COMS12200 lecture: week #4

- The logic gates we've built are high*er*-level components than the low-level transistors we started off with ...
- ... now we can start to build genuinely *high*-level components:
  - We'd like components that are clos(er) to we might regard as useful computation.
  - The first step is to build high-level components that perform some computation continuously, i.e., continuously compute an output given some inputs.
  - We call such designs **combinatorial** logic; they just combine low-level logic gates together.
- Challenge: given the specification (e.g., a truth table) for some Boolean function f, produce a Boolean expression e (i.e., implement a circuit) that can compute it.

Notes:

An Aside: Design patterns

- **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

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

  - How? We simply let

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

Notes:

- Strictly speaking, f is a **vectorial** Boolean function whereas each f_i is a genuine Boolean function (in the sense they each have a 1-bit output).

An Aside: Design patterns

- **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 since clearly $r = r'$; this essentially shares the gate between two usage points.

Notes:

An Aside: Design patterns

- **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; if x and y are m-bit values then
    $$r = x \wedge y$$
    is computed by
    $$r_i = x_i \wedge y_i$$
    for $0 \leq i < m$, i.e., m separate instantiations of the 2-input, 1-bit gate.

Notes:

An Aside: Design patterns

- **Cascading**:
  - Imagine, for example, that we are given a 2-input, 1-bit AND gate.
  - An n-input, 1-bit AND gate is simply a cascade of 2-input, 1-bit AND gates, i.e.,

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

for $n = 4$ is the same as

$$r = (x_0 \wedge x_1) \wedge (x_2 \wedge x_3).$$

  - Notice we've built an AND tree, and that the balanced tree above is more attractive than equivalents such as

$$r = x_0 \wedge (x_1 \wedge (x_2 \wedge x_3)).$$

Notes:

Mechanical 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.

Notes:

Mechanical Derivation (2): Method #1

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

Following the algorithm produces:

Notes:

Mechanical Derivation (2): Method #1

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	$\rightsquigarrow \quad i = 1$
1	0	1	$\rightsquigarrow \quad i = 2$
1	1	0	

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.

Notes:

Mechanical Derivation (2): Method #1

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	$\rightsquigarrow$	$t_1 = \neg x \wedge y$
1	0	1	$\rightsquigarrow$	$t_2 = x \wedge \neg y$
1	1	0		

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 $t_2 = x \wedge \neg y$.

Notes:

Mechanical Derivation (2): Method #1

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	$\rightsquigarrow$	$t_1 = \neg x \wedge y$
1	0	1	$\rightsquigarrow$	$t_2 = x \wedge \neg y$
1	1	0		

Following the algorithm produces:

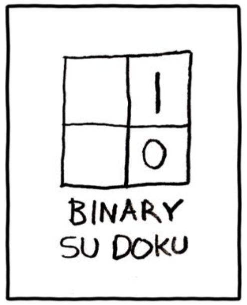
3. The expression implementing the function is therefore

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

   which is in SoP form.

Notes:

Mechanical Derivation (3): Method #2 (Karnaugh map)



http://xkcd.com/74/

Notes:

Mechanical Derivation (4): Method #2 (Karnaugh map)

Algorithm (Karnaugh map)

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 $(p \times q)$-element grid, st. $p \cdot q = 2^n$ 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.

Notes:

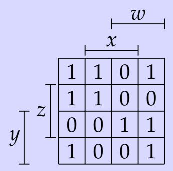
- Note that the typesetting of Karnaugh maps in these slides is a bit awkward, and arguably not standard, basically due to the difficulty of typesetting small fonts.
  Rather than write the binary values of each variable along the top and side, instead the rendered image shows where they are 1 via a range marked by a line: the variable is 1 within cells inside the range, and 0 otherwise.

Mechanical Derivation (5): Method #2 (Karnaugh map)

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



1	1	0	1
1	1	0	0
0	0	1	1
1	0	0	1

Each group translates into one term of the SoP form expression

$$r \;=$$

Notes:

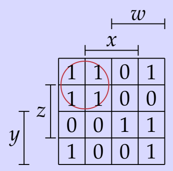
- The first two steps wrt. use of a Karnaugh map are simple: drawing a grid of an appropriate size then filling it with entries from the truth table are both trivial.
- Forming the groups is a little harder, in the sense that their "quality" (i.e., number and size) dictates how optimised the resulting expression will be. This example has some non-intuitive cases: the blue group might not be obvious for example, *but* is within the rules (although it is sort of inside-out, it is rectangular and a power-of-two in size).
- The last step is the hardest: we need to translate each group into a term that covers it. Put another way, we want a term that specifies just the cells in that group. In a sense, the more variables that exist within a term place more restrictions on which cells we specify; this highlights the fact that larger groups therefor contain fewer variables, and are therefore simpler. This example has three groups:
  - The red group spans columns 0 and 1 and rows 0 and 1; provided $w = 0$ and $y = 0$ we specify *just* those cells, so the expression is $\neg w \wedge \neg y$. That is, $w = 0$ restricts us to columns 0 and 1 (columns 2 and 3 have $w = 1$) and $y = 0$ restricts us to rows 0 and 1 (rows 2 and 3 have $y = 1$). Note that the values of x and z don't matter: cells in the group hold the value 1 regardless of x and y.
  - The green group spans columns 2 and 3 in row 2; provided $w = 1$, $y = 1$ and $z = 1$ we specify *just* those cells, so the expression is $w \wedge y \wedge z$. That is, $w = 1$ restricts us to columns 2 and 3 (columns 0 and 1 have $w = 0$) and $y = 1$ and $z = 1$ restricts us to row 2 (rows 0, 1 and 3 have at least one of $y = 0$ or $z = 0$). Note that the value of x doesn't matter: cells in the group hold the value 1 regardless of x.
  - The blue group spans columns 0 and 3 and rows 0 and 3; provided $x = 0$ and $z = 0$ we specify *just* those cells, so the expression is $\neg x \wedge \neg z$. That is, $x = 0$ restricts us to columns 0 and 3 (columns 1 and 2 have $x = 1$) and $z = 0$ restricts us to rows 0 and 3 (rows 1 and 2 have $z = 1$). Note that the values of w and y don't matter: cells in the group hold the value 1 regardless of w and y.

Mechanical Derivation (5): Method #2 (Karnaugh map)

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



1	1	0	1
1	1	0	0
0	0	1	1
1	0	0	1

Each group translates into one term of the SoP form expression

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

Notes:

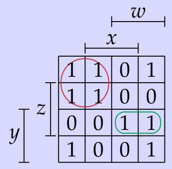
- The first two steps wrt. use of a Karnaugh map are simple: drawing a grid of an appropriate size then filling it with entries from the truth table are both trivial.
- Forming the groups is a little harder, in the sense that their "quality" (i.e., number and size) dictates how optimised the resulting expression will be. This example has some non-intuitive cases: the blue group might not be obvious for example, *but* is within the rules (although it is sort of inside-out, it is rectangular and a power-of-two in size).
- The last step is the hardest: we need to translate each group into a term that covers it. Put another way, we want a term that specifies just the cells in that group. In a sense, the more variables that exist within a term place more restrictions on which cells we specify; this highlights the fact that larger groups therefor contain fewer variables, and are therefore simpler. This example has three groups:
  - The red group spans columns 0 and 1 and rows 0 and 1; provided $w = 0$ and $y = 0$ we specify *just* those cells, so the expression is $\neg w \wedge \neg y$. That is, $w = 0$ restricts us to columns 0 and 1 (columns 2 and 3 have $w = 1$) and $y = 0$ restricts us to rows 0 and 1 (rows 2 and 3 have $y = 1$). Note that the values of x and z don't matter: cells in the group hold the value 1 regardless of x and y.
  - The green group spans columns 2 and 3 in row 2; provided $w = 1$, $y = 1$ and $z = 1$ we specify *just* those cells, so the expression is $w \wedge y \wedge z$. That is, $w = 1$ restricts us to columns 2 and 3 (columns 0 and 1 have $w = 0$) and $y = 1$ and $z = 1$ restricts us to row 2 (rows 0, 1 and 3 have at least one of $y = 0$ or $z = 0$). Note that the value of x doesn't matter: cells in the group hold the value 1 regardless of x.
  - The blue group spans columns 0 and 3 and rows 0 and 3; provided $x = 0$ and $z = 0$ we specify *just* those cells, so the expression is $\neg x \wedge \neg z$. That is, $x = 0$ restricts us to columns 0 and 3 (columns 1 and 2 have $x = 1$) and $z = 0$ restricts us to rows 0 and 3 (rows 1 and 2 have $z = 1$). Note that the values of w and y don't matter: cells in the group hold the value 1 regardless of w and y.

Mechanical Derivation (5): Method #2 (Karnaugh map)

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

$$
\begin{array}{rcl}
r & = & (\neg w \wedge \neg y) \vee \\
  &   & (w \wedge y \wedge z)
\end{array}
$$

Notes:

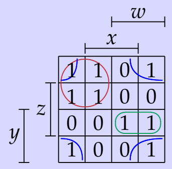
- The first two steps wrt. use of a Karnaugh map are simple: drawing a grid of an appropriate size then filling it with entries from the truth table are both trivial.
- Forming the groups is a little harder, in the sense that their "quality" (i.e., number and size) dictates how optimised the resulting expression will be. This example has some non-intuitive cases: the blue group might not be obvious for example, *but* is within the rules (although it is sort of inside-out, it is rectangular and a power-of-two in size).
- The last step is the hardest: we need to translate each group into a term that covers it. Put another way, we want a term that specifies just the cells in that group. In a sense, the more variables that exist within a term place more restrictions on which cells we specify; this highlights the fact that larger groups therefor contain fewer variables, and are therefore simpler. This example has three groups:
  - The red group spans columns 0 and 1 and rows 0 and 1; provided $w = 0$ and $y = 0$ we specify *just* those cells, so the expression is $\neg w \wedge \neg y$. That is, $w = 0$ restricts us to columns 0 and 1 (columns 2 and 3 have $w = 1$) and $y = 0$ restricts us to rows 0 and 1 (rows 2 and 3 have $y = 1$). Note that the values of x and z don't matter: cells in the group hold the value 1 regardless of x and y.
  - The green group spans columns 2 and 3 in row 2; provided $w = 1$, $y = 1$ and $z = 1$ we specify *just* those cells, so the expression is $w \wedge y \wedge z$. That is, $w = 1$ restricts us to columns 2 and 3 (columns 0 and 1 have $w = 0$) and $y = 1$ and $z = 1$ restricts us to row 2 (rows 0, 1 and 3 have at least one of $y = 0$ or $z = 0$). Note that the value of x doesn't matter: cells in the group hold the value 1 regardless of x.
  - The blue group spans columns 0 and 3 and rows 0 and 3; provided $x = 0$ and $z = 0$ we specify *just* those cells, so the expression is $\neg x \wedge \neg z$. That is, $x = 0$ restricts us to columns 0 and 3 (columns 1 and 2 have $x = 1$) and $z = 0$ restricts us to rows 0 and 3 (rows 1 and 2 have $z = 1$). Note that the values of w and y don't matter: cells in the group hold the value 1 regardless of w and y.

Mechanical Derivation (5): Method #2 (Karnaugh map)

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

$$
\begin{array}{rcl}
r & = & (\neg w \wedge \neg y) \vee \\
  &   & (w \wedge y \wedge z) \vee \\
  &   & (\neg x \wedge \neg z)
\end{array}
$$

Notes:

- The first two steps wrt. use of a Karnaugh map are simple: drawing a grid of an appropriate size then filling it with entries from the truth table are both trivial.
- Forming the groups is a little harder, in the sense that their "quality" (i.e., number and size) dictates how optimised the resulting expression will be. This example has some non-intuitive cases: the blue group might not be obvious for example, *but* is within the rules (although it is sort of inside-out, it is rectangular and a power-of-two in size).
- The last step is the hardest: we need to translate each group into a term that covers it. Put another way, we want a term that specifies just the cells in that group. In a sense, the more variables that exist within a term place more restrictions on which cells we specify; this highlights the fact that larger groups therefor contain fewer variables, and are therefore simpler. This example has three groups:
  - The red group spans columns 0 and 1 and rows 0 and 1; provided $w = 0$ and $y = 0$ we specify *just* those cells, so the expression is $\neg w \wedge \neg y$. That is, $w = 0$ restricts us to columns 0 and 1 (columns 2 and 3 have $w = 1$) and $y = 0$ restricts us to rows 0 and 1 (rows 2 and 3 have $y = 1$). Note that the values of x and z don't matter: cells in the group hold the value 1 regardless of x and y.
  - The green group spans columns 2 and 3 in row 2; provided $w = 1$, $y = 1$ and $z = 1$ we specify *just* those cells, so the expression is $w \wedge y \wedge z$. That is, $w = 1$ restricts us to columns 2 and 3 (columns 0 and 1 have $w = 0$) and $y = 1$ and $z = 1$ restricts us to row 2 (rows 0, 1 and 3 have at least one of $y = 0$ or $z = 0$). Note that the value of x doesn't matter: cells in the group hold the value 1 regardless of x.
  - The blue group spans columns 0 and 3 and rows 0 and 3; provided $x = 0$ and $z = 0$ we specify *just* those cells, so the expression is $\neg x \wedge \neg z$. That is, $x = 0$ restricts us to columns 0 and 3 (columns 1 and 2 have $x = 1$) and $z = 0$ restricts us to rows 0 and 3 (rows 1 and 2 have $z = 1$). Note that the values of w and y don't matter: cells in the group hold the value 1 regardless of w and y.

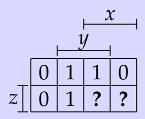
Mechanical Derivation (6): Method #2 (Karnaugh map)

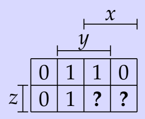
- Karnaugh maps can accommodate **don't care** states:
  - Sometimes we don't care whether the output for a given input combination is 0 or 1; we mark this with **?** rather than 0 or 1.
  - Although we don't care what value a ? element takes, treating it like a 1 allows us to include it in a group and as 0 to avoid covering it.

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

$r =$ $\qquad\qquad$ $r =$

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

Notes:

- Note the differing shape of this grid versus the previous example: since there are $n = 3$ variables, we set $p = 2$ and $q = 4$ st. the grid contains $2 \cdot 4 = 2^3 = 8$ cells.
- Adopting the same approach as the previous example, by not ignoring the don't care states (i.e., assuming they are 0) we have two groups. However, opting to treat one of them as a 1 (which is fine: by definition we don't care what the output is) we only have one:
  - The red group spans column 1 and rows 0 and 1; provided $x = 0$ and $y = 1$ we specify *just* those cells, so the expression is $\neg x \wedge y$. That is, $x = 0$ and $y = 1$ restricts us to column 1 (columns 0, 2 and 3 have at least one of $x = 1$ or $y = 0$) which is all we need because the group spans *all* rows. Note that the value of z doesn't matter: cells in the group hold the value 1 regardless of z.
  - The green group spans columns 1 and 2, in row 0; provided $y = 1$ and $z = 0$ we specify *just* those cells, so the expression is $y \wedge \neg z$. That is, $y = 1$ restricts us to columns 1 and 2 (columns 0 and 3 have $y = 0$) and $z = 0$ restricts us to row 0 (row 1 has $z = 1$). Note that the value of x doesn't matter: cells in the group hold the value 1 regardless of x.
  - The blue group spans columns 1 and 2 and rows 0 and 1; provided $y = 1$ we specify *just* those cells, so the expression is y That is, $y = 1$ restricts us to columns 1 and 2 (columns 0 and 3 have $y = 0$) which is all we need because the group spans *all* rows. Note that the values of x and z don't matter: cells in the group hold the value 1 regardless of x and z.

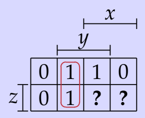
Mechanical Derivation (6): Method #2 (Karnaugh map)

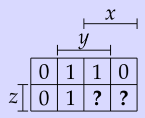
- Karnaugh maps can accommodate **don't care** states:
  - Sometimes we don't care whether the output for a given input combination is 0 or 1; we mark this with **?** rather than 0 or 1.
  - Although we don't care what value a ? element takes, treating it like a 1 allows us to include it in a group and as 0 to avoid covering it.

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

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

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

Notes:

- Note the differing shape of this grid versus the previous example: since there are $n = 3$ variables, we set $p = 2$ and $q = 4$ st. the grid contains $2 \cdot 4 = 2^3 = 8$ cells.
- Adopting the same approach as the previous example, by not ignoring the don't care states (i.e., assuming they are 0) we have two groups. However, opting to treat one of them as a 1 (which is fine: by definition we don't care what the output is) we only have one:
  - The red group spans column 1 and rows 0 and 1; provided $x = 0$ and $y = 1$ we specify *just* those cells, so the expression is $\neg x \wedge y$. That is, $x = 0$ and $y = 1$ restricts us to column 1 (columns 0, 2 and 3 have at least one of $x = 1$ or $y = 0$) which is all we need because the group spans *all* rows. Note that the value of z doesn't matter: cells in the group hold the value 1 regardless of z.
  - The green group spans columns 1 and 2, in row 0; provided $y = 1$ and $z = 0$ we specify *just* those cells, so the expression is $y \wedge \neg z$. That is, $y = 1$ restricts us to columns 1 and 2 (columns 0 and 3 have $y = 0$) and $z = 0$ restricts us to row 0 (row 1 has $z = 1$). Note that the value of x doesn't matter: cells in the group hold the value 1 regardless of x.
  - The blue group spans columns 1 and 2 and rows 0 and 1; provided $y = 1$ we specify *just* those cells, so the expression is y That is, $y = 1$ restricts us to columns 1 and 2 (columns 0 and 3 have $y = 0$) which is all we need because the group spans *all* rows. Note that the values of x and z don't matter: cells in the group hold the value 1 regardless of x and z.

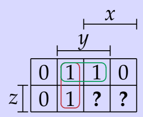
Mechanical Derivation (6): Method #2 (Karnaugh map)

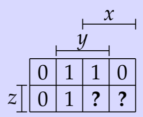
- Karnaugh maps can accommodate **don't care** states:
  - Sometimes we don't care whether the output for a given input combination is 0 or 1; we mark this with **?** rather than 0 or 1.
  - Although we don't care what value a ? element takes, treating it like a 1 allows us to include it in a group and as 0 to avoid covering it.

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

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

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

Notes:

- Note the differing shape of this grid versus the previous example: since there are $n = 3$ variables, we set $p = 2$ and $q = 4$ st. the grid contains $2 \cdot 4 = 2^3 = 8$ cells.
- Adopting the same approach as the previous example, by not ignoring the don't care states (i.e., assuming they are 0) we have two groups. However, opting to treat one of them as a 1 (which is fine: by definition we don't care what the output is) we only have one:
  - The red group spans column 1 and rows 0 and 1; provided $x = 0$ and $y = 1$ we specify *just* those cells, so the expression is $\neg x \wedge y$. That is, $x = 0$ and $y = 1$ restricts us to column 1 (columns 0, 2 and 3 have at least one of $x = 1$ or $y = 0$) which is all we need because the group spans *all* rows. Note that the value of z doesn't matter: cells in the group hold the value 1 regardless of z.
  - The green group spans columns 1 and 2, in row 0; provided $y = 1$ and $z = 0$ we specify *just* those cells, so the expression is $y \wedge \neg z$. That is, $y = 1$ restricts us to columns 1 and 2 (columns 0 and 3 have $y = 0$) and $z = 0$ restricts us to row 0 (row 1 has $z = 1$). Note that the value of x doesn't matter: cells in the group hold the value 1 regardless of x.
  - The blue group spans columns 1 and 2 and rows 0 and 1; provided $y = 1$ we specify *just* those cells, so the expression is y That is, $y = 1$ restricts us to columns 1 and 2 (columns 0 and 3 have $y = 0$) which is all we need because the group spans *all* rows. Note that the values of x and z don't matter: cells in the group hold the value 1 regardless of x and z.

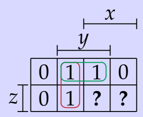
Mechanical Derivation (6): Method #2 (Karnaugh map)

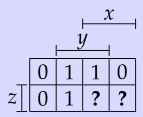
- Karnaugh maps can accommodate **don't care** states:
  - Sometimes we don't care whether the output for a given input combination is 0 or 1; we mark this with **?** rather than 0 or 1.
  - Although we don't care what value a ? element takes, treating it like a 1 allows us to include it in a group and as 0 to avoid covering it.

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

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

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

Notes:

- Note the differing shape of this grid versus the previous example: since there are $n = 3$ variables, we set $p = 2$ and $q = 4$ st. the grid contains $2 \cdot 4 = 2^3 = 8$ cells.
- Adopting the same approach as the previous example, by not ignoring the don't care states (i.e., assuming they are 0) we have two groups. However, opting to treat one of them as a 1 (which is fine: by definition we don't care what the output is) we only have one:
  - The red group spans column 1 and rows 0 and 1; provided $x = 0$ and $y = 1$ we specify *just* those cells, so the expression is $\neg x \wedge y$. That is, $x = 0$ and $y = 1$ restricts us to column 1 (columns 0, 2 and 3 have at least one of $x = 1$ or $y = 0$) which is all we need because the group spans *all* rows. Note that the value of z doesn't matter: cells in the group hold the value 1 regardless of z.
  - The green group spans columns 1 and 2, in row 0; provided $y = 1$ and $z = 0$ we specify *just* those cells, so the expression is $y \wedge \neg z$. That is, $y = 1$ restricts us to columns 1 and 2 (columns 0 and 3 have $y = 0$) and $z = 0$ restricts us to row 0 (row 1 has $z = 1$). Note that the value of x doesn't matter: cells in the group hold the value 1 regardless of x.
  - The blue group spans columns 1 and 2 and rows 0 and 1; provided $y = 1$ we specify *just* those cells, so the expression is y That is, $y = 1$ restricts us to columns 1 and 2 (columns 0 and 3 have $y = 0$) which is all we need because the group spans *all* rows. Note that the values of x and z don't matter: cells in the group hold the value 1 regardless of x and z.

Dan Page (page@cs.bris.ac.uk) © 2014-5
Computer Architecture (new format) Slide 11 of 14

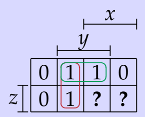
Mechanical Derivation (6): Method #2 (Karnaugh map)

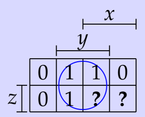
- Karnaugh maps can accommodate **don't care** states:
  - Sometimes we don't care whether the output for a given input combination is 0 or 1; we mark this with **?** rather than 0 or 1.
  - Although we don't care what value a ? element takes, treating it like a 1 allows us to include it in a group and as 0 to avoid covering it.

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

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

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

Notes:

- Note the differing shape of this grid versus the previous example: since there are $n = 3$ variables, we set $p = 2$ and $q = 4$ st. the grid contains $2 \cdot 4 = 2^3 = 8$ cells.
- Adopting the same approach as the previous example, by not ignoring the don't care states (i.e., assuming they are 0) we have two groups. However, opting to treat one of them as a 1 (which is fine: by definition we don't care what the output is) we only have one:
  - The red group spans column 1 and rows 0 and 1; provided $x = 0$ and $y = 1$ we specify *just* those cells, so the expression is $\neg x \wedge y$. That is, $x = 0$ and $y = 1$ restricts us to column 1 (columns 0, 2 and 3 have at least one of $x = 1$ or $y = 0$) which is all we need because the group spans *all* rows. Note that the value of z doesn't matter: cells in the group hold the value 1 regardless of z.
  - The green group spans columns 1 and 2, in row 0; provided $y = 1$ and $z = 0$ we specify *just* those cells, so the expression is $y \wedge \neg z$. That is, $y = 1$ restricts us to columns 1 and 2 (columns 0 and 3 have $y = 0$) and $z = 0$ restricts us to row 0 (row 1 has $z = 1$). Note that the value of x doesn't matter: cells in the group hold the value 1 regardless of x.
  - The blue group spans columns 1 and 2 and rows 0 and 1; provided $y = 1$ we specify *just* those cells, so the expression is y That is, $y = 1$ restricts us to columns 1 and 2 (columns 0 and 3 have $y = 0$) which is all we need because the group spans *all* rows. Note that the values of x and z don't matter: cells in the group hold the value 1 regardless of x and z.

Conclusions

- Take away points:
  1. There are a *huge* number of issues to consider when designing even simple components, e.g.,
     - how do we describe what the circuit should do?
     - what sort of standard 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 design patterns, and mechanical techniques are the key concepts to focus on: these allow you do produce an effective implementation most easily.
  3. In many cases, use of appropriate **Electronic Design Automation (EDA)** tools can provide (semi-)automatic solutions.

Notes:

References and Further Reading

[1] Wikipedia: Gray code.
http://en.wikipedia.org/wiki/Gray_code.

[2] Wikipedia: Karnaugh map.
http://en.wikipedia.org/wiki/Karnaugh_map.

[3] M. Karnaugh.
The map method for synthesis of combinatorial logic circuits.
Transactions of American Institute of Electrical Engineers, 72(9):593–599, 1953.

[4] D. Page.
Chapter 2: Basics of digital logic.
In *A Practical Introduction to Computer Architecture*. Springer-Verlag, 1st edition, 2009.

[5] W. Stallings.
Chapter 11: Digital logic.
In *Computer Organisation and Architecture*. Prentice-Hall, 9th edition, 2013.

[6] A.S. Tanenbaum.
Section 3.1: Gates and Boolean algebra.
In *Structured Computer Organisation* [8].

[7] A.S. Tanenbaum.
Section 3.2: Basic digital logic circuits.
In *Structured Computer Organisation* [8].

Notes:

References and Further Reading

[8] A.S. Tanenbaum.
Structured Computer Organisation.
Prentice-Hall, 6th edition, 2012.

Notes: