



# Choosing Between Verilog Data Types

<b>Module</b>	<b>4</b>
Revision	1.0
Version	1.0

Estimated Time:

- Lecture
- Lab

This module examines the Verilog value set and data types. It explores nets, variables, and a form of constant called a parameter.

# Module Objective

In this module, you:

- Choose and use the Verilog data types correctly.

## Topics

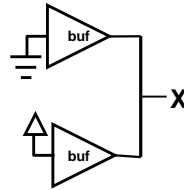
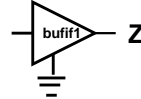
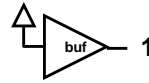
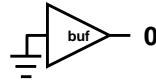
- Logic values
- Net and Register type rules and example
- Declaring vectors
  - Truncation and padding
- Defining literal values
- Declaring nets
- Declaring variables
- Declaring arrays of nets and variables
- Declaring module parameters

48

Your objective is to appropriately choose and effectively use the Verilog data types. To do that, you need to know what values can be represented, and how to represent those values using literals, constants, variables, and nets and how to create aggregates.

## Value Set

Value	Associated Informal Terms
0	Zero, Low, False, Logic Low, Ground, VSS
1	One, High, True, Logic High, Power, VDD, VCC
Z	HiZ, High Impedance, Tri-State, Undriven, Unconnected, Driver Disabled
X	Uninitialized, Unknown (bus contention)



49

The Verilog value set consists of the **four basic values**:

- 0 – To represent a logic zero, low, or false condition;
- 1 – To represent a logic one, high, or true condition;
- z – To represent a high-impedance state; and
- x – To represent an unknown logic value.

**The simulator initializes most nets to the high-impedance state.** The exception is nets of the **trireg type**, which because they represent capacitive nets, initialize to the unknown value. Upon commencing the simulation, the simulator propagates the values of net drivers onto the nets. A net that has no drivers will remain at its initialized value throughout the simulation. This situation is usually the result of user error, as there is seldom good reason to leave a net undriven.

**The simulator initializes most variables to the unknown value.** The exception is variables of the **real type**, which it initializes to 0 because it is the only type that cannot hold high-impedance or unknown values. A variable that is never assigned a value will remain at its initialized value throughout the simulation. This situation is usually the result of user error, as there is seldom good reason to declare a variable and not use it.

The appearance during simulation of a high-impedance value on a net is usually due to its drivers being disabled. In real hardware this situation either has a short duration or does not occur because bus keepers pull the net to a high or low logic state.

The appearance during simulation of an unknown value on a net is usually due to a clash between drivers driving different values. In real hardware, this situation will either not exist or have an extremely short duration.

The appearance during simulation of a high-impedance value on a variable is due to an assignment of that value to the variable, either deliberately because the variable represents one of the drivers of a bus net, or by assigning the value of a net to the variable.

The appearance during simulation of an unknown value on a variable is due to an assignment of that value to the variable, either deliberately because the simulator cannot resolve the value of the assigned expression, or by assigning the value of a net to the variable. In real hardware the variable will assume the 0 or 1 state.

# Data Types

Verilog provides three *groups* of value objects and different types in each group:

- **Nets**
  - Represent physical connection between hardware component.  
supply0, supply1, tri/wire, tri0, tri1, triand/wand, trior/wor, trireg
- **Variables**
  - Represent abstract storage storage in behavioral modeling.  
integer, real, reg, time, realtime
- **Parameters**
  - Run-time constants  
localparam, parameter, specparam

50

Procedures communicate by **passing events**, and by **passing data via nets and shared variables** informally called *signals*. Verilog does not actually have things called *signals*. Verilog has three *groups* of value objects and only a very few types in each group:

- Verilog has ***nets*** to represent physical connections between structures and objects.  
`tri/wire triand/wand trior/wor trireg ...`
- Verilog has ***variables*** to represent abstract storage elements.  
`integer real reg time realtime`
- Verilog has two simulation-time constants and an annotatable constant.  
`Localparam, parameter and specparam`  
These are constants and is illegal to modify their value at run time.

# Net and Register Type Rules

You specify the type upon declaration.

- **Nets** and **reg** are one-bit wide unless you also specify their range.
- A port declaration implicitly declares a one-bit **wire** net unless you explicitly declare it otherwise.

Rules govern your use of data types:

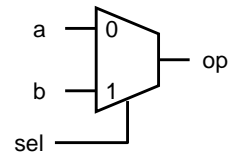
- Variables can only be driven inside procedures.
- Nets are driven everywhere else (outside procedures).
- Constants are for unchanging or instance-specific values.

```
// 1995 list of ports syntax
module mux (a, b, sel, op);
  input a, b, sel;
  output op;
  reg op;
  ...
```

Must be variable to  
assign in procedure

```
// 2001 list of port declarations
module mux (
  input a, b, sel,
  output reg op
);
  ...
```

```
always @*
  if (sel == 1)
    op = b;
  else
    op = a;
```



51

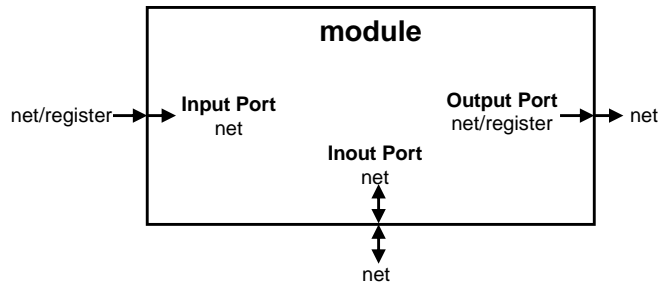
- A data item has associated with it the data values it can have and rules for how you use it.
- A net is the recipient of its drivers' values.
- A variable is an item you procedurally assign values to.
- A port is a net or variable that the instantiating module can connect its own net or variable to.
- A parameter is like a variable but has a constant value.
- Ports, nets, and variables of the reg type are a single bit unless you declare them with a range.

These module headers show two ways to declare ports:

- You can list the ports in the module header and later declare them as a module item; and
- As of the Verilog 2001 update, you can directly declare the ports in the module header.

## Type Rules in Connectivity

- Module input ports are always nets.
- Module output ports are variables if driven by a procedural block, or nets in all other cases.
- Connections to the input ports of a module instance are variables if driven by a procedural block, or nets in all other cases.
- Connections to the output ports of a module instance are always nets.
- Connections to bidirectional inout ports are always nets.



52

A port permits an external net or variable to connect to an internal net or variable, with restrictions.

Here are some common user errors, along with their typical error messages:

- You make a procedural assignment to a signal that you either declared as a net or you forgot to declare so is thus implicitly a net.  
This is an illegal assignment.
- You connect an output from an instance to a signal declared as a register. This is illegal as a module output can only drive a net.  
This is an illegal output port specification.
- You declare a signal as a module input and as a register. This is illegal as inputs can only be nets.  
These are incompatible declarations.

## Net and Register Type Example

```
module mone(output reg y_out,  
            input wire a_in, b_in);  
    always @(a_in or b_in)  
        y_out = a_in && b_in;  
endmodule
```

y\_out is reg in mone –  
assigned in a procedural block

a\_in, b\_in are wire in mone  
and mtwo – module inputs

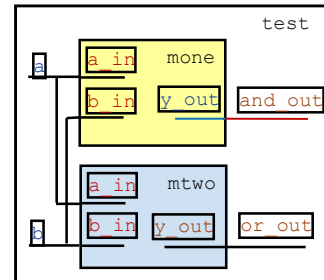
```
module mtwo(output wire y_out,  
            input wire a_in, b_in);  
    assign y_out = a_in || b_in;  
endmodule
```

y\_out is wire in mtwo – assigned  
by a continuous assignment

```
module test;  
    reg a, b;  
    wire and_out, or_out;  
    mone u1 (and_out, a, b);  
    mtwo u2 (or_out, a, b);  
    initial begin  
        a = 0;  
        b = 0;  
        ...  
    end  
endmodule
```

and\_out, or\_out are  
wire in module test  
– instance outputs

a, b are reg in module test –  
assigned in an initial  
procedural block



Here, we are using fully defined Verilog-2001 ANSI-C syntax for module port declarations for clarity.

In both module mone and mtwo, the input ports a\_in and b\_in must be declared as net data types.

In module mone, the output y\_out must be declared as a register data type as it is assigned from the always procedural block.

In module mtwo, the output y\_out must be declared as a net data type as it is driven from the continuous assign statement.

In module test, a and b must be declared as register data types as they are assigned from the initial procedural block, but and\_out and or\_out must be declared as net data types as they are driven from the instance output ports.

Therefore a connection like a -> a\_in, or y\_out -> and\_out actually changes data type as it crosses the module boundary.

## Declaring Vectors

A vector is a net or reg with a range specification.

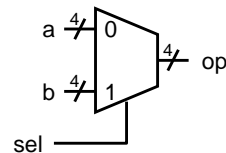
You specify the vector's range when declaring the variable:

- [msb\_constant\_expression : lsb\_constant\_expression]
- Range can be ascending or descending.
- Bounds can be negative, zero or positive.
- Bounds must be constant expressions.
- Individual bits can be selected from a vector.

```
module mux4 (  
    input wire [3:0] a, b,  
    input wire      sel,  
    output reg  [3:0] op  
);  
    4-bit vector reg  
    always @(a or b or sel)  
        if (sel == 1)  
            op = b;  
        else  
            op = a;  
endmodule
```



[width-1:0] is the de-facto standard.



Ports, nets, and variables of the reg type are a single bit unless you declare them with a range.

To declare a multiple bit port, net, or reg variable, you need to specify a range. The range provides addresses for the individual bits. The only restriction on the range bounds is that they must be constant expressions. Either or both bound can be negative, zero, or positive, and the range can be ascending or descending.



## Using Vector Ranges

Access vector selections whatever way you declared the range.

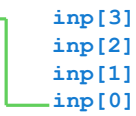
- Select one or more contiguous bits.

```
input  [3:0] inp;  
output [3:0] outp;  
assign outp = inp;
```

```
// outp[3] ← inp[3]  
// outp[2] ← inp[2]  
// outp[1] ← inp[1]  
// outp[0] ← inp[0]
```

```
input  [3:0] inp;  
output [3:0] outp;  
assign outp[3] = inp[0];
```

```
// outp[3] ← inp[3]  
// outp[2] ← inp[2]  
// outp[1] ← inp[1]  
// outp[0] ← inp[0]
```



```
reg [3:0] ireg;  
reg [0:3] oreg;  
always @(...) begin  
    oreg = ireg;
```

```
// oreg[0] ← ireg[3]  
// oreg[1] ← ireg[2]  
// oreg[2] ← ireg[1]  
// oreg[3] ← ireg[0]
```

*This page does not contain notes.*

# Assigning Between Different Widths

Vector widths do not need to match in an assignment!

- Source wider than target truncates value from msb.
- Unsigned source shorter than target zero-extends value.
- Signed source shorter than target sign-extends value.
  - Selections and concatenations are not considered signed.

```
reg [3:0] zbus; // 4 bits
reg [5:0] widebus; // 6 bits
always @(...) begin
    zbus = widebus; // same as
    zbus = widebus[3:0];

    //          ← widebus[5]
    //          ← widebus[4]
    // zbus[3] ← widebus[3]
    // zbus[2] ← widebus[2]
    // zbus[1] ← widebus[1]
    // zbus[0] ← widebus[0]
```

```
reg [3:0] zbus; // 4 bits
reg [5:0] widebus; // 6 bits
always @(...) begin
    widebus = zbus; // same as
    widebus = {2'b00,zbus};

    // widebus[5] ← 0
    // widebus[4] ← 0
    // widebus[3] ← zbus[3]
    // widebus[2] ← zbus[2]
    // widebus[1] ← zbus[1]
    // widebus[0] ← zbus[0]
```

56

You can assign between vectors of different widths:

- Verilog truncates the leftmost bits when assigning a wider vector to a narrower vector. If you want to select some range other than the rightmost bits then you must specify that range.
- Verilog pads the leftmost bits with 0 when assigning a narrower vector to a wider vector. If you want to assign something other than 0, then you need to use the concatenation operator to construct a wider expression.

# Defining Literal Values

You can specify a literal value as:

**<size>'<base><value>**

- **size** is an optional positive decimal number of bits:
  - If unsized is at least 32 bits.
- **base** is a character to indicate binary, octal, decimal, or hexadecimal radix.
  - B/b, O/o, D/d, H/h
  - Verilog-2001: Optionally preceded by an “s” character to indicate a signed value.
- **value** is legal digits for base:
  - Can include “\_” if not 1st character.
  - Can include Z/z and X/x digits if base is binary, octal, hexadecimal.
  - Verilog-2001: can be single Z/z or X/x digit if decimal base.
  - **value** is itself an unsigned number.

```
...  
reg [3:0] zbus;  
...  
zbus = 4'b1001; // 1001  
zbus = 4'o05;   // 0101  
zbus = 4'd14;   // 1110  
zbus = 4'h2f;   // 1111  
...
```



Appropriately sizing the literal avoids padding and truncation!

## More Examples

8'b1100_0001	8-bit binary
9'o017	9-bit octal
10'd1000	10-bit decimal (unsigned)
16'hff01	16-bit hexadecimal
12	32-bit decimal (signed)
'h83a	32-bit hexadecimal

You specify integer literals in binary, octal, decimal or hexadecimal format.

- You can specify a decimal integer literal as an optional sign followed by a sequence of decimal digits. A number in this format is a signed number.
- You can specify any integer literal as an optional sign followed by an optional size followed by a single quote followed by one or two base format characters and then followed by a sequence of digits appropriate to the radix.
  - A literal that you do not size has a size of at least 32 bits and most implementations make it exactly 32 bits.
  - There must be no white space between the single quote character and the base format.
  - The base format is not sensitive to character case and consists of a b, o, d or h character to indicate the radix. The Verilog 2001 update allows an optional preceding s character to indicate a signed value.
  - The digit sequence is not sensitive to character case and consists of digits appropriate for the radix. For the binary, octal or hexadecimal radices, you can use the z and x characters for any or all digits. The Verilog 2001 update also permits a decimal value to be a single z or x character to indicate a value that is all high impedance or all unknown. Except for the first digit, you can insert underscore (\_) characters anywhere you need them to improve readability. You can substitute the question mark (?) character for a z character. The reason for this will become obvious when you study the case statement.

# Automatic Extension of Unsigned Literals

For **sized literals** (e.g., 1'b1):

- Verilog pads to left with 0 to match size of wider expression.

```
reg [7:0] a;  
...  
a = 8'b11; // 00000011  
a = 8'o11; // 00001001  
a = 8'd11; // 00001011  
a = 8'h11; // 00010001  
a = 'b11; // 00000011  
a = 'o11; // 00001001  
a = 11; // 00001011
```

For **unsized literals** (e.g., 'b1):

- Size is 32 bits.
- If leftmost digit is 1 or 0:
  - Verilog left-fills with 0 to make 32 bits and pads to left with 0 to match size of wider expression.
- If leftmost digit is Z or X:
  - Verilog left-fills with that leftmost digit to make 32 bits.
  - Verilog-1995 then pads to left with 0 to match size of wider expression.
  - Verilog-2001 then pads to left with that leftmost digit to match size of wider expression.

```
reg [11:0] b;  
...  
b = 12'hzzz; // zzzzzzzzzzzz  
b = 12'hz; // zzzzzzzzzzzz  
b = 12'h0z; // 00000000zzzz  
b = 12'hz0; // zzzzzzzz0000
```

The Verilog 2001 standard permits signed based literals. The sign bit can be 0,1, z or x. Verilog 2001 sign-extends signed based literals to match the width of the enclosing expression.

Verilog zero-extends unsigned based literals except in one situation. If the provided value has fewer digits than the size indicates and the leftmost bit is z or x, then Verilog extends the provided value with that z or x bit value up to the size of the literal. This made it easy to fill an entire vector with z or x. The standard guarantees that the size of an unsized literal is at least 32 bits and most implementations made it exactly 32 bits. What follows is the tricky part: The Verilog 1995 standard then zero-extended that 32-bit expression to match the width of the enclosing expression. The Verilog 2001 update continues to extend the z or x up to the width of the enclosing expression.

# Variable Vector Selection

## Bit-Select

- A single bit-select index may be variable.

## Range-Select

- In Verilog 1995, range-select indices must be constant.
- In Verilog 2001, a range-select can use a variable index and a constant width:
  - Base expression (variable)
  - Width expression (constant)
  - Offset direction
    - Positive +
    - Negative -
  - Offset indicates if the width is added or subtracted from the base.

```
reg [7:0] vect;  
reg [1:0] slice;  
reg [2:0] index;  
initial begin  
    vect = 8'b10011001;  
    index = 4;  
    slice[0] = vect[index];  
    ...  
end
```

```
index = 4;  
slice = vect[4:3];  
slice = vect[index:index-1];
```



Error Range select bounds cannot be variable

```
index = 4;  
slice = vect[index -: 2];
```

Verilog 2001 allows constant width from variable index

[base\_expr +: width\_expr]

[base\_expr -: width\_expr]

Individual bits can be selected from a vector using a variable or expression index.

However in extracting a range of bits from a vector, Verilog 1995 requires that both bounds of the range are constants or constant expressions. So a variable cannot be used in the extraction of a range of bits.

*identifier [msb\_constant\_expression:lsb\_constant\_expression]*

Verilog 2001 provides an alternative syntax that permits the starting point of a range select to be a variable or variable expression, as long as the number of bits selected is a constant. In effect, to select a constant-width range from a variable starting bit position. The constant offset can be positive or negative from the starting point.

*identifier [base\_expression +: width\_constant\_expression]*

*identifier [base\_expression -: width\_constant\_expression]*

Where:

*base\_expression* is the variable starting bit position.

*width\_constant\_expression* is a constant plus or minus offset.

# Declaring Nets

Nets behave like physical wires.  
Values are driven onto them.  
Verilog has several net types.

- Most commonly used is **wire**.

Verilog implicitly declares wires:

- Verilog-1995: Undeclared identifiers connected to instance ports.
- Verilog-2001: Also undeclared identifier driven by continuous assignment.
- Continuous assignments and instance output/inout ports drive nets.
- Change default with compiler directive.
  - ``default_nettype <nettype>`

## Net Types

```
wire, tri          // float to Z
wor, trior         // wired-or
wand, triand       // wired-and
tri0, tri1         // float to 0,1
triereg           // capacitive
supply1, supply0  // power rails
uwire             // unresolved
```

## Examples

```
wire a, b, sel; // Scalar wires
wire [31:0] w1; // Vector wire
wand c;         // Scalar wired-and
tri [15:0] busa; // Vector bus
```

```
module halfadd (
  input a, b, output sum, carry );
  // a & b are wire by default
  // drive by continuous assign
  assign sum = a ^ b;
  assign carry = a & b;
endmodule
```

60

Nets are continuously driven by their drivers. Verilog automatically propagates a new value onto a net when the drivers of the net change value. A net driver can be a continuous assignment or a module or primitive output.

Verilog has various net types for modeling design-specific and technology-specific functionality. The most common net type is a wire.

Verilog implicitly declares wire nets in some contexts in which you use a net without first declaring it. You can override this using the ``default_nettype <net_type>` compiler directive. With this directive, such undeclared nets default to the net type you specify in the compiler directive. For this directive, Verilog 2001 adds the none net type. In other words, do not allow implicit net declarations.

Net Types	Functionality
wire, tri	For standard interconnection wires (default)
supply1	For power rails only in netlists
supply0	For ground rails only in netlists
wor, trior	For multiple drivers that are Wire-ORed
wand, triand	For multiple drivers that are Wire-ANDed
triereg	For nets with capacitive storage
tri1, tri0	For nets that pull up or down when not driven
uwire	Verilog 2005 unresolved wire accepts only one driver

# Undeclared Identifier in Verilog

- A net is inferred for undeclared identifier by default in Verilog 2001.
- What's the issue?
  - *svm* is undeclared, so defaults to a single bit wire (implicit wire).
  - No compilation error/warning message.
  - *sum* output port is un-driven and so becomes value Z.
  - Mistyped identifier defaults to net/wire.
- What is the solution?
  - Check simulation carefully!
  - Use Verilog2001 `default\_nettype none which gives compilation error on *svm*.

```
module halfadd (  
    input a, b, output sum, carry );  
    // a & b are wire by default  
    // drive by continuous assign  
    assign svm = a ^ b;  
    assign carry = a & b;  
endmodule
```

Undeclared identifier  
defaults to wire.

- Along with a typo, even lower case and uppercase identifiers (user-defined name of some such as the module, wire, variable, or the name of a function) are perceived as different in Verilog. If an identifier is typed in uppercase by mistake, and is used on the left-hand side of a continuous assignment, then an implicit net declaration is inferred, and no error or warning is reported.
- No error/warning is reported if undeclared identifier is used for connecting an instance of a module (makes design debug difficult).

For the compiler directive `default\_nettype <net\_type>, Verilog 2001 adds the none net type for compiler directive. In other words, it does not allow any implicit net declarations.

## Usage of `default\_nettype none

- Implicit wires can be avoided by using the ``default_nettype none` compiler directive.
- Any undeclared identifier is a compilation error/warning when this directive is used.
- Directive can affect the compilation of other files once turned on.

```
`default_nettype none
module halfadd (
  input a, b, output sum, carry );
  // a,b,sum,carry are implicit wires
  assign svm = a ^ b;
  assign carry = a & b;
endmodule
```

Compilation warning on a, b, sum and carry

ERROR: Undeclared identifier

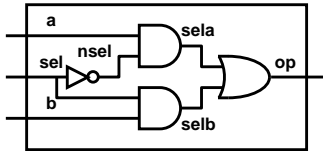
- Explicit declaration of identifier type is needed to avoid the error. This can be:  
`input wire a, b, output wire sum, carry`

When ``default_nettype none` compiler directive is used, implicit data types are disabled. This can make any undeclared identifier name a syntax error. Hence, as a limitation of this directive, benefits of implicit data types declarations are lost. Another limitation is that once this compiler directive turned on in one file, then the compilation of other files also gets affected. To avoid this, the directive ``default_nettype wire` should be added at the end of each file where implicit nets have been turned off using ``default_nettype none`.



## Merging Net Declaration and Assignment

You can merge the net declaration and assignment – known as “net declaration assignment”.



```
module mux (a, b, sel, op);
    input sel, b, a;
    output op;
    wire nsel, sela, selb;
    assign nsel = ~sel;
    assign selb = sel & b;
    assign sela = nsel & a;
    assign op = sela | selb;
endmodule
```

Bitwise OR

```
module mux (a, b, sel, op);
    input sel, b, a;
    output op;
    wire nsel = ~sel;
    wire selb = sel & b;
    wire sela = nsel & a;
    assign op = sela | selb;
endmodule
```

You can merge the net declaration with an assignment to make a net declaration assignment.

The first example explicitly declares the wires nsel, nsela and nselb and later makes continuous assignments to them.

The second example merges the wire declaration with a continuous assignment. You can still make additional continuous assignments to the nets. Remember that a net can have multiple drivers.

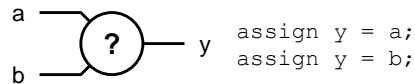
Ports that you do not declare otherwise are by default single-bit wire nets. Both examples also show a continuous assignment to the output port wire net.

# Resolving Multiple Drivers of a Net

Multiple drivers can drive a net.

- Exception: Verilog-2005 **uwire**

Only a net can resolve the value of multiple drivers.



Two names  
- same type  
wire y;  
tri y;

		b			
		0	1	z	x
a	0	0	x	0	x
	1	x	1	1	x
	z	0	1	z	x
	x	x	x	x	x

Two names  
- same type  
wand y;  
triand y;

		b			
		0	1	z	x
a	0	0	0	0	0
	1	0	1	1	x
	z	0	1	z	x
	x	0	x	x	x

Two names  
- same type  
wor y;  
trior y;

		b			
		0	1	z	x
a	0	0	1	0	x
	1	1	1	1	1
	z	0	1	z	x
	x	x	1	x	x

64

Verilog resolves the value of a net driven by multiple drivers. Assuming that the drivers all have the same strength:

- The conflict of 0 and 1 values on a wire net results in an unknown value; and
- The conflict of 0 and 1 values on a wired-logic net results in either a 0 or a 1 value.

The wire, wired-and (wand) and wired-or (wor) net types have two names. You can alternatively refer to them as tri, triand, and trior types, respectively. Net type wire and tri have identical functionality. They can be used in different names for better readability. That is, tri can be used in places where multiple drivers are present. Urban legend claims that the “tri” names exist to remind the user that they can assign high-impedance values to these nets. The urban legend neglects to explain the tri0, tri1, and trireg nets that cannot have high-impedance values.

The wired-logic nets exist to model technology-dependent logic conflict resolution:

- The wired-AND net type to model open collector logic
- The wired-OR net type to model emitter-coupled logic

Verilog 2005 added the uwire net type, an unresolved wire that allows no more than one driver.

# Declaring Variables

Variables hold values that you procedurally assign. They keep the value until you assign a different value.

- **reg** – 4-state unsigned 1-bit
  - Can declare multi-bit vector
  - Verilog-2001 adds **reg signed**
- **integer** – 4-state signed 32-bit
- **time** – 4-state unsigned 64-bit
- **real** – Double-precision float
- **realtime** – Same as real

## Examples

```
reg m, n, op; // Scalar reg
reg [31:0] rv; // Vector reg
integer i; // integer
time timeA; // time
real pi; // real
```

```
module mux (
    input a, b, sel,
    output reg op
);

always @(a, b, sel)
    if (sel == 1)
        op = b;
    else
        op = a;

endmodule
```

65

A variable retains its value until you assign a new value to it.

- The reg type holds a four-state integral value. A reg is by default unsigned but with the Verilog 2001 update you can declare it signed. Arithmetic operations treat the value of a signed reg as a signed value. A reg is by default single bit wide, but you can declare an arbitrary range for it. The reg is the most common variable type for hardware design.
- The integer type holds a four-state signed integral value. The standard guarantees it to be at least 32 bits and most implementations make it exactly 32 bits. Arithmetic operations treat the value of an integer as a signed value.
- The time type holds a four-state unsigned integral value. The standard guarantees it to be at least 64 bits and most implementations make it exactly 64 bits. A time is always unsigned. It is exactly the same as a 64-bit unsigned reg vector. People typically use variables of this type to hold simulation time values.
- The real type holds a double-precision floating-point value as described by IEEE Std 754-1985 Binary Floating-Point Arithmetic.
- The realtime type is another name for real. If you use it that way, you can use a realtime to remind yourself that its value is a real version of the simulation time.

## Making Assignments to Variables

You *read* a variable's value from inside or outside a procedure.

You *write* a variable's value only from *inside* a procedure.

Within a procedure you write *only* to variables.

```
module mux (  
    input a, b, sel,  
    output op // wire by default  
);  
  
always @(a, b, sel)  
    if (sel == 1)  
        op = b;  
    else  
        op = a;  
  
endmodule
```

ERROR: Attempt to procedurally assign a net!

```
module mux (  
    input a, b, sel,  
    output reg op  
);  
  
assign op = sel? b : a;  
  
endmodule
```

ERROR: Attempt to continuously assign a variable!

Conditional Operator

66

A variable can appear anywhere that you read its value.

Assignments to variables are procedural assignments. They exist only as a procedural statement.

The first example erroneously attempts to make a procedural assignment to a net. Remember that a port is implicitly declared as a single-bit net unless you declare it otherwise.

The second example explicitly declares the port to be a reg, but then erroneously attempts to make a continuous assignment to the variable.

# Assigning Between Integers and Vectors

## Reminder

- **reg** – 4-state unsigned any width
  - Verilog-2001 adds **reg signed**
- **integer** – 4-state signed 32 bit

Assignment between **reg** and **integer** follows previous rules:

- If value wider than target:
  - Truncates value
- If value shorter than target:
  - If *unsigned* value
    - Zero-extends
  - If *signed* value
    - Sign-extends

```
integer i;  
reg [3:0] r;  
  
initial  
begin  
    i = 17; // 00...10001  
    r = i;  //      0001  
    i = r;  // 00...00001  
    i = -3; // 11...11101  
    r = i;  //      1101  
    i = r;  // 00...01101  
end
```

Truncated

Truncated

Signage Lost



What is the result if "r" is signed?

You saw previously that Verilog truncates wide literal constants to fit shorter target widths and extends shorter literal constants to fit wider target widths, extending unsigned values with zero and signed values with the sign bit. This is exactly the case also for assignments between integers and vector reg variables.

This example assigns the value 17 to an integer variable and assigns the integer to a 4-bit unsigned vector reg variable. The assignment is a bit-for-bit replacement, so the vector reg variable gets the value 1. Upon assigning the value of the vector reg variable to the integer variable, Verilog first zero-extends the value to match the width of the integer variable, which is almost always 32 bits.

The example then assigns the value -3 to the integer variable and assigns the integer to the 4-bit unsigned vector reg variable. The assignment is a bit-for-bit replacement, so the unsigned vector reg variable gets the value 13. The signage is lost. Upon assigning the value of the vector reg variable to the integer variable, Verilog zero-extends the value to match the width of the integer variable.

## Declaring Arrays of Nets and Variables

Verilog-1995 supports one-dimensional arrays of **integer**, **reg**, and **time**.

```
integer int_a [0:99]; // array of 100 integer
reg [7:0] reg_a [0:99]; // array of 100 8-bit vectors
```

- You access one element (word) at a time by indexing to that word.
- You cannot directly take bit or part selects of a word (use an intermediate variable).

```
reg [7:0] word, array [0:255]; // memory word and array
reg bit;
...
word = array[5]; // access address 5
word = array[10]; // access address 10
bit = word[7]; // access bit 7 of extracted word
```

Verilog-2001 supports multi-dimensional arrays of **integer**, **real**, **realtime**, **reg**, **time**, and **nets**.

```
reg reg_b [0:99] [7:0]; // 100x8 array of 1-bit elements
```

- You can directly take bit and part selects of an indexed element.

```
reg [7:0] array [0:255]; // memory array
bit = array[10][7];
```

68

As with programming languages, Verilog supports arrays:

- The Verilog 1995 standard permits one-dimensional arrays of integer, time and reg. The reg elements can be either single-bit elements or vector reg elements. People refer to an array of vector reg elements as a Verilog “memory”. The Verilog 2001 standard supports arrays of integer, time, reg, real and even nets, and they can have any number of dimensions.
- The Verilog 1995 standard did not accommodate directly taking a bit-select or a part-select of an indexed memory element. You had to extract the element to a variable and then take a bit-select or part-select of the variable. The Verilog 2001 standard does accommodate directly taking a bit-select or a part-select of an indexed memory element.

# Declaring Module Parameters

A module **parameter** is an instance-specific constant:

- Parameterizes the module definition
  - Width, depth, frequency, time ...
- Can override for each individual instance

```
// Example parameter list
parameter INTEGER_P = 8,
           REAL_P    = 2.039,
           VECTOR_P  = 16'bx;
```



Uppercase constants  
improve readability!

```
// Verilog-1995 syntax
module mux (a, b, sel, op);
  parameter WIDTH = 2;
  input  [WIDTH-1:0] a, b;
  input                               sel;
  output [WIDTH-1:0] op;
  reg    [WIDTH-1:0] op;
  ...
endmodule
```

```
// Verilog-2001 alternative
module mux
#(parameter integer WIDTH = 2)
(input wire [WIDTH-1:0] a, b,
 input wire             sel,
 output reg  [WIDTH-1:0] op);
  ...
endmodule
```



Verilog-2001 allows you to specify a  
parameter type and to sign and size vector  
parameters.

Module parameters that you declare with the parameter keyword are constants that you can change for each instance of the module, thus you can use them to “parameterize” the instance, for example, to establish different widths and depths for each instance of a memory module. Parameters are not variables, so you cannot change them during the simulation run time.

The first example uses the Verilog 1995 list of ports syntax and declares the ports later as a module item. Before declaring the ports, it declares a module parameter to establish the port width and sets the parameter’s default value to 2. You can modify this parameter for each individual instance of the module.

The second example uses the Verilog 2001 list of port declarations syntax. The example needs to declare the parameter before the ports because it uses the parameter value to establish the port width. It declares the parameter using the Verilog 2001 module parameter port list syntax.

The Verilog 1995 syntax does not require a type declaration for a parameter – the parameter assumes the type of its initial value and you can change its type upon module instantiation by providing it a value of a different type. The Verilog 2001 syntax permits a type declaration for a parameter and any new value you provide it must be of that type.

# Local Parameters and Parameter Passing

Localparams are true constants. Unlike parameters, localparams cannot be overridden from the next level of hierarchy.

- Use for constants that should never be overridden upon instantiation
  - For a module that is not instantiated (testbench?)
  - For “enumerations” (FSM states?)

Verilog-2001 provides three ways to override module parameters:

- Redefinition using `defparam`
- Positional parameter override during instantiation
- Named parameter override during instantiation
  - New in Verilog-2001



Parameter override using `defparam` can be difficult to track.

Verilog-1995

```
module us_mult (a,b,product);  
  
    parameter width_a = 5;  
    parameter width_b = 5;  
    localparam op_width = width_a + width_b;  
  
    input  [width_a-1:0] a;  
    input  [width_b-1:0] b;  
    output [op_width-1:0] product;  
  
    assign product = a * b;  
  
endmodule
```

Verilog-2001

```
defparam u2.width_b = 7;  
  
us_mult #(5,7) u1 (.a (a_net),  
    .b (b_net), .product (a_b_mult));  
  
us_mult #(.width_a(5)) u1 (.a (a_net),  
    .b (b_net), .product (a_b_mult));
```

The Verilog 2001 update provides the **localparam** construct – exactly like a parameter except that you cannot change it. For a module parameter that should *not* change on a per-module basis, you should use the **localparam** keyword instead of the **parameter** keyword.

To modify the parameters of a module instance, you make a *parameter value assignment* upon instantiating the module. A parameter value assignment is a parenthesized list of parameter assignments before the module instance name and prefixed with a hash (“#”) character. You may see the hash character used in other contexts to specify a propagation delay, but in the context of a module instantiation, it is a parameter value assignment or parameter value override during instantiation.

Your list of parameter assignments can use either the Verilog 1995 ordered parameter assignment syntax or the Verilog 2001 named parameter assignment syntax, but you cannot mix the two syntaxes in the same instantiation. These syntaxes are similar to the ordered and named port connection syntax, but with the ordered parameter assignment syntax, you cannot merely insert a comma as a place holder – you need to provide values for all earlier parameters if you want to provide a value for a later parameter. This example uses the ordered parameter assignment syntax, but you should in general use the named parameter assignment syntax to make your code more readable.

You can also override a parameter value by using a **defparam** statement. A parameter can be modified using the **defparam** statement during compilation time. You can make this override from anywhere in the design by using a hierarchical parameter name. Your use of this override is superfluous when done locally and can potentially cause much confusion when not done locally as it might be difficult to track.

`localparam` cannot be directly modified by `defparam` statements. **Specparam** provides timing and delay values and can be modified through SDF annotation only. (Refer Appendix C for SDF annotation.)



## Module Summary

A net behaves like a physical wire driven by logic:

- Implicitly declared nets default to type **wire**.
- You drive nets by continuous assignments and by module and primitive outputs.

A variable stores a value:

- You update a variable only by a procedural assignment.
- You make a procedural assignment only to a variable.
- Assignments between **integer** and **reg** ignore signage.

Expressions that read nets and variables can exist inside or outside procedures.

A module port you do not declare otherwise is implicitly declared **wire**.

- Input ports are nets driven externally by nets and/or variables.
- Inout ports are nets connected externally to nets.
- Output ports are nets or variables externally driving nets.

This module examined the Verilog value set and data types. It explored nets, variables, and a form of constant called a parameter.

## Module Review

1. What are the four logic values in the Verilog value set?
2. Rewrite the binary value `8'b11010011` in hexadecimal.
3. How many bits wide is the integer type?
4. What is the primary difference between a net and a variable?

*This page does not contain notes.*

## Module Review Solutions

1. What are the four logic values in the Verilog value set?
  - 0 1 z x
2. Rewrite the binary value `8'b11010011` in hexadecimal.
  - `8'hd3`
3. How many bits wide is the integer type?
  - 32
4. What is the primary difference between a net and a variable?
  - A net is continuously driven, behaving like a physical wire driven by logic. A variable represents storage, storing a value until a new one is assigned. In the language, you can assign variables only from procedural statements and nets only from continuous assignments.

*This page does not contain notes.*

## Module Exercise

Code this two-input AND operation using a: (i) **wire** (ii) **reg**.



*This page does not contain notes.*

## Module Exercise Solution

Code this two-input AND operation using a: (i) **wire** (ii) **reg**.



Solution:

```
wire y = a & b;
```

```
reg y;  
always @* y = a & b;
```

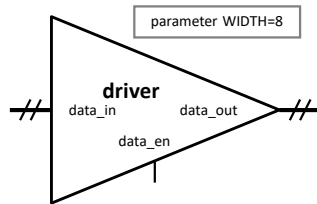
*This page does not contain notes.*

# Lab



## Lab 4-1 Modeling a Data Driver

- Use a Verilog literal value while describing a parameterized-width bus driver.



Your objective is to appropriately choose and correctly use the Verilog data types.

For this lab, you use a Verilog literal value while describing a parameterized-width bus driver.