# BEE 271 Digital circuits and systems

Spring 2017

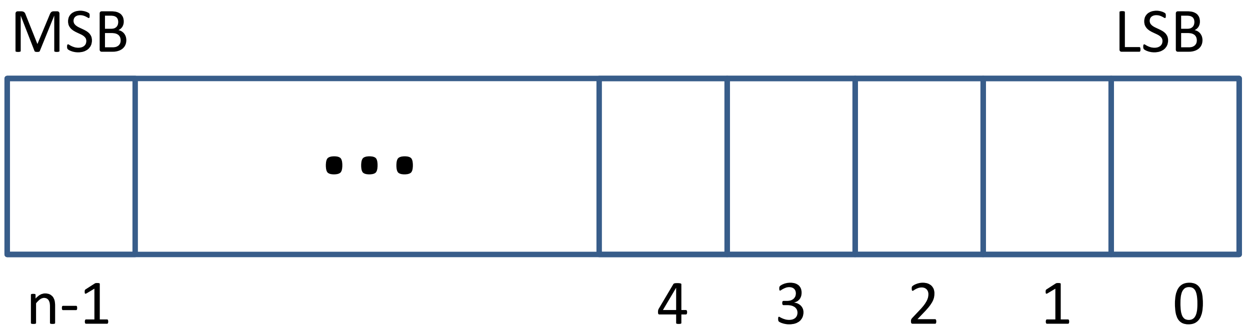
Lecture 7: Multiplexers and Shannon's expansion

Nicole Hamilton

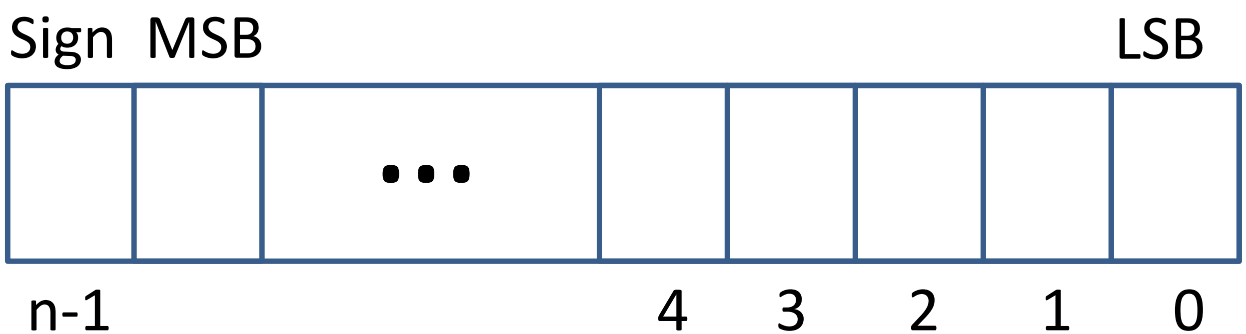
https://faculty.washington.edu/kd1uj

# Topics

1. Signed numbers
2. Adders
3. Multiplexers
4. Shannon’s expansion




Unsigned binary




Signed binary

| $b_3 b_2 b_1 b_0$ | 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 |

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

# 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

| $b_3 b_2 b_1 b_0$ | 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
```

# 2's complement

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

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 $2^n$ minus the value if it's negative.

For example, in 4-bit 2's complement:

+5 = 0101
-5 = 1011

Notice that adding these as unsigned numbers 0101 + 1011 = 10000 = $2^n$, which overflows to 0.

# 2's complement

| $b_3 b_2 b_1 b_0$ | 2's complement |
|---|---|
| 0111 | $+7$ |
| 0110 | $+6$ |
| 0101 | $+5$ |
| 0100 | $+4$ |
| 0011 | $+3$ |
| 0010 | $+2$ |
| 0001 | $+1$ |
| 0000 | $+0$ |
| 1000 | $-8$ |
| 1001 | $-7$ |
| 1010 | $-6$ |
| 1011 | $-5$ |
| 1100 | $-4$ |
| 1101 | $-3$ |
| 1110 | $-2$ |
| 1111 | $-1$ |

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

Then, in 2's complement representation K is obtained by subtracting P from $2^n$ , namely

$$K = 2^n - P$$

# Deriving 2's complement

For a positive n-bit number P, let $K_1$ and $K_2$ denote its 1's and 2's complements, respectively.

$$K_1 = (2^n - 1) - P$$
$$K_2 = 2^n - P$$

Since $K_2 = K_1 + 1$, the 2's complement can computed by inverting all bits of P and then adding 1.

```verilog
module TwosComplementA( input [ 15:0 ] A,
      output [ 15:0 ] minusA );

   assign minusA = ~A + 1;

endmodule
```

Two’s complement in Verilog.

```
module TwosComplementB( input [ 15:0 ] A,
      output [ 15:0 ] minusA );

   assign minusA = -A;

endmodule
```

Two’s complement in Verilog.

```
module TwosComplementC( input signed [ 15:0 ] A,
      output signed [ 15:0 ] minusA );

   assign minusA = -A;

endmodule
```

Two’s complement in Verilog.

# Subtraction in Two's Complement

A - B = A + ( -B ) = A + ~B + 1

1) 0010 - 0110

2) 1011 - 1001

3) 1011 - 0001

# Subtraction in Two's Complement

A - B = A + ( -B ) = A + ~B + 1

1) 0010 - 0110
   +2 +6

0010 (+2)
1001 (~6)
<u>0001 (+1)</u>
1100 (-4)

2) 1011 - 1001

3) 1011 - 0001

# Subtraction in Two's Complement

A - B = A + ( -B ) = A + ~B + 1

1) 0010 - 0110

2) 1011 - 1001
   -5 -7

1011 (-5)
0110 (~-7)
0001 (+1)
0010 (+2)

3) 1011 - 0001

# Subtraction in Two's Complement

A - B = A + ( -B ) = A + ~B + 1

1) 0010 - 0110

2) 1011 - 1001

3) 1011 - 0001

-5 +1

| | |
|---|---|
| 1011 | (-5) |
| 1110 | (~1) |
| 0001 | (+1) |
| 1010 | (-6) |

# Sign Extension

To convert from N-bit to M-bit 2's Complement ($N<M$), simply duplicate sign bit:

1. Convert $(0010)_2$ to 8-bit 2's Complement

2. Convert $(1011)_2$ to 8-bit 2's Complement

# Sign Extension

To convert from N-bit to M-bit 2’s Complement (N<M), simply duplicate sign bit:

1. Convert $(0010)_2$ to 8-bit 2’s Complement

   0000 0010

2. Convert $(1011)_2$ to 8-bit 2’s Complement

   1111 1011

```
module SignExtendA( input [ 7:0 ] a,
      output [ 15:0 ] b );

   // Sign-extend both a, replicating A[ 7 ]
   // through eight positions.

   assign b = { { 8 { a[ 7 ] } }, a };

endmodule
```

Sign extend in Verilog using replication and concatenation.

```verilog
module SignExtendB( input signed [ 7:0 ] a,
      output signed [ 15:0 ] b );

   // Let the compiler sign-extend a using
   // the signed keyword.

   assign b = a;

endmodule
```

Sign extend in Verilog using the signed keyword.

```
module AddUnsigned( input [ 3:0 ] A, input [ 7:0 ] B,
      output [ 9:0 ] sum );

   // Verilog pads high-order bits with zeros.

   assign sum = A + B;

endmodule
```

Adding unsigned numbers of different sizes in Verilog.

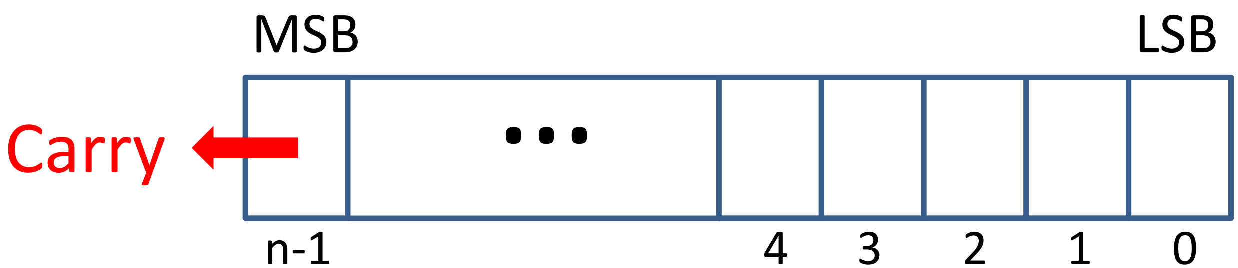
```
module AddSignedA( input [ 3:0 ] A, input [ 7:0 ] B,
      output [ 9:0 ] sum );

   // Must sign-extend both A and B, replicating
   // A[ 3 ] through six positions and B[ 7 ] through
   // two positions.

   assign sum = { { 6{ A[ 3 ] } }, A } +
                { 2{ B[ 7 ] } }, B };

endmodule
```

Adding signed numbers of different sizes in Verilog.

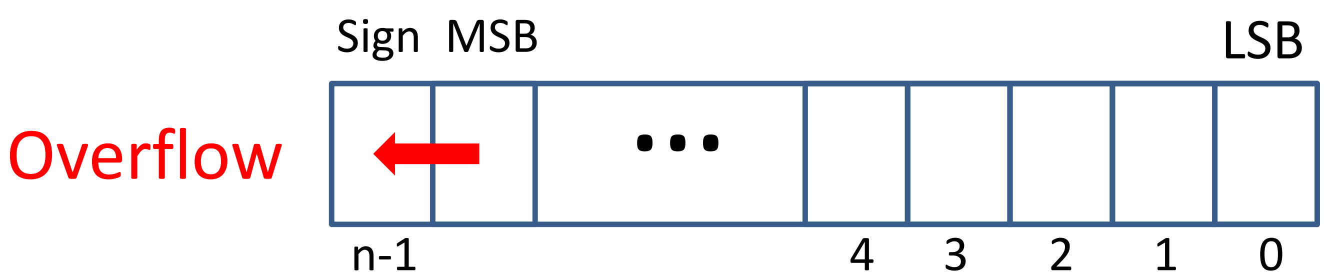
```verilog
module AddSignedB( input signed [ 3:0 ] A,
      input signed [ 7:0 ] B,
      output signed [ 9:0 ] sum );

   // Let the compiler sign-extend A and B using
   // the signed keyword.

   assign sum = A + B;

endmodule
```

Adding signed numbers of different sizes in Verilog.

# Unsigned addition




# Signed addition




Processor instruction sets have both carry and overflow status bits so only one add instruction is needed for either signed or unsigned addition.

```
module AdderWithOverflowA( input [ 15:0 ] a, b,
      output [ 15:0 ] s, carryOut, overflow );

   // Carryout is a carry from the MSB in
   // unsigned arithmetic.

   // Overflow is a carry from the MSB in
   // signed arithmetic into the sign bit.

   assign { carryOut, s } = a + b,
          overflow = a[ 15 ] == b[ 15 ] &&
                     a[ 15 ] != s[ 15 ];

endmodule
```

Overflow and carry detection in Verilog.

```
module AdderWithOverflowB( input [ 15:0 ] a, b,
      output [ 15:0 ] s, carryOut, overflow );

   // Carryout is a carry from the MSB in
   // unsigned arithmetic.

   // Overflow is a carry from the MSB in
   // signed arithmetic into the sign bit.

   assign { carryOut, s } = a + b,
          overflow = ~a[ 15 ] & ~b[ 15 ] &  s[ 15 ] |
                      a[ 15 ] &  b[ 15 ] & ~s[ 15 ];

endmodule
```

Overflow and carry detection in Verilog.

```verilog
module AdderWithOverflowC( input [ 15:0 ] a, b,
      output [ 15:0 ] s, carryOut, output reg overflow );

   // Carryout is a carry from the MSB in
   // unsigned arithmetic.

   // Overflow is a carry from the MSB in
   // signed arithmetic into the sign bit.

   wire [ 0:2 ] signbits= { a[ 15 ], b[ 15 ], s[ 15 ] };
   assign { carryOut, s } = a + b;

   always @( * )
      case ( signbits )
         'b001, 'b110:  overflow = 1;
         default:  overflow = 0;
      endcase

endmodule
```

Overflow and carry detection in Verilog.

```verilog
module AdderWithOverflowD( input [ 15:0 ] a, b,
      output [ 15:0 ] s, carryOut, overflow );

   // Carryout is a carry from the MSB in
   // unsigned arithmetic.

   // Overflow is a carry from the MSB in
   // signed arithmetic into the sign bit.

   wire [ 0:2 ] signbits = { a[ 15 ], b[ 15 ], s[ 15 ] };
   assign { carryOut, s } = a + b,
          overflow = signbits == 'b001 || signbits == 'b110;

endmodule
```

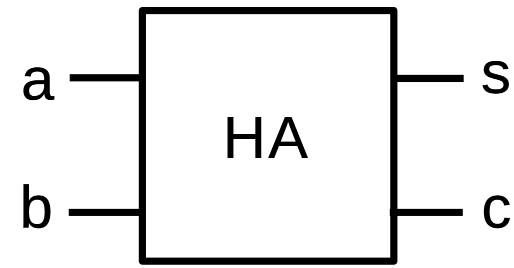
Overflow and carry detection in Verilog.

# Adders

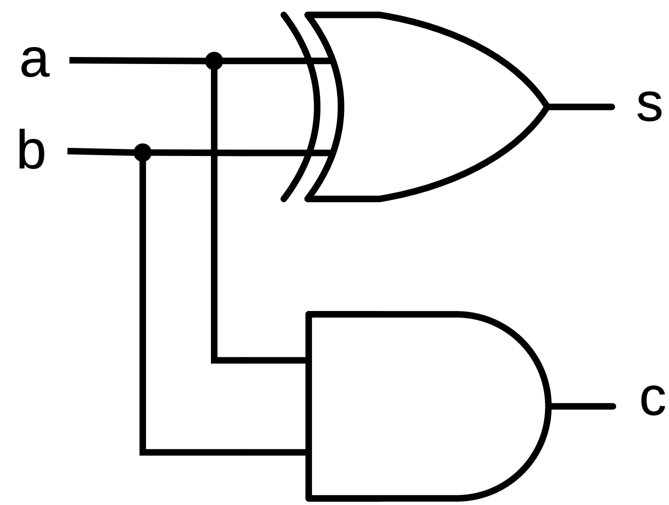
# When we add numbers we get carries.

| In decimal | In binary |
|---|---|
| *110* | *011* |
| 1492 | 1011 |
| + 525 | + 011 |
| 2017 | 1110 |

| | | | | |
|---|---|---|---|---|
| a | 0 | 0 | 1 | 1 |
| +b | +0 | +1 | +0 | +1 |
| c s | 0 0 | 0 1 | 0 1 | 1 0 |




| a | b | c | s |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |




Addition of one-bit binary numbers.

| $c_i$ | $x_i$ | $y_i$ | $c_{i+1}$ | $s_i$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

(a) Truth table

| $c_i$ \ $x_i y_i$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | | 1 | | 1 |
| 1 | 1 | | 1 | |

$$s_i = x_i \oplus y_i \oplus c_i$$

| $c_i$ \ $x_i y_i$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | | | 1 | |
| 1 | | 1 | 1 | 1 |

$$c_{i+1} = x_i y_i + x_i c_i + y_i c_i$$

(b) Karnaugh maps

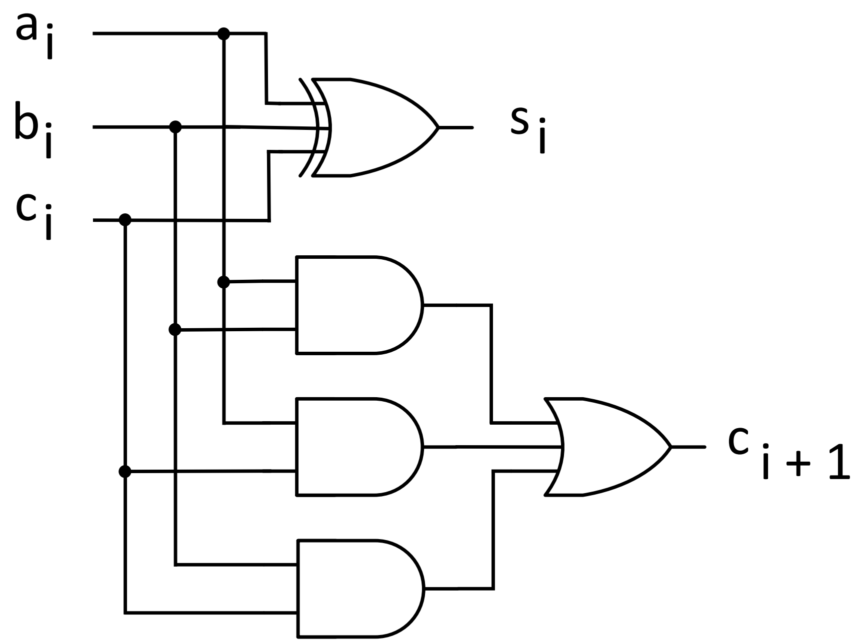
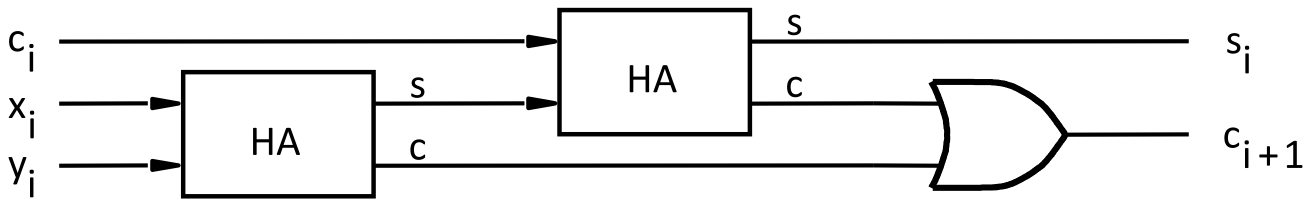
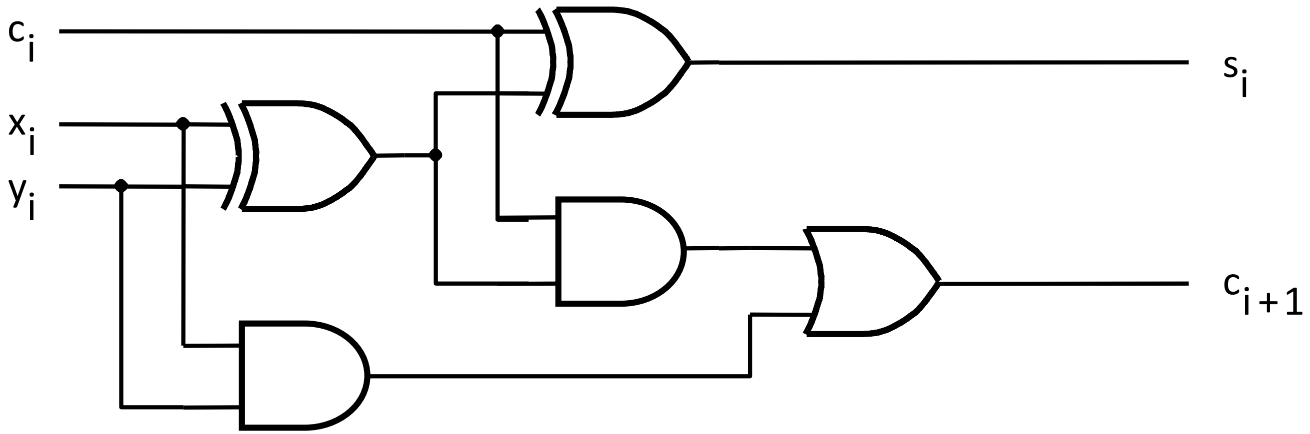



Figure 3.3. Full-adder.

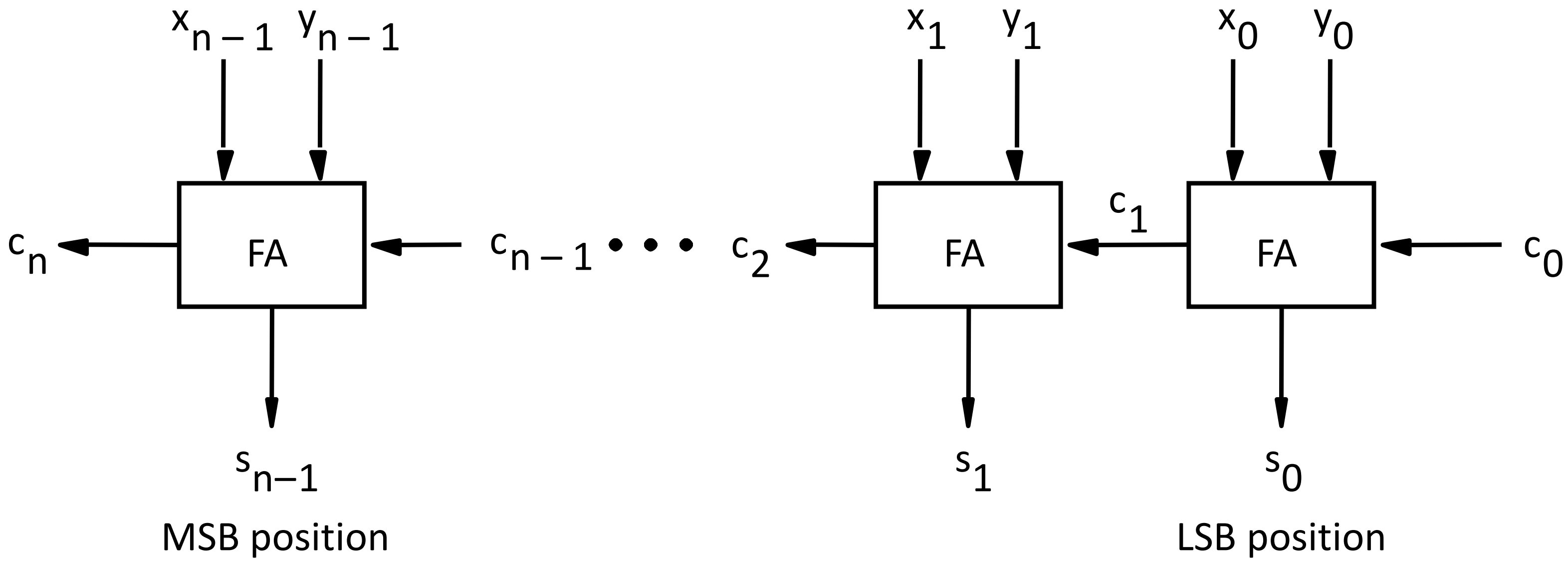



Block diagram



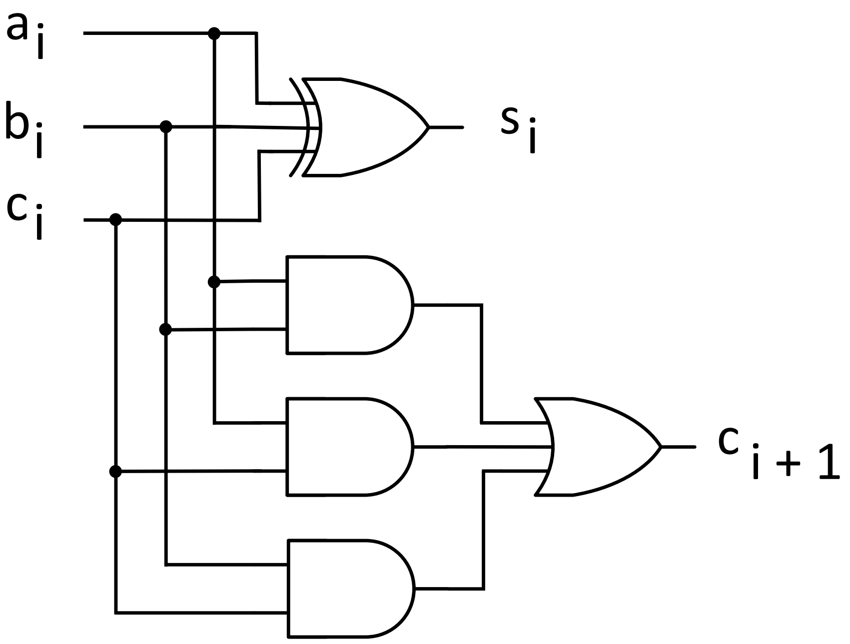

Detailed diagram

A full adder built using half adders.



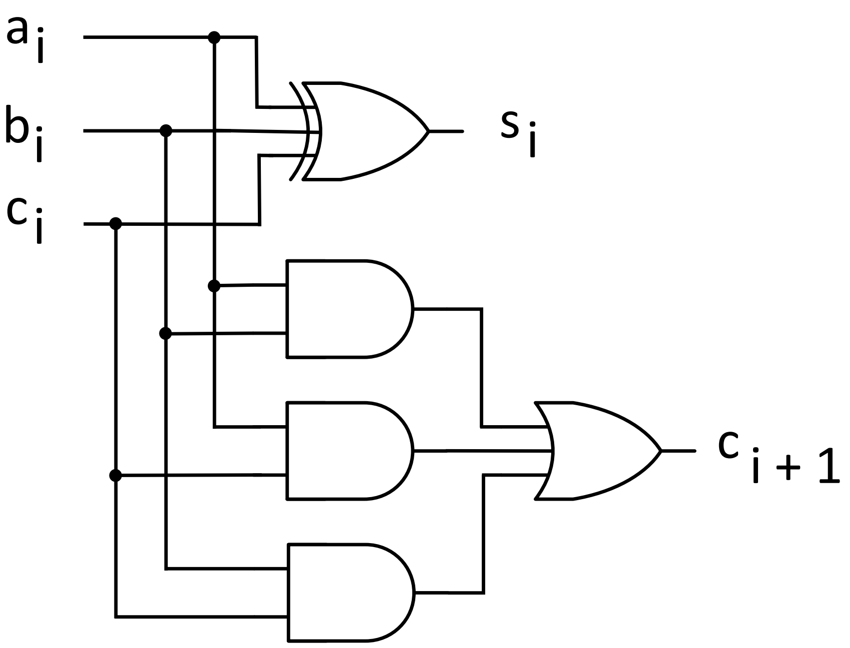

An n-bit ripple-carry adder.

```
module FullAdderA( input cin, a, b,
       output s, cout );

   wire x, y, z;
   xor ( s, a, b, cin );
   and ( x, a, b );
   and ( y, a, cin );
   and ( z, b, cin );
   or ( cout, x, y, z );

endmodule
```



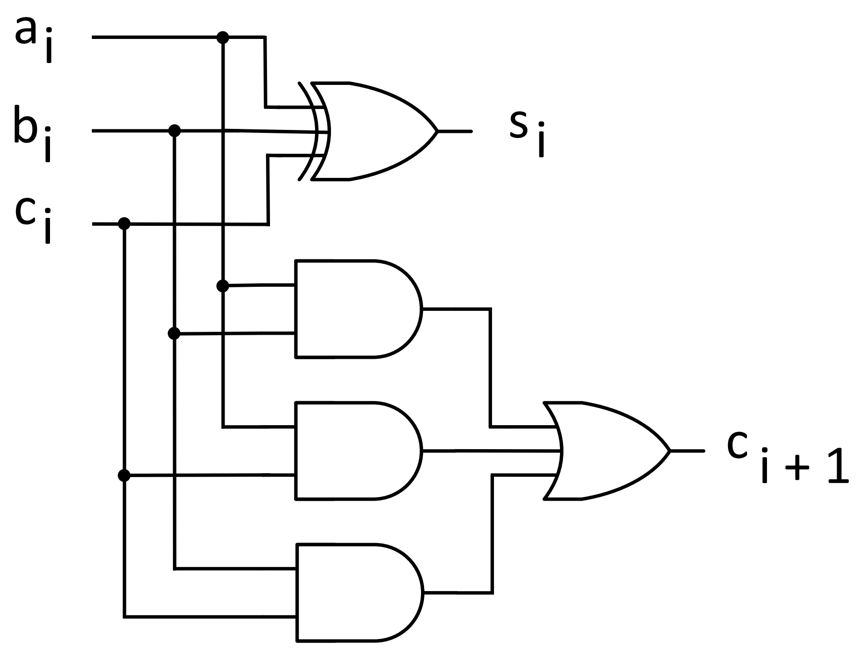

A full adder in Verilog.

```
module FullAdderB( input cin, a, b,
       output s, cout );

   wire x, y, z;
   xor ( s, a, b, cin );
   and ( x, a, b ),
       ( y, a, cin ),
       ( z, b, cin );
   or ( cout, x, y, z );

endmodule
```



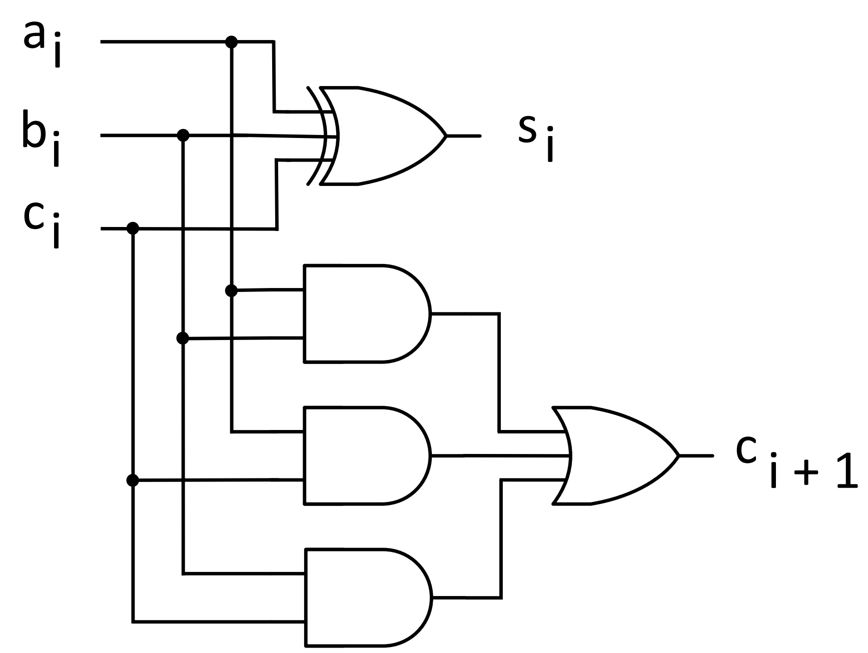

A full adder in Verilog.

```
module FullAdderC( input cin, a, b,
       output s, cout );

   assign s = a ^ b ^ cin;
   assign cout = a & b | a & cin | b & cin;

endmodule
```



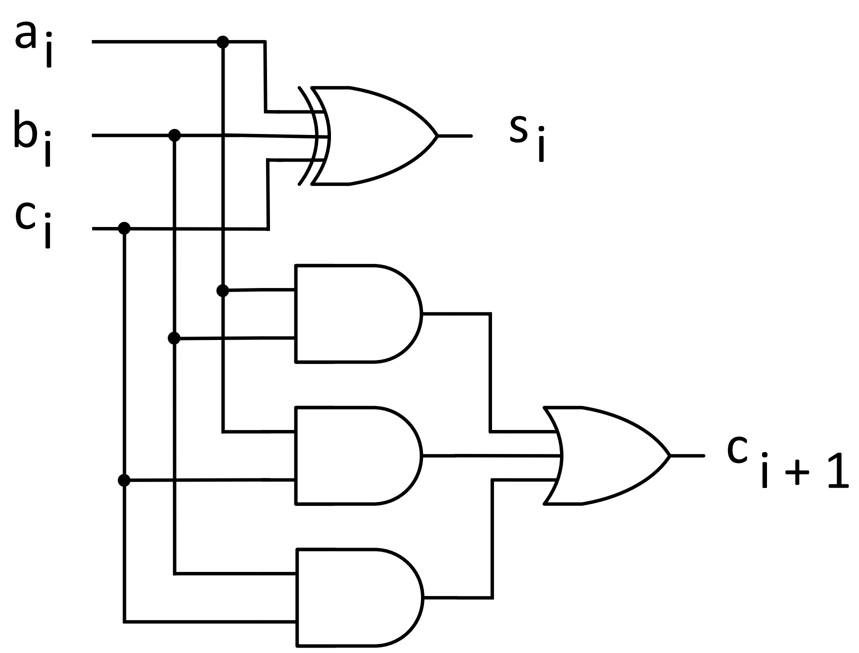

A full adder in Verilog.

```verilog
module FullAdderD( input cin, a, b,
      output s, cout );

   assign s = a ^ b ^ cin,
          cout = a & b | a & cin | b & cin;

endmodule
```



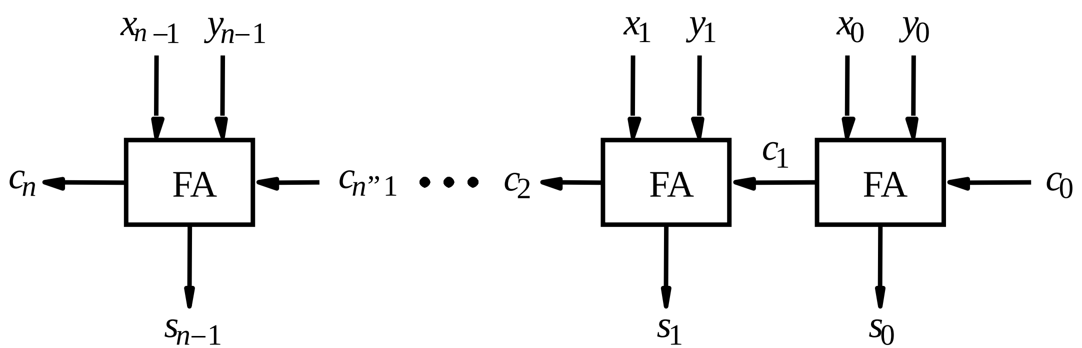

A full adder in Verilog.

```
module FullAdderE( input cin, a, b,
       output s, cout );

   assign { cout, s } = a + b + cin;

endmodule
```




A full adder in Verilog.

```
module FourBitAdderA( input cin, input [ 3:0 ] a, b,
      output cout, output [ 3:0 ] s );

   wire [ 3:1 ] c;
   FullAdderE( cin,    a[ 0 ], b[ 0 ], s[ 0 ], c[ 1 ] );
   FullAdderE( c[ 1 ], a[ 1 ], b[ 1 ], s[ 1 ], c[ 2 ] );
   FullAdderE( c[ 2 ], a[ 2 ], b[ 2 ], s[ 2 ], c[ 3 ] );
   FullAdderE( c[ 3 ], a[ 3 ], b[ 3 ], s[ 3 ], cout );

endmodule
```




A 4-bit adder in Verilog.

```
module NBitAdderA( input cin, input [ n - 1:0 ] a, b,
      output cout, output [ n - 1:0 ] s );

   parameter n = 16;
   wire [ n:0 ] c;
   assign c[ 0 ] = cin,
          cout   = c[ n ];

   generate
      genvar i;
      for ( i = 0; i <= n; i = i + 1 )
         begin : fa
         FullAdderE stage( c[ i ], a[ i ], b[ i ], s[ i ],
            c[ i + 1 ] );
         end
   endgenerate

endmodule
```

An n-bit adder in Verilog.

```
module NBitAdderB( input cin, input [ n - 1:0 ] a, b,
      output reg cout, output reg [ n - 1:0 ] s );

   parameter n = 16;
   reg [ n:0 ] c;
   integer i;

   always @( * )
      begin
      c[ 0 ] = cin;
      for ( i = 0; i < n; i = i + 1 )
         begin
         s[ i ] = a[ i ] ^ b[ i ] ^ c[ i ];
         c[ i + 1 ] = a[ i ] & b[ i ] |
                      a[ i ] & c[ i ] |
                      b[ i ] & c[ i ];
         end
      cout = c[ n ];
      end

endmodule
```

An n-bit adder in Verilog.

```
module NBitAdderC( input cin, input [ n - 1:0 ] a, b,
      output reg cout, output reg [ n - 1:0 ] s );

   parameter n = 16;

   always @( * )
      begin
      reg [ n:0 ] c;
      integer i;
      c[ 0 ] = cin;
      for ( i = 0; i < n; i = i + 1 )
         begin
         s[ i ] = a[ i ] ^ b[ i ] ^ c[ i ];
         c[ i + 1 ] = a[ i ] & b[ i ] |
                      a[ i ] & c[ i ] |
                      b[ i ] & c[ i ];
         end
      cout = c[ n ];
      end

endmodule
```

An n-bit adder in Verilog.

```verilog
module NBitAdderD( input cin, input [ n - 1:0 ] a, b,
      output cout, output [ n - 1:0 ] s );

   parameter n = 16;
   assign { cout, s } = a + b + cin;

endmodule
```

An n-bit adder in Verilog.

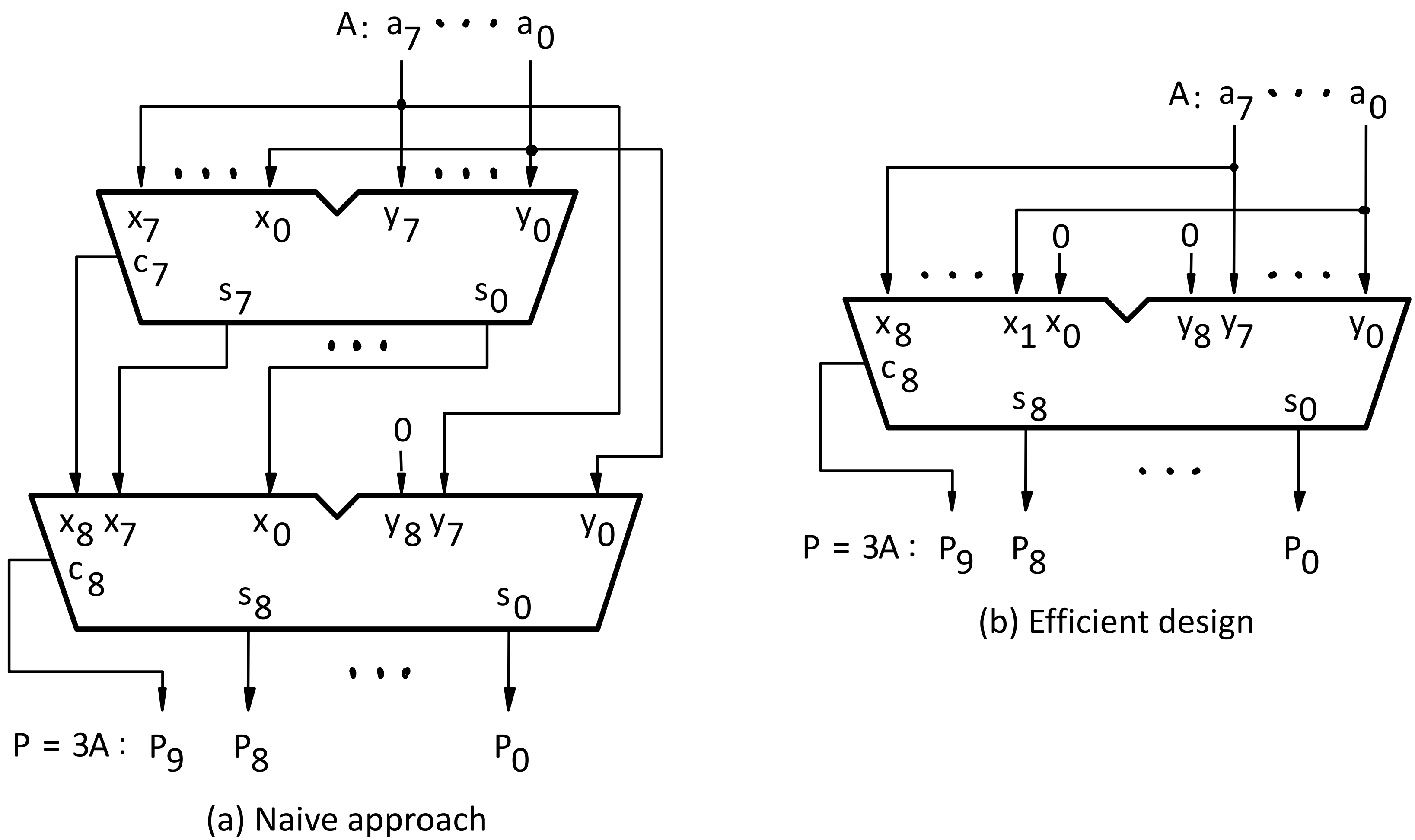


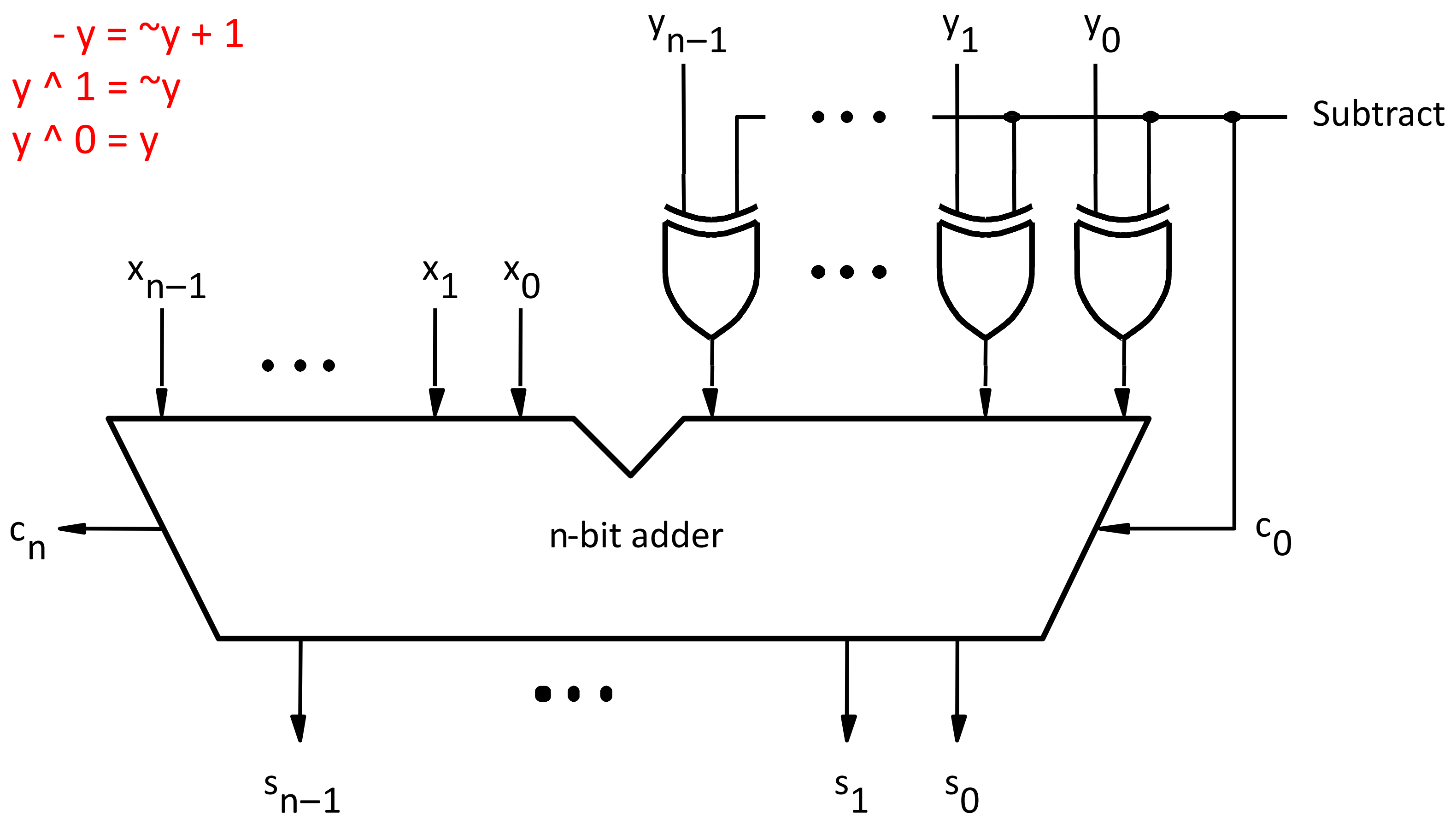

Figure 3.6. Circuit that multiplies an eight-bit unsigned number by 3.

$\{ cn, s \} = Subtract\ ?\ x - y : x + y$

$-y = \sim y + 1$

$y \wedge 1 = \sim y$

$y \wedge 0 = y$




An adder/subtractor unit

# Critical path

# Performance

Measure the largest delay from operands being presented as inputs until all output bits are valid.

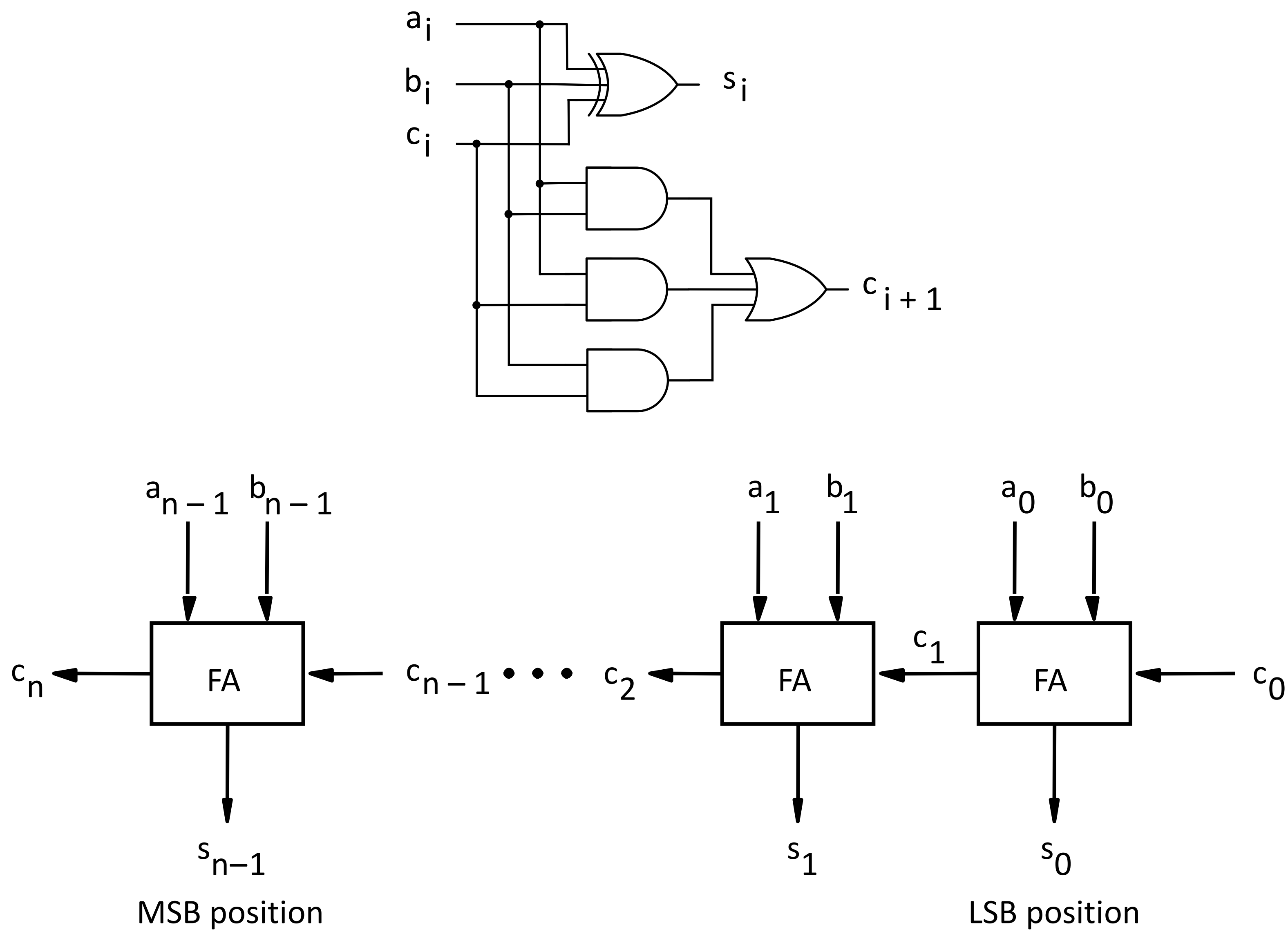
Often referred to as the *critical path delay*.

# Performance

Performance gains often come from insights into how to structure a problem, e.g., into *layers, modules* or *hierachies*, thus allowing an efficient solution.

These are often called *architectural* decisions.

Addition offers a classic example where an insight about when carries are generated or propagated allows a dramatic shortening of the critical path.




An n-bit ripple-carry adder has 2 gate delays per bit.

# Generate & propagate

$$c_{i+1} = a_i\, b_i + a_i\, c_i + b_i\, c_i$$

$$c_{i+1} = a_i\, b_i + ( a_i + b_i )\, c_i$$

$$g_i = a_i\, b_i = \textit{Generate carry}$$

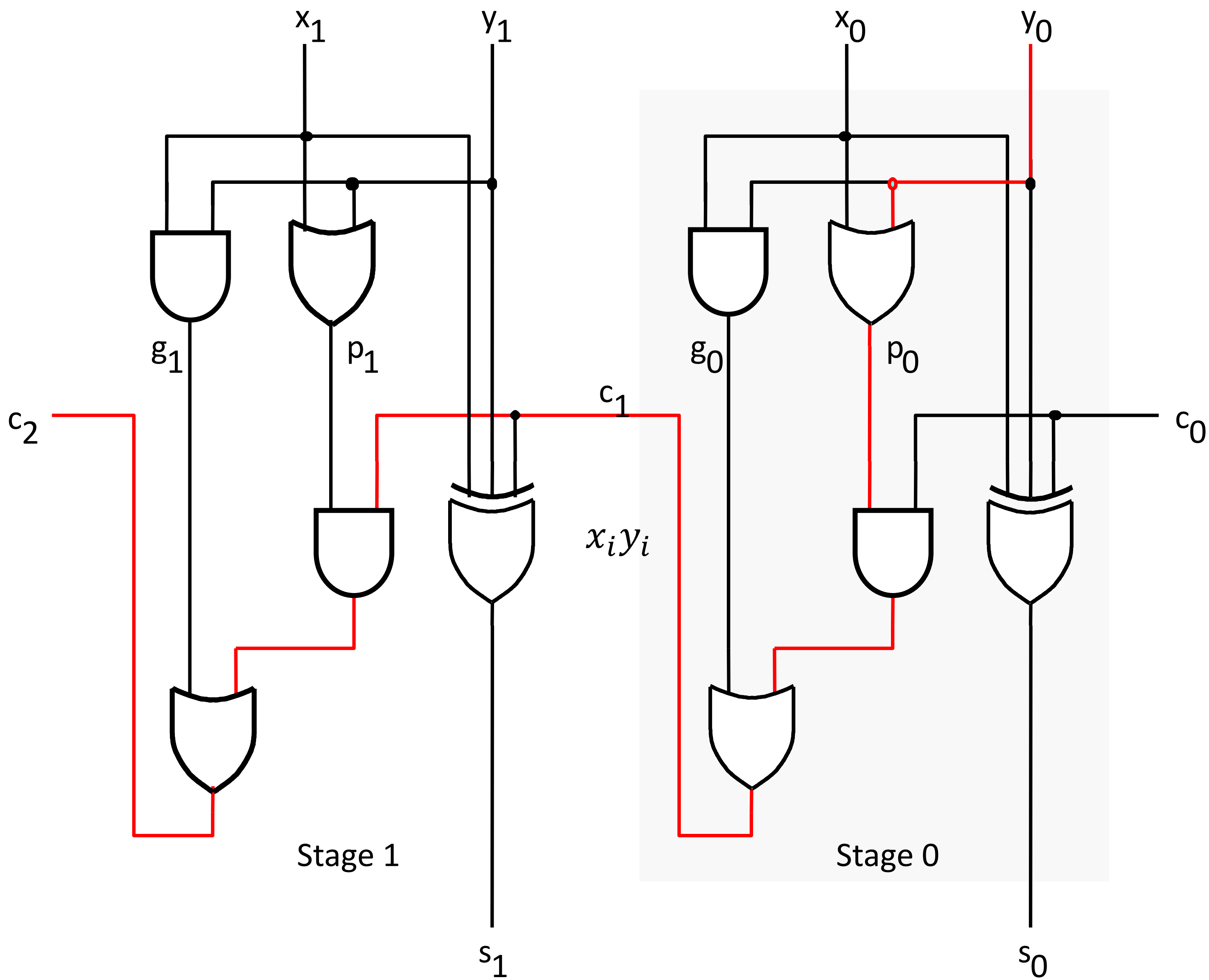
$$p_i = a_i + b_i = \textit{Propagate carry}$$

$$c_{i+1} = g_i + p_i\, c_i$$

# Generate & propagate

By themselves, generate and propagate offer no advantage if all you do is build a ripple adder.

It just adds one more gate delay.




A ripple adder using carry propagate and carry generate.
Delay = 2n + 1 gate delays, where n = number of stages (bits)

# Carry-lookahead

The key insight that all of the gi and pi terms have only one gate delay.

So it's possible to simply expand terms to figure out whether a carry into an *entire block* of, say 4 or 8 or 16 bits, will generate a carry out

You can do that with only two more gate delays if you have gates with enough inputs.

This is called *carry-lookahead*.

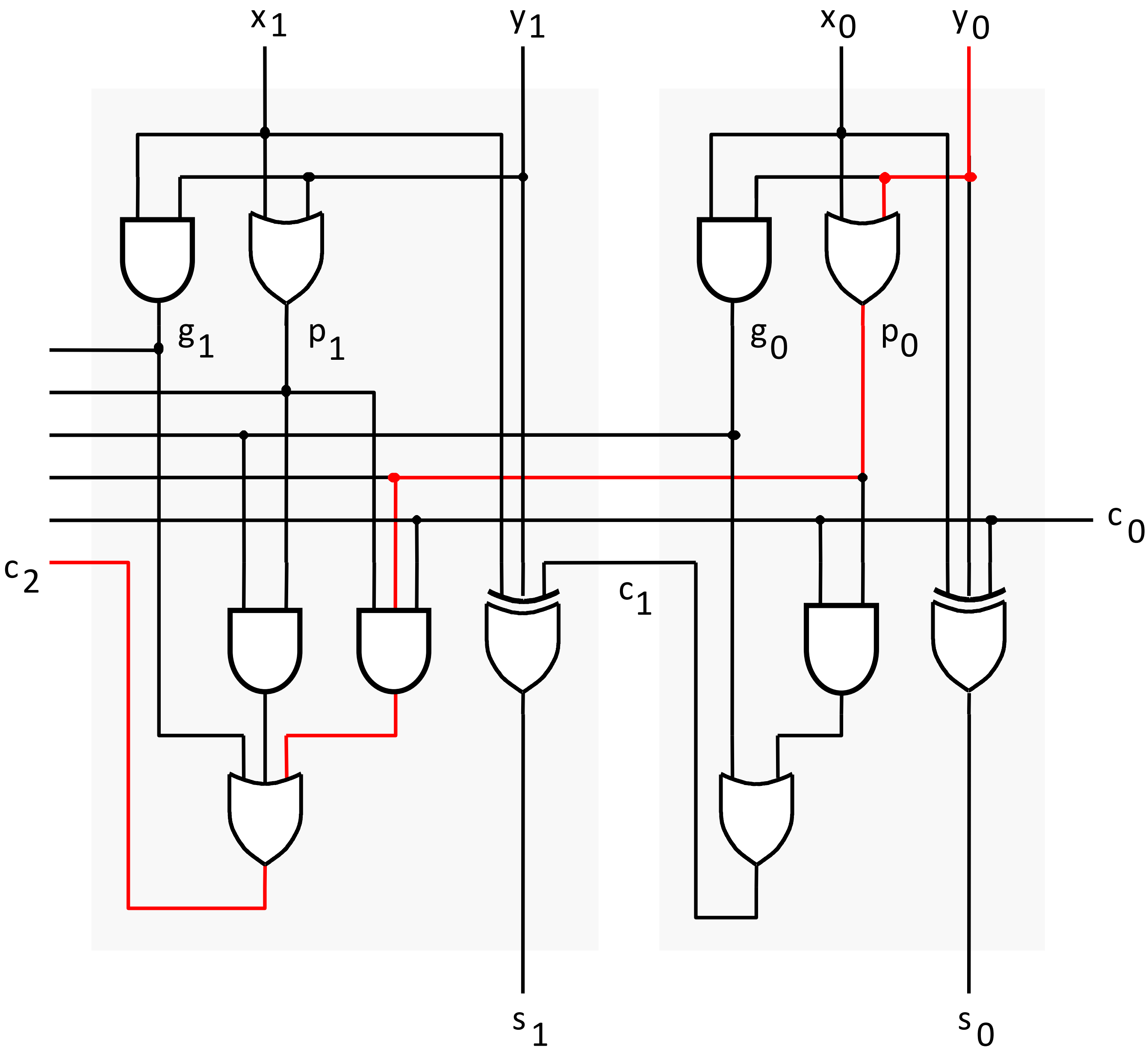



Figure 3.15. The first two stages of a carry-lookahead adder.

# Carry lookahead

$$c_{i+1} = g_i + p_i\, c_i$$

$$c_1 = g_0 + p_0 c_0$$

$$c_2 = g_1 + p_1\left(g_0 + p_0 c_0\right) = g_1 + p_1 g_0 + p_1 p_0 c_0$$

$$c_3 = g_2 + p_2\left(g_1 + p_1 g_0 + p_1 p_0 c_0\right)$$

$$= g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0$$

$$\vdots$$

$$c_{i+1} = g_i + \sum_{j=0}^{i-1}\left(g_j \prod_{k=j+1}^{i} p_k\right) + c_0 \prod_{k=0}^{i} p_k$$

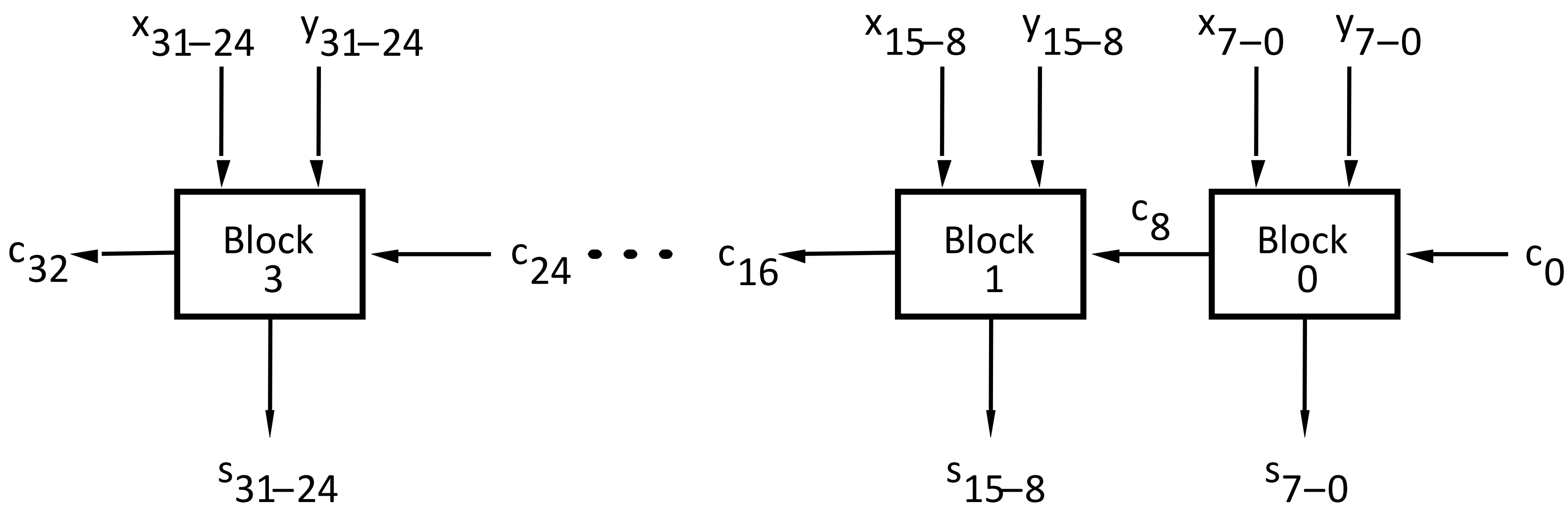



Figure 3.16. A hierarchical carry-lookahead adder with ripple-carry between blocks.

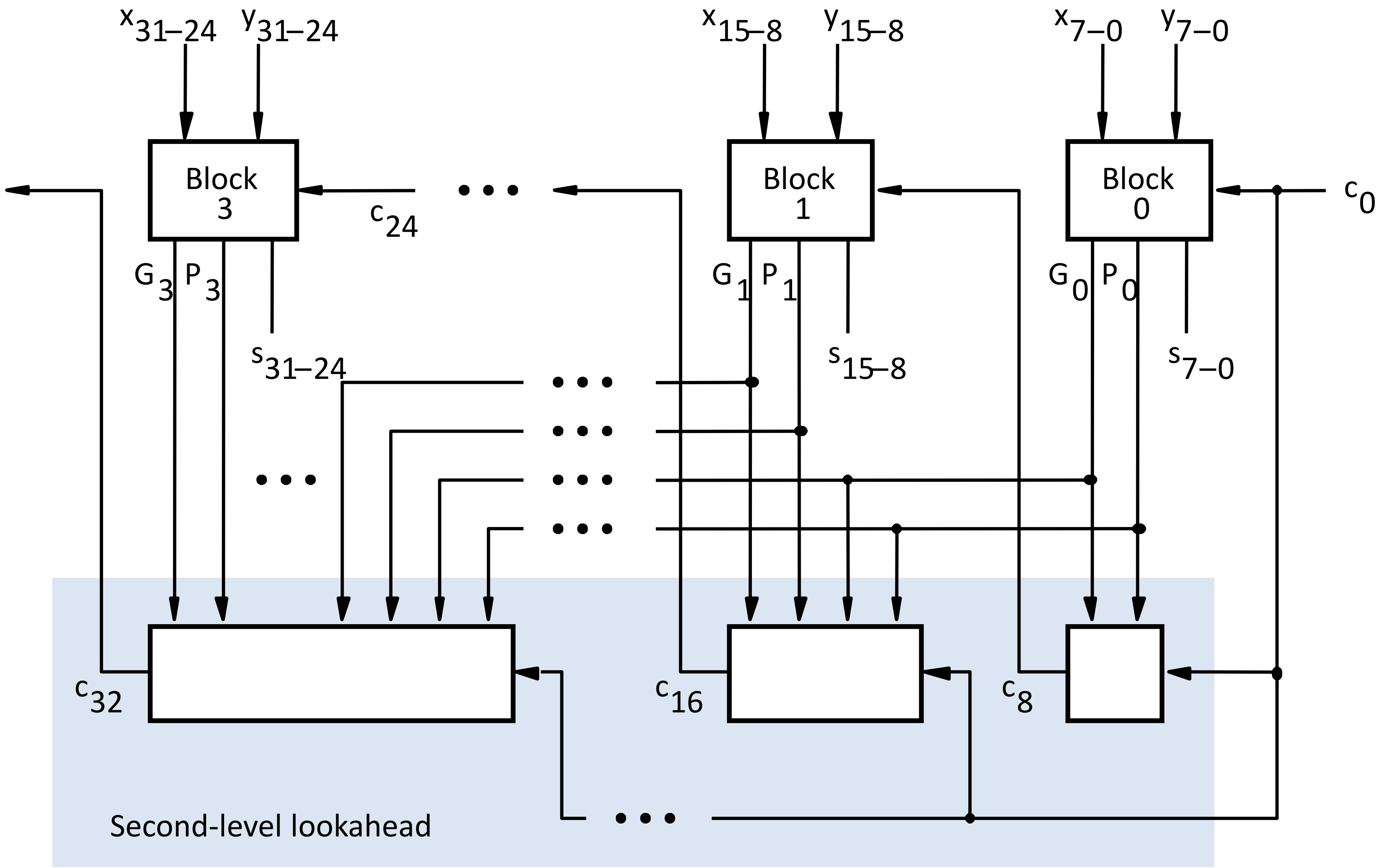
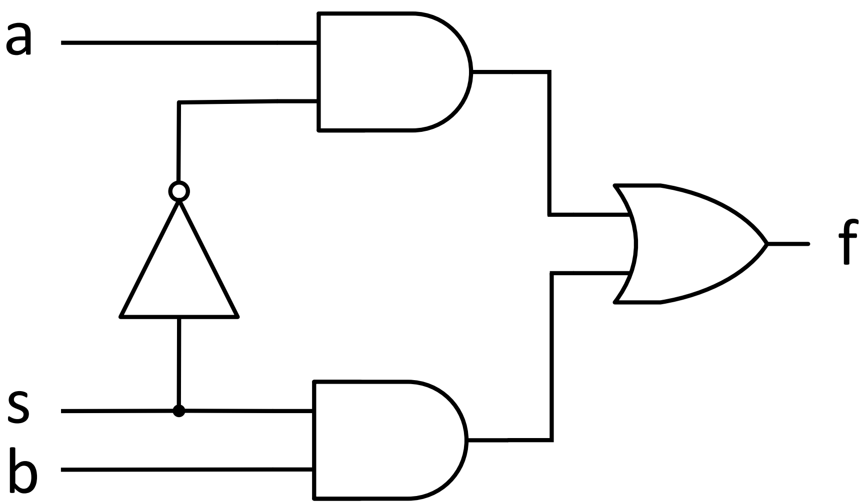


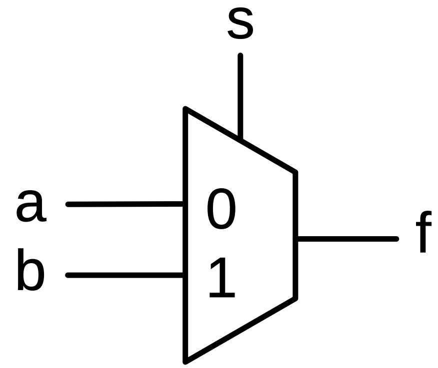

Figure 3.17. A hierarchical carry-lookahead adder.

# 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



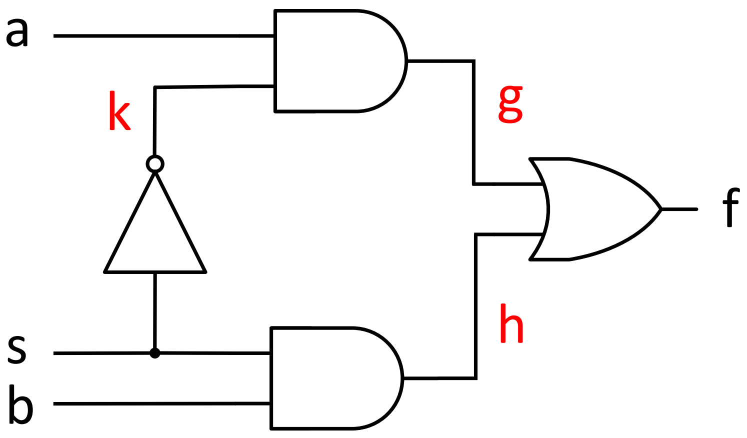




| s | f |
|---|---|
| 0 | a |
| 1 | b |

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

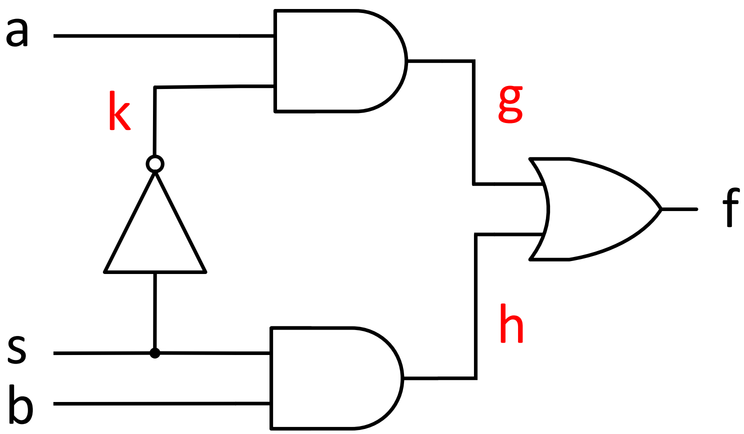
# Multiplexer

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



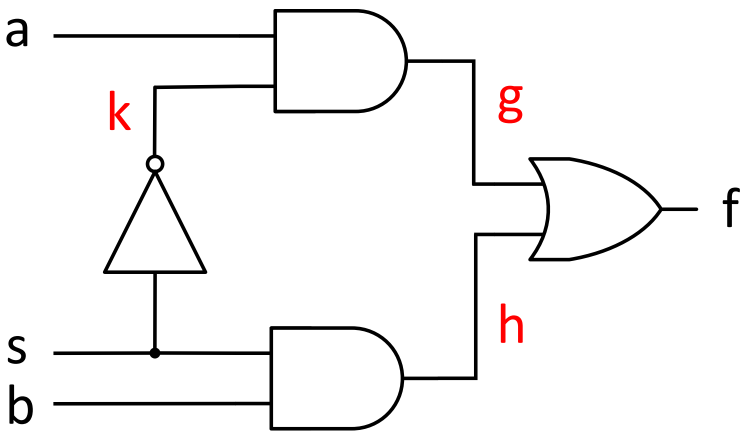

```
module Mux2To1A(
        input  s, a, b,
        output f );

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

endmodule
```

Structural code for a multiplexer.



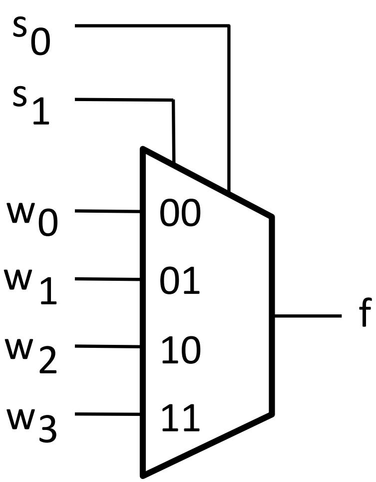

```
module Mux2To1D(
        input  s, a, b,
        output f );

    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
```

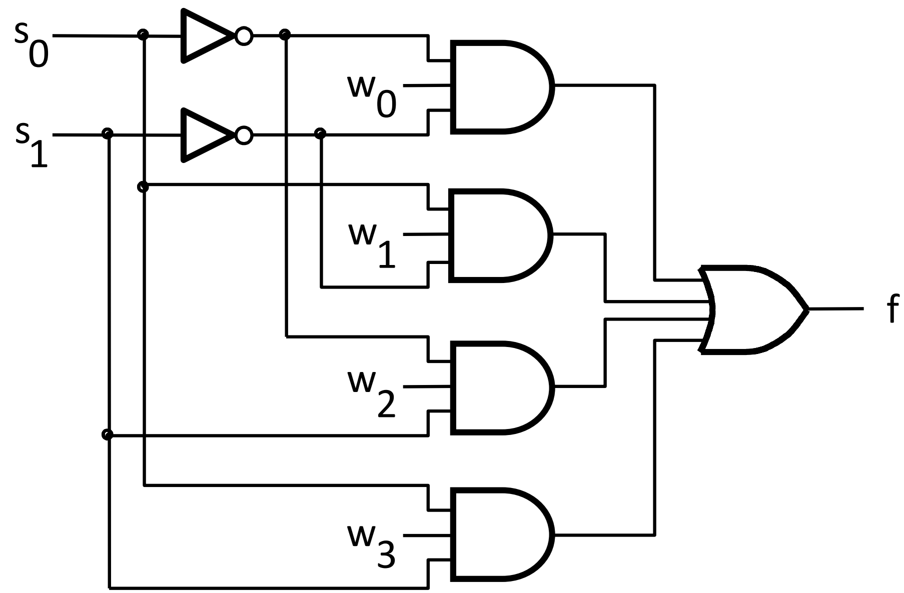
Behavioral description of a multiplexer.




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.

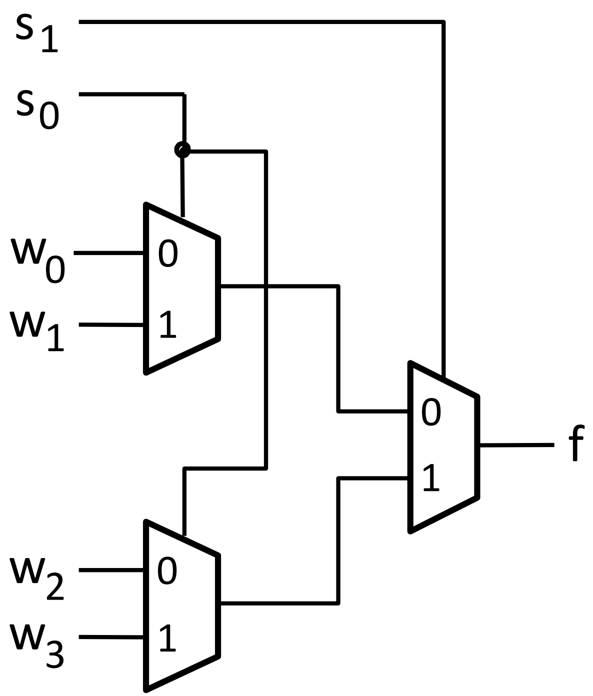
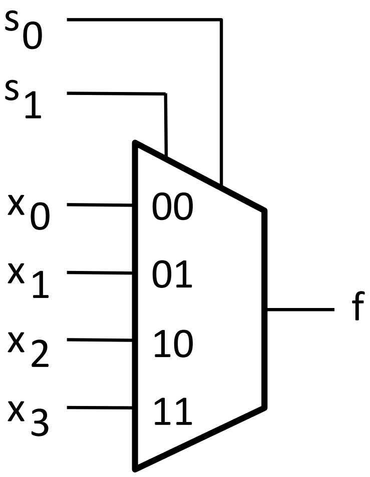


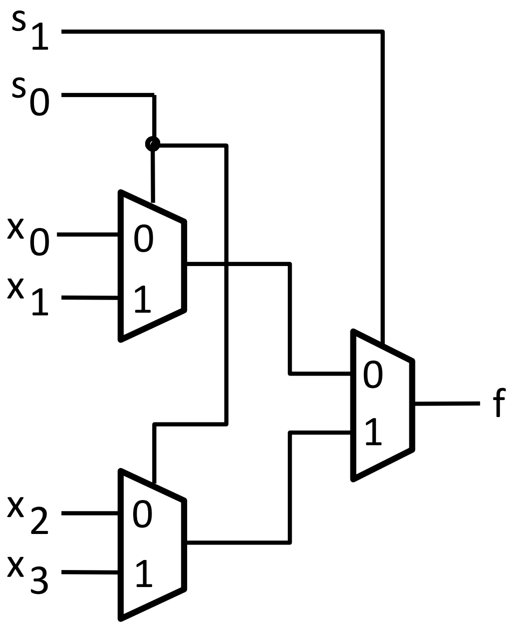

Figure 4.3. Using 2-to-1 multiplexers to build a 4-to-1 multiplexer.

```verilog
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
```



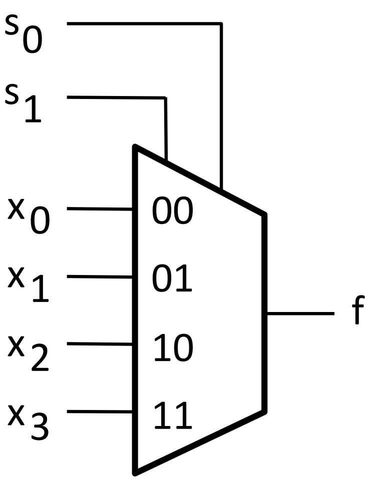

A 4-to-1 multiplexer.

```
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
```



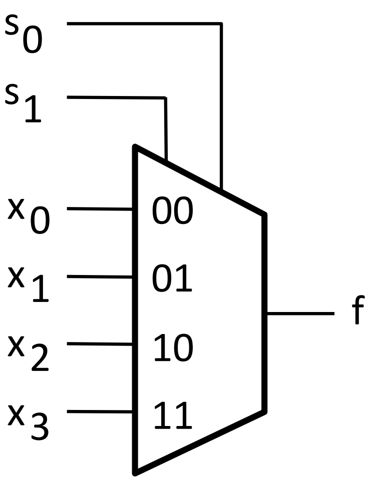

A 4-to-1 multiplexer.

```
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
```



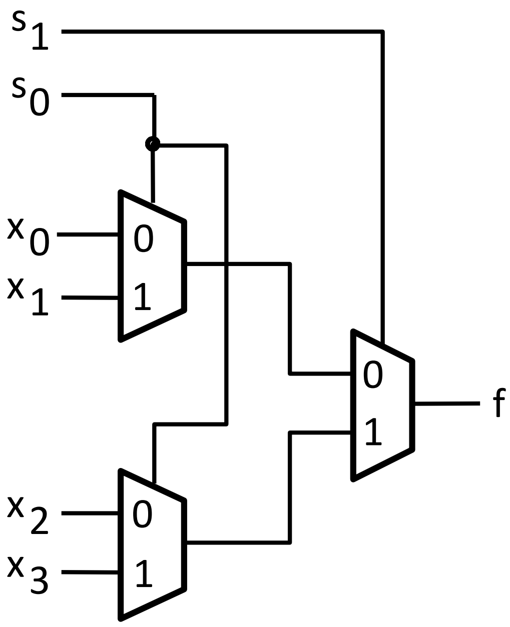

A 4-to-1 multiplexer.

```
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
```



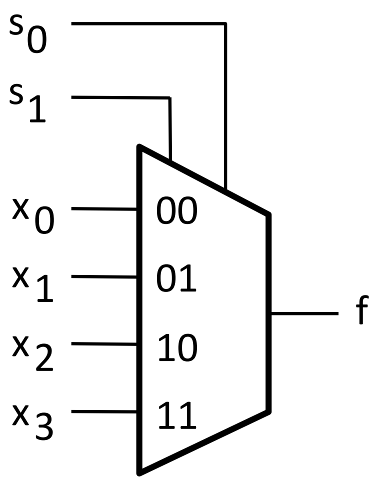

A 4-to-1 multiplexer.

```
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
```

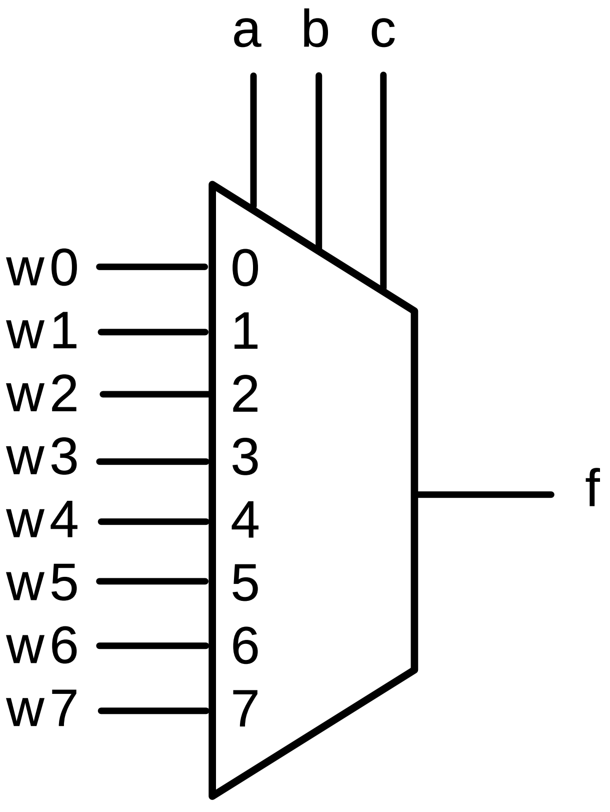



A 4-to-1 multiplexer.

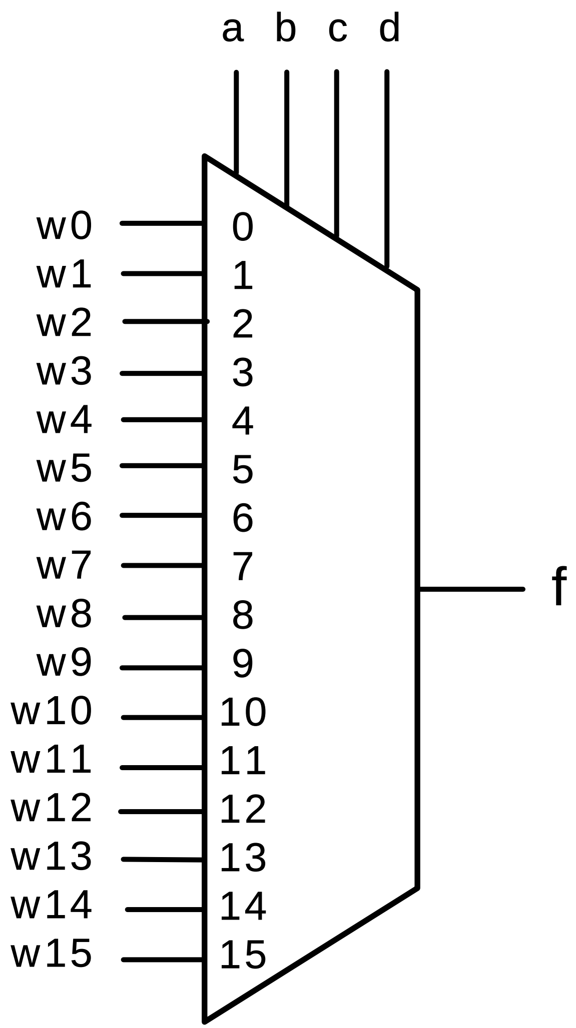
```
module Mux4to1F( input [ 0:3 ] x, input [ 1:0 ] s,
      output f );

   assign f = x[ s ];

endmodule
```



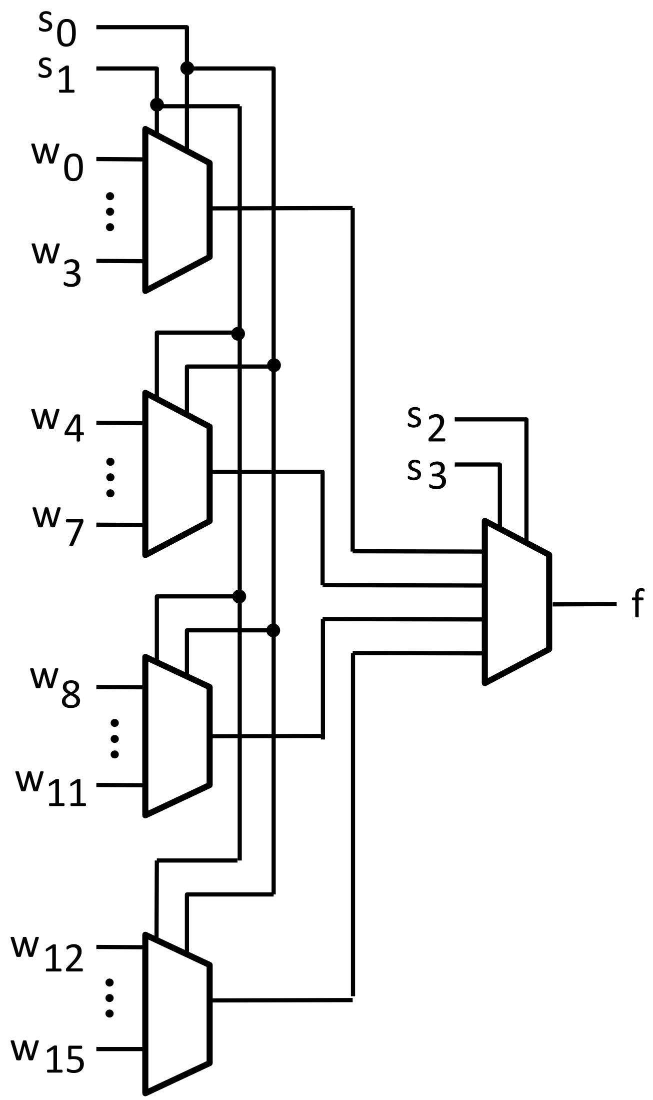

A 4-to-1 multiplexer.




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.

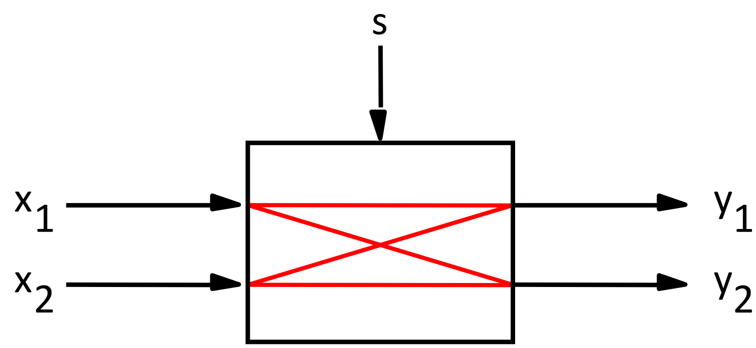
```
module Mux16to1B( input [ 0:15 ] w,
      input [ 3:0 ] s, output reg f );

   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
```

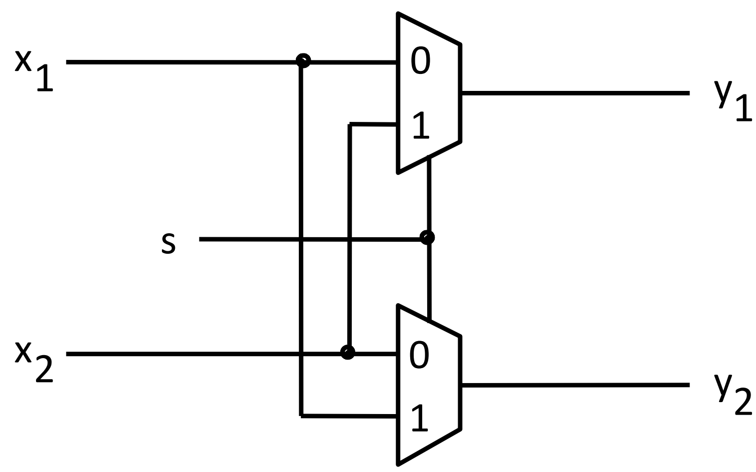
```
module Mux16to1D( input [ 0:15 ] w,
      input [ 3:0 ] s, output reg f );

   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



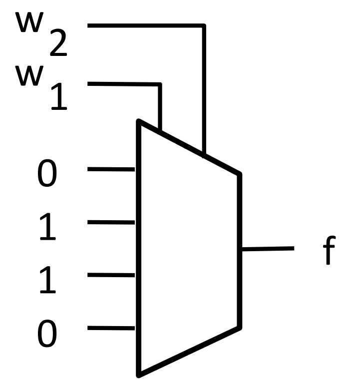

A 2 x 2 crossbar switch




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 |

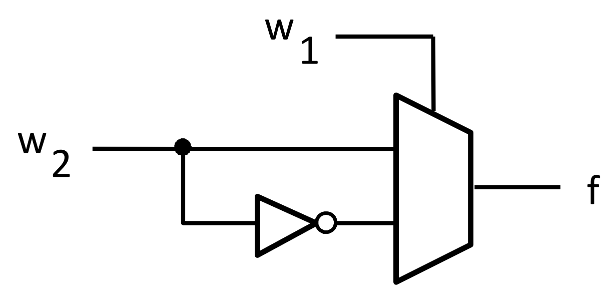



Implementation using a 4-to-1 multiplexer

| $w_1$ | $w_2$ | f |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

| $w_1$ | f |
|---|---|
| 0 | $w_2$ |
| 1 | $w_2'$ |

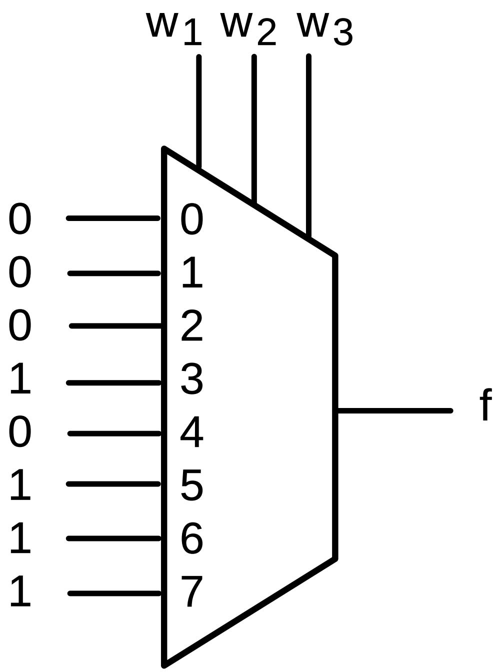
Modified truth table




Circuit

Synthesis of a logic function using multiplexers.

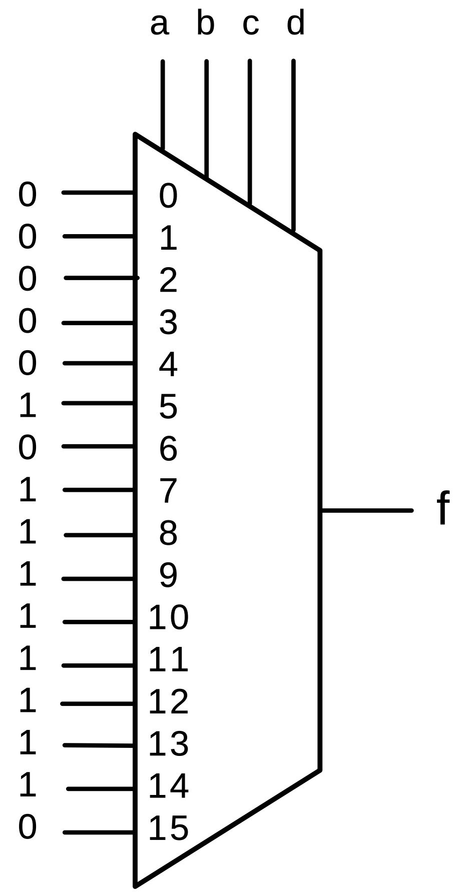
| $w_1$ | $w_2$ | $w_3$ | $f$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |




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

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

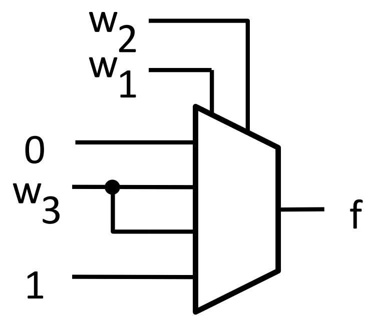
| ab \ cd | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | | | | |
| 01 | | 1 | 1 | |
| 11 | 1 | 1 | | 1 |
| 10 | 1 | 1 | 1 | 1 |



| $w_1$ | $w_2$ | $w_3$ | $f$ |
|---|---|---|---|
| 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 |

| $w_1$ | $w_2$ | $f$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | $w_3$ |
| 1 | 0 | $w_3$ |
| 1 | 1 | 1 |

Modified truth table




Circuit

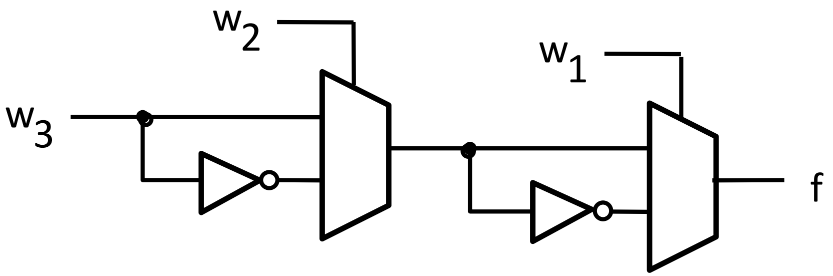
A 3-input majority function using a 4-to-1 multiplexer.

| $w_1$ | $w_2$ | $w_3$ | f |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |

$w_2 \wedge w_3$ (rows with $w_1 = 0$)

$(w_2 \wedge w_3)'$ (rows with $w_1 = 1$)

Truth table

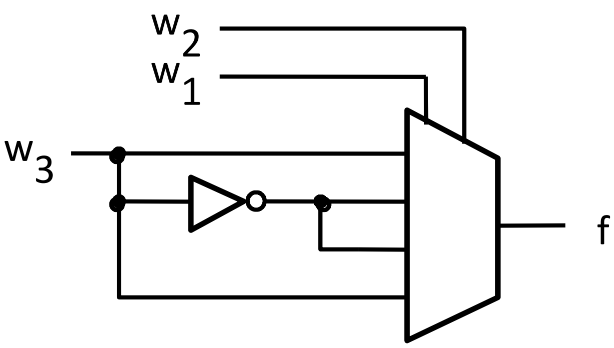



Circuit

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.

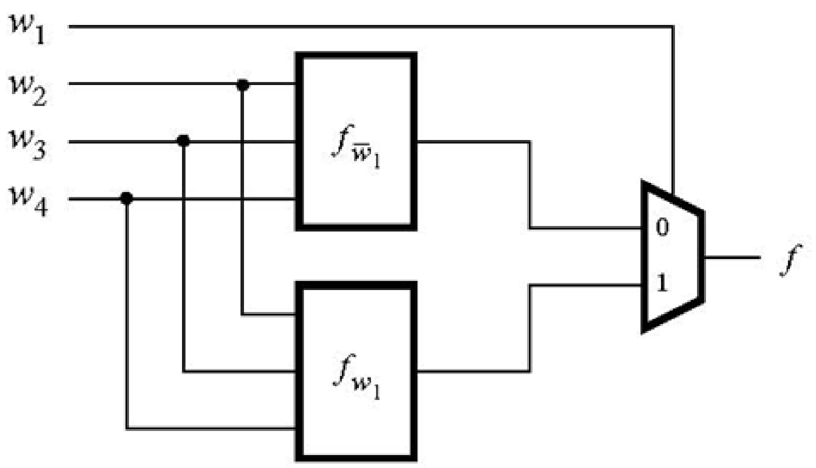
# Shannon's expansion theorem

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 $f_{w1}$



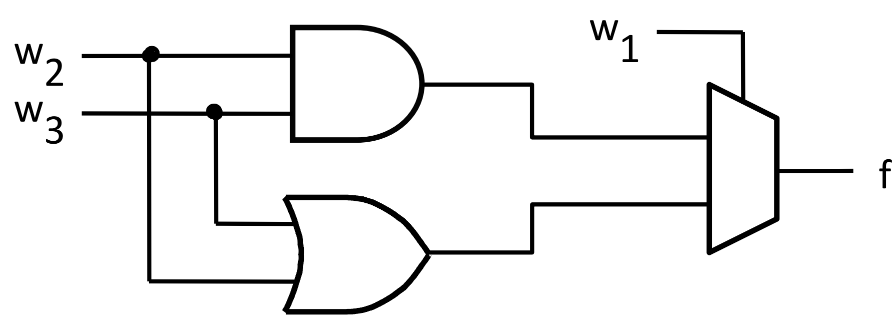

(a) Shannon's expansion of the function *f*.

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

| $w_1$ | $w_2$ | $w_3$ | $f$ |
|---|---|---|---|
| 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 |

| $w_1$ | $f$ |
|---|---|
| 0 | $w_2 w_3$ |
| 1 | $w_2 + w_3$ |

Truth table




Circuit

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

Shannon's expansion for a 2-in mux:

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

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

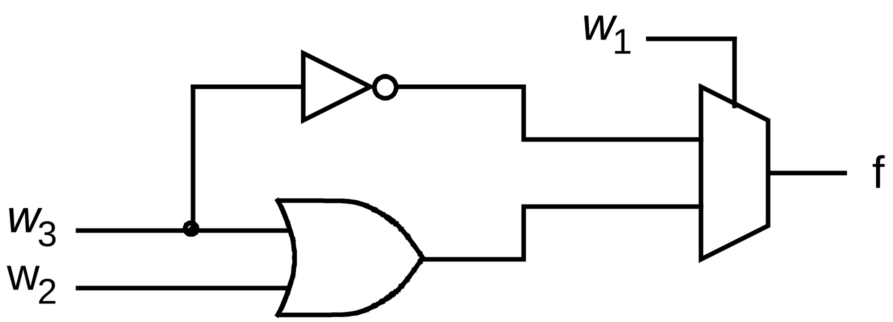
For a 4-in mux, expand again:

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

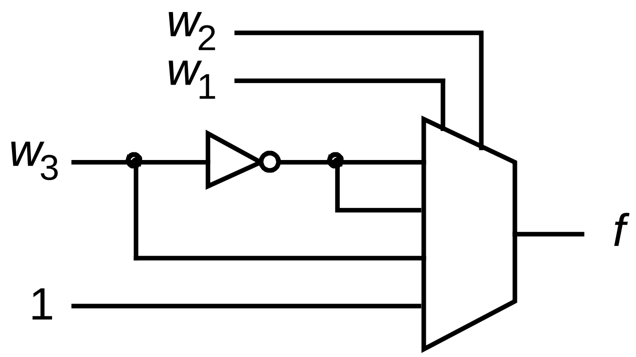
$+ w1 w2' f_{w1 w2'} + w1 w2 f_{w1 w2}$

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

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




(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 )

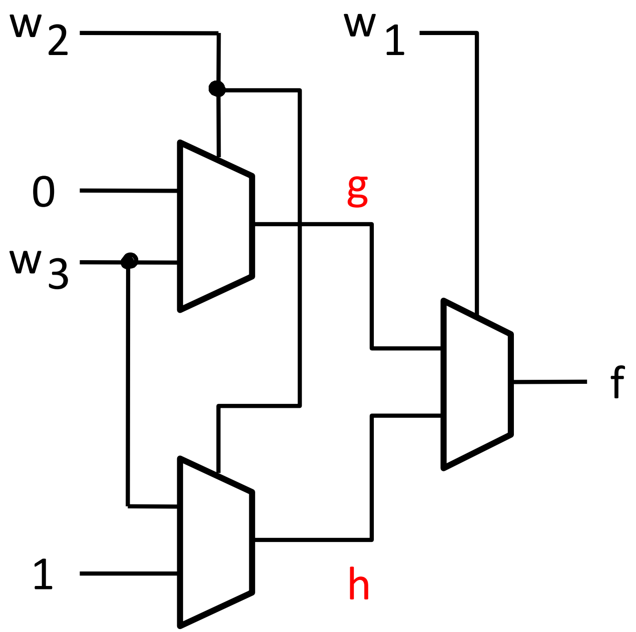



Figure 4.12. A 3-input majority function.