Using the New Verilog-2001 Standard

Part One: Modeling Designs

by

Stuart Sutherland

Sutherland HDL, Inc.

Portland, Oregon

©2001

Verilog® is a registered trademark of Cadence Design Systems, San Jose, CA
### About Stuart Sutherland and Sutherland HDL, Inc.

- **Sutherland HDL, Inc.** (founded 1992)
  - Provides expert Verilog HDL and PLI design services
  - Provides Verilog HDL and PLI Training
  - Located near Portland Oregon, World-wide services
- **Mr. Stuart Sutherland**
  - Over 13 years experience with Verilog
  - Worked as a design engineer on military flight simulators
  - Senior Applications Engineer for Gateway Design Automation, the founding company of Verilog
  - Author of the popular “Verilog HDL Quick Reference Guide” and “The Verilog PLI Handbook”
  - Involved in the IEEE 1364 Verilog standardization

### Seminar Objectives

- **The focus of this seminar is on understanding what is new in the Verilog-2001 standard**
  - An overview of the Verilog HDL
  - Details on the major enhancements in Verilog-2001
  - Ideas on how you can use these enhancements, *today*
- **Assumptions:**
  - You have a background in hardware engineering
  - You are at least familiar with using Verilog-1995
Part 1 covers Verilog-2001 enhancements that primarily affect **modeling hardware**
- ANSI C style port lists
- Sensitivity list enhancements
- Model attributes
- Signed data types and signed arithmetic
- Multidimensional arrays

Part 2 covers Verilog-2001 enhancements that primarily affect **verifying hardware**
- New compiler directives
- Enhanced File I/O
- Re-entrant tasks and recursive functions
- Generate blocks
- Configuration blocks
- Deep submicron timing accuracy enhancements

The IEEE Std. 1364-2001 Verilog standard is official
- Work on the standard was finished in March, 2000
- IEEE balloting on the standard was completed in July, 2000
- Clarifications to the standard as a result of ballot comments were approved in December, 2000
- The IEEE officially ratified the standard in March, 2001
Why a New Standard?

- Add enhancements to Verilog
  - Design methodologies are evolving
  - System level design, intellectual property models, design re-use, very deep submicron, etc.
  - Cliff Cummings’ “Top Five Enhancement Requests” from a survey at the HDLCon 1996 conference

- Clarify ambiguities in Verilog 1364-1995
  - Verilog-2001 more clearly defines Verilog syntax and semantics

Goals for Verilog-2001

- Enhance Verilog for
  - Higher level, abstract system level modeling
  - Intellectual Property (IP) modeling
  - Greater timing accuracy for very deep submicron

- Make Verilog even easier to use
  - Eliminate redundancies in declarations
  - Simplify syntax of some verbose constructs

- Correct errata and ambiguities

- Maintain backward compatibility

- Ensure that EDA vendors will implement all enhancements!
Overview of HDL Enhancements

- 30+ major enhancements were added to the Verilog HDL
  - Brief description and examples
  - New reserved words

- Errata and clarifications
  - Dozens of corrections were made to the 1364-1995 standard
  - Do not affect Verilog users
  - Very important to Verilog tool implementors
  - Not listed in this paper — refer to the 1364-2001 Verilog Language Reference Manual (LRM)

Support For Verilog-2001

- Several simulator and synthesis companies are working on adding support for the Verilog-2001 enhancements

- Simulators:
  - Model Technology ModelSim — currently supports most new features
  - Co-Design SystemSim — currently supports most new features
  - Synopsys VCS — planned Q3-2001 support for several new features
  - Cadence NC-Verilog and Verilog-XL — no announced release date

- Synthesis:
  - Synopsys Presto (replaces DC compiler) — currently supports a synthesizable subset of Verilog-2001 enhancements
  - Cadence BuildGates — no announced release date
  - Exemplar Leonardo Spectrum — no announced release date

Information last updated July, 2001
History of the Verilog HDL

- 1984: Gateway Design Automation introduced the Verilog-XL digital logic simulator
  - The Verilog language was part of the Verilog-XL simulator
  - The language was mostly created by 1 person, Phil Moorby
  - The language was intended to be used with only 1 product
- 1989: Gateway merged into Cadence Design Systems
- 1990: Cadence made the Verilog HDL public domain
  - Open Verilog International (OVI) controlled the language
- 1995: The IEEE standardized the Verilog HDL (IEEE 1364)
- 2001: The IEEE enhanced the Verilog HDL for modeling scalable designs, deep sub-micron accuracy, etc.

Quick Review: Contents of a Verilog Model

- Verilog **modules** are the building blocks for Verilog designs
- Modules may represent:
  - An entire design
  - Major hierarchical blocks within a design
  - Individual components within a design

```verilog
module name (ports);
  port declarations
  data type declarations
  functionality
  timing
endmodule
```

Modules are completely self contained
- The only things “global” in Verilog are the names of modules and primitives
- Verilog does not have global variables or functions
Quick Review: Module Declarations

- Module name is a user-defined name for the model
- Module ports are signals that pass in and out
- Port declarations define the direction and size of each port
- Data type declarations define signals used inside the module

Verilog-2001 Combines Port and Data Type Declarations

- The port direction and the data type of the signal can be combined into one statement
- Reduces the number of times the port name is typed
- Does not change functionality
Verilog-2001 Adds ANSI C Style Port Declarations

- The port direction and data type of the signal can be included in the port list
- Further reduces the number of times the port name is typed
- Makes Verilog more consistent with C
- Does not change functionality

module addbit (input wire a, input wire b, input wire ci, output reg sum, co);
endmodule

Quick Review: Modeling Abstraction Levels

- Functionality can be represented at various levels of “abstraction”

**Behavioral Models**
(function only)
If enable is true
for (i=0; i<=15; i=i+1)

**RTL (Register Transfer Level) Models**
(function only, with all clock timing)
At every positive edge of clock
result_register = a + b + carry

**Structural Models**
(also called Gate Level Models)
(function & structure)

Verilog-2001 has many enhancements that affect the Behavioral, RTL and Structural levels of modeling
Quick Review: Abstract Model Functionality

- Abstract functionality is represented using procedures
  - Begin with the keywords `initial` or `always`
  - Contain programming statements
  - Multiple statements are grouped with `begin` and `end`
  - Behavioral and RTL models use the same Verilog constructs

```
module addbit (input wire a, input wire b, input wire ci, output reg sum, co);

initial
begin
  sum = 0;
  co = 0;
end

always @(a or b or ci)
begin
  {co, sum} = a + b + ci;
end
endmodule
```

Quick Review: Detailed Model Functionality

- Structural functionality is modeled with a netlist
  - A netlist is a list of components and connections
  - The components may be primitive instances (this page)
  - The components may be module instances (next page)

```
module addbit (input wire a, input wire b, input wire ci, output reg sum, co);

wire n1, n2, n3;

xor g1 (n1, a, b);
xor g2 (sum, n1, ci);
and g3 (n2, a, b);
and g4 (n3, n1, ci);
or g5 (co, n2, n3);

endmodule
```
Quick Review: Using Modules as Components

- A netlist can use **module instances** as components

```
module add4 (input wire [3:0] r1, r2, ci; output wire [3:0] out; output wire co);
wire ci, c1, c2, c3, co; // internal nets
addbit i1 (r1[0], r2[0], ci, out[0], c1);
addbit i2 (r1[1], r2[1], c1, out[1], c2);
addbit i3 (r1[2], r2[2], c2, out[2], c3);
addbit i4 (r1[3], r2[3], c3, out[3], co);
endmodule
```

Quick Review: More On Verilog Procedures

- RTL models use **procedures** to represent functionality
- An *initial* procedure will execute once
  - When the procedure is completed, it is not re-executed
- An *always* procedure is an infinite loop
  - When the procedure is completed, it returns to the top and starts over
- *always* procedures model the continuous operation of hardware
  - *initial* procedures are primarily for the simulation test bench
  - Within a procedure, statements between the begin—end execute in the order they are listed
Quick Review: Controlling Verilog Procedures

- *initial* and *always* procedures may contain 3 types of timing:
  - Time based delays — the *#* token
    
    ```
    always 
    #2 sum = a + b; 
    ```
    delays execution of the next statement for a specific amount of time

  - Edge sensitive delays — the *@* token
    
    ```
    always @(posedge clock) sum <= a + b; 
    ```
    delays execution of the next statement until a change occurs on a signal

  - Level sensitive delays — the *wait* keyword
    
    ```
    always wait (enable == 1) sum = a + b; 
    ```
    delays execution of the next statement until a logic test evaluates as TRUE

Each time control delays execution of the next statement or statement group

Quick Review: Procedural Block Activation

- All procedures *automatically* become active at time zero

Time 0

```
initial begin 
  a = 0; 
  b = 0; 
  #10 a = 1; 
  ... 
end 
```

```
initial begin 
  sum = 0; 
  end 
```

```
always @(a or b) begin 
  sum = a + b; 
  end 
```

```
always @(posedge clk) begin 
  q <= sum; 
  end 
```

Note: Procedures are not like subroutines, which must be called in order to be activated
Quick Review: Procedural Sensitivity Lists

- The execution of a statement within a procedure can be controlled using an event-control sensitivity list.
- Procedures automatically become active at time zero.
- Execution of statements is delayed until a change occurs on a signal in the “sensitivity list”.

```verilog
always @(a or b or ci) begin
  sum = a + b + ci;
end
```

- `<edge>` may be `posedge` or `negedge`.
- If no edge is specified, then any transition is used.
- Sensitivity to multiple signals is specified using an “or” separated list.

A sensitivity list controls when the following statement group is executed.

Note: this is the word “or”, not a logical “OR”.

Verilog-2001 Allows Comma-Separated Sensitivity Lists

- The word `or` in an event-control sensitivity list confuses new Verilog users.
- It is the same word as the logical `or` gate primitive.
- Every other list in Verilog uses a comma (,) as a separator, instead of a word.

- Verilog-2001 allows a comma to be used in an event control sensitivity list.
- Makes Verilog easier to use.
- Does not change functionality.

```verilog
always @(a, b, ci) begin
  sum = a + b + ci;
end
```
Quick Review: RTL Modeling Definition

- Register Transfer Level modeling defines the registers in a design, and the combinational logic on the register inputs.
- The values to be stored in registers must be defined for every clock cycle.

Guidelines:
- All module outputs should be registered.
- Only use one clock per module.

Quick Review: Combinational vs. Sequential Logic

- In combinational logic, the output is a direct reflection of the input values.

- In sequential logic, the output is a reflection of an internally stored value.
Quick Review: Modeling Sequential Logic

- RTL models of sequential logic are represented with an always procedure
- The procedures “triggers” on a specific clock edge

```verilog
module dff32 (q, d, clock, reset);
//port and data type declarations
always @(posedge clock, negedge reset)
begin
if (!reset) //active low reset
q <= 32'b0;
else
q <= d;
end
endmodule
```

The sensitivity list triggers on the active edges of the clock or any other asynchronous inputs

Quick Review: Modeling Combinational Logic

- An `always` procedure must re-evaluate the outputs whenever an “input” changes value
- An “input” is any signal used to determine the value of assignments

```verilog
module mux (y, a, b, sel);
output y;
input a, b, sel;
wire a, b, sel;
reg y;
always
if (sel == 1'b0)
y = a;
else
y = b;
endmodule
```

How can this be modeled so that y is re-evaluated every time an input changes?
Verilog-2001 Adds
A Combinational Logic Sensitivity

- Verilog-2001 adds a “wildcard” token to indicate a combinational logic sensitivity list
- The @* token is a time control which indicates that the control is automatically sensitive to any change on any “input” to the statement or group-of-statements that follows
- An “input” is any signal whose value is read by the statement or statement group

Verilog-1995
```
always @(sel or a or b or c or d)
case (sel)
  2'b00: y = a;
  2'b01: y = b;
  2'b10: y = c;
  2'b11: y = d;
endcase
```

Verilog-2001
```
always @*
case (sel)
  2'b00: y = a;
  2'b01: y = b;
  2'b10: y = c;
  2'b11: y = d;
endcase
```

Quick Review:
Synthesis Pragmas

- Synthesis has **full case** and **parallel case** commands
- Full case informs the synthesis tool that it is logically impossible for undefined cases to occur
- Parallel case informs the synthesis tool that the case items do not need to be evaluated in the sequence listed
- Synopsys and other synthesis companies embed their commands in Verilog comments (called “pragmas”)

```
// FSM with one-hot encoding
always @(state)
case (state) //synopsys parallel_case synopsys full_case
  3'b001: next_state = 3'b010;
  3'b010: next_state = 3'b100;
  3'b100: next_state = 3'b001;
endcase
```
New Verilog-2001 Standard
Part 1: Modeling Hardware

by Sutherland HDL, Inc., Portland, Oregon, © 2001

Verilog-2001 Adds Attributes

Sutherland
HD L

Verilog-2001 adds “attribute” properties

- A standard means to specify tool specific information within Verilog models
- Adds new tokens (* and *)
- Eliminates need to hide commands in comments

Verilog-1995
```verilog
case (state) /* synopsys full_case */
```

Verilog-2001
```verilog
(* rtl_synthesis, full_case *) case (state)
```

- The Verilog-2001 standard does not define any attributes
- Software vendors can define proprietary attributes
- Other standards might define standard attributes
  - For example, the Verilog Synthesis Interoperability Group is proposing a standard set of synthesis attributes

Quick Review:
Integer Numbers

Sutherland
HD L

- Numbers can be simple, **unsized decimal** values
  - Default to “at least” 32 bit **signed** values

<table>
<thead>
<tr>
<th>Example</th>
<th>Binary</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>00...001</td>
<td>unsized 32-bit decimal value</td>
</tr>
</tbody>
</table>

- Numbers can be **sized, based** values: `<size>`'`<base>`'`<value>`
  - `<size>` (optional) is the number of bits (defaults to at least 32 bits)
  - `<base>` is `b`, `B`, `o`, `O`, `d`, `D`, `h`, `H` (binary, octal, decimal, hex)
  - `<value>` is 0-9 a-f A-F x X z Z _
    - A ? in a value is the same as Z (high impedance)
    - An underscore (_) is ignored — used for readability
  - Values with a radix are **unsigned** values

<table>
<thead>
<tr>
<th>Example</th>
<th>Binary</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>8'hCA</td>
<td>11001010</td>
<td>sized hex</td>
</tr>
<tr>
<td>'hf</td>
<td>0...0111</td>
<td>unsized hex</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Example</th>
<th>Binary</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>8'hF?</td>
<td>1111ZZZZ</td>
<td>sized hex</td>
</tr>
<tr>
<td>6'b01_0011</td>
<td>010011</td>
<td>sized binary</td>
</tr>
</tbody>
</table>

**Note:** Decimal numbers cannot not use values of X, Z or ?
Quick Review: Integer Numbers — continued

- Verilog adjusts the <value> to match the specified <size>
  - When <size> is fewer bits than <value>, the left-most bits of value are truncated
  - When <size> is more bits than <value>, the left-most bits are filled, based on the value of the left-most bit of <value>
    - If the left-most bit is 0 or 1, the value is filled with 0
    - If the left-most bit is Z, the value is filled with Z
    - If the left-most bit is X, the value is filled with X

<table>
<thead>
<tr>
<th>Example</th>
<th>Binary</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2'hCA</td>
<td>10</td>
<td>truncated</td>
</tr>
<tr>
<td>6'hA</td>
<td>001010</td>
<td>0 filled</td>
</tr>
<tr>
<td>64'bz</td>
<td>ZZ...ZZZZ</td>
<td>Z filled</td>
</tr>
</tbody>
</table>

Verilog-2001 adds to these rules on left-extending a value (see next page)

Verilog-2001 Adds Signed, Based Integer Numbers

- In Verilog-1995, an integer number with a base specified was always treated as an unsigned value
- Verilog-2001 adds an optional sign specifier before the base: <size><s><base><value>
  - Affects left-extension when size is more bits than value
    - If unsigned, then left extend with 0 if left-most bit is 0 or 1 (fill with Z if left-most bit is Z, and X if left-most bit is X)
    - If signed, then sign-extend (fill with value of left-most bit)
  - Affects math operations
  - Affects assignment statements

<table>
<thead>
<tr>
<th>Example</th>
<th>Binary</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>6'hA</td>
<td>001010</td>
<td>0 filled</td>
</tr>
<tr>
<td>6'shA</td>
<td>111010</td>
<td>sign extended</td>
</tr>
</tbody>
</table>
Quick Review:
Module Port Declarations

- Module ports may be declared as:
  - input, output or inout (bi-directional)
  - Modules may have any number of ports

```verbatim
<port_direction> <list_of_identifiers>;
<port_direction> [msb:lsb] <list_of_identifiers>;
```

```
module sram(address, rw, ce, data, parity);
  input [11:0] address;
  input rw, ce;
  inout [7:0] data;
  output parity;
```

As previously shown, Verilog-2001 also supports ANSI C style port declarations.

Quick Review:
Net Data Types

- Net data types represent structural connections in a design
- Each net type has resolution functionality that is used to model different types of connections (CMOS, ECL, etc.)

<table>
<thead>
<tr>
<th>Net Data Type</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>wire or tri</td>
<td>Interconnecting wire (models CMOS)</td>
</tr>
<tr>
<td>wor or trior</td>
<td>Multiple drivers OR together (models ECL)</td>
</tr>
<tr>
<td>wand or triand</td>
<td>Multiple drivers AND together (models Open-Collector)</td>
</tr>
<tr>
<td>tri0</td>
<td>Net pulls down when not driven (pull strength)</td>
</tr>
<tr>
<td>tril</td>
<td>Net pulls up when not driven (pull strength)</td>
</tr>
<tr>
<td>supply0</td>
<td>Net has a constant logic 0 (supply strength)</td>
</tr>
<tr>
<td>supply1</td>
<td>Net has a constant logic 1 (supply strength)</td>
</tr>
<tr>
<td>trireg</td>
<td>Stores last value when not driven (models capacitance)</td>
</tr>
</tbody>
</table>

In Verilog-1995, all net data types are unsigned (the most-significant bit is not a sign bit).
Quick Review: Verilog Variable Data Types

- Variable data types are used in procedures
- Variables are assigned values in *initial* & *always* procedures
- Verilog variables do not infer hardware registers!

<table>
<thead>
<tr>
<th>Variable Data Type</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>reg</td>
<td>Unsigned variable or any bit size</td>
</tr>
<tr>
<td>integer</td>
<td>Signed 32-bit variable</td>
</tr>
<tr>
<td>time</td>
<td>Unsigned 64-bit variable</td>
</tr>
<tr>
<td>real</td>
<td>Double precision floating point variable</td>
</tr>
<tr>
<td>event</td>
<td>Momentary flag with no value or storage</td>
</tr>
</tbody>
</table>

The Verilog-1995 (and earlier) standards referred to these data types as “registers”
Verilog-2001 refers to these data types as “variables” to avoid confusing the data types with hardware flip-flops

Quick Review: Data Type Declarations

- Net type declaration syntax:
  - `<net_type> <list_of_identifiers>;`
  - `<net_type> [msb:lsb] <list_of_identifiers>;`
  - `reg <list_of_identifiers>;`
  - `reg [msb:lsb] <list_of_identifiers>;`
  - `<variable_type> <list_of_identifiers>;`

```verilog
wire a, b, ci; //three scalar (1-bit) wires
wire [31:0] busA, busB; //two 32-bit buses — little endian
wire [0:31] busC; //one 32-bit bus — big endian
reg [15:0] busD; //one 16 bit unsigned variable
integer i, j, k; //three integer variables (32-bit)
```
Verilog-2001 Adds Variable Declaration With Initialization

- Verilog-2001 permits initializing variables at the time they are declared
- The initialization is executed in time-step zero, just like initial procedures

Verilog-1995: `reg clock;
initial
  clk = 0;`

Verilog-2001: `reg clock = 0;`

Initialization assignments occur in time 0, and can be executed in any order along with other time 0 events.

Verilog-2001 Adds Signed Ports, Reg and Net Data Types

- In Verilog-1995, the reg variable, all net data types and module ports were always treated as unsigned
- Verilog-2001 allows reg and net types as well as module ports to be declared as signed
- Affects math operations
- Affects assignment statements

module add4 (
  input wire signed [63:0] r1, r2,
  input wire ci;
  output reg signed [63:0] out;
  output reg co);
...
endmodule

When ports and data types are declared separately (Verilog-1995 style) then if either the port or the data type is declared as signed, the other will inherit the sign property.
Quick Review: Implicit Net Declarations

- An undeclared signal will default to a *net* data type, *if*:
  - It is connected to a primitive instance or module instance
  - It is on the left-hand side of a continuous assignment *and* it is also a port of that module
- The implicit data type is *wire*
  - Can be changed with the `default_nettype` compiler directive
- The default net size is determined by context
  - Port size if the implicit net is also a port of the module
  - Scalar if the implicit net is an internal signal

```verilog
module cpu (data, address, reset);
output [7:0] data; // 8-bit port
data_prom (data, enable, ...);
...
```

Quick Review: Continuous Assignment Statements

- A *continuous assignment* is a special concurrent process that continuously evaluates and updates a net data type
  ```verilog
  assign #<delay> <net> = <expression>;
  ```
- Declared outside of initial or always procedures
- Automatically becomes active at simulation time zero
- May use operators, but *not* programming statements

```verilog
module adder64 (sum, a, b, ci);
// declarations
wire [63:0] sum;
assign sum = a + b + ci;
endmodule
```

What if sum had not been declared?

- With Verilog-1995, sum would be an implicit net *IF* it is also a port, otherwise the assignment would be an error
- sum is continuously assigned the value of `a + b + ci`
Verilog-2001 Adds Default Nets with Continuous Assigns

- In Verilog-1995, the left-hand side must be explicitly declared, if not connected to a port of the module.
- Verilog-2001 will default to a net data type on the left-hand side of any continuous assignment.
- The left-hand side net will default to 1-bit wide, if not connected to a port of the module.

```
module mult32 (y, a, b);
output [63:0] y;
input [31:0] a, b;
assign y = a * b; //defaults to wire, width of port y
assign eq = (a == b); //ERROR: 'n' not declared
endmodule
```

Verilog-1995

```
module mult32 (y, a, b);
output [63:0] y;
input [31:0] a, b;
assign y = a * b; //defaults to 1-bit wire
endmodule
```

Verilog-2001

Verilog-2001 Can Disable Default Net Declarations

- Verilog-2001 provides a means to disable default net declarations.

```
`default_nettype none

module bad_chip (output wire [7:0] o1, input wire [7:0] n0, input wire [7:0] n1);
assign o1 = n0 & n1; //there are 3 typo's in this line!
endmodule
```

- Verilog-1995 would compile without an error, requiring debugging in simulation.
- Verilog-2001 will report an error on the undefined signals.
Quick Review: Vector Bit and Part Selects

- Bits or parts of a vector are selected with a bit index or bit range
  - A *bit select* references a discrete bit within a vector
  - A *part select* references consecutive bits within a vector
- In Verilog-1995, part selects had to use constant expressions for the left and right indices

```verilog
given: wire [15:0] data;
data //selects the entire vector
data[7] //selects one bit out of the vector
data[15:8] //selects eight bits out of the vector
```

```verilog
data[i] //variable bit selects are legal
data[i+8:i] //ILLEGAL; variable part selects not allowed
```

Verilog-2001 Adds Variable Vector Part Selects

- Verilog-2001 adds the capability to use variables to select a group of bits from a vector
  - The starting point of the part-select can vary
  - The width of the part-select remains constant

```verilog
reg [63:0] word;
reg [3:0] byte_num; //a value from 0 to 7
wire [7:0] byteN = word[byte_num*8 +: 8];
```

The starting point of the part-select is variable
The width of the part-select is constant

+: indicates the part-select increases from the starting point
-: indicates the part-select decreases from the starting point
Quick Review: Arrays of Variables

- In Verilog-1995, 1-dimensional arrays of variable data types may be declared

\[
<\text{variable}_\text{type}> \ [<\text{vector}_\text{size}>] \ <\text{identifier}> \ [\text{first}_\text{addr} : \text{last}_\text{addr}];
\]

- A entire word in the array is selected using an address index

\[
data = \text{RAM}[100]; \quad // \text{read value in address 100 of array}
\]

\[
\text{RAM}[\text{addr}_\text{bus}] = \text{data}; \quad // \text{write to an address in RAM}
\]

- Bit and part selects from an array are not allowed in Verilog-1995

Verilog-2001 Adds Multi-dimensional Arrays

- Verilog-2001 adds:
  - Multidimensional arrays of any variable data type
  - Multidimensional arrays of any net data type
Verilog-2001 Adds Array Bit and Part Selects

- Verilog-2001 allows:
  - Bit-selects out of an array
  - Part-selects out of an array

```verilog
// select the high-order byte of one word in a 2-dimensional array of 32-bit reg variables
reg [31:0] array2 [0:255][0:15];
wire [7:0] out2 = array2[100][7][31:24];
```

Quick Review: Verilog Assignment Rules

- Assignments are made from right to left, with the LSB of the right-hand side assigned to the LSB of the left-hand side

  ```
given: reg [3:0] a, y; and a is 4'b0010
y = a; will transfer the value of a, working from right to left
0 0 1 0 = 0 0 1 0
```

- If the right-hand side width is different than the left-hand side:
  - If the LHS is smaller, the left-most bits are truncated
  - In Verilog-1995, if the LHS is larger, the left-most bits are always filled with zero

  ```
given: reg [3:0] a, b;
Example    Stores    Notes
a = 8'hF1   4'b0010   the left-most bits are truncated
b = 2'b11   4'b0011   the left-most bits are zero filled
```
Using the New Verilog-2001 Standard
Part 1: Modeling Hardware
by Sutherland HDL, Inc., Portland, Oregon, © 2001

Verilog-2001
Automatic Width Extension Past 32 bits

- In Verilog-1995, Verilog assignments zero fill when the left-hand side is wider than the right-hand side
- The widths of the RHS must be hard-coded for correct results

```verilog
reg [63:0] data;
integer i = -3; //32-bit wide signed negative value
```

Verilog-1995

```verilog
data = i; //fills with 'h00000000fffffffffd
data = 'bz; //fills with 'h0000000000000000
```

Verilog-2001

```verilog
data = i; //fills with 'hfffffffffffffffd
data = 'bz; //fills with 'hzzzzzzzzzzzzzzzz
```

- Verilog-2001 will:
  - Sign-extend signed data types to the width of the left-hand side
  - Extend a logic Z or X to the width of the left-hand side

```verilog
data = i; //fills with 'h00000000fffffffffd
data = 'bz; //fills with 'h0000000000000000
```

Quick Review:
Verilog Operator Tokens

- Bit wise and shift operators operate on each bit of a vector
  - ~ & | ^ ~^ ~^ >> << (example: (a_bus & b_bus))
- Unary reduction operators collapse a vector to a 1-bit result
  - & ~& | ~| ^ ^~ ^~ (example: parity = ^data)
- Logical and relational operators perform true/false tests
  - ! && || < <= > => == != === !== (examples: (a <= b) (a && b) (!a))
- Mathematical operators perform calculations
  - + - * / % (example: a + b)
- The conditional operator performs a true/false test
  - ? : (example: out = (enable==1)? in : 8'bz;)
- The concatenate operators join signals together
  - { } {{ }} (example: {a_bus,b_bus})

Refer to the Verilog Quick Reference Guide for a description of each operator
Verilog-2001 Adds
A Power Operator

- Verilog-2001 adds an exponential power operator
  - Represented by the ** token
  - Similar to the C pow() function
  - If either operand is real, a real value is returned
  - If both operands are integers, an integer value is returned

```verilog
module ram (...);
    parameter WORD_SIZE = 64;
    parameter ADDR_SIZE = 24;

    reg [WORD_SIZE-1:0] core [0:(2**ADDR_SIZE)-1];
    ...
```

Verilog-2001 Adds
Arithmetic Shift Operators

- Verilog-2001 adds arithmetic shift operators
  - The >>> token does an arithmetic shift right, filling with the value of the sign bit
    - Different than the >> bit shift right operator, which always fills with zero
  - The <<<< token does an arithmetic shift left, filling with zeros
    - Same functionality as the << bit shift left operator

```verilog
Given: in = 8'b11001010;
assign out = in >> 3;  //bit shift right results in 'b00011001
assign out = in >>> 3; //arithmetic shift right results in 'b11111001
```
Quick Review: Arithmetic Operator Special Rules

- Arithmetic operations are based on the data type of the operands
- Floating point arithmetic is used if either operand is real
- Integer arithmetic is used if both operands are integer
  - Signed arithmetic is used if both operands are signed
  - Unsigned arithmetic is used if either operand is unsigned

The rules for wire and reg data types match how logic gates would perform the operation

Examples:

```
reg [3:0] m, n; integer i; real r;
m = 5  n = 2  i = -2  r = -2.0
m / n  is
m / i  is
5 / i  is
m /2.0  is
5 / r  is
```

Verilog-2001 Adds Sign “Casting” System Functions

- Verilog-2001 adds several signed arithmetic enhancements
  - Signed net, reg and port declarations
  - Vectors of any size can be signed, instead of just the 32-bit integer variable
  - Signed function returns
  - New $signed() and $unsigned() system functions can “cast” a value to signed or unsigned

The rules for operations do not change — Verilog-2001 just gives more signed operands

Examples:

```
reg [3:0] m;
integer i;
m = 5  i = -2
m / i  is
$signed(m) / i  is
```
Congratulations! This concludes Part 1 of the workshop “Using the New Verilog-2001 Standard”.

Where can I learn even more?

- If you are a design engineer, we recommend: “Comprehensive Verilog HDL for Design Engineers”
  - By Sutherland HDL, Inc. — www.sutherland-hdl.com
  - Our 4-day workshop covers the entire Verilog language, including the new Verilog-2001 features, with lots of labs

- If you are a verification engineer, we recommend: “Advanced Verilog PLI Training”
  - By Sutherland HDL, Inc. — www.sutherland-hdl.com
  - A 4-day workshop on customizing and extending Verilog simulators by linking in C-based models, test routines, etc.

Additional Resources: Verilog & Synthesis Books

- www.verilog-2001.com
  - Information about the Verilog-2001 standard
  - Stuart Sutherland—easy place for keywords, syntax, etc.

- IEEE Std 1364-2001
  - IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language

- The Verilog Hardware Description Language
  - Donald Thomas & Phil Moorby—good Verilog introduction
  - The Verilog PLI Handbook
  - Stuart Sutherland—using the PLI to extend the Verilog HDL

Check www.sutherland-hdl.com for a list of over 30 books.
Additional Resources:

**Verilog & Synthesis Resources**

- www.sutherland-hdl.com
  - Stuart Sutherland's web site — lots of Verilog web links
- comp.lang.verilog newsgroup
  - Great place to get quick answers to Verilog questions
  - Other newsgroups: comp.lang.vhdl, comp.cad.synthesis, comp.arch.fpga
- ESNUG - E-mail Synopsys Users Group
  - John Cooley — jcooley@world.std.com
- Verification Guild — Verilog/VHDL verification newsletter
  - Janick Bergeron's newsletter on design verification — www.janick.bergeron.com

---

**Verilog & Synthesis Conferences**

- HDLCon — International HDL Conference
  - Formerly IVC/VIUF (International Verilog Conference / VHDL International Users Forum)
  - Good conference for information about Verilog/VHDL software tools
  - www.hdlcon.org
- SNUG — Synopsys Users Group Conference
  - The best technical conference on Verilog/VHDL design methodologies and synthesis
  - www.synopsys.com (click on the SNUG link)