

Verilog

- Operator, operand, expression and control -

May 2014

Ando KI

Contents

- Expression
- Operand
- Operator
- Bitwise and reduction operators
- Concatenation and replication example
- Sign example
- Shift example
- Expression bit-length
- Bit-length example
- Conditional statements
 - if-else/if-else-if
 - case
- Looping statements
 - forever example
 - repeat example
 - while example
 - for example
- control & conditional constructs
 - ◆ if-else/if-else-if
 - ◆ case
 - ◆ looping
 - repeat
 - for
 - while
 - forever

Verilog expression

■ Verilog expression combines operands with operators to produce a result.

- ◆ Scalar expression results a scalar (single-bit) result.
 - The least significant bit of the result is used when the expression results in multi-bit.

```
expression ::= primary
              | unary_operator { attribute_instance } primary
              | expression binary_operator { attribute_instance } expression
              | conditional_expression

primary ::= number
           | hierarchical_identifier [ { [ expression ] } [ range_expression ] ]
           | concatenation | multiple_concatenation
           | function_call | system_function_call
           | ( mintypmax_expression ) | string
```

Verilog operand

■ Verilog operand

- ◆ Constant number including real
- ◆ Net
- ◆ Variables of type reg, integer, time, real, and realtime
- ◆ Net bit-select
- ◆ Bit-select of type reg, integer, and time
- ◆ Net part-select
- ◆ Part-select of type reg, integer, and time
- ◆ Array element (not whole array)
- ◆ A call to a user-defined function or system-function that returns any of the above

■ signed and unsigned constant

- ◆ An integer with no base specification shall be a signed value in 2's complement form.
 - 12 and -12
- ◆ An integer with an unsigned base specification shall be interpreted as unsigned value.
 - 'd12
 - -'d12 == -32'd12 == FFFFFFF4
- ◆ Sign base should be used for signed integer.
 - -'sd12
 - or -SD12

Verilog operator (1/2)

operator	meaning	Integer express	Real expression
unary + unary -	Unary operators	Yes	Yes
{ } { }	Concatenation, replication	Yes	No
+ - * / **	Arithmetic	Yes	Yes
%	Modulus	Yes	No
> >= < <=	Relational	Yes	Yes
!	Logical negation	Yes	Yes
&&	Logical and/or	Yes	Yes
== !=	Logical equality/inequality	Yes	Yes
=== !==	Case equality/inequality	Yes	No
~ & ^	Bit-wise negation/and/or/xor/	Yes	No
~^ or ^~	Bit-wise equivalence	Yes	No
& ~& ~ ^ ~^ ~	Reduction and/hand/or/nor/xor/xnor	Yes	No
<< >>	Logical left/right shift	Yes	No
<<< >>>	Arithmetic left/right shift	Yes	No
? :	Conditional	Yes	Yes
or	Event or	Yes	Yes

Copyright © 2014 by Ando Ki

Introduction to Verilog-HDL expression (5)

Verilog operator (2/2)

- ❏ Logical operators: &&, ||, !
 - ◆ Result in one bit value
- ❏ Bitwise operators (infix): &, |, ~, ^, ~^, ^~
 - ◆ Operation on bit by bit basis
- ❏ Reduction operators (prefix): &, |, ^, ~&, ~|, ~^
 - ◆ Result in one bit value
- ❏ Logical shift operators: >>, <<
 - ◆ Result in the same size, always fills zero
- ❏ Concatenation operators: {, }
- ❏ Replication: {n{X}}
- ❏ Relational operators: >, <, >=, <=
 - ◆ Result in one bit value
- ❏ Logical equality operators: ==, !=
 - ◆ Result in either true or false
- ❏ 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: +, -

operator	Precedence
unary + unary -	Highest precedence
**	
* / %	
%	
+ - (binary)	
<< >> <<< >>>	
< <= > >=	
== != === !==	
& ~&	
^ ~^ ^~	
~	
&&	
? :	Lowest precedence

Copyright © 2014 by Ando Ki

Introduction to Verilog-HDL expression (6)

Bit-wise and reduction operators

Bit-wise binary AND

&	0	1	x	z
0	0	0	0	0
1	0	1	x	x
x	0	x	x	x
z	0	x	x	x

Bit-wise binary XOR

^	0	1	x	z
0	0	1	x	x
1	1	0	x	x
x	x	x	x	x
z	x	x	x	x

Bit-wise binary XNOR

\sim ^	0	1	x	z
0	1	0	x	x
1	0	1	x	x
x	x	x	x	x
z	x	x	x	x

Bit-wise binary OR

	0	1	x	z
0	0	1	x	x
1	1	1	1	1
x	x	1	x	x
z	x	1	x	x

Bit-wise unary negation

~	
0	1
1	0
x	x
z	x


Unary reduction example

Operand	&	~&		~	^	~^	Comments
4'b0000	0	1	0	1	0	1	No bits set
4'b1111	1	0	1	0	0	1	All bits set
4'b0110	0	1	1	0	0	1	Even number of bits set
4'b1000	0	1	1	0	1	0	Odd number of bits set

Copyright © 2014 by Ando Ki

Introduction to Verilog-HDL expression (7)

Concatenation and replication example

 'code/verilog/expression/replicate' example.

```
module top;
  reg a;
  reg [1:0] b;
  reg [3:0] c;
  reg [6:0] d;
  reg [9:0] e;
  initial begin
    a = 1'b1;
    b = 2'b01;
    c = {4{a}}; // replication of 1'b1
    d = {3{b}}; // replication of 2'b01
    e = {b, c, b}; // concatenation
    $display("a=%b b=%b c=%b d=%b e=%b", a, b, c, d, e);
  end
endmodule
```

a=1 b=01 c=1111 d=0010101 e=0001111101



Copyright © 2014 by Ando Ki

Introduction to Verilog-HDL expression (8)

Sign example (1/2)

❏ 'code/verilog/expression/sign' example.

```
module top;
  integer      intA, intB, intC, intD;
  reg [15:0] regA, regB, regC;
  reg signed [15:0] regSA, regSB;

  initial begin
    intA = -12 / 3;    // The result is -4.
    intB = -'d 12 / 3; // The result is 1431655761.
    intC = -'sd 12 / 3; // The result is -4.
    intD = -4'sd 12 / 3; // -4'sd12 is the negative of the 4-bit
                        // quantity 1100, which is -4. -(-4) = 4.
                        // The result is 1.
    $display("intA: -12 / 3    => %d 0x%h", intA, intA);
    $display("intB: -'d 12 / 3 => %d 0x%h", intB, intB);
    $display("intC: -'sd 12 / 3 => %d 0x%h", intC, intC);
    $display("intD: -4'sd 12 / 3 => %d 0x%h", intD, intD);
    //-----
  end
endmodule
```



Sign example (2/2)

❏ 'code/verilog/expression/sign' example.

```
intA = -4'd12;    // intA will be unsigned FFFFFFFF4
regA = intA / 3;  // expression result is -4,
                  // intA is an integer data type, regA is 65532
regB = -4'd12;    // regB is 65524
intB = regB / 3;  // expression result is 21841,
                  // regB is a reg data type
intC = -4'd12 / 3; // expression result is 1431655761.
                  // -4'd12 is effectively a 32-bit reg data type
regC = -12 / 3;   // expression result is -4, -12 is effectively
                  // an integer data type. regC is 65532
regSA = -12 / 3;  // expression result is -4. regSA is a signed reg
regSB = -4'sd12 / 3; // expression result is 1. -4'sd12 is actually 4.

$display("intA : -4'd12    => %d 0x%h", intA, intA);
$display("regA : intA / 3  => %d 0x%h", regA, regA);
$display("regB : -4'd12    => %d 0x%h", regB, regB);
$display("intB : regB / 3   => %d 0x%h", intB, intB);
$display("intC : -4'd12 / 3 => %d 0x%h", intC, intC);
$display("regC : -12 / 3   => %d 0x%h", regC, regC);
$display("regSA: -12 / 3   => %d 0x%h", regSA, regSA);
$display("regSB: -4'sd12 / 3 => %d 0x%h", regSB, regSB);

end
endmodule
```

```
# intA : -12 / 3    =>    -4 0xffffffc
# intB : -'d 12 / 3 => 1431655761 0x55555551
# intC : -'sd 12 / 3 =>    -4 0xffffffc
# intD : -4'sd 12 / 3 =>    1 0x00000001
# intA : -4'd12     =>   -12 0xfffffff4
# regA : intA / 3    => 65532 0xfffc
# regB : -4'd12     => 65524 0xfffc
# intB : regB / 3    => 21841 0x00005551
# intC : -4'd12 / 3 => 1431655761 0x55555551
# regC : -12 / 3    => 65532 0xfffc
# regSA : -12 / 3   =>    -4 0xfffc
# regSB : -4'sd12 / 3 =>    1 0x0001
```



Shift example

❏ 'code/verilog/expression/shift' example.

```
module top;
  reg [3:0] valueL, resultL, resultLS;
  reg signed [3:0] valueA, resultA, resultAS;
  initial begin
    valueL = 4'b1000;
    resultL = (valueL >> 2);
    resultLS = (valueL >>> 2); // be careful
    valueA = 4'b1000;
    resultA = (valueA >> 2); // be careful
    resultAS = (valueA >>> 2);
    //-----
    $display("valueL : %d 0x%h", valueL, valueL);
    $display("resultL : (valueL >> 2) ==> %d 0x%h", resultL, resultL);
    $display("resultLS : (valueL >>> 2) ==> %d 0x%h", resultLS, resultLS);
    $display("valueA : %d 0x%h", valueA, valueA);
    $display("resultA : (valueA >> 2) ==> %d 0x%h", resultA, resultA);
    $display("resultAS : (valueA >>> 2) ==> %d 0x%h", resultAS, resultAS);
  end
endmodule
```

```
# valueL : 8 0x8
# resultL : (valueL >> 2) ==> 2 0x2
# resultLS : (valueL >>> 2) ==> 2 0x2
# valueA : -8 0x8
# resultA : (valueA >> 2) ==> 2 0x2
# resultAS : (valueA >>> 2) ==> -2 0xe
```



Expression bit-length

❏ The Verilog uses the bit length of the operands to determine how many bits to use while evaluating an expression.

```
reg [15:0] a, b; // 16-bit regs
reg [15:0] sumA; // 16-bit reg
reg [16:0] sumB; // 17-bit reg
sumA = a + b; // expression evaluates using 16 bits
// it can lose carry overflow.
sumB = a + b; // expression evaluates using 17 bits
sumA = (a+b)>>1; // will not work properly
```

Expression	Bit length	Comments
Unsize constant number ^a	Same as integer	
Sized constant number	As given	
i op j, where op is: + - * / % & ^ ~	max(L(i),L(j))	
op i, where op is: + - ~	L(i)	
i op j, where op is: == != == != > < >= <=	1 bit	Operands are sized to max(L(i),L(j))
i op j, where op is: &&	1 bit	All operands are self-determined
op i, where op is: & ~& ~ ^ ~^ ~	1 bit	All operands are self-determined
i op j, where op is: >> << *+ >>> <<<	L(i)	j is self-determined
i ? j : k	max(L(j),L(k))	i is self-determined
{i,...j}	L(i)+...+L(j)	All operands are self-determined
{i(j,...k)}	i * (L(j)+...+L(k))	All operands are self-determined

Bit-length example

 'code/verilog/expression/bitlength'

example.

```
module top;
  reg [3:0] a, b, sumA;
  reg [4:0] sumB;
  initial begin
    b = 10;
    for (a=0; a<10; a=a+1) begin
      sumA = a + b;
      $display("sumA (%d) = a (%d) + b (%d)", sumA, a, b);
      b = b + 1;
    end
    b = 10;
    for (a=0; a<10; a=a+1) begin
      sumB = a + b;
      $display("sumB (%d) = a (%d) + b (%d)", sumB, a, b);
      b = b + 1;
    end
    b = 10;
    for (a=0; a<10; a=a+1) begin
      sumA = (a + b)>>1; // incorrect result
      sumB = (a + b)>>1; // correct result
      $display("sumA (%d) = (a (%d) + b (%d)) >> 1; %d", sumA, a, b, sumB);
      b = b + 1;
    end
  end
endmodule
```



Copyright © 2014 by Ando Ki

Introduction to Verilog-HDL expression (13)

Bit-length example

```
# sumA (10) = a (0) + b (10)
# sumA (12) = a (1) + b (11)
# sumA (14) = a (2) + b (12)
# sumA (0) = a (3) + b (13)
# sumA (2) = a (4) + b (14)
# sumA (4) = a (5) + b (15)
# sumA (6) = a (6) + b (0)
# sumA (8) = a (7) + b (1)
# sumA (10) = a (8) + b (2)
# sumA (12) = a (9) + b (3)
# sumB (10) = a (0) + b (10)
# sumB (12) = a (1) + b (11)
# sumB (14) = a (2) + b (12)
# sumB (16) = a (3) + b (13)
# sumB (18) = a (4) + b (14)
# sumB (20) = a (5) + b (15)
# sumB (6) = a (6) + b (0)
# sumB (8) = a (7) + b (1)
# sumB (10) = a (8) + b (2)
# sumB (12) = a (9) + b (3)
# sumA (5) = (a (0) + b (10)) >> 1; 5
# sumA (6) = (a (1) + b (11)) >> 1; 6
# sumA (7) = (a (2) + b (12)) >> 1; 7
# sumA (0) = (a (3) + b (13)) >> 1; 8
# sumA (1) = (a (4) + b (14)) >> 1; 9
# sumA (2) = (a (5) + b (15)) >> 1; 10
# sumA (3) = (a (6) + b (0)) >> 1; 3
# sumA (4) = (a (7) + b (1)) >> 1; 4
# sumA (5) = (a (8) + b (2)) >> 1; 5
# sumA (6) = (a (9) + b (3)) >> 1; 6
```

Overflow occurs

Overflow occurs



Copyright © 2014 by Ando Ki

Introduction to Verilog-HDL expression (14)

Conditional statement: if-else/if-else-if

```
conditional_statement ::=
```

```
    if_else_statement  
    | if_else_if_statement
```

```
if_else_statement ::=
```

```
    if ( expression ) statement [ else statement ]
```

```
if_else_if_statement ::=
```

```
    if ( expression ) statement  
    { else if ( expression ) statement }  
    [ else statement ]
```

Statement will be a single statement or a block of statement that is a group of states enclosed by 'begin' and 'end'.

■ The conditional statement (if-else or if-else-if statement) is used to make a decision about whether a statement is executed.

- ◆ If the expression evaluates to true (that is, has a nonzero known value), the first statement shall be executed.
- ◆ If it evaluates to false (that is, has a zero value or the value is x or z), the first statement shall not execute. If there is an else statement and expression is false, the else statement shall be executed.

Case statement

```
case_statement ::=
```

```
    case0_statement  
    | casez_statement  
    | casex_statement
```

```
case0_statement ::=
```

```
    case ( expression )  
    case_item { case_item } endcase
```

```
casez_statement ::=
```

```
    casez ( expression )  
    case_item { case_item } endcase
```

```
casex_statement ::=
```

```
    casex ( expression )  
    case_item { case_item } endcase
```

```
case_item ::=
```

```
    expression { , expression } :  
    statement  
    | default [ : ] statement
```

■ The case statement is a multiway decision statement that tests whether an expression matches one of a number of other expressions and branches accordingly. The default statement shall be optional.

■ In a case expression comparison, the comparison only succeeds when each bit matches exactly with respect to the values 0, 1, x, and z.

- ◆ The bit length of all the expressions shall be equal so that exact bitwise matching can be performed.

■ Don't care cases

- ◆ 'casez' treats z or ? symbols in the expression or case items as don't-cares.
- ◆ 'casex' treats z or x or ? symbols as don't-cares.

Looping statements

loop_statement ::=

```

    forever statement
  | repeat ( expression ) statement
  | while ( expression ) statement
  | for ( variable_assignment ;
        expression ;
        variable_assignment )
    statement

```

forever

- Continuously executes a statement.

repeat

- Executes a statement a fixed number of times. If the expression evaluates to unknown or high impedance, it shall be treated as zero, and no statement shall be executed.

while

- Executes a statement until an expression becomes false. If the expression starts out false, the statement shall not be executed at all.

Looping statements provide a means of controlling the execution of a statement zero, one, or more times.

for

- Controls execution of its associated statement(s) by a three-step process, as follows:

- Executes an assignment normally used to initialize a variable that controls the number of loops executed.
- Evaluates an expression. If the result is zero, the for loop shall exit. If it is not zero, the for loop shall execute its associated statement(s) and then perform step c). If the expression evaluates to an unknown or high-impedance value, it shall be treated as zero.
- Executes an assignment normally used to modify the value of the loop-control variable, then repeats step b).

Forever example

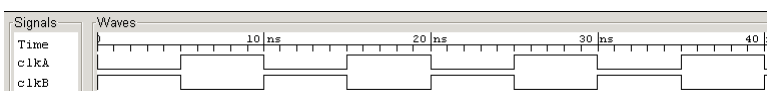
'code/verilog/expression/forever'
example.

```

module top;
  reg clkA, clkB;
  initial begin
    clkA = 0;
    forever #5 clkA = ~clkA;
  end
  initial begin
    clkB = 1;
    forever clkB = #5 ~clkB;
  end
  initial begin
    $dumpfile("wave.vcd");
    $dumpvars(1);
    #2000 $finish;
  end
endmodule

```

forever statement;



Repeat example

■ 'code/verilog/expression/repeat' example.

```
module top;
integer count;
initial begin
count = 10;
repeat (count) begin
$display("%d count down", count);
count = count - 1;
end
end
endmodule
```

■ The loop will execute 10 times regardless of whether the value of count changes after entry to the loop.

repeat (expression) statement;



While example

■ 'code/verilog/expression/while' example.

```
module top;
integer count;
initial begin
count = 10;
while (count>0) begin
$display("%d count down", count);
count = count - 1;
end
end
endmodule
```

while (expression) statement;



For example

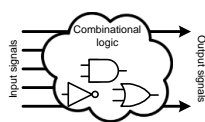
❏ 'code/verilog/expression/for' example.

```
module top;
integer count;
initial begin
count = 10;
for (count=10; count>0; count=count-1) begin
$display("%d count down", count);
end
end
endmodule
```

```
for (initial; condition; step)
statement;
```

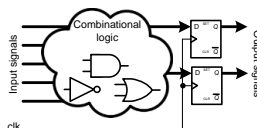


Coding



```
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;
```

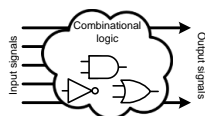
**Combinational logic inference by
continuous assignment**
(note wire and assign)



```
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")

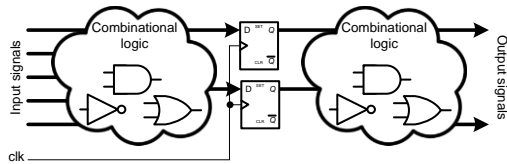


```
wire sig_in_a;
wire sig_in_b;
reg sig_out_c;
reg sig_out_d;

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;
```

Copyright © 2014 by Ando Ki

Reading

IEEE Std. 1364-2001, IEEE Standard Verilog Hardware Description Language.
(Chapter 4. Expressions; Chapter 9. Behavioral modeling)

Copyright © 2014 by Ando Ki

Introduction to Verilog-HDL expression (24)