Verilog HDL

A guide to Digital Design and Synthesis

Samir Palnitkar

SunSoft Press
1996

Part 3 Appendices

Strength Modeling and Advanced Net Definitions

A

A.1 Strength Levels

Verilog allows signals to have logic values and strength values. Logic values are `0`, `1`, `x`, and `z`. Logic strength values are used to resolve combinations of multiple signals and to represent behavior of actual hardware elements as accurately as possible. Several logic strengths are available. Table A-1 shows the strength levels for signals. Driving strengths are used for signal values that are driven on a net. Storage strengths are used to model charge storage in **`trireg`** type nets, which are discussed later in the appendix.

Table A-1 Strength Levels

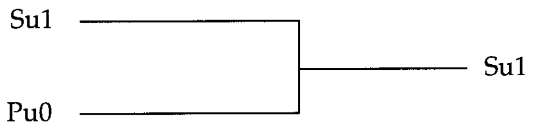
Strength Level	Abbreviation	Degree	Strength Type
supply1	Su1	strongest 1	driving
strong1	St1	↑	driving
pull1	Pu1		driving
large1	La1		storage
weak1	We1		driving
medium1	Me1		storage
small1	Sm1		storage
highz1	HiZ1	weakest 1	high impedance
highz	HiZ0	weakest 0	high impedance
small0	Sm0		storage
medium0	Me0		storage
weak0	We0		driving
large0	La0		storage
pull0	Pu0		driving
strong0	St0	↓	driving
supply0	Su0	strongest 0	driving

A.2 Signal Contention

Logic strength values can be used to resolve signal contention on nets that have multiple drivers.There are many rules applicable to resolution of contention. However, two cases of interest that are most commonly used are described below.

A.2.1 Multiple Signals with Same Value and Different Strength

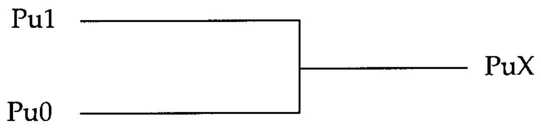
If two signals with same known value and different strength drive the same net, the signal with the higher strength wins.



In the example shown, **supply** strength is greater than **pull**. Hence, *Su1* wins.

A.2.2 Multiple Signals with Opposite Value and Same Strength

When two signals with opposite value and same strength combine, the resulting value is **x**.



A.3 Advanced Net Types

We discussed resolution of signal contention by using strength levels. There are other methods to resolve contention without using strength levels. Verilog provides advanced net declarations to model logic contention.

A.3.1 tri

The keywords **wire** and **tri** have identical syntax and function. However, separate names are provided to indicate the purpose of the net. Keyword **wire** denotes nets with single drivers, and **tri** is denotes nets that have multiple drivers. A multiplexer, as defined below, uses the **tri** declaration.

```
module mux(out, a, b, control);
output out;
input a, b, control;
tri out;
wire a, b, control;

bufif0 b1(out, a, control); //drives a when control = 0; z otherwise
bufif1 b2(out, b, control); //drives b when control = 1; z otherwise

endmodule
```

The net is driven by *b1* and *b2* in a complementary manner. When *b1* drives *a, b2* is tristated; when *b2* drives *b, b1* is tristated. Thus, there is no logic contention. If there is contention on a **tri** net, it is resolved by using strength levels. If there are two signals of opposite values and same strength, the resulting value of the **tri** net is **x**.

A.3.2 trireg

Keyword **trireg** is used to model nets having capacitance that stores values. The default strength for **trireg** nets is **medium**. Nets of type **trireg** are in one of the two states:

- *Driven state*—At least one driver drives a **0**, **1**, or **x** value on the net. The value is continuously stored in the **trireg** net. It takes the strength of the driver.
- *Capacitive state.* All drivers on the net have high impedance (**z**) value. The net holds the last driven value The strength is **small**, **medium**, or **large** (default is **medium**).

```
trireg (large) out;
wire a, control;

bufif1 (out, a, control); // net out gets value of a when control = 1;
                          //when control = 0, out retains last value of a
                          //instead of going to z. strength is large.
```

A.3.3 tri0 and tri1

Keywords **tri0** and **tri1** are used to model resistive **pulldown** and **pullup** devices. A **tri0** net has a value **0** if nothing is driving the net. Similarly, **tri1** net has a value **1** if nothing is driving the net. The default strength is **pull**.

```
tri0 out;
wire a, control;

bufif1 (out, a, control); //net out gets the value of a when control = 1;
                          //when control = 0,out gets the value 0 instead
                          //of z. If out were declared as tri1, the
                          //default value of out would be 1 instead of 0.
```

A.3.4 supply0 and supply1

Keyword **supply1** is used to model a power supply. Keyword **supply0** is used to model ground. Nets declared as **supply1** or **supply0** have constant logic value and a strength level **supply** (strongest strength level).

```
supply1 vcc;   //all nets connected to vcc are connected to power supply
supply0 gnd; //all nets connected to gnd are connected to ground
```

A.3.5 wor, wand, trior, and triand

When there is logic contention, if we simply use a **tri** net, we will get an **x**. This could be indicative of a design problem. However, sometimes the designer needs to resolve the final logic value when there are multiple drivers on the net, without using strength levels. Keywords **wor**, **wand**, **trior**, and **triand** are used to resolve such conflicts. Nets **wand** perform the *and* operation on multiple driver logic values. If any value is **0**, the value of the net **wand** is **0**. Net **wor** performs the *or* operation on multiple driver values. If any value is **1**, the net **wor** is **1**. Nets **triand** and **trior** have the same syntax and function as the nets **wor** and **wand**. The example below explains the function.

```
wand out1;
wor out2;

buf (out1, 1'b0);
buf (out1, 1'b1); //out1 is a wand net; gets the final value 1'b0

buf (out2, 1'b0);
buf (out2, 1'b1); //out2 is a wor net; gets the final value 1'b1
```

List of PLI Routines B

A list of PLI **acc_** and **tf_** routines is provided. VPI routines are not listed. Names, argument list, and a brief description of the routine are shown for each PLI routine. For details regarding the use of each PLI routine, refer to the *IEEE Language Reference Manual.*

B.1 Conventions

Conventions to be used for arguments are shown below.

Convention	Meaning
char *format	Pass formatted string
char *	Pass name of object as a string
<u>underlined arguments</u>	Arguments are optional
*	Pointer to the data type
....	More arguments of the same type

B.2 Access Routines

Access routines are classified into five categories: handle, next, value change link, fetch, and modify routines.

B.2.1 Handle Routines

Handle routines return handles to objects in the design. The name of handle routines always starts with the prefix **acc_handle_**.

Table B-1 Handle Routines

Return Type	Name	Argument List	Description
handle	acc_handle_by_name	(char *name, handle scope)	Object from name relative to scope.
handle	acc_handle_condition	(handle object)	Conditional expression for module path or timing check handle.
handle	acc_handle_conn	(handle terminal);	Get net connected to a primitive, module path, or timing check terminal.
handle	acc_handle_datapath	(handle modpath);	Get a handle to data path for an edge-sensitive module path.
handle	acc_handle_hiconn	(handle port);	Get hierarchically higher net connection to a module port.
handle	acc_handle _interactive_scope	( );	Get the handle to the current simulation interactive scope.
handle	acc_handle_loconn	(handle port);	Get hierarchically lower net connection to a module port.
handle	acc_handle_modpath	(handle module, char *src, char *dest); or (handle module, handle src, handle dest);	Get handle to module path whose source and destination are specified. Module path can be specified by names or handles.
handle	acc_handle_notifier	(handle tchk);	Get notifier register associated with a particular timing check.
handle	acc_handle_object	(char *name);	Get handle for any object, given its full or relative hierarchical path name.
handle	acc_handle_parent	(handle object);	Get handle for own primitive or containing module or an object.
handle	acc_handle_path	(handle outport, handle inport);	Get handle to path from output port of a module to input port of another module.
handle	acc_handle_pathin	(handle modpath);	Get handle for first net connected to the input of a module path.
handle	acc_handle_pathout	(handle modpath);	Get handle for first net connected to the output of a module path.

Table B-1 Handle Routines (Continued)

Return Type	Name	Argument List	Description
handle	acc_handle_port	(handle module, int port#);	Get handle for module port. Port# is the position from the left in the module definition (starting with 0).
handle	acc_handle_scope	(handle object);	Get the handle to the scope containing an object.
handle	acc_handle _simulated_net	(handle collapsed_net_handle);	Get the handle to the net associated with a collapsed net.
handle	acc_handle_tchk	(handle module, int tchk_type, char *netname1, int edge1, ...);	Get handle for a specified timing check of a module or cell.
handle	acc_handle_tchkarg1	(handle tchk);	Get net connected to the first argument of a timing check.
handle	acc_handle_tchkarg2	(handle tchk);	Get net connected to the second argument of a timing check.
handle	acc_handle_terminal	(handle primitive, int terminal#);	Get handle for a primitive terminal. Terminal# is the position in the argument list.
handle	acc_handle_tfarg	(int arg#);	Get handle to argument arg# of calling system task or function that invokes the PLI routine.
handle	acc_handle_tfinst	( );	Get the handle to the current user defined system task or function.

B.2.2 Next Routines

Next routines return the handle to the next object in the linked list of a given object type in a design. Next routines always start with the prefix **acc_next_** and accept reference objects as arguments. Reference objects are shown with a prefix *current_*.

Table B-2 Next Routines

Return Type	Name	Argument List	Description
handle	acc_next	(int obj_type_array[], handle module, handle current_object);	Get next object of a certain type within a scope. Object types such as accNet or accRegister are defined in obj_type_array.

Table B-2 Next Routines (Continued)

Return Type	Name	Argument List	Description
handle	acc_next_bit	(handle vector, handle current_bit);	Get next bit in a vector port or array.
handle	acc_next_cell	(handle module, handle current_cell);	Gets next cell instance in a module. Cells are defined in a library.
handle	acc_next_cell_load	(handle net, handle current_cell_load);	Get next cell load on a net.
handle	acc_next_child	(handle module, handle current_child);	Get next module instance appearing in this module
handle	acc_next_driver	(handle net, handle current_driver_terminal);	Get next primitive terminal driver that drives the net.
handle	acc_next_hiconn	(handle port, handle current_net);	Get next higher net connection.
handle	acc_next_input	(handle path_or_tchk, handle current_terminal);	Get next input terminal of a specified module path or timing check.
handle	acc_next_load	(handle net, handle current_load);	Get next primitive terminal driven by a net independent of hierarchy.
handle	acc_next_loconn	(handle port, handle current_net);	Get next lower net connection to a module port.
handle	acc_next_modpath	(handle module, handle path);	Get next path within a module.
handle	acc_next_net	(handle module, handle current_net);	Get the next net in a module.
handle	acc_next output	(handle path, handle current_terminal);	Get next output terminal of a module path or data path.
handle	acc_next_parameter	(handle module, handle current_parameter);	Get next parameter in a module.
handle	acc_next_port	(handle module, handle current_port);	Get the next port in a module port list.
handle	acc_next_portout	(handle module, handle current_port);	Get next output or inout port of a module.
handle	acc_next_primitive	(handle module, handle current_primitive);	Get next primitive in a module.
handle	acc_next_scope	(handle scope, handle current_scope);	Get next hierarchy scope within a certain scope.
handle	acc_next_specparam	(handle module, handle current_specparam);	Get next specparam declared in a module.

Table B-2 Next Routines (Continued)

Return Type	Name	Argument List	Description
handle	acc_next_tchk	(handle module, handle current_tchk);	Get next timing check in a module.
handle	acc_next_terminal	(handle primitive, handle current_terminal);	Get next terminal of a primitive.
handle	acc_next_topmod	(handle current_topmod);	Get next top level module in the design.

B.2.3 Value Change Link (VCL) Routines

VCL routines allow the user system task to add and delete objects from the list of objects that are monitored for value changes. VCL routines always begin with the prefix **acc_vcl_** and do not return a value.

Table B-3 Value Change Link Routines

Return Type	Name	Argument List	Description
void	acc_vcl_add	(handle object, int (*consumer_routine) (), char *user_data, int VCL_flags);	Tell the Verilog simulator to call the consumer routine with value change information whenever the value of an object changes.
void	acc_vcl_delete	(handle object, int (*consumer_routine) (), char *user_data, int VCL_flags);	Tell the Verilog simulator to stop calling the consumer routine when the value of an object changes.

B.2.4 Fetch Routines

Fetch routines can extract a variety of information about objects. Information such as full hierarchical path name, relative name, and other attributes can be obtained. Fetch routines always start with the prefix **acc_fetch_**.

Table B-4 Fetch Routines

Return Type	Name	Argument List	Description
int	acc_fetch_argc	( );	Get the number of invocation command-line arguments.
char **	acc_fetch_argv	( );	Get the array of invocation command-line arguments.
double	acc_fetch_attribute	(handle object, char *attribute, double default);	Get the attribute of a parameter or specparam.

Table B-4 Fetch Routines (Continued)

Return Type	Name	Argument List	Description
char *	acc_fetch_defname	(handle object);	Get the defining name of a module or a primitive instance.
int	acc_fetch_delay_mode	(handle module);	Get delay mode of a module instance.
bool	acc_fetch_delays	(handle object, double *rise, double *fall, double *turnoff); (handle object, double *d1, *d2, *d3, *d4 *d5, *d6);	Get typical delay values for primitives, module paths, timing checks, or module input ports.
int	acc_fetch_direction	(handle object);	Get the direction of a port or terminal, i.e., input, output, or inout.
int	acc_fetch_edge	(handle path_or_tchk_term);	Get the edge specifier type of a path input or output terminal or a timing check input terminal.
char *	acc_fetch_fullname	(handle object);	Get the full hierarchical name of any name object or module path.
int	acc_fetch_fulltype	(handle object);	Get the type of the object. Return a predefined integer constant that tells type.
int	acc_fetch_index	(handle port_or_terminal);	Get the index for a port or terminal for gate, switch, UDP instance, module, etc. Zero returned for the first terminal.
void	acc_fetch_location	(p_location loc_p, handle object);	Get the location of an object in a Verilog source file. p_location is a predefined data structure that has file name and line number in the file.
char *	acc_fetch_name	(handle object);	Get instance of object or module path within a module.
int	acc_fetch_paramtype	(handle parameter);	Get the data type of parameter, integer, string, real, etc.
double	acc_fetch_paramval	(handle parameter);	Get value of parameter or specparam. Must cast return values to integer, string, or double.
int	acc_fetch_polarity	(handle path);	Get polarity of a path.

Table B-4 Fetch Routines (Continued)

Return Type	Name	Argument List	Description
int	acc_fetch_precision	( );	Get the simulation time precision.
bool	acc_fetch_pulsere	(handle path, double *r1, double *e1,double *r2, double *e2.........)	Get pulse control values for module paths based on reject values and e_values for transitions.
int	acc_fetch_range	(handle vector, int *msb, int *lsb);	Get the most significant bit and least significant bit range values of a vector.
int	acc_fetch_size	(handle object);	Get number of bits in a net, register, or port.
double	acc_fetch_tfarg	(int arg#);	Get value of system task or function argument indexed by arg#.
int	acc_fetch_tfarg_int	(int arg#);	Get integer value of system task or function argument indexed by arg#.
char *	acc_fetch_tfarg_str	(int arg#);	Get string value of system task or function argument indexed by arg#.
void	acc_fetch _timescale_info	(handle object, p_timescale_info timescale_p);	Get the time scale information for an object. p_timescale_info is a pointer to a predefined time scale data structure.
int	acc_fetch_type	(handle object);	Get the type of object. Return a predefined integer constant such as accIntegerVar, accModule, etc.
char *	acc_fetch_type_str	(handle object);	Get the type of object in string format. Return a string of type accIntegerVar, accParameter, etc.
char *	acc_fetch_value	(handle object, char *format);	Get the logic or strength value of a net, register, or variable in the specified format.

B.2.5 Utility Access Routines

Utility access routines perform miscellaneous operations related to access routines.

Table B-5 Utility Access Routines

Return Type	Name	Argument List	Description
void	acc_close	( );	Free internal memory used by access routines and reset all configuration parameters to default values.
handle *	acc_collect	(handle *next_routine, handle ref_object, int *count);	Collect all objects related to a particular reference object by successive calls to an acc_next routine. Return an array of handles.
bool	acc_compare_handles	(handle object1, handle object2);	Return true if both handles refer to the same object.
void	acc_configure	(int config_param, char *config_value);	Set parameters that control the operation of various access routines.
int	acc_count	(handle *next_routine, handle ref_object);	Count the number of objects in a reference object such as a module. The objects are counted by successive calls to the acc_next routine
void	acc_free	(handle *object_handles);	Free memory allocated by acc_collect for storing object handles.
void	acc_initialize	( );	Reset all access routine configuration parameters. Call when entering a user-defined PLI routine.
bool	acc_object_in_typelist	(handle object, int object_types[]);	Match the object type or property against an array of listed types or properties.
bool	acc_object_of_type	(handle object, int object_type);	Match the object type or property against a specific type or property.
int	acc_product_type	( );	Get the type of software product being used.
char *	acc_product_version	( );	Get the version of software product being used.

Table B-5 Utility Access Routines (Continued)

Return Type	Name	Argument List	Description
int	acc_release_object	(handle object);	Deallocate memory associated with an input or output terminal path.
void	acc_reset_buffer	( );	Reset the string buffer.
handle	acc_set _interactive_scope	( );	Set the interactive scope of a software implementation.
void	acc_set_scope	(handle module, char *module_name);	Set the scope for searching for objects in a design hierarchy.
char *	acc_version	( );	Get the version of access routines being used.

B.2.6 Modify Routines

Modify routines can modify internal data structures.

Table B-6 Modify Routines

Return Type	Name	Argument List	Description
void	acc_append_delays	(handle object, double rise, double fall, double z); or (handle object, double d1, ..., double d6); or (handle object, double limit); or (handle object double delay[]);	Add delays to existing delay values for primitives, module paths, timing checks, or module input ports. Can specify rise/fall/turn-off or 6 delay or timing check or min:typ:max format.
bool	acc_append_pulsere	(handle path, double r1, ..., double r12, double e1, ..., double e12);	Add to the existing pulse control values of a module path.
void	acc_replace_delays	(handle object, double rise, double fall, double z); or (handle object, double d1, ..., double d6); or (handle object, double limit); or (handle object double delay[]);	Replace delay values for primitives, module paths, timing checks, or module input ports. Can specify rise/fall/turn-off or 6 delay or timing check or min:typ:max format.
bool	acc_replace_pulsere	(handle path, double r1, ..., double r12, double e1, ..., double e12);	Set pulse control values of a module path as a percentage of path delays.
void	acc_set_pulsere	(handle path, double reject, double e);	Set pulse control percentages for a module path.
void	acc_set_value	(handle object, p_setval_value value_P, p_setval_delay delay_P);	Set value for a register or a sequential UDP.

B.3 Utility (tf_) Routines

Utility (tf_) routines are used to pass data in both directions across the Verilog/user C routine boundary. All the tf_ routines assume that operations are being performed on current instances. Each tf_ routine has a tf_i counterpart in which the instance pointer where the operations take place has to be passed as an additional argument at the end of the argument list.

B.3.1 Get Calling Task/Function Information

Table B-7 Get Calling Task/Function Information

Return Type	Name	Argument List	Description
char *	tf_getinstance	( );	Get the pointer to the current instance of the simulation task or function that called the user's PLI application program.
char *	tf_mipname	( );	Get the Verilog hierarchical path name of the simulation module containing the call to the user's PI application program.
char *	tf_ispname	( )	Get the Verilog hierarchical path name of the scope containing the call to the user's PLI application program.

B.3.2 Get Argument List Information

Table B-8 Get Argument List Information

Return Type	Name	Argument List	Description
int	tf_nump	( );	Get the number of parameters in the argument list.
int	tf_typep	(int param_index#);	Get the type of a particular parameter in the argument list.
int	tf_sizep	(int param_index#);	Get the length of a parameter in bits.
t_tfexprinfo *	tf_expinfo	(int param_index#, struct t_tfexprinfo *exprinfo_p);	Get information about a parameter expression.
t_tfexprinfo *	tf_nodeinfo	(int param_index#, struct t_tfexprinfo *exprinfo_p);	Get information about a node value parameter.

B.3.3 Get Parameter Values

Table B-9 Get Parameter Values

Return Type	Name	Argument List	Description
int	tf_getp	(int param_index#);	Get the value of parameter in integer form.
double	tf_getrealp	(int param_index#);	Get the value of a parameter in double-precision floating-point form.
int	tf_getlongp	(int *aof_highvalue, int para_index#);	Get parameter value in long 64-bit integer form.
char *	tf_strgetp	(int param_index#, char format_character);	Get parameter value as a formatted character string.
char *	tf_getcstringp	(int param_index#);	Get parameter value as a C character string.
void	tf_evaluatep	(int param_index#);	Evaluate a parameter expression and get the result.

B.3.4 Put Parameter Value

Table B-10 Put Parameter Values

Return Type	Name	Argument List	Description
void	tf_putp	(int param_index#, int value);	Pass back an integer value to the calling task or function.
void	tf_putrealp	(int param_index#, double value;	Pass back a double-precision floating-point value to the calling task or function.
void	tf_putlongp	(int param_index#, int lowvalue, int highvalue);	Pass back a double-precision 64-bit integer value to the calling task or function.
void	tf_propagatep	(int param_index#);	Propagate a node parameter value.
int	tf_strdelputp	(int param_index#, int bitlength, char format_char, int delay, int delaytype, char *value_p);	Pass back a value and schedule an event on the parameter. The value is expressed as a formatted character string, and the delay, as an integer value.
int	tf_strrealdelputp	(int param_index#, int bitlength, char format_char, int delay, double delaytype, char *value_p);	Pass back a string value with an attached real delay.
int	tf_strlongdelputp	(int param_index#, int bitlength, char format_char, int lowdelay,int highdelay, int delaytype, char *value_p);	Pass back a string value with an attached long delay.

B.3.5 Monitor Parameter Value Changes

Table B-11 Monitor Parameter Value Changes

Return Type	Name	Argument List	Description
void	tf_asynchon	( );	Enable a user PLI routine to be called whenever a parameter changes value.
void	tf_asynchoff	( );	Disable asynchronous calling.
void	tf_synchronize	( );	Synchronize parameter value changes to the end of the current simulation time slot.
void	tf_rosynchronize	( );	Synchronize parameter value changes and suppress new event generation during current simulation time slot.
int	tf_getpchange	(int param_index#);	Get the number of the parameter that changed value.
int	tf_copypvc_flag	(int param_index#);	Copy a parameter value change flag.
int	tf_movepvc_flag	(int param_index#);	Save a parameter value change flag.
int	tf_testpvc_flag	(int param_index#);	Test a parameter value change flag.

B.3.6 Synchronize Tasks

Table B-12 Synchronize Tasks

Return Type	Name	Argument List	Description
int	tf_gettime	( );	Get current simulation time in integer form.
	tf_getrealtime		
int	tf_getlongtime	(int *aof_hightime);	Get current simulation time in long integer form.
char *	tf_strgettime	( );	Get current simulation time as a character string.
int	tf_getnextlongtime	(int *aof_lowtime, int *aof_hightime);	Get time of the next scheduled simulation event.
int	tf_setdelay	(int delay);	Cause user task to be reactivated at a future simulation time expressed as an integer value delay.
int	tf_setlongdelay	(int lowdelay, int highdelay);	Cause user task to be reactivated after a long integer value delay.
int	tf_setrealdelay	(double delay, char *instance);	Activate the misctf application at a particular simulation time.

Table B-12 Synchronize Tasks (Continued)

Return Type	Name	Argument List	Description
void	tf_scale_longdelay	(char *instance, int lowdelay, int hidelay, int *aof_lowtime, int *aof_hightime);	Convert a 64-bit integer delay to internal simulation time units.
void	tf_scale_realdelay	(char *instance, double delay, double *aof_realdelay);	Convert a double-precision floating-point delay to internal simulation time units.
void	tf_unscale_longdelay	(char *instance, int lowdelay, int hidelay, int *aof_lowtime, int *aof_hightime);	Convert a delay from internal simulation time units to the time scale of a particular module.
void	tf_unscale_realdelay	(char *instance, double delay, double *aof_realdelay);	Convert a delay from internal simulation time units to the time scale of a particular module.
void	tf_clearalldelays	( );	Clear all reactivation delays.
int	tf_strdelputp	(int param_index#, int bitlength, char format_char, int delay, int delaytype, char *value_p);	Pass back a value and schedule an event on the parameter. The value is expressed as a formatted character string and the delay as an integer value.
int	tf_strrealdelputp	(int param_index#, int bitlength, char format_char, int delay, double delaytype, char *value_p);	Pass back a string value with an attached real delay.
int	tf_strlongdelputp	(int param_index#, int bitlength, char format_char, int lowdelay,int highdelay, int delaytype, char *value_p);	Pass back a string value with an attached long delay.

B.3.7 Long Arithmetic

Table B-13 Long Arithmetic

Return Type	Name	Argument List	Description
void	tf_add_long	(int *aof_low1, int *aof_high1, int low2, int high2);	Add two 64-bit long values
void	tf_subtract_long	(int *aof_low1, int *aof_high1, int low2, int high2);	Subtract one long value from another.
void	tf_multiply_long	(int *aof_low1, int *aof_high1, int low2, int high2);	Multiply two long values.
void	tf_divide_long	(int *aof_low1, int *aof_high1, int low2, int high2);	Divide one long value by another.

Table B-13 Long Arithmetic (Continued)

Return Type	Name	Argument List	Description
int	tf_compare_long	(int low1, int high1, int low2, int high2);	Compare two long values.
char *	tf_longtime_tostr	(int lowtime, int hightime);	Convert a long value to a character string.
void	tf_real_to_long	(double real, int *aof_low, int *aof_high);	Convert a real number to a 64-bit integer.
void	tf_long_to_real	(int low, int high, double *aof_real);	Convert a long integer to a real number

B.3.8 Display Messages

Table B-14 Display Messages

Return Type	Name	Argument List	Description
void	io_printf	(char *format, arg1,......);	Write messages to the standard output and log file.
void	io_mcdprintf	(char *format, arg1,......);	Write messages to multiple-channel descriptor files.
void	tf_error	(char *format, arg1,......);	Print error message.
void	tf_warning	(char *format, arg1,......);	Print warning message.
void	tf_message	(int level, char facility, char code, char *message, arg1, ...);	Print error and warning messages, using the Verilog simulator's standard error handling facility.
void	tf_text	(char *format, arg1, ...);	Store error message information in a buffer. Displayed when tf_message is called.

B.3.9 Miscellaneous Utility Routines

Table B-15 Miscellaneous Utility Routines

Return Type	Name	Argument List	Description
void	tf_dostop	( );	Halt the simulation and put the system in interactive mode.
void	tf_dofinish	( );	Terminate the simulation.
char *	mc_scanplus_args	(char *startarg);	Get command line plus (+) options entered by the user in interactive mode.

Table B-15 Miscellaneous Utility Routines

Return Type	Name	Argument List	Description
void	tf_write_save	(char *blockptr, int blocklength);	Write PLI application data to a save file.
int	tf_read_restart	(char *blockptr, int block_length);	Get a block of data from a previously written save file.
void	tf_read_restore	(char *blockptr, int blocklength);	Retrieve data from a save file.
void	tf_dumpflush	( );	Dump parameter value changes to a system dump file.
char *	tf_dumpfilename	( );	Get name of system dump file.

B.3.10 Housekeeping Tasks

Table B-16 Housekeeping Tasks

Return Type	Name	Argument List	Description
void	tf_setworkarea	(char *workarea);	Save a pointer to the work area of a PLI application task/function instance.
char *	tf_getworkarea	( );	Retrieve pointer to a work area.
void	tf_setroutine	(char (*routine) ());	Store pointer to a PLI application task/function.
char *	tf_getroutine	( );	Retrieve pointer to a PLI application task/function.
void	tf_settflist	(char *tflist);	Store pointer to a PLI application task/function instance.
char *	tf_gettflist	( );	Retrieve pointer to a PLI application task/function instance.

List of Keywords, System Tasks, and Compiler Directives C

C.1 Keywords

Keywords are predefined, nonescaped identifiers that define the language constructs. An escaped identifier is never treated as a keyword. All keywords are defined in lowercase. The list is sorted in alphabetical order.

always	and	assign	attribute
begin	buf	bufif0	bufif1
case	casex	casez	cmos
deassign	default	defparam	disable
edge	else	end	endattribute
endcase	endfunction	endmodule	endprimitive
endspecify	endtable	endtask	event
for	force	forever	fork
function	highz0	highz1	if
initial	inout	input	integer
join	large	macromodule	medium
module	nand	negedge	nmos
nor	not	notif0	notif1
or	output	parameter	pmos
posedge	primitive	pull0	pull1
pulldown	pullup	rcmos	real
realtime	reg	release	repeat
rnmos	rpmos	rtran	rtranif0
rtranif1	scalared	signed	small
specify	specparam	strength	strong0
strong1	supply0	supply1	table
task	time	tran	tranif0
tranif1	tri	tri0	tri1
triand	trior	trireg	unsigned
vectored	wait	wand	weak0
weak1	while	wire	wor
xnor	xor		

C.2 System Tasks and Functions

The following is a list of keywords frequently used by Verilog simulators for names of system tasks and functions. Not all system tasks and functions are explained in this book. For details, refer to *Verilog HDL Language Reference Manual.* This list is sorted in alphabetical order.

`$bitstoreal`	`$countdrivers`	`$display`	`$fclose`
`$fdisplay`	`$fmonitor`	`$fopen`	`$fstrobe`
`$fwrite`	`$finish`	`$getpattern`	`$history`
`$incsave`	`$input`	`$itor`	`$key`
`$list`	`$log`	`$monitor`	`$monitoroff`
`$monitoron`	`$nokey`		

C.3 Compiler Directives

The following is a list of keywords frequently used by Verilog simulators for specifying compiler directives. Only the most frequently used directives are discussed in the book. For details, refer to *Verilog HDL Language Reference Manual.* This list is sorted in alphabetical order.

`'accelerate`	`'autoexpand_vectornets`	`'celldefine`
`'default_nettype`	`'define`	`'define`
`'else`	`'endcelldefine`	`'endif`
`'endprotect`	`'endprotected`	`'expand_vectornets`
`'ifdef`	`'include`	`'noaccelerate`
`'noexpand_vectornets`	`'noremove_gatenames`	`'nounconnected_drive`
`'protect`	`'protected`	`'remove_gatenames`
`'remove_netnames`	`'resetall`	`'timescale`
`'unconnected_drive`		

Formal Syntax Definition D

The following items summarize the format of the formal syntax descriptions:

1. Whitespace may be used to separate lexical tokens.
2. Angle brackets surround each description item and are *not* literal symbols; that is, they do not appear in a source example of a syntax item.
3. **<name>** in lowercase is a syntax construct item defined by other syntax construct items or by lexical token items (see next item).
4. **<NAME>** in uppercase is a lexical token item. Its definition is a terminal node in the description hierarchy; that is, its definition does not contain any syntax construct items.
5. **<name>?** is an optional item.
6. **<name>*** is zero, one, or more items.
7. **<name>+** is one or more items.
8. **<name> <,<name>>*** is a comma-separated list of items with at least one item in the list.
9. **if** [condition] is a condition placed on one of several definitions.
10. **<name> ::=** gives a syntax definition to an item.
11. ||= introduces an alternative syntax definition.
12. **name** is a literal (a keyword). For example, the definition **<event_declaration> ::= event <name_of_event>** stipulates that the keyword "event" precedes the name of an event in an event declaration.
13. (. . .) places parenthesis symbols in a definition. These parentheses are literals required by the syntax being defined. Other literal symbols can also appear in a definition (for example, . and :).

In Verilog syntax, a period (.) may not be preceded or followed by a space.

D.1 Source Text

```
<source_text>
        ::= <description>*
<description>
        ::= <module>
        ||= <primitive>
<module>
        ::= module <name_of_module> <list_of_ports>? ;
            <module_item>*
            endmodule
        ||= macromodule <name_of_module> <list_of_ports>? ;
            <module_item>*
            endmodule
<name_of_module>
        ::= <IDENTIFIER>
<list_of_ports>
        ::= ( <port> <,<port>>* )
<port>
        ::= <port_expression>?
        ||= . <name_of_port> ( <port_expression>? )
<port_expression>
        ::= <port_reference>
        ||= { <port_reference> <,<port_reference>>* }
<port_reference>
        ::= <name_of_variable>
        ||= <name_of_variable> [ <constant_expression> ]
        ||= <name_of_variable> [ <constant_expression> : <constant_expression> ]
<name_of_port>
        ::= <IDENTIFIER>
<name_of_variable>
        ::= <IDENTIFIER>
<module_item>
        ::= <parameter_declaration>
        ||= <input_declaration>
        ||= <output_declaration>
        ||= <inout_declaration>
        ||= <net_declaration>
        ||= <reg_declaration>
```

||= <time_declaration>
||= <integer_declaration>
||= <real_declaration>
||= <event_declaration>
||= <gate_declaration>
||= <UDP_instantiation>
||= <module_instantiation>
||= <parameter_override>
||= <continuous_assign>
||= <specify_block>
||= <initial_statement>
||= <always_statement>
||= <task>
||= <function>

<UDP>

::= primitive <name_of_UDP> (<name_of_variable> <,<name_of_variable>>*) ;
    <UDP_declaration>+
    <UDP_initial_statement>?
    <table_definition>
    endprimitive

<name_of_UDP>

::= <IDENTIFIER>

<UDP_declaration>

::= <output_declaration>
||= <reg_declaration>
||= <input_declaration>

<UDP_initial_statement>

::= initial <output_terminal_name> = <init_val> ;

<init_val>

::= 1'b0
||= 1'b1
||= 1'bx
||= 1
||= 0

<table_definition>

::= table <table_entries> endtable

<table_entries>

::= <combinational_entry>+
||= <sequential_entry>+

<combinational_entry>
    ::= <level_input_list> : <OUTPUT_SYMBOL> ;
<sequential_entry>
    ::= <input_list> : <state> : <next_state> ;
<input_list>
    ::= <level_input_list>
    ||= <edge_input_list>
<level_input_list>
    ::= <LEVEL_SYMBOL>+
<edge_input_list>
    ::= <LEVEL_SYMBOL>* <edge> <LEVEL_SYMBOL>*
<edge>
    ::= (<LEVEL_SYMBOL> <LEVEL_SYMBOL>)
    ||= <EDGE_SYMBOL>
<state>
    ::= <LEVEL_SYMBOL>
<next_state>
    ::= <OUTPUT_SYMBOL>
    ||= - (This is a literal hyphen, see Chapter 12 for details).
<OUTPUT_SYMBOL> *is one of the following characters*:
    0 1 x X
<LEVEL_SYMBOL> *is one of the following characters:*
    0 1 x X ? b B
<EDGE_SYMBOL> *is one of the following characters:*
    r R f F p P n N *
<task>
    ::= task <name_of_task> ; <tf_declaration>*<statement_or_null> endtask
<name_of_task>
    ::= <IDENTIFIER>
<function>
    ::= function <range_or_type>? <name_of_function> ;
        <tf_declaration>+
        <statement>
        endfunction
<range_or_type>
    ::= <range>
    ||= integer
    ||= real

<name_of_function>
    ::= <IDENTIFIER>
<tf_declaration>
    ::= <parameter_declaration>
    ||= <input_declaration>
    ||= <output_declaration>
    ||= <inout_declaration>
    ||= <reg_declaration>
    ||= <time_declaration>
    ||= <integer_declaration>
    ||= <real_declaration>
    ||= <event_declaration>

D.2 Declarations

<parameter_declaration>
    ::= parameter <list_of_param_assignments> ;
<list_of_param_assignments>
    ::=<param_assignment><,<param_assignment>*
<param_assignment>
    ::=<<identifier> = <constant_expression>>
<input_declaration>
    ::= input <range>? <list_of_variables> ;
<output_declaration>
    ::= output <range>? <list_of_variables> ;
<inout_declaration>
    ::= inout <range>? <list_of_variables> ;
<net_declaration>
    ::= <NETTYPE> <expandrange>? <delay>? <list_of_variables> ;
    ||= trireg <charge_strength>? <expandrange>? <delay>? <list_of_variables> ;
<NETTYPE> *is one of the following keywords:*
    wire tri tri1 supply0 wand triand tri0 supply1 wor trior trireg
<expandrange>
    ::= <range>
    ||= scalared <range>
    ||= vectored <range>
<delay>
    ::=

<reg_declaration>
    ::= reg <range>? <list_of_register_variables> ;
<time_declaration>
    ::= time <list_of_register_variables> ;
<integer_declaration>
    ::= integer <list_of_register_variables> ;
<real_declaration>
    ::= real <list_of_variables> ;
<event_declaration>
    ::= event <name_of_event> <,<name_of_event>>* ;
<continuous_assign>
    ::= assign <drive_strength>? <delay>? <list_of_assignments> ;
    ||= <NETTYPE> <drive_strength>? <expandrange>? <delay>?
        <list_of_assignments> ;
<parameter_override>
    ::= defparam <list_of_param_assignments> ;
<list_of_variables>
    ::= <name_of_variable> <,<name_of_variable>>*
<name_of_variable>
    ::= <IDENTIFIER>
<list_of_register_variables>
    ::= <register_variable> <,<register_variable>>*
<register_variable>
    ::= <name_of_register>
    ||= <name_of_memory> [<constant_expression> : <constant_expression>]
<constant_expression>
    ::=
<name_of_register>
    ::= <IDENTIFIER>
<name_of_memory>
    ::= <IDENTIFIER>
<name_of_event>
    ::= <IDENTIFIER>
<charge_strength>
    ::= (small)
    ||= (medium)
    ||= (large)
<drive_strength>
    ::= (<STRENGTH0> , <STRENGTH1>)

||= (<STRENGTH1> , <STRENGTH0>)

<STRENGTH0> *is one of the following keywords:*

supply0 strong0 pull0 weak0 highz0

<STRENGTH1> *is one of the following keywords:*

supply1 strong1 pull1 weak1 highz1

<range>

::= [<constant_expression> : <constant_expression>]

<list_of_assignments>

::= <assignment> <,<assignment>>*

<expression>

::=

<assignment>

::=

D.3 Primitive Instances

<gate_declaration>

::= <GATETYPE> <drive_strength>? <delay>? <gate_instance>
<,<gate_instance>>* ;

<GATETYPE> *is one of the following keywords:*

and nand or nor xor xnor buf bufif0 bufif1 not notif0 notif1 pulldown pullup
nmos rnmos pmos rpmos cmos rcmos tran rtran tranif0 rtranif0 tranif1 rtranif1

<drive_strength>

::=(<STRENGTH0>,<STRENGTH1>)
||=(<STRENGTH1>,<STRENGTH0>)

<delay>

::= # <number>
||= # <identifier>
||= # (<mintypmax_expression> <,<mintypmax_expression>>?
<,<mintypmax_expression>>?)

<gate_instance>

::= <name_of_gate_instance>? (<terminal> <,<terminal>>*)

<name_of_gate_instance>

::= <IDENTIFIER>

<UDP_instantiation>

::= <name_of_UDP> <drive_strength>? <delay>? <UDP_instance>
<,<UDP_instance>>* ;

<name_of_UDP>
    ::= <IDENTIFIER>
<UDP_instance>
    ::= <name_of_UDP_instance>? (<terminal> <,<terminal>>*)
<name_of_UDP_instance>
    ::= <IDENTIFIER>
<terminal>
    ::= <expression>
    ||= <IDENTIFIER>

D.4 Module Instantiations

<module_instantiation>
    ::= <name_of_module> <parameter_value_assignment>?
        <module_instance> <,<module_instance>>* ;
<name_of_module>
    ::= <IDENTIFIER>
<parameter_value_assignment>
    ::= # (<expression> <,<expression>>*)
<module_instance>
    ::= <name_of_instance> (<list_of_module_connections>?)
<name_of_instance>
    ::= <IDENTIFIER>
<list_of_module_connections>
    ::= <module_port_connection> <,<module_port_connection>>*
    ||= <named_port_connection> <,<named_port_connection>>*
<module_port_connection>
    ::= <expression>
    ||= <NULL>
<NULL>
    ::= nothing—*this form covers the case of an empty item in a list—for example,*
        (a, b, , d)
<named_port_connection>
    ::= .< IDENTIFIER> (<expression>)
<expression>
    ::=

D.5 Behavioral Statements

<initial_statement>

```
        ::= initial <statement>
```

<always_statement>

```
        ::= always <statement>
```

<statement_or_null>

```
        ::= <statement>
        ||= ;
```

<statement>

```
        ::=<blocking assignment> ;
        ||= <non-blocking assignment> ;
        ||= if ( <expression> ) <statement_or_null>
        ||= if ( <expression> ) <statement_or_null>
            else <statement_or_null>
        ||= case ( <expression> ) <case_item>+ endcase
        ||= casez ( <expression> ) <case_item>+ endcase
        ||= casex ( <expression> ) <case_item>+ endcase
        ||= forever <statement>
        ||= repeat ( <expression> ) <statement>
        ||= while ( <expression> ) <statement>
        ||= for ( <assignment> ; <expression> ; <assignment> )
            <statement>
        ||= <delay_control> <statement_or_null>
        ||= <event_control> <statement_or_null>
        ||= wait ( <expression> ) <statement_or_null>
        ||= -> <name_of_event> ;
        ||= <seq_block>
        ||= <par_block>
        ||= <task_enable>
        ||= <system_task_enable>
        ||= disable <name_of_task> ;
        ||= disable <name_of_block> ;
        ||= force <assignment> ;
        ||= release <lvalue> ;
```

<assignment>

```
        ::= <lvalue> = <expression>
```

<blocking assignment>

::= <lvalue> = <expression>

||= <lvalue> = <delay_control> <expression> ;

||= <lvalue> = <event_control> <expression> ;

<non-blocking assignment>

::= <lvalue> <= <expression>

||= <lvalue> <= <delay_control> <expression> ;

||= <lvalue> <= <event_control> <expression> ;

<lvalue>

::=

<expression>

::=

<case_item>

::= <expression> <,<expression>>* : <statement_or_null>

||= default : <statement_or_null>

||= default <statement_or_null>

<seq_block>

::= begin <statement>* end

||= begin : <name_of_block> <block_declaration>* <statement>* end

<par_block>

::= fork <statement>* join

||= fork : <name_of_block> <block_declaration>* <statement>* join

<name_of_block>

::= <IDENTIFIER>

<block_declaration>

::= <parameter_declaration>

||= <reg_declaration>

||= <integer_declaration>

||= <real_declaration>

||= <time_declaration>

||= <event_declaration>

<task_enable>

::= <name_of_task> ;

||= <name_of_task> (<expression> <,<expression>>*) ;

<system_task_enable>

::= <name_of_system_task> ;

||= <name_of_system_task> (<expression> <,<expression>>*) ;

<name_of_system_task>
::= $<SYSTEM_IDENTIFIER>
Please note: The $ may not be followed by a space.

<SYSTEM_IDENTIFIER>
::= An <IDENTIFIER> assigned to an existing system task or function.

D.6 Specify Section

<specify_block>
::= specify <specify_item>* endspecify

<specify_item>
::= <specparam_declaration>
||= <path_declaration>
||= <level_sensitive_path_declaration>
||= <edge_sensitive_path_declaration>
||= <system_timing_check>
||= <sdpd>

<specparam_declaration>
::= specparam <list_of_param_assignments> ;

<list_of_param_assignments>
::=<param_assignment><,<param_assignment>>*

<param_assignment>
::=<<identifier>=<constant_expression>>

<path_declaration>
::= <path_description> = <path_delay_value> ;

<path_description>
::= (<specify_input_terminal_descriptor> =>
<specify_output_terminal_descriptor>)
||= (<list_of_path_inputs> *> <list_of_path_outputs>)

<list_of_path_inputs>
::= <specify_input_terminal_descriptor> <,<specify_input_terminal_descriptor>>*

<list_of_path_outputs>
::= <specify_output_terminal_descriptor>
<,<specify_output_terminal_descriptor>>*

<specify_input_terminal_descriptor>
::= <input_identifier>
||= <input_identifier> [<constant_expression>]
||= <input_identifier> [<constant_expression> : <constant_expression>]

<specify_output_terminal_descriptor>
::= <output_identifier>
||= <output_identifier> [<constant_expression>]
||= <output_identifier> [<constant_expression> : <constant_expression>]

<input_identifier>
::= the <IDENTIFIER> of a module input or inout terminal

<output_identifier>
::= the <IDENTIFIER> of a module output or inout terminal.

<path_delay_value>
::= <path_delay_expression>
||= (<path_delay_expression>, <path_delay_expression>)
||= (<path_delay_expression>, <path_delay_expression>, <path_delay_expression>)
||= (<path_delay_expression>, <path_delay_expression>, <path_delay_expression>, <path_delay_expression>, <path_delay_expression>, <path_delay_expression>)

<path_delay_expression>
::= <constant_mintypmax_expression>

<system_timing_check>
::= $setup(<timing_check_event>, <timing_check_event>, <timing_check_limit> <,<notify_register>>?) ;
||= $hold(<timing_check_event>, <timing_check_event>, <timing_check_limit> <,<notify_register>>?) ;
||= $period(<controlled_timing_check_event>, <timing_check_limit> <,<notify_register>>?) ;
||= $width(<controlled_timing_check_event>, <timing_check_limit> <,<constant_expression>,<notify_register>>?) ;
||= $skew(<timing_check_event>, <timing_check_event>, <timing_check_limit> <,<notify_register>>?) ;
||= $recovery(<controlled_timing_check_event>, <timing_check_event>, <timing_check_limit> <,<notify_register>>?) ;
||= $setuphold(<timing_check_event>, <timing_check_event>, <timing_check_limit>, <timing_check_limit> <,<notify_register>>?) ;

<timing_check_event>
::= <timing_check_event_control>? <specify_terminal_descriptor> <&&& <timing_check_condition>>?

<specify_terminal_descriptor>
::= <specify_input_terminal_descriptor>

||=<specify_output_terminal_descriptor>

<controlled_timing_check_event>

::= <timing_check_event_control> <specify_terminal_descriptor>
 <&&& <timing_check_condition>>?

<timing_check_event_control>

::= posedge
||= negedge
||= <edge_control_specifier>

<edge_control_specifier>

::= **edge** [<edge_descriptor><,<edge_descriptor>>*]

<edge_descriptor>

::= 01
|| 10
|| 0x
|| x1
|| 1x
|| x0

<timing_check_condition>

::= <SCALAR_EXPRESSION>
||= ~<SCALAR_EXPRESSION>
||= <SCALAR_EXPRESSION> == <scalar_constant>
||= <SCALAR_EXPRESSION> === <scalar_constant>
||= <SCALAR_EXPRESSION> != <scalar_constant>
||= <SCALAR_EXPRESSION> !== <scalar_constant>

<SCALAR_EXPRESSION> *is a one bit net or a bit select of an expanded vector net.*

::= **<timing_check_limit>**

::= <expression>

<scalar_constant>

::= 1'b0
||= 1'b1
||= 1'B0
||= 1'B1

<notify_register>

::= <identifier>

<level_sensitive_path_declaration>

::= if (<conditional_port_expression>)
 (<specify_terminal_descriptor> <polarity_operator>?=>
 <specify_terminal_descriptor>) = <path_delay_value>;

||= if (<conditional_port_expression>)
(<list_of_path_inputs> <polarity_operator>? *>
<list_of_path_outputs>) = <path_delay_value>;
Please note: The following two symbols are literal symbols, not syntax description conventions:

*> =>

<conditional_port_expression>
::= <port_reference>
||= <UNARY_OPERATOR><port_reference>
||= <port_reference><BINARY_OPERATOR><port_reference>

<polarity_operator>
::= +
||= -

<edge_sensitive_path_declaration>
::=<if (<expression>)>? (<edge_identifier>?
<specify_terminal_descriptor>=>
(<specify_terminal_descriptor> <polarity_operator> ?:
<data_source_expression>)) = <path_delay_value>;
||=<if (<expression>)>? (<edge_identifier>?
<specify_terminal_descriptor> *>
(<list_of_path_outputs> <polarity_operator> ?:
<data_source_expression>)) =<path_delay_value>;

<data_source_expression>
Any expression, including constants and lists. Its width must be one bit or equal to the destination's width. If the destination is a list, the data source must be as wide as the sum of the bits of the members.

<edge_identifier>
::= posedge
||= negedge

<sdpd>
::=if(<sdpd_conditional_expression>)<path_description>=
<path_delay_value>;

<sdpd_conditional_expresssion>
::=<expression><BINARY_OPERATOR><expression>
||=<UNARY_OPERATOR><expression>

D.7 Expressions

<lvalue>

::= <identifier>
||= <identifier> [<expression>]
||= <identifier> [<constant_expression> : <constant_expression>]
||= <concatenation>

<constant_expression>

::=<expression>

<mintypmax_expression>

::= <expression>
||= <expression> : <expression> : <expression>

<expression>

::= <primary>
||= <UNARY_OPERATOR> <primary>
||= <expression> <BINARY_OPERATOR> <expression>
||= <expression> <QUESTION_MARK> <expression> : <expression>
||= <STRING>

<UNARY_OPERATOR> *is one of the following tokens:*

+ - ! ~ & ~& | ^| ^ ~^

<BINARY_OPERATOR> *is one of the following tokens:*

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

<QUESTION_MARK> *is ? (a literal question mark).*

<STRING> *is text enclosed in "" and contained on one line.*

<primary>

::= <number>
||= <identifier>
||= <identifier> [<expression>]
||= <identifier> [<constant_expression> : <constant_expression>]
||= <concatenation>
||= <multiple_concatenation>
||= <function_call>
||= (<mintypmax_expression>)

<number>

::= <DECIMAL_NUMBER>
||= <UNSIGNED_NUMBER>? <BASE> <UNSIGNED_NUMBER>
||= <DECIMAL_NUMBER>.<UNSIGNED_NUMBER>
||= <DECIMAL_NUMBER><.<UNSIGNED_NUMBER>>?
E<DECIMAL_NUMBER>

||= <DECIMAL_NUMBER><.<UNSIGNED_NUMBER>>?
e<DECIMAL_NUMBER>

Please note: Embedded spaces are illegal in Verilog numbers, but embedded underscore characters can be used for spacing in any type of number.

<DECIMAL_NUMBER>

::= A number containing a set of any of the following characters, optionally preceded by + or -

0123456789_

<UNSIGNED_NUMBER>

::= A number containing a set of any of the following characters:

0123456789_

<NUMBER>

Numbers can be specified in decimal, hexadecimal, octal or binary and may optionally start with a **+** or -. The <BASE> token controls what number digits are legal. <BASE> must be one of **d**, **h**, **o**, or **b**, for the bases **d**ecimal, **h**exadecimal, **o**ctal, and **b**inary, respectively. A number can contain any set of the following characters that is consistent with <BASE>:

0123456789abcdefABCDEFxXzZ?

<BASE> *is one of the following tokens:*

'b 'B 'o 'O 'd 'D 'h 'H

<concatenation>

::= { <expression> <,<expression>>* }

<multiple_concatenation>

::= { <expression> { <expression> <,<expression>>* } }

<function_call>

::= <name_of_function> (<expression> <,<expression>>*)

||= <name_of_system_function> (<expression> <,<expression>>*)

||= <name_of_system_function>

<name_of_function>

::= <identifier>

<name_of_system_function>

::= $<SYSTEM_IDENTIFIER>

Please note: The **$** may not be followed by a space.

<SYSTEM_IDENTIFIER>

::= An <IDENTIFIER> assigned to an existing system task or function

D.8 General

<identifier>

::= <IDENTIFIER><.<IDENTIFIER>>*

Please note: The period may not be preceded or followed by a space.

<IDENTIFIER>

An identifier is any sequence of letters, digits, dollar signs ($), and underscore (_) symbols, except that the first must be a letter or the underscore; the first character may not be a digit or $. Upper- and lowercase letters are considered to be different. Identifiers may be up to 1024 characters long. Some Verilog-based tools do not recognize identifier characters beyond the 1024th as a significant part of the identifier. Escaped identifiers start with the backslash character (\) and may include any printable ASCII character. An escaped identifier ends with white space. The leading backslash character is not considered to be part of the identifier.

<delay>

::= # <number>
||= # <identifier>
||= # (<mintypmax_expression> <,<mintypmax_expression>>?
<,<mintypmax_expression>>?)

<mintypmax_expression>

::=

<delay_control>

::= # <number>
||= # <identifier>
||= # (<mintypmax_expression>)

<event_control>

::= @ <identifier>
||= @ (<event_expression>)

<event_expression>

::= <expression>
||= posedge <SCALAR_EVENT_EXPRESSION>
||= negedge <SCALAR_EVENT_EXPRESSION>
||= <event_expression> or <event_expression>*

<SCALAR_EVENT_EXPRESSION> *is an expression that resolves to a one bit value.*

Verilog Tidbits E

Answers to common Verilog questions are provided in this appendix.

Origins of Verilog HDL

Verilog HDL originated around 1983 at Gateway Design Automation, which was then located in Acton, Massachusetts. The company was privately held at that time by Dr. Prabhu Goel, the inventor of the PODEM test generation algorithm. Verilog HDL was introduced into the EDA market in 1985 as a simulator product. Verilog HDL was designed by Phil Moorby, who was later to become the Chief Designer for *Verilog-XL* and the first Corporate Fellow at Cadence Design Systems. Gateway Design Automation grew rapidly with the success of *Verilog-XL* and was finally acquired by Cadence Design Systems, San Jose, CA in 1989.

Open Verilog International (OVI)

Verilog HDL was opened to the public by Cadence Design Systems in 1990. OVI was formed to standardize and promote Verilog HDL and related design automation products. They can be reached by email at *ovi@netcom.com*, by phone at 408-353-8899, or by regular mail at 15466 Los Gatos Boulevard, Suite 109-071, Los Gatos, CA 95032.

Interpreted, Compiled, Native Compiled Simulators

Verilog simulators come in three flavors, based on the way they perform the simulation.

Interpreted simulators read in the Verilog HDL design, create data structures in memory, and run the simulation interpretively. A compile is performed each time the simulation is run, but the compile is usually very fast. An example of an interpreted simulator is *Verilog-XL*.

Compiled code simulators read in the Verilog HDL design and convert it to equivalent C code (or some other programming language). The C code is then compiled by a standard C compiler to get the binary executable. The binary is executed to run the simulation. Compile time is usually long for compiled code simulators, but in general the execution speed is faster compared to interpreted simulators. An example of compiled code simulator is Chronologic VCS.

Native compiled code simulators read in the Verilog HDL design and convert it directly to binary code for a specific machine platform. The compilation is optimized and tuned separately for each machine platform. Of course, that means that a native compiled code simulator for a Sun workstation will not run on an HP workstation, and vice versa. Because of fine tuning, native compiled code simulators can yield significant performance benefits.

Event-Driven Simulation, Oblivious Simulation

Verilog simulators typically use an *event-driven* or an *oblivious simulation* algorithm. An event-driven algorithm processes elements in the design only when signals at the inputs of these elements change. Intelligent scheduling is required to process elements. Oblivious algorithms process all elements in the design, irrespective of changes in signals. Little or no scheduling is required to process elements.

Cycle-Based Simulation

Cycle-based simulation is useful for synchronous designs where operations happen only at active clock edges. Cycle simulators work on a cycle-by-cycle basis. Timing information between two clock edges is lost. Significant performance advantages can be obtained by using cycle simulation.

Fault Simulation

Fault simulation is used to deliberately insert *stuck-at* or *bridging* faults in the reference circuit. Then, a test pattern is applied and the output of the faulty circuit and the reference circuit are compared. The fault is said to be *detected* if the outputs mismatch. A set of test patterns is developed for testing the circuit.

Verilog Newsgroup

comp.lang.verilog is a newsgroup that provides discussion about Verilog HDL-related activities.

Verilog FTP Site

Do an anonymous ftp to *ftp.cray.com:/pub/comp.lang.verilog*. Various interesting resources, such as newsgroup archives, answers to frequently asked questions, Verilog parsers, Verilog-to-VHDL translators, Verilog modes for GNU Emacs, speedup notes, etc., are available. The README file in that directory gives complete information about available resources.

Verilog Simulators

Veriwell simulator is available from Wellspring solutions for SPARC, Macintosh, DOS, and Linux. The simulator is available via ftp at *iii.net:/pub/pub-site/wellspring/* . Use is not restricted for source files under 1000 lines.

Viper simulator is available from InterHDL for SPARC and DOS. The simulator is available via ftp at *ftp.netcom.com:/pub/el/eli* . Use is not restricted for sources files under 1000 lines.

Verilog Related Mosaic Sites

If you are using mosaic, you can find Verilog HDL-related information on a lot of WWW sites. A few interesting sites are listed below.

Cadence - http://www.cadence.com/

Cadmazing - http://www.cadmazing.com/cadmazing/pages/da.html

Chronologic Simulation - http://www.chronologic.com/index.html

EE Times - http://techweb.cmp.com/eet/

Synopsys - http://www.synopsys.com/

IVC (International Verilog Conference) - http://www.e2w3.com/ivcconf.html

Verilog Examples F

This appendix contains the source code for two examples.

- The first example is a synthesizable model of a FIFO implementation.
- The second example is a behavioral model of a 256K × 16 DRAM.

These examples are provided to give the reader a flavor of real-life Verilog HDL usage. The reader is encouraged to look through the source code to understand coding style and the usage of Verilog HDL constructs.

F.1 Synthesizable FIFO Model

This example describes a synthesizable implementation of a FIFO. The FIFO depth and FIFO width in bits can be modified by simply changing the value of two parameters, *`FWIDTH* and *`FDEPTH*. For this example, the FIFO depth is 4 and the FIFO width is 32 bits. The input/output ports of the FIFO are shown in Figure F-1.

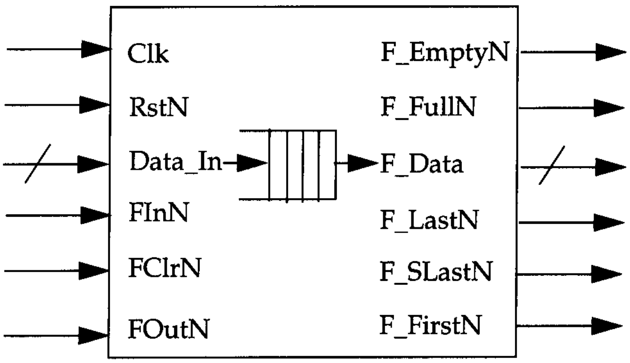


Figure F-1 FIFO Input/Output Ports

Input ports

All ports with a suffix "N" are low asserted.

Clk	Clock signal
RstN	Reset signal
Data_In	32-bit data into the FIFO
FInN	Write into FIFO Signal
FClrN	Clear Signal to FIFO
FOutN	Read from FIFO Signal

Output ports

F_Data	32-bit output data from FIFO
F_FullN	Signal indicating that FIFO is full
F_EmptyN	Signal indicating that FIFO is empty
F_LastN	Signal indicating that FIFO has space for one data value
F_SLastN	Signal indicating that FIFO has space for two data values
F_FirstN	Signal indicating that there is only one data value in FIFO

The Verilog HDL code for the FIFO implementation is shown in Example F-1.

Example F-1 Synthesizable FIFO Model

```
///////////////////////////////////////////////////////////////////////
// FileName:  "Fifo.v"
// Author   :  Venkata Ramana Kalapatapu
// Company  :  Sand Microelectronics Inc.,
// Profile :  Sand develops Simulation Models, Synthesizable Cores and
//            Performance Analysis Tools for Processors, buses and
//            memory products.  Sand's products include models for
//           industry-standard components and custom-developed models
//            for specific simulation environments.
//
//            For more information on Sand, contact us at
//            (408)-441-7138 by telephone, (408)-441-7538 by fax, or
//            email your specific needs to sales@sandmicro.com
///////////////////////////////////////////////////////////////////////


`define  FWIDTH     32            // Width of the FIFO.
`define  FDEPTH     4             // Depth of the FIFO.
```

Example F-1 *Synthesizable FIFO Model (Continued)*

```
`define  FCWIDTH    2          // Counter Width of the FIFO 2 to power
                                  // FCWIDTH = FDEPTH.

module FIFO(  Clk,
              RstN,
              Data_In,
              FClrN,
              FInN,
              FOutN,

              F_Data,
              F_FullN,
              F_LastN,
              F_SLastN,
              F_FirstN,
              F_EmptyN
           );



input                         Clk;       // CLK signal.
input                         RstN;      // Low Asserted Reset signal.
input  [(`FWIDTH-1):0]        Data_In;   // Data into FIFO.
input                         FInN;      // Write into FIFO Signal.
input                         FClrN;     // Clear signal to FIFO.
input                         FOutN;     // Read from FIFO signal.

output  [(`FWIDTH-1):0]       F_Data;    // FIFO data out.
output                       F_FullN;   // FIFO full indicating signal.
output                      F_EmptyN; // FIFO empty indicating signal.
output                        F_LastN;   // FIFO Last but one signal.
output                        F_SLastN;  // FIFO SLast but one signal.
output                        F_FirstN;  // Signal indicating only one
                                         // word in FIFO.


reg                F_FullN;
reg                F_EmptyN;
reg                F_LastN;
reg                F_SLastN;
reg                F_FirstN;

reg    [`FCWIDTH:0]      fcounter; //counter indicates num of data in FIFO
reg     [(`FCWIDTH-1):0]   rd_ptr;        // Current read pointer.
```

Example F-1 Synthesizable FIFO Model (Continued)

```
reg     [(`FCWIDTH-1):0]    wr_ptr;       // Current write pointer.
wire    [(`FWIDTH-1):0]     FIFODataOut; // Data out from FIFO MemBlk
wire    [(`FWIDTH-1):0]     FIFODataIn;  // Data into FIFO MemBlk

wire    ReadN  = FOutN;
wire    WriteN = FInN;

assign F_Data     = FIFODataOut;
assign FIFODataIn = Data_In;


    FIFO_MEM_BLK memblk(.clk(Clk),
                        .writeN(WriteN),
                        .rd_addr(rd_ptr),
                        .wr_addr(wr_ptr),
                        .data_in(FIFODataIn),
                        .data_out(FIFODataOut)
                       );



    // Control circuitry for FIFO. If reset or clr signal is asserted,
    // all the counters are set to 0. If write only the write counter
    // is incremented else if read only read counter is incremented
    // else if both, read and write counters are incremented.
    // fcounter indicates the num of items in the FIFO. Write only
    // increments the fcounter, read only decrements the counter, and
    // read && write doesn't change the counter value.
    always @(posedge Clk or negedge RstN)
    begin

       if(!RstN) begin
           fcounter    <= 0;
           rd_ptr      <= 0;
           wr_ptr      <= 0;
       end
       else begin

           if(~FClrN ) begin
               fcounter    <= 0;
               rd_ptr      <= 0;
               wr_ptr      <= 0;
           end
```

Example F-1 *Synthesizable FIFO Model (Continued)*

```
          else begin

              if(~WriteN)
                  wr_ptr <= wr_ptr + 1;

              if(~ReadN)
                  rd_ptr <= rd_ptr + 1;

              if(~WriteN && ReadN && F_FullN)
                  fcounter <= fcounter + 1;

              else if(WriteN && ~ReadN && F_EmptyN)
                  fcounter <= fcounter - 1;
         end
      end
   end



   // All the FIFO status signals depends on the value of fcounter.
   // If the fcounter is equal to afdepth, indicates FIFO is full.
   // If the fcounter is equal to zero, indicates the FIFO is empty.



   // F_EmptyN signal indicates FIFO Empty Status. By default it is
   // asserted, indicating the FIFO is empty. After the First Data is
   // put into the FIFO the signal is deasserted.
   always @(posedge Clk or negedge RstN)
   begin

      if(!RstN)
         F_EmptyN <= 1'b0;

      else begin
         if(FClrN==1'b1) begin

            if(F_EmptyN==1'b0 && WriteN==1'b0)

                F_EmptyN <= 1'b1;

            else if(F_FirstN==1'b0 && ReadN==1'b0 && WriteN==1'b1)
```

Example F-1 Synthesizable FIFO Model (Continued)

```
                    F_EmptyN <= 1'b0;
            end
            else
               F_EmptyN <= 1'b0;
         end
      end

   // F_FirstN signal indicates that there is only one datum sitting
   // in the FIFO. When the FIFO is empty and a write to FIFO occurs,
   // this signal gets asserted.
   always @(posedge Clk or negedge RstN)
   begin

      if(!RstN)

         F_FirstN <= 1'b1;

      else begin
         if(FClrN==1'b1) begin

            if((F_EmptyN==1'b0 && WriteN==1'b0) ||
               (fcounter==2 && ReadN==1'b0 && WriteN==1'b1))

                  F_FirstN <= 1'b0;

            else if (F_FirstN==1'b0 && (WriteN ^ ReadN))
                  F_FirstN <= 1'b1;
         end
         else begin

            F_FirstN <= 1'b1;
         end
      end
   end


   // F_SLastN indicates that there is space for only two data words
   //in the FIFO.
   always @(posedge Clk or negedge RstN)
   begin

      if(!RstN)
```

Example F-1 Synthesizable FIFO Model (Continued)

```
          F_SLastN <= 1'b1;

       else begin

          if(FClrN==1'b1) begin

             if( (F_LastN==1'b0 && ReadN==1'b0 && WriteN==1'b1) ||
              (fcounter == (`FDEPTH-3) && WriteN==1'b0 && ReadN==1'b1))

                 F_SLastN <= 1'b0;


             else if(F_SLastN==1'b0 && (ReadN ^ WriteN) )
                 F_SLastN <= 1'b1;

          end
          else
             F_SLastN <= 1'b1;

       end
    end

    // F_LastN indicates that there is one space for only one data
    // word in the FIFO.
    always @(posedge Clk or negedge RstN)
    begin

       if(!RstN)

          F_LastN <= 1'b1;

       else begin
          if(FClrN==1'b1) begin

             if ((F_FullN==1'b0 && ReadN==1'b0)   ||
              (fcounter == (`FDEPTH-2) && WriteN==1'b0 && ReadN==1'b1))

                 F_LastN <= 1'b0;

             else if(F_LastN==1'b0 && (ReadN ^ WriteN) )
                 F_LastN <= 1'b1;
          end
          else
```

Example F-1 *Synthesizable FIFO Model (Continued)*

```
            F_LastN <= 1'b1;
      end
   end


   // F_FullN indicates that the FIFO is full.
   always @(posedge Clk or negedge RstN)
   begin

      if(!RstN)

          F_FullN <= 1'b1;

      else begin
          if(FClrN==1'b1)  begin

              if (F_LastN==1'b0 && WriteN==1'b0 && ReadN==1'b1)

                   F_FullN <= 1'b0;

              else if(F_FullN==1'b0 && ReadN==1'b0)

                   F_FullN <= 1'b1;
          end
          else
              F_FullN <= 1'b1;

      end
   end

endmodule



///////////////////////////////////////////////////////////////////
//
//
//    Configurable memory block for fifo. The width of the mem
//    block is configured via FWIDTH. All the data into fifo is done
//    synchronous to block.
//
//    Author : Venkata Ramana Kalapatapu
//
```

Example F-1 *Synthesizable FIFO Model (Continued)*

```
////////////////////////////////////////////////////////////////////

module FIFO_MEM_BLK( clk,
                     writeN,
                     wr_addr,
                     rd_addr,
                     data_in,
                     data_out
                   );


input                    clk;        // input clk.
input  writeN;  //Write Signal to put data into fifo.
input  [(`FCWIDTH-1):0]  wr_addr;    // Write Address.
input  [(`FCWIDTH-1):0]  rd_addr;    // Read Address.
input  [(`FWIDTH-1):0]   data_in;    // DataIn in to Memory Block

output [(`FWIDTH-1):0]   data_out; //Data Out from the Memory
                                   // Block(FIFO)

wire   [(`FWIDTH-1):0] data_out;

reg    [(`FWIDTH-1):0] FIFO[0:(`FDEPTH-1)];



assign data_out  = FIFO[rd_addr];

always @(posedge clk)
begin

   if(writeN==1'b0)
      FIFO[wr_addr] <= data_in;
end

endmodule
```

F.2 Behavioral DRAM Model

This example describes a behavioral implementation of a 256K × 16 DRAM. The DRAM has 256K 16-bit memory locations. The input/output ports of the DRAM are shown in Figure F-2.

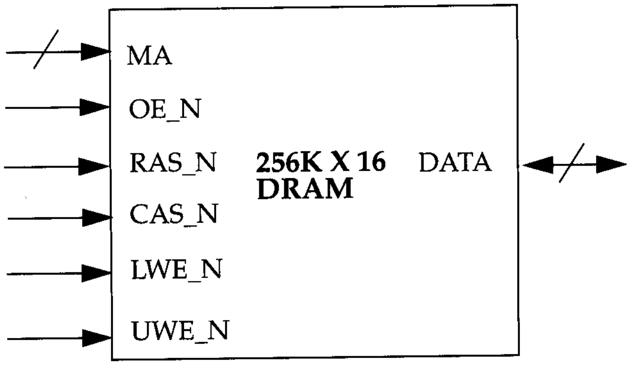


Figure F-2 DRAM Input/Output Ports

Input ports

All ports with a suffix "N" are low asserted.

MA	10-bit memory address
OE_N	Output enable for reading data
RAS_N	Row address strobe for asserting row address
CAS_N	Column address strobe for asserting column address
LWE_N	Lower write enable to write lower 8 bits of DATA into memory
UWE_N	Upper write enable to write upper 8 bits of DATA into memory

Inout ports

DATA	16-bit data as input or output. Write input if *LWE_N* or *UWE_N* is asserted. Read output if *OE_N* is asserted.

The Verilog HDL code for the DRAM implementation is shown in Example F-2.

Example F-2 *Behavioral DRAM Model*

```
//////////////////////////////////////////////////////////////////////////
// FileName:   "dram.v" - functional model of a 256K x 16 DRAM
// Author  :   Venkata Ramana Kalapatapu
// Company :   Sand Microelectronics Inc.,
// Profile :  Sand develops Simulation Models,Synthesizable Cores, and
//             Performance Analysis Tools for Processors, buses and
//             memory products. Sand's products include models for
//             industry-standard components and custom-developed
//             models for specific simulation environments.
//
//             For more information on Sand, contact us at
//            (408)-441-7138 by telephone, (408)-441-7538 by fax, or
//             email your specific needs to sales@sandmicro.com
//////////////////////////////////////////////////////////////////////////

module DRAM( DATA,
             MA,
             RAS_N,
             CAS_N,
             LWE_N,
             UWE_N,
             OE_N);

  inout [15:0]   DATA;
  input [9:0]    MA;
  input          RAS_N;
  input          CAS_N;
  input          LWE_N;
  input          UWE_N;
  input          OE_N;

  reg    [15:0]  memblk [0:262143];  // Memory Block. 256K x 16.
  reg    [9:0]   rowadd;             // RowAddress Upper 10 bits of MA.
  reg    [7:0]   coladd;             // ColAddress Lower 8 bits of MA.
  reg    [15:0]  rd_data;            // Read Data.
  reg    [15:0]  temp_reg;

  reg        hidden_ref;
  reg        last_lwe;
  reg        last_uwe;
  reg        cas_bef_ras_ref;
  reg        end_cas_bef_ras_ref;
```

Example F-2 Behavioral DRAM Model (Continued)

```
reg        last_cas;
reg        read;
reg        rmw;
reg        output_disable_check;
integer    page_mode;

assign #5 DATA=(OE_N===1'b0 && CAS_N===1'b0) ? rd_data : 16'bz;

parameter infile = "ini_file";  //Input file for preloading the Dram.

initial
begin
   $readmemh(infile, memblk);
end


always @(RAS_N)
begin

   if(RAS_N == 1'b0 ) begin
       if(CAS_N == 1'b1 ) begin
           rowadd = MA;
       end
       else
           hidden_ref = 1'b1;
   end
   else
           hidden_ref = 1'b0;
end


always @(CAS_N)
   #1 last_cas = CAS_N;


always @(CAS_N or LWE_N or UWE_N)
begin

   if(RAS_N===1'b0 && CAS_N===1'b0 ) begin

       if(last_cas==1'b1)
           coladd = MA[7:0];
```

Example F-2 *Behavioral DRAM Model (Continued)*

```
        if(LWE_N!==1'b0 && UWE_N!==1'b0)  begin  // Read Cycle.

            rd_data = memblk[{rowadd, coladd}];
            $display("READ : address = %b, Data = %b",
          {rowadd,coladd}, rd_data );
        end
        else if(LWE_N===1'b0 && UWE_N===1'b0) begin
                                           // Write Cycle both bytes.
            memblk[{rowadd,coladd}] = DATA;
            $display("WRITE: address = %b, Data = %b",
          {rowadd,coladd}, DATA );
        end
        else if(LWE_N===1'b0 && UWE_N===1'b1) begin
                                           // Lower Byte Write Cycle.

            temp_reg = memblk[{rowadd, coladd}];
            temp_reg[7:0] = DATA[7:0];
            memblk[{rowadd,coladd}] = temp_reg;
        end
        else if(LWE_N===1'b1 && UWE_N===1'b0) begin
                                           // Upper Byte Write Cycle.

            temp_reg = memblk[{rowadd, coladd}];
            temp_reg[15:8] = DATA[15:8];
            memblk[{rowadd,coladd}] = temp_reg;
        end
    end
end


// Refresh.
always @(CAS_N or RAS_N)
begin

     if(CAS_N==1'b0  && last_cas===1'b1 && RAS_N===1'b1) begin
         cas_bef_ras_ref = 1'b1;
     end

    if(CAS_N===1'b1 && RAS_N===1'b1 && cas_bef_ras_ref==1'b1) begin
         end_cas_bef_ras_ref = 1'b1;
         cas_bef_ras_ref = 1'b0;
     end
```

Example F-2 *Behavioral DRAM Model (Continued)*

```
      if( (CAS_N===1'b0 && RAS_N===1'b0) && end_cas_bef_ras_ref==1'b1
)
            end_cas_bef_ras_ref = 1'b0;

  end

endmodule
```

Bibliography

Manuals

Open Verilog International, *Verilog HDL Language Reference Manual(LRM).*

Open Verilog International, *The Programming Language Interface (PLI) Manual.*

Open Verilog International, *OVI Standard Delay File (SDF) Format Manual.*

IEEE 1364 Standard, *Verilog Hardware Description Language Reference Manual (LRM).*

Books

E. Sternheim, Rajvir Singh, Rajeev Madhavan, Yatin Trivedi, *Digital Design and Synthesis with Verilog HDL*, Automata Publishing Company, CA, 1993. ISBN 0-9627488-2-X

Donald Thomas, Phil Moorby, *The Verilog Hardware Description Language*, Kluwer Academic Publishers, MA, 1994. ISBN 0-7923-9523-9

Quick Reference Guides

Rajeev Madhavan, *Verilog HDL Reference Guide*, Automata Publishing Company, CA, 1993. ISBN 0-9627488-4-6

Stuart Sutherland, *Verilog HDL 2.0 Language Reference Guide*, Sutherland Consulting, CO, 1993.

Magazines

Integrated System Design, Verecom Group, CA.

Index