BEE 271 Digital circuits and systems

Spring 2017

Lecture 4: Karnaugh maps and Verilog

Nicole Hamilton

https://faculty.washington.edu/kd1uj

Topics

1. minterms and Maxterms
2. Sum of products (SOP)
3. Product of sums (POS)
4. Karnaugh maps
5. Verilog

Two ways to synthesize a function

Sum of products: Collect all rows where f = 1 using minterms.

Product of sums: Collect all rows where f = 0 using Maxterms.

minterms and Maxterms

A *minterm* is 1 for only one row.

A *Maxterm* is 0 for only one row.

Minterms and maxterms for all possible combinations of 3 variables

Row	x1	x2	x3	Minterm	Maxterm
0	0	0	0	m0 = x1' x2' x3'	M0 = x1 + x2 + x3
1	0	0	1	m1 = x1' x2' x3	M1 = x1 + x2 + x3'
2	0	1	0	m2 = x1' x2 x3'	M2 = x1 + x2' + x3
3	0	1	1	m3 = x1' x2 x3	M3 = x1 + x2' + x3'
4	1	0	0	m4 = x1 x2' x3'	M4 = x1' + x2 + x3
5	1	0	1	m5 = x1 x2' x3	M5 = x1' + x2 + x3'
6	1	1	0	m6 = x1 x2 x3'	M6 = x1' + x2' + x3
7	1	1	1	m7 = x1 x2 x3	M7 = x1' + x2' + x3'

Minterms are small m

Maxterms are big M

Minterms (small m)

Row	x1	x2	x3	Minterm
0	0	0	0	m0 = x1' x2' x3'
1	0	0	1	m1 = x1' x2' x3
2	0	1	0	m2 = x1' x2 x3'
3	0	1	1	m3 = x1' x2 x3
4	1	0	0	m4 = x1 x2' x3'
5	1	0	1	m5 = x1 x2' x3
6	1	1	0	m6 = x1 x2 x3'
7	1	1	1	m7 = x1 x2 x3

A *minterm* is 1 for only one row.

It's an AND expression in which each of the input variables appears once.

Each variable can be in complemented, or uncomplemented, e.g., x' or x.

To match a row in a truth table, use the *uncomplemented* form to match a 1 and the *complemented* form to match a 0.

For example, x1 x2' x3 matches the row where
(x1, x2, x3) = (1, 0, 1)

Maxterms (big M)

Row	x1	x2	x3	Maxterm
0	0	0	0	M0 = x1 + x2 + x3
1	0	0	1	M1 = x1 + x2 + x3'
2	0	1	0	M2 = x1 + x2' + x3
3	0	1	1	M3 = x1 + x2' + x3'
4	1	0	0	M4 = x1' + x2 + x3
5	1	0	1	M5 = x1' + x2 + x3'
6	1	1	0	M6 = x1' + x2' + x3
7	1	1	1	M7 = x1' + x2' + x3'

A *maxterm* is a 0 for only one matching row.

It's an OR expression in which each of the input variables appears once.

Each variable can be in complemented, or uncomplemented, e.g., x' or x.

To match a row in a truth table, use the *complemented* form to match a 1 and the *uncomplemented* form to match a 0.

For example, x1' + x2 + x3' matches the row where
(x1, x2, x3) = (1, 0, 1).

Sum of products

Collect all rows where f = 1 using minterms.

$$f = \sum \left(m_i \bullet f_i \right)$$

Where f_i is the desired result for row i.
If f_i is 0, we can eliminate that term.

Exercise: A 3-variable function we'd like to synthesize

Row	x1	x2	x3	f(x1, x2, x3)
0	0	0	0	0
1	0	0	1	1
2	0	1	0	0
3	0	1	1	0
4	1	0	0	1
5	1	0	1	1
6	1	1	0	1
7	1	1	1	0

$$f = \sum \left(m_i \bullet f_i \right)$$

Using minterms for the rows where we want ones:

Exercise: A 3-variable function we'd like to synthesize

Row	x1	x2	x3	f(x1, x2, x3)
0	0	0	0	0
1	0	0	1	1
2	0	1	0	0
3	0	1	1	0
4	1	0	0	1
5	1	0	1	1
6	1	1	0	1
7	1	1	1	0

$$f = \sum \left(m_i \bullet f_i \right)$$

Using minterms for the rows where we want ones:

f = Σm(1, 4, 5, 6) = m1 + m4 + m5 + m6

= (m1 + m5) + (m4 + m6)

= (x1’ x2’ x3 + x1 x2’ x3) + (x1 x2’ x3’ + x1 x2 x3’)

= (x1’ + x1) x2’ x3 + x1 (x2’ + x2) x3’

= x2’ x3 + x1 x3’

Product of sums

Collect all rows where f = 0 using Maxterms.

$$f = \prod \left(M_i + f_i \right)$$

Where f_i is the desired result for row i.
If f_i is 1, we can eliminate that term.

Exercise: A 3-variable function we'd like to synthesize

Row	x1	x2	x3	f(x1, x2, x3)
0	0	0	0	0
1	0	0	1	1
2	0	1	0	0
3	0	1	1	0
4	1	0	0	1
5	1	0	1	1
6	1	1	0	1
7	1	1	1	0

$$f = \prod \left(M_i + f_i \right)$$

Using Maxterms for the rows where we want zeros:

f =

Exercise: A 3-variable function we'd like to synthesize

Row	x1	x2	x3	f(x1, x2, x3)
0	0	0	0	0
1	0	0	1	1
2	0	1	0	0
3	0	1	1	0
4	1	0	0	1
5	1	0	1	1
6	1	1	0	1
7	1	1	1	0

$$f = \prod \left(M_i + f_i \right)$$

Using Maxterms for the rows where we want zeros:

f = ΠM(0, 2, 3, 7) = M0 • M2 • M3 • M7

Exercise: A 3-variable function we'd like to synthesize

Row	x1	x2	x3	f(x1, x2, x3)
0	0	0	0	0
1	0	0	1	1
2	0	1	0	0
3	0	1	1	0
4	1	0	0	1
5	1	0	1	1
6	1	1	0	1
7	1	1	1	0

$$f = \prod \left(M_i + f_i \right)$$

Using Maxterms for the rows where we want zeros:

f = ΠM(0, 2, 3, 7) = M0 • M2 • M3 • M7

= (x1 + x2 + x3) (x1 + x2’ + x3) (x1 + x2’ + x3’) (x1’ + x2’ + x3’)

Exercise: A 3-variable function we'd like to synthesize

Row	x1	x2	x3	f(x1, x2, x3)
0	0	0	0	0
1	0	0	1	1
2	0	1	0	0
3	0	1	1	0
4	1	0	0	1
5	1	0	1	1
6	1	1	0	1
7	1	1	1	0

$$f = \prod \left(M_i + f_i \right)$$

Using Maxterms for the rows where we want zeros:

f = ΠM(0, 2, 3, 7) = M0 • M2 • M3 • M7

= (x1 + x2 + x3) (x1 + x2’ + x3) (x1 + x2’ + x3’) (x1’ + x2’ + x3’)

= ((x1 + x3) + x2) ((x1 + x3) + x2’) (x1 + (x2’ + x3’)) (x1’ + (x2’ + x3’))

Using Maxterms for the rows where we want zeros:

$f = M0 \bullet M2 \bullet M3 \bullet M7$

$= (x1 + x2 + x3)(x1 + x2' + x3)(x1 + x2' + x3')(x1' + x2' + x3')$

$= ((x1 + x3) + x2) ((x1 + x3) + x2') (x1 + (x2' + x3'))(x1' + (x2' + x3'))$

Combining theorem:

14a.	$x \bullet y + x \bullet y' = x$
14b.	$(x + y) \bullet (x + y') = x$

$f = ((x1 + x3) + x2) ((x1 + x3) + x2') (x1 + (x2' + x3')) (x1' + (x2' + x3'))$

$= (x1 + x3) (x2' + x3')$

Exercise: A 3-variable function we'd like to synthesize

Row	x1	x2	x3	f(x1, x2, x3)
0	0	0	0	0
1	0	0	1	1
2	0	1	0	0
3	0	1	1	0
4	1	0	0	1
5	1	0	1	1
6	1	1	0	1
7	1	1	1	0

$$f = \prod \left(M_i + f_i \right)$$

Using Maxterms for the rows where we want zeros:

f = ΠM(0, 2, 3, 7) = M0 • M2 • M3 • M7

= (x1 + x2 + x3) (x1 + x2’ + x3) (x1 + x2’ + x3’) (x1’ + x2’ + x3’)

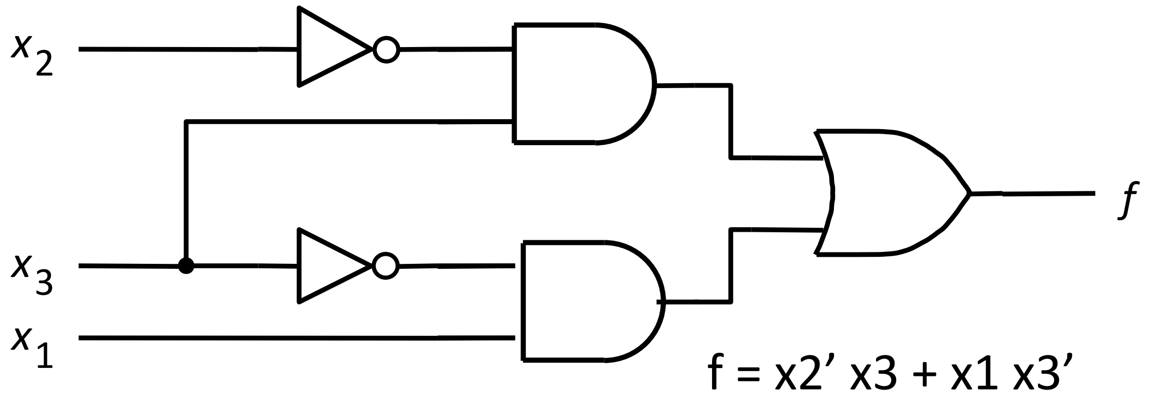
= ((x1 + x3) + x2) ((x1 + x3) + x2’) (x1 + (x2’ + x3’)) (x1’ + (x2’ + x3’))

= (x1 + x3)(x2 + x2’)(x2’ + x3’)(x1 + x1’)

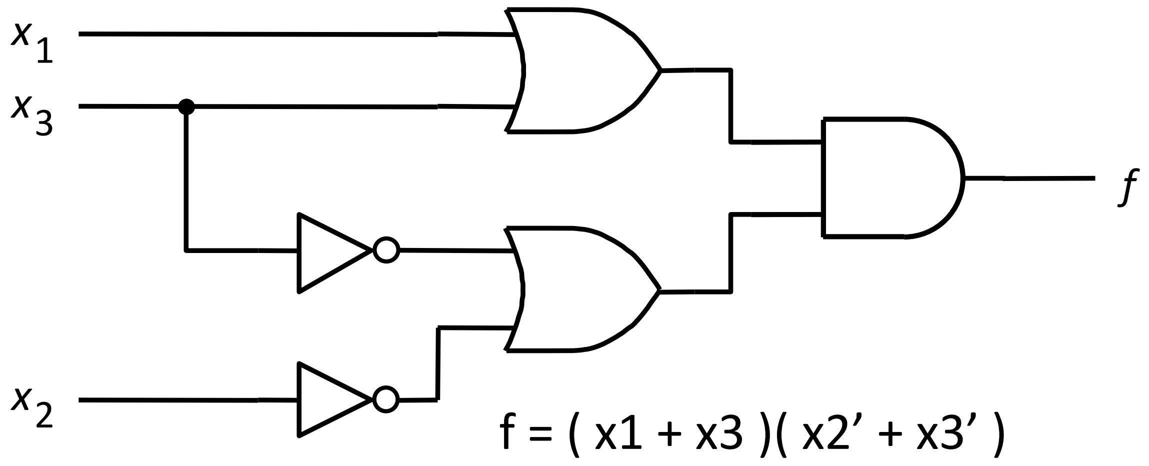
= (x1 + x3)(x2’ + x3’)

Exercise: A 3-variable function we'd like to synthesize

Row	x1	x2	x3	f(x1, x2, x3)
0	0	0	0	0
1	0	0	1	1
2	0	1	0	0
3	0	1	1	0
4	1	0	0	1
5	1	0	1	1
6	1	1	0	1
7	1	1	1	0



(a) A minimal sum-of-products realization



(b) A minimal product-of-sums realization

Figure 2.24. Two realizations of the function.

Karnaugh maps

Want simplest forms but the algebra is difficult.

Karnaugh maps

Invented by Maurice Karnaugh in 1954 as a graphical method for simplifying Boolean equations.

http://www.ithistory.org/sites/default/files/honor-roll/Maurice%20Karnaugh.jpg

Karnaugh maps

Truth table

row	a b	f
0	0 0	
1	0 1	
2	1 0	
3	1 1	

$\Rightarrow$

$$\begin{array}{cc|cc} & & b & \\ & & 0 & 1 \\ \hline a & 0 & \mathit{0} & \mathit{1} \\ & 1 & \mathit{2} & \mathit{3} \end{array}$$

$$\begin{array}{cc|cc} & & a & \\ & & 0 & 1 \\ \hline b & 0 & \mathit{0} & \mathit{2} \\ & 1 & \mathit{1} & \mathit{3} \end{array}$$

Map rows in a truth table to cells in a matrix.
May choose either assignment of columns and rows.

Example

Truth table

row	a b	f
0	0 0	*1*
1	0 1	*1*
2	1 0	*0*
3	1 1	*1*

		b	
		0	1
a	0	*1*	*1*
	1	*0*	*1*

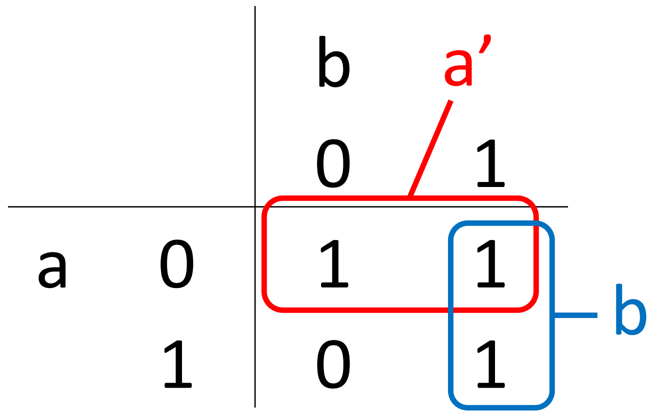
		a	
		0	1
b	0	*1*	*0*
	1	*1*	*1*

Fill in the desired output values.

Truth table

row	a b	f
0	0 0	1
1	0 1	1
2	1 0	0
3	1 1	1

Karnaugh map



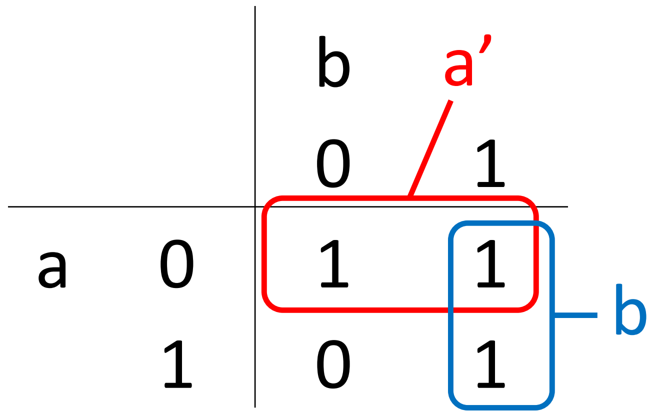
Use the Combining property to group neighboring cells where the output should be the same.

14a. $x \bullet y + x \bullet y' = x$

Truth table

row	a b	f
0	0 0	1
1	0 1	1
2	1 0	0
3	1 1	1

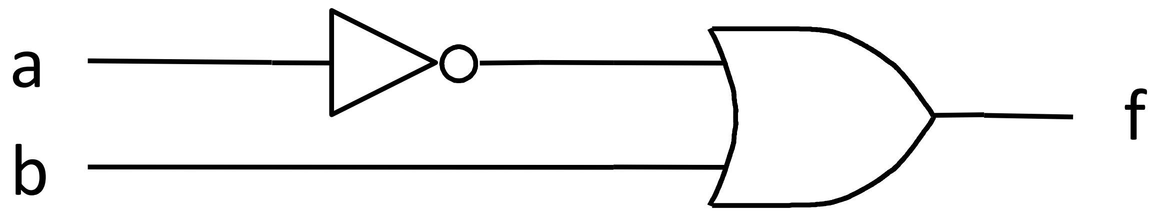
Karnaugh map



		b 0	1
a	0	1	1
	1	0	1

$f = a' + b$

Form the minimal SOP solution.



A function of 3 variables

Truth table

row	a b c	f
0	0 0 0	
1	0 0 1	
2	0 1 0	
3	0 1 1	
4	1 0 0	
5	1 0 1	
6	1 1 0	
7	1 1 1	

Karnaugh map

		bc			
		00	01	11	10
a	0	*0*	*1*	*3*	*2*
	1	*4*	*5*	*7*	*6*

Map the rows to a 2 x 4 matrix.
Columns are arranged so each differs by only 1 bit from the next.

Example

Truth table

row	a b c	f
0	0 0 0	*0*
1	0 0 1	*0*
2	0 1 0	*0*
3	0 1 1	*1*
4	1 0 0	*0*
5	1 0 1	*1*
6	1 1 0	*1*
7	1 1 1	*1*

Karnaugh map

		bc			
		00	01	11	10
a	0	*0*	*0*	*1*	*0*
	1	*0*	*1*	*1*	*1*

Example

Truth table

row	a b c	f
0	0 0 0	0
1	0 0 1	0
2	0 1 0	0
3	0 1 1	1
4	1 0 0	0
5	1 0 1	1
6	1 1 0	1
7	1 1 1	1

Karnaugh map

		bc			
		00	01	11	10
a	0	0	0	1	0
	1	0	1	1	1

f = a b + a c + b c

A function of four variables

Truth table

row	a b c d	f
0	0 0 0 0	
1	0 0 0 1	
2	0 0 1 0	
3	0 0 1 1	
4	0 1 0 0	
5	0 1 0 1	
6	0 1 1 0	
7	0 1 1 1	
8	1 0 0 0	
9	1 0 0 1	
10	1 0 1 0	
11	1 0 1 1	
12	1 1 0 0	
13	1 1 0 1	
14	1 1 1 0	
15	1 1 1 1	

Karnaugh map

		cd			
		00	01	11	10
ab	00	*0*	*1*	*3*	*2*
	01	*4*	*5*	*7*	*6*
	11	*12*	*13*	*15*	*14*
	10	*8*	*9*	*11*	*10*

		ab			
		00	01	11	10
cd	00	*0*	*4*	*12*	*8*
	01	*1*	*5*	*13*	*9*
	11	*3*	*7*	*15*	*11*
	10	*2*	*6*	*14*	*10*

Map the rows to a 4 x 4 matrix either way. (I use the top one.)

Example

Truth table

row	a b c d	f
0	0 0 0 0	*0*
1	0 0 0 1	*0*
2	0 0 1 0	*0*
3	0 0 1 1	*1*
4	0 1 0 0	*1*
5	0 1 0 1	*0*
6	0 1 1 0	*1*
7	0 1 1 1	*0*
8	1 0 0 0	*0*
9	1 0 0 1	*0*
10	1 0 1 0	*0*
11	1 0 1 1	*1*
12	1 1 0 0	*1*
13	1 1 0 1	*0*
14	1 1 1 0	*1*
15	1 1 1 1	*0*

Karnaugh map

		cd			
		00	01	11	10
ab	00	*0*	*0*	*1*	*0*
	01	*1*	*0*	*0*	*1*
	11	*1*	*0*	*0*	*1*
	10	*0*	*0*	*1*	*0*

Copy the outputs to the Karnaugh map.

Example

Truth table

row	a b c d	f
0	0 0 0 0	0
1	0 0 0 1	0
2	0 0 1 0	0
3	0 0 1 1	1
4	0 1 0 0	1
5	0 1 0 1	0
6	0 1 1 0	1
7	0 1 1 1	0
8	1 0 0 0	0
9	1 0 0 1	0
10	1 0 1 0	0
11	1 0 1 1	1
12	1 1 0 0	1
13	1 1 0 1	0
14	1 1 1 0	1
15	1 1 1 1	0

Karnaugh map

		cd			
		00	01	11	10
ab	00	0	0	1	0
	01	1	0	0	1
	11	1	0	0	1
	10	0	0	1	0

f = b d’ + b’ c d

Notice that the Karnaugh map “wraps” vertically and horizontally.

Example

Truth table

row	a b c d	f
0	0 0 0 0	0
1	0 0 0 1	0
2	0 0 1 0	0
3	0 0 1 1	1
4	0 1 0 0	1
5	0 1 0 1	0
6	0 1 1 0	1
7	0 1 1 1	0
8	1 0 0 0	0
9	1 0 0 1	0
10	1 0 1 0	0
11	1 0 1 1	1
12	1 1 0 0	1
13	1 1 0 1	0
14	1 1 1 0	1
15	1 1 1 1	0

Karnaugh map

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

f = b d’ + b’ c d

If we’re collecting 1’s for an SOP solution, we can leave out the 0’s.

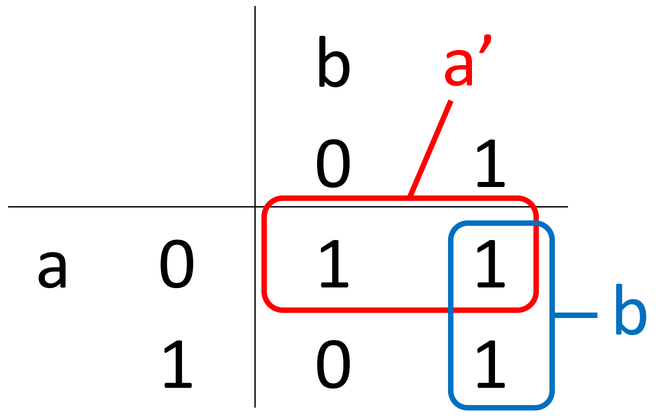
SOP terminology

Literal	A variable or its complement, e.g., x or x'.
Product term	A product, e.g., $x\ y'\ z$, of some number of literals.
Implicant	A product term for which the output is 1. That product term *implies* the output is true.
Prime implicant	An implicant that cannot be combined with another with fewer literals.

Truth table

row	a b	f
0	0 0	1
1	0 1	1
2	1 0	0
3	1 1	1

Karnaugh map



f = a’ + b

Each row or cell where f = 1 is an *implicant*.
The ***prime*** *implicants* are a’ and b.

Cover A collection of implicants that account for all cases for which the output = 1.

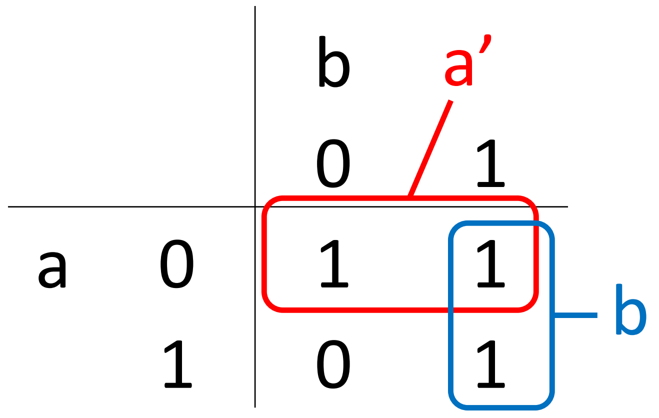
Essential prime implicant

A *prime implicant* that *must* be included in any *cover*.

Truth table

row	a b	f
0	0 0	1
1	0 1	1
2	1 0	0
3	1 1	1

Karnaugh map



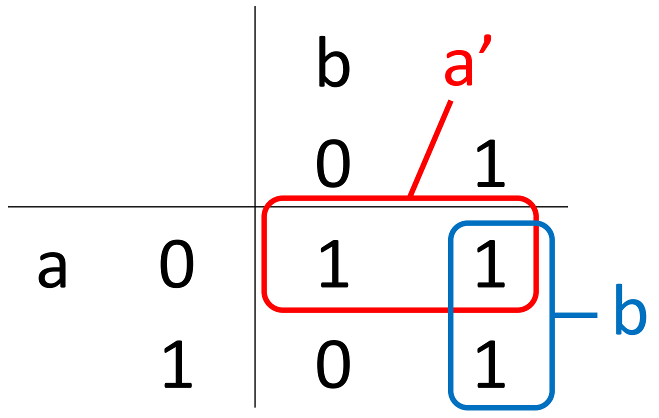
$f = a' + b$

a’ and b form a *cover* for f.
Both are *essential prime implicants*.

Truth table

row	a b	f
0	0 0	1
1	0 1	1
2	1 0	0
3	1 1	1

Karnaugh map



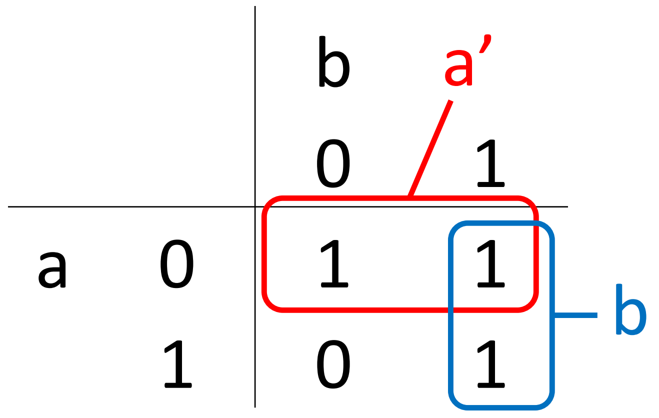
f = a’ + b

For a function of *n* variables, there will be 2^n rows in the truth table and 2^n cells in the Karnaugh map.

Truth table

row	a b	f
0	0 0	1
1	0 1	1
2	1 0	0
3	1 1	1

Karnaugh map



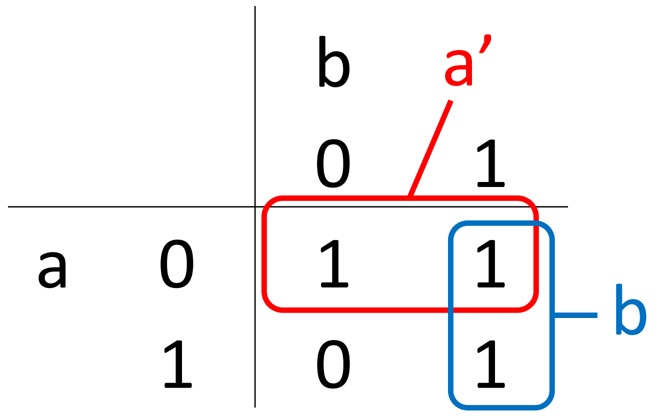
$f = a' + b$

The number of cells in an implicant must be a power of 2.

Truth table

row	a b	f
0	0 0	1
1	0 1	1
2	1 0	0
3	1 1	1

Karnaugh map



f = a' + b

For a function of *n* variables, if an implicant has *k* literals, it must cover 2^{n-k} cells.

Examples

		b	
		0	1
a	0	0	1
	1	0	1

f =

		b	
		0	1
a	0	1	1
	1	0	1

g =

		bc			
		00	01	11	10
a	0	0	1	1	1
	1	0	0	1	0

h =

		bc			
		00	01	11	10
a	0	0	0	1	1
	1	0	0	1	1

j =

Examples

		b	
		0	1
a	0	0	1
	1	0	1

f = b

		b	
		0	1
a	0	1	1
	1	0	1

g =

		bc			
		00	01	11	10
a	0	0	1	1	1
	1	0	0	1	0

h =

		bc			
		00	01	11	10
a	0	0	0	1	1
	1	0	0	1	1

j =

Examples

		b	
		0	1
a	0	0	1
	1	0	1

f = b

		b	
		0	1
a	0	1	1
	1	0	1

g = a’ + b

		bc			
		00	01	11	10
a	0	0	1	1	1
	1	0	0	1	0

h =

		bc			
		00	01	11	10
a	0	0	0	1	1
	1	0	0	1	1

j =

Examples

		b	
		0	1
a	0	0	1
	1	0	1

f = b

		b	
		0	1
a	0	1	1
	1	0	1

g = a’ + b

		bc			
		00	01	11	10
a	0	0	1	1	1
	1	0	0	1	0

h = a’ c + b c + a’ b

		bc			
		00	01	11	10
a	0	0	0	1	1
	1	0	0	1	1

j =

Examples

		b	
		0	1
a	0	0	1
	1	0	1

f = b

		b	
		0	1
a	0	1	1
	1	0	1

g = a’ + b

		bc			
		00	01	11	10
a	0	0	1	1	1
	1	0	0	1	0

h = a’ c + b c + a’ b

		bc			
		00	01	11	10
a	0	0	0	1	1
	1	0	0	1	1

j = b

We can also use Karnaugh maps to help us with Boolean algebra.

Simplify

$k = a' b c' + a b c' + a b c$

Simplify

$k = a'\,b\,c' + a\,b\,c' + a\,b\,c$
$\quad = \Sigma m(\ 2, 6, 7\ )$

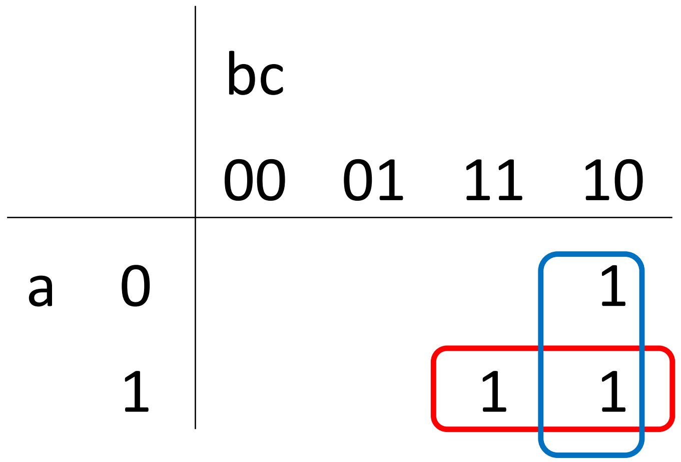
		bc			
		00	01	11	10
a	0				1
	1			1	1

1. Create the Karnaugh map.

Simplify

$k = a'\,b\,c' + a\,b\,c' + a\,b\,c$

$\quad = \Sigma m(2, 6, 7)$

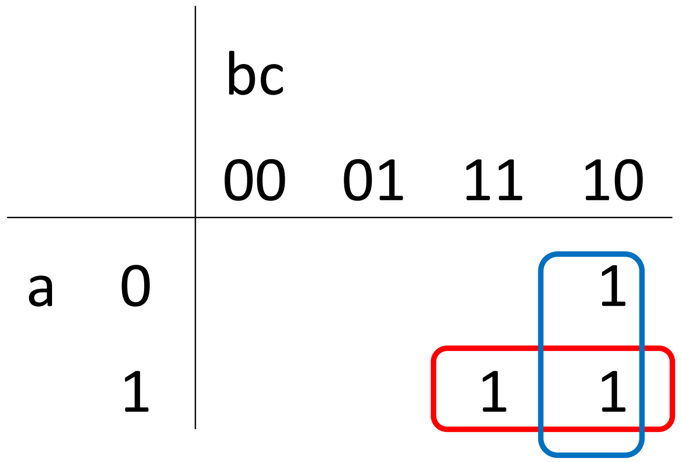


2. Find the prime implicants.

Simplify

$$\begin{aligned} k &= a'\,b\,c' + a\,b\,c' + a\,b\,c \\ &= a'\,b\,c' + a\,b\,c' + a\,b\,c' + a\,b\,c \\ &= b\,c'\,(\,a' + a\,) + a\,b\,(\,c' + c\,) \\ &= b\,c' + a\,b \end{aligned}$$

We'll need to duplicate the middle term.



3. Use this as a guide to solving the algebra.

Simplify

$k = a' b' c' + a' b c' + a b c' + a b c$

Simplify

$k = a' \, b' \, c' + a' \, b \, c' + a \, b \, c' + a \, b \, c$

$\quad = \Sigma m(0, 2, 6, 7)$

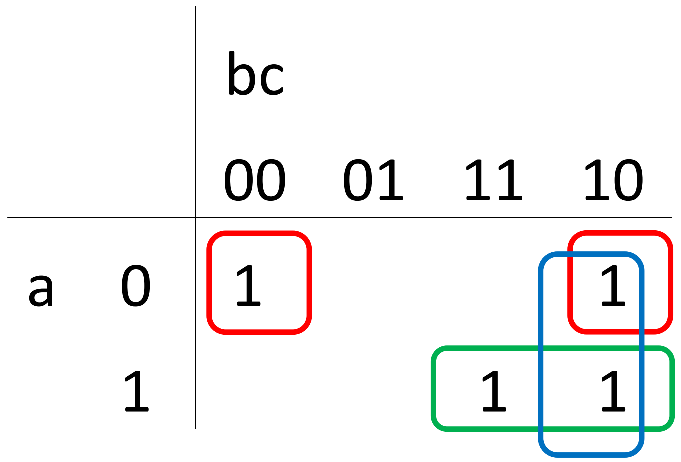
		bc			
		00	01	11	10
a	0	1			1
	1			1	1

1. Create the Karnaugh map.

Simplify

$k = a'\,b'\,c' + a'\,b\,c' + a\,b\,c' + a\,b\,c$

$\quad = \Sigma m(\,0,\,2,\,6,\,7\,)$

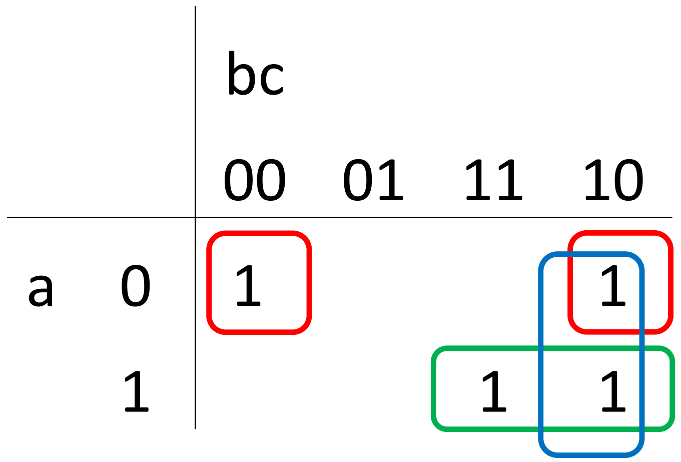


2. Find the prime implicants.

Simplify

$$k = a'\,b'\,c' + a'\,b\,c' + a\,b\,c' + a\,b\,c$$
$$= a'\,(\,b' + b\,)\,c' + a\,b\,(\,c + c'\,)$$
$$= a'\,c' + a\,b$$

b c’ is non-essential.
So we should combine the other terms instead.



3. Use this as a guide to solving the algebra.

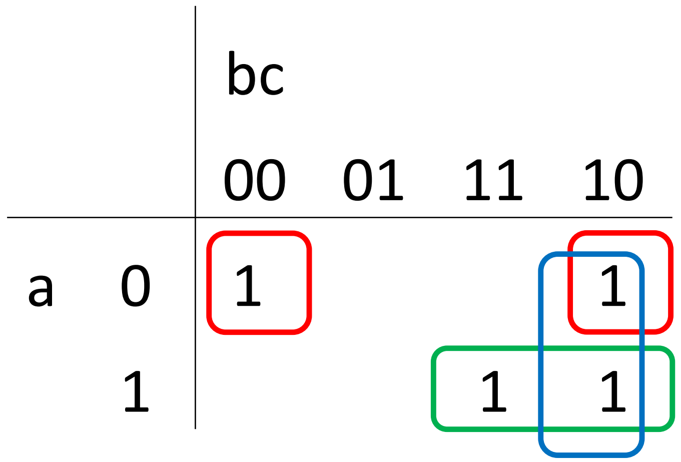
Simplify

$k = a'\,b'\,c' + a'\,b\,c' + a\,b\,c' + a\,b\,c$

$\quad = a'\,(\,b' + b\,)\,c' + a\,b\,(\,c + c'\,)$

$\quad = a'\,c' + a\,b$

$b\,c'$ is non-essential.
So we should combine the other terms instead.



3. Use this as a guide to solving the algebra.

Simplify

$m = a' b' c + a' b c + a b' c' + a b' c$

Simplify

m = a’ b’ c + a’ b c + a b’ c’ + a b’ c
= Σm(1, 3, 4, 5)

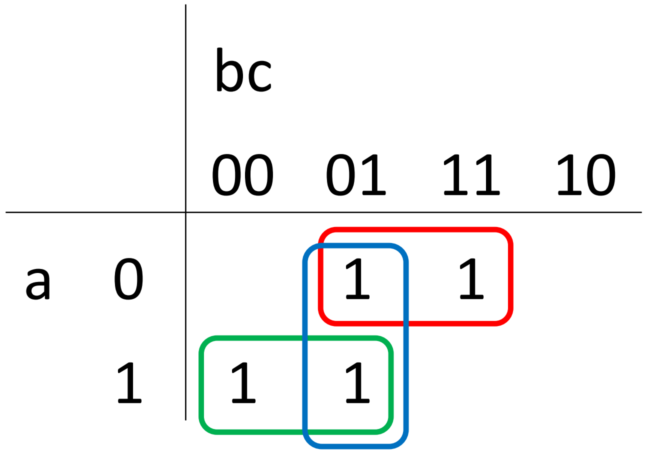
		bc			
		00	01	11	10
a	0		1	1	
	1	1	1		

1. Create the Karnaugh map.

Simplify

m = a’ b’ c + a’ b c + a b’ c’ + a b’ c

  = Σm(1, 3, 4, 5)

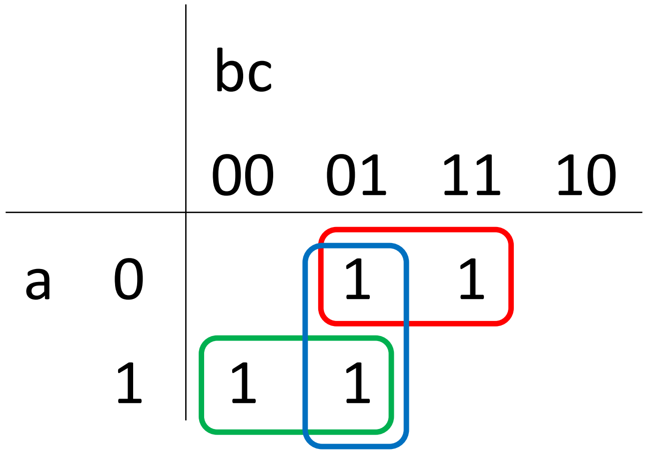


		bc			
		00	01	11	10
a	0		1	1	
	1	1	1		

2. Find the prime implicants.

Simplify

m = a’ b’ c + a’ b c + a b’ c’ + a b’ c
  = a’ (b’ + b) c + a b’ (c + c’)
  = a’ c + a b’



		bc			
		00	01	11	10
a	0		1	1	
	1	1	1		

b’ c is non-essential.
So we should combine the other terms instead.

3. Use this as a guide to solving the algebra.

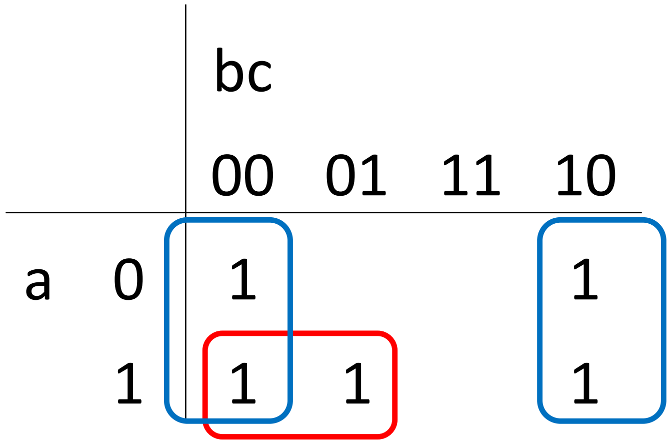
Simplify

$f(a, b, c) = \Sigma m(0, 2, 4, 5, 6)$

Write directly from the Karnaugh map

$f(a, b, c) = \Sigma m(0, 2, 4, 5, 6)$

$= a\,b' + c'$



		bc			
		00	01	11	10
a	0	1			1
	1	1	1		1

Write directly from the Karnaugh map

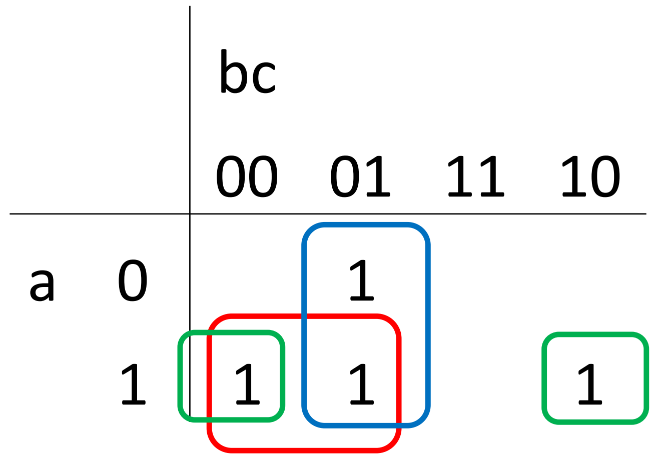
$f(a, b, c) = \Sigma m(1, 4, 5, 6)$

Write directly from the Karnaugh map

$f(a, b, c) = \Sigma m(1, 4, 5, 6)$

$= a\,c' + b'\,c$

$a\,b'$ is not essential.



		bc			
		00	01	11	10
a	0		1		
	1	1	1		1

Examples

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

f =

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

g =

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

h =

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

j =

Examples

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

f = a d’ + a’ b c

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

g =

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

h =

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

j =

Examples

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

f = a d’ + a’ b c

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

g = a + b c

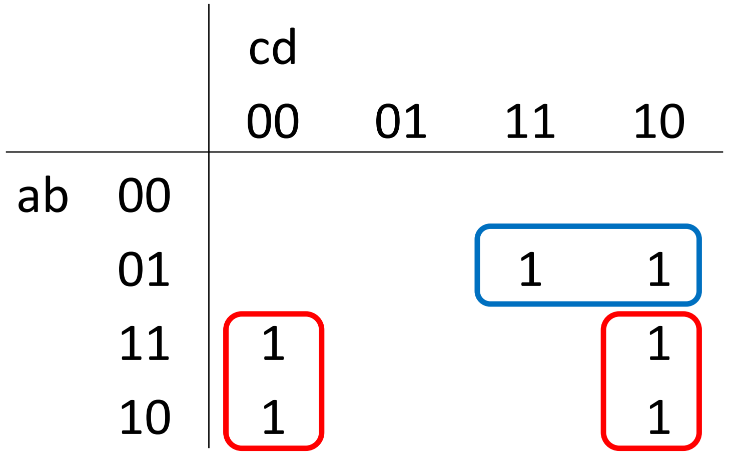
		cd			
		00	01	11	10
ab	00	1			1
	01				
	11	1	1	1	
	10	1	1		1

h =

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

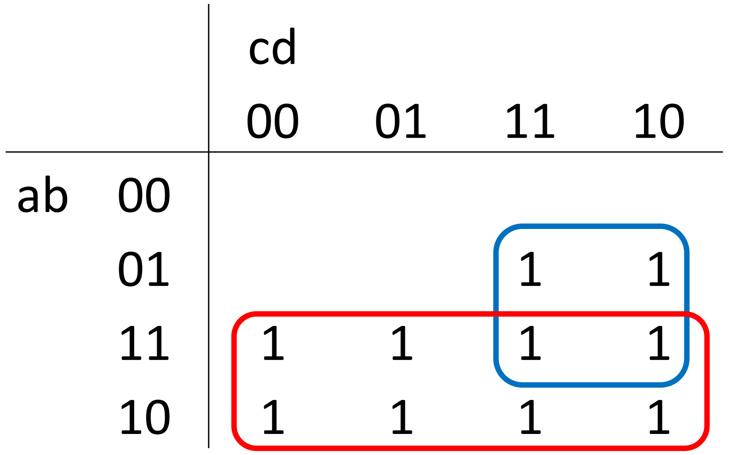
j =

Examples



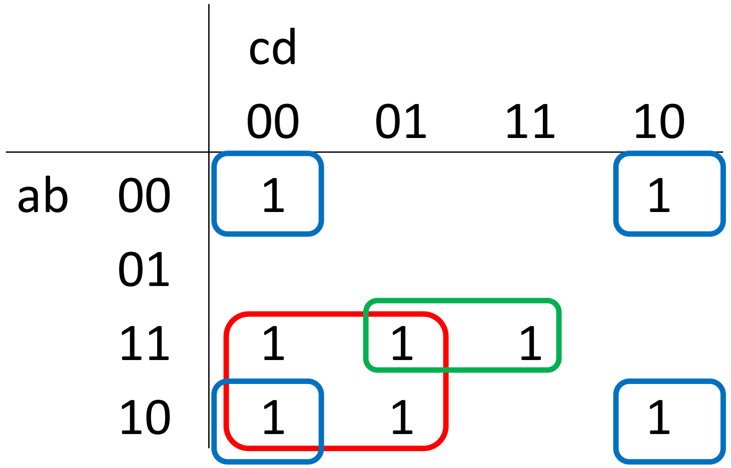
		cd			
		00	01	11	10
ab	00				
	01			1	1
	11	1			1
	10	1			1

f = a d’ + a’ b c



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

g = a + b c



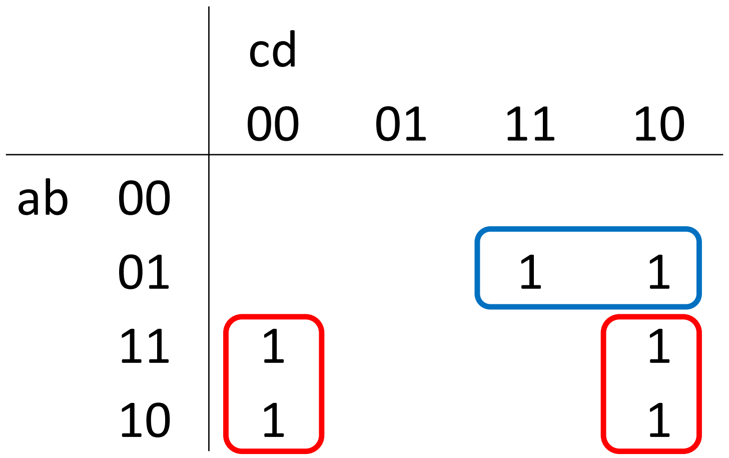
		cd			
		00	01	11	10
ab	00	1			1
	01				
	11	1	1	1	
	10	1	1		1

h = a c’ + b’ d’ + a b d

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

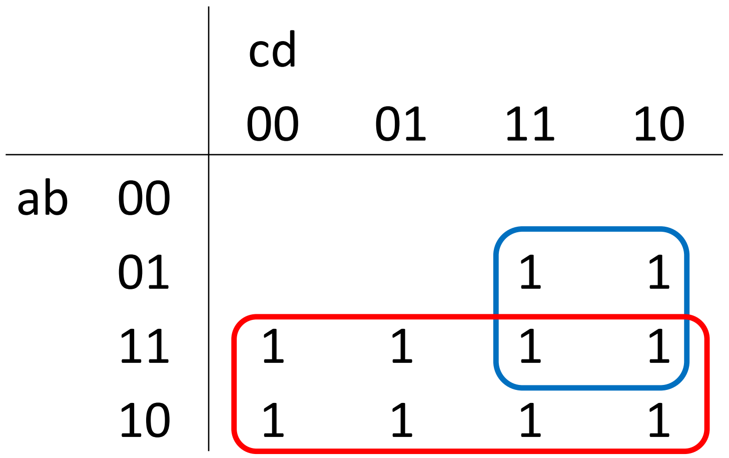
j =

Examples



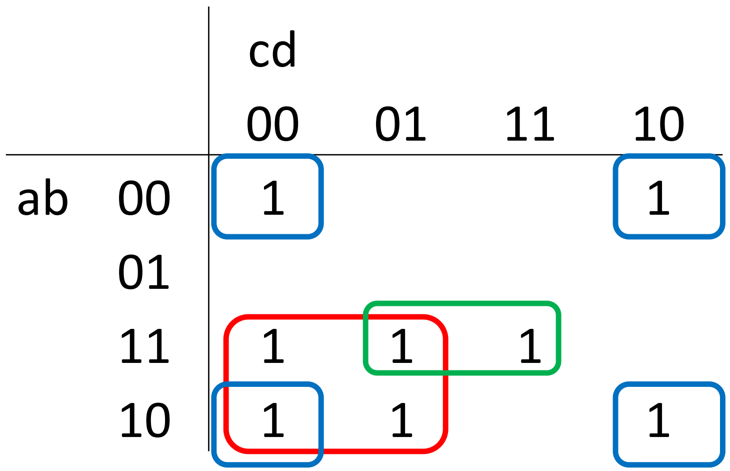
		cd			
		00	01	11	10
ab	00				
	01			1	1
	11	1			1
	10	1			1

f = a d’ + a’ b c



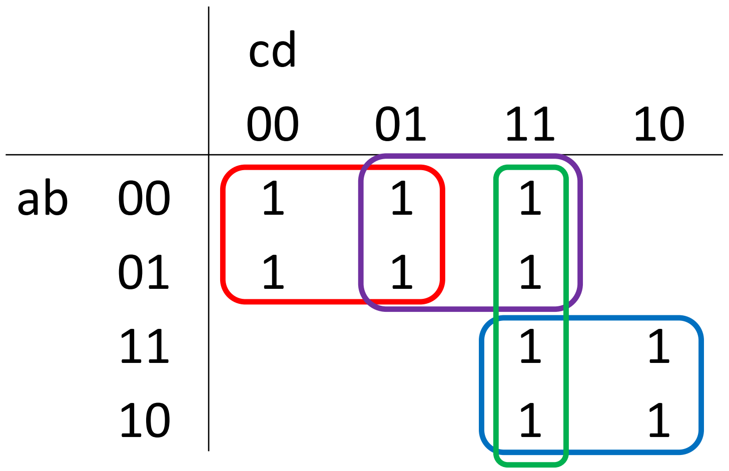
		cd			
		00	01	11	10
ab	00				
	01			1	1
	11	1	1	1	1
	10	1	1	1	1

g = a + b c



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

h = a c’ + b’ d’ + a b d



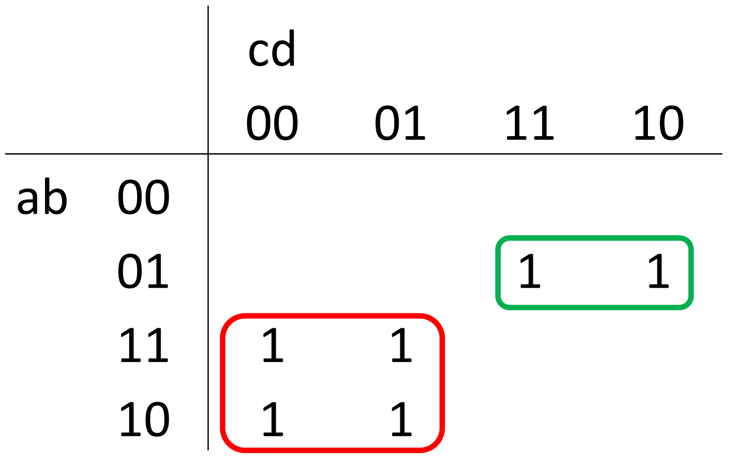
		cd			
		00	01	11	10
ab	00	1	1	1	
	01	1	1	1	
	11			1	1
	10			1	1

Either: j = a’ c’ + a c + c d

j = a’ c’ + a c + a’ d

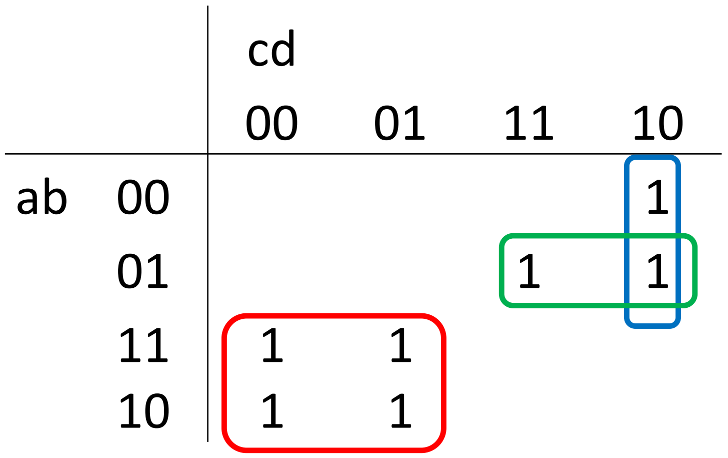
A 5-variable Karnaugh map

Must be done in *layers*.



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

e = 0



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

e = 1

f = a c’ + a’ b c + a’ c d’ e

Realistically, Karnaugh maps are impractical beyond 5 variables. (We turn the problem over to software.)

Often, there are many different networks than can realize a given function.

Some may be simpler than others.

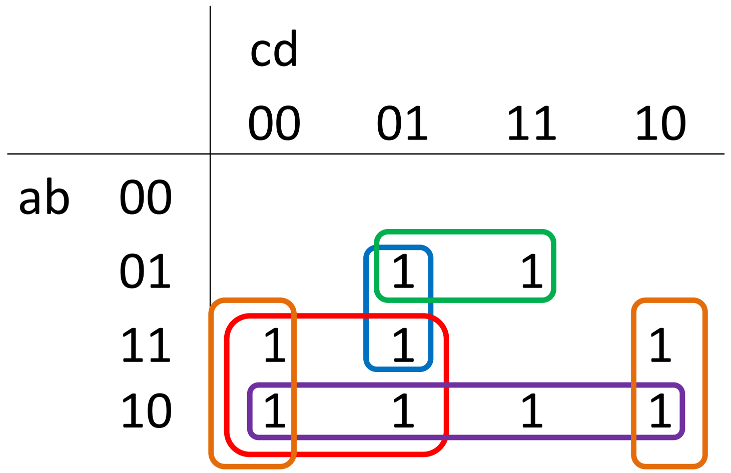
$$f(a, b, c, d) = \Sigma m(5, 7, 8, 9, 10, 11, 12, 13, 14)$$

Not all the implicants are needed.

Which are the essential prime implicants?

What cells are still not covered?

Of the rest, which do you choose?



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

f =

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

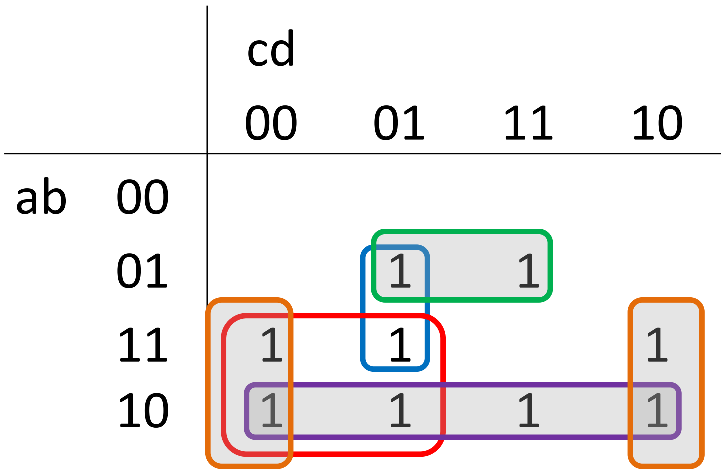
Not all the implicants are needed.

Which are the essential prime implicants?

$a' b d + a b' + a d'$

What cells are still not covered?

Of the rest, which do you choose?



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

f =

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

Not all the implicants are needed.

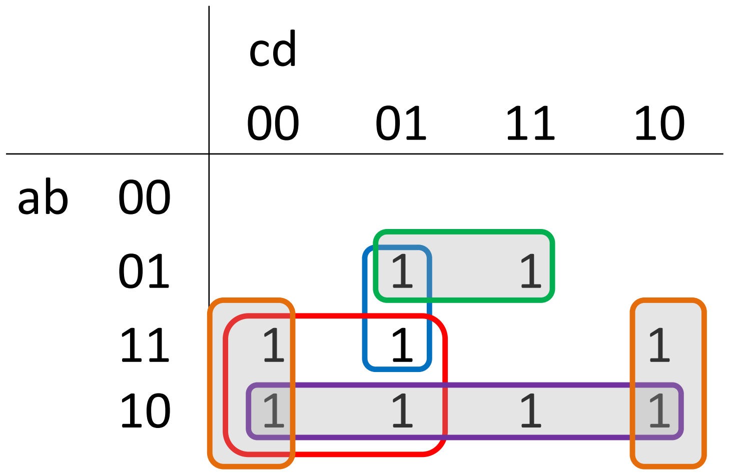
Which are the essential prime implicants?

$a'bd + ab' + ad'$

What cells are still not covered?

Only 1101, covered by either ac' or $bc'd$

Of the rest, which do you choose?



f =

f(a, b, c, d) = Σm(5, 7, 8, 9, 10, 11, 12, 13, 14)

Not all the implicants are needed.

Which are the essential prime implicants?

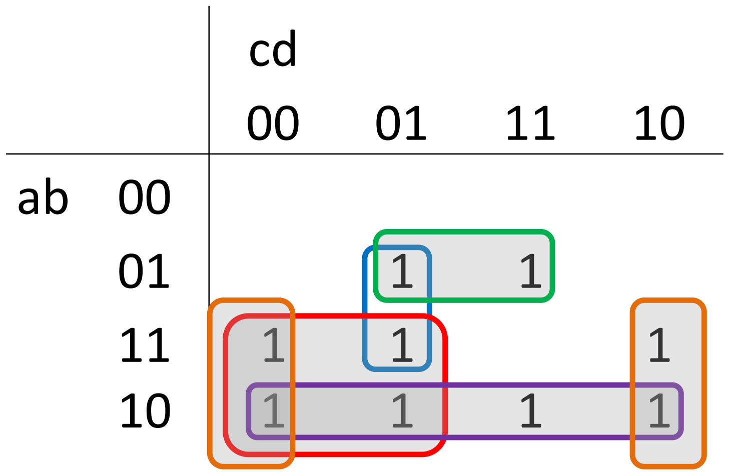
a’ b d + a b’ + a d’

What cells are still not covered?

Only 1101, covered by either a c’ or b c’ d

Of the rest, which do you choose?

Obviously, a c’.



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

f = a’ b d + a b’ + a d’ + a c’

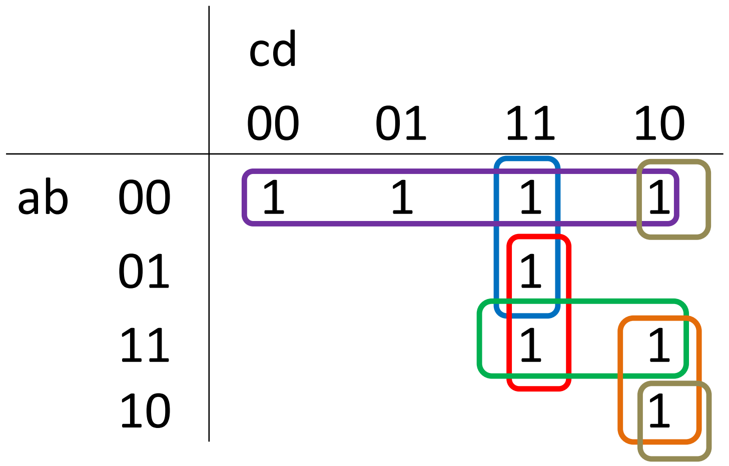
$$f(a, b, c, d) = \Sigma m(0, 1, 2, 3, 7, 10, 14, 15)$$

Not all the implicants are needed.

Which are the essential prime implicants?

What cells are still not covered?

Of the rest, which do you choose?



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

f =

$f(a, b, c, d) = \Sigma m(0, 1, 2, 3, 7, 10, 14, 15)$

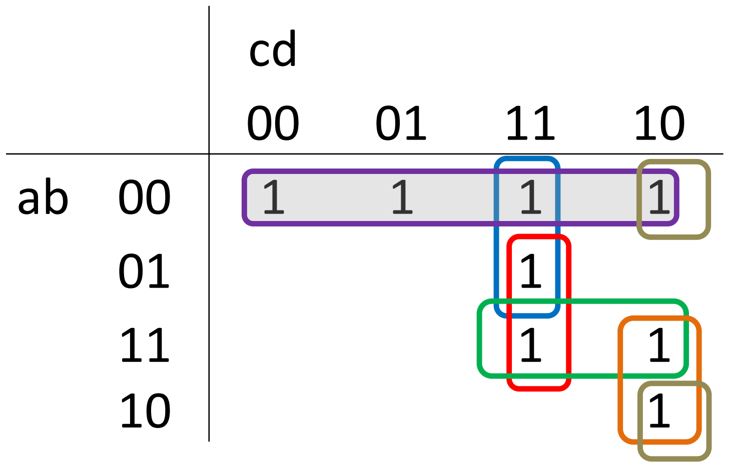
Not all the implicants are needed.

Which are the essential prime implicants?

Only a’ b’

What cells are still not covered?

Of the rest, which do you choose?



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

f =

f(a, b, c, d) = Σm(0, 1, 2, 3, 7, 10, 14, 15)

Not all the implicants are needed.

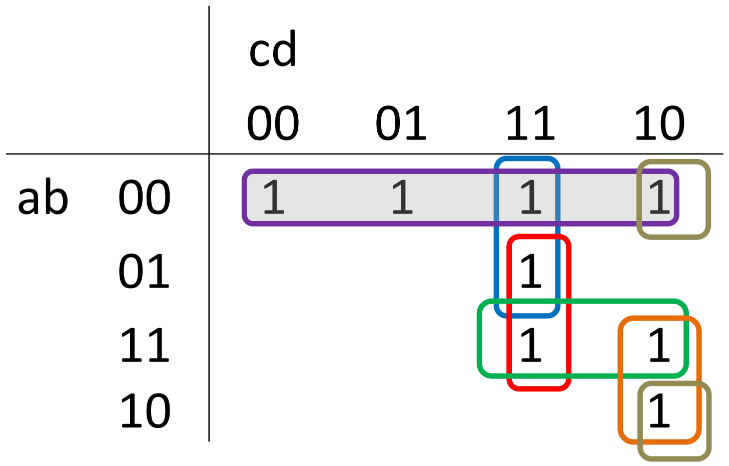
Which are the essential prime implicants?

Only a' b'

What cells are still not covered?

Everything else.

Of the rest, which do you choose?



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

f =

f(a, b, c, d) = Σm(0, 1, 2, 3, 7, 10, 14, 15)

Not all the implicants are needed.

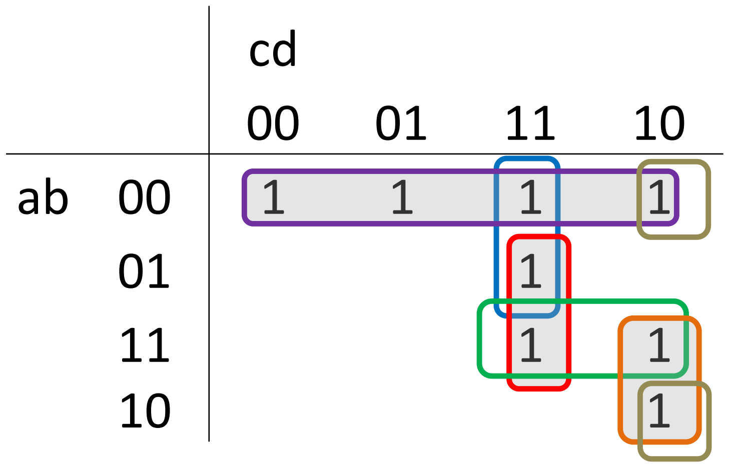
Which are the essential prime implicants?

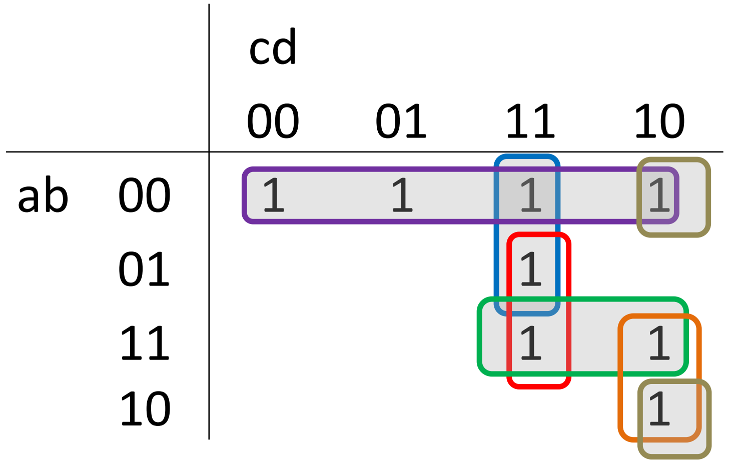
Only a’ b’

What cells are still not covered?

Everything else.

Of the rest, which do you choose?





Two choices.

$f(a, b, c, d) = \Sigma m(0, 1, 2, 3, 7, 10, 14, 15)$

Not all the implicants are needed.

Which are the essential prime implicants?

Only a’ b’

What cells are still not covered?

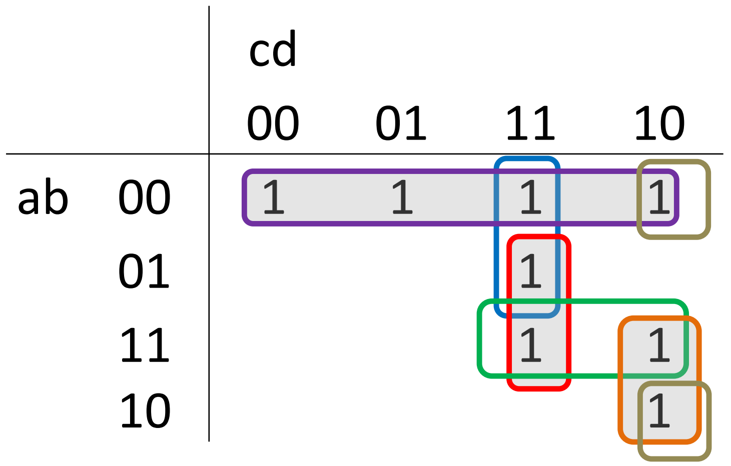
Everything else.

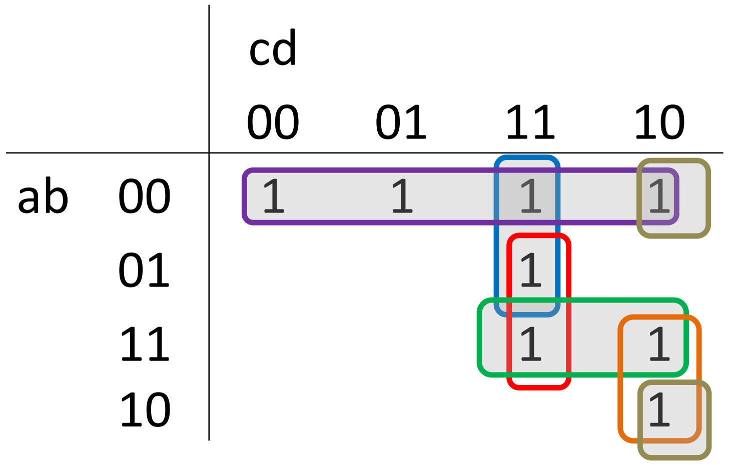
Of the rest, which do you choose?

Two choices.

b c d + a c d’

a’ c d + a b c + b’ c d’





f(a, b, c, d) = Σm(0, 1, 2, 3, 7, 10, 14, 15)

Not all the implicants are needed.

Which are the essential prime implicants?

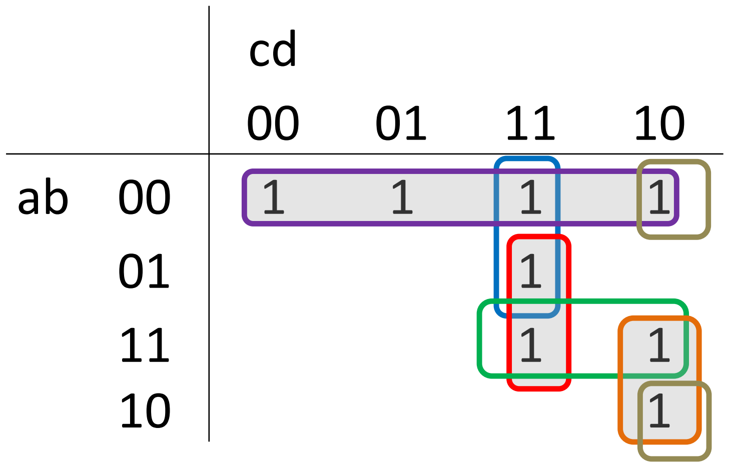
Only a’ b’

What cells are still not covered?

Everything else.

Of the rest, which do you choose?

Best to choose b c d + a c d’.



f = a’ b’ + b c d + a c d’

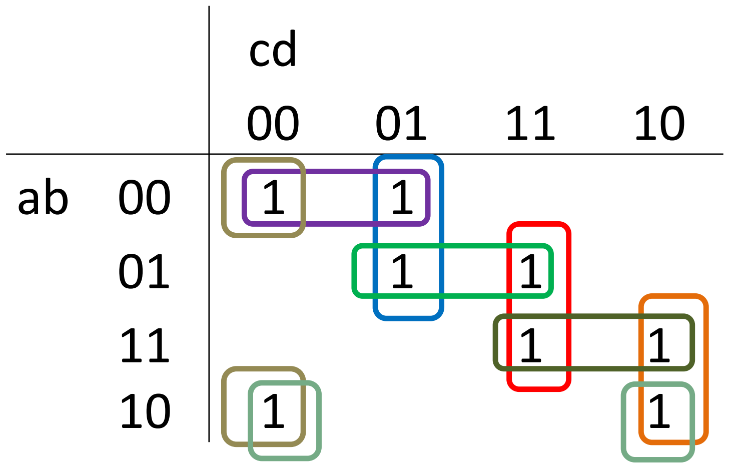
$$f(a, b, c, d) = \Sigma m(0, 1, 5, 7, 8, 10, 14, 15)$$

Not all the implicants are needed.

Which are the essential prime implicants?

What cells are still not covered?

Of the rest, which do you choose?



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

f =

$$f(a, b, c, d) = \Sigma m(0, 1, 5, 7, 8, 10, 14, 15)$$

Not all the implicants are needed.

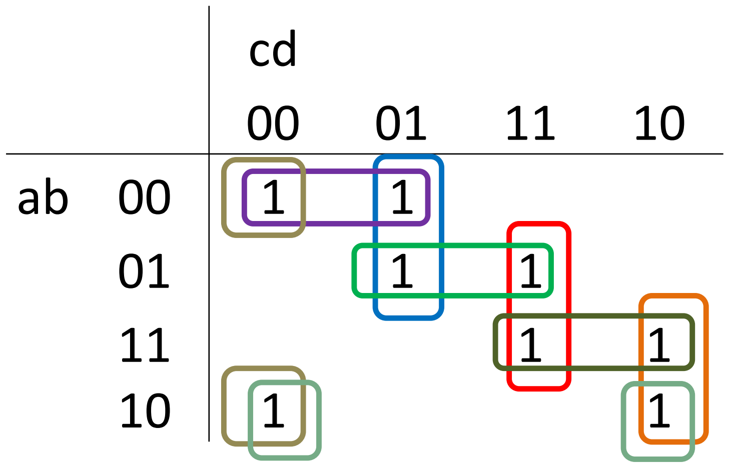
Which are the essential prime implicants?

There are none.

What cells are still not covered?

Everything.

Of the rest, which do you choose?



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

f =

$f(a, b, c, d) = \Sigma m(0, 1, 5, 7, 8, 10, 14, 15)$

Not all the implicants are needed.

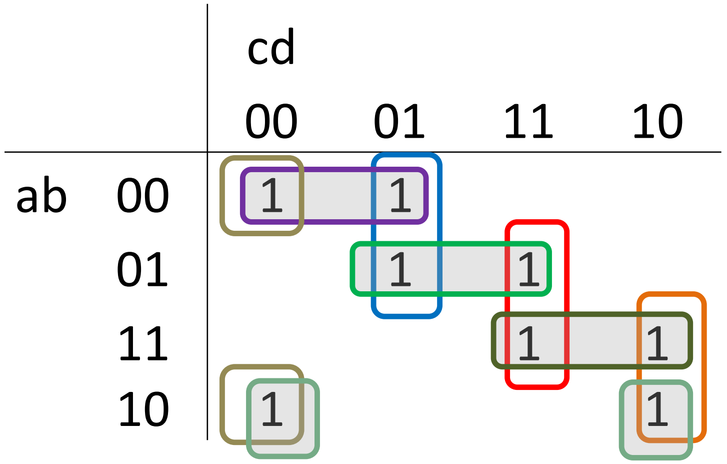
Which are the essential prime implicants?

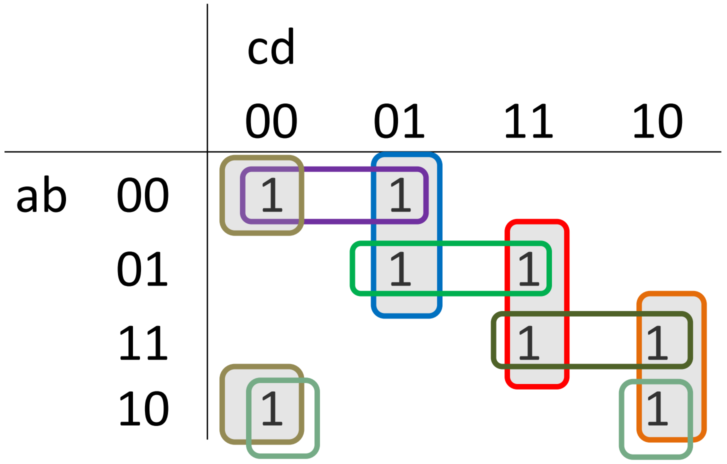
There are none.

What cells are still not covered?

Everything.

Of the rest, which do you choose?





Two choices.

f(a, b, c, d) = Σm(0, 1, 5, 7, 8, 10, 14, 15)

Not all the implicants are needed.

Which are the essential prime implicants?

There are none.

What cells are still not covered?

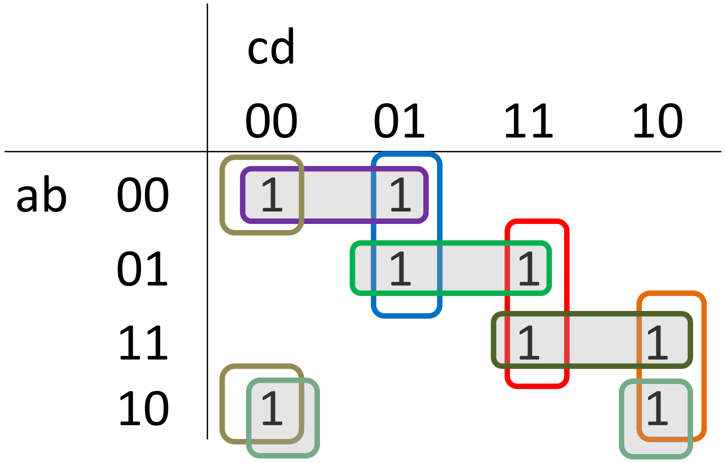
Everything.

Of the rest, which do you choose?

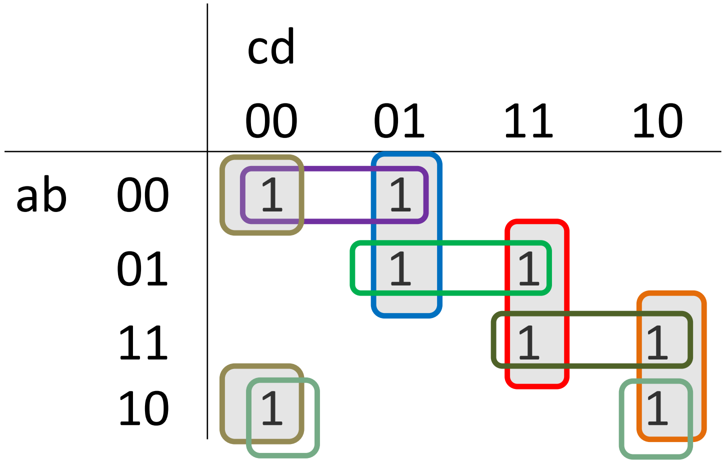
Two equivalent choices.

f = a’ b’ c’ + a’ b d + a b c + a b’ d’

f = b’ c’ d’ + a’ c’ d + b c d + a c d’



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



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

Karnaugh maps can also be used for POS minimization, collecting zeros instead of ones.

$f = \Pi M(2, 3, 6)$

$= (a + b')(b' + c)$

		bc			
		00	01	11	10
a	0			0	0
	1				0

$f = \Pi M(0, 1, 2, 3, 4, 6, 15)$

$= (a + d)(a + b)(a' + b' + c' + d')$

		cd			
		00	01	11	10
ab	00	0	0	0	0
	01	0			0
	11			0	
	10				

Examples

		cd 00	01	11	10
ab	00				
	01			0	0
	11	0			0
	10	0			0

f =

		cd 00	01	11	10
ab	00				
	01			0	0
	11	0	0	0	0
	10	0	0	0	0

g =

		cd 00	01	11	10
ab	00	0			0
	01				
	11	0	0	0	
	10	0	0		0

h =

		cd 00	01	11	10
ab	00	0	0	0	
	01	0	0	0	
	11			0	0
	10			0	0

j =

Examples

		cd			
		00	01	11	10
ab	00				
	01			0	0
	11	0			0
	10	0			0

$f = (a' + d)(a + b' + c')$

		cd			
		00	01	11	10
ab	00				
	01			0	0
	11	0	0	0	0
	10	0	0	0	0

$g =$

		cd			
		00	01	11	10
ab	00	0			0
	01				
	11	0	0	0	
	10	0	0		0

$h =$

		cd			
		00	01	11	10
ab	00	0	0	0	
	01	0	0	0	
	11			0	0
	10			0	0

$j =$

Examples

cd

		00	01	11	10
ab	00				
	01			0	0
	11	0			0
	10	0			0

$f = (a' + d)(a + b' + c')$

cd

		00	01	11	10
ab	00				
	01			0	0
	11	0	0	0	0
	10	0	0	0	0

$g = a'(b' + c')$

cd

		00	01	11	10
ab	00	0			0
	01				
	11	0	0	0	
	10	0	0		0

h =

cd

		00	01	11	10
ab	00	0	0	0	
	01	0	0	0	
	11			0	0
	10			0	0

j =

Examples

		cd			
		00	01	11	10
ab	00				
	01			0	0
	11	0			0
	10	0			0

$f = (a' + d)(a + b' + c')$

		cd			
		00	01	11	10
ab	00				
	01			0	0
	11	0	0	0	0
	10	0	0	0	0

$g = a'(b' + c')$

		cd			
		00	01	11	10
ab	00	0			0
	01				
	11	0	0	0	
	10	0	0		0

$h = (a' + c)(b + d)(a' + b' + d')$

		cd			
		00	01	11	10
ab	00	0	0	0	
	01	0	0	0	
	11			0	0
	10			0	0

$j =$

Examples

		cd			
		00	01	11	10
ab	00				
	01			0	0
	11	0			0
	10	0			0

$f = (a' + d) (a + b' + c')$

		cd			
		00	01	11	10
ab	00				
	01			0	0
	11	0	0	0	0
	10	0	0	0	0

$g = a' (b' + c')$

		cd			
		00	01	11	10
ab	00	0			0
	01				
	11	0	0	0	
	10	0	0		0

$h = (a' + c) (b + d) (a' + b' + d')$

		cd			
		00	01	11	10
ab	00	0	0	0	
	01	0	0	0	
	11			0	0
	10			0	0

$j = (a + c) (a' + c') (c' + d')$

$j = (a + c) (a' + c') (a + d')$

don’t cares

$$f = \Sigma m(1, 5, 8, 9, 10) + D(3, 7, 11, 15)$$

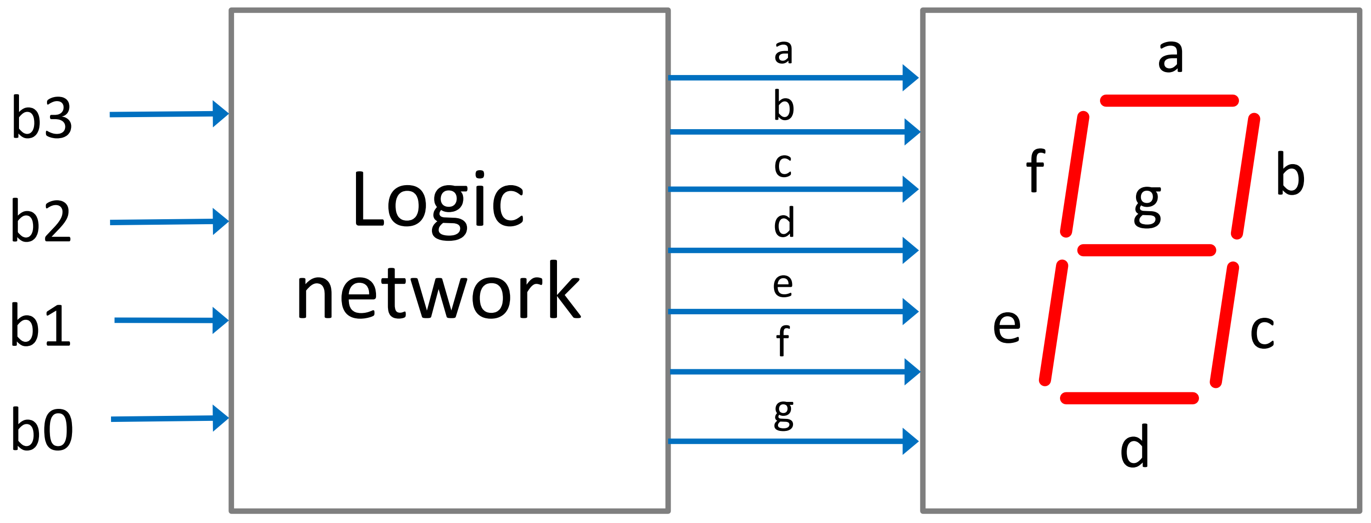
A ***don’t care*** is a cell where we really don’t care whether the output is a 1 or a 0.

We can decide whether to make it a 1 or a 0 depending on which makes for a simpler circuit.

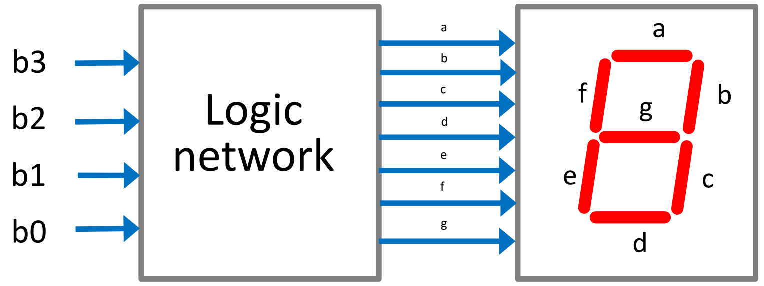
		cd 00	01	11	10
ab	00		1	d	
	01		1	d	
	11			d	
	10	1	1	d	1

$$f = a' d + a b'$$

Example: A seven segment decoder to be used for displaying BCD digits 0 through 9 only.



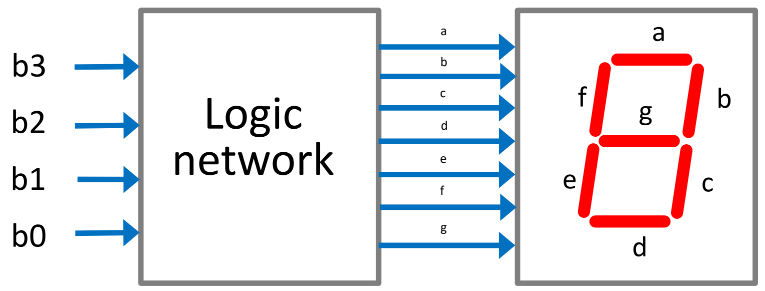
Start with a truth table and a blank Karnaugh map with don’t cares.



decimal	BCD	a b c d e f g
0	0 0 0 0	1 1 1 1 1 1 0
1	0 0 0 1	0 1 1 0 0 0 0
2	0 0 1 0	1 1 0 1 1 0 1
3	0 0 1 1	1 1 1 1 0 0 1
4	0 1 0 0	0 1 1 0 0 1 1
5	0 1 0 1	1 0 1 1 0 1 1
6	0 1 1 0	1 0 1 1 1 1 1
7	0 1 1 1	1 1 1 0 0 0 0
8	1 0 0 0	1 1 1 1 1 1 1
9	1 0 0 1	1 1 1 1 0 1 1

	b1 b0 00	01	11	10
b3 b2 00				
01				
11	d	d	d	d
10			d	d

Using don’t-cares for segment a.



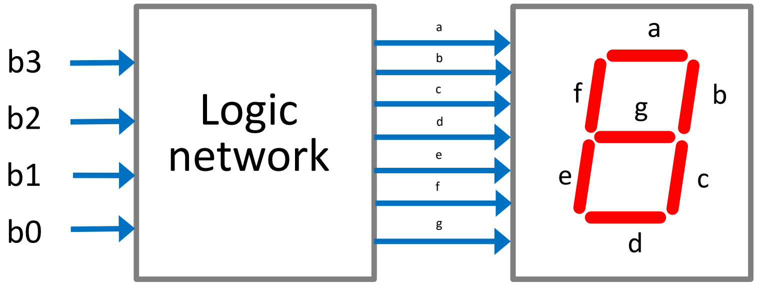
decimal	BCD	a b c d e f g
0	0 0 0 0	1 1 1 1 1 1 0
1	0 0 0 1	0 1 1 0 0 0 0
2	0 0 1 0	1 1 0 1 1 0 1
3	0 0 1 1	1 1 1 1 0 0 1
4	0 1 0 0	0 1 1 0 0 1 1
5	0 1 0 1	1 0 1 1 0 1 1
6	0 1 1 0	1 0 1 1 1 1 1
7	0 1 1 1	1 1 1 0 0 0 0
8	1 0 0 0	1 1 1 1 1 1 1
9	1 0 0 1	1 1 1 1 0 1 1

	b1 b0			
	00	01	11	10
b3 b2 00		$\boxed{0}$		
01	0			
11	d	d	d	d
10			d	d

$$a = \Pi M(1, 4) + D(10, 11, 12, 13, 14, 15)$$

$$= (b2' + b1 + b0)(b3 + b2 + b1 + b0')$$

Using don’t-cares for segment b.



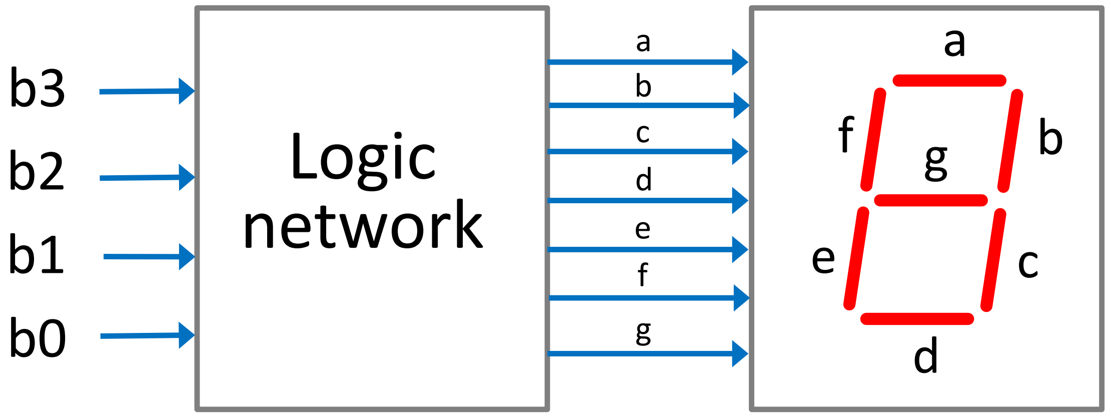
decimal	BCD	a b c d e f g
0	0 0 0 0	1 1 1 1 1 1 0
1	0 0 0 1	0 1 1 0 0 0 0
2	0 0 1 0	1 1 0 1 1 0 1
3	0 0 1 1	1 1 1 1 0 0 1
4	0 1 0 0	0 1 1 0 0 1 1
5	0 1 0 1	1 0 1 1 0 1 1
6	0 1 1 0	1 0 1 1 1 1 1
7	0 1 1 1	1 1 1 0 0 0 0
8	1 0 0 0	1 1 1 1 1 1 1
9	1 0 0 1	1 1 1 1 0 1 1

b3 b2 \ b1 b0	00	01	11	10
00				
01		0		0
11	d	d	d	d
10			d	d

$$b = \Pi M(5, 6) + D(10, 11, 12, 13, 14, 15)$$
$$= (b2' + b1 + b0')\,(b2' + b1' + b0)$$

Continue for c, d, e, f and g.

Multiple-input / multiple-output problems



Often offer opportunities to share terms.

(But they can be difficult to find by hand.)

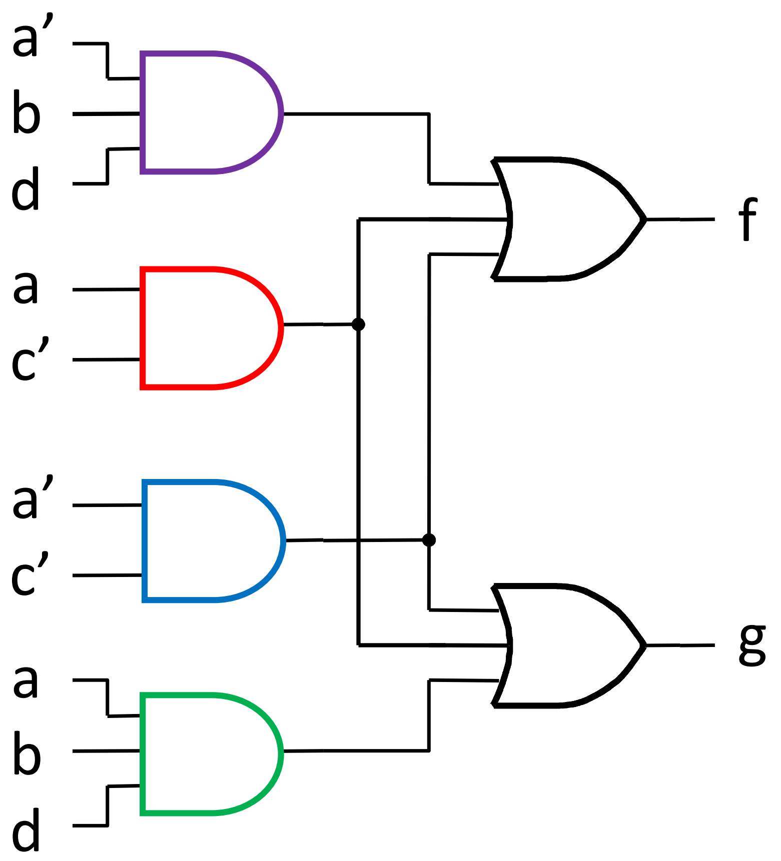
Example: Simple sharing of terms

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

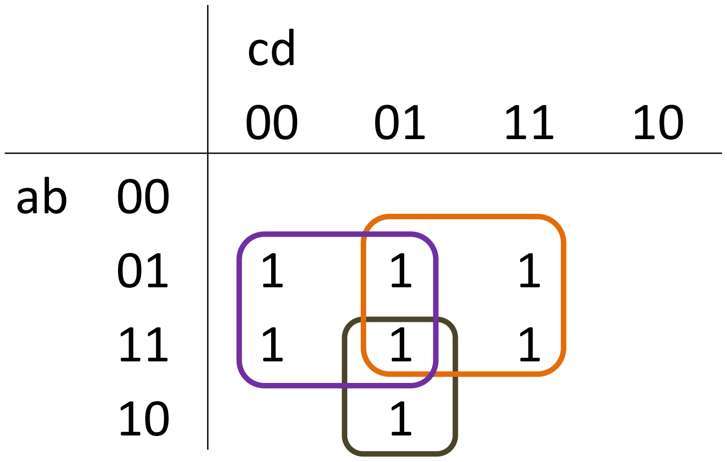
$f = a\,c' + a'\,c + a'\,b\,d$

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

$g = a\,c' + a'\,c + a\,b\,d$

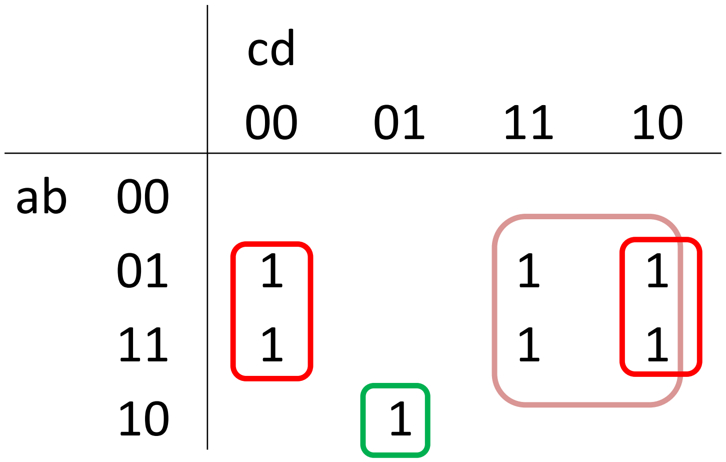


Example: Sharing requiring joint optimization



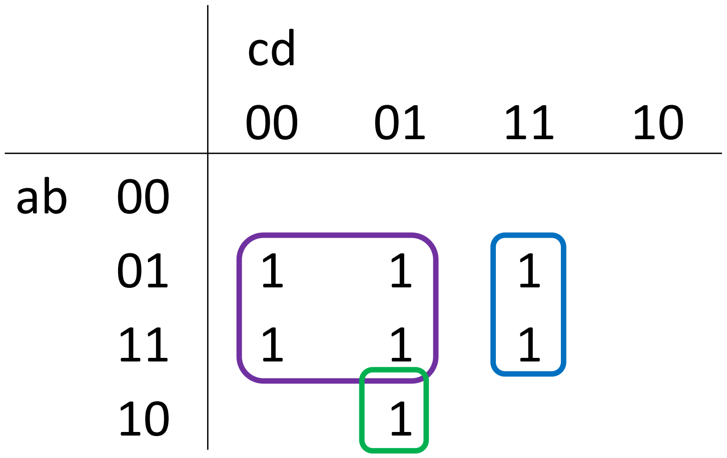
ab \ cd	00	01	11	10
00				
01	1	1	1	
11	1	1	1	
10		1		

Individually optimized h



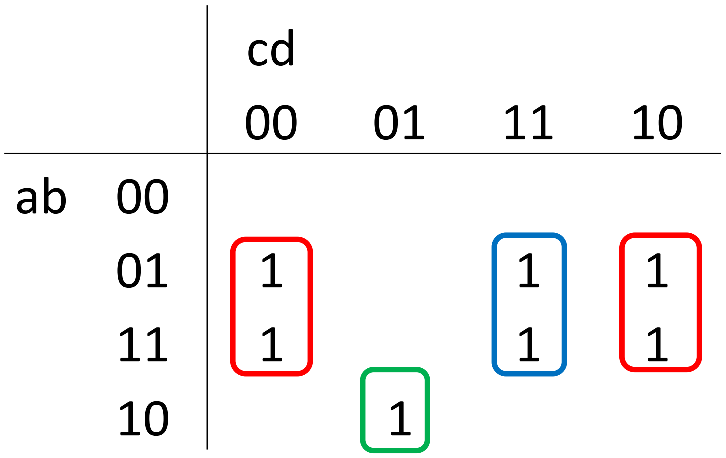
ab \ cd	00	01	11	10
00				
01	1		1	1
11	1		1	1
10		1		

Individually optimized j



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

Jointly optimized h



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

Jointly optimized j

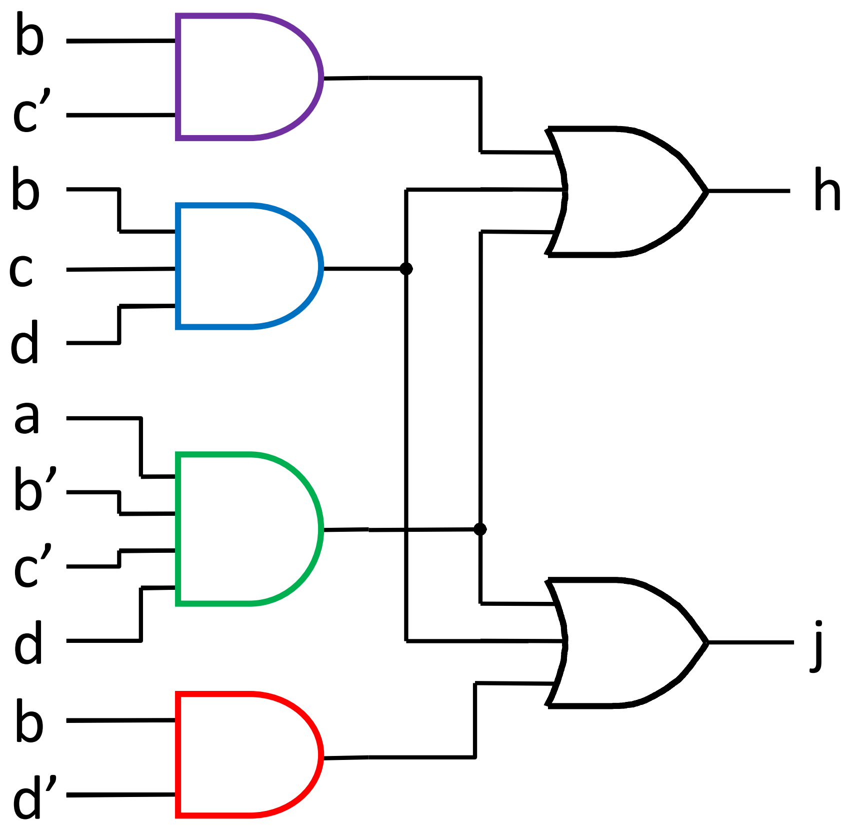
Example: Sharing requiring joint optimization

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

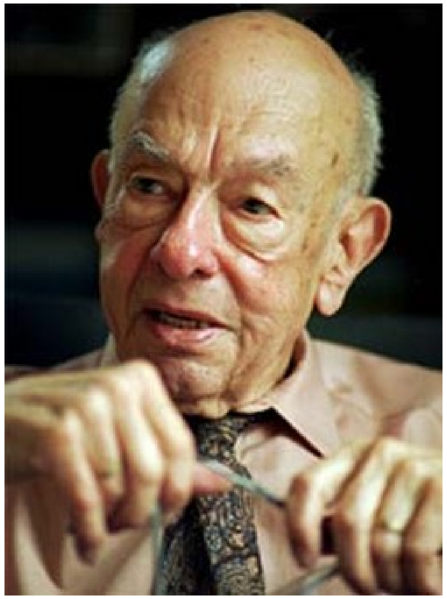
$h = b\,c' + b\,c\,d + a\,b'\,c'\,d$

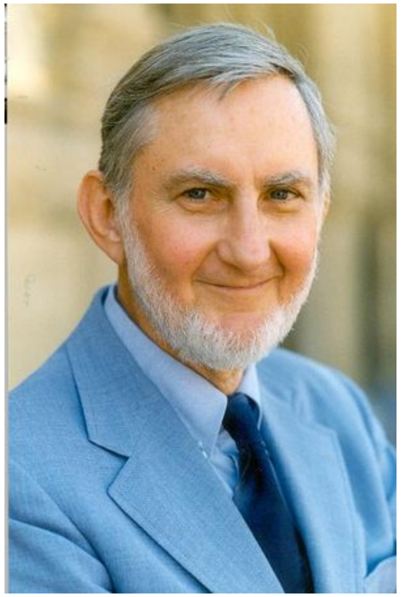
ab \ cd	00	01	11	10
00				
01	1		1	1
11	1		1	1
10		1		

$j = b\,d' + b\,c\,d + a\,b'\,c'\,d$



Multiple-input, multiple-output minimization


Willard Van Orman Quine


Edward J. McCluskey

One of the first algorithmic methods was *Quine-McCluskey minimization* invented in 1952, but the runtime grows exponentially with the number of variables.

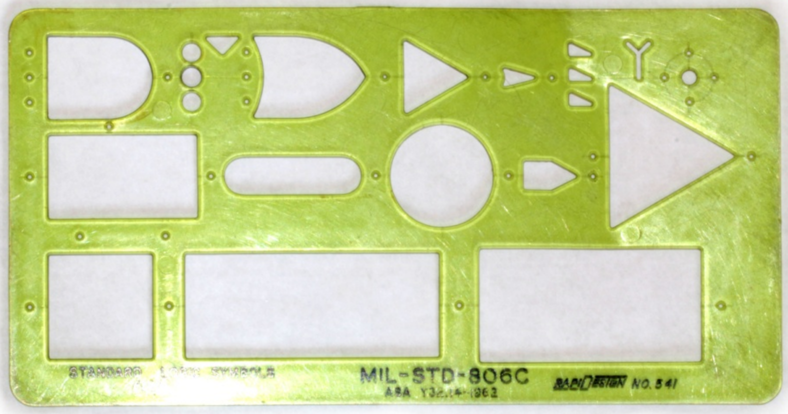
Today, we turn the problem over to compilers.

Image sources: http://ethw.org/Edward_McCluskey
http://www.philosophybasics.com/philosophers_quine.html

Verilog

40 years ago

BRUNING


STANDARD LOGIC SYMBOLS
MIL-STD-806C
ASA Y32.14-1962
RAPIDESIGN NO. 541

Today

We no longer draw gates for complex designs.

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

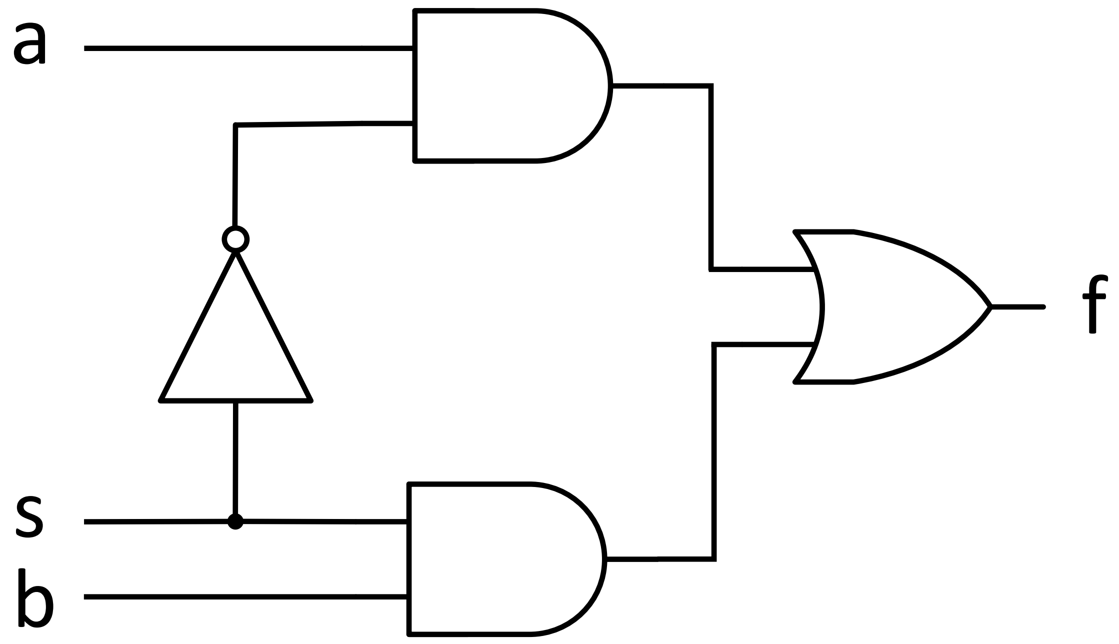
HDL

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.

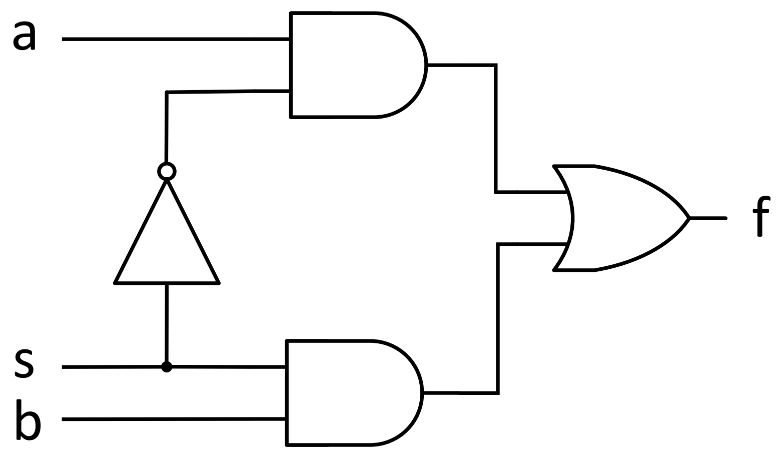
The Multiplexer



s	f
0	a
1	b

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

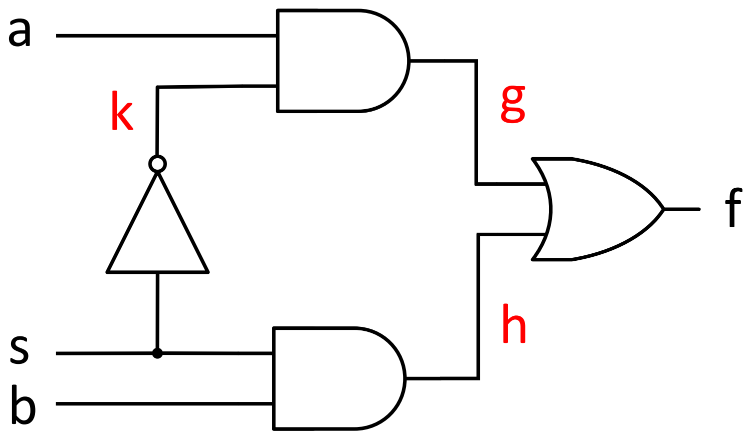
Three ways to represent the multiplexer in Verilog



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

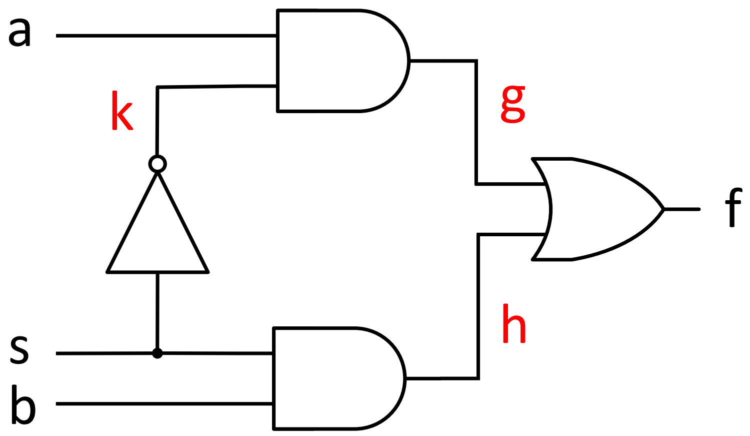
1. Structural representation

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
```

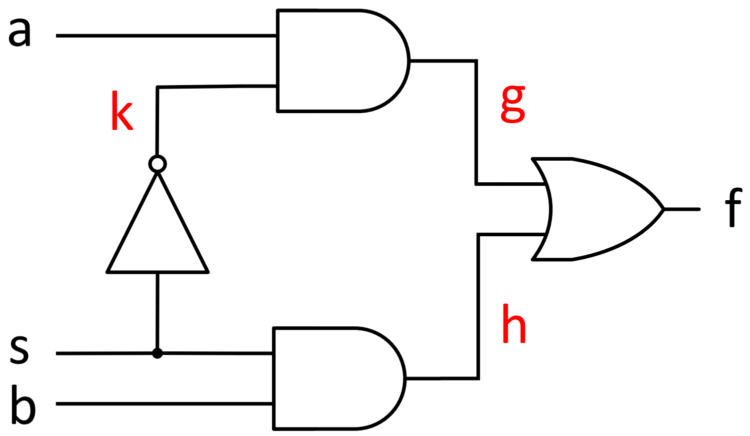
Verilog code for a multiplexer.



```
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”

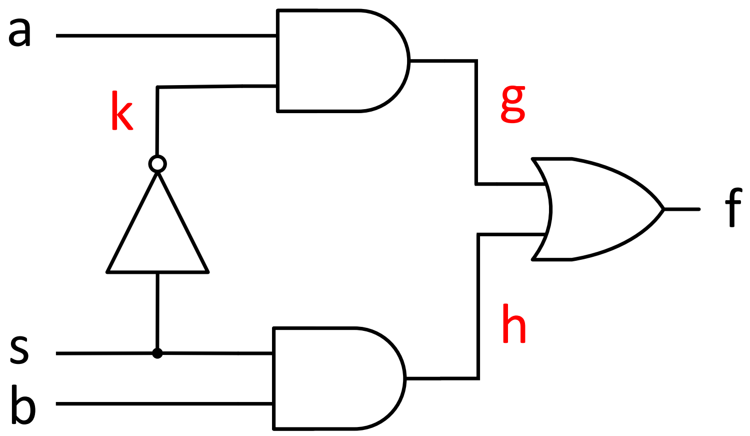
Verilog code for a multiplexer.



```
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.

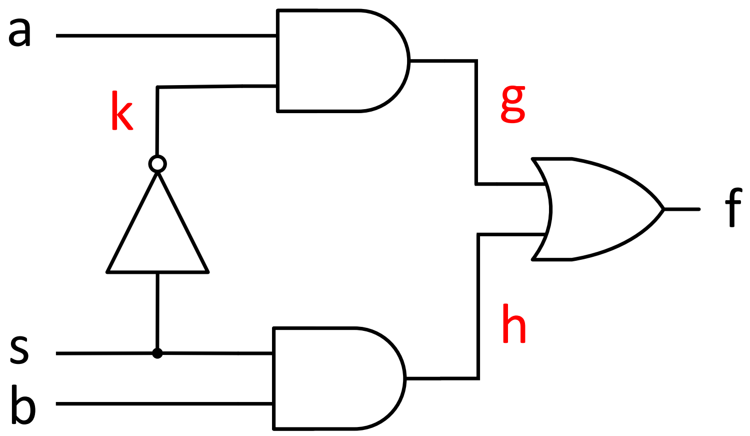
Verilog code for a multiplexer.



```
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.

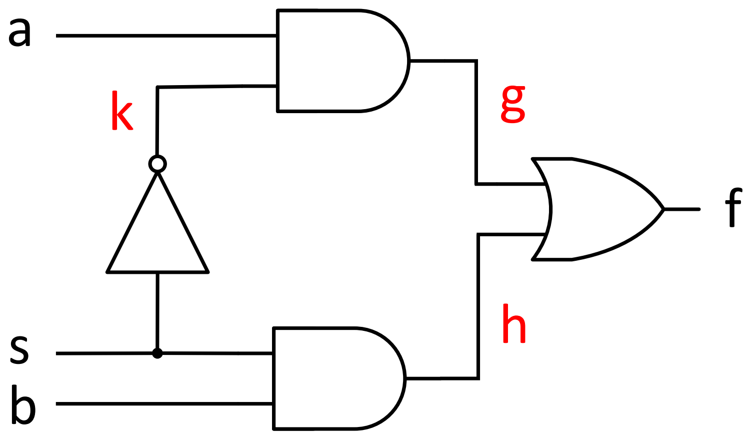
Verilog code for a multiplexer.



```
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.

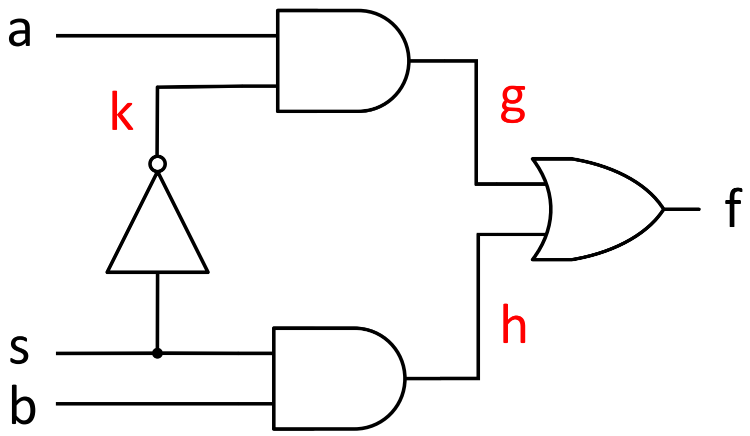
Verilog code for a multiplexer.



```
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.

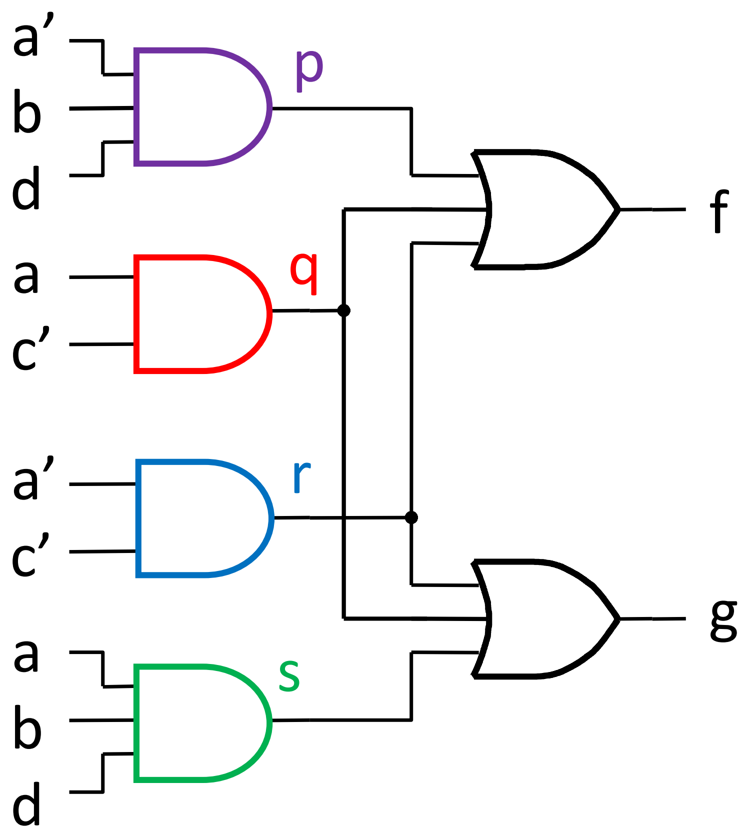
Verilog code for a multiplexer.



```
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 //.

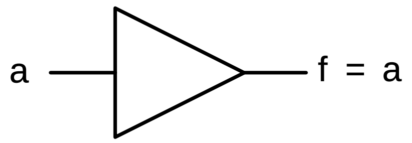
Verilog code for a multiplexer.



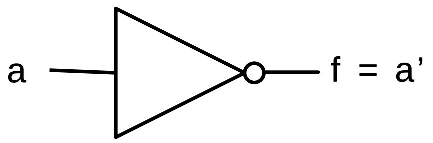
```
module MultiOutput(
      input a, b, c, d,
      output f, g );

   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.

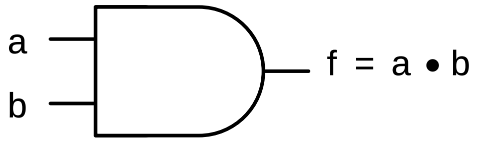
Basic gates in Verilog



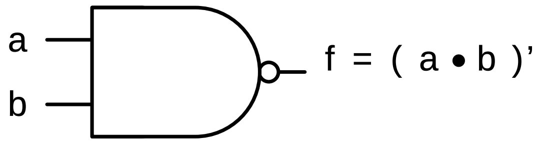
```
buf( f, a );
```



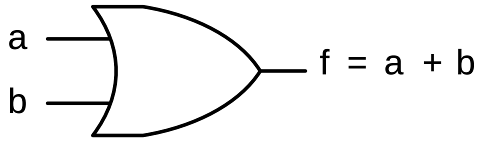
```
not( f, a );
```



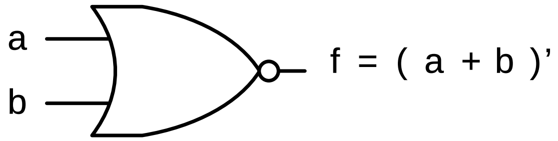
```
and( f, a, b, … );
```



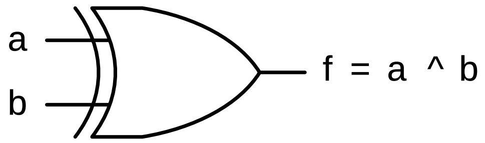
```
nand( f, a, b, … );
```



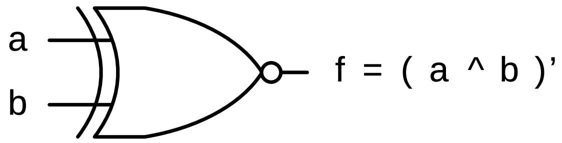
```
or( f, a, b, … );
```



```
nor( f, a, b, … );
```

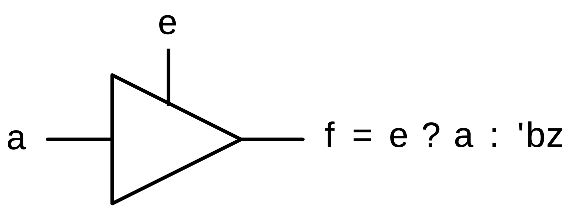


```
xor( f, a, b, … );
```

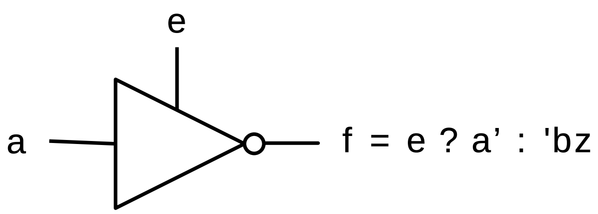


```
xnor( f, a, b, … );
```

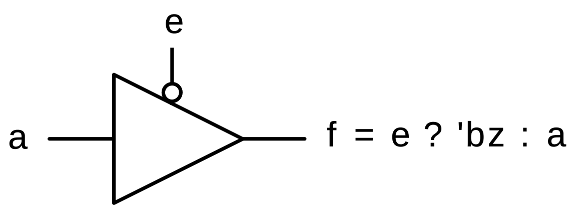
Tri-state drivers in Verilog



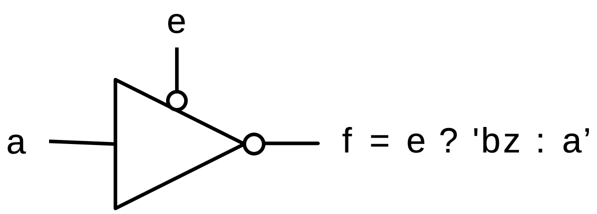
bufif1(f, a, e);



notif1(f, a, e);

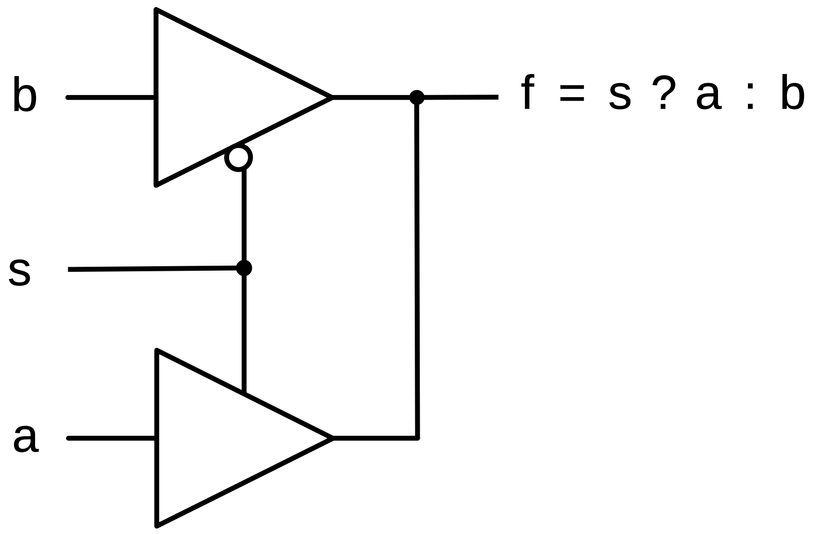


bufif0(f, a, e);

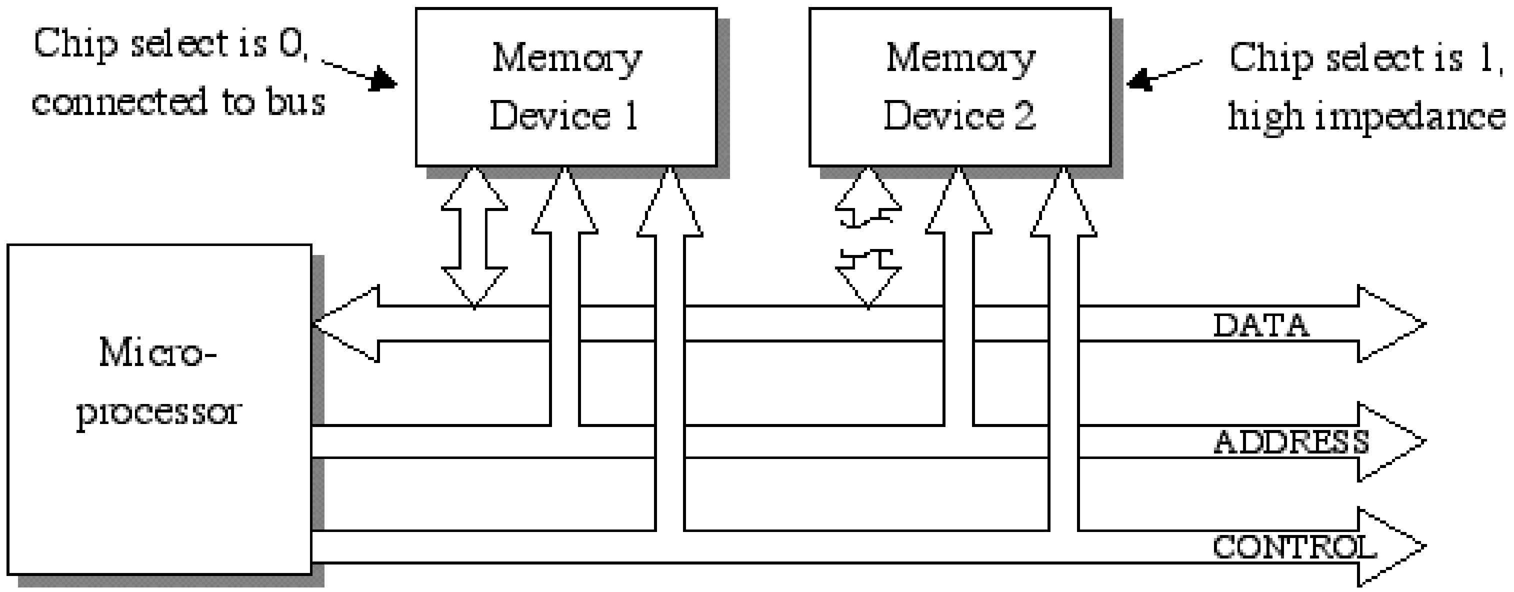


notif0(f, a, e);

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



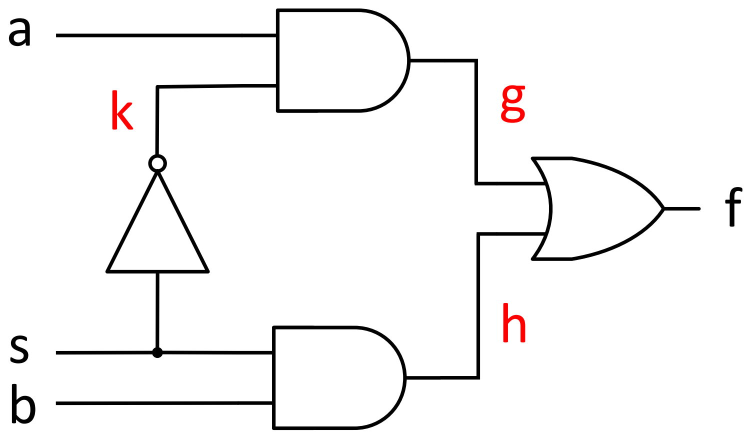
A multiplexer built from 2 tri-state drivers.



A more realistic application as a device or chip select.

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

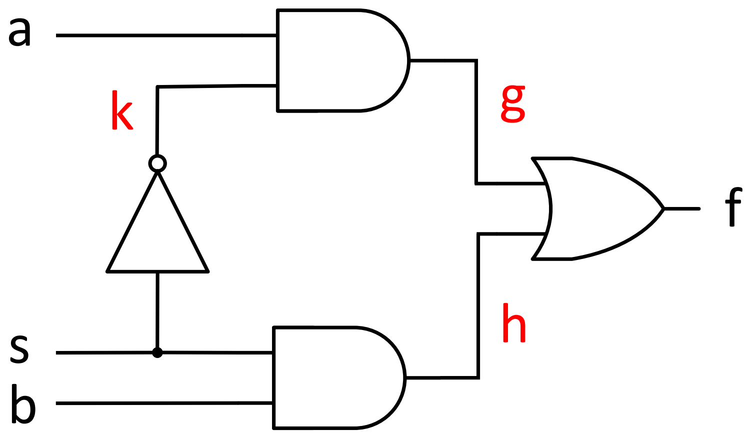
2. Boolean expressions



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

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

endmodule
```

f will continuously reflect the value of the RHS.

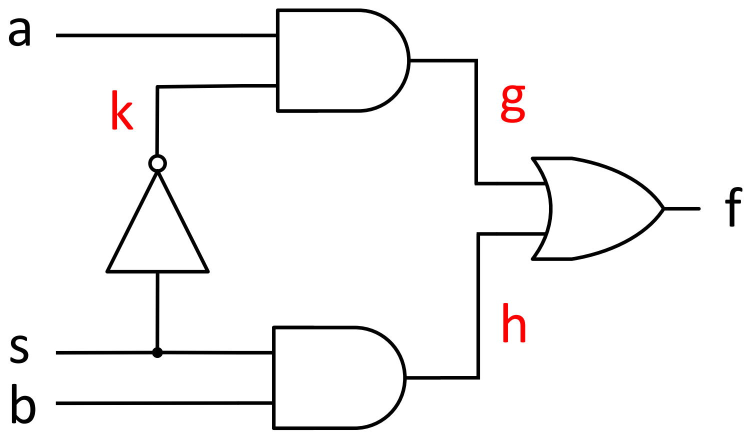
Continuous assignment



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

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

endmodule
```

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

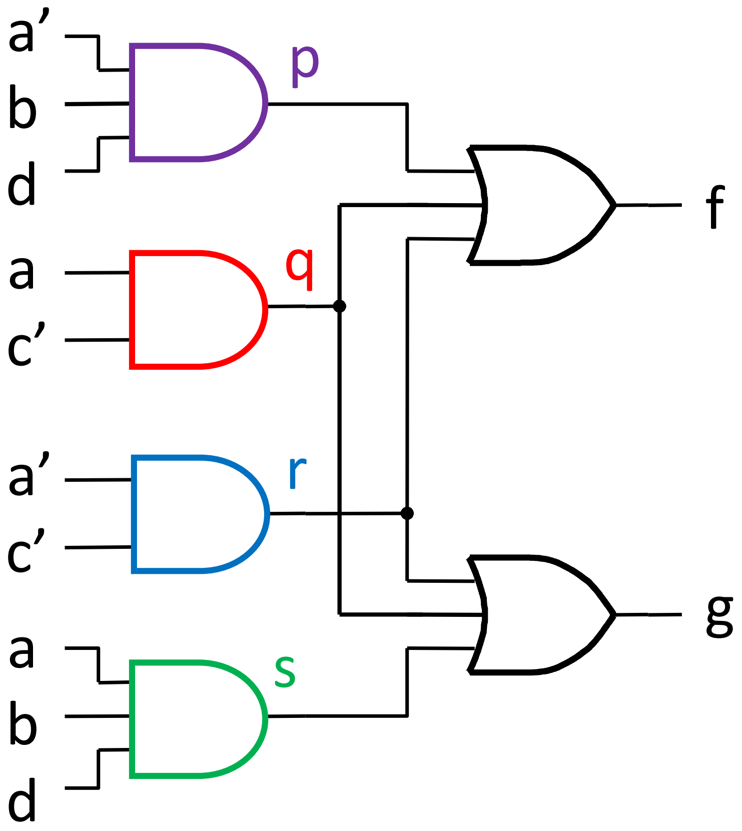
Continuous assignment



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

    assign f = s ? b : a;

endmodule
```

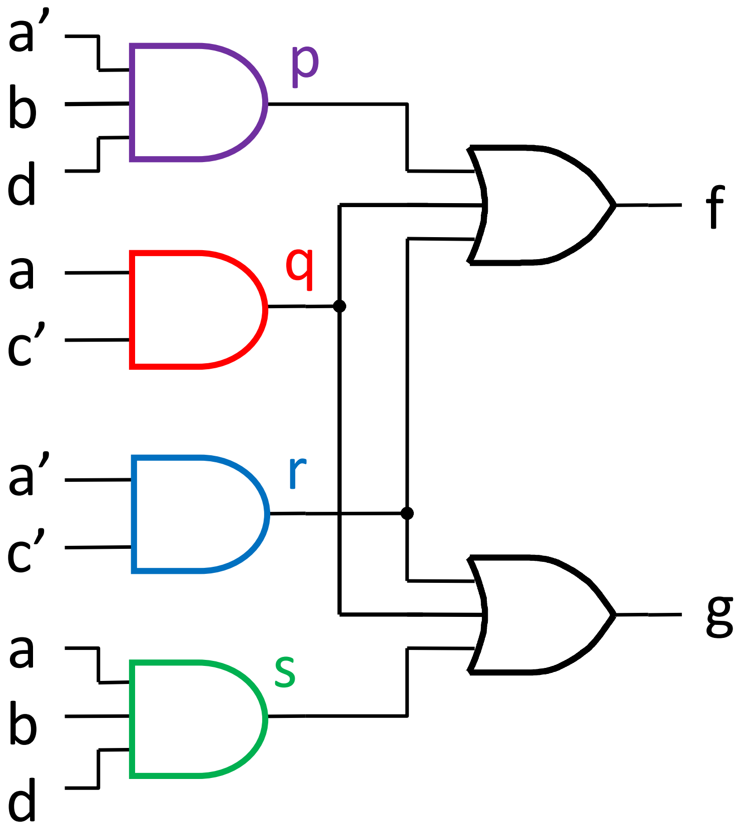
If s is true, f = b,
otherwise, f = a.

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

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

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.

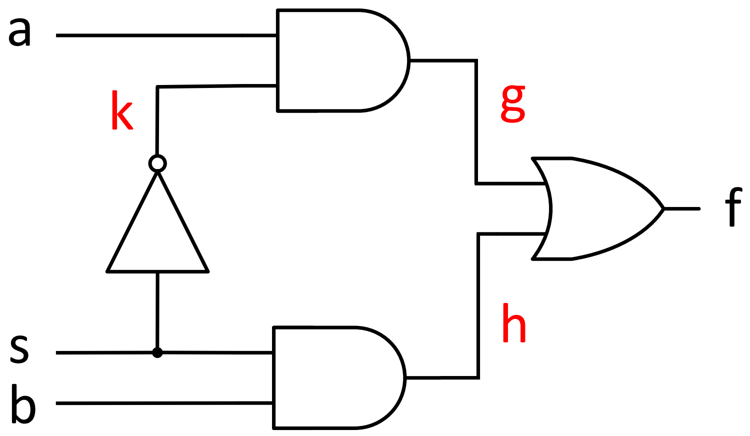
Operator	Meaning	Result
Arithmetic		
- a	Arithmetic complement	-5
+ a	Unary plus.	5
a * b	Multiplication.	15
a / b	Integer division.	1
a % b	Modulo (remainder) division.	2
Shifting		
a >> b	Shift a right b bits	0000
a << b	Shift a left b bits	1000

Basic Verilog operators

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

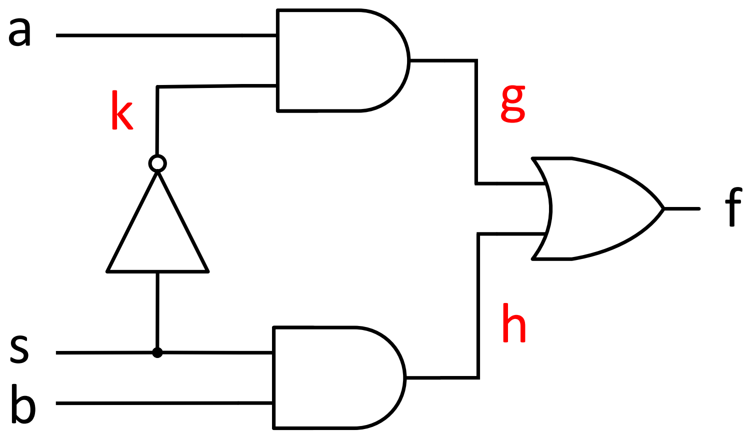
Operator	Meaning	Result
Concatenation and replication		
{ a, b }	Concate a and b.	010011
{ b { a } }	Replicate and concatenate b copies of a. b must be a constant.	010010010
Conditional (Trinary)		
a ? b : c	If a is non-zero, result = b. Otherwise, result = c.	011

3. Behavioral description



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

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

endmodule
```

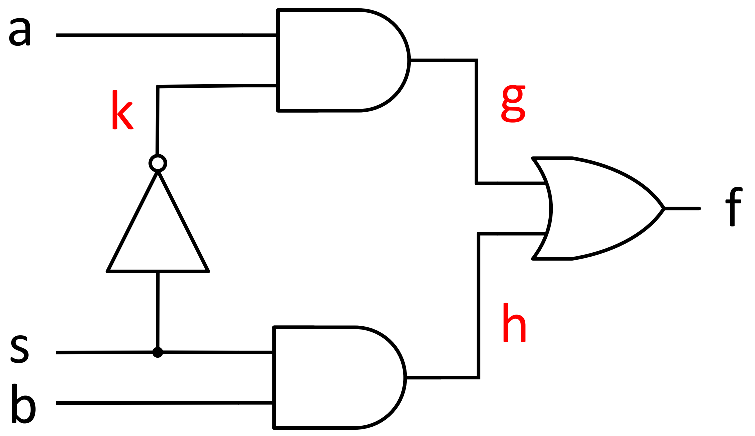
Behavioral specification of a multiplexer.



```
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*.

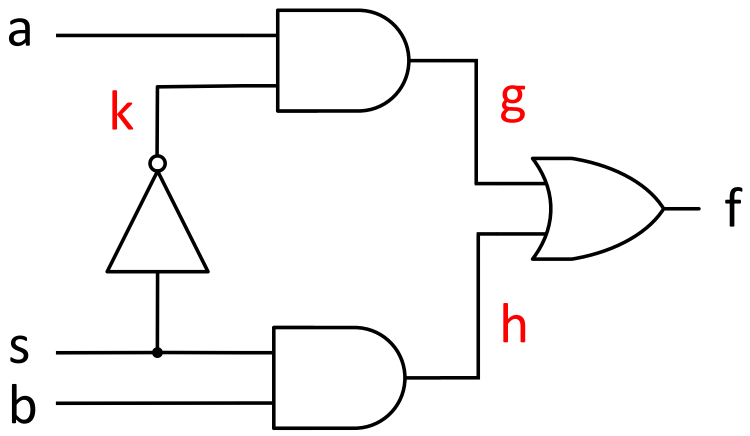
Behavioral specification of a multiplexer.



```
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.)

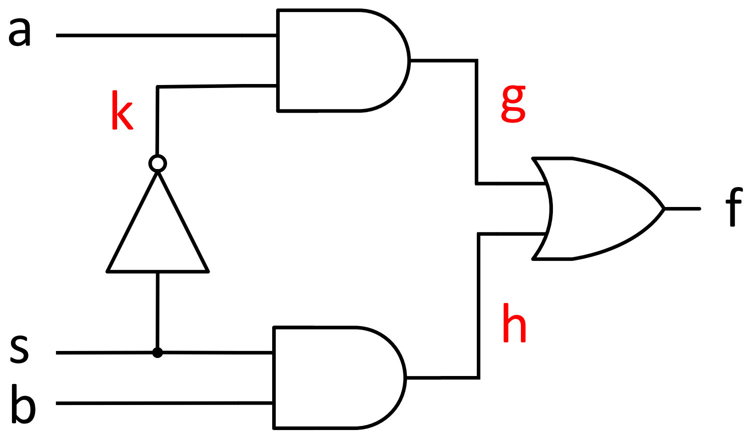
Behavioral specification of a multiplexer.



```
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:

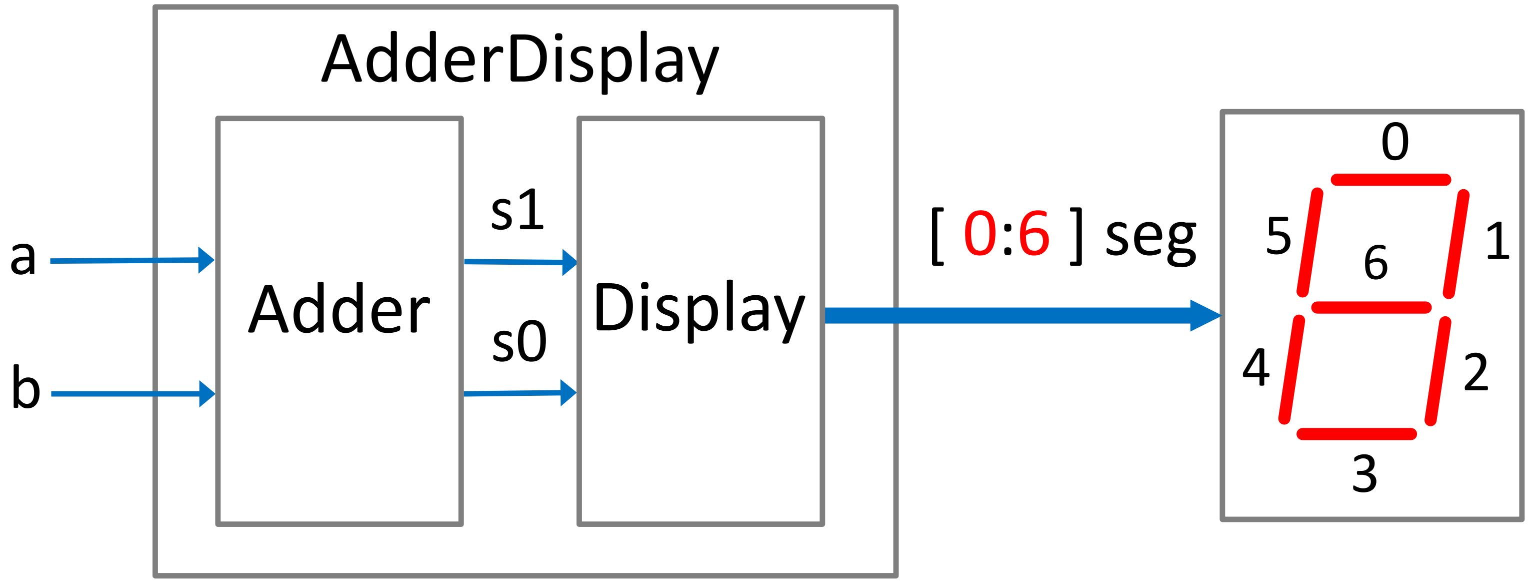
1. Before the name to indicate the size in a definition.
2. After the name when indexing.

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 ];
```

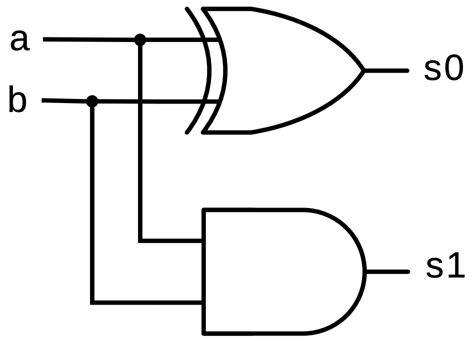
You can combine modules to create new modules.



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
```

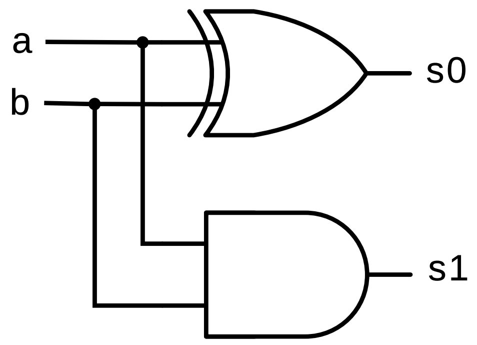


A one-bit adder in Verilog.

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

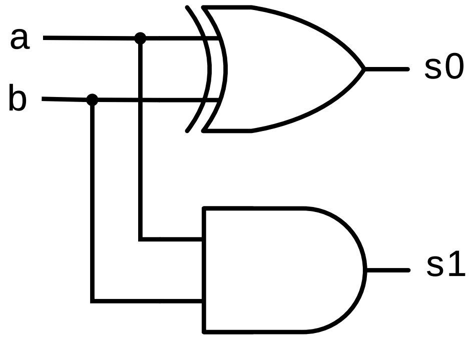
```
module Adder2( input a, b,
        output s1, s0 );

    assign { s1, s0 } = a + b;

endmodule
```



A one-bit adder in Verilog.

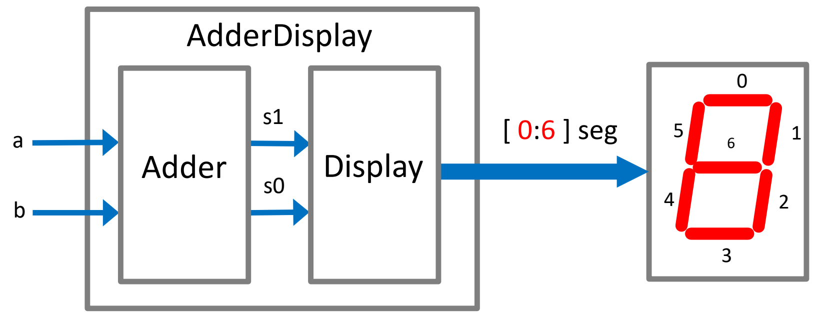
a	0	0	1	1
+b	+0	+1	+0	+1
s1 s0	0 0	0 1	0 1	1 0

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
```

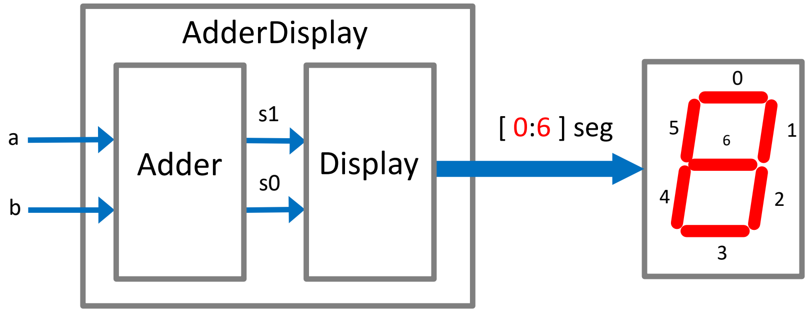
A one-bit adder in Verilog.



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

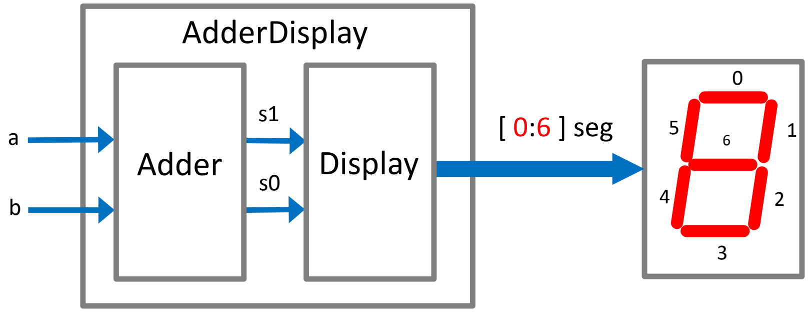
```
module Display( input s1, s0,
      output [ 0:6 ] seg );

   // 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
```

A very simple display driver in Verilog.



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

   // Use concatenation.

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

endmodule
```



```
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.

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

   // What does this do for each combination
   // of C1 and C2?

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

endmodule
```

```
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
```

```
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
```

```
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

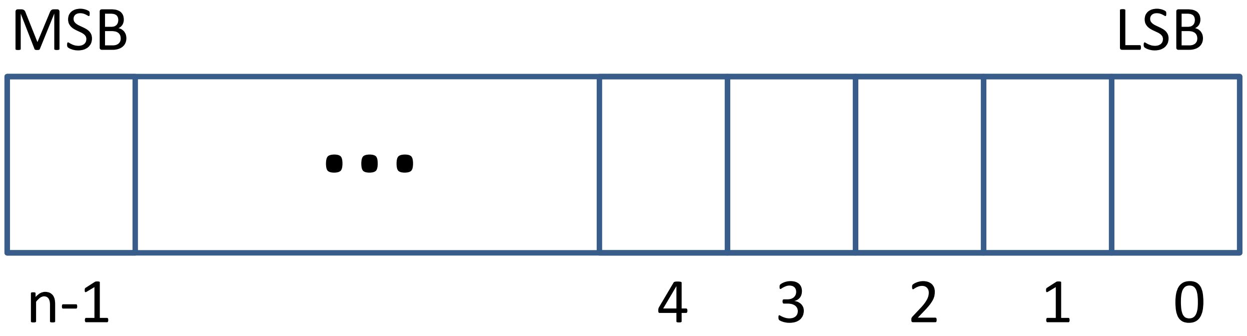
- All bits represent the magnitude of a positive integer

Signed numbers

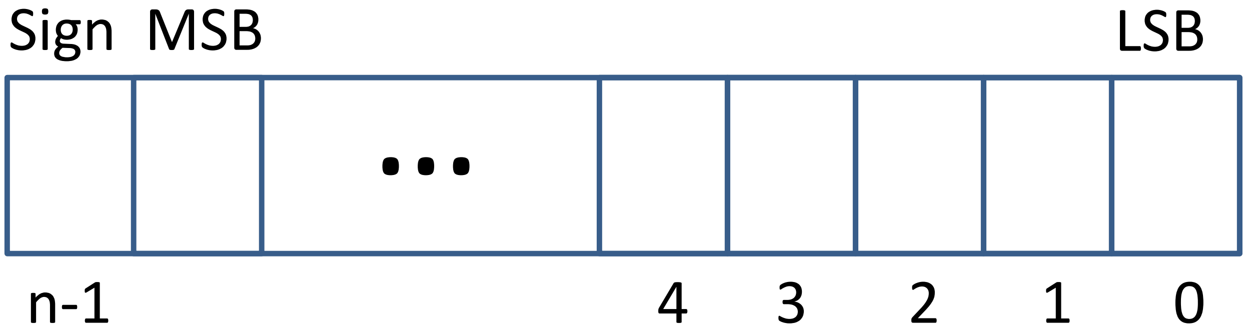
- Left-most bit represents the sign.

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



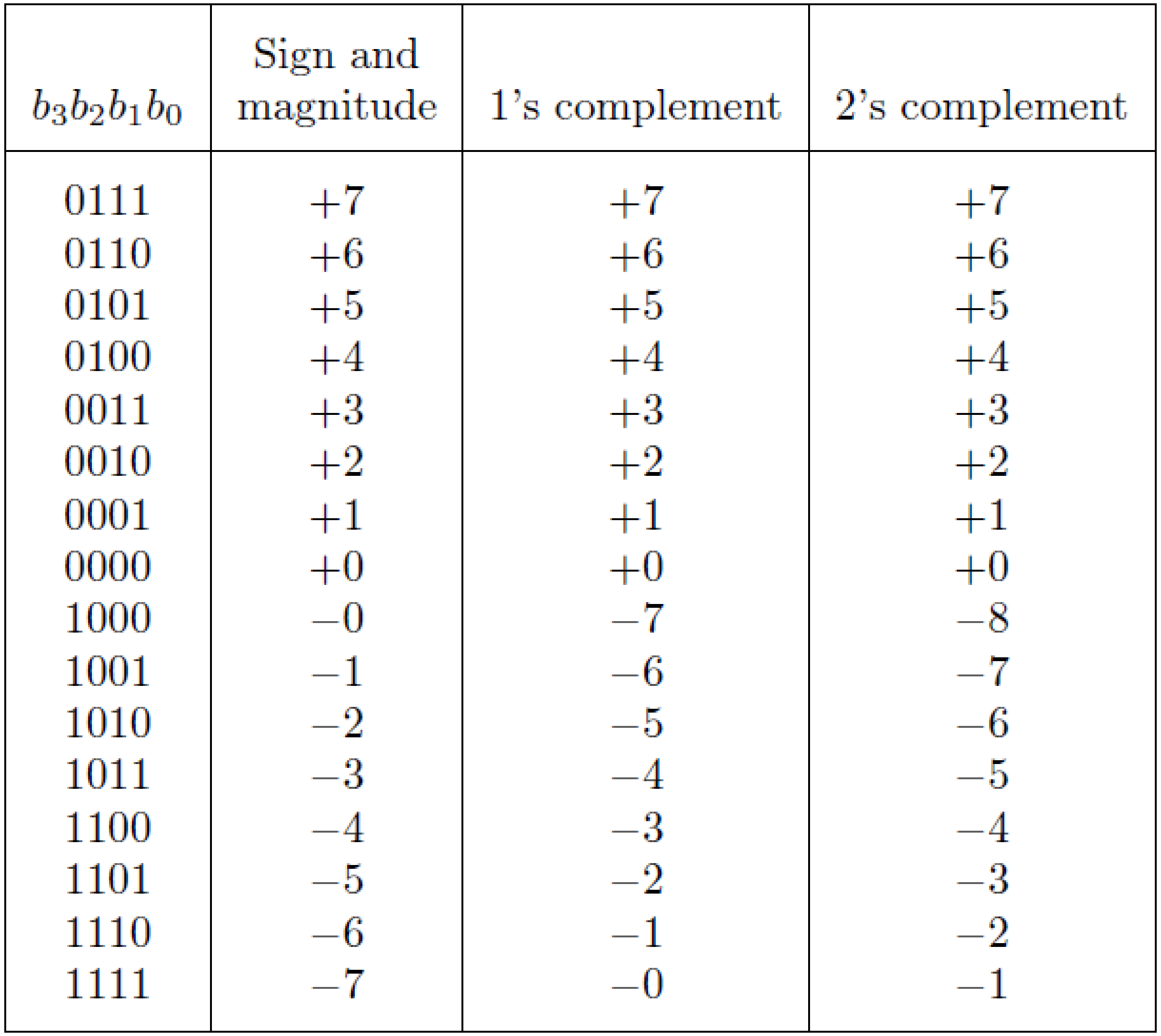
Signed binary

Negative numbers can be represented in following ways:

Sign + magnitude

1’s complement

2’s complement

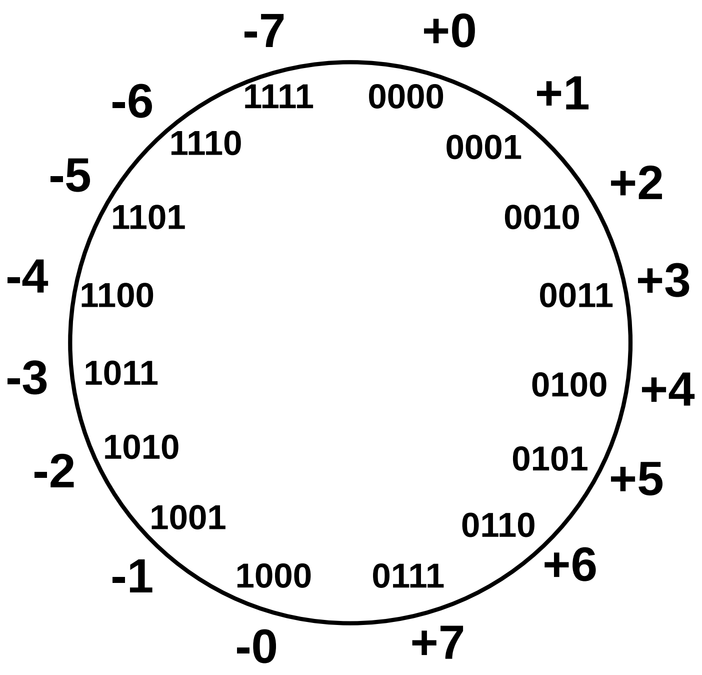


$b_3b_2b_1b_0$	Sign and magnitude	1's complement	2's complement
0111	+7	+7	+7
0110	+6	+6	+6
0101	+5	+5	+5
0100	+4	+4	+4
0011	+3	+3	+3
0010	+2	+2	+2
0001	+1	+1	+1
0000	+0	+0	+0
1000	−0	−7	−8
1001	−1	−6	−7
1010	−2	−5	−6
1011	−3	−4	−5
1100	−4	−3	−4
1101	−5	−2	−3
1110	−6	−1	−2
1111	−7	−0	−1

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)
```

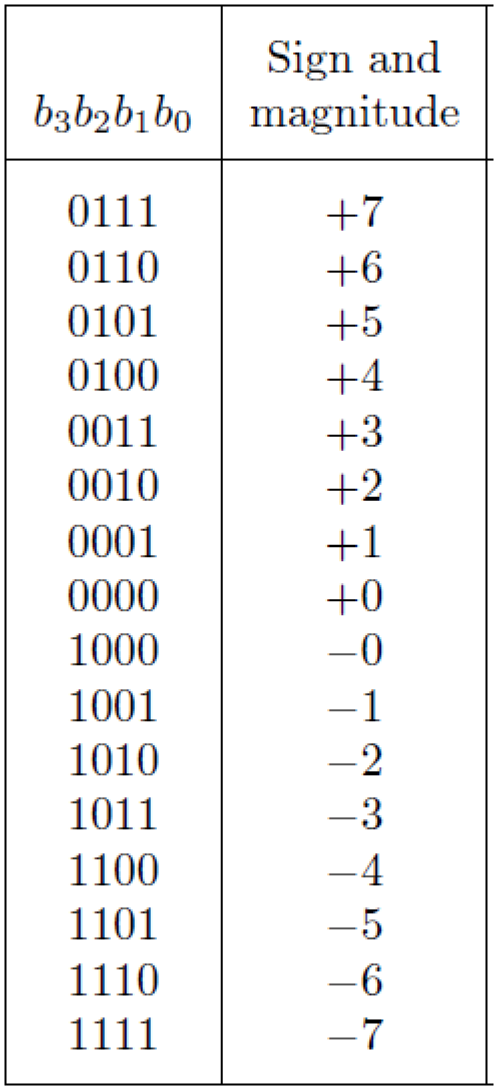
Adding with sign + magnitude

$b_3b_2b_1b_0$	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

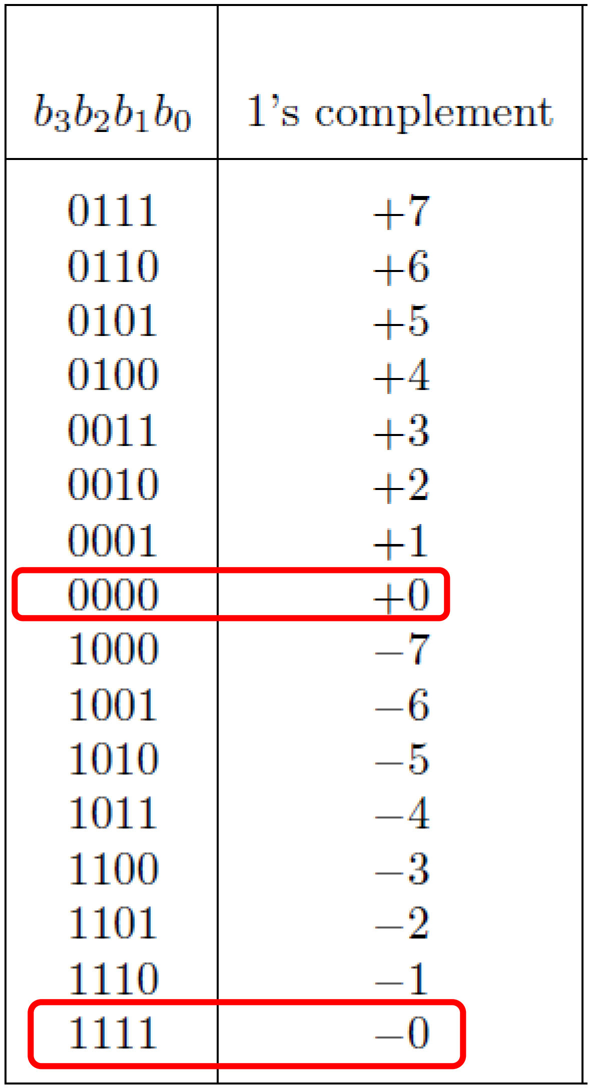
But if the signs are different, it doesn't work.

$$\begin{array}{rl} 1010 & (-2) \\ +\ \underline{0011} & (+3) \\ 1101 & (-5)\ \text{Wrong} \end{array}$$

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

$$\begin{array}{rl} 011 & (+3) \\ -\ \underline{\quad 010} & (-2 \text{ w/o the sign}) \\ 001 & \end{array}$$

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).

1’s complement

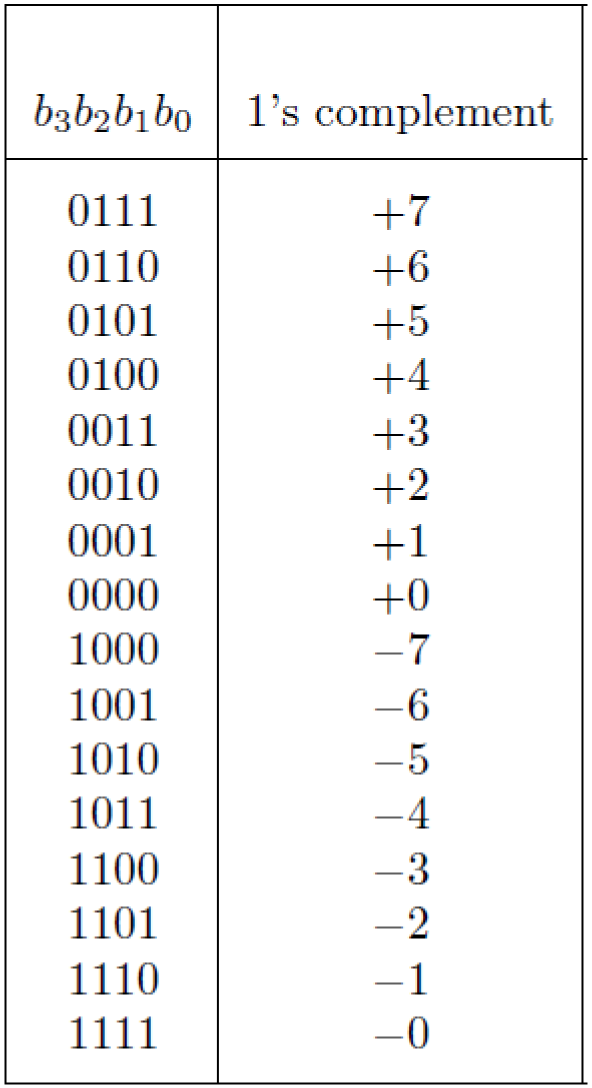
Let K be the negative equivalent of an n-bit positive number P.

Then, in 1’s complement representation K is obtained by subtracting P from $2^n - 1$, namely

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

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

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.

$$\begin{array}{rl} 0010 & (+2) \\ +\ 0011 & (+3) \\ \hline 0101 & (+5) \end{array}$$

$$\begin{array}{rl} 1101 & (-2) \\ +\ 1100 & (-3) \\ \hline 1001 & (-6)\ \text{Wrong} \end{array}$$

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

Correct by adding the overflow.

```
                            1101   (-2)
   1101   (-2)            + 1100   (-3)
 + 0011   (+3)             11001
 1 0000                   +    1
 +    1                     0001   (-6)
   0001   (+1)
```

$$\begin{array}{r} (+5) \\ +\,(+2) \\ \hline (+7) \end{array} \qquad \begin{array}{r} 0\,1\,0\,1 \\ +\;0\,0\,1\,0 \\ \hline 0\,1\,1\,1 \end{array} \qquad\qquad \begin{array}{r} (-5) \\ +\,(+2) \\ \hline (-3) \end{array} \qquad \begin{array}{r} 1\,0\,1\,0 \\ +\;0\,0\,1\,0 \\ \hline 1\,1\,0\,0 \end{array}$$

Two values of 0:

+0 = 0000

-0 = 1111

$$\begin{array}{r} (+5) \\ +\,(-2) \\ \hline (+3) \end{array} \qquad \begin{array}{r} 0\,1\,0\,1 \\ +\;1\,1\,0\,1 \\ \hline 1\,0\,0\,1\,0 \\ \color{red}{\llcorner\!\lrcorner \xrightarrow{\quad} 1} \\ \hline \color{red}{0\,0\,1\,1} \end{array} \qquad\qquad \begin{array}{r} (-5) \\ +\,(-2) \\ \hline (-7) \end{array} \qquad \begin{array}{r} 1\,0\,1\,0 \\ +\;1\,1\,0\,1 \\ \hline 1\,0\,1\,1\,1 \\ \color{red}{\llcorner\!\lrcorner \xrightarrow{\quad} 1} \\ \hline \color{red}{1\,0\,0\,0} \end{array}$$

Overflow means you crossed over 2 zeros.

Figure 3.8. Examples of 1's complement addition.