

modules, ports, & data types in Verilog

Note Title

9/9/2012

Module 2

By Gandhi Puvvada

*Reference:*

The Verilog 2001 Reference Guide by ESPERAN

Esperan has graciously provided the .pdf file of their Verilog guide for use by USC students and faculty. This lecture includes some extracts from their Verilog guide. Hence these lecture slides are appropriate for use by USC students and faculty only.

Please do not post these slides or the video recording of the lecture anywhere outside USC and do not make this accessible to anyone outside USC.

The logo for Esperan, featuring a stylized blue 'E' with a red starburst above it, followed by the word 'esperan' in a blue, cursive script font.



1. Verilog is Case-sensitive

Keywords are all in lower-case:

```
example:
module
endmodule
if else
case for
```

2. Comments:

// single-line comment starts with

// double slash!

/* multi-line

is like in "C" comment */

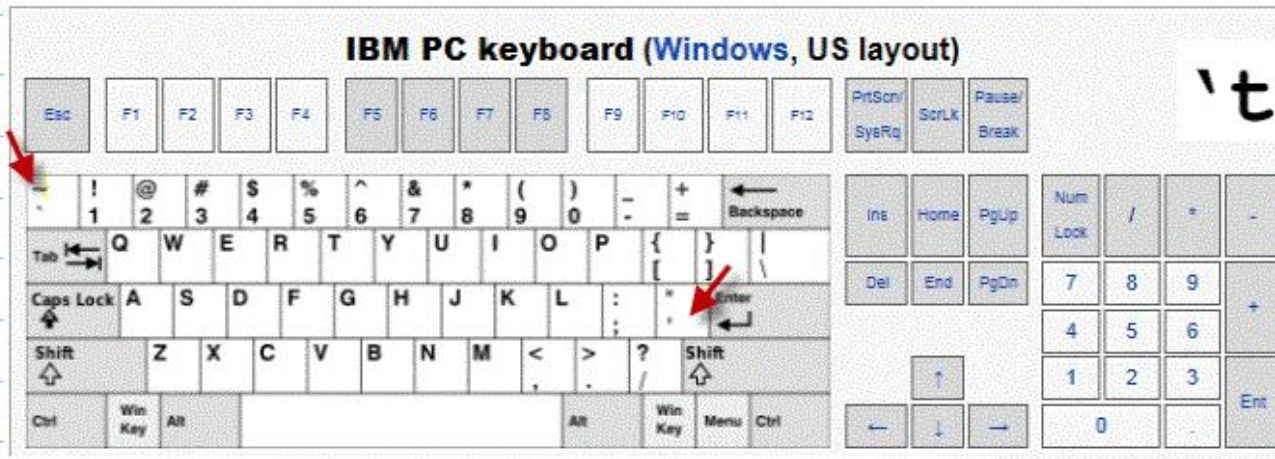
```
/*
- - -
- - -
- - -
- - - */
```

3. Compiler directives $\Rightarrow$ example 'define 'timescale

4. System calls: \$display \$fopen

'Compiler Directives

Directives are preceded by the **accent grave character** (') and are **not** terminated by a semicolon.



'timescale

page 21

page 18

```
'define width 7
reg ['width:0] vec8;
reg ['width:0] mema [1:1024];
```

'timescale 1ns/100ps

'timescale unit / precision

page 15

Compiler Directives

Standard IEEE 1364-2001 compiler directives

<code>`celldefine</code>	<code>`define</code>
<code>`endcelldefine</code>	<code>`undef</code>
<code>`default_nettype</code>	<code>`ifdef</code>
<code>`include</code>	<code>`ifndef</code>
<code>`resetall</code>	<code>`else</code>
<code>`timescale</code>	<code>`elsif</code>
<code>`unconnected_drive</code>	<code>`endif</code>
<code>`nounconnected_drive</code>	

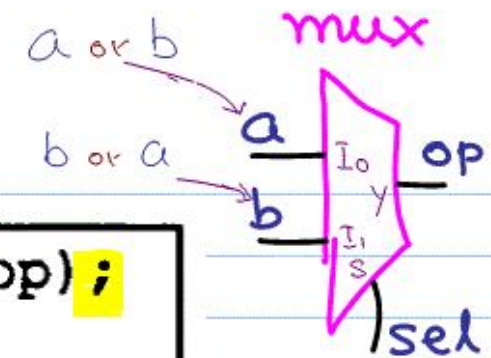
System Tasks and Functions

Selected IEEE 1364-2001 system tasks and functions:-

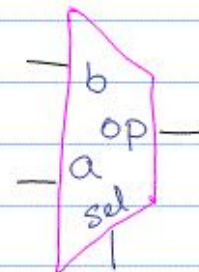
<code>\$bitstoreal</code>	<code>\$random</code>
<code>\$display(b/h/o)</code>	<code>\$readmem(b/h)</code>
<code>\$fclose</code>	<code>\$realtime</code>
<code>\$fdisplay(b/h/o)</code>	<code>\$realtobits</code>
<code>\$fopen</code>	

Esperan pages 94, 95, 96

module



Verilog 2001
preferred
notation



```

module mux (a, b, sel, op);
    input [7:0] a, b;
    input sel;
    output [7:0] op;
    reg [7:0] op;
    wire sel; wire [7:0] a, b;
    always @(a, or b, or sel)
        op = sel ? a : b;
endmodule
  
```

One of the
two common
data types

wire
and
reg

wire
is the default
data type.

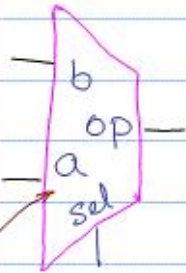
wire necessary to declare

<condition> ? <true_expr> : <false_expr>;

page 96

Verilog 2001 Port list

```
module mux (input  wire [7:0] a,  
            input  wire [7:0] b,  
            input  wire sel,  
            output reg [7:0] op);  
  
    always @(a, or b, or sel)  
        op = sel ? a : b;  
endmodule
```

note
a

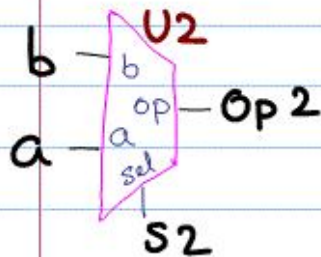
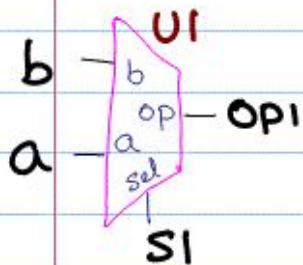
page 95

Module Instantiation Named and Positional Port Connection

```
mux U1 (.a(a), .b(b), .sel(s1), .op(op1));
```

.<port name>(<variable_name>)

• pin (system_label)



```
module muxes (a, b, s1, s2, op1, op2);
  input [7:0] a, b;
  input s1, s2;
  output [7:0] op1, op2;

  mux U1 (.a(a), .b(b),
          .sel(s1), .op(op1));
  mux U2 (a, b, s2, op2);

endmodule
```

page 95

Parameterized Modules

```

module muxn (a, b, sel, op);
  parameter WIDTH = 2;
  input [WIDTH-1:0] a, b;
  input sel;
  output [WIDTH-1:0] op;

  assign op = sel ? a : b;

endmodule

```

Verilog 2001 notation page 92

```

module mux_size
  #(parameter WIDTH = 8)
  (input wire [WIDTH-1:0] a,
   input wire [WIDTH-1:0] b,
   input wire sel,
   output reg [WIDTH-1:0] op);
  ...
endmodule

```

```

muxn #(8) U1 (.a(a), .b(b), .sel(select),
              .op(op1));

```

Data Types: wire and

pages 98, 99, 119, 120

Net Data Types

wire	For standard interconnection (wire is the default)
tri	
wand wor	For multiple drivers that are Wire-ORed
wand, triand	For multiple drivers that are Wire-ANDed
triereg	For nets with capacitive storage
tri0, tri1	For nets that pull up or down when not driven
supply0 supply1	For power or ground rails in netlists only

Register Data Types

reg	Unsigned integer variable of varying bit width	Most common register type. Can be declared as scalar or vector.
integer	Signed integer variable, 32-bits wide.	Used for signed arithmetic and general use. Always 32 bits.
time	Unsigned integer variable, 64-bits wide.	Used for manipulating simulation time, delays etc
real	Signed floating-point variable, double precision.	Used for real numbers

Net Data Types

wire tri	For standard interconnection (wire is the default)
-------------	---

Register Data Types

reg	Unsigned integer variable of varying bit width	Most common register type. Can be declared as scalar or vector.
integer	Signed integer variable, 32-bits wide.	Used for signed arithmetic and general use. Always 32 bits.

page 95

```

module muxn (a, b, sel, op);
  parameter WIDTH = 2;
  input [WIDTH-1:0] a, b;
  input sel;
  output [WIDTH-1:0] op;

  assign op = sel ? a : b;

endmodule

```

Op defaults to wire type.

page 94

```

module mux (a, b, sel, op);
  input [7:0] a, b;
  input sel;
  output [7:0] op;
  reg [7:0] op;

  always @(a or b or sel)
    op = sel ? a : b;

endmodule

```

MSB

reg [7:0] op;

vector
data item
[MSB:LSB]

```

always @(a or b or sel)
  op = sel ? a : b;

```

Language designer's notion of **NET** types and **REGISTER** types

NET

Wire constantly driven

Example: L.H.S of a continuous assign statement

REGISTER

reg Occasionally driven

Example: L.H.S of a sequential statement
in an always or initial
procedural block.

Note: R.H.S of any statement can contain both **NET** type and **REGISTER** type data items.

wire data type

page 98

```
wire nota;           // Scalar wire
wire [7:0] w1, w2;    // Two 8-bit wires
wand c;              // Scalar wired-AND net
tri [15:0] busa;      // 16-bit tri-state bus

always @(a)
    nota = ~a;        // procedural
assign nota = ~a;      // continuous
```



?

reg and integer data types

page 119

```
reg [3:0] vect; // 4-bit unsigned vector
reg [2:0] p, q; // 2 3-bit unsigned vector
integer aint; // 32-bit signed integer
reg s; // single bit reg scalar
time delay; // time value
```

```
always @(p) p = ~q; // procedural ☒
```

```
initial aint = 5; // procedural ☒
```

```
assign p = ~q; // continuous ☒
```

?

page 17

'default_nettype

Undeclared variables default to a single bit wire in Verilog.

```
module test;
    assign undec = 1'b0; // wire
    ...
```

Verilog2001 default_nettype none

Verilog2001 adds the net type option **none**. With this option, any undeclared variables will generate a compiler error. This prevents connectivity problems due to a mis-type in a variable name.

none is **not** a reserved word

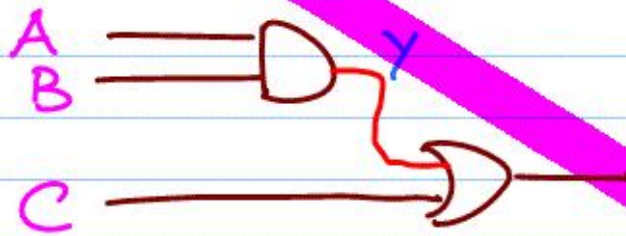
```
'default_nettype none
module test;
    //error - undefined object undec
    assign undec = 1'b0;
    ...
```

Primitive gates ← Unimportant for EE201L

pages 114, 115, 116, 117, 118

Primitives are PREDEFINED LOW-LEVEL modules for gate-level modeling. ← Sentence from ESPERAN

preferred for manual coding



Not preferred for manual coding

```
assign D = (A & B) | C;
```

```
assign Y = A & B;
assign D = Y | C;
```

```
and (Y, A, B);
or (D, Y, C);
```

First net is the output.

Rest of them are inputs.

An array of NETs or REGISTERS

Useful in modeling **memories**.

```
reg [7:0] mem [0:1023];
```

Width of
a location

Number of
locations

```
reg [15:0] a, b, c;
```

Three 16-bit registers named
a, b, c

need not be physical registers!

1 KB (1 kilo Byte)



page 92

Constants (same as Literals)

Rules and Examples

If size is omitted, it defaults to 32-bits. If base is omitted, it defaults to decimal. If the most significant bit of value is 0, x or z, then the bit will be automatically extended to the full width of size.

```
8'b1100_0001 // 8-bit binary
64'hff01      // 64-bit hexadecimal
12            // 32-bit decimal (default)
'h83a         // 32-bit hexadecimal
32'bz         // 32-bit z extension
```

page 92 continued

It is good practice to size a literal to match the value and the variable to which it is assigned. Remember over-long values are **truncated**, and over-short value **padded** with 0 from the most-significant bit.

```
reg [2:0] zbus;  
...  
// size/value too long, truncated:  
zbus = 4'b1001; // 001  
// value too long, truncated:  
zbus = 3'ha; // 010  
// value too short, padded:  
zbus = 3'b10; // 010  
// size/value too short, padded:  
zbus = 2'b1; // 001
```

page 92 continued

In Verilog2001, literal values can be qualified as **signed** (2's complement) using modifier **s** before base.

```
integer int = 4'sb1001 // -7
```

page 120

Register data type **integer**for signed
numbers

Integer Assignment

Arithmetic using integer variables is signed values (2's complement values). reg variables are unsigned (binary). reg and integer variables can be freely assigned to each other, with over-sized data being truncated, and under-sized data padded.

When a negative integer value is assigned to a reg variable, the reg variable will contain the 2's complement value, but as reg types are unsigned, the value is treated as a positive number. So if this value is then assigned back to an integer variable, the integer variable will contain a positive value.

page 120 continued

```
integer int;  
reg [3:0] data;  
initial  
begin  
  int = 129;  
  data = int;  
  int = data;  
  int = -3;  
  data = int;  
  int = data;  
end
```

sign extension

-64

+1

11..1000001

data = int; // data = 0001

int = data; // int = 1

int = -3; // 11..1111101

data = int; // data = 1101

int = data; // int = 13

page 123

signed

A Verilog2001 type **qualifier** which defines a register or net vector type as representing signed data.

Syntax

```
reg signed <[size]> name;
net_type signed <[size]> name;
```

where *size* defines the upper and lower bounds of the vector. and *net_type* is any net data type (e.g. wire).

It is now not necessary to use integer type just because you need to deal with signed numbers.

```
reg signed <[size]> name;
net_type signed <[size]> name;
```

Rules and Examples

By default, only integer and real data types are signed (i.e. 2's complement representation). The signed qualifier allows variables of vector types such as reg and wire to represent signed data.

```
reg signed [3:0] sreg;
reg [3:0] usreg;
integer aint = -5;

initial begin
    sreg = aint; // sreg = 4'b1011 = -5
    aint = sreg; // aint = -5
    usreg = aint; // usreg = 4'b1011 = +11
    aint = usreg; // aint = +11
    ...
end
```

```
reg signed [3:0] sreg;
reg [3:0] usreg;
```