BEE 271 Digital circuits and systems

Spring 2017

Lecture 5: Verilog and signed numbers

Nicole Hamilton

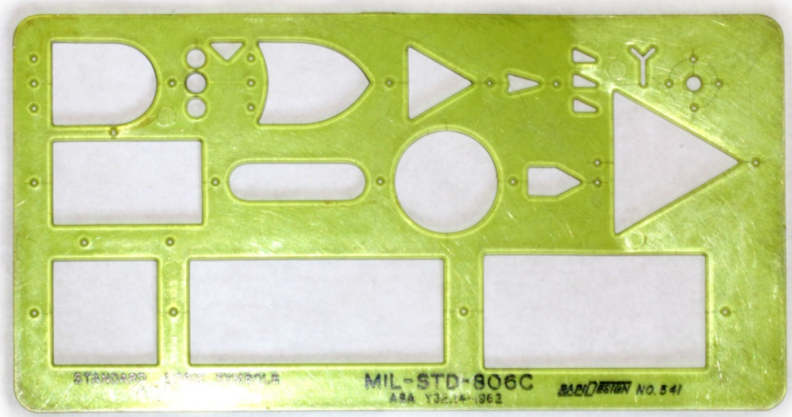
https://faculty.washington.edu/kd1uj

Topics

1. Verilog
2. Signed numbers

Verilog

40 years ago



Today

We no longer draw gates for complex designs.

Hardware description languages (*HDLs*) have replaced schematics.

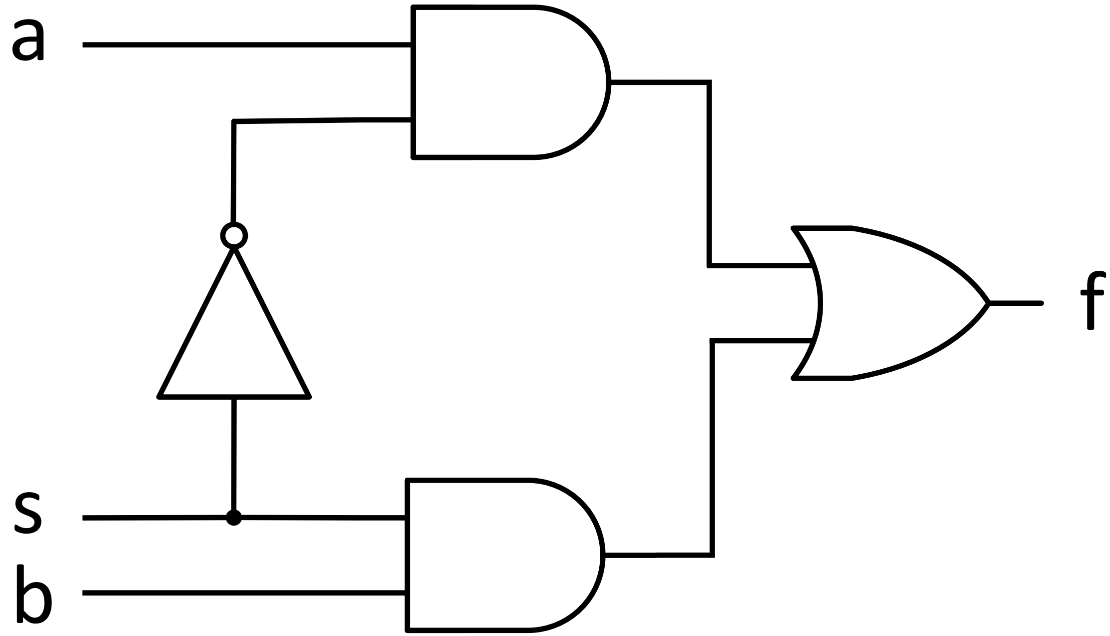
HDL

A *hardware description language* (*HDL*) allows us to describe logic circuits as if we were writing software in C or Java.

The advantages of Verilog

1. It’s far more productive. You can do in a weekend what might take two months with pencil and paper.
2. The compiler does all the multiple-input/multiple-output logic minimization for you.
3. For a complex design, it’s much easier to read than a schematic crawling with wires.
4. Your design is saved as an ordinary text file. You can edit it with any editor you like.
5. It’s portable. There’s an IEEE standard for the language.
6. You can compile it any vendor’s FPGA or even to a semi-custom or custom chip at a foundry.
7. It’s more similar to C than VHDL, which looks more like ADA.

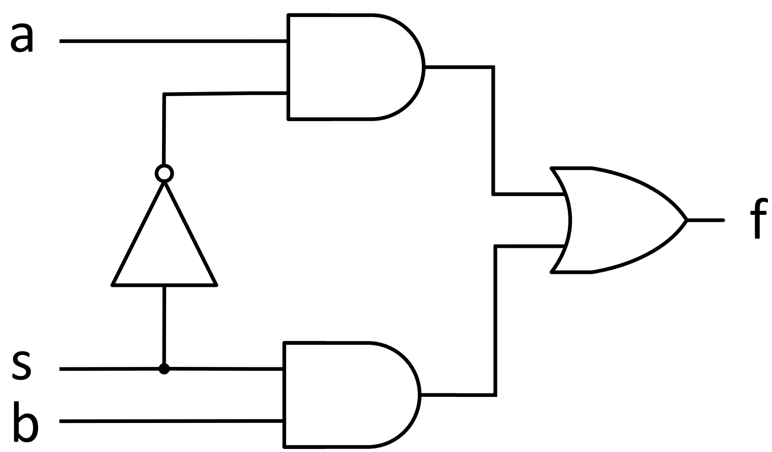
The Multiplexer



s	f
0	a
1	b

Selects a or b based on s, *multiplexing* these signals onto the output f.

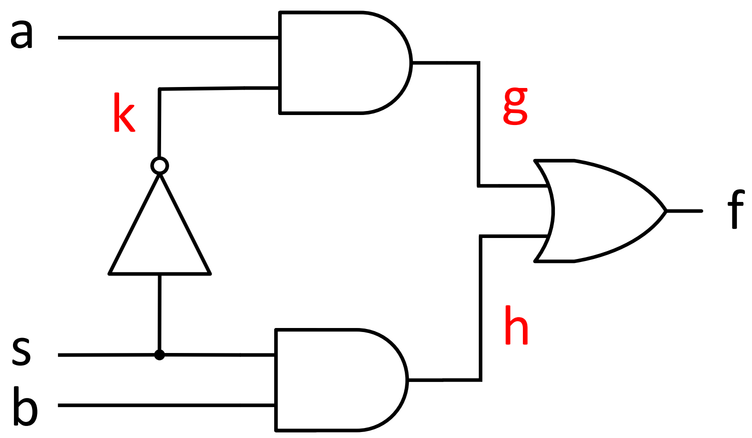
Three ways to represent the multiplexer in Verilog



1. Structural representation as gates.
2. Boolean expressions.
3. Behavioral description.

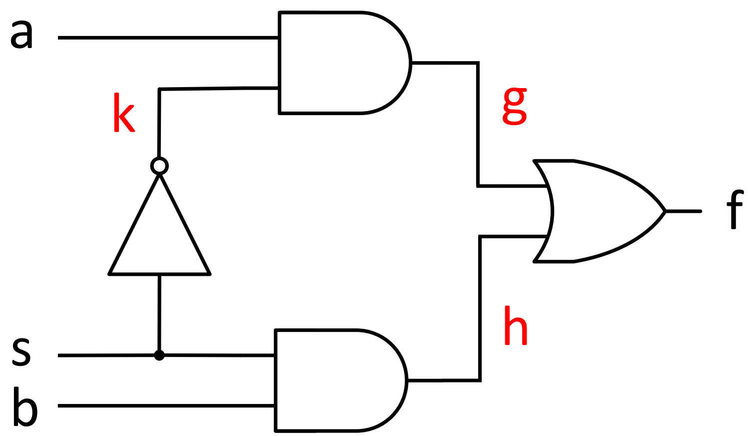
1. Structural representation

Structural representation as gates




```
module Mux2To1A(
        input  s, a, b,
        output f );

    wire g, h, k;
    not ( k, s );
    and ( g, k, a );
    and ( h, s, b );
    or  ( f, g, h );

endmodule
```

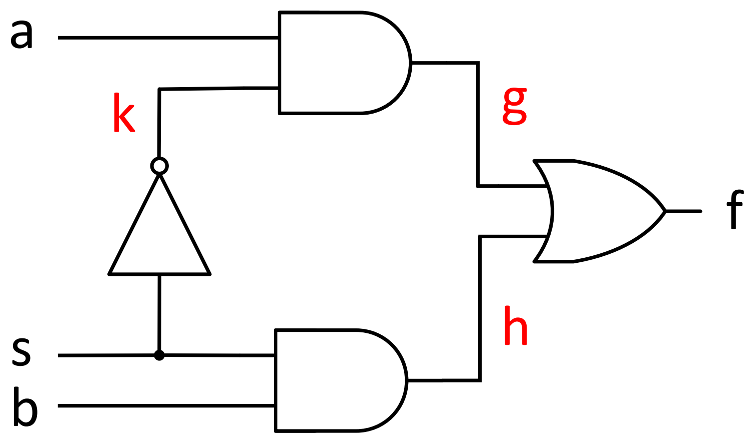
Verilog code for a multiplexer.




```
module Mux2To1A(
        input  s, a, b,
        output f );

    wire g, h, k;
    not ( k, s );
    and ( g, k, a );
    and ( h, s, b );
    or  ( f, g, h );

endmodule
```

The "ports"

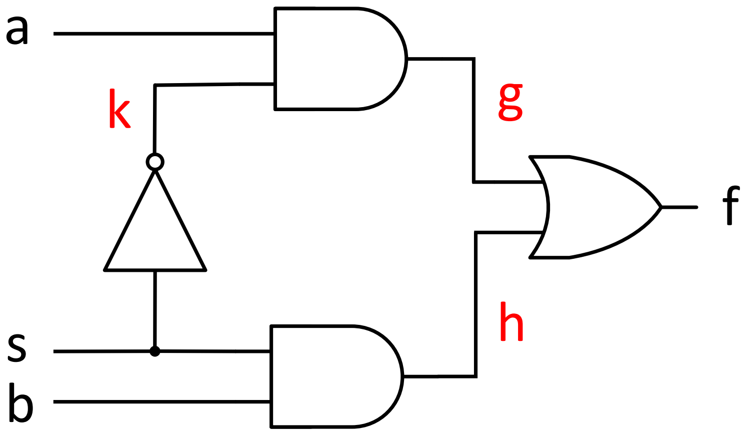
Verilog code for a multiplexer.




```
module Mux2To1C( s, a, b, f );

  input s, a, b;
  output f;

  wire g, h, j, k;
  not ( k, s );
  and ( g, k, a );
  and ( h, s, b );
  or  ( f, g, h );

endmodule
```

Alternate form of specifying the ports.

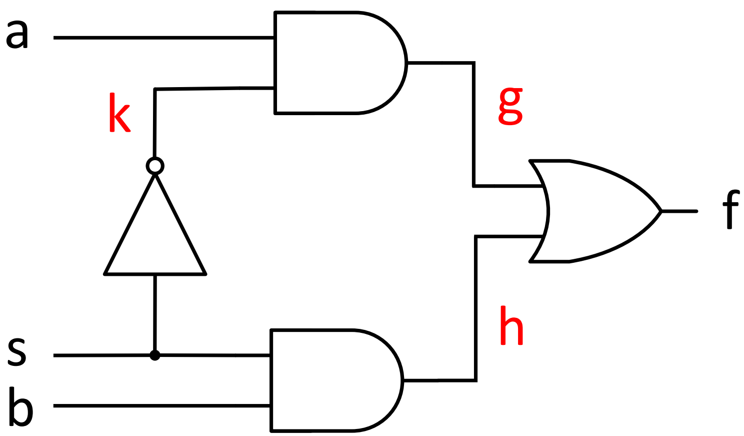
Verilog code for a multiplexer.




```
module Mux2To1A(
       input  s, a, b,
       output f );

   wire g, h, k;
   not ( k, s );
   and ( g, k, a );
   and ( h, s, b );
   or  ( f, g, h );

endmodule
```

Wires constantly reflect the value of whatever they're connected to.

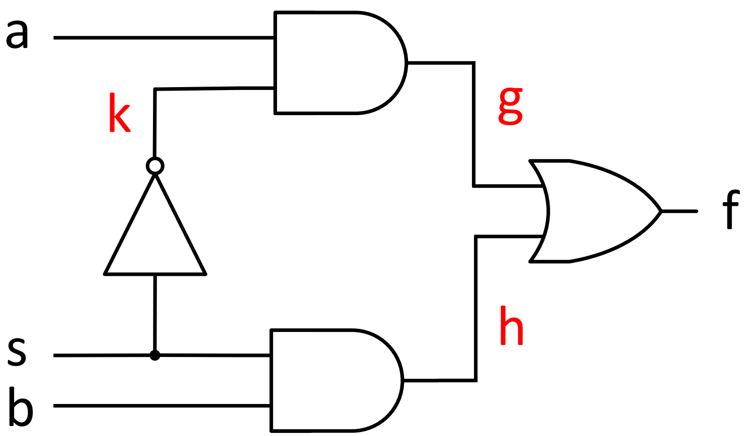
Verilog code for a multiplexer.




```
module Mux2To1B(
        input  s, a, b,
        output f );

    not ( k, s );
    and ( g, k, a );
    and ( h, s, b );
    or  ( f, g, h );

endmodule
```

Undeclared variables default to 1-bit wires.

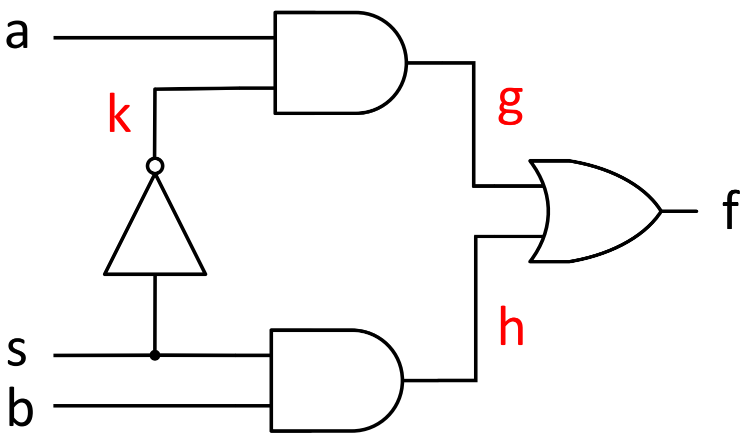
Verilog code for a multiplexer.




```
module Mux2To1A(
        input  s, a, b,
        output f );

    wire g, h, k;
    not ( k, s );
    and ( g, k, a );
    and ( h, s, b );
    or  ( f, g, h );

endmodule
```

The first port to a gate is the output.

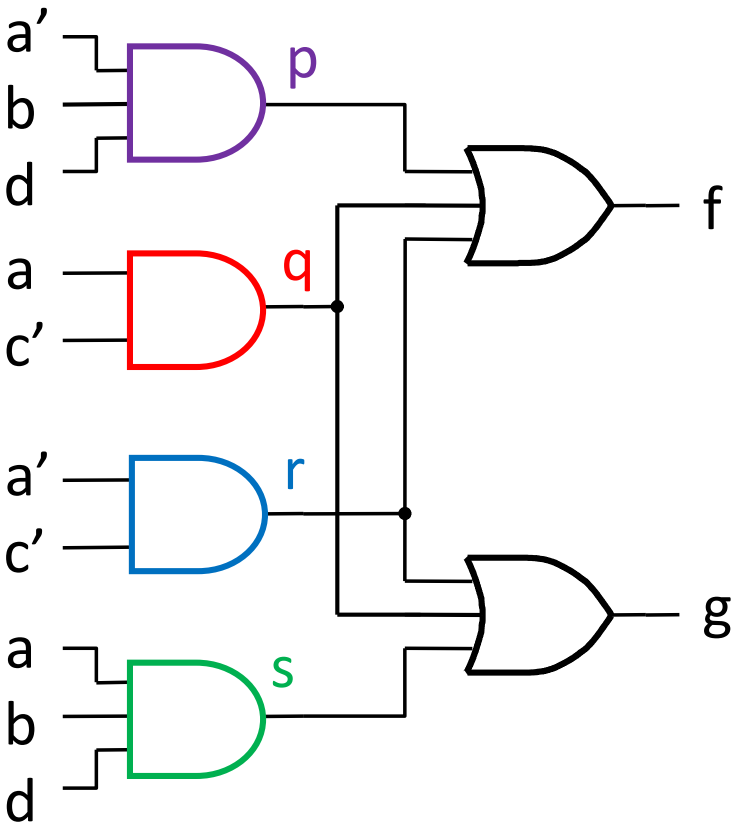
Verilog code for a multiplexer.




```
module Mux2To1A(
        input  s, a, b,
        output f );

    // This 2-in mux module.

    wire g, h, k;
    not ( k, s );
    and ( g, k, a );
    and ( h, s, b );
    or  ( f, g, h );

endmodule
```

Comments start with //.

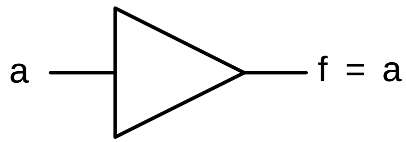
Verilog code for a multiplexer.



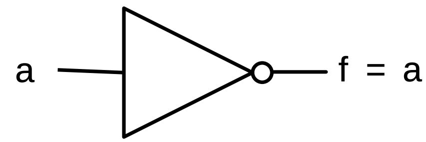

```
module MultiOutput(
      input a, b, c, d,
      output f, g );

   wire p, q, r, s;
   and ( p, ~a, b, d );
   and ( q, a, ~c );
   and ( r, ~a, ~c );
   and ( s, a, b, d );
   or ( f, p, q, r );
   or ( g, q, r, s );

endmodule
```

A multiple output example.

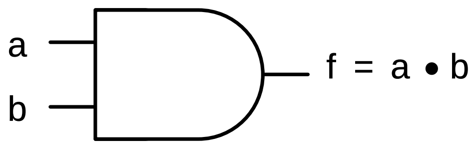
Basic gates in Verilog



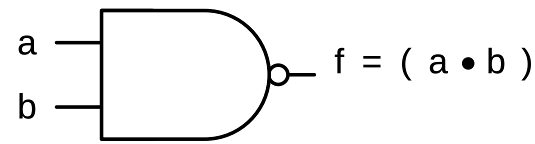
buf(f, a);



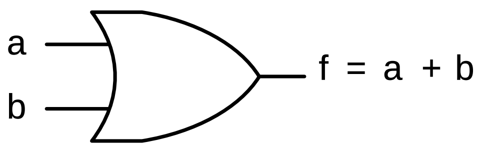
not(f, a);



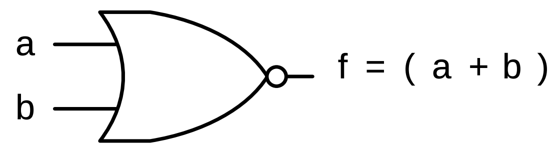
and(f, a, b, …);



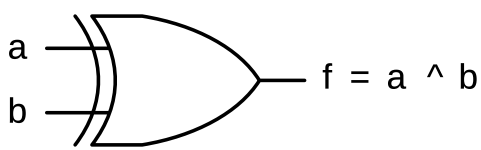
nand(f, a, b, …);



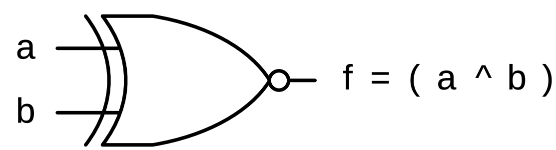
or(f, a, b, …);



nor(f, a, b, …);

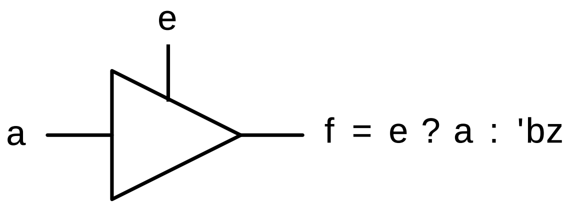


xor(f, a, b, …);

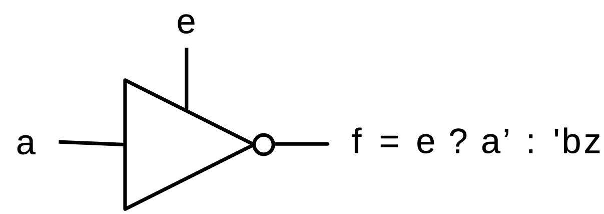


xnor(f, a, b, …);

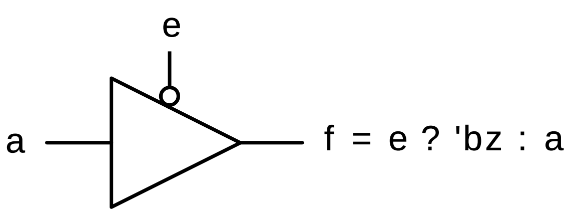
Tri-state drivers in Verilog



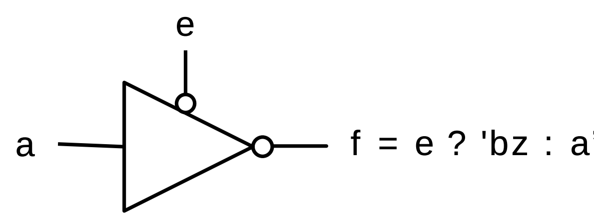
bufif1(f, a, e);



notif1(f, a, e);

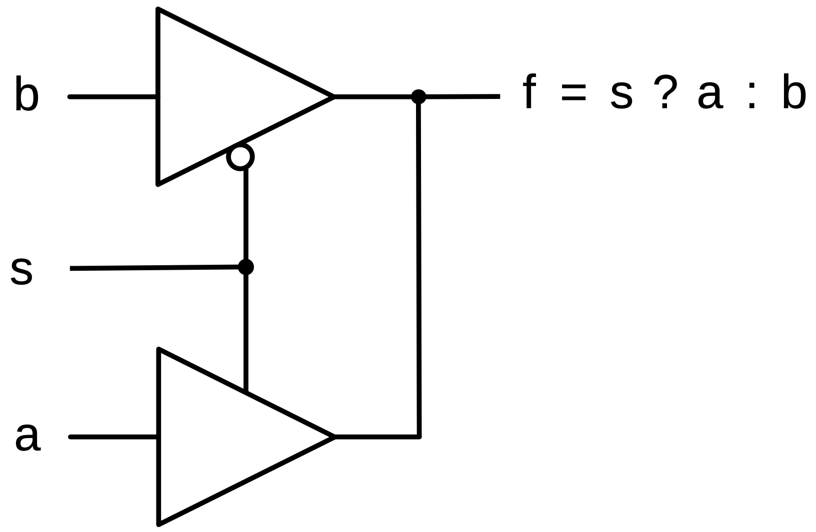


bufif0(f, a, e);

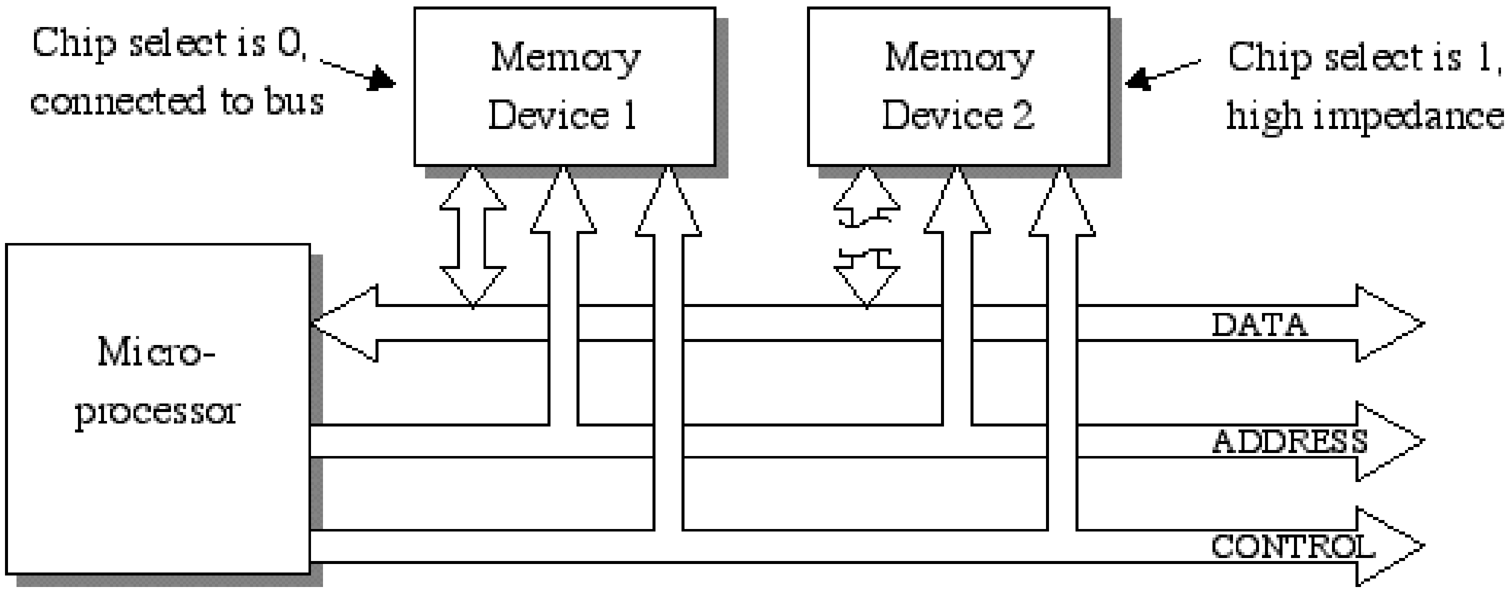


notif0(f, a, e);

A tri-state driver presents a high impedance (high Z) load unless enabled. It's as if it's disconnected.



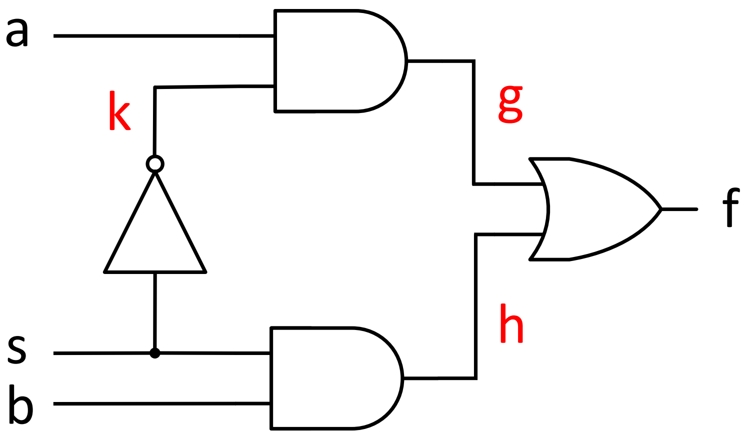
A multiplexer built from 2 tri-state drivers.



A more realistic application as a device or chip select.

Image source: http://faculty.etsu.edu/tarnoff/ntes2150/memory/memory.htm

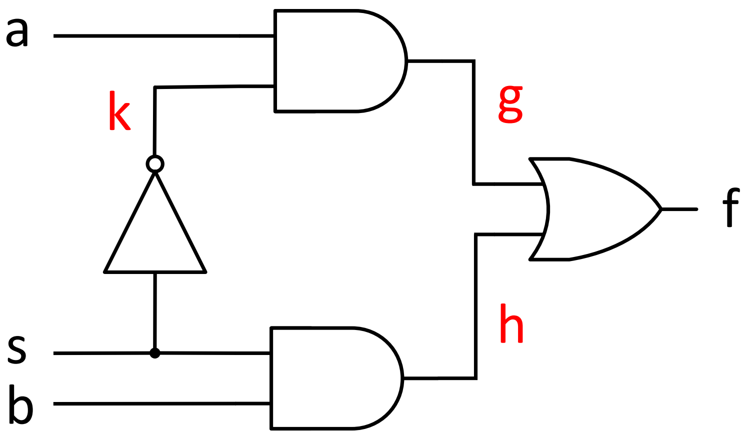
2. Boolean expressions




```
module Mux2To1D(
        input  s, a, b,
        output f );

    assign f = ~s & a | s & b;

endmodule
```

f will continuously reflect the value of the RHS.

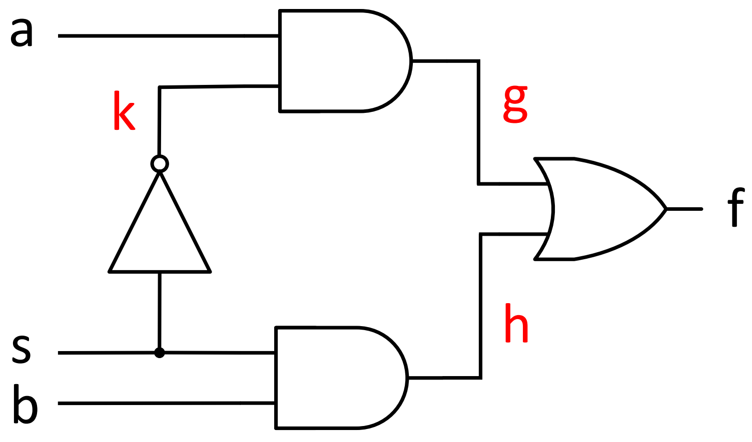
Continuous assignment




```
module Mux2To1D(
        input  s, a, b,
        output f );

    assign f = ~s & a | s & b;

endmodule
```

~ is done before &, which is done before |.

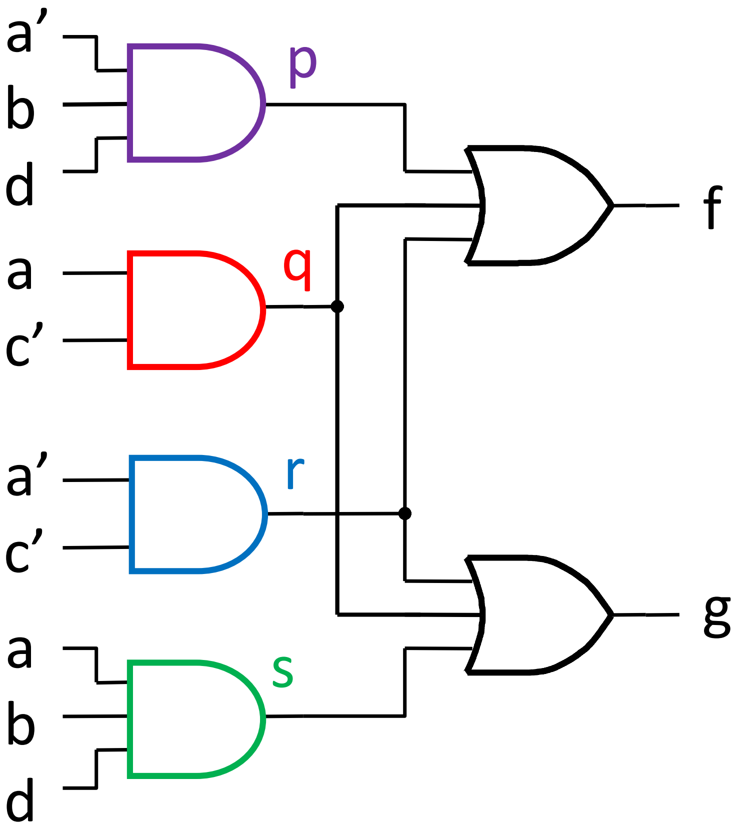
Continuous assignment




```
module Mux2To1E(
        input  s, a, b,
        output f );

    assign f = s ? b : a;

endmodule
```

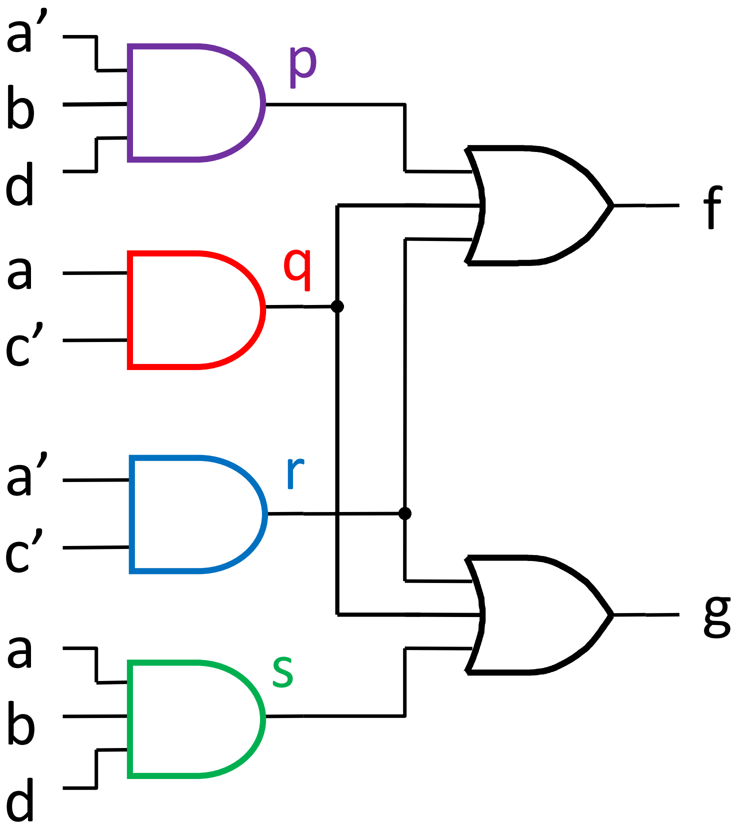
If s is true, f = b,
otherwise, f = a.

The trinary operator.




```
module MultiOutput(
        input a, b, c, d,
        output f, g );

    wire p, q, r, s;
    assign p = ~a & b & d;
    assign q = a & ~c;
    assign r = ~a & ~c;
    assign s = a & b & d;
    assign f = p | q | r;
    assign g = q | r | s;

endmodule
```

A multiple output example.




```
module MultiOutput(
        input a, b, c, d,
        output f, g );

    wire p, q, r, s;
    assign p = ~a & b & d,
           q = a & ~c,
           r = ~a & ~c,
           s = a & b & d,
           f = p | q | r,
           g = q | r | s;

endmodule
```

assign statements can be chained with commas.

Verilog operator precedence

<table>
<tr><td>( )   [ ]</td><td>Grouping.</td></tr>
<tr><td>~   -   !   +   & | ^</td><td>Unary bitwise, arithmetic and logical complements and plus and the AND, OR and XOR reduction operators. Right to left associativity.</td></tr>
<tr><td>*   /   %</td><td>Multiplication, division and remainder.</td></tr>
<tr><td>+   -</td><td>Addition and subtraction.</td></tr>
<tr><td><<   >></td><td>Bit-shifting.</td></tr>
<tr><td><   <=   >=   ></td><td>Relation testing.</td></tr>
<tr><td>==   !=</td><td>Equality testing.</td></tr>
<tr><td>&</td><td>Bitwise AND.</td></tr>
<tr><td>^</td><td>Bitwise XOR.</td></tr>
<tr><td>|</td><td>Bitwise OR.</td></tr>
<tr><td>&&</td><td>Logical AND.</td></tr>
<tr><td>||</td><td>Logical OR.</td></tr>
<tr><td>? :</td><td>Trinary conditional operator.</td></tr>
<tr><td>=   <=</td><td>Blocking and non-blocking assignment.</td></tr>
<tr><td>{ }   { { } }</td><td>Concatenation and replication.</td></tr>
</table>

Source: Table 11-2—Operator precedence and associativity, *IEEE Standard for SystemVerilog*, IEEE Std 1800-2012, IEEE, 2013, p. 221.

Basic Verilog operators

Assume a = 4’b0101 = 5, b = 3’b011 = 3.

Operator	Meaning	Result
Bitwise		
~ a	Bitwise inversion	1010
a & b	Bitwise *AND* = a b	0001
a | b	Bitwise *OR* = a + b	0111
a ^ b	Bitwise *XOR* = a’ b + a b	0110
Logical		
! a	Logical *NOT*: 1 if all bits of a = 0	0
a && b	Logical *AND*: 1 if both a and b are non-zero	1
a || b	Logical *OR*: 1 if either a or b is non-zero	1
Reduction		
& a	*AND* of all bits in a	0
| a	*OR* of all bits in a	1
^ a	*XOR* of all bits in a	0

Basic Verilog operators

Assume a = 4’b0101 = 5, b = 3’b011 = 3.

Operator	Meaning	Result
Relational		
a == b	a *equals* b	0
a != b	a *not equal* b	1
a > b	a *greater than* b	1
a < b	a *less than* b	0
a >= b	a *greater than or equal* b	1

Basic Verilog operators

Assume a = 4’b0101 = 5, b = 3’b011 = 3.

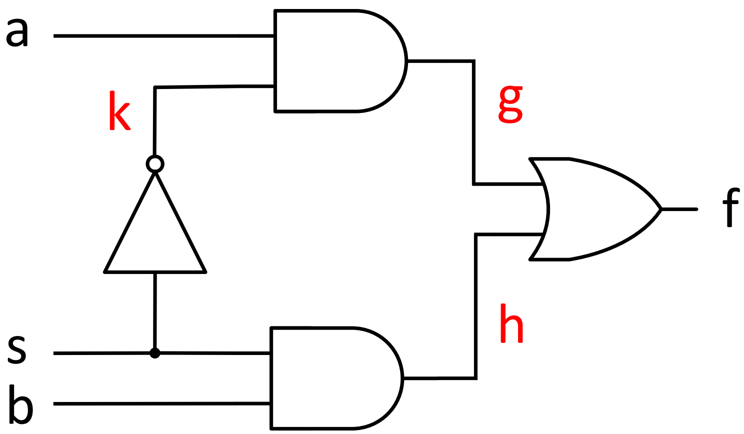
Operator	Meaning	Result
Arithmetic		
- a	Arithmetic complement	-5
+ a	Unary plus.	5
a * b	Multiplication.	15
a / b	Integer division.	1
a % b	Modulo (remainder) division.	2
Shifting		
a >> b	Shift a right b bits	0000
a << b	Shift a left b bits	1000

Basic Verilog operators

Assume a = 2 = 3’b010, b = 3’b011, c = 3’b101.

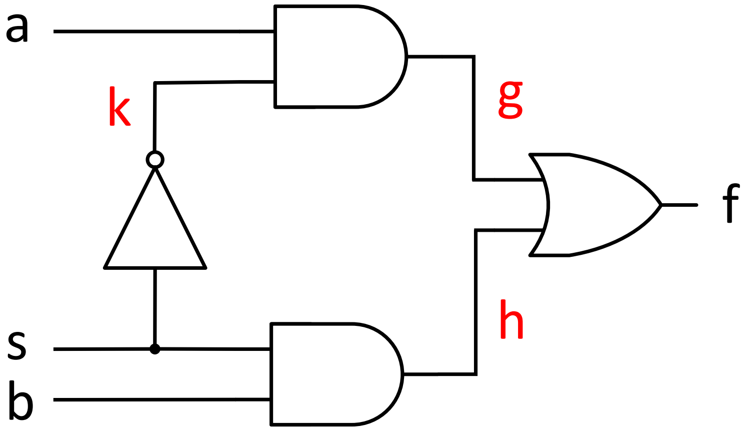
Operator	Meaning	Result
Concatenation and replication		
{ a, b }	Concate a and b.	010011
{ b { a } }	Replicate and concatenate b copies of a. b must be a constant.	010010010
Conditional (Trinary)		
a ? b : c	If a is non-zero, result = b. Otherwise, result = c.	011

3. Behavioral description




```
module Mux2To1F(
        input s, a, b,
        output reg f );

    always @( s, a, b )
        if ( s )
            f = b;
        else
            f = a;

endmodule
```

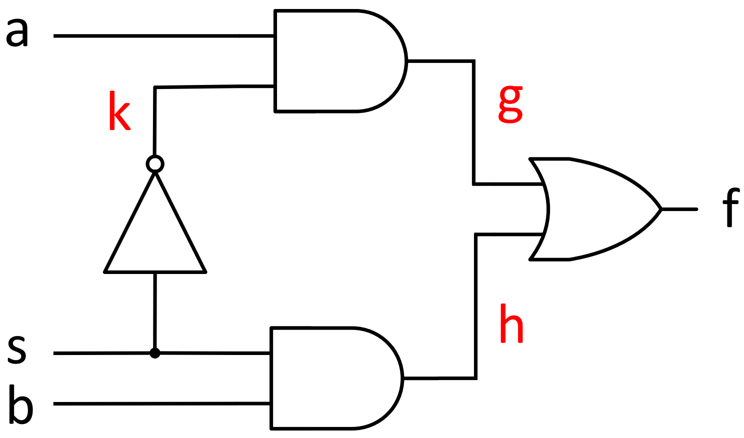
Behavioral specification of a multiplexer.




```
module Mux2To1F(
        input s, a, b,
        output reg f );

    always @( s, a, b )
        if ( s )
            f = b;
        else
            f = a;

endmodule
```

The *sensitivity list*. Anytime s, a or b changes, the circuit should *behave as described*.

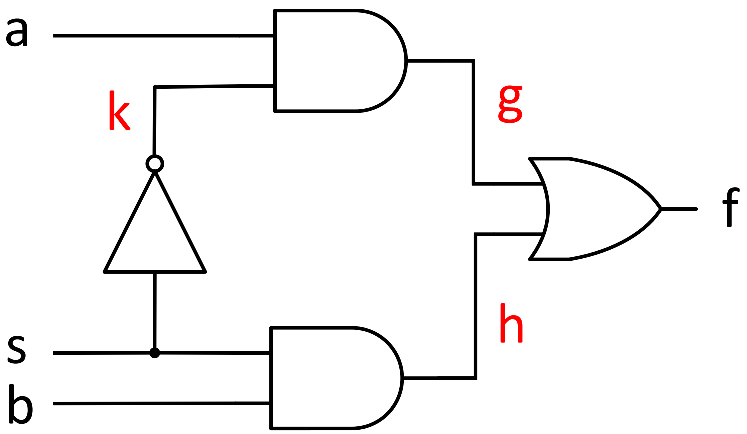
Behavioral specification of a multiplexer.




```
module Mux2To1F(
        input s, a, b,
        output reg f );

    always @( * )
        if ( s )
            f = b;
        else
            f = a;

endmodule
```

An * means anything referenced. (Let the compiler figure it out.)

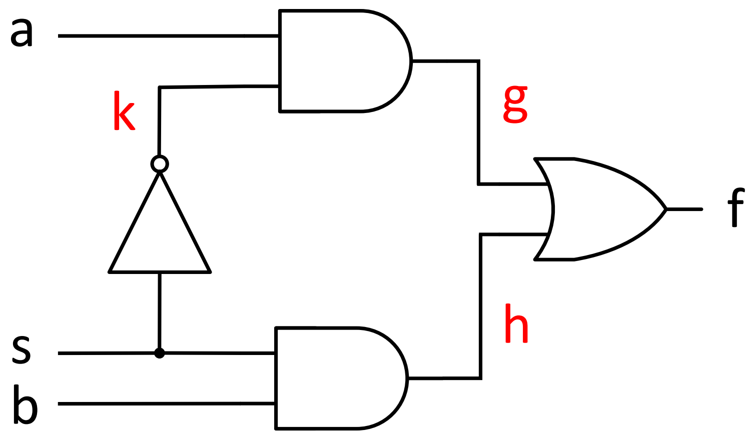
Behavioral specification of a multiplexer.




```
module Mux2To1F(
        input s, a, b,
        output reg f );

    always @( * )
        if ( s )
            f = b;
        else
            f = a;

endmodule
```

This is not software.
The compiler
understands it
should create a
circuit that behaves
this way,

Behavioral specification of a multiplexer.




```
module Mux2To1F(
        input s, a, b,
        output reg f );

    always @( * )
        if ( s )
            f = b;
        else
            f = a;

endmodule
```

A reg variable holds the last value assigned to it.

Behavioral specification of a multiplexer.

Verilog language details

Literals

[*size*] ['*radix*]*constant*

Size is in number of ***bits***. Default is 32 bits.

radix is the number base. Default is decimal.

d	decimal
b	binary
h	hexadecimal
o	octal

Each bit in the *constant* can have 1 of 4 values. Underscores can be inserted for readability.

0	logic value 0
1	logic value 1
z	tri-state (high impedance)
x	unknown

Examples:

```
1
4'hF
32'h2
128
16'd512
5
4'b01xz
16'b0000_0001_0101_1000
```

Identifiers

Any combination of A-Z, a-z, 0-9, _ and $.

Cannot start with 0-9.

Cannot be a Verilog keyword.

Case sensitive.

Values

Scalars: One bit wide.

Vectors: Strings of any number of bits.

Vectors

Square brackets to specify the range, i.e., the numbering of the bits.

Brackets and bit numbers go:

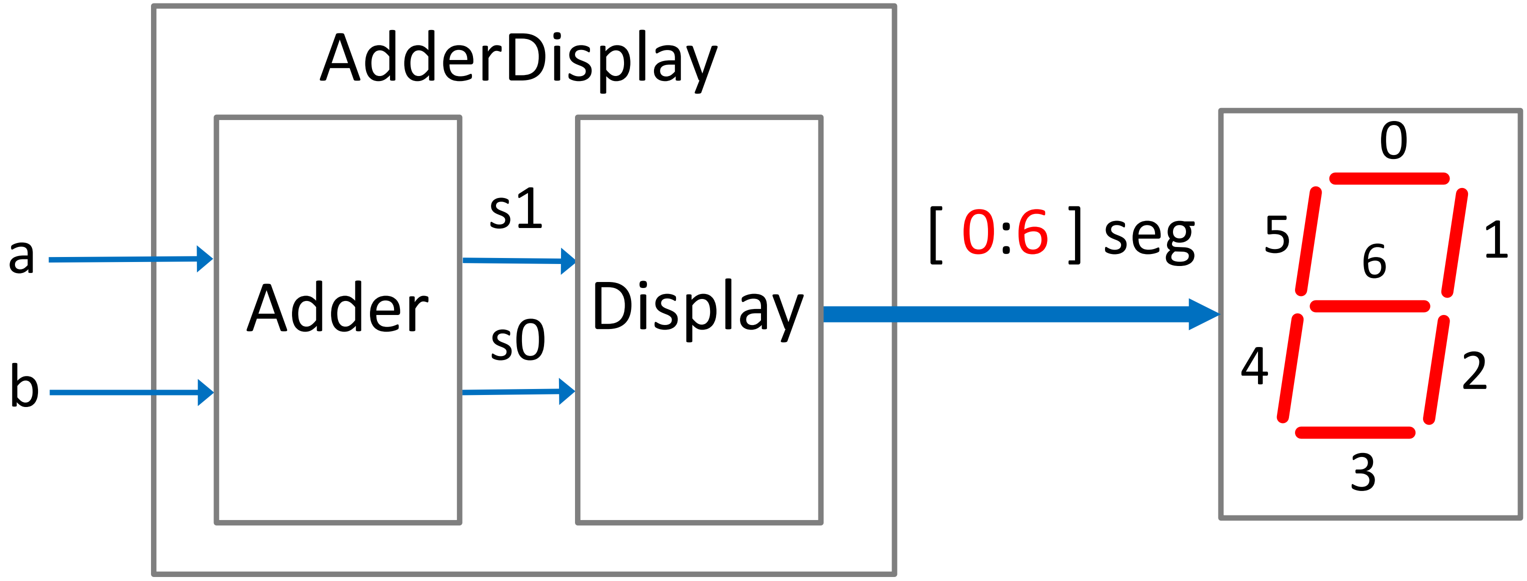
1. Before the name to indicate the size in a definition.
2. After the name when indexing.

Numbering can go up or down and the limits can be either negative or positive.

Examples:

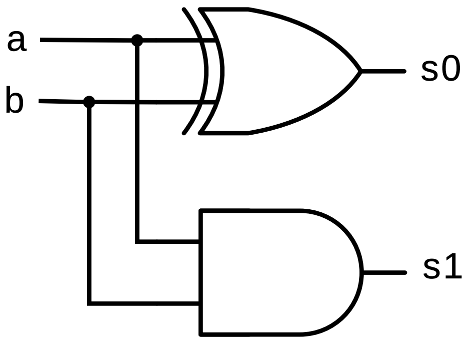
```
input [ -8:-15 ] A;
wire [ 3:0 ] lowPart;
wire lowBit;
assign lowPart = A[ -12:-15 ];
assign lowBit = lowPart[ 0 ];
```

You can combine modules to create new modules.



a	0	0	1	1
+b	+0	+1	+0	+1
s1 s0	0 0	0 1	0 1	1 0

a	b	s1	s0
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0



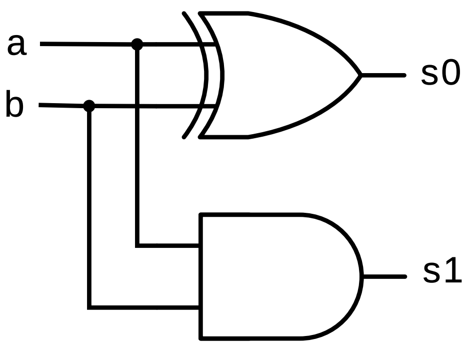

```
module Adder( input a, b,
       output s1, s0 );

    assign s1 = a & b;
    assign s0 = a ^ b;

endmodule
```

A one-bit adder in Verilog.

a	0	0	1	1
+b	+0	+1	+0	+1
s1 s0	0 0	0 1	0 1	1 0

a	b	s1	s0
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

```
module Adder2( input a, b,
      output s1, s0 );

   assign { s1, s0 } = a + b;

endmodule
```

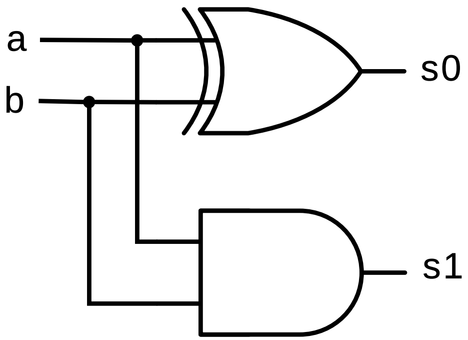


A one-bit adder in Verilog.

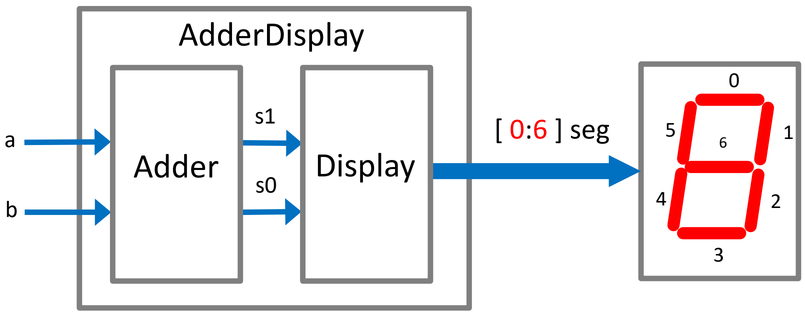
a	0	0	1	1
+b	+0	+1	+0	+1
s1 s0	0 0	0 1	0 1	1 0

a	b	s1	s0
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

```
module Adder3( input a, b,
      output [ 1:0 ] s );

   assign s = a + b;

endmodule
```



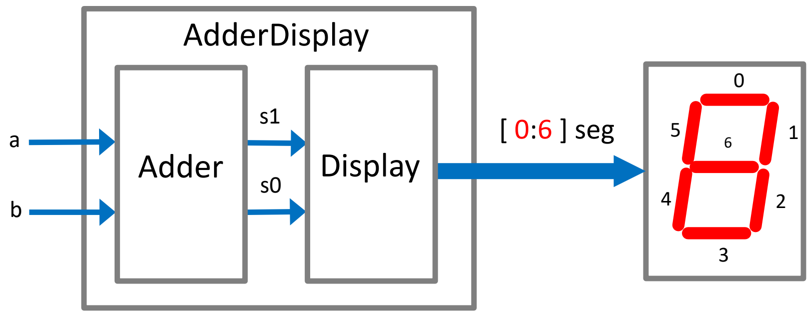
A one-bit adder in Verilog.




```
module Display( input s1, s0,
      output [ 0:6 ] seg );

   // Only works for 0, 1 or 2.

   assign seg[ 0 ] = ~s0,
          seg[ 1 ] = 1,
          seg[ 2 ] = ~s1,
          seg[ 3 ] = ~s0,
          seg[ 4 ] = ~s0,
          seg[ 5 ] = ~s1 & ~s0,
          seg[ 6 ] = s1 & ~s0;

endmodule
```

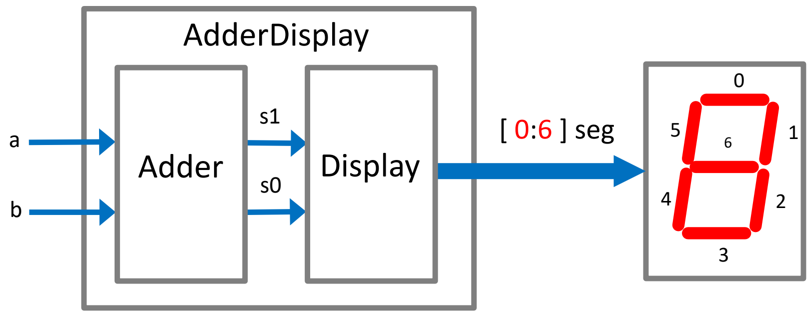
Display	s1 s0	seg[0:6]
0	0 0	1 1 1 1 1 1 0
1	0 1	0 1 1 0 0 0 0
2	1 0	1 1 0 1 1 0 1

A very simple display driver in Verilog.




```
module Display( input s1, s0,
       output [ 0:6 ] seg );

    // Use concatenation.

    assign seg = { ~s0,
                   1,
                   ~s1,
                   ~s0,
                   ~s0,
                   ~s1 & ~s0,
                   s1 & ~s0 };

endmodule
```

Display	s1 s0	seg[0:6]
0	0 0	1 1 1 1 1 1 0
1	0 1	0 1 1 0 0 0 0
2	1 0	1 1 0 1 1 0 1




```
module AdderDisplay( input a, b,
      output [ 0:6 ] seg);

   wire s1, s0;
   Adder U1( a, b, s1, s0 );
   Display U2( s1, s0, seg );

endmodule
```

Hierarchical Verilog code for the AdderDisplay.

Logical Function Unit

Create a unit that can compute the AND, OR, or XOR of two inputs A and B, based upon control lines C0 and C1.

```verilog
module ALU1( input A, B, C0, C1,
      output f );

   // What does this do for each combination
   // of C1 and C0?

   assign f = C1 ? A ^ B : C0 ? A | B : A & B;

endmodule
```

```verilog
module ALU2( input A, B, C0, C1,
      output reg f );

   always @( * )
      if ( C1 )
         f = A ^ B;
      else
         if ( C0 )
            f = A | B;
         else
            f = A & B;

endmodule
```

```verilog
module ALU3( input A, B, C0, C1,
      output reg f );

   always @( * )
      case ( { C1, C0 } )
         2'b00: f = A & B;
         2'b01: f = A | B;
         2'b10: f = A ^ B;
         // what about 2'b11?
      endcase

endmodule
```

```
module ALU3( input A, B, C0, C1,
      output reg f );

   always @( * )
      case ( { C1, C0 } )
         2'b00: f = A & B;
         2'b01: f = A | B;
         2'b10: f = A ^ B;
         // what about 2'b11?
      endcase

endmodule
```

This forces the compiler to assume that in the 2’b11 case, f should retain its present value, which it can only do by adding memory, turning this into a sequential machine.

This is called *implied memory.*

```verilog
module ALU4( input A, B, C0, C1,
      output reg f );

   always @( * )
      case ( { C1, C0 } )
         2'b00: f = A & B;
         2'b01: f = A | B;
         2'b10: f = A ^ B;
         2'b11: f = A ^ B;
      endcase

endmodule
```

```verilog
module ALU5( input A, B, C0, C1,
      output reg f );

   always @( * )
      casex ( { C1, C0 } )
         2'b00: f = A & B;
         2'b01: f = A | B;
         2'b1x: f = A ^ B;
      endcase

endmodule
```

Chapter 3

Number Representation and Arithmetic Circuits

Binary numbers

Unsigned numbers

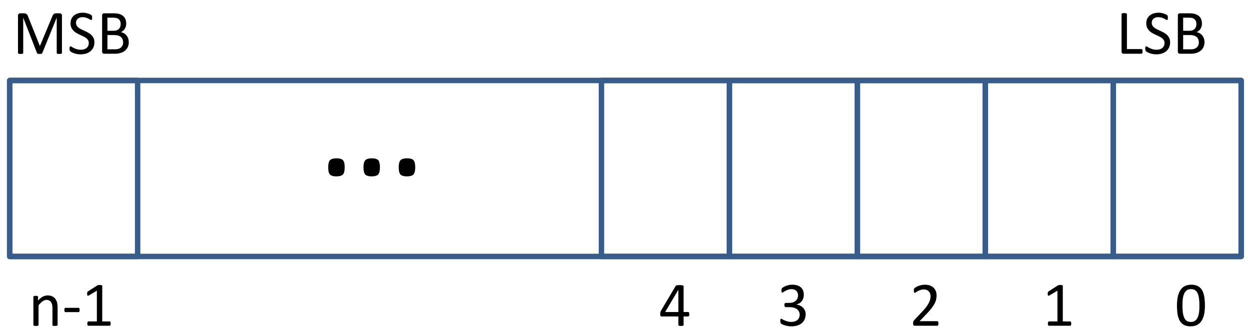
- All bits represent the magnitude of a positive integer

Signed numbers

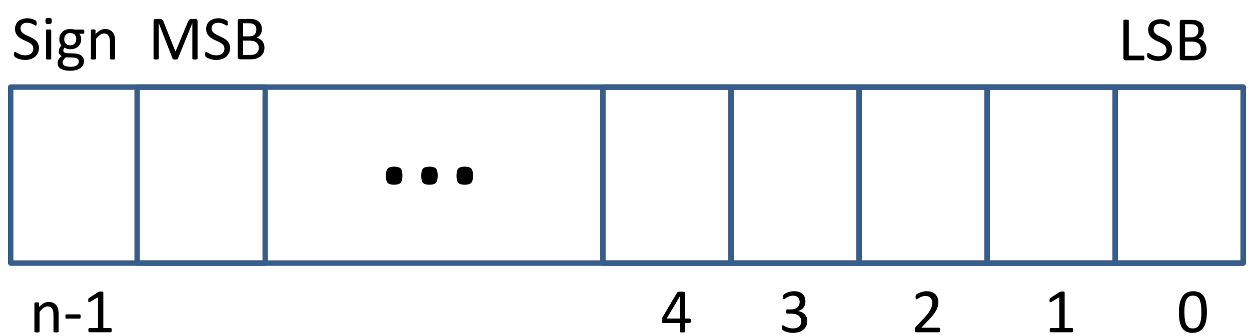
- Left-most bit represents the sign.

Negative Numbers

- Need an efficient way to represent negative numbers in binary
  - Both positive & negative numbers will be strings of bits
  - Use fixed-width formats (4-bit, 16-bit, etc.)
- Must provide efficient mathematical operations
  - Addition & subtraction with potentially mixed signs
  - Negation (multiply by -1)



Unsigned binary



Signed binary

Negative numbers can be represented in following ways:

Sign + magnitude

1's complement

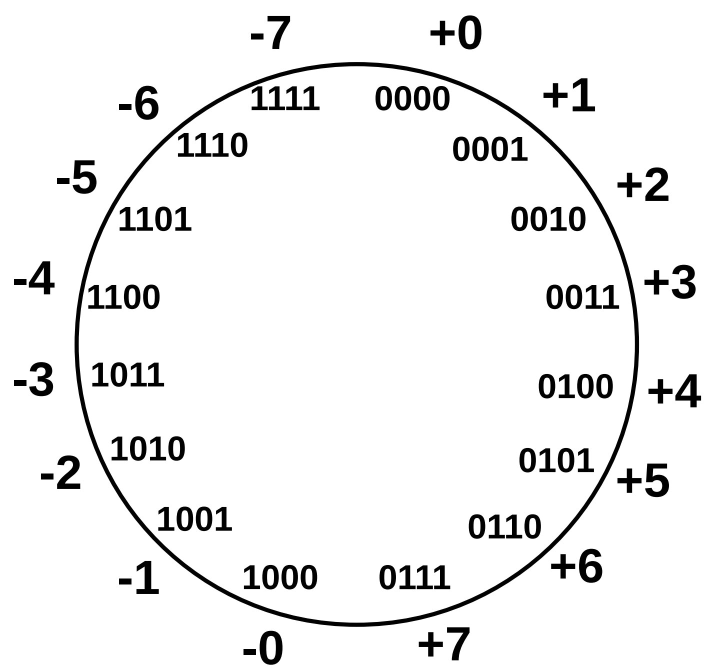
2's complement

$b_3b_2b_1b_0$	Sign and magnitude	1's complement	2's complement
0111	+7	+7	+7
0110	+6	+6	+6
0101	+5	+5	+5
0100	+4	+4	+4
0011	+3	+3	+3
0010	+2	+2	+2
0001	+1	+1	+1
0000	+0	+0	+0
1000	−0	−7	−8
1001	−1	−6	−7
1010	−2	−5	−6
1011	−3	−4	−5
1100	−4	−3	−4
1101	−5	−2	−3
1110	−6	−1	−2
1111	−7	−0	−1

Table 3.2. Interpretation of four-bit signed integers.

Sign + magnitude

$b_3b_2b_1b_0$	Sign and magnitude
0111	+7
0110	+6
0101	+5
0100	+4
0011	+3
0010	+2
0001	+1
0000	+0
1000	−0
1001	−1
1010	−2
1011	−3
1100	−4
1101	−5
1110	−6
1111	−7



Sign + magnitude

$b_3b_2b_1b_0$	Sign and magnitude
0111	+7
0110	+6
0101	+5
0100	+4
0011	+3
0010	+2
0001	+1
0000	+0
1000	−0
1001	−1
1010	−2
1011	−3
1100	−4
1101	−5
1110	−6
1111	−7

The first bit is the sign (+ or -) and the rest of the bits are the value as a positive binary number.

For example, in 4-bit sign + magnitude:

+5 = 0101
-5 = 1101

Sign + magnitude addition

```
  0 0 1 0  (+2)
+ 0 1 0 0  (+4)
```

```
  1 0 1 0  (-2)
+ 1 1 0 0  (-4)
```

```
  0 0 1 0  (+2)
+ 1 1 0 0  ( -4)
```

```
  1 0 1 0  ( -2)
+ 0 1 0 0  (+4)
```

Sign + magnitude addition

```
    0 0 1 0  (+2)
  + 0 1 0 0  (+4)
  ---------
    0 1 1 0  (+6)
```

```
    1 0 1 0  (-2)
  + 1 1 0 0  (-4)
  ---------
    0 1 1 0  (+6)
```

```
    0 0 1 0  (+2)
  + 1 1 0 0  ( -4)
  ---------
    1 1 1 0  (-2)
```

```
    1 0 1 0  ( -2)
  + 0 1 0 0  (+4)
  ---------
    1 1 1 0  ( -6)
```

Adding with sign + magnitude

$b_3b_2b_1b_0$	Sign and magnitude
0111	+7
0110	+6
0101	+5
0100	+4
0011	+3
0010	+2
0001	+1
0000	+0
1000	−0
1001	−1
1010	−2
1011	−3
1100	−4
1101	−5
1110	−6
1111	−7

If both operands have the same sign, adding works.

0010	(+2)
+ 0011	(+3)
0101	(+5)

1010	(-2)
+ 1011	(-3)
1101	(-5)

Problem with sign + magnitude

$b_3b_2b_1b_0$	Sign and magnitude
0111	+7
0110	+6
0101	+5
0100	+4
0011	+3
0010	+2
0001	+1
0000	+0
1000	−0
1001	−1
1010	−2
1011	−3
1100	−4
1101	−5
1110	−6
1111	−7

But if the signs are different, it doesn't work.

```
  1010     (-2)
+ 0011     (+3)
  1101     (-5)  Wrong
```

Must compare and subtract the smaller from the larger and use the sign of the larger for the result.

```
   011     (+3)
-  010     (-2 w/o the sign)
   001
```

1’s complement

$b_3b_2b_1b_0$	1’s complement
0111	+7
0110	+6
0101	+5
0100	+4
0011	+3
0010	+2
0001	+1
0000	+0
1000	−7
1001	−6
1010	−5
1011	−4
1100	−3
1101	−2
1110	−1
1111	−0

The first bit is the sign (+ or -) and the rest of the bits are the value as a binary number if it’s positive or with the bits inverted if it’s negative.

For example, in 4-bit 1’s complement:

+5 = 0101
-5 = 1010

Notice that 0 has two values: 0000 (+0) and 1111 (-0).

Adding in 1's complement

$b_3b_2b_1b_0$	1's complement
0111	+7
0110	+6
0101	+5
0100	+4
0011	+3
0010	+2
0001	+1
0000	+0
1000	−7
1001	−6
1010	−5
1011	−4
1100	−3
1101	−2
1110	−1
1111	−0

If both operands are positive, adding works, not other wise.

0010	(+2)	
+ 0011	(+3)	
0101	(+5)	
1101	(-2)	
+ 1100	(-3)	
1001	(-6)	Wrong

Adding in 1's complement

$b_3b_2b_1b_0$	1's complement
0111	+7
0110	+6
0101	+5
0100	+4
0011	+3
0010	+2
0001	+1
0000	+0
1000	−7
1001	−6
1010	−5
1011	−4
1100	−3
1101	−2
1110	−1
1111	−0

If either operand is negative, it's off by one because when there is an overflow, you cross two zeros, 1111 and 0000.

```
   1101  (-2)         1101  (-2)
 + 0011  (+3)       + 1100  (-3)
 ------             ------
   0000               1001  (-6)
```

Correct by adding the overflow.

```
   1101  (-2)         1101  (-2)
 + 0011  (+3)       + 1100  (-3)
 ------             ------
  10000              11001
 +    1             +    1
 ------             ------
   0001  (+1)         0001  (-6)
```

1's complement

$b_3b_2b_1b_0$	1's complement
0111	$+7$
0110	$+6$
0101	$+5$
0100	$+4$
0011	$+3$
0010	$+2$
0001	$+1$
0000	$+0$
1000	-7
1001	-6
1010	-5
1011	-4
1100	-3
1101	-2
1110	-1
1111	-0

Let K be the negative equivalent of an n-bit positive number P.

The 1's complement representation of K is:

$$K = (2^n - 1) - P$$

This means that K can be obtained by inverting all bits of P.

$$\begin{array}{r} (+5) \\ +\,(+2) \\ \hline (+7) \end{array} \qquad \begin{array}{r} 0101 \\ +\,0010 \\ \hline 0111 \end{array} \qquad\qquad \begin{array}{r} (-5) \\ +\,(+2) \\ \hline (-3) \end{array} \qquad \begin{array}{r} 1010 \\ +\,0010 \\ \hline 1100 \end{array}$$

Two values of 0:

+0 = 0000

-0 = 1111

$$\begin{array}{r} (+5) \\ +\,(-2) \\ \hline (+3) \end{array} \qquad \begin{array}{r} 0101 \\ +\,1101 \\ \hline 10010 \\ 1 \\ \hline 0011 \end{array} \qquad\qquad \begin{array}{r} (-5) \\ +\,(-2) \\ \hline (-7) \end{array} \qquad \begin{array}{r} 1010 \\ +\,1101 \\ \hline 10111 \\ 1 \\ \hline 1000 \end{array}$$

Overflow means you crossed over 2 zeros.

Figure 3.8. Examples of 1's complement addition.