Verilog Basic Language Constructs

- Lexical convention, data types and so on -

May 2014

Ando KI

Contents

- BNF
- Comment
- Identifier
- Keyword
- Operator
- Number
- Sting
- Compiler directives
- Data types

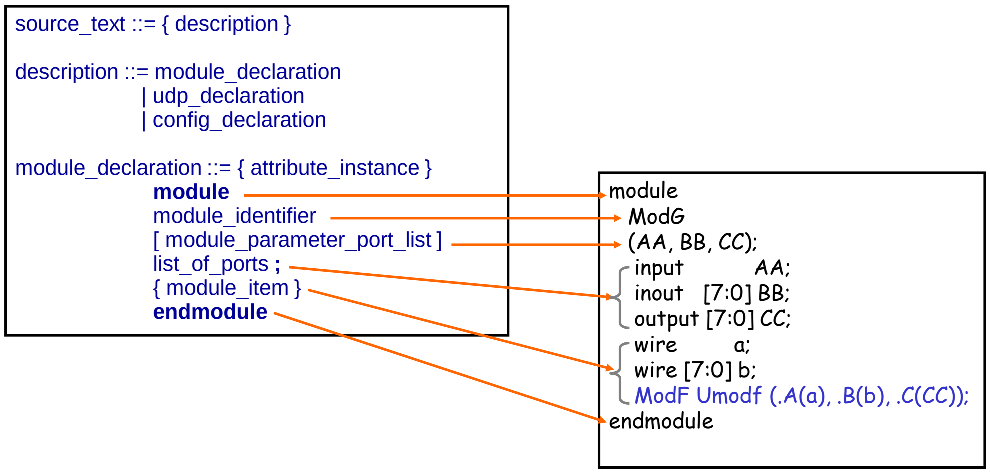
BNF: Backus-Naur Form

- A formal definition for a language syntax is a set of rules for forming valid programs in the language.
- BNF is the most widely used scheme to define syntax formally.
- A BNF definition of a language syntax is a set of syntax equations.
  - What is being defined appears on the left, and
  - The definition appears on the right.
  - Alternatives are separated by vertical bars: i.e., ‘a | b’ stands for “a or b”.
  - Square brackets indicate optional: ‘[a]’ stands for an optional a.

to_be_defined ::= definition_1 | definition_2 [| definition_3]

Be defined to be *Or, alternatively* *Optional*

Verilog source text



```
source_text ::= { description }

description ::= module_declaration
              | udp_declaration
              | config_declaration

module_declaration ::= { attribute_instance }
                   **module**
                   module_identifier
                   [ module_parameter_port_list ]
                   list_of_ports ;
                   { module_item }
                   **endmodule**
```

```
module
  ModG
  (AA, BB, CC);
   input        AA;
   inout  [7:0] BB;
   output [7:0] CC;
   wire         a;
   wire [7:0] b;
   ModF Umodf (.A(a), .B(b), .C(CC));
endmodule
```

Bold type characters are ‘terminals’.

Verilog lexical tokens

- Types of lexical tokens
  - White space: spaces, tabs(\t), newlines(\n), and EOF (end of file)
  - Comment
  - Identifier: a unique name (string) given to an object
  - Keyword: predefined non-escaped identifiers that are used to define the language constructs. All keywords are lower-case.
  - Operator
  - Number
  - String: a sequence of characters enclosed by double quotes (“ ”) and contained on a single line
  - Compiler directives: identifier following the ` character (the ASCII value 0x60, called open quote or accent grave) controls how a compiler should process its input

Comments

- A comment is a programming language construct used to embed information in the source code of a computer program and it is ignored by compiler.
- one-line comment
  - A single line comment begins with ‘//’ and ends with a newline.
- block comment
  - A block comment begins with ‘/*’ and ends with ‘*/’.

```
comment ::=
      one_line_comment
    | block_comment

one_line_comment ::= // comment_text \n

block_comment ::= /* comment_text */

comment_text ::= { Any_ASCII_character }
```

```
// a single comment line

// a comment line
// another comment line

/* a single comment line */

/* a multiple-line comment line 1
    line 2
    line 3
    line 4 */
```

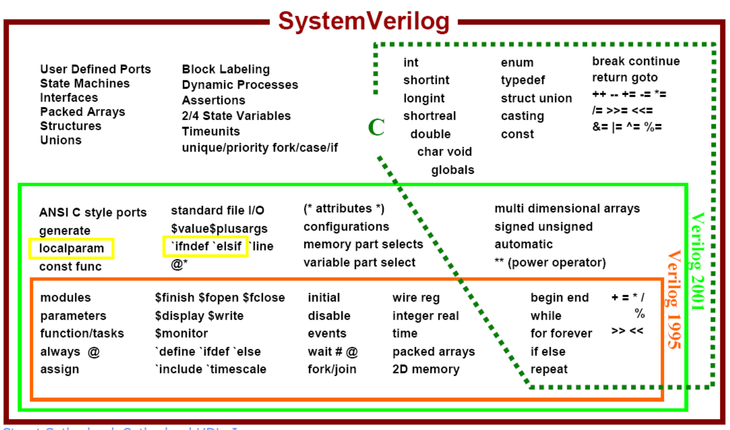
Identifier

- Verilog identifier is a unique name given to an object in a design.
  - Name is case sensitive.
- Identifier can be ‘simple identifier’ and ‘escaped identifier’.
- Simple identifier is a sequence of letters (alphabetic characters) and digits (numeric characters) including ‘_’ and ‘$’.
  - The first character should not be a number or ‘$’.
- Escaped identifier starts with backslash (‘\’) and ends with white space.
  - A means of including any of the printable ASCII characters in an identifier (33 (0x21) ~ 126 (0x7E)).

```
// simple identifier
shiftreg_a
busa_index
Error_confition
merger_ab
Merger_AB
_bus3
N$657

// escaped identifier
\busa+index
\-clock
\**error-condition***
\net1/\net2
\{a,b}
\a*(b+c)
```

Verilog keywords

- A keyword is a word or identifier that has a particular meaning to the programming language.
- Verilog is case-sensitive
  - All Verilog keywords are lower case.
  - However, don’t use case sensitive feature since some simulator are case insensitive.



Stuart Sutherland, Sutherland HDL, Inc.

Verilog-1995 keywords

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

Verilog-2001/2005 keywords (1/2)

always	design	forever	large
and	disable	fork	liblist
assign	edge	function	library
automatic	else	generate	localparam
begin	end	genvar	macromodule
buf	endcase	highz0	medium
bufif0	endconfig	highz1	module
bufif1	endfunction	if	nand
case	endgenerate	ifnone	negedge
casex	endmodule	incdir	nmos
casez	endprimitive	include	nor
cell	endspecify	initial	noshowcancelled
cmos	endtable	inout	not
config	endtask	input	notif0
deassign	event	instance	notif1
default	for	integer	or
defparam	force	join	output

Verilog-2001/2005 keywords (2/2)

parameter	rpmos	tran	wire
pmos	rtran	tranif0	wor
posedge	rtranif0	tranif1	xnor
primitive	rtranif1	tri	xor
pull0	scalared	tri0	
pull1	showcancelled	tri1	
pulldown	signed	triand	
pullup	small	trior	
pulsestyle_onevent	specify	trireg	
pulsestyle_ondetect	specparam	unsigned	
rcmos	strong0	use	
real	strong1	vectored	
realtime	supply0	wait	
reg	supply1	wand	
release	table	weak0	
repeat	task	weak1	
rnmos	time	while	

Verilog value set

Value	Meaning in logic level
0	Low logic level or false condition
1	High logic level or true condition
X, x	Unknown logic level
Z, z	High impedance logic level

The use of 'x' and 'z' in defining the value of a number is case insensitive.
'x' and 'z' will be considered false in if expression.
An 'x' in hexadecimal shall be 4-bit unknown. An 'x' in octal shall be 3-bit unknown. An 'x' in binary shall be 1-bit unknown. The same as 'z' case.
'?' is equivalent to z in literal number values.

For switch level modeling of MOS devices
- Driving (signal) strengths
  - ✓Strength1: (for logic 1) supply1, strong1, full1, weak1
  - ✓Strength0: (for logic 0) supply0, strong0, full0, weak0
- 3 charge strengths: small, medium, large

Verilog numbers (1/3)

- Verilog constant number can be specified as integer constant or real constant.
- Integer constants
  - Decimal (123, 4’d15)
  - Hexadecimal (‘h12F, 4’haBcD)
  - Octal (‘o763, 3’o7)
  - Binary (‘b1010, 4’b1100)
  - Sized constant form: 4’d15, 4’hAbCd)
  - Unsized constant form (123, ‘h12F)
  - Signed integer
    - Negative numbers shall be represented in 2’s complement format.
  - Unsigned integer

```
// Unsized constant numbers
 659         // is a decimal number
 ‘d659
 'h837FF    // is a hexadecimal number
 'o7460      // is an octal number
 4af         // is illegal (hexadecimal format requires 'h) – 'h4af

// Sized constant numbers
 4'b1001    // is a 4-bit binary number
 5'D3       // is a 5-bit decimal number
 3'b01x     // is a 3-bit number with the least significant bit unknown
 12'hx      // is a 12-bit unknown number
 16'hz      // is a 16-bit high-impedance number

// Using sign with constant numbers
 8'd-6      // this is illegal syntax
 -8'd6      // this defines the two's complement of 6,
            // held in 8 bits--equivalent to -(8'd 6)
 4'shf      // this denotes the 4-bit number '1111', to
            // be interpreted as a 2's complement number,
            // or '-1'. This is equivalent to -4'h1
 -4'sd15  // this is equivalent to -(-4'd 1), or '0001'
 16'sd?    // the same as 16'sbz
```

Verilog numbers (2/3)

- Integer constants
  - An ‘x’ shall be set 4-bit to unknown in the hexadecimal base, 3-bit in the octal base, and 1 bit in the binary base.
  - Similarly, a z shall set 4, 3 and 1, respectively, to the high-impedance value.
  - The underscore (‘_’) shall be legal anywhere in a number except as the first character.
  - Sized negative constant numbers and sized constant numbers are sign-extended when assigned to a reg data type, regardless of whether the reg itself is signed or not.

```
// Automatic left padding
 reg [11:0] a, b, c, d;
 initial begin
    a = 'h x;      // yields xxx
    b = 'h 3x;    // yields 03x
    c = 'h z3;    // yields zz3
    d = 'h 0z3;  // yields 0z3
 end

 reg  [84:0]      e, f, g;
 e = 'h5;        // yields {82{1'b0},3'b101}
 f = 'hx;        // yields {85{1'hx}}
 g = 'hz;        // yields {85{1'hz}}

// Using underscore character in numbers
 27_195_000
 16'b0011_0101_0001_1111
 32'h12ab_f001
```

Verilog numbers (3/3)

- Real constants
  - Decimal notation
  - Scientific notation
  - Real number and real variable are prohibited in the following cases.
    - Edge description (posedge, negedge)
    - Bit-select or part-select reference
    - Index expression of bit-select or part-select
- Real to integer conversion
  - Real number shall be converted to integer by rounding the real number to the nearest integer, rather than by truncating.

```
1.2
0.1
2394.26331
1.2E12 (the exponent symbol can be e or E)
1.30e-2
0.1e-0
23E10
29E-2
236.123_763_e-12 (underscores are ignored)

// illegal
.12
9.
4.E3
.2e-7

// conversion
 integer A;
 A = 35.7 // A will be 36
 A = 35.5 // A will be 36
 A = -1.5 // A will be -2
 A = 1.5  // A will be 2
```

Verilog constant number (1/2)

number ::= decimal_number | octal_number | binary_number | hex_number | real_number
decimal_number ::= [sign] unsigned_number | [size] decimal_base unsigned_number
binary_number ::= [size] binary_base binary_digit { _ | binary_digit }
octal_number ::= [size] octal_base octal_digit { _ | octal_digit }
hex_number ::= [size] hex_base hex_digit { _ | hex_digit }
real_number ::= [sign] unsigned_number **.** unsigned_number |
[sign] unsigned_number [**.** unsigned_number] **e** [sign] unsigned_number |
[sign] unsigned_number [**.** unsigned_number] **E** [sign] unsigned_number
sign ::= **+** | **-**
size ::= unsigned_number
unsigned_number ::= decimal_digit { _ | decimal_digit }
decimal_base ::= **'[s|S]d** | **'[s|S]D**
binary_base ::= **'[s|S]b** | **'[s|S]B**
octal_base ::= **'[s|S]o** | **'[s|S]O**
hex_base ::= **'[s|S]h** | **'[s|S]H**
decimal_digit ::= **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**
binary_digit ::= **x** | **X** | **z** | **Z** | **0** | **1**
octal_digit ::= **x** | **X** | **z** | **Z** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7**
hex_digit ::= **x** | **X** | **z** | **Z** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **a** | **b** | **c** | **d** | **e** | **f** | **A** | **B** | **C** | **D** | **E** | **F**

Underscore (_) can be used to enhance readability

Only decimal digit can be come first

Use single quotation, not back-quotation mark.

Bold type characters are 'terminals'.

Note that this is not complete version.

Verilog constant number (2/2)

```
1'b0       // a single bit, zero value
`b0        // 32-bit, all zeros
32'b0      // 32-bit, all zeros (32'b0000_0000_0000_0000_0000_0000_0000_0000)
4'ha       // a 4-bit hexa (4'b1010)
5'h5       // a 5-bit hexa (5'b0_0101)
4'hz       // a 4-bit high-impedance (4'bZZZZ)
4'h?ZZ? // the same as 4'hz since ? is an alternate form of z.
4'bx       // a 4-bit, all unknown
4'd9       // a 4-bit decimal
9          // a 32-bit number (32'b0000_0000_0000_0000_0000_0000_0000_1001)
A          // an illegal number
{10{1'b1}} // replication case
```

Operators (1/2)

- Logical operators: &&, ||, !
  - Result in one bit value
- Bitwise operators (infix): &, |, ~, ^, ~^, ^~
  - Operation on bit by bit basis
- Reduction operators (prefix): &, |, ^, ~&, ~|, ~^, ^~
  - Result in one bit value
- Shift operators: >>, <<
  - Result in the same size, always fills zero
  - Any X/x or Z/z will result in all X

operator	operation
&	bitwise and
|	bitwise or
~	bitwise not
^	bitwise xor
~^ or ^~	bitwise xnor

operator	operation
&&	Logical and
||	Logical or
!	Logical not

operator	operation
&	and
|	or
^	xor
~&	nand
~|	nor
~^ or ^~	xnor

operator	operation
>>	Shift right
<<	Shift left

Operators (2/2)

- Concatenation operators: {, }
  - Replication: {n{X}}
- Relational operators: >, <, >=, <=
  - Result in one bit value
- Logical equality operators: ==, !=
  - Result in either true or false
  - Any X/x or Z/z results in X
- Case equality operators: ===, !==
  - Exact match including X/x and Z/z
- Conditional operators: ? :
  - Like 2-to-1 mux
- Arithmetic/math operators: +, -, *, /, %
  - If any operand is x the result is x.
- Unary: +, -

```
reg [7:0] a;
reg [3:0] b, c;

a = {b, c};
a = b;
a = {4{1'b1},c};
a = {5{2'b01}; // 10'b0101010101
```

Forms of negation and AND/OR

- Negation

```
reg [3:0] a, b, c;

a = 5;
b = 0;
c = 0;
b = !a;  // logical negation: b will be 4'b0
c = ~a; // bit-wise negation: c will be 4'b1010
```

- AND/OR

```
reg [3:0] a, b, c, d;

a = 4'b1000 &   4'b0001; // a will be 4'b0
b = 4'b1000 && 4'b0001; // b will be 4'b0001

c = 4'b1000 |   4'b0001; // c will be 4'b1001
d = 4'b1000 || 4'b0001; // d will be 4'b0001
```

String

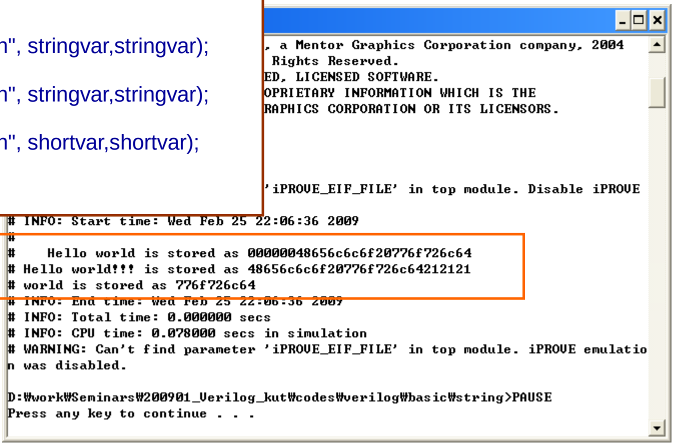
- A string is a sequence of character enclosed by double quotes (“”) and contained on a single line.
- One character is represented by one 8-bit ASCII value.
- A string can be an unsigned integer constant represented by a sequence of 8-bit ASCII values.
- The contents on the left are padded with 0 when variable width is wider than required.
- The leftmost characters will be lost when variable width is narrower than required.

```
// string variable declaration
 reg [8*12:1]  stringvar; // 8*12 or 96-bit wide
 initial begin
    stringvar="Helloworld!";
 end

// string manipulation
 module string_test;
    reg [8*14:1] stringvar;
   reg [8*5:1]  shortvar;
   initial begin
      stringvar = "Hello world";
      $display("%s is stored as %h", stringvar,stringvar);
      stringvar = {stringvar,"!!!"};
      $display("%s is stored as %h", stringvar,stringvar);
      shortvar = "Hello world";
      $display("%s is stored as %h", shortvar,shortvar);
   end
 endmodule
// the output
Hello world is stored as 00000048656c6c6f20776f726c64
Hello world!!! is stored as 48656c6c6f20776f726c64212121
world is stored as 776f726c64
```

String example

- ‘code/verilog/basic/string’ example.

```
module top;
  reg [8*14:1] stringvar;
  reg [8*5:1]  shortvar;
  initial begin
     stringvar = "Hello world";
     $display("%s is stored as %h", stringvar,stringvar);
     stringvar = {stringvar,"!!!"};
     $display("%s is stored as %h", stringvar,stringvar);
     shortvar = "Hello world";
     $display("%s is stored as %h", shortvar,shortvar);
  end
endmodule
```



```
, a Mentor Graphics Corporation company, 2004
 Rights Reserved.
ED, LICENSED SOFTWARE.
OPRIETARY INFORMATION WHICH IS THE
RAPHICS CORPORATION OR ITS LICENSORS.

'iPROVE_EIF_FILE' in top module. Disable iPROVE

# INFO: Start time: Wed Feb 25 22:06:36 2009
#
#    Hello world is stored as 00000048656c6c6f20776f726c64
# Hello world!!! is stored as 48656c6c6f20776f726c64212121
# world is stored as 776f726c64
# INFO: End time: Wed Feb 25 22:06:36 2009
# INFO: Total time: 0.000000 secs
# INFO: CPU time: 0.078000 secs in simulation
# WARNING: Can't find parameter 'iPROVE_EIF_FILE' in top module. iPROVE emulatio
n was disabled.

D:₩work₩Seminars₩200901_Verilog_kut₩codes₩verilog₩basic₩string>PAUSE
Press any key to continue . . .
```

DIY

Compiler directives

- Compiler directive or preprocessor directive
- `define
- `undef
- `include
- `ifdef, `else, and `endif
- `ifndef, `elsif
- `timescale

‘define and ‘undef

- ‘define
  - The directive `define creates a macro for text substitution.
  - Once a text macro name is defined, it can be used anywhere in a source description.
  - Redefinition of a text macro is allowed and the latest definition of a particular macro read by the compiler prevails when the macro name is encountered in the source text.
- `undef
  - The directive `undef is used to undefine a previously defined text macro.

Use back-quotation, not single quotation mark

```
`define SEL_A     4’b0000
`define SEL_B     4’b0001
`define FILE_X    “my_file.txt”
```

```
`undef SEL_A
`undef SEL_B
`undef FILE_X
```

`include

- `include
  - The `include compiler directive allows one to include the contents of a source file in another file during compilation.
  - The file name in the `include directive can be a full or relative path name.
  - A file included in the source using the `include compiler directive may contain other `include compiler directives.
  - The `include construct cannot be used recursively.
  - Implementations may limit the maximum number of levels to which include files can be nested.

Use back-quotation, not single quotation mark

```
`include "my_file.v"
```

`ifdef, `ifndef, `else, `elsif and `endif

- `ifdef - `elsif - `else - `endif
  - The `ifdef compiler directive checks for the definition of a macro.
  - If the macro is defined, then the line following the `ifdef directive are included.
  - If the macro is not defined and an `else directive exists, then the source is compiled.

```
 `ifdef   CERTAIN_MACRO
             `include "my_first.v"
`elsif ANOTHER_MACRO
              // bla bla ..
 `else
              `include "my_second.v"
 `endif
```

- `ifndef - `elsif - `else - `endif
  - The `ifndef compiler directive checks for the definition of a macro.
  - If the macro is not defined, then the line following the `ifndef directive are included.
  - If the macro is defined and an `else directive exists, then the source is compiled.

```
 `ifndef   CERTAIN_MACRO
             `include "my_first.v"
`elsif ANOTHER_MACRO
              // bla bla ..
 `else
              `include "my_second.v"
 `endif
```

`timescale

- It specifies the time unit and time precision of the modules.
  - It specifies the unit of measurement for time and delay values.
  - It specifies the degree of accuracy for delays.
    - Any time calculations would be internally rounded to the nearest the time precision.
- Its effect spans to all modules that follow this directive until another `timescale directive is read.

```
timescale_compiler_directive ::=
    `timescale time_unit / time_precision
```

Use back-quotation mark.

```
`timescale 1 ns / 1 ps
module xx (yy, zz);
  ……
endmodule
```

Char. string	unit	remarks
s	Second	-
ms	10^{-3} second	Milli
us	10^{-6} second	Micro
ns	10^{-9} second	Nano
ps	10^{-12} second	Pico
fs	10^{-15} second	Femto

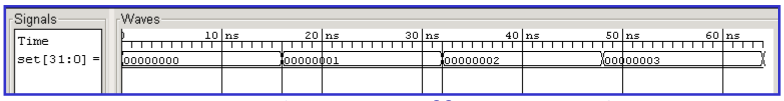
Time scale example

- ‘code/verilog/basic/timescale’ example.

```
`timescale 10 ns / 1 ns
module top;
  integer set;
  parameter d = 1.55;
  initial begin
      set = 0;
    #d set = 1; // 1.55*10   --> 15.5 --> 16
    #d set = 2; // 16+1.55*10 --> 31.5 --> 32
    #d set = 3; // 32+1.55*10 --> 47.5 --> 48
    #d set = 4; // 48+1.55*10 --> 63.5 --> 64
  end
  initial begin
    $dumpfile("wave.vcd");
    $dumpvars(1);
  end
endmodule
```

- The `timescale 10 ns / 1 ns compiler directive specifies that the time unit for module test is 10 ns. As a result, the time values in the module are multiples of 10 ns, rounded to the nearest 1 ns; therefore, the value stored in parameter d is scaled to a delay of 16 ns.
- The value of parameter d is rounded from 1.55 to 1.6 according to the time precision.

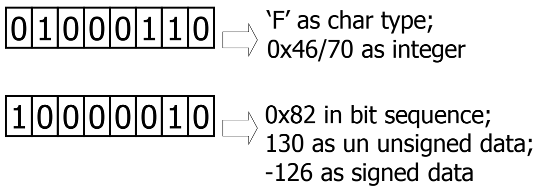
DIY

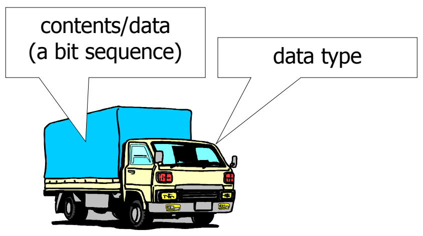


16 32 48

What is data type

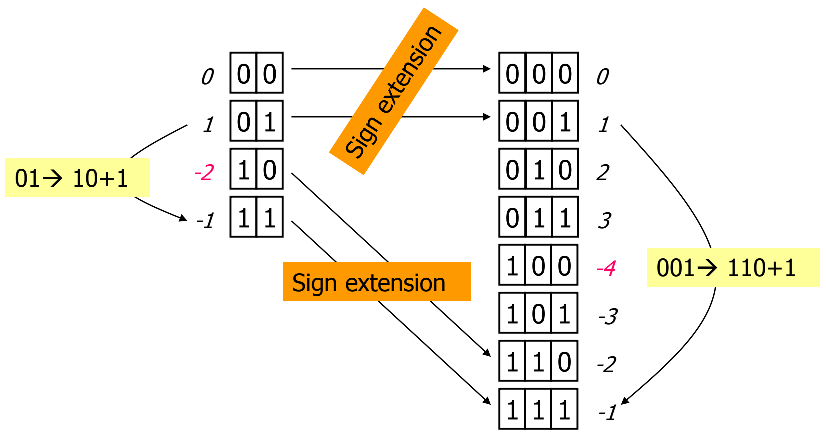
- Data type is the way how we interpret the meaning of bit sequences.





Signed and unsigned

- Signed data types
  - 2's complement signed
  - In 2's complement notation, one more negative value than positive value can be represented.



Verilog data types

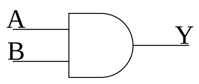
- Verilog data type is designed to represent the data storage and transmission elements found in the digital hardware.
- Verilog data types
  - Net
    - wire
  - Variable
    - Reg
    - Integer
    - Real
    - Time
    - Realtime

Nets (1/2)

- Represents a hardware wire
- Driven by logic
- Equal z (high-impedance) when unconnected
- Initial value is x (unknown)
- Should be used with continuous assignment to assign value, not procedural assignment
- Types of nets
  - wire, wand, wor
  - tri, tri0, tri1, triand, trior, trireg
  - supply0 – global net GND
  - supply1 – global net VCC (VDD)

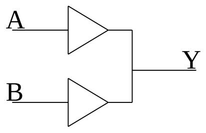
Types		
wire	Just wire	Only one driver is allowed
wand	Wired-AND	Multiple-driver
wor	Wired-OR	Multiple-driver
tri	Tri-state	The same as ‘wire’, but 3 states

Nets (2/2)



```
wire Y;  // declaration
assign Y = A & B;
```

Y	A=0	A=1
B=0	0	0
B=1	0	1

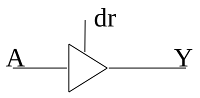


```
wand Y;  // declaration
assign Y = A;
assign Y = B;
```

Y	A=0	A=1
B=0	0	0
B=1	0	1

```
wor Y;  // declaration
assign Y = A;
assign Y = B;
```

Y	A=0	A=1
B=0	0	1
B=1	1	1



```
tri Y;  // declaration
assign Y = (dr) ? A : z;
```

'z' or 'Z' means high-impedance.

Signal resolution

Table 3-2—Truth table for wire and tri nets

wire/ tri	0	1	x	z
0	0	x	x	0
1	x	1	x	1
x	x	x	x	x
z	0	1	x	z

Table 3-3—Truth tables for wand and triand nets

wand/ triand	0	1	x	z
0	0	0	0	0
1	0	1	x	1
x	0	x	x	x
z	0	1	x	z

Table 3-4—Truth tables for wor and trior nets

wor/ trior	0	1	x	z
0	0	1	x	0
1	1	1	1	1
x	x	1	x	x
z	0	1	x	z

Register, integer, real, and time

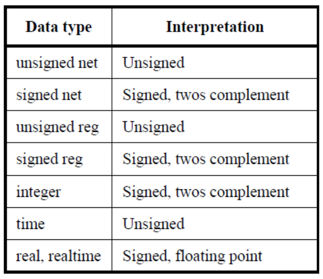
- Register
  - Variable that stores unsigned value
  - Can be a register (value holding unit) in real hardware
  - Only one type: ‘**reg**’
  - Can be signed by ‘**reg signed**’
- Variable that stores signed value
  - Need initialization if needed, since not initialized automatically.
  - Only one type: ‘**integer**’
  - Normally 32-bit
- Real that stores signed floating point value
  - Initialized to 0.0 automatically.
  - Only one type: ‘**real**’
- Time is a special data type for simulation time
  - Only one type: ‘**time**’
  - Normally 64-bit
  - `‘**realtime**’ stores time values as real number

```
// declaration
   reg         my_scalar_regA;
   reg         my_scalar_regB;
   reg [7:0] my_vector_reg;

// use inside procedure (initial or always block)
   my_scalar_regA = 1;
   my_scalar_regB = my_scalar_regA;
   my_vector_reg  = 8'hFA;

// declaration
   integer my_int;
   real      my_real;
   time     my_time;

// use inside procedure
   my_int = -11220;
   my_real = -3.99;
   my_time = $time; // get current sim. time
```

Data type interpretation

- Data type interpretation by arithmetic operators
  - Net, reg or time shall be treated as unsigned value by default.
  - Integer, real or realtime shall be treated as signed value by default.

Data type	Interpretation
net	Unsigned
reg	Unsigned
integer	Signed, 2’s complement
time	Unsigned
real	Signed, floating point



Data type	Interpretation
unsigned net	Unsigned
signed net	Signed, twos complement
unsigned reg	Unsigned
signed reg	Signed, twos complement
integer	Signed, twos complement
time	Unsigned
real, realtime	Signed, floating point

Assignments

- The assignment is the basic mechanism for placing values into nets and variables.

Statement type	Left-hand side (LHS)
Continuous	Net (wire)
Procedure	Variable (reg, integer, or time variable)

```
module full_adder(sum,cout,in1,in2,cin,clk,resetb);
   output sum, cout;
   input  in1, in2, cin;
   input  clk, resetb;

   wire   sum, cout;
   reg    rin1, rin2, rcin;
   wire   s1, c1; wire   s2, c2;

   always @ (posedge clk or negedge resetb) begin
      if (resetb==1'b0) begin
         rin1 <= 1'b0;  rin2 <= 1'b0;  rcin <= 1'b0;
      end else begin
         rin1 <= in1;  rin2 <= in2; rcin <= cin;
      end
   end

   half_adder_gate ha1 (.S(s1), .C(c1), .A(rin1), .B(rin2));
   half_adder_rtl    ha2 (.S(s2), .C(c2), .A(s1),   .B(rcin));

   assign sum = s2;
   assign cout = c1|c2;
endmodule
```

Procedural assignment

Continuous assignment

- The continuous assignment
  - It assigns values to nets.
  - It occurs whenever the value of the right-hand side changes.

```
wire   wire_tmp1;
assign  wire_tmp1 = my_value;
wire wire_tmp2 = my_value;
```

- The procedural assignment
  - It assigns values to variables.
  - It occurs within procedures, such as always, initial, task, and function.

```
wire  [3:0] w;
reg [3:0] a, b, c, d;
initial a = 4'h4;
always @ (c) b = c;
always @ (w) d = w;
```

Register assignment

- A register may be assigned value only within
  - A procedural statement, i.e., statement within initial block or always block
  - A user-defined sequential primitive
  - A task
  - A function
- A register may never be assigned value by
  - A primitive gate output
  - A continuous assignment

```
wire   wire_tmp1, wire_tmp2;
reg     reg_tmp;

and Uand(reg_tmp, wire_tmp1, wire_tmp2); // illegal

 assign reg_tmp = wire_tmp1 & wire_tmp2; // illegal
```

Vector and array data type

- Vector represents bus
  - Left number is MS bit
  - Vector assignment by position
- Array is a collection of the same data type

```
// vectors
   wire [3:0]  busA; // busA[3] is MSB
   reg  [1:4]  busB;  // busB[1] is MSB
   reg  [1:0]  busC;

// subfield access and positional assignment
   busC = busA[2:1]; // busC[1] = busA[2];
                     // busC[0] = busA[1];
```

```
// array
   integer    my_int[1:5]; // 5 integer variables
   reg        my_reg[0:99];
   reg [7:0]  my_mem[0:9];
   reg        my_x[7:0][0:10];
```

my_x: two-dimensional one-bit register.

my_mem: 8-bit memory from address 0 to 9.

Array declaration and assignments

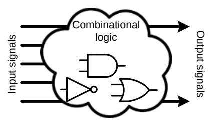
- Array declarations

```
reg [7:0] mema[0:255]; // declares a memory mema of 256 8-bit registers. The indices are 0 to 255
reg arrayb[7:0][0:255];  // declare a two-dimensional array of one bit registers
wire w_array[7:0][5:0]; // declare array of wires
integer inta[1:64];       // an array of 64 integer values
time chng_hist[1:1000] // an array of 1000 time values
integer t_index;
```

- Assignment to array elements

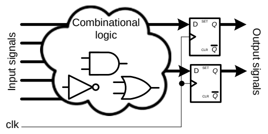
```
mema = 0;       // Illegal syntax- Attempt to write to entire array
arrayb[1] = 0;  // Illegal Syntax - Attempt to write to elements [1][0]..[1][255]
arrayb[1][12:31] = 0; // Illegal Syntax - Attempt to write to elements [1][12]..[1][31]
mema[1] = 0;      // Assigns 0 to the second element of mema
arrayb[1][0] = 0; // Assigns 0 to the bit referenced by indices [1][0]
inta[4] = 33559;  // Assign decimal number to integer in array
chng_hist[t_index] = $time; // Assign current simulation time to element addressed by integer index
```

Coding



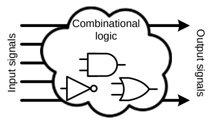
```
wire sig_in_a;
wire sig_in_b;
wire sig_out_c;
wire sig_out_d = sig_in_a & sig_in_b;
assign sig_out_c = sig_in_a | sig_in_b;
```

Combinational logic inference by continuous assignment (note wire and assign)



```
wire sig_in_a;
wire sig_in_b;
reg sig_out_c;
reg sig_out_d;

always @ (posedge clk) begin
    sig_out_d <= sig_in_a & sig_in_b;
    sig_out_c <= sig_in_a | sig_in_b;
end
```

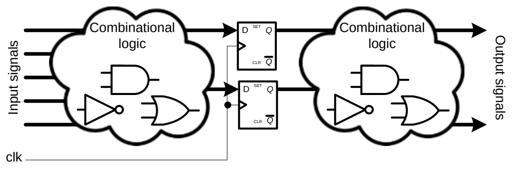
Register or Flip-Flop inference by always block (note “posedge clk”)



```
wire sig_in_a;
wire sig_in_b;
reg sig_out_c;
reg sig_out_d;

always @ ( * ) begin
    sig_out_d <= sig_in_a & sig_in_b;
    sig_out_c <= sig_in_a | sig_in_b;
end
```

Combinational logic inference by always block (note event list or sensitivity list)

Coding



```
wire sig_in_a;
wire sig_in_b;
wire sig_out_c;
wire sig_out_d;
reg sig_reg_x;
reg sig_reg_y

always @ (posedge clk) begin
    sig_reg_x <= sig_in_a & sig_in_b;
    sig_reg_y <= sig_in_a | sig_in_b;
end

assign sig_out_c = sig_reg_x & sig_reg_y;
assign sig_out_d = sig_reg_x | sig_reg_y;
```