- Problem: we have $\neg$, $\wedge$, and $\vee$, so given the specification

x_{n-1}	$\cdots$	x_1	x_0	r
0	$\cdots$	0	0	1
0	$\cdots$	0	1	0
0	$\cdots$	1	0	1
0	$\cdots$	1	1	0
$\vdots$		$\vdots$	$\vdots$	$\vdots$
1	$\cdots$	1	1	0

  for some Boolean function

$$r = f(x_0, x_1, \ldots, x_{n-1}),$$

  design a Boolean expression e which can compute it.

- Solution: ?

COMS10015 lecture: week #3

- Agenda: **combinatorial logic** design, where, crucially,
  - the output is a function of the input only,
  - computation is viewed as being continuous,

  via coverage of

  1. special-purpose design patterns,
  2. special-purpose building blocks, and
  3. general-purpose derivation.

- Pattern #1: **decomposition**.
  - Any n-input, m-output Boolean function

$$f : \mathbb{B}^n \rightarrow \mathbb{B}^m$$

    can be rewritten as m *separate* n-input, 1-output Boolean functions, say

$$\begin{array}{lcl} f_0 & : & \mathbb{B}^n \rightarrow \mathbb{B} \\ f_1 & : & \mathbb{B}^n \rightarrow \mathbb{B} \\ & \vdots & \\ f_{m-1} & : & \mathbb{B}^n \rightarrow \mathbb{B} \end{array}$$

  - As such, we have

$$f(x) \equiv f_0(x) \parallel f_1(x) \parallel \ldots \parallel f_{m-1}(x).$$

- Pattern #2: **sharing**.
  - Imagine, for example, that we are given a 2-input, 1-bit AND gate.
  - If, within some larger circuit, we compute

$$r = x \wedge y$$

and then, somewhere else,

$$r' = x \wedge y$$

then we can replace the two AND gates with one: clearly

$$r = r',$$

so we can share one definition between two usage points.

- Pattern #3: **independent replication**.
  - Imagine, for example, that we are given a 2-input, 1-bit AND gate.
  - A 2-input, m-bit AND gate is simply replication of 2-input, 1-bit AND gates, i.e.,

    $$r = x \wedge y$$

    is computed via

    $$r_i = x_i \wedge y_i$$

    for $0 \leq i < m$,
  - for $n = 4$, as an example, this means

    $$\begin{aligned} r_0 &= x_0 \wedge y_0 \\ r_1 &= x_1 \wedge y_1 \\ r_2 &= x_2 \wedge y_2 \\ r_3 &= x_3 \wedge y_3 \end{aligned}$$

- Pattern #4: **dependent replication**.
  - Imagine, for example, that we are given a 2-input, 1-bit AND gate.
  - An n-input, 1-bit AND gate is simply replication of 2-input, 1-bit AND gates, i.e.,

$$r = \bigwedge_{i=0}^{n-1} x_i$$

    is computed via

$$r = x_0 \wedge (x_1 \wedge \cdots (x_{n-1})),$$

  - for $n = 4$, as an example, this means

$$\begin{aligned} r &= x_0 \wedge (x_1 \wedge x_2 \wedge (x_3)) \\ &= x_0 \wedge x_1 \wedge x_2 \wedge x_3 \\ &= (x_0 \wedge x_1) \wedge (x_2 \wedge x_3) \end{aligned}$$

- Concept: the following building blocks can support most forms of choice
  1. a **multiplexer**
     - has m inputs,
     - has 1 output,
     - uses a $(\lceil \log_2(m) \rceil)$-bit control signal input to choose which input is connected to the output,

     while
  2. a **demultiplexer**
     - has 1 input,
     - has m outputs,
     - uses a $(\lceil \log_2(m) \rceil)$-bit control signal input to choose which output is connected to the input,

  noting that
  - the input(s) and output(s) are n-bit, but clearly must match up,
  - the connection made is continuous, since both components are combinatorial.

- Concept: by analogy,
  1. the C `switch` statement

Listing

```
1 switch( c ) {
2   case 0 : r = w; break;
3   case 1 : r = x; break;
4   case 2 : r = y; break;
5   case 3 : r = z; break;
6 }
```

  acts similarly to a 4-input multiplexer,
  2. the C `switch` statement

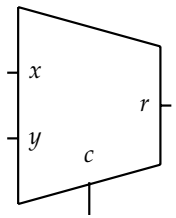
Listing

```
1 switch( c ) {
2   case 0 : r0 = x; break;
3   case 1 : r1 = x; break;
4   case 2 : r2 = x; break;
5   case 3 : r3 = x; break;
6 }
```

  acts similarly to a 4-output demultiplexer.

Definition

The behaviour of a **2-input, 1-bit multiplexer** component



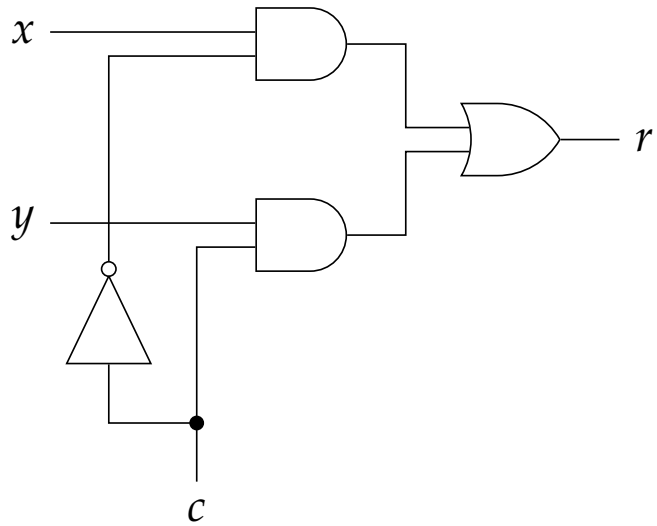
is described by the truth table

c	x	y	r
0	0	**?**	0
0	1	**?**	1
1	**?**	0	0
1	**?**	1	1

which can be used to derive the following implementation:

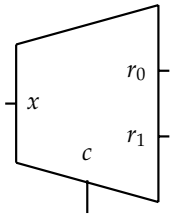
$$r \quad = \quad (\neg c \wedge x) \vee (c \wedge y)$$

Circuit



Definition

The behaviour of a **2-output, 1-bit demultiplexer** component



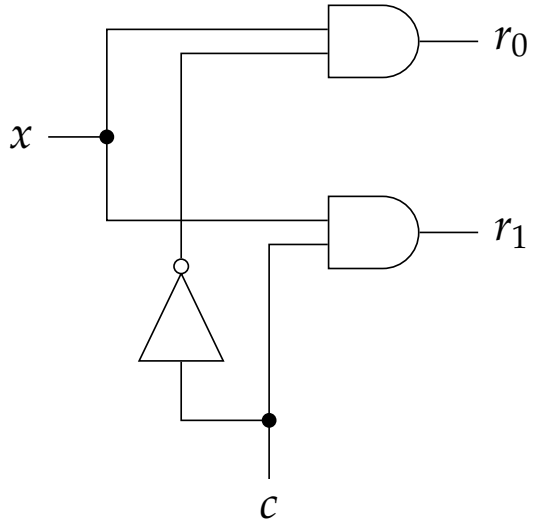
is described by the truth table

c	x	r_1	r_0
0	0	**?**	0
0	1	**?**	1
1	0	0	**?**
1	1	1	**?**

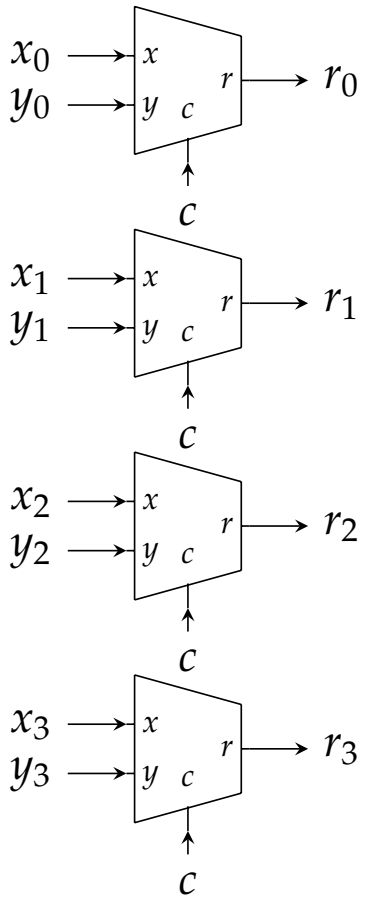
which can be used to derive the following implementation:

$$\begin{aligned} r_0 &= \neg c \wedge x \\ r_1 &= c \wedge x \end{aligned}$$

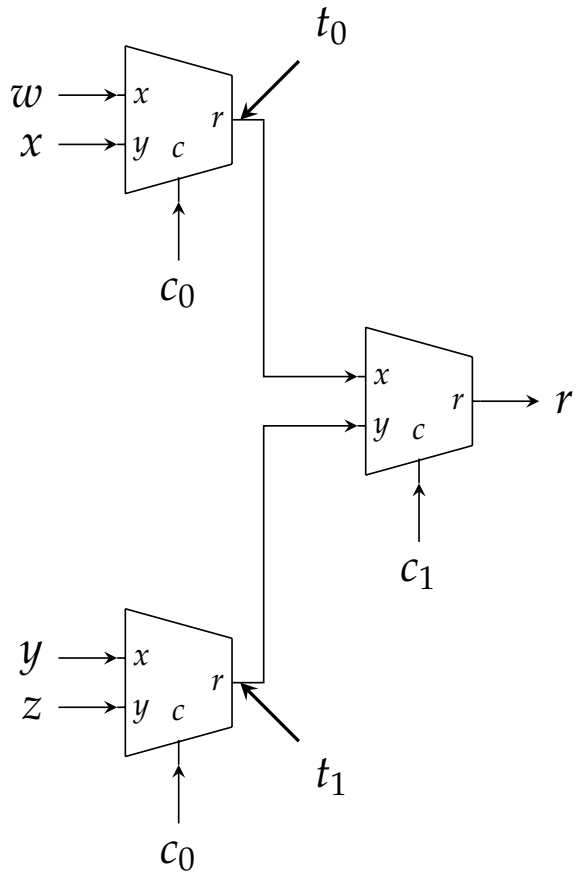
Circuit



Circuit (2-input, 4-bit multiplexer via independent replication)



Circuit (4-input, 1-bit multiplexer via dependent replication)



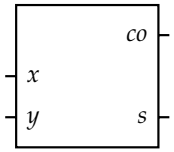
- Concept: the following building blocks can support most forms of arithmetic
  1. a **half-adder**
     - has 2 inputs: x and y,
     - computes the 2-bit result $x + y$,
     - has 2 outputs: a sum s, and a carry-out co (which are the LSB and MSB of result),

     while
  2. a **full-adder**
     - has 3 inputs: x and y plus a carry-in ci,
     - computes the 2-bit result $x + y + ci$,
     - has 2 outputs: a sum s, and a carry-out co (which are the LSB and MSB of result),

  where all inputs and outputs are 1-bit.

Definition

The behaviour of a **half-adder** component



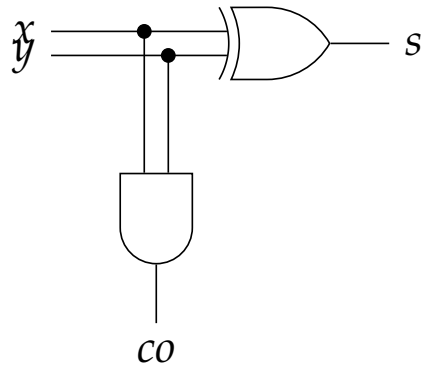
is described by the truth table

x	y	co	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

which can be used to derive the following implementation:

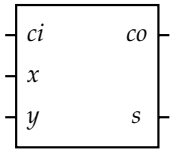
$$\begin{aligned} co &= x \wedge y \\ s &= x \oplus y \end{aligned}$$

Circuit



Definition

The behaviour of a **full-adder** component



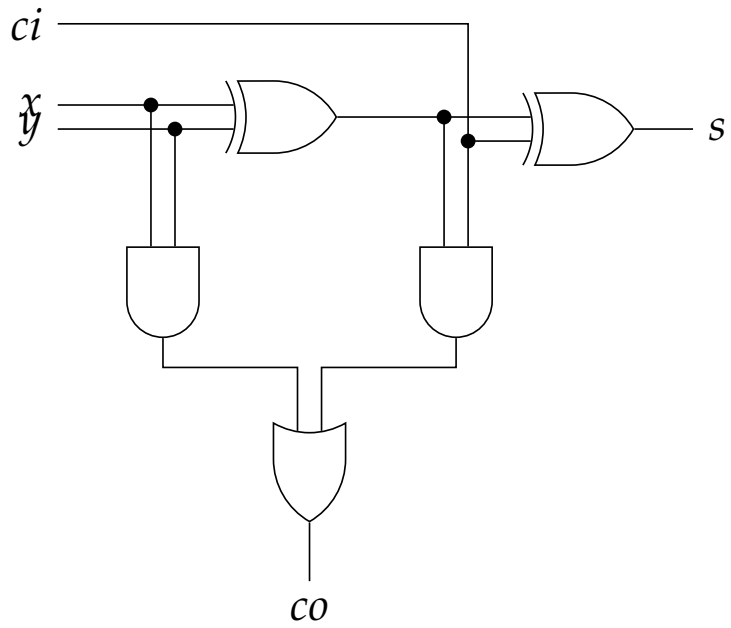
is described by the truth table

ci	x	y	co	s
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

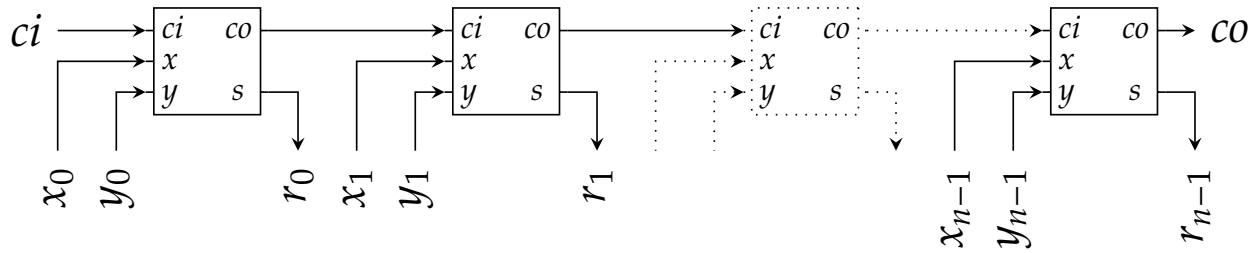
which can be used to derive the following implementation:

$$\begin{aligned} co &= (x \wedge y) \vee (x \wedge ci) \vee (y \wedge ci) \\ &= (x \wedge y) \vee ((x \oplus y) \wedge ci) \\ s &= x \oplus y \oplus ci \end{aligned}$$

Circuit



Circuit (n-bit addition)



- Concept: the following building blocks can support most forms of comparison
  1. an **equality comparator**
     - has 2 inputs x and y,
     - computes the 1 output as

$$r = \begin{cases} 1 & \text{if } x = y \\ 0 & \text{otherwise} \end{cases}$$

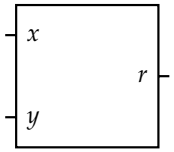
  while
  2. a **less-than comparator**
     - has 2 inputs x and y,
     - computes the 1 output as

$$r = \begin{cases} 1 & \text{if } x < y \\ 0 & \text{otherwise} \end{cases}$$

  where all inputs and outputs are 1-bit.

Definition

The behaviour of an **equality comparator** component



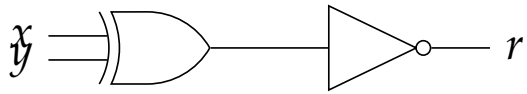
is described by the truth table

x	y	r
0	0	1
0	1	0
1	0	0
1	1	1

which can be used to derive the following implementation:

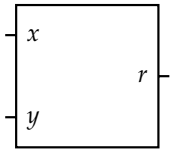
$$r = \neg(x \oplus y)$$

Circuit



Definition

The behaviour of a **less-than comparator** component



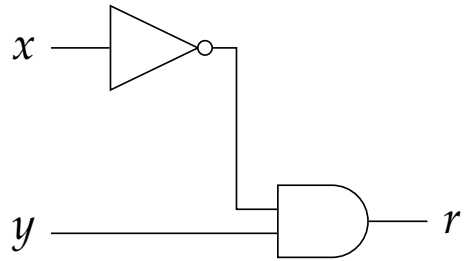
is described by the truth table

x	y	r
0	0	0
0	1	1
1	0	0
1	1	0

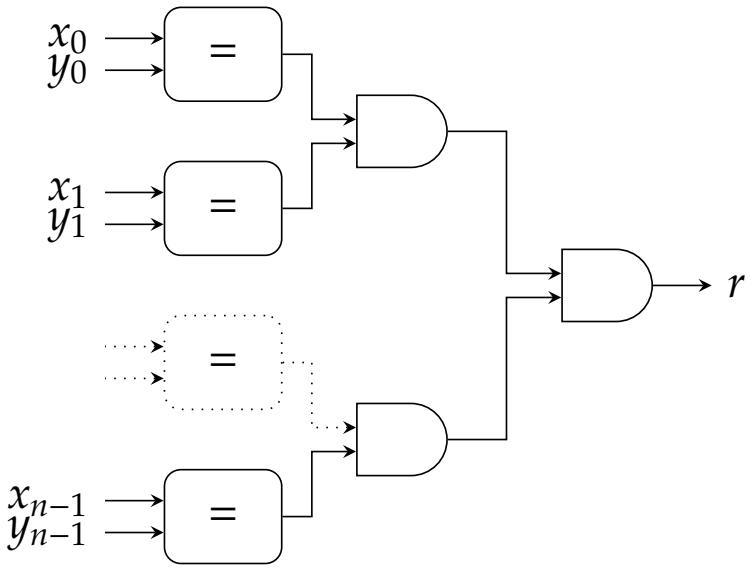
which can be used to derive the following implementation:

$$r = \neg x \wedge y$$

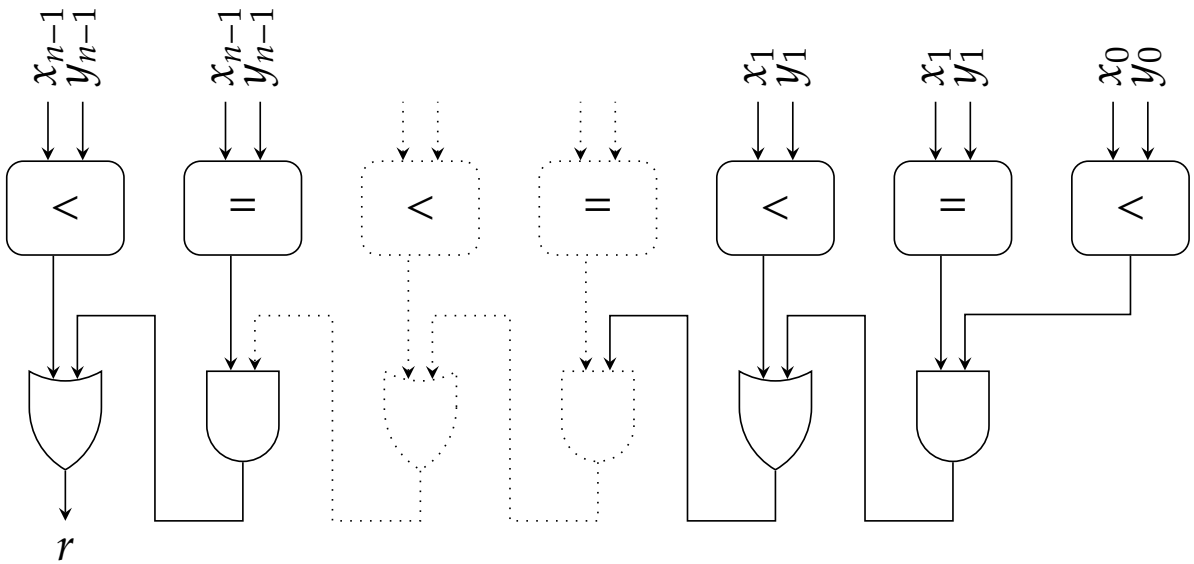
Circuit



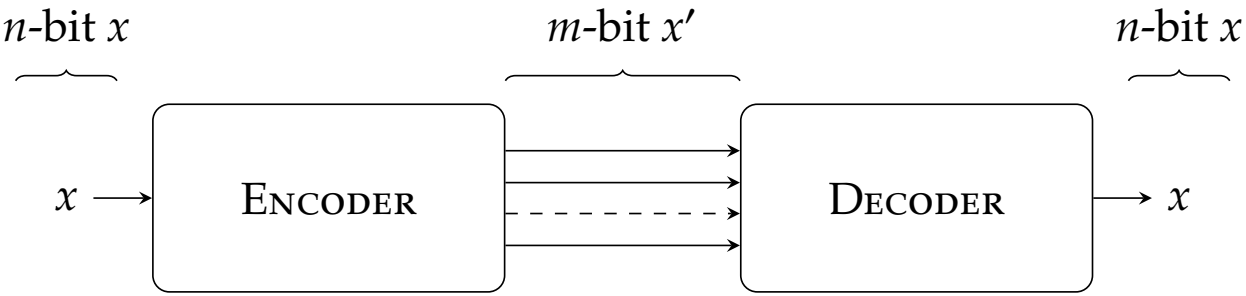
Circuit (n-bit equality comparison)



Circuit (n-bit less-than comparison)



- Concept: informally, **encoders** and **decoders** can be viewed as *translators*, i.e.,



or, more formally,

1. an n-to-m encoder translates an n-bit input into some m-bit code word, and
2. an m-to-n decoder translates an m-bit code word back into the same n-bit output

where if only one output (resp. input) is allowed to be 1 at a time, we call it a **one-of-many** encoder (resp. decoder).

- A *general* building block is impossible since it depends on the scheme for encoding/decoding: consider an example such that
  1. to encode, take n inputs, say x_i for $0 \leq i < n$, and produce a unsigned integer x' that determines which $x_i = 1$,
  2. to decode, take x' and set the correct $x_i = 1$

  where for all $j \neq i$, $x'_j = 0$.

Definition (example encoder)

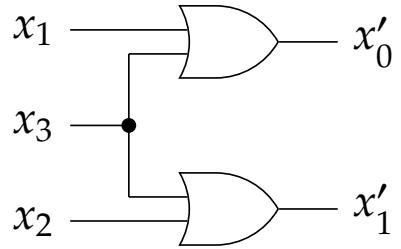
The example encoder is described by the truth table

x_3	x_2	x_1	x_0	x'_1	x'_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

which can be used to derive the following implementation:

$$\begin{aligned} x'_0 &= x_1 \vee x_3 \\ x'_1 &= x_2 \vee x_3 \end{aligned}$$

Circuit (example encoder)



Definition (example decoder)

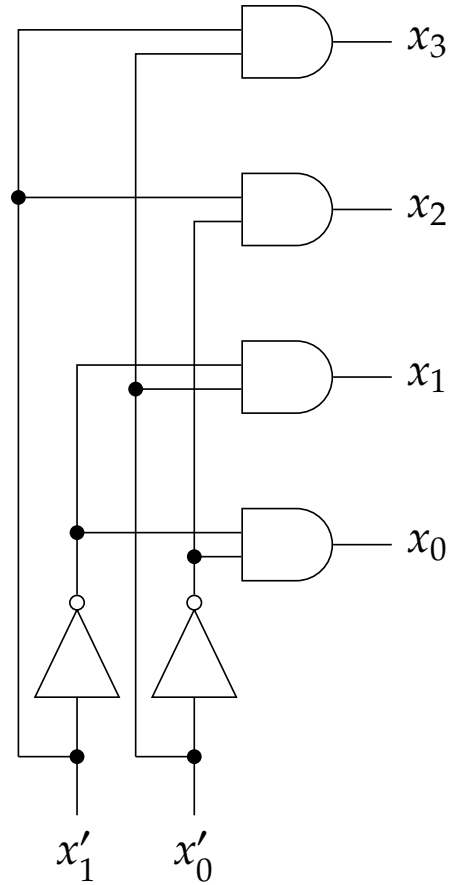
The example decoder is described by the truth table

x'_1	x'_0	x_3	x_2	x_1	x_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

which can be used to derive the following implementation:

$$\begin{aligned} x_0 &= \neg x'_0 \wedge \neg x'_1 \\ x_1 &= x'_0 \wedge \neg x'_1 \\ x_2 &= \neg x'_0 \wedge x'_1 \\ x_3 &= x'_0 \wedge x'_1 \end{aligned}$$

Circuit (example decoder)



Part 3: general-purpose derivation (1)

Method #1

Algorithm

Input: A truth table for some Boolean function f, with n inputs and 1 output
Output: A Boolean expression e that implements f

First let $\mathcal{I}_j$ denote the j-th input for $0 \leq j < n$ and $\mathcal{O}$ denote the single output:

1. Find a set T such that $i \in T$ iff. $\mathcal{O} = 1$ in the i-th row of the truth table.
2. For each $i \in T$, form a term t_i by AND'ing together all the variables while following two rules:
   - 2.1 if $\mathcal{I}_j = 1$ in the i-th row, then we use
     $$\mathcal{I}_j$$
     as is, but
   - 2.2 if $\mathcal{I}_j = 0$ in the i-th row, then we use
     $$\neg \mathcal{I}_j.$$
3. An expression implementing the function is then formed by OR'ing together all the terms, i.e.,
   $$e = \bigvee_{i \in T} t_i,$$
   which is in SoP form.

Example

Consider the example of deriving an expression for XOR, i.e.,

$$r = f(x, y) = x \oplus y,$$

a function described by the following truth table:

f		
x	y	r
0	0	0
0	1	1
1	0	1
1	1	0

Example

Consider the example of deriving an expression for XOR, i.e.,

$$r = f(x, y) = x \oplus y,$$

a function described by the following truth table:

f		
x	y	r
0	0	0
0	1	1
1	0	1
1	1	0

$\rightsquigarrow$ $i = 1$
$\rightsquigarrow$ $i = 2$

Following the algorithm produces:

1. Looking at the truth table, it is clear there are
   - $n = 2$ inputs that we denote $\mathcal{I}_0 = x$ and $\mathcal{I}_1 = y$, and
   - one output that we denote $\mathcal{O} = r$.

   Clearly $T = \{1, 2\}$ since $\mathcal{O} = 1$ in rows 1 and 2, while $\mathcal{O} = 0$ in rows 0 and 3.

Example

Consider the example of deriving an expression for XOR, i.e.,

$$r = f(x, y) = x \oplus y,$$

a function described by the following truth table:

f		
x	y	r
0	0	0
0	1	1
1	0	1
1	1	0

$\rightsquigarrow \quad t_1 = \neg x \wedge y$

$\rightsquigarrow \quad t_2 = x \wedge \neg y$

Following the algorithm produces:

2. Each term t_i for $i \in T = \{1, 2\}$ is formed as follows:
   - For $i = 1$, we find
     - $\mathcal{I}_0 = x = 0$ and so we use $\neg x$,
     - $\mathcal{I}_1 = y = 1$ and so we use y

     and hence form the term $t_1 = \neg x \wedge y$.
   - For $i = 2$, we find
     - $\mathcal{I}_0 = x = 1$ and so we use x,
     - $\mathcal{I}_1 = y = 0$ and so we use $\neg y$

     and hence form the term $t_2 = x \wedge \neg y$.

Example

Consider the example of deriving an expression for XOR, i.e.,

$$r = f(x, y) = x \oplus y,$$

a function described by the following truth table:

f		
x	y	r
0	0	0
0	1	1
1	0	1
1	1	0

$\rightsquigarrow \quad t_1 = \neg x \wedge y$

$\rightsquigarrow \quad t_2 = x \wedge \neg y$

Following the algorithm produces:

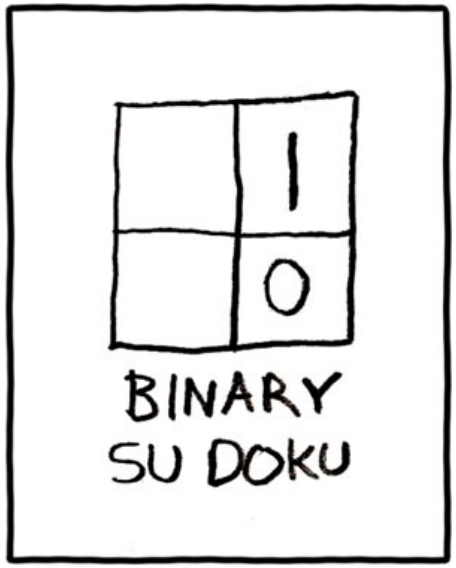
3. The expression implementing the function is therefore

$$\begin{aligned} e &= \bigvee_{i \in T} t_i \\ &= \bigvee_{i \in \{1,2\}} t_i \\ &= (\neg x \wedge y) \vee (x \wedge \neg y) \end{aligned}$$

which is in SoP form.

Part 3: general-purpose derivation (3)

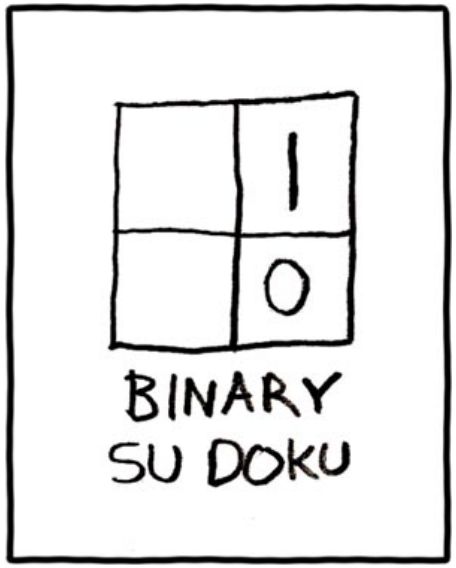
Method #2: Karnaugh map



https://xkcd.com/74

Part 3: general-purpose derivation (3)

Method #2: Karnaugh map



- Idea:

$$
\begin{aligned}
(x \wedge y) \vee (x \wedge \neg y) &\equiv x \wedge (y \vee \neg y) && \text{(distribution)} \\
&\equiv x \wedge 1 && \text{(inverse)} \\
&\equiv x && \text{(identity)}
\end{aligned}
$$

https://xkcd.com/74

Algorithm

Input: A truth table for some Boolean function f, with n inputs and 1 output
Output: A Boolean expression e that implements f

1. Draw a rectangular $(p \times q)$-element grid, st.
   1.1 $p \equiv q \equiv 0 \pmod 2$, and
   1.2 $p \cdot q = 2^n$

   and each row and column represents one input combination; order rows and columns according to a **Gray code**.
2. Fill the grid elements with the output corresponding to inputs for that row and column.
3. Cover rectangular groups of adjacent 1 elements which are of total size 2^m for some m; groups can "wrap around" edges of the grid and overlap.
4. Translate each group into one term of an SoP form Boolean expression e where
   4.1 *bigger* groups, and
   4.2 *less* groups

   mean a simpler expression.

Part 3: general-purpose derivation (5)

Method #2: Karnaugh map

Example

Natural sequence			**Gray code** sequence		
$\langle 0,0,0,0 \rangle$	$\mapsto$	$0_{(10)}$	$\langle 0,0,0,0 \rangle$	$\mapsto$	$0_{(10)}$
$\langle 1,0,0,0 \rangle$	$\mapsto$	$1_{(10)}$	$\langle 1,0,0,0 \rangle$	$\mapsto$	$1_{(10)}$
$\langle 0,1,0,0 \rangle$	$\mapsto$	$2_{(10)}$	$\langle 1,1,0,0 \rangle$	$\mapsto$	$3_{(10)}$
$\langle 1,1,0,0 \rangle$	$\mapsto$	$3_{(10)}$	$\langle 0,1,0,0 \rangle$	$\mapsto$	$2_{(10)}$
$\langle 0,0,1,0 \rangle$	$\mapsto$	$4_{(10)}$	$\langle 0,1,1,0 \rangle$	$\mapsto$	$6_{(10)}$
$\langle 1,0,1,0 \rangle$	$\mapsto$	$5_{(10)}$	$\langle 0,0,1,0 \rangle$	$\mapsto$	$4_{(10)}$
$\langle 0,1,1,0 \rangle$	$\mapsto$	$6_{(10)}$	$\langle 1,0,1,0 \rangle$	$\mapsto$	$5_{(10)}$
$\langle 1,1,1,0 \rangle$	$\mapsto$	$7_{(10)}$	$\langle 1,1,1,0 \rangle$	$\mapsto$	$7_{(10)}$
$\vdots$			$\vdots$		

Example

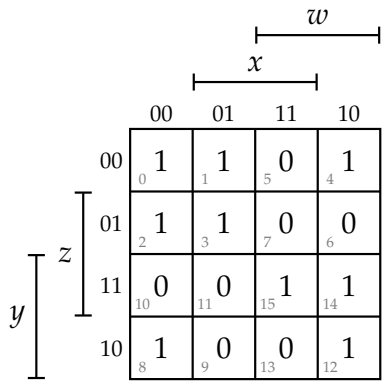
Consider an example 4-input, 1-output function:

w	x	y	z	r
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	1
0	1	1	0	0
0	1	1	1	0
1	0	0	0	1
1	0	0	1	0
1	0	1	0	1
1	0	1	1	1
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

Example

Consider an example 4-input, 1-output function:

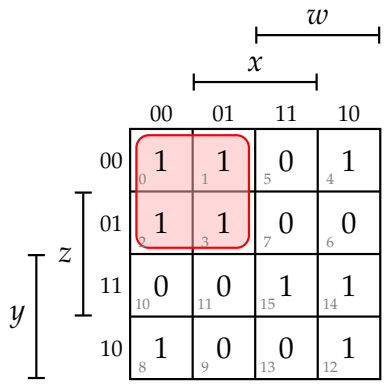
w	x	y	z	r
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	1
0	1	1	0	0
0	1	1	1	0
1	0	0	0	1
1	0	0	1	0
1	0	1	0	1
1	0	1	1	1
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1



Example

Consider an example 4-input, 1-output function:

w	x	y	z	r
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	1
0	1	1	0	0
0	1	1	1	0
1	0	0	0	1
1	0	0	1	0
1	0	1	0	1
1	0	1	1	1
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1



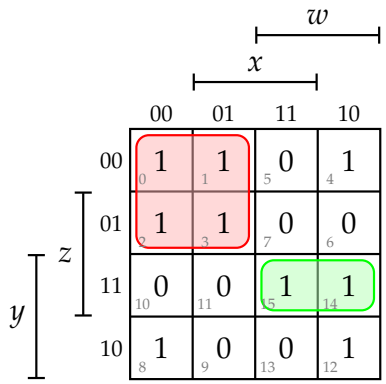
Each group translates into one term of the SoP form expression

$$r = (\quad \neg w \quad \wedge \quad \neg y \quad)$$

Example

Consider an example 4-input, 1-output function:

w	x	y	z	r
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	1
0	1	1	0	0
0	1	1	1	0
1	0	0	0	1
1	0	0	1	0
1	0	1	0	1
1	0	1	1	1
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1



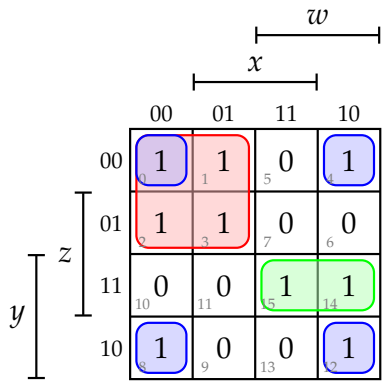
Each group translates into one term of the SoP form expression

$$r = (\ \neg w \ \wedge\ \neg y\ ) \vee (\ w \ \wedge\ y \ \wedge\ z\ )$$

Example

Consider an example 4-input, 1-output function:

w	x	y	z	r
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	1
0	1	1	0	0
0	1	1	1	0
1	0	0	0	1
1	0	0	1	0
1	0	1	0	1
1	0	1	1	1
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1



Each group translates into one term of the SoP form expression

$$r = (\neg w \wedge \neg y) \vee (w \wedge y \wedge z) \vee (\neg x \wedge \neg z)$$

Example

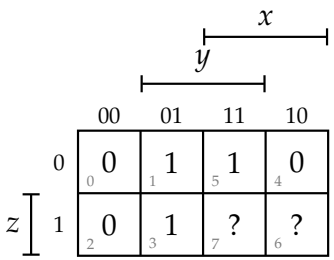
Consider an example 3-input, 1-output function:

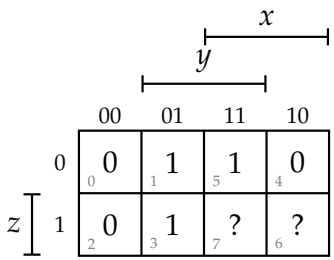
x	y	z	r
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	**?**
1	1	0	1
1	1	1	**?**

Example

Consider an example 3-input, 1-output function:

x	y	z	r
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	**?**
1	1	0	1
1	1	1	**?**

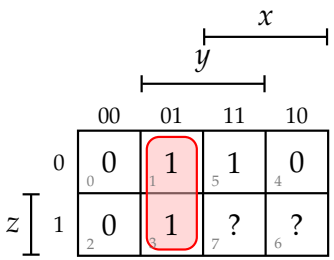


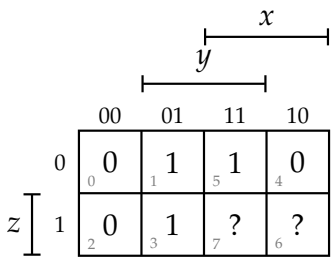


Example

Consider an example 3-input, 1-output function:

x	y	z	r
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	**?**
1	1	0	1
1	1	1	**?**





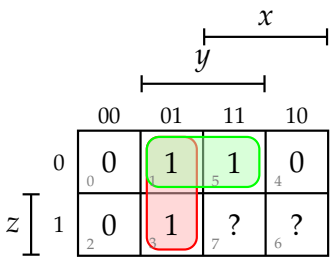
Each group translates into one term of the SoP form expressions

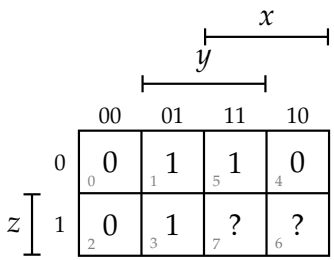
$$r = (\neg x \wedge y)$$

Example

Consider an example 3-input, 1-output function:

x	y	z	r
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	**?**
1	1	0	1
1	1	1	**?**





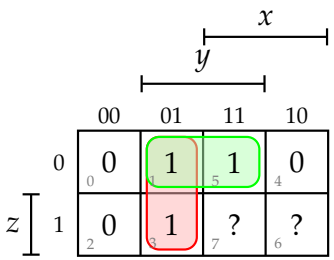
Each group translates into one term of the SoP form expressions

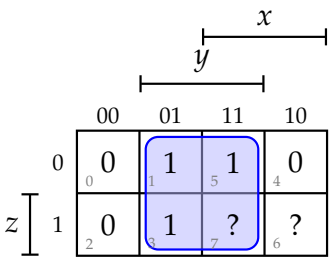
$$r = (\neg x \wedge y) \vee (y \wedge \neg z)$$

Example

Consider an example 3-input, 1-output function:

x	y	z	r
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	**?**
1	1	0	1
1	1	1	**?**





Each group translates into one term of the SoP form expressions

$$r = (\neg x \wedge y) \vee (y \wedge \neg z) \qquad r = y$$

where effective use of don't care states yields a clear improvement!

Example

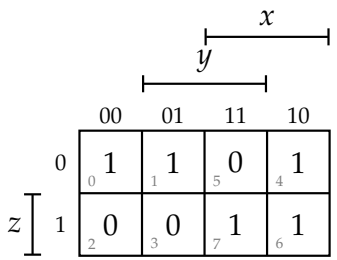
Consider an example 3-input, 1-output function:

x	y	z	r
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

Example

Consider an example 3-input, 1-output function:

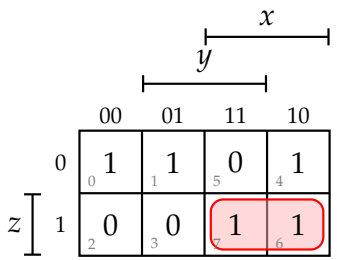
x	y	z	r
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1



Example

Consider an example 3-input, 1-output function:

x	y	z	r
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1



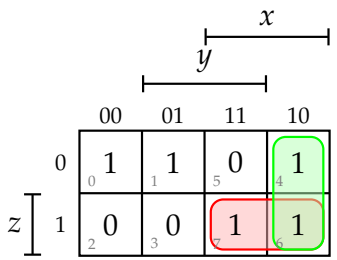
Each group translates into one term of the SoP form expression

$$r = (\quad x \quad \wedge \quad z \quad)$$

Example

Consider an example 3-input, 1-output function:

x	y	z	r
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1



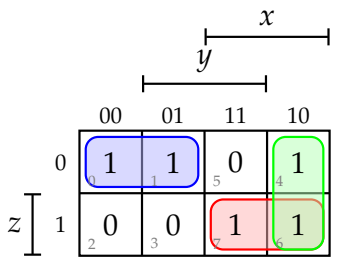
Each group translates into one term of the SoP form expression

$$r = (\ x\ \wedge\ z\ ) \vee (\ x\ \wedge\ \neg y\ )$$

Example

Consider an example 3-input, 1-output function:

x	y	z	r
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1



Each group translates into one term of the SoP form expression

$$r = (\ x \wedge z\ ) \vee (\ x \wedge \neg y\ ) \vee (\ \neg x \wedge \neg z\ )$$

Example

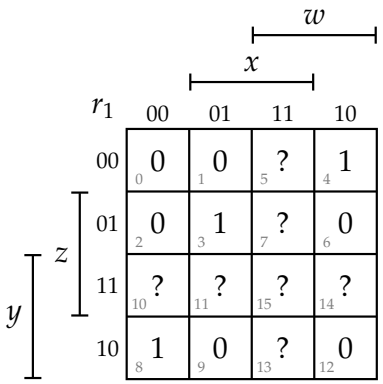
Consider an example 4-input, 2-output function:

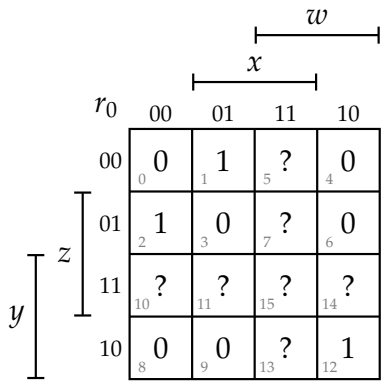
w	x	y	z	r_1	r_0
0	0	0	0	0	0
0	0	0	1	0	1
0	0	1	0	1	0
0	0	1	1	**?**	**?**
0	1	0	0	0	1
0	1	0	1	1	0
0	1	1	0	0	0
0	1	1	1	**?**	**?**
1	0	0	0	1	0
1	0	0	1	0	0
1	0	1	0	0	1
1	0	1	1	**?**	**?**
1	1	0	0	**?**	**?**
1	1	0	1	**?**	**?**
1	1	1	0	**?**	**?**
1	1	1	1	**?**	**?**

Example

Consider an example 4-input, 2-output function:

w	x	y	z	r_1	r_0
0	0	0	0	0	0
0	0	0	1	0	1
0	0	1	0	1	0
0	0	1	1	?	?
0	1	0	0	0	1
0	1	0	1	1	0
0	1	1	0	0	0
0	1	1	1	?	?
1	0	0	0	1	0
1	0	0	1	0	0
1	0	1	0	0	1
1	0	1	1	?	?
1	1	0	0	?	?
1	1	0	1	?	?
1	1	1	0	?	?
1	1	1	1	?	?

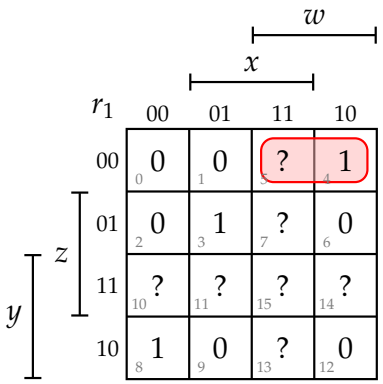


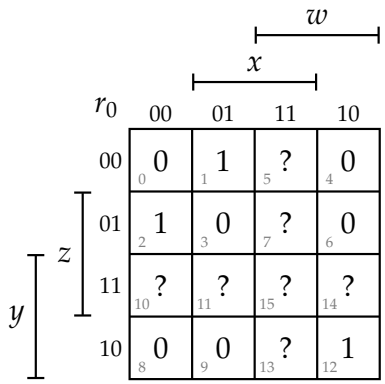


Example

Consider an example 4-input, 2-output function:

w	x	y	z	r_1	r_0
0	0	0	0	0	0
0	0	0	1	0	1
0	0	1	0	1	0
0	0	1	1	?	?
0	1	0	0	0	1
0	1	0	1	1	0
0	1	1	0	0	0
0	1	1	1	?	?
1	0	0	0	1	0
1	0	0	1	0	0
1	0	1	0	0	1
1	0	1	1	?	?
1	1	0	0	?	?
1	1	0	1	?	?
1	1	1	0	?	?
1	1	1	1	?	?





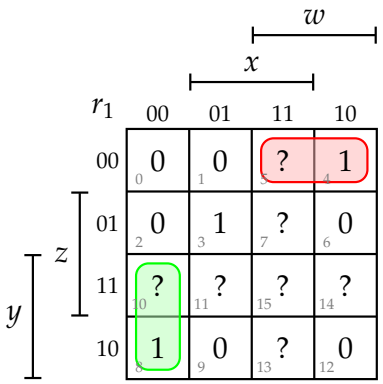
Each group translates into one term of the SoP form expressions

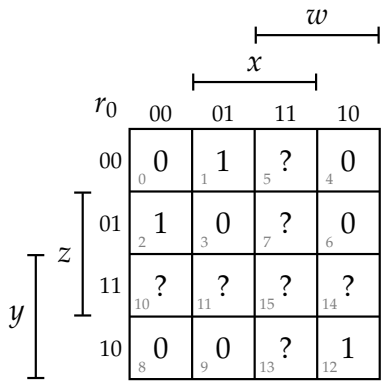
$$r_1 = (\quad w \quad \wedge \quad \neg y \quad \wedge \quad \neg z \quad)$$

Example

Consider an example 4-input, 2-output function:

w	x	y	z	r_1	r_0
0	0	0	0	0	0
0	0	0	1	0	1
0	0	1	0	1	0
0	0	1	1	?	?
0	1	0	0	0	1
0	1	0	1	1	0
0	1	1	0	0	0
0	1	1	1	?	?
1	0	0	0	1	0
1	0	0	1	0	0
1	0	1	0	0	1
1	0	1	1	?	?
1	1	0	0	?	?
1	1	0	1	?	?
1	1	1	0	?	?
1	1	1	1	?	?





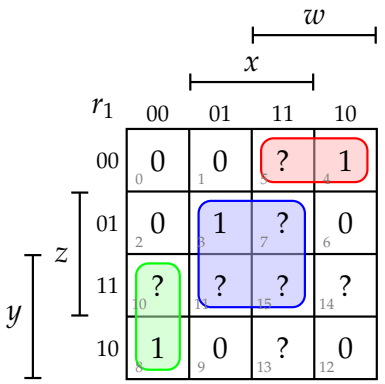
Each group translates into one term of the SoP form expressions

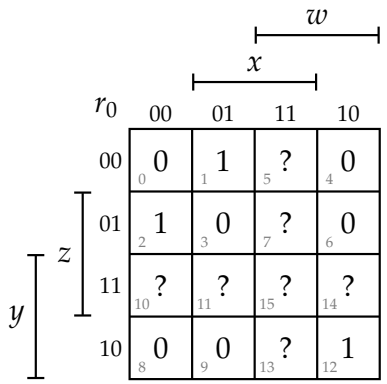
$$r_1 = (\ w \wedge \neg y \wedge \neg z\ ) \vee (\ y \wedge \neg w \wedge \neg x\ )$$

Example

Consider an example 4-input, 2-output function:

w	x	y	z	r_1	r_0
0	0	0	0	0	0
0	0	0	1	0	1
0	0	1	0	1	0
0	0	1	1	?	?
0	1	0	0	0	1
0	1	0	1	1	0
0	1	1	0	0	0
0	1	1	1	?	?
1	0	0	0	1	0
1	0	0	1	0	0
1	0	1	0	0	1
1	0	1	1	?	?
1	1	0	0	?	?
1	1	0	1	?	?
1	1	1	0	?	?
1	1	1	1	?	?





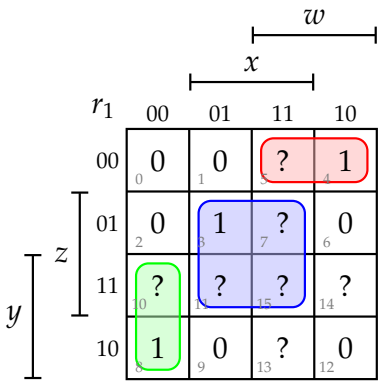
Each group translates into one term of the SoP form expressions

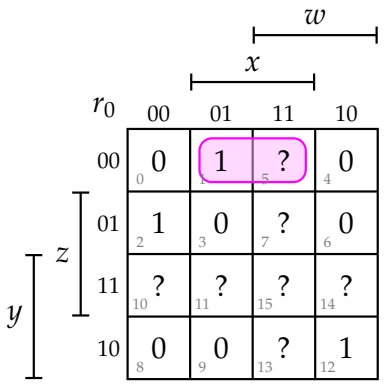
$$r_1 = (\ w \wedge \neg y \wedge \neg z\ ) \vee (\ y \wedge \neg w \wedge \neg x\ ) \vee (\ x \wedge z\ )$$

Example

Consider an example 4-input, 2-output function:

w	x	y	z	r_1	r_0
0	0	0	0	0	0
0	0	0	1	0	1
0	0	1	0	1	0
0	0	1	1	?	?
0	1	0	0	0	1
0	1	0	1	1	0
0	1	1	0	0	0
0	1	1	1	?	?
1	0	0	0	1	0
1	0	0	1	0	0
1	0	1	0	0	1
1	0	1	1	?	?
1	1	0	0	?	?
1	1	0	1	?	?
1	1	1	0	?	?
1	1	1	1	?	?





Each group translates into one term of the SoP form expressions

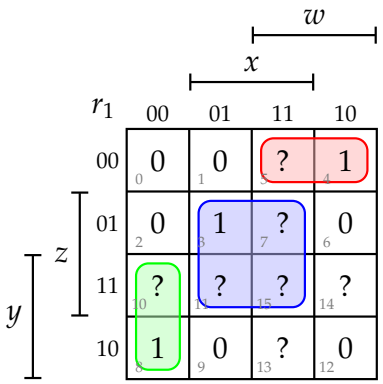
$$r_1 = (\ w \wedge \neg y \wedge \neg z\ ) \vee (\ y \wedge \neg w \wedge \neg x\ ) \vee (\ x \wedge z\ )$$

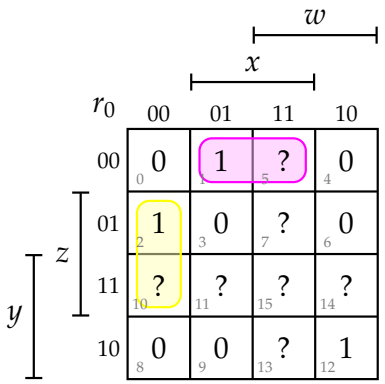
$$r_0 = (\ x \wedge \neg y \wedge \neg z\ )$$

Example

Consider an example 4-input, 2-output function:

w	x	y	z	r_1	r_0
0	0	0	0	0	0
0	0	0	1	0	1
0	0	1	0	1	0
0	0	1	1	?	?
0	1	0	0	0	1
0	1	0	1	1	0
0	1	1	0	0	0
0	1	1	1	?	?
1	0	0	0	1	0
1	0	0	1	0	0
1	0	1	0	0	1
1	0	1	1	?	?
1	1	0	0	?	?
1	1	0	1	?	?
1	1	1	0	?	?
1	1	1	1	?	?





Each group translates into one term of the SoP form expressions

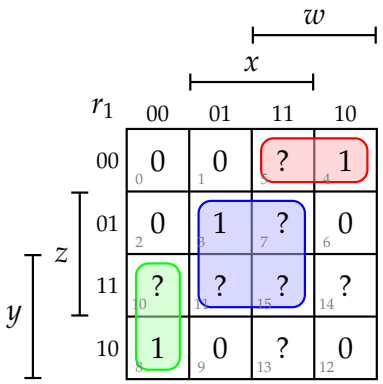
$$r_1 = (\ w \wedge \neg y \wedge \neg z\ ) \vee (\ y \wedge \neg w \wedge \neg x\ ) \vee (\ x \wedge z\ )$$

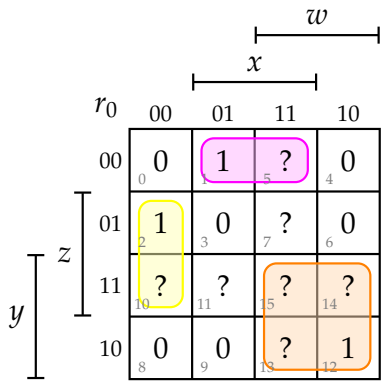
$$r_0 = (\ x \wedge \neg y \wedge \neg z\ ) \vee (\ z \wedge \neg w \wedge \neg x\ )$$

Example

Consider an example 4-input, 2-output function:

w	x	y	z	r_1	r_0
0	0	0	0	0	0
0	0	0	1	0	1
0	0	1	0	1	0
0	0	1	1	?	?
0	1	0	0	0	1
0	1	0	1	1	0
0	1	1	0	0	0
0	1	1	1	?	?
1	0	0	0	1	0
1	0	0	1	0	0
1	0	1	0	0	1
1	0	1	1	?	?
1	1	0	0	?	?
1	1	0	1	?	?
1	1	1	0	?	?
1	1	1	1	?	?





Each group translates into one term of the SoP form expressions

$$r_1 = (\ w \wedge \neg y \wedge \neg z\ ) \vee (\ y \wedge \neg w \wedge \neg x\ ) \vee (\ x \wedge z\ )$$

$$r_0 = (\ x \wedge \neg y \wedge \neg z\ ) \vee (\ z \wedge \neg w \wedge \neg x\ ) \vee (\ w \wedge y\ )$$

Conclusions

- Take away points:
  1. There are a *huge* number of challenges, even with (relatively) simple problems, e.g.,
     - how do we describe what the design should do?
     - how do we structure the design?
     - what sort of standard cell library do we use?
     - do we aim for the fewest gates?
     - do we aim for shortest critical path?
     - how do we cope with propagation delay and fan-out?
     - ...
  2. The three themes we've covered, i.e.,
     - high-level design patterns,
     - low-level, mechanical derivation and optimisation of Boolean expressions,
     - building-block components,

     allows us to address such challenges: in combination, they support development of effective (combinatorial) design and implementation.
  3. In many cases, use of appropriate **Electronic Design Automation (EDA)** tools can provide (semi-)automatic solutions.

Additional Reading

- *Wikipedia: Combinational logic*. URL: https://en.wikipedia.org/wiki/Combinational_logic.
- D. Page. "Chapter 2: Basics of digital logic". In: *A Practical Introduction to Computer Architecture*. 1st ed. Springer, 2009.
- W. Stallings. "Chapter 11: Digital logic". In: *Computer Organisation and Architecture*. 9th ed. Prentice Hall, 2013.
- A.S. Tanenbaum and T. Austin. "Section 3.2.2: Combinatorial circuits". In: *Structured Computer Organisation*. 6th ed. Prentice Hall, 2012.

References

[1] *Wikipedia: Combinational logic*. URL: https://en.wikipedia.org/wiki/Combinational_logic (see p. 59).

[2] D. Page. "Chapter 2: Basics of digital logic". In: *A Practical Introduction to Computer Architecture*. 1st ed. Springer, 2009 (see p. 59).

[3] W. Stallings. "Chapter 11: Digital logic". In: *Computer Organisation and Architecture*. 9th ed. Prentice Hall, 2013 (see p. 59).

[4] A.S. Tanenbaum and T. Austin. "Section 3.2.2: Combinatorial circuits". In: *Structured Computer Organisation*. 6th ed. Prentice Hall, 2012 (see p. 59).