and extern modules for separation compilation support
  - extension of port declarations to support interfaces, events, and variables
  - $root to provide unambiguous access using hierarchical references
- interfaces to encapsulate communication and facilitate communication-oriented design
- functional coverage
- DPI for clean, efficient interoperation with other languages (C provided)
- assertion API
- coverage API
- data read API
- Verilog procedure interface (VPI) extension for SystemVerilog constructs
- concurrent assertion formal semantics

Extended Literal Values

- integer literals and logic literals
  - unsized literal with a preceding apostrophe (’), e.g.,
    `’0, ’1, ’X, ’x, ’Z, ’z // sets all bits to this value`
- real literals: fixed-point format or exponent format
- time literals: **s**, **ms**, **us**, **ns**, **ps**, **fs**, **step**
- string literals enclosed in quotes: **\v** for vertical tab, **\f** for form feed, **\a** for bell, **\x**02 for hex number

- array literals: replicate operator (**{{}}**), index/type keys and default values, e.g.,
```
int n[1:2][1:3] = ‘{‘{0,1,2}, ‘{3{4}}};
int m[1,2][1,6] = '{2{'{3{4,5}}}};
    // same as '{'{4,5,4,5,4,5},'{4,5,4,5,4,5}}
typedef int triple [1:3]; $mydisplay(triple'{0,1,2});
triple b = '{1:1, default:0}; // indexes 2 and 3 assigned 0
```
- structure literals, e.g.,
```
typedef struct {int a; shortreal b;} ab;
ab c = '{0, 0.0}; /* same as c = '{a:0, b:0.0}; c =
'{default:0}; or c = ab'{int:0, shortreal:0.0}; */
ab abarr[1:0] = '{'{1, 1.0}, '{2, 2.0}};
typedef struct {int a,b[4];} ab_t;
int a,b,c;
ab_t v1[1:0] [2:0] = '{2{'{3{a,'{2{b,c}}}}}};
// expands to '{'{3{'{a,{2{b,c}}}}}, '{3{{a,'{2{b,c}}}}}}
/* expands to
'{'{'{a,'{2{b,c}}},'{a,'{2{b,c}}},'{a,'{2{b,c}}}},
'{'{a,'{2{b,c}}},'{a,'{2{b,c}}},'{a,'{2{b,c}}} } } */
/* expands to
'{'{'{a,'{b,c,b,c}},'{a,'{b,c,b,c}},'{a,'{b,c,b,c}}},
'{'{a,'{b,c,b,c}},'{a,'{b,c,b,c}},'{a,'{b,c,b,c}}}} */
```

Data Types

A data type is a set of values and a set of operations that can be performed on those values. Data types can be used to declare data objects or to define user-defined data types that are constructed from other data types.

- integer types
  - 2-state - can simulate faster and take less memory: **shortint** (16-bit signed), **int** (32-bit signed), **longint** (64-bit signed), **byte** (8-bit signed or ASCII character), **bit** (unsigned with user-defined vector size)
  - 4-state - can have unknown (‘x) and high-impedance (‘z) values: **logic** (unsigned with user-defined vector size), **reg** (unsigned with user-defined vector size), **integer** (32-bit signed), **time** (64-bit unsigned)
- integral types - the data types that can represent a single basic integer data type: **packed array**, **packed struct**, **packed union**, **enum**, **time**. A simple bit vector type is the data types that can directly represent a one-dimensional packed array of bits.
- real types: **real** (64-bit signed), **shortreal** (32-bit signed)
- **void** data type - for function returns nothing or represent nonexistent data
- **chandle** data type - for storing pointers passed using DPI (default **null**)
  - only allow the following operation with another chandle variable or null or Boolean values: equality (**==**), inequality (**!=**), case equality (**===**), case inequality (**!==**)
  - only allow assignment from another chandle or null
  - chandles can be inserted into associative arrays, can be used within a class, can be passed as arguments to functions or tasks, and can be returned from functions
  - chandles shall not be assigned to variables of any other type, shall not be used as follows: as ports, in sensitivity lists or event expressions, in continuous assignments, in untagged unions, in packed types
- **string**: variable length array of bytes indexed from 0
  - string operators: ==, !=, <, <=, >, >=, {str1, str2, …, strN), {multiplier{str}}, str[index], string.method(…)
  - str.method(…): len( ), putc(int index, byte c), getc(int index), toupper( ), tolower( ), compare(string s), icompare(string s), substr(int i, int j), atoi( ), atohex( ), atooct( ), atobin( ), atoreal( ), itoa(integer i), hextoa(integer i), atoreal( ), octtoa(integer i), bintoa(integer i), realtoa(real r)
- **event** data type: event variables can be explicitly triggered and waited for
  - syntax: **event <**var_name> **[= (<**initial_value> | **null**) **];**
- user-defined types: **typedef** (forward definition and actual definition)
- enumeration data types with strong type checking
  - methods: first( ), last( ), next(int unsigned i=1), prev(int unsigned i=1), num( ), name(int unsigned i)
- structures and unions
  - packed and unpacked, signed and unsigned, 2-state and 4-state
  - a tagged union saves a value and a tag (or a member name) for strong type access checking
- class is declared using the **class … endclass** keywords
  - class properties
  - methods
- casting: a data type can be changed by using a cast (‘) operation
  - static casting
    - <type> ‘ (<expression>) or <type> ‘ { { <literal>, …, <literal> } }
    - <size> ‘ (<expression>)
    - **signed** ‘ (<expression>)
    - $shortrealtobits, $bitstoshortreal, $bits, $itor, $rtoi, $bitstoreal, $realtobits, $signed, $unsigned
  - dynamic casting: $cast
  - bit-stream casting
    - for converting between different aggregate types
  - example

    uses bit-stream casting to model a control packet transfer over a data stream:
```
typedef struct {
    shortint address;
    reg [3:0] code;
    byte command [2];
} Control;
typedef bit Bits [36:1];
```

```
Control p;
Bits stream[$];
p = ...              // initialize control packet
// append packet to unpacked queue of bits
stream = {stream, Bits'(p)}
Control q;
// convert stream back to a Control packet
q = Control'(stream[0]);
stream = stream[1:$]; // remove packet from stream
```

uses bit-stream casting to model a data packet transfer over a byte stream:

```
typedef struct {
    byte length;
    shortint address;
    byte payload[];
    byte chksum;
} Packet;
function Packet genPkt();
    Packet p;
    void'( randomize( p.address, p.length, p.payload )
    with { p.length > 1 && p.payload.size == p.length; }
    );
    p.chksum = p.payload.xor();
    return p;
endfunction
```

The byte stream is modeled using a queue, and a bit-stream cast is used to send the packet over the stream.

```
typedef byte channel_type[$];
channel_type channel;
channel = {channel, channel_type'(genPkt())};
And the code to receive the packet:
Packet p;
int size;
size = channel[0] + 4;
// convert stream to Packet
p = Packet'( channel[0 : size - 1] );
// remove packet data from stream
channel = channel[ size, $ ];
```

Arrays

- in Verilog, all data types can be declared as arrays
- a dimension declared before the object name is referred to as the vector width dimension, and the dimensions declared after the object name are referred to as the array dimensions
- SystemVerilog uses the term packed array to refer to the dimensions declared before the object name, and the term unpacked array is used to refer to the dimensions declared after the object name; a packed array is guaranteed to be represented as a contiguous set of bits, and an unpacked array may or may not be so represented
- multi-dimensional arrays
- uses part-select to refer to a selection of one or more contiguous bits of a single dimension packed array, use slice to refer to a selection of one or more contiguous elements of an array
- array querying functions: $left, $right, $low, $high, $increment, $size, $dimensions, and $unpacked_dimensions
- dynamic arrays: any dimension of an unpacked array whose size can be set or changed at run time
  - **new [** expression **]** [**(** expression **)**]
  - size( )
  - delete( )
- array assignment between fixed-size arrays and dynamic arrays
- arrays as arguments: pass by value
- associative arrays
  - indexing operator: wildcard index type *, string index, class index, integer or int index, signed packed array index, unsigned packed array index, packed struct index, user-defined type index
  - methods: num( ), delete([**input** index]), exists(**input** index), first(**ref** index), last(**ref** index), next(**ref** index), prev(**ref** index)
  - associative array assignment
  - associative arrays are passed as arguments
  - associative array literals use the ‘ {index:value} syntax, index can be **default**
- queues with position 0 represents the first element, and $ represent the last
  - queues are declared using the same syntax as unpacked arrays, but specifying $ as the array size
  - empty queue { }
  - right bound [$:N], where N+1 is the maximum size of the queue
  - operators: indexing, concatenation, slicing, equality
  - methods: size( ), insert(**input int** index, **input** type item), delete(**int** index), pop_front( ), pop_back( ), push_front(**input** type item), push_back(**input** type item),
- array manipulation methods
  - syntax: expr**.**array_method { attribute_instance } [**(** arguments **)**] [**with (** expr **)**]
  - array locator methods: find( ), find_index( ), find_first( ), find_first_index( ), find_last( ), find_last_index( ), min( ), max( ), unique(0, unique_index( )
  - array ordering methods: reverse( ), sort( ), rsort( ), shuffle( )
  - array reduction methods: sum( ), product( ), and( ), or( ), xor( )
  - iterator index querying: index( )

Data Declarations

- data have to be declared before they are used, except implicit nets
- forms of data: literals, parameters, constants (genvars parameters, localparams, specparams), variables (static or dynamic), nets (reg, wire, logic), attributes
- constants

- 3 constructs for defining elaboration-time constants: the **parameter**, **localparam** and **specparam**; the default value of a parameter of an instantiated module can be overridden in each instance of the module using one of the following:
  - implicit in-line parameter redefinition (e.g., foo #(value, value) u1 (...);)
  - explicit in-line parameter redefinition (e.g., foo #(.name(value), .name(value)) u1 (...);)
  - **defparam** statements, using hierarchical path names to redefine each parameter
- value parameters: a module, interface, program, or class can have parameters, which are set during elaboration and are constant during simulation
- **$** as a parameter value to represent unbounded range specification
  - $isunbounded(const_expr)
- type parameters: **parameter type**
- parameter port lists: the **parameter** keyword can be omitted, and a parameter can be depend on earlier parameter
- **const** constants are set during simulation

- variables declared with **var** have default type of **logic**

type	default
4-state integral	'X
2-state integral	'0
real, shortreal	0.0
enumeration	base type default initial value
string	"" (empty string)
event	new event
class	null
chandle (opaque handle)	null

- nets: **trireg**, **wire**, **reg**
  - a net can be written by continuous assignments, by primitive output, or through module port
  - **assign**, **force**, **release**
- scope and lifetime: **automatic** and **static**
  - global and static, local and static, local and automatic
- signal aliasing: the members of an alias list are signals whose bits share the same physical nets, aliasing is performed at elaboration time and can't be undone
- type compatibility: 5 levels
  - matching type: **typedef**, anonymous **enum**, **struct**, or **union**
  - equivalent type
  - assignment compatible: have implicit casting rules
  - cast compatible: have explicit casting rules
  - nonequivalent or type incompatible
- type operator

Classes

- class properties and methods
- constructor **new**( )
- static class properties shared by all instances of the class using **static**
- static class method with automatic variable lifetime: **static task** foo( ); … **end task**
- nonstatic class method with static variable lifetime: **task static** foo( ); … **end task**
- shallow copy (putting an object after **new**) v.s. deep copy (custom code is typically needed)
- **this** and **super**
- it is always legal to assign a subclass variable to a variable of a class higher in the inheritance tree, but it is never legal to directly assign a superclass variable to a variable of one of its subclasses; it is legal to assign a superclass handle to a subclass variable if the superclass handle refers to an object of the given subclass, and use $cast( ) to check whether the assignment is legal
- unqualified (public), **local** (private), and **protected**
- **const**: read-only
  - global constants with initial values (optionally with **static**), instance constants without
- abstract class using **virtual**
- polymorphism
- class scope resolution operator **::**
- the extern qualifier for out-of-block declarations
- parameterized classes
  - the combination of a generic class and the actual parameter values is called a specialization (or variant)
- typedef class: for cross-referencing
- memory management: automatic garbage collection

Operators and Expressions

- assignment_operator ::= **=** | **+=** | **-=** | ***=** | **/=** | **%=** | **&=** | **|=** | **^=** | **<<=** | **>>=** | **<<<=** | **>>>=**
- conditional_expression ::= cond_predicate **?** { attribute_instance } expression **:** expression
- unary_operator ::= **+** | **-** | **!** | **~** | **&** | **~&** | **|** | **~|** | **^** | **~^** | **^~**
- binary_operator ::= **+** | **-** | ***** | **/** | **%** | **==** | **!=** | **===** | **!==** | **==?** | **!=?** | **&&** | **||** | ****** | **<** | **<=** | **>** | **>=** | **&** | **|** | **^** | **^~** | **~^** | **>>** | **<<** | **>>>** | **<<<**
- inc_or_dec_operator ::= **++** | **--**
- unary_module_path_operator ::= **!** | **~** | **&** | **~&** | **|** | **~|** | **^** | **~^** | **^~**
- binary_module_path_operator ::= **==** | **!=** | **&&** | **||** | **&** | **|** | **^** | **^~** | **~^**
- built-in package: **std::**
- concatenation using braces (**{ }**) or replication operator (multiple concatenation)
- assignment patterns for assigning struct fields and array elements using (**'{ }**) either by positions, by type**:**value, or by member**:**value or by **default:**value
  - array assignment pattern, structure assignment pattern,

- tagged union expression and member access: **union tagged {** … **}**
- aggregate expressions
- operator overloading: **bind** op **function** type func_name **(** formals **)**
  - match formal types exactly or the actual types are implicitly cast to formal types
  - the operators that can be overloaded are the arithmetic operators, the relational operators, and assignment
- streaming operators (pack/unpack)
  - **>>** causes data to be streamed in left-to-right order
  - **<<** causes data to be streamed in right-to-left order
  - streaming dynamically sized data using **with [**expr[**+**|**-**]:expr**]**
- conditional operator
- set membership: expr **inside {** open_range_list **}**

Scheduling Semantics

- evaluation event and simulation time
- time wheel or time-ordered set of linked lists
  - a time slot is divided into a set of ordered regions as table below; provide predictable interaction between the design and testbench code (including PLI callbacks)

from previous time slot

region	***semantics***	***note***
preponed	The #1step sampling delay provides the ability to sample data immediately before entering the current time slot	IEEE1364
pre-active	allows PLI application routines (cbAfterDelay, cbNextSimTime, cbAtStartOfSimTime) to read and write values and create events before events in the Active region are evaluated	IEEE1364
active	holds current events being evaluated	IEEE1364 iterative
inactive	holds the events to be evaluated after all the active events are processed	IEEE1364 iterative
pre-NBA	allows PLI application routines (cbNBASynch, cbReadWriteSynch) to read and write values and create events before the events in the NBA region are evaluated	IEEE1364 iterative
NBA	A nonblocking assignment creates an event in this region	IEEE1364 iterative
post-NBA	allows PLI application routines (cbReadWriteSynch) to read and write values and create events after the events in the NBA region are evaluated	IEEE1364 iterative
observed	for the evaluation of the property expressions when they are triggered	iterative
post-observed	allows PLI application routines (currently no PLI callback yet) to read values after properties are evaluated	iterative
reactive	property pass/fail code shall be scheduled here of the current time slot	iterative
re-inactive	a #0 control delay specified in a program block schedules the process for resumption in this region	iterative
pre-postponed	allows PLI application routines (cbAtEndOfSimTime) to read and write values and create events after processing all other regions except the Postponed region	IEEE1364 iterative
postponed	cbReadOnlySynch No new value changes are allowed to happen in the time slot	IEEE1364

to next time slot

- the SystemVerilog simulation reference algorithm

```
execute_simulation {
  T = 0;
  initialize the values of all nets and variables;
  schedule all initialization events into time 0 slot;
  while (some time slot is nonempty) {
    move to the next future nonempty time slot and set T;
    execute_time_slot (T);
  }
}
execute_time_slot {
  execute_region (preponed);
  execute_region (pre-active);
  while (any region in [active...pre-postponed] is
  nonempty) {
    while (any region in [active...post-observed] is
    nonempty) {
      execute_region (active);
      R = first nonempty region in [active...post-
      observed];
      if (R is nonempty) move events in R to the active
      region;
    }
```

```
        while  (any  region  in  [reactive...re-inactive]  is
        nonempty) {
          execute_region (reactive);
          R  =  first  nonempty  region  in  [reactive...re-
          inactive];
          if (R is nonempty)
            move events in R to the reactive region;
        }
        if (all regions in [active … re-inactive] are empty)
          execute_region (pre-postponed);
      }
      execute_region (postponed);
    }
    execute_region {
      while (region is nonempty) {
        E = any event from region;
        remove E from the region;
        if (E is an update event) {
          update the modified object;
          evaluate  processes  sensitive  to  the  object  and
          possibly schedulefurther events for execution;
        } else { /* E is an evaluation event */
          evaluate the process associated with the event and
          possibly schedule further events for execution;
        }
      }
    }
```

- the PLI callback control points
  - 2 kinds of PLI callbacks: those are executed immediately when some specific activity occurs and those that are explicitly registered as a one-shot evaluation event
  - Callbacks and their event region (in the table above)

Procedural Statements and Control Flow

- procedural statements: **initial**, **final**, **always**, **always_comb**, **always_latch**, **always_ff**, **task**, **function**
- control flow
  - selection, loops, jumps
  - task and function calls
  - sequential and parallel blocks
  - timing control
- blocking (**=**) and nonblocking (**<=**) assignments
- selection statements: **if**, **else if**, **else**, **case**, **casez**, **casex**, **default**, **endcase**, **unique** (for mutual exclusive and can be executed in parallel), **priority** (ordered evaluation), **inside** (for set membership), **matches** (using **&&&** in if statements), **?:**
- loop statements: **forever**, **repeat**, **while**, **for**, **do while**, **foreach**
- jump statements: **return**, **break**, **continue**, **disable**
- named blocks and statement labels: **begin end**, **fork join**, **join_any**, **join_none**
- event control: **@**, **#**, **iff**, **posedge**, **negedge**

- **sequence**, **triggered**
- level-sensitive sequence control: **wait**

```
sequence abc;
   @(posedge clk) a ##1 b ##1 c;
endsequence
sequence de;
   @(negedge clk) d ##[2:5] e;
endsequence
program check;
   initial begin
      wait( abc.triggered || de.triggered );
      if( abc.triggered )
         $display( "abc succeeded" );
      if( de.triggered )
         $display( "de succeeded" );
   end
endprogram
```

Processes

- New always blocks – design intent is understood **-** IEEE1800 does not specify which constructs are synthesizable and which are not, EDA vendors implement differently and portability will be an issue
- **always_comb** for modeling combinational logic behavior
  - inferred/implicit sensitivity list (within the block or within any function called within the block), automatically executes once at time zero, the variables written on the LHS of assignment shall not be written to by another process
- **always_latch** for modeling level triggered latch logic behavior
- **always_ff** for modeling edge triggered synthesizable sequential logic behavior

```
always_ff  @(posedge  clock  iff  reset  ==  0  or  posedge  reset)
begin
   r1 <= reset ? 0 : r2 + 1;
   ...
end
```

- **fork ... join** for creating concurrent processes

control option	description
join	the parent process blocks until all the processes spawned by this fork complete
join_any	the parent process blocks until any one of the processes spawned by this fork complete
join_none	the parent process continues to execute concurrently with all the processes spawned by this fork, and the spawned processes do not start executing until the parent thread executes a blocking statement

Assertions

- assertions specify behaviors of the system, and are primarily used to
  - validate the behavior of a design

  - provide functional coverage and generate input stimulus for validation.
- immediate assertions follow simulation event semantics and are executed like a procedure statements
  - the immediate assertion statement is a test of an expression performed when the statement is executed in the procedure code; the expression is non-temporal and is interpreted the same way as an expression in the condition of a procedural **if** statement
  - syntax: **assert (** expression **)** [pass_statement] [**else** fail_statement]
  - if an assertion fails and no **else** clause is specified, the tool shall, by default, call $error, unless a tool specific option, such as a command-line option, is enabled to suppress the failure
  - there are 4 severity levels: $fatal, $error, $warning, and $info; all the severity system tasks shall print a tool-specific message indicating the severity of the failure and specific information about the specific failure, which shall include the following information:
    - the file name and line number of the assertion statement
    - the hierarchical name of the assertion, if it is labeled, or the scope of the assertion if it is not labeled
- a concurrent assertion is based on clock semantics and is evaluated only at the occurrence of a clock tick; the values of variables used in the evaluation are the sampled values, thus a predictable result can be obtained from the evaluation, regardless of the simulator’s internal mechanism of ordering events and evaluating events
  - syntax: **assert property (** expression **)** [pass_statement] [**else** fail_statement]