BEE 271 Digital circuits and systems

Spring 2017

Lecture 8: Multiplexers and Shannon's expansion

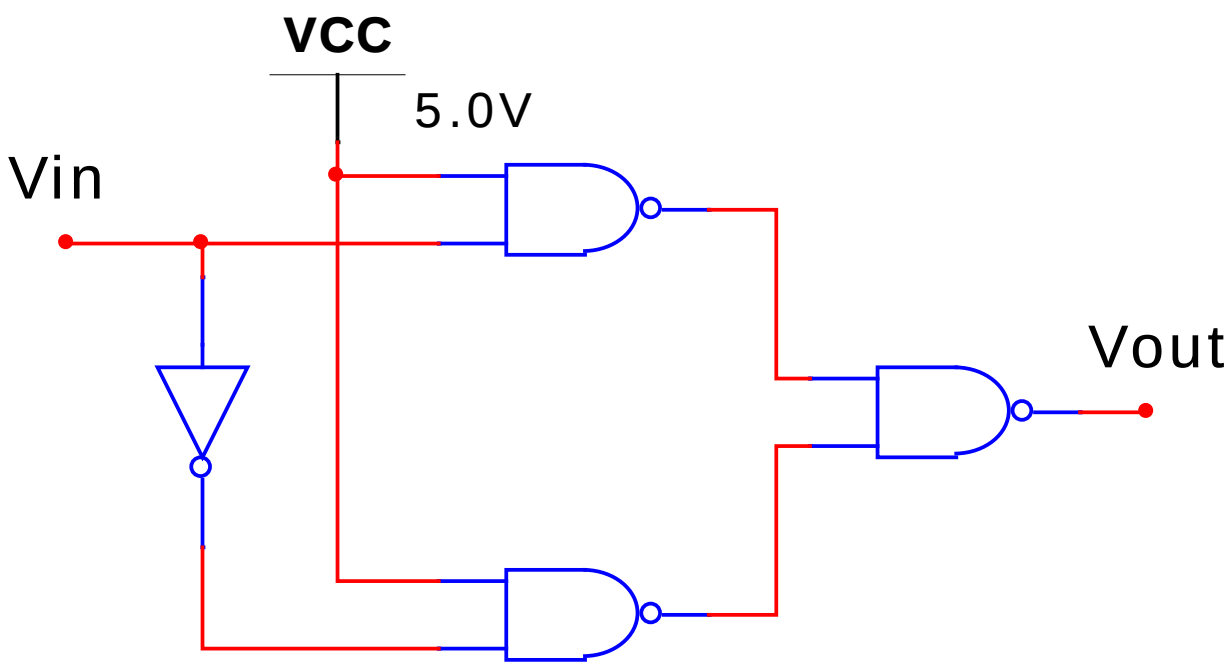
Nicole Hamilton

https://faculty.washington.edu/kd1uj

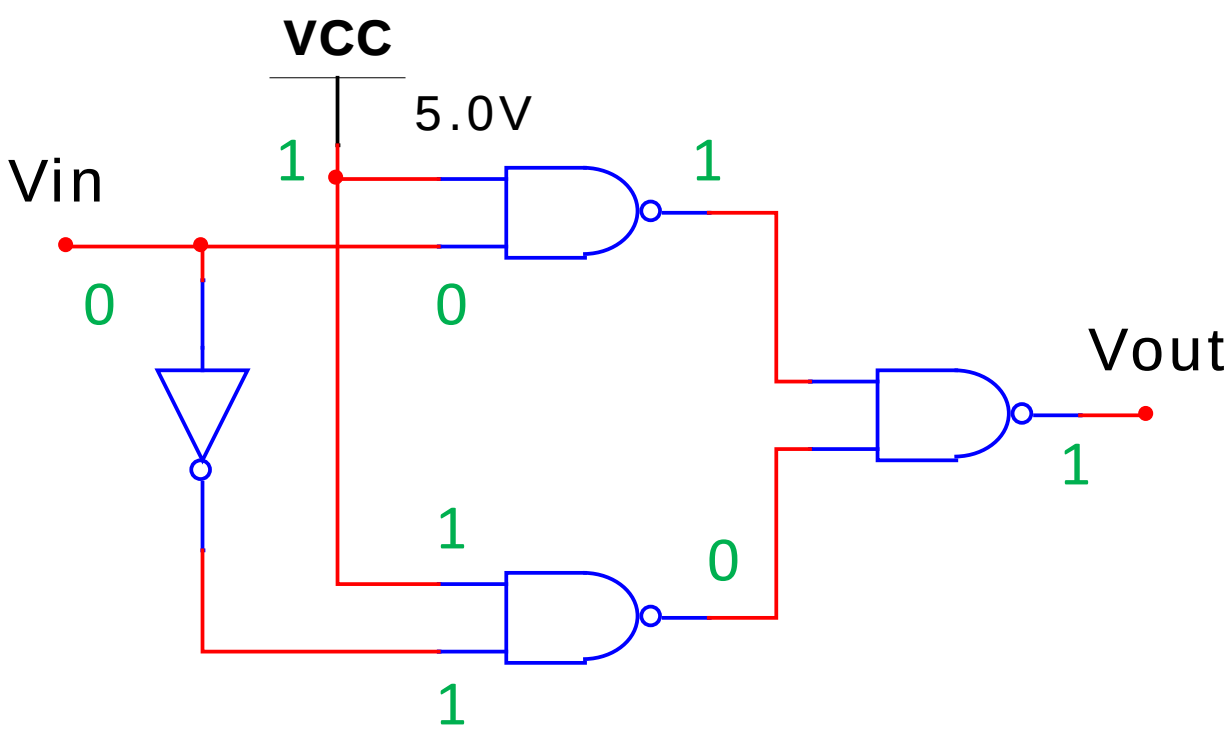
Topics

1. Hazards
2. Multiplexers
3. Shannon’s expansion

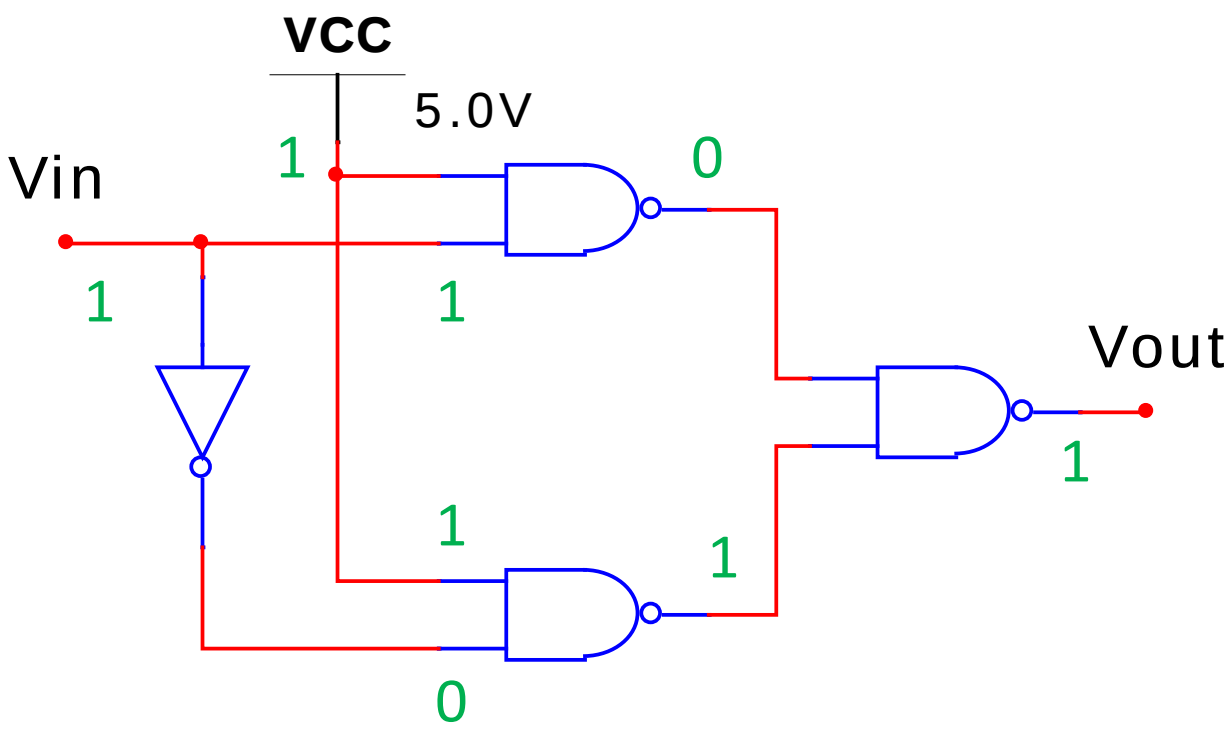
A circuit with a hazard

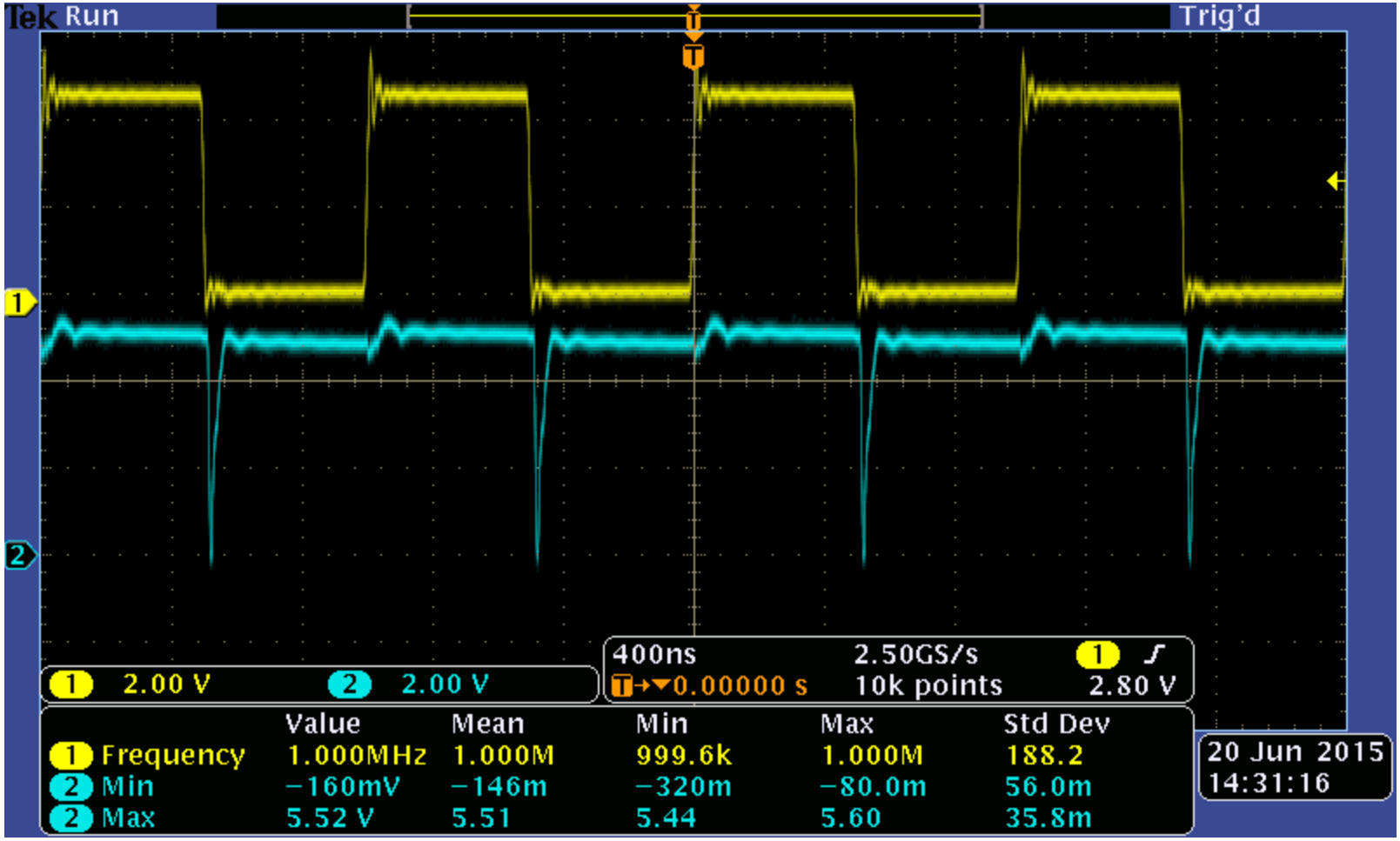


If Vin = 0, Vout = 1

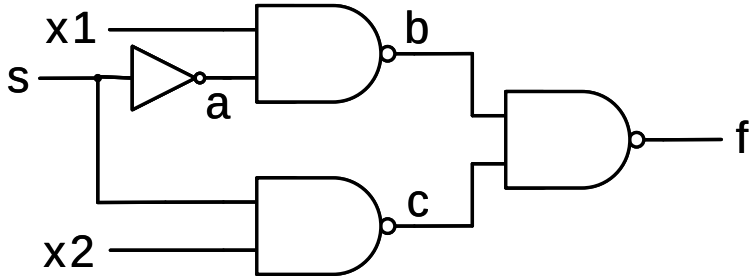


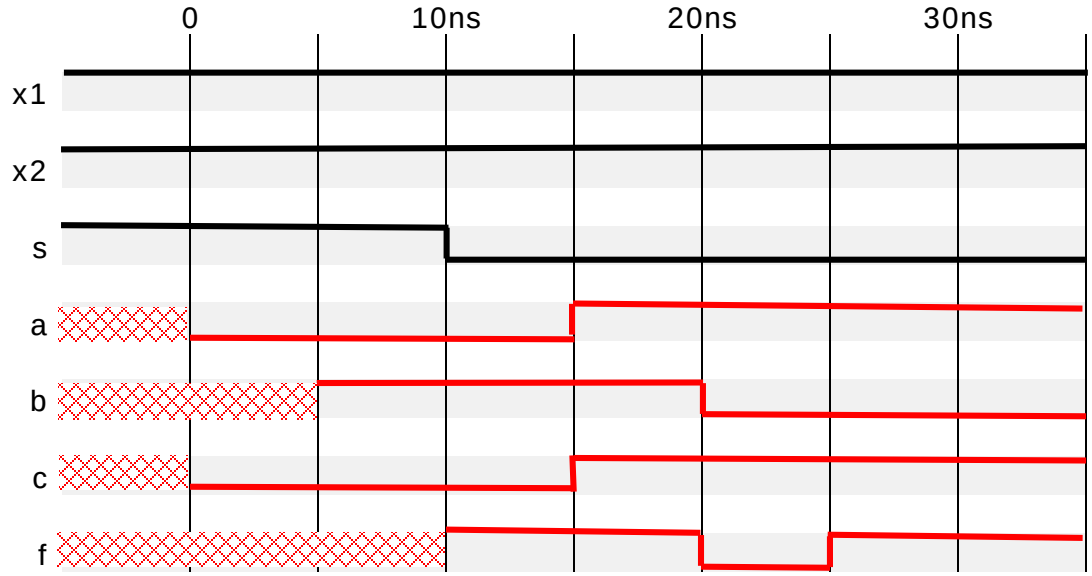
If Vin = 1, Vout = 1 (same)

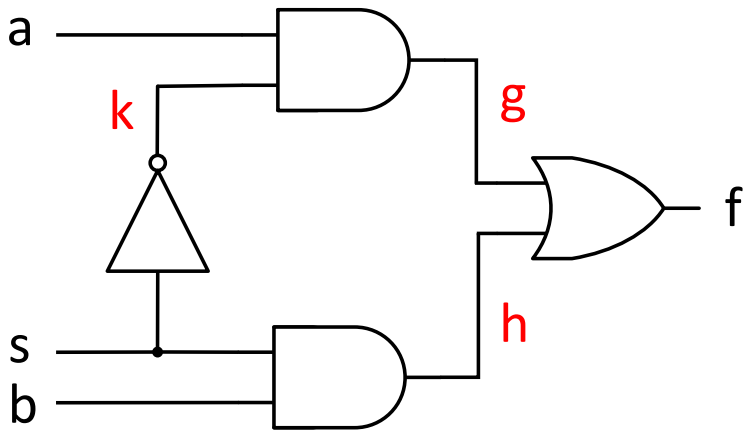



Tek Run
Trig'd
1
2
1 2.00 V
2 2.00 V
400ns
2.50GS/s
1
0.00000 s
10k points
2.80 V
Value Mean Min Max Std Dev
1 Frequency 1.000MHz 1.000M 999.6k 1.000M 188.2
2 Min −160mV −146m −320m −80.0m 56.0m
2 Max 5.52 V 5.51 5.44 5.60 35.8m
20 Jun 2015
14:31:16

From homework 2

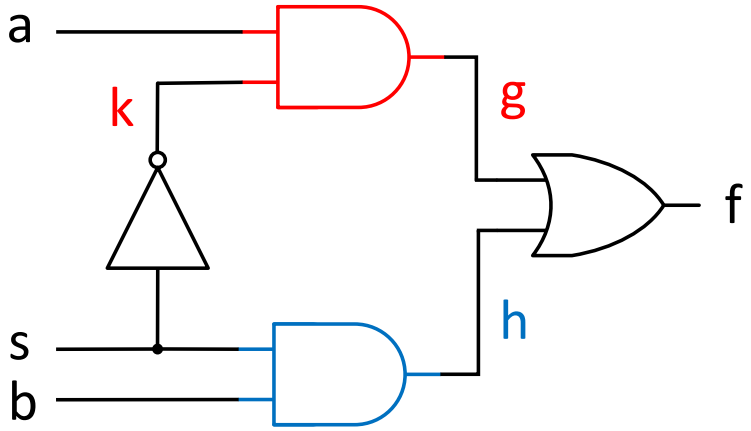







```
module Mux2To1wHazard(
       input  s, a, b,
       output f );

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

endmodule
```

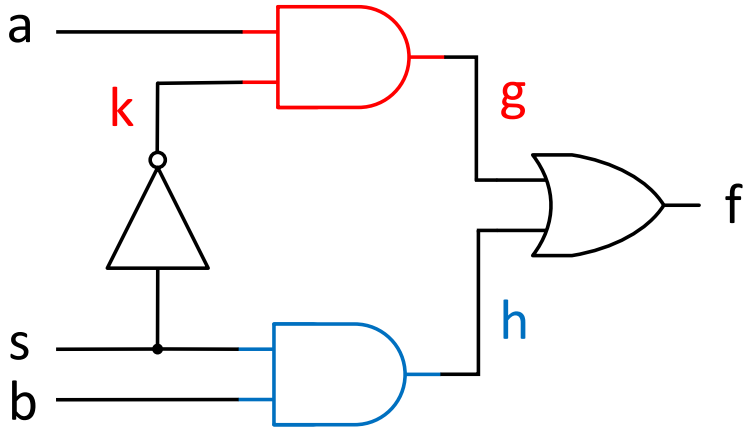
A circuit with a hazard



		a b				
		00	01	11	10	
s	0	0	0	1	1	g = a s’
	1	0	1	1	0	

h = b s

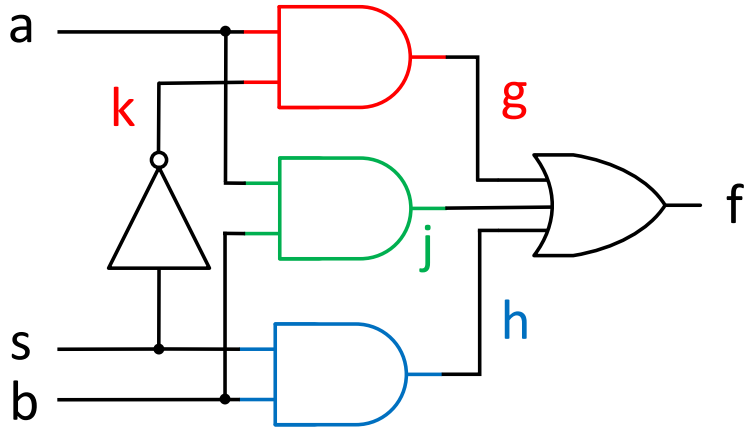
The hazard happens because we’re switching between implicants that can never be assumed to have exactly identical path delays.



		a b				
		00	01	11	10	
s	0	0	0	1	1	g = a s’
	1	0	1	1	0	

h = b s

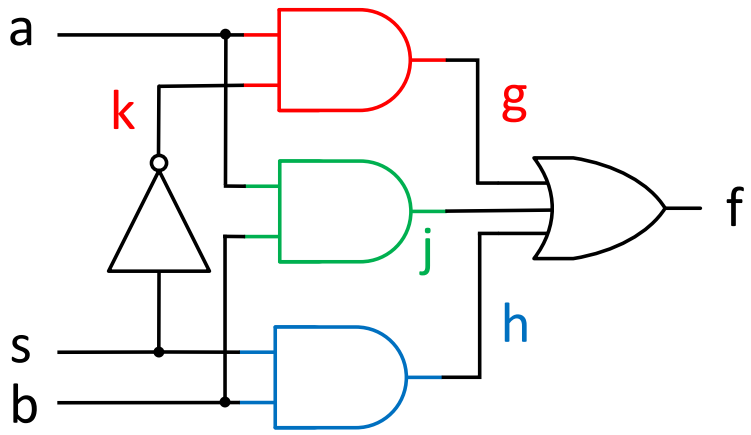
It’s called a *static one hazard* because we expect the output to remain 1 if a and b are both 1 and it doesn’t.



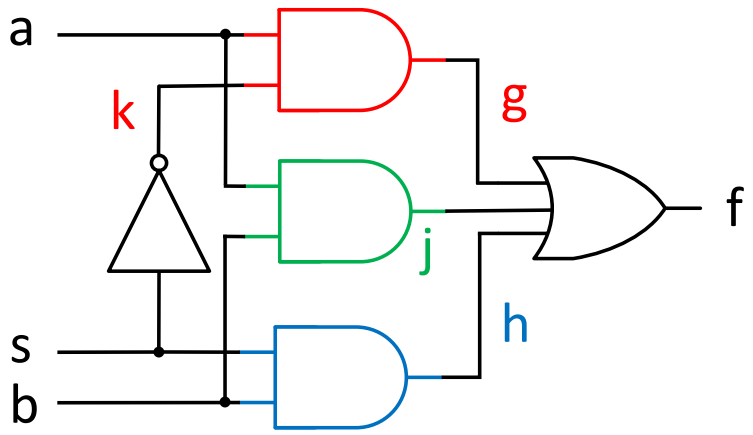
		a b: 00	01	11	10	
s	0	0	0	1	1	g = a s’
	1	0	1	1	0	

h = b s j = a b

We can eliminate the hazard by adding one more term so that if a and b are both 1, f = 1 no matter what the value of s.



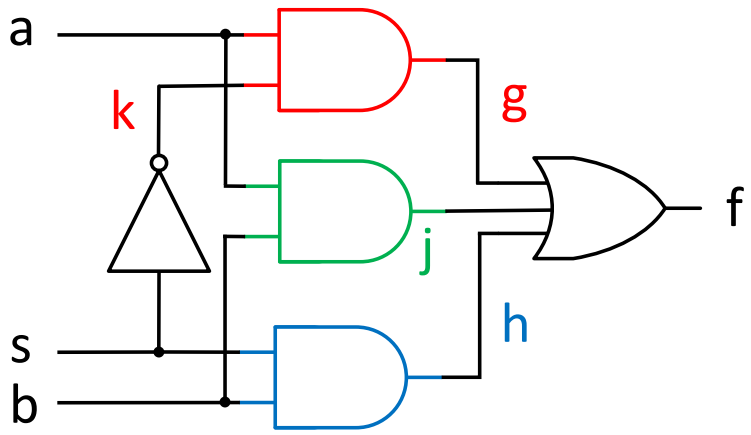

```
module Mux2To1HazardFree(
       input  x1, x2, s,
       output f );

   wire g, h, j, k;
   not ( k, s );
   and ( g, k, x1 ),
       ( h, s, x2 ),
       ( j, x1, x2 );
   or  ( f, g, h, j );

endmodule
```




```
module Mux2To1B(
      input  x1, x2, s,
      output f );

   assign f = s ? x2 : x1;

endmodule
```

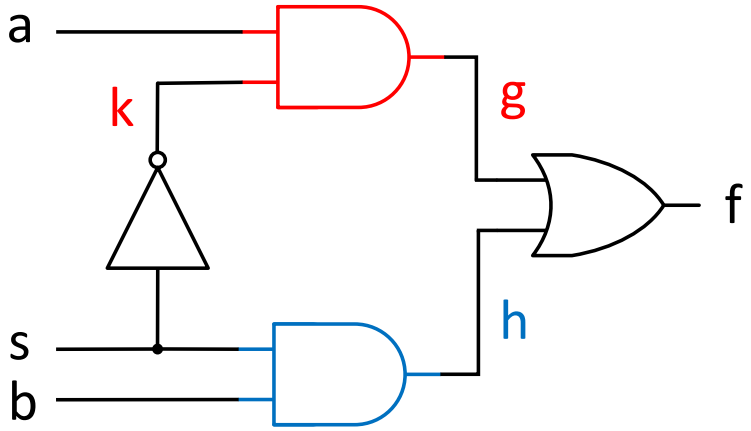
```
module Mux2To1C(
      input  x1, x2, s,
      output f );

   assign f = ~s & x1 | s & x2;

endmodule
```

The Verilog compiler will automatically add the extra gate to make continuous assignments and behavioral descriptions hazard-free.




```
module Mux2To1D(
      input x1, x2, s,
      output reg f );

   always @( * )
      if ( s )
         f = x2;
      else
         f = x1;

endmodule
```

The Verilog compiler will automatically add the extra gate to make continuous assignments and behavioral descriptions hazard-free.

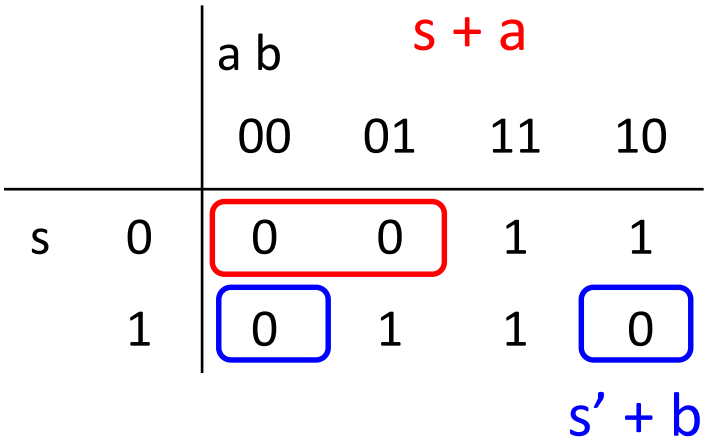


		a b				
		00	01	11	10	
s	0	0	0	1	1	g = a s’
	1	0	1	1	0	

h = b s

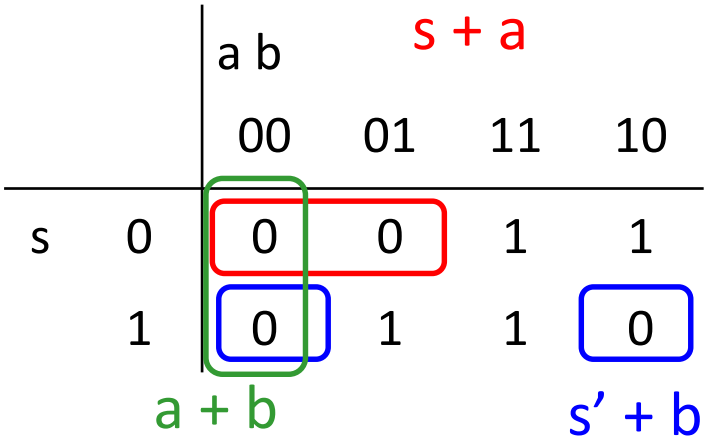
A *static one hazard* happens because you’re collecting 1’s for an SOP solution.

f = (s + a) (s’ + b)



A *static zero hazard* happens you’re collecting 0’s for a POS solution. If a and b are both 0, it will glitch to 1 when switches to the slower implicant.

f = (s + a) (s’ + b) (a + b)

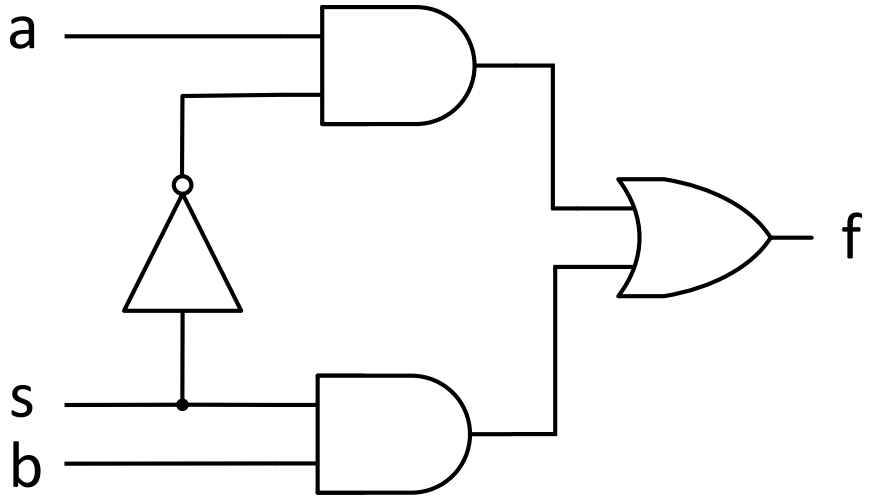


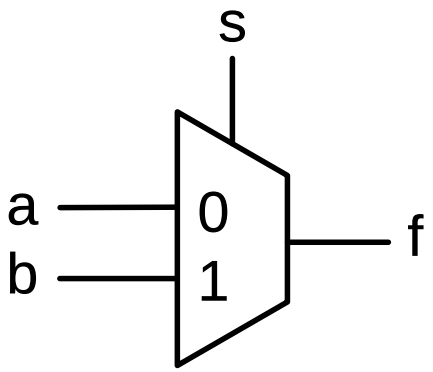
To fix a *static zero hazard*, we add back the extra POS term to ensure that if both a and b = 0, the output will remain 0.

multiplexers and decoders

- A *multiplexer* is a many-to-one function, selecting from a set of inputs (which could be vectors).
- An *encoder* or *decoder* translates from one encoding to another.
  1. Select highest priority.
  2. 4-bit binary to 1-of-16 select.
  3. 4-bit binary to 7-segment display.

The Multiplexer



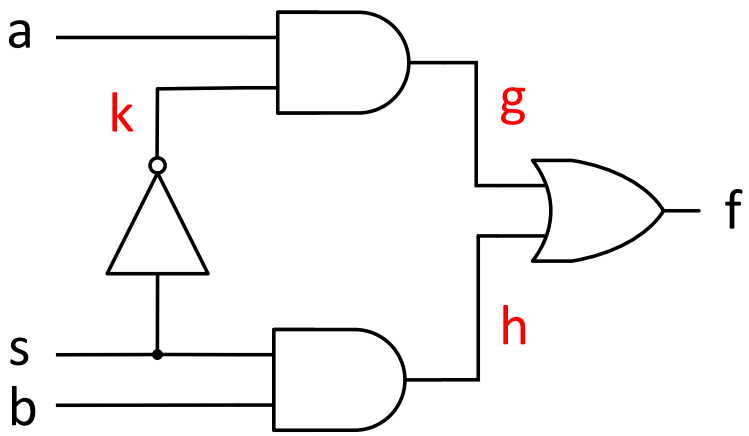


s	f
0	a
1	b

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

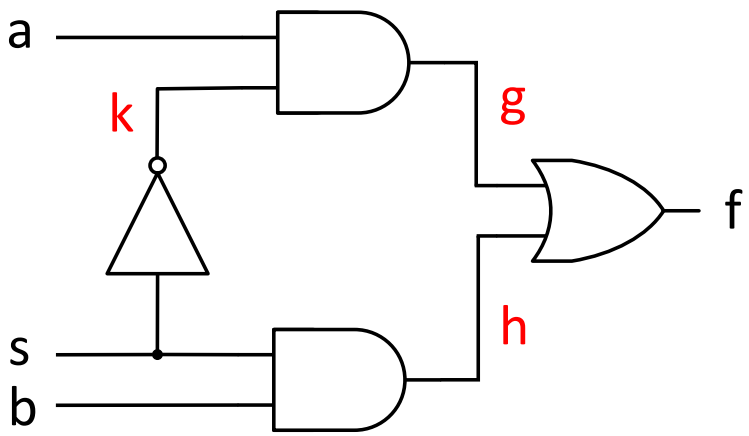
Multiplexer

1. An element that selects data from one of many input lines and directs it to a single output line
2. Input: 2^N input lines and N selection lines
3. Output: The data from *one* selected input line
4. Multiplexer often abbreviated as MUX



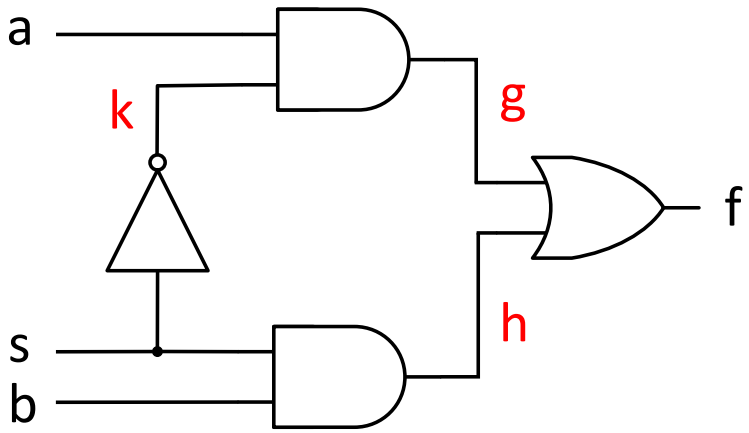

```
module Mux2To1A(
       input  s, a, b,
       output f );

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

endmodule
```

Structural code for a multiplexer.



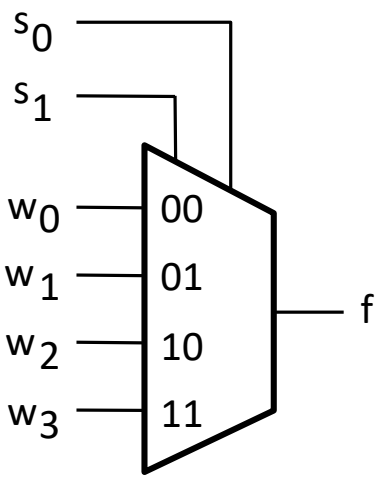

```
module Mux2To1D(
       input  s, a, b,
       output f );

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

endmodule
```

Continuous assignment.




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

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

endmodule
```

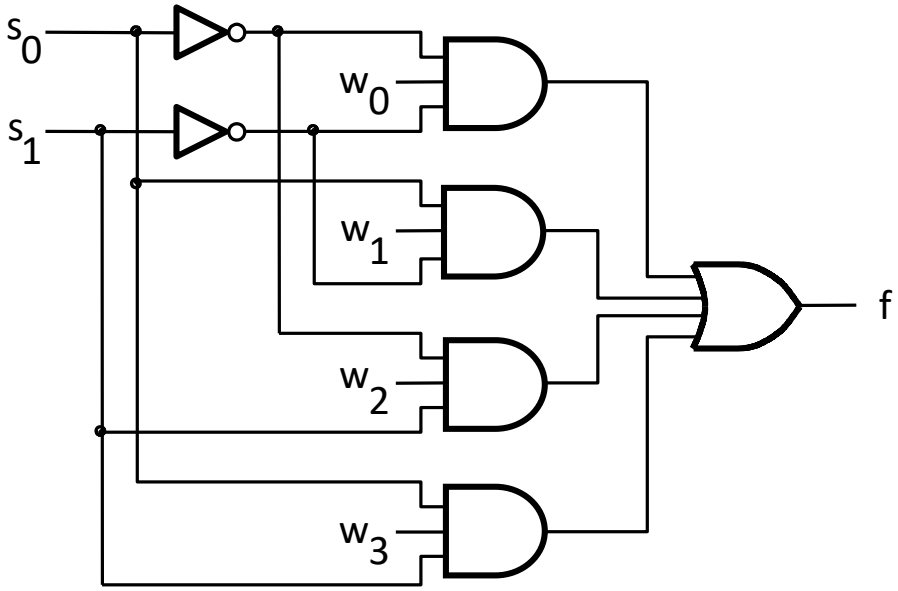
Behavioral description of a multiplexer.



Schematic symbol

s_1	s_0	f
0	0	w_0
0	1	w_1
1	0	w_2
1	1	w_3

Truth table



Circuit

A 4-to-1 multiplexer.

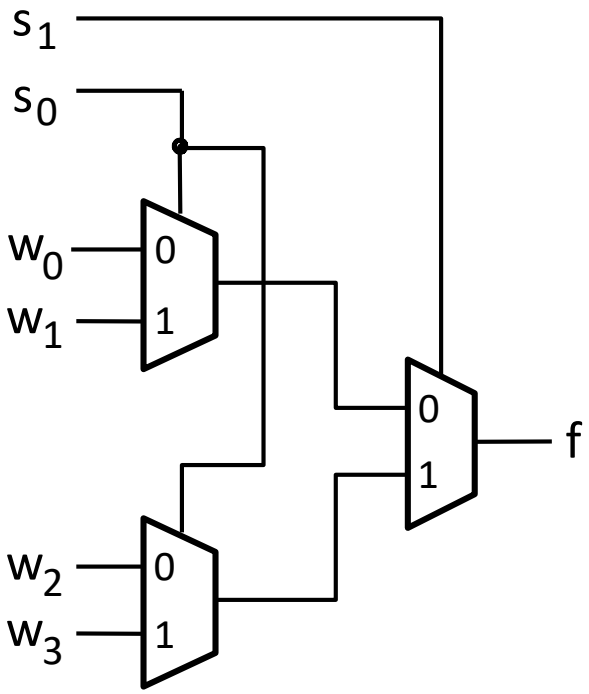
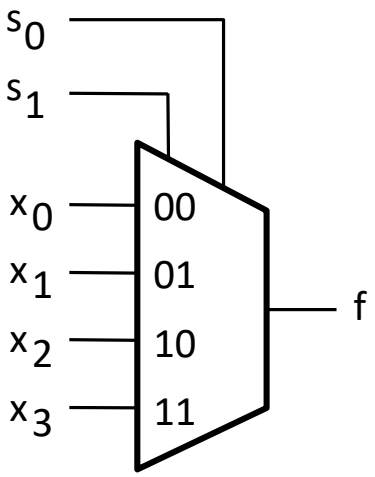


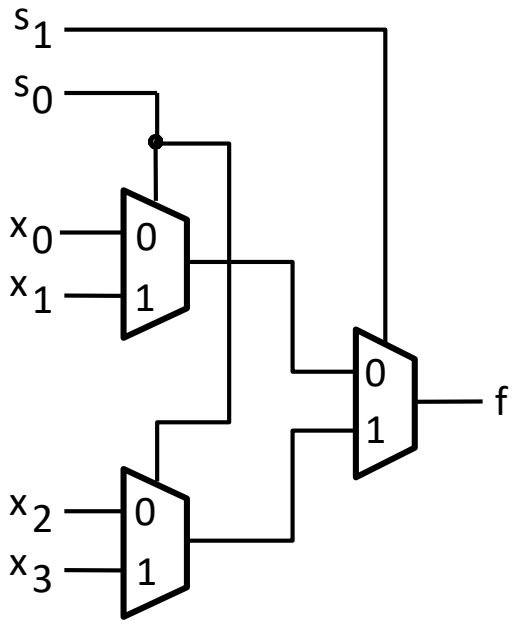
Figure 4.3. Using 2-to-1 multiplexers to build a 4-to-1 multiplexer.

```
module Mux4to1A( input [ 0:3 ] x, input [ 1:0 ] s,
      output f );

   assign f = s == 0 ? x[ 0 ] :
                  s == 1 ? x[ 1 ] :
                     s == 2 ? x[ 2 ] : x[ 3 ];
endmodule
```



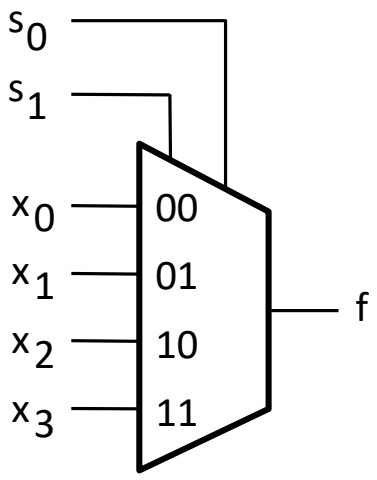
A 4-to-1 multiplexer.

```
module Mux4to1B( input [ 0:3 ] x, input [ 1:0 ] s,
      output f );

   assign f = s[ 1 ] ?
                  s[ 0 ] ? x[ 3 ] : x[ 2 ] :
                  s[ 0 ] ? x[ 1 ] : x[ 0 ];

endmodule
```



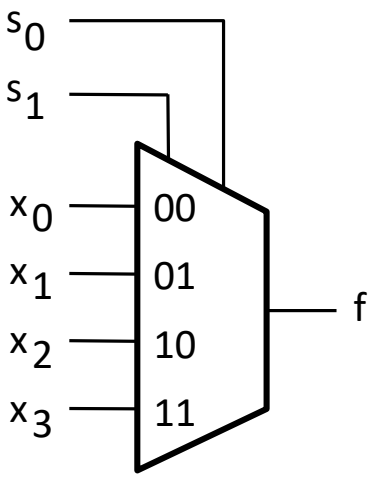
A 4-to-1 multiplexer.

```
module Mux4to1C( input [ 0:3 ] x, input [ 1:0 ] s,
      output reg f );

   always @( * )
      if ( s == 0 )
         f = x[ 0 ];
      else
         if ( s == 1 )
            f = x[ 1 ];
         else
            if ( s == 2 )
               f = x[ 2 ];
            else
               f = x[ 3 ];
endmodule
```



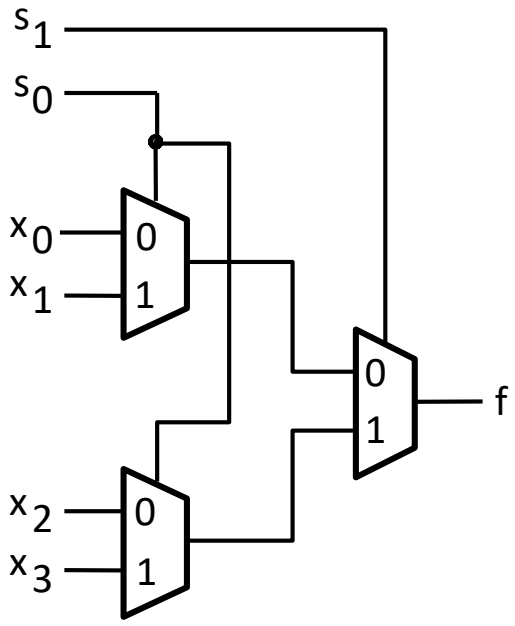
A 4-to-1 multiplexer.

```
module Mux4to1D( input [ 0:3 ] x, input [ 1:0 ] s,
      output reg f );

   always @( * )
      case ( s )
         0: f = x[ 0 ];
         1: f = x[ 1 ];
         2: f = x[ 2 ];
         3: f = x[ 3 ];
      endcase

endmodule
```



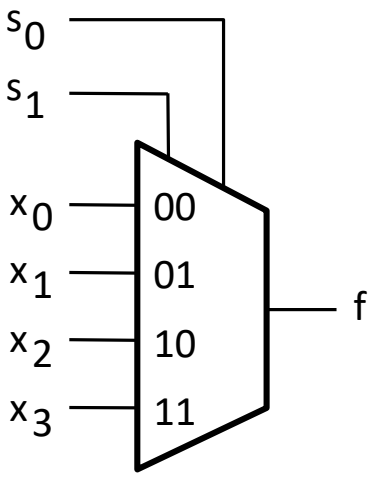
A 4-to-1 multiplexer.

```
module Mux4to1E( input [ 0:3 ] x, input [ 1:0 ] s,
      output f );

   wire a, b;
   Mux2to1A ma ( x[ 0 ], x[ 1 ], s[ 0 ], a );
   Mux2to1A mb ( x[ 2 ], x[ 3 ], s[ 0 ], b );
   Mux2to1A mf ( a,      b,      s[ 1 ], f );

endmodule
```

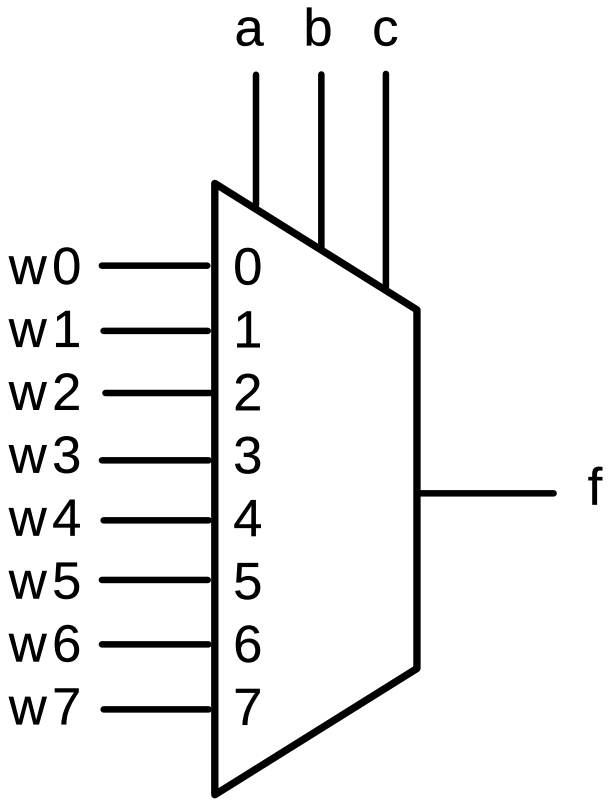


A 4-to-1 multiplexer.

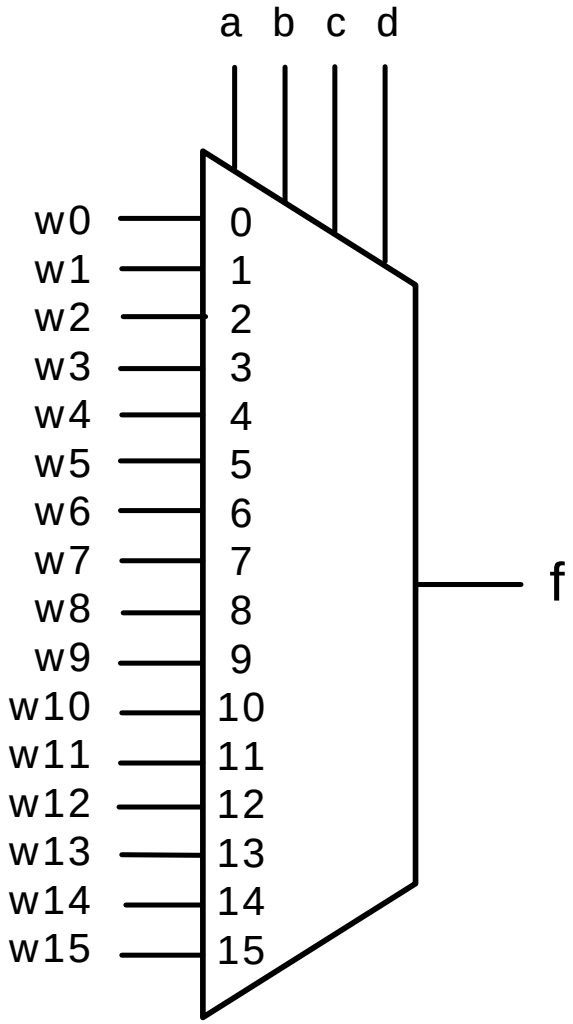
```
module Mux4to1F( input [ 0:3 ] x, input [ 1:0 ] s,
      output f );

   assign f = x[ s ];

endmodule
```



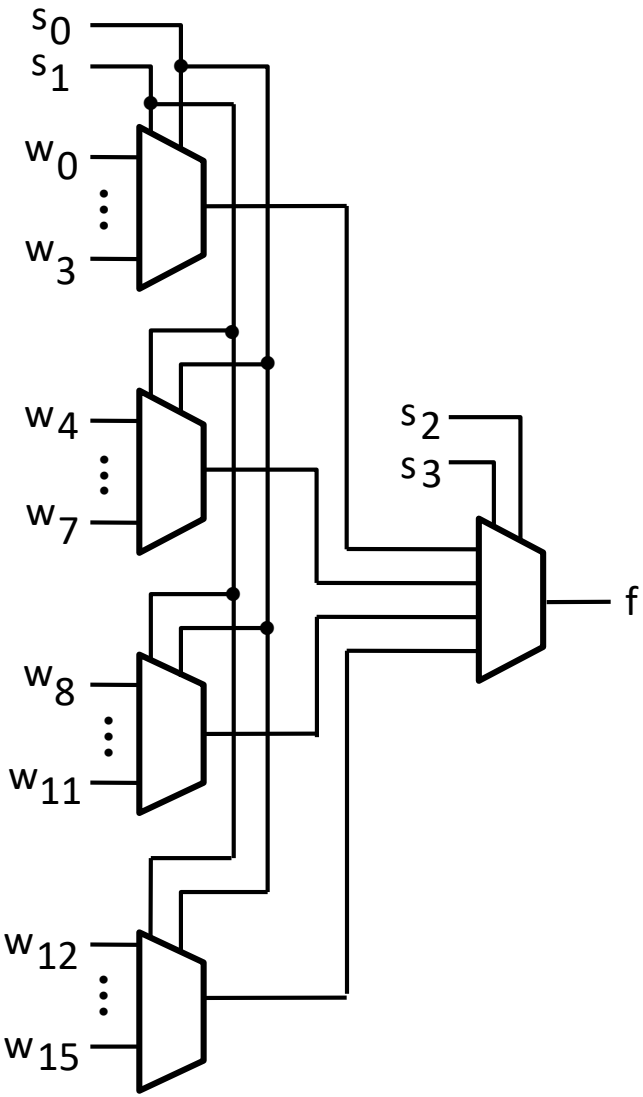
A 4-to-1 multiplexer.



An 8-to-1 multiplexer.



A 16-to-1 multiplexer.



A 16-to-1 multiplexer built from 4-to-1 multiplexers.

```
module Mux16to1A( input [ 0:15 ] w,
      input [ 3:0 ] s, output f );

   wire [ 0:3 ] m;

   Mux4to1 m1 ( w[  0:3  ], s[ 1:0 ], m[ 0 ] );
   Mux4to1 m2 ( w[  4:7  ], s[ 1:0 ], m[ 1 ] );
   Mux4to1 m3 ( w[  8:11 ], s[ 1:0 ], m[ 2 ] );
   Mux4to1 m4 ( w[ 12:15 ], s[ 1:0 ], m[ 3 ] );
   Mux4to1 m5 ( m[  0:3  ], s[ 3:2 ], f      );

endmodule
```

A16-to-1 multiplexer.

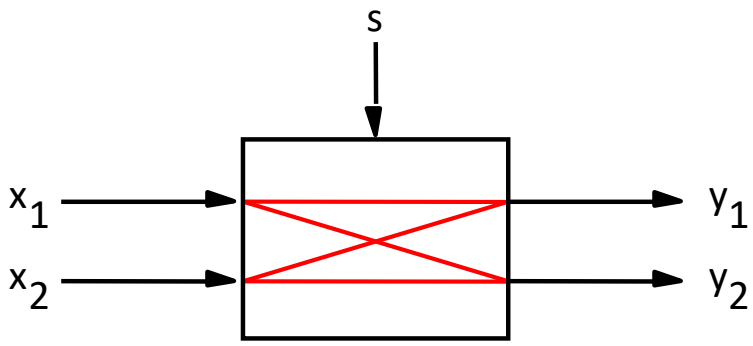
```
module Mux16to1B( input [ 0:15 ] w,
      input [ 3:0 ] s, output reg f );

   always @( * )
      case ( s )
           0: f = w[  0 ];
           1: f = w[  1 ];
           2: f = w[  2 ];
           3: f = w[  3 ];
           4: f = w[  4 ];
           5: f = w[  5 ];
           6: f = w[  6 ];
           7: f = w[  7 ];
           8: f = w[  8 ];
           9: f = w[  9 ];
          10: f = w[ 10 ];
          11: f = w[ 11 ];
          12: f = w[ 12 ];
          13: f = w[ 13 ];
          14: f = w[ 14 ];
          15: f = w[ 15 ];
       endcase

endmodule
```

```
module Mux16to1C( input [ 0:15 ] w,
      input [ 3:0 ] s, output reg f );

   always @( * )
      begin
      if ( s ==  0 ) f = w[  0 ];
      if ( s ==  1 ) f = w[  1 ];
      if ( s ==  2 ) f = w[  2 ];
      if ( s ==  3 ) f = w[  3 ];
      if ( s ==  4 ) f = w[  4 ];
      if ( s ==  5 ) f = w[  5 ];
      if ( s ==  6 ) f = w[  6 ];
      if ( s ==  7 ) f = w[  7 ];
      if ( s ==  8 ) f = w[  8 ];
      if ( s ==  9 ) f = w[  9 ];
      if ( s == 10 ) f = w[ 10 ];
      if ( s == 11 ) f = w[ 11 ];
      if ( s == 12 ) f = w[ 12 ];
      if ( s == 13 ) f = w[ 13 ];
      if ( s == 14 ) f = w[ 14 ];
      if ( s == 15 ) f = w[ 15 ];
      end

endmodule
```

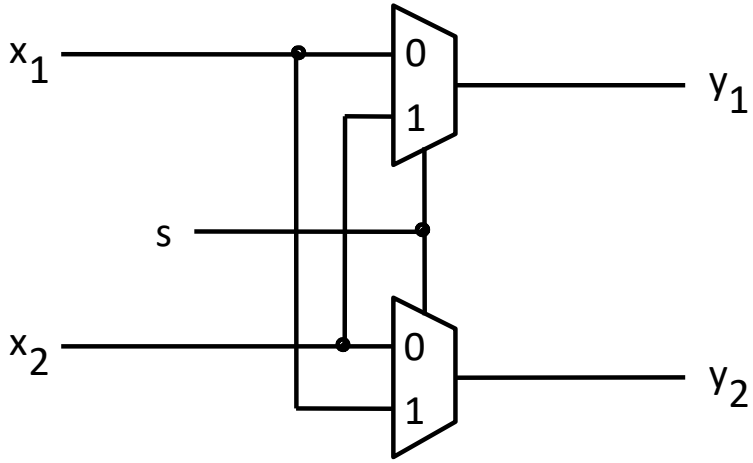
```
module Mux16to1D( input [ 0:15 ] w,
      input [ 3:0 ] s, output reg f );

   always @( * )
      begin
      integer p;
      for ( p = 0;  p < 16;  p = p + 1 )
         if ( s == p )
            f = w[ p ];
      end

endmodule
```

```
module Mux16to1E( input [ 0:15 ] w,
      input [ 3:0 ] s, output f );

   assign f = w[ s ];

endmodule
```

Synthesis of logic functions using multiplexers



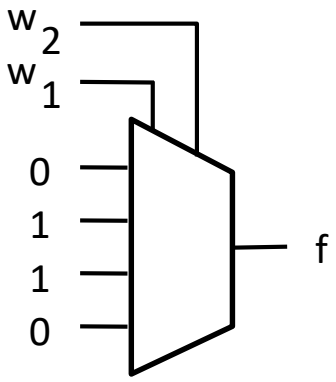
A 2 x 2 crossbar switch



Implementation using multiplexers

A practical application of multiplexers.

w_1	w_2	f
0	0	0
0	1	1
1	0	1
1	1	0

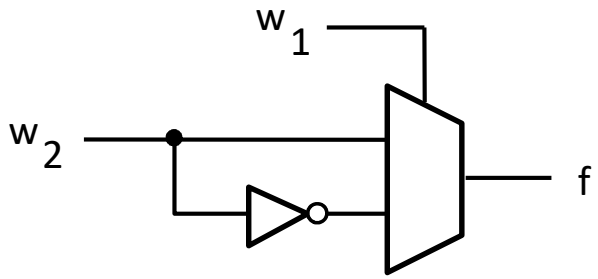


Implementation using a 4-to-1 multiplexer

w_1	w_2	f
0	0	0
0	1	1
1	0	1
1	1	0

w_1	f
0	w_2
1	w_2'

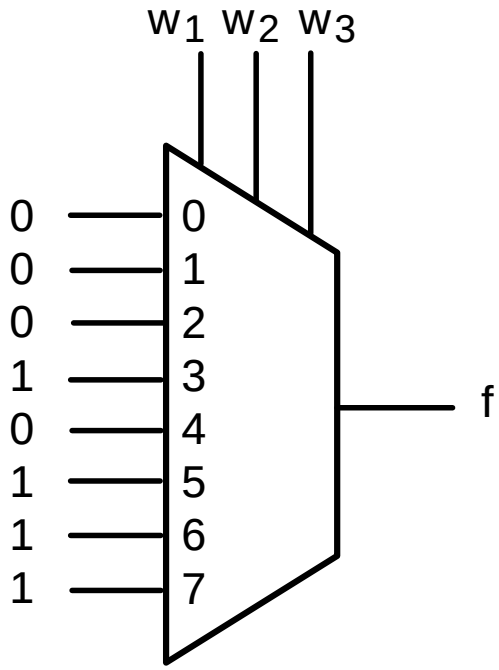
Modified truth table



Circuit

Synthesis of a logic function using multiplexers.

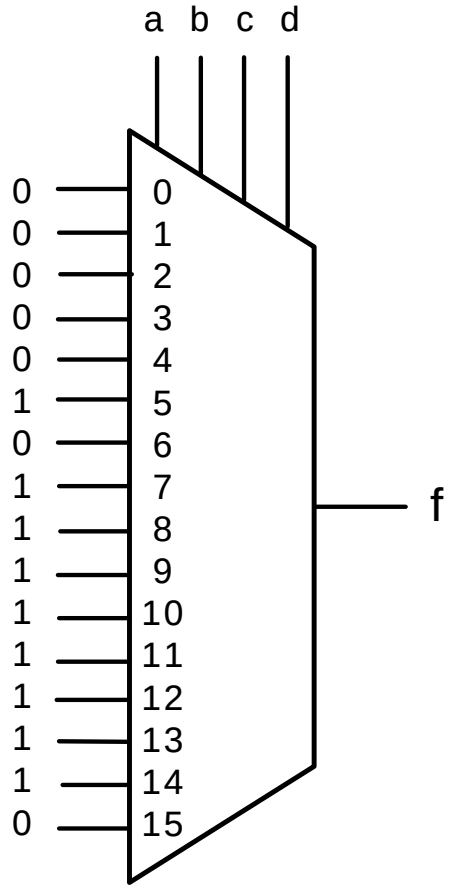
w_1	w_2	w_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

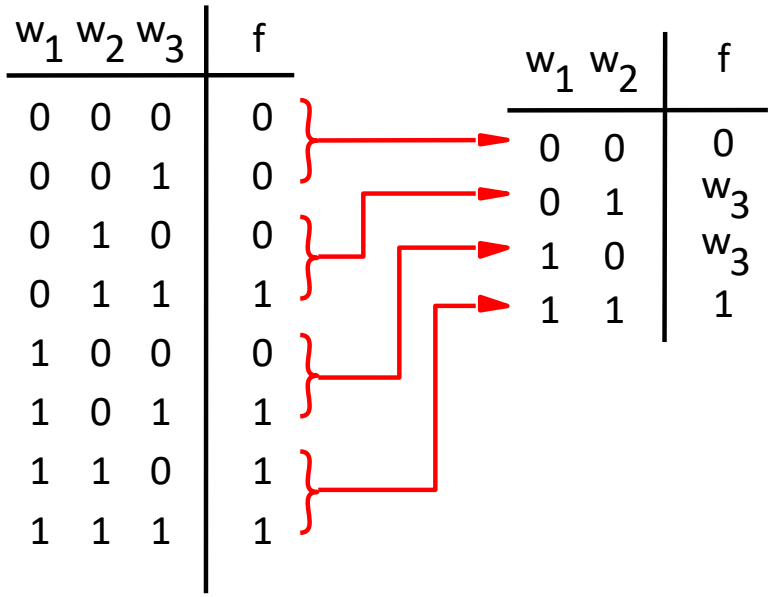


Synthesis of 3-input function using an 8-to-1 multiplexer.

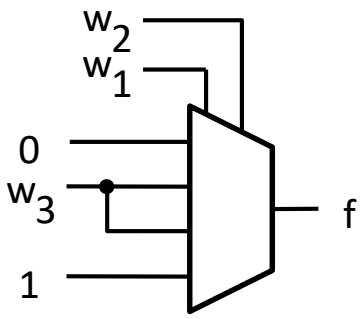
$$f(a, b, c, d) = \Sigma m(5, 7, 8, 9, 10, 11, 12, 13, 14)$$

ab \ cd	00	01	11	10
00				
01		1	1	
11	1	1		1
10	1	1	1	1





Modified truth table

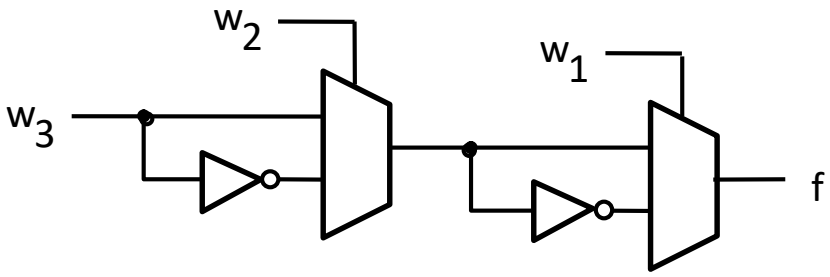


Circuit

A 3-input majority function using a 4-to-1 multiplexer.

w_1	w_2	w_3	f	
0	0	0	0	
0	0	1	1	
0	1	0	1	w_2 ^ w_3
0	1	1	0	
1	0	0	1	
1	0	1	0	
1	1	0	0	(w_2 ^ w_3)'
1	1	1	1	

Truth table

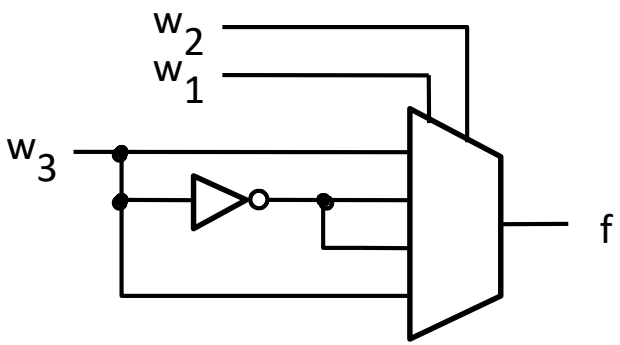


Circuit

A 3-input XOR implemented with 2-to-1 multiplexers.

w_1	w_2	w_3	f	
0	0	0	0	w_3
0	0	1	1	
0	1	0	1	w_3'
0	1	1	0	
1	0	0	1	w_3'
1	0	1	0	
1	1	0	0	w_3
1	1	1	1	

Truth table



Circuit

A 3-input XOR function implemented with a 4-to-1 multiplexer.

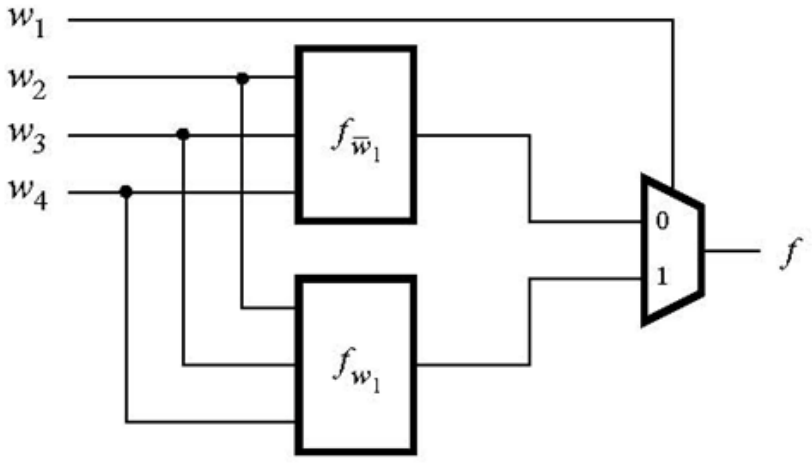
Shannon's expansion theorem

Any Boolean function $f(w_1, w_2, \ldots, w_n)$ can be written in the form:

$$f(w_1, w_2, \ldots, w_n) = w_1' f(0, w_2, \ldots, w_n) + w_1 f(1, w_2, \ldots, w_n)$$

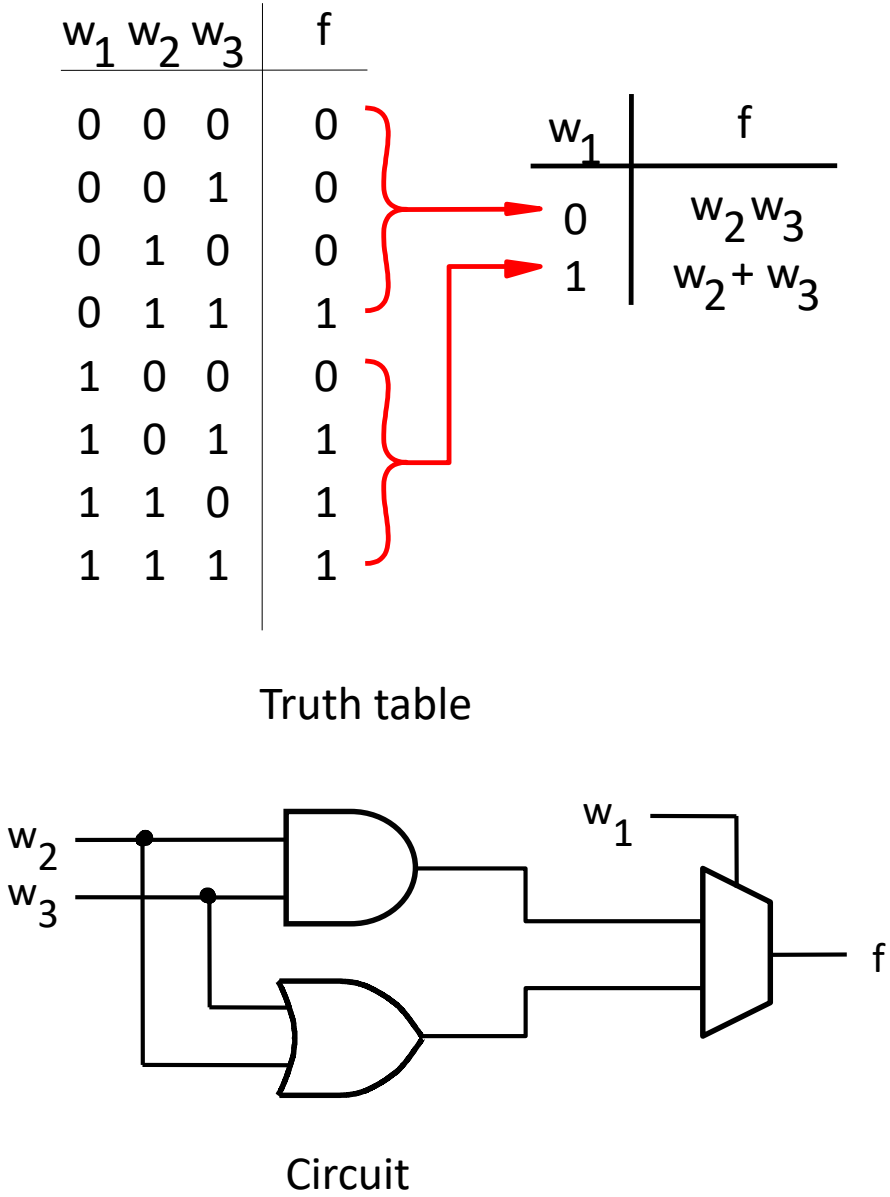
$f(0, w_2, \ldots, w_n)$ is a *cofactor* of f with respect to w_1', written $f_{w1'}$

$f(1, w_2, \ldots, w_n)$ is a *cofactor* of f with respect to w_1, written f_{w1}



(a) Shannon's expansion of the function *f*.

Figure 4.10. The three-input majority function implemented using a 2-to-1 multiplexer.



$f = w1' w3' + w1 w2 + w1 w3$

Shannon's expansion for a 2-in mux:

$f = w1' f_{w1'} + w1 f_{w1}$

$= w1' (w3') + w1 (w2 + w3)$

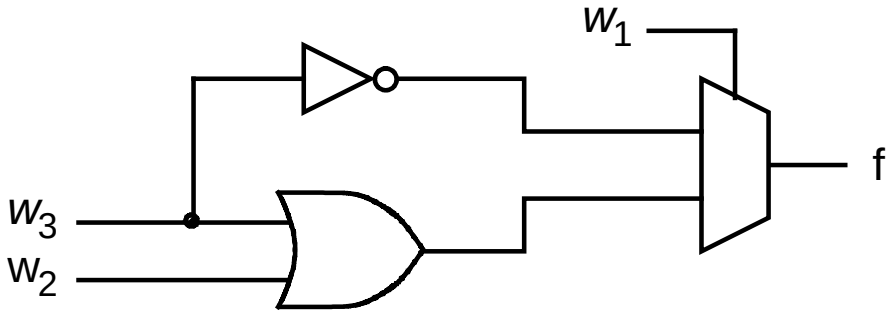
For a 4-in mux, expand again:

$f = w1' w2' f_{w1' w2'} + w1' w2 f_{w1' w2}$

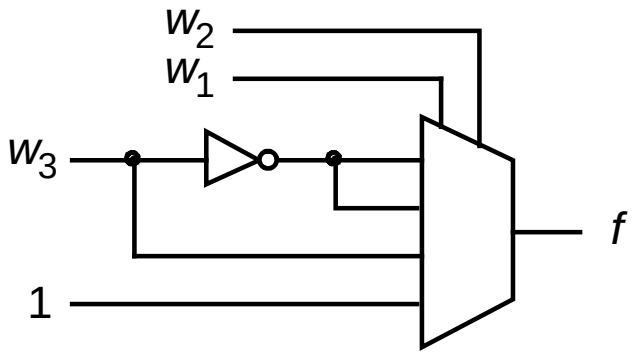
$+ w1 w2' f_{w1 w2'} + w1 w2 f_{w1 w2}$

$= w1' w2' (w3') + w1' w2 (w3')$

$+ w1 w2' (w3) + w1 w2 (1)$



(a) Using a 2-to-1 multiplexer



(b) Using a 4-to-1 multiplexer

Figure 4.11. The circuits synthesized in Example 4.5.

f = w1 w2 + w1 w3 + w2 w3

Shannon's expansion:

f = w1' (w2 w3) + w1 (w2 + w3 + w2 w3)

= w1' (w2 w3) + w1 (w2 + w3)

Let g = w2 w3 and h = w2 + w3.

Expanding both g and h using w2 gives:

g = w2' (0) + w2 (w3)

h = w2' (w3) + w2 (1)

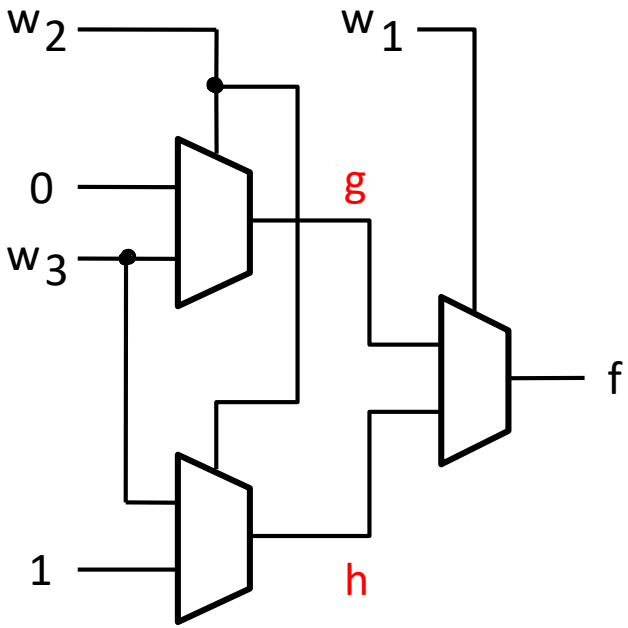


Figure 4.12. A 3-input majority function.

Decoders in Verilog

Select one of many outputs
or transform data.

```
module Decode2to4A( input [ 1:0 ] s, input enable,
      output reg [ 0:3 ] f );

   always @( * )
      if ( enable )
         case ( s )
            0: f = 4'b1000;
            1: f = 4'b0100;
            2: f = 4'b0010;
            3: f = 4'b0001;
         endcase
      else
         f = 4'b0000;

endmodule
```

A 2-to-4 binary decoder.

```
module Decode2to4B( input [ 1:0 ] s, input enable,
      output reg [ 0:3 ] f );

   always @( * )
      case ( { enable, s } )
         3'b100:  f = 4'b1000;
         3'b101:  f = 4'b0100;
         3'b110:  f = 4'b0010;
         3'b111:  f = 4'b0001;
         default: f = 4'b0000;
      endcase

endmodule
```

A 2-to-4 binary decoder.

```
module Decode2to4C( input [ 1:0 ] s, input enable,
      output reg [ 0:3 ] f );

   always @( * )
      casex ( { enable, s } )
         3'b0xx:  f = 4'b0000;
         3'b100:  f = 4'b1000;
         3'b101:  f = 4'b0100;
         3'b110:  f = 4'b0010;
         3'b111:  f = 4'b0001;
      endcase

endmodule
```

A 2-to-4 binary decoder.

```
module Decode4to16A( input [ 3:0 ] s, input enable,
      output reg [ 0:15 ] f );

   wire [ 0:3 ] m;

   Decode2to4 d1( s[ 3:2 ], enable, m[  0:3  ] );
   Decode2to4 d2( s[ 1:0 ], m[ 0 ], f[  0:3  ] );
   Decode2to4 d3( s[ 1:0 ], m[ 1 ], f[  4:7  ] );
   Decode2to4 d4( s[ 1:0 ], m[ 2 ], f[  8:11 ] );
   Decode2to4 d5( s[ 1:0 ], m[ 3 ], f[ 12:15 ] );

endmodule
```

A 4-to-16 binary decoder.

```
module Decode4to16B(input [ 3:0 ] s, input enable,
      output reg [ 0:15 ] f );

   assign f = enable ? 1 << s : 0;

endmodule
```

A 4-to-16 binary decoder.