SystemVerilog Quick Reference

Product Version: IUS 8.2
Release Date: May 2008

This quick reference describes the SystemVerilog constructs supported by Cadence Design Systems.

Note: Numbers in parentheses indicate the section of the *IEEE 1800 Standard for SystemVerilog*. Limitations for each construct are described in the *SystemVerilog Reference*.

Abbreviations

expr—expression args—arguments var—variable
num—number decl—declaration id—identifier

Data Types

logic [*range*] *name* [*range*]

(4.3) A 1-bit, unsigned 4-state integer.

bit [*range*] *name* [*range*]

(4.3) A 1-bit, unsigned 4-state integer.

byte *name* [*range*]

(4.3) An 8-bit, signed 2-state integer or ASCII character.

shortint *name* [*range*]

(4.3) A 16-bit, signed 2-state integer.

int *name* [*range*]

(4.3) 32-bit, signed 2-state integer.

longint *name* [*range*]

(4.3) 64-bit, signed 2-state integer.

chandle *name*;

(4.6) Stores pointers that are passed using DPI.

string *name* [= *initial_value*];

(4.7) Stores and manipulates ASCII strings.

enum *data_type* {*item1*, *item2*...} *var*

(4.9) User-defined enumerated type with a set of explicitly named values.

typedef *data_type name*;

(4.9) Declares a user-defined type.

struct [**packed** [*signing*]] { *structure members* } *name*;

(4.11) Groups variables or constraints under a single name.

union [**packed** [*signing*]] { *union members* } *name*;

(4.11) Similar to a structure, but members share the same storage space.

uwire *variable_name*;

Behaves like a single-driver wire net.

[**virtual**] **class** *name* [*port_list*] [**extends** *class_name*];
class_item
endclass [:*name*]

(4.12) Object-oriented data type that encapsulates data members and methods.

event *variable_name* [= *initial_value*];

(4.3) Can be assigned, compared, and passed as arguments.

function void *name[port_list];*
function_body
endfunction

(12.3.1) Represents non-existent data. Casting to `void` discards a function's return value.

Arrays and Queues

data_type [packed dimensions] *name* [unpacked dimensions];

(5.2) Packed dimensions are declared before the array name. Unpacked dimensions are declared after the array name.

data_type name[];

(5.6) A dynamic array is one dimension of an unpacked array, whose size can be set or changed during simulation.

data_type name [*index_type*];

(5.9) An associative array is declared using a data type as its special array size.

data_type name [$[:*constant_expr*]];

(5.14) A queue is a dynamic array that can be dynamically resized as data is ready or written.

Operators

= += -= *= /= %= &= ^= |=
<<= >>= <<<= >>>=

(8.2) Assignment operators can be used on integral types and within statements.

++ --

(8.2) Increment and decrement operators can be used on integral types.

==? !=?

(8.5) Wild equality operator (==?) treats X or Z values in the right-hand operand as "don't care". Not equal counterpart is (!=?).

== equality != inequality [] indexing
{... ; ...} concatenation <, <=, >, >= boolean comparison

(4.7) Supported operators for strings.

<<, >>, <<<, and >>> shift operators % modulus operator
/ division * multiplication

(13.4) Supported operators for constraint expressions.

Assignment Patterns

typeName'{*key*:*value*;{, *key*:*value*}}
or
'{*simple_type* | **default**:*value*; {, *key*:*value*}}

(8.13) Specifies the correspondence between a collection of expressions and the elements of an array or the members of a structure.

Procedural Statements and Control Flow

[**unique** | **priority**] **if** (*condition*) *statement*
[**else** *statement*]

(10.4) The `unique` keyword indicates that the order of the decision statements is not important and that they can be evaluated in parallel. The `priority` keyword indicates that the order of the decision statements is important, and that tools must maintain the priority encoding.

forever *statement_or_null*
repeat (*expression*) *statement_or_null*
while (expression) *statement_or_null*

(10.5) SystemVerilog enhances these Verilog loop statements by allowing null statements.

do *statement_or_null* **while** (*condition*);

(10.5.1) Similar to Verilog `while` loop, except the condition is checked after the loop executes.

for (*for_initialization* ; *expr* ; *for_step*) *statement_or_null*

(10.5.2) Enhances Verilog for by allowing loop variables and by supporting multiple initializer and step assignments.

foreach (*name* [*loop_vars*]) *statement*

(10.5.3) Iterates over the elements of an array.

return [*expression*]; | **break**; | **continue**;

(10.6) Jump statements.

final *function_statement*

(10.7) Executes when simulation ends, without delays.

begin [:*block_id*]{*block_item_declaration*}
{*statement_or_null*}
end [:*block_id*]

(10.8) Groups statements so that they execute in sequence.

always @ (*event* **iff** *expression*)

(10.10) `iff` adds conditional qualification to an event control.

Processes

always_comb *procedural_statement*;

(11.2) Describes combinational logic.

always_latch *procedural_statement*;

(11.3) Describes latched logic.

always_ff *procedural_statement*;

(11.4) Describes registered logic.

fork [:*block_id*]{*block_item_declaration*}
{*statement_or_null*} **join** | **join_any** | **join_none** [:*block_id*]

(11.6) `fork...join` completes when all spawned processes finish. `fork...join_any` completes when any of the processes finish. `join_none` completes a `fork...join` block immediately.

wait fork;

(11.8.1) Stops execution until all spawned processes finish.

disable fork;

(11.8.2) Disables all active threads of a calling process

Tasks and Functions

task [**automatic** | **static**] [*interfaceOrclassId*] *task_id*
[(*ports*)];
task_declarations
statements_or_null
endtask[:*task_id*]

(12.1) Tasks can be used to hold blocks of statements or to execute a command sequence.

function [**automatic** | **static**] [**signed** | **unsigned**]
[*rangeOrtype*] [*interfaceOrclassid*] *function_id* [(*ports*)];
function_declarations
statements_or_null
endfunction[:*function_id*]

(12.3) Groups statements together. Defines new logical or mathematical functions.

void' (*some_function*());

(12.3.2) Discards a function's return values.

subroutine (**ref** *type argument*);

(12.4.2) Passes tasks and functions by reference.

subroutine (**ref** *type argument*);

(12.4.3) Specifies default argument values for tasks and functions.

Random Constraints

rand | **randc** <*property*>;

(13.3) Specifies that a property is either a random variable (`randc`) or a random-cyclic (`randc`) variable.

[**static**] **constraint** *name* {*constraint_block*}

(13.4) Declares a constraint

Constraint Blocks

expr **inside** {*set*};

(13.4.3) Specifies a set of legal values for a given variable.

expr **dist** {*value_range* **:=** | **:/** *dist_weight*, ...}

(13.4.4) Specifies a set of weighted values.

expr **->** *constraint_set*

(13.4.5) Constrains values when a condition is successful.

if (*expr*) *constraint_set* [**else** *constraint_set*]

(13.4.6) Constrains values when a condition is met.

foreach (*array_id* [*loop_vars*]) *constraint_set*

(13.4.7) Uses loop variables and indexing expressions to specify iteration over elements in an array.

solve *identifier_list* **before** *identifier_list*;

(13.4.9) Defines the order in which random values should be generated.

Randomization Methods and Functions

randomize(<*variable_or_property*>**)**

(13.5.1) Randomizes variables or class properties.

value = **$urandom** [**(** *seed* **)**] **;**

(13.12.1) Generates unsigned, 32-bit random numbers.

value = **$urandom_range (***maxval*, *minval***);**

(13.12.2) Generates a random number within a specified range.

$srandom (*seed***);**

(13.12.3) Manually sets the RNG seed for subsequent calls.

get_randstate();

(13.12.4) Gets current state of an object's RNG.

set_randstate(*state***);**

(13.12.5) Sets the state of an object's RNG.

randcase
expression : *statement_or_null*;
{*expression* : *statement_or_null*;}
endcase

(13.15) Case statement that randomly selects one of its branches, based on a branch weight.

randsequence ([*production_id***])**
production { *production* }
endsequence

(13.16) Defines a rules for generating a random sequence.

randsequence Production Statements

if (*expression***)** *true_productionitem*
[**else** *false_productionitem*]

(13.16.2) Specifies a conditional production

case (*expression***)**
expression {,*expression*}**:** *production_item1***;**
expression {,*expression*}**:** *production_item2***;**
...
default: *default_production***;**
endcase

(13.6.3) Selects a production from a set of alternatives.

repeat (*expression***)** *production_item*

(13.6.4) Generates a production a set number of times.

Interprocess Synchronization and Communication

semaphore *name* [**=new(***N***)];**

(14.2) Built-in class used for synchronization and the mutual exclusion of resources.

mailbox [**#(**<*type*>**)**] *name* [**=new()];**

(14.3) Class-based FIFO structure; allows procedures to safely exchange data.

->> [*delay_or_event_control*] *hierarchical_event_id***;**

(14.5.2) Non-blocking event trigger operator.

*hierarchical_event_id***.triggered**

(14.5.4) Specifies that an event persist throughout the time step in which it was triggered.

Clocking Blocks

[**default**] **clocking** *name* **@(***clocking_event***);**
default *default_skew***;** |
*clocking_direction list_of_clocking_decl_assign***;**
endclocking[**:** *name*]

(15.1) Defines a group of signals that are synchronized to a specific clock.

Program Blocks

program *name* [**(***port_list***)**];
program items
endprogram[**:** *name*]

(16.1) Similar to a module, but facilitates the creation of a testbench and has special syntax and semantic restrictions.

Packages

package *name***;**
[*timeunits_decl*] {{*attribute_instance*} *package_item*}
endpackage [**:***name*]

(19.2) Mechanism for sharing declarations among modules, interfaces, and programs.

*package_id***::***item_name* //Class scope resolution operator
import *package_id* **:: *;** // Wildcard import
import *package_id***::***item_name***;** // Explicit import

(19.2.1) Ways to reference items within a package.

Interfaces

interface *name* [**(***port_list***)**];
interface items
endinterface [**:** *name*]

(20.2) Encapsulates the communication between blocks of a digital system.

modport *name* **(***port_list***);**

(20.4) Defines the direction of ports in an interface declaration.

*interface_port_name***.***task_function_name*(*args*)**;**

(20.6) Declares tasks and functions in interfaces.

virtual [**interface**] *name interface_id***;**

(20.8) Variable that represents an interface instance.

System Tasks and Functions

$root

(19.4) Lets you refer explicitly to a top-level instance.

$bits (*expression***);**

(22.3) Returns the number of bits represented by an expression.

$left $right $low $high $increment $size $dimensions $unpacked_dimensions

(5.5) Supported array querying functions.

Compiler Directives

`define

(23.2) Text substitution macro

`begin_keywords** and **`end_keywords

(23.4) Defines reserved keywords for a block of code.

`remove_keyword** and **`restore_keyword

Removes particular keywords from any set of keywords.

Direct Programming Interface

import {**"DPI" | "DPI-C"**} [**context** | **pure**] [*c_id* **=**]
function *function_data_type function_id* **(**[*tf_port_list*]**);**

(26.4.4) Imports C function using DPI.

export {**"DPI" | "DPI-C"**} [*c_id* **=**] **function** | **task** *taskfunc_id*;

(26.6) Exports a SystemVerilog task or function using DPI.

String Methods

*Str***.len()**	*Str***.hextoa(***integer***)**
*Str***.getc(***int***)**	*Str***.octtoa(***integer***)**
*Str***.toupper()**	*Str***.bintoa(***integer***)**
*Str***.tolower()**	*Str***.compare(***Str2***)**
*Str***.itoa(***integer***)**	*Str***.icompare(***string***)**
*Str***.realtoa(***real***)**	*Str***.atoreal()**
*Str***.atoi()**	*Str***.substr(***intA***,** *intB***)**
*Str***.atobin()**	*Str***.atohex()**
*Str***.atooct()**	*Str***.putc(***int***,** *byte***)**

Enumeration Type Methods

*enum***.first**	*enum***.prev**
*enum***.last**	*enum***.num**
*enum***.next**	*enum***.name**

Array Locator Methods for Queues

*queue_id***.***method* **(***arguments***) with (***expression***)**

(5.15) Syntax for using array locator methods on queues.

*queueid***.find**	*queueid***.find_index**
*queueid***.find_first**	*queueid***.find_last**
*queueid***.find_first_index**	*queueid***.find_last_index**

(5.15.1) Supported array locator methods for queues.

Dynamic Array Methods

new[*expression***][(***expression***)]]**

*arrayid***.size**	*arrayid***.delete**

Associative Array Methods

*arrayid***.num**	*arrayid***.last(***index***)**
*arrayid***.delete[(***index***)]**	*arrayid***.next(***index***)**
*arrayid***.exists(***index***)**	*arrayid***.prev(***index***)**
*arrayid***.first(***index***)**	

Queue Methods

*queueid***.insert(***index***,** *object***)**	*queueid***.size**
*queueid***.delete(***index***)**	*queueid***.pop_front**

cadence

queueid.**push_front(***object***)**	*queueid*.**pop_back**
queueid.**push_back(***object***)**	

Semaphore Methods	
new(*keyCount***)**	**put(***keyCount***)**
get(*keyCount***)**	**try_get(***keyCount***)**

Mailbox Methods	
new(*bound***)**	*mailbox_id*.**get(***msg***)**
mailbox_id.**num()**	*mailbox_id*.**try_get(***msg***)**
mailbox_id.**put(***msg***)**	*mailbox_id*.**peek(***msg***)**
mailbox_id.**try_put(***msg***)**	*mailbox_id*.**try_peek(***msg***)**

Note: This document does not cover Assertions or Coverage.