Logic gate level

Part 3: minimizing circuits
Improving circuit efficiency

• Efficiency of combinatorial circuit depends on number & arrangement of its gates
• Sum-of-products expansion can aid in finding set of logic gates to implement circuit, but may contain more terms than necessary
• Combining terms in sum-of-products expansion leads to simpler expression of circuit, using fewer gates and inputs
Why minimize circuits?

- Reducing number of gates can lead to more reliable, less expensive chip
- Minimization means more circuits per chip
- Minimization reduces time used by circuit to complete its output
- Downside: minimization algorithms are computationally intensive; no algorithm has yet been devised to minimize circuits involving more than 25 variables
Minimizing $a'b'd' + a'c'd' + a'bc'd'$

- Using the commutative and associative properties, we can write the original expression as:
  $$a'b'd' + (a'c'd') + (a'c'd')b$$

- Recall absorption property:
  $$x + xy = x$$ and $$x(x+y) = x$$

- If we let $$x = (a'c'd')$$ and $$y = b$$, we get:
  $$a'b'd' + a'c'd'$$
Minimized circuit
Minimizing number of gates

- Two-level networks are desirable because of their speed
- Can sometimes reduce number of gate in 2-level network and retain processing speed of 2 gate delays
Minimizing number of gates

• To minimize a 2-level net, transform expression into combination of minterms:
  – minterm: ANDed expression that contains all input variables exactly once
  – Can transform any OR of ANDs to an OR of minterms
Example

• Suppose $X(a,b,c) = abc + a'bc + ab$

• First two terms (abc and a’bc) are already minterms; to transform the third:
  
  $ab = ab(c+c') = abc + abc'$

• Since abc is already in the expression, we can strike the duplicate, and end up with:
  
  $abc + a’bc + abc'$
Canonical expression

• Canonical expression: OR of set of unique minterms
• Each minterm in expression represents a 1 in truth table result column
• Sigma notation: shorthand for canonical expression (see following example)
Sigma notation

• For the expression we minimized a couple of slides ago, the canonical expression is $a'bc + abc' + abc$, which is depicted in the truth table to the right.

• Rows 3, 6, and 7 depict the expression, so the sigma notation is: $X(a,b,c) = \Sigma(3,6,7)$

<table>
<thead>
<tr>
<th>Row #</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>result (X)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
The canonical expression is the OR of minterms; its dual is the AND of OR terms

- Each term contains all inputs
- No duplicate terms
- Corresponds to 0s in truth table
- Uses Pi notation instead of Sigma

Dual canonical expression for \(a'bc + abc' + abc\) is

\[(a+b+c)(a+b+c')(a+b'+c)(a'+b+c)(a'+b+c')\]
Dual canonical expression

- Uses same truth table as before, but this time we look at the 0 rows
- Pi notation is:
  \[ X(a,b,c) = \Pi(0,1,2,4,5) \]
Karnaugh maps

• Minimization of 2-level networks is based on concept of distance
• Distance between 2 minterms is the number of places in which they differ
• A Karnaugh map is a kind of truth table arranged so that adjacent entries represent minterms that differ by one
Karnaugh maps

• Graphical method for finding terms to combine in a Boolean function involving a relatively small number of variables

• For a Boolean function in 2 variables, there are four possible minterms in the sum-of-products expansion

• A K-map for such a function consists of 4 cells, with a 1 placed in the cell representing a minterm if it is present in the expansion

• An even numbered group of adjacent 1s represents a term that can be eliminated without changing the outcome of the circuit
Example 1

Suppose you have the function $F(x,y)$, represented by the table below:

<table>
<thead>
<tr>
<th>$x$</th>
<th>$y$</th>
<th>$F$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

The sum-of-products expansion of $F$ is $xy + xy'$

The circuit that corresponds to this sum-of-products expansion is shown below:
Example 1

Using a Karnaugh map, we can simplify this circuit considerably:

When 1s occur in two adjacent cells in the K-map, the minterms represented by those cells can be combined into a product involving just one of the variables.

We can see from the K-map that the value of y doesn’t affect the outcome of the function; in general, when there are 1s in two adjacent cells in the K-map, the minterms represented by those cells can be combined into a product involving just one of the variables.
Example 1

Finally, then, the function $F$ with the truth table:

<table>
<thead>
<tr>
<th>$x$</th>
<th>$y$</th>
<th>$F$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

And the K-map:

Can be represented with the circuit:

We can verify this using laws of Boolean algebra and the original sum-of-products expression:

$$xy + xy' = x(y + y') = x(1) = x$$ (using distribution, complement, and identity laws)
Example 2

Truth table for expression \( X \):

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Canonical expression:
\( X = a'b' + ab' + ab \)

K-map:

<table>
<thead>
<tr>
<th></th>
<th>b'</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>a'</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>a</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Circled column shows:
\( a'b' + ab' = b' \)

Circled row shows:
\( ab' + ab = a \)

So expression simplifies to:
\( X = a + b' \)
Karnaugh maps in 3 or more variables

• A K-map in n variables is a grid of $2^n$ cells
  – Each cell represents the possible minterms in n variables
  – Two cells in a K-map are adjacent if the corresponding minterms differ in exactly one literal
  – Cells may be adjacent even if the map doesn’t show them next to one another

• The table below shows the minterm values for a 3-variable K-map; note that 000 and 101, for example, differ by 1 term and are thus considered adjacent

<table>
<thead>
<tr>
<th></th>
<th>000</th>
<th>001</th>
<th>011</th>
<th>010</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>000</td>
<td>001</td>
<td>011</td>
<td>010</td>
</tr>
<tr>
<td>10</td>
<td>100</td>
<td>101</td>
<td>111</td>
<td>110</td>
</tr>
</tbody>
</table>
Example 3

Use a K-map to minimize this sum-of-products expansion:

\[ xyz + x'y'z + xy'z \]

The Karnaugh map shows two different areas of adjacency:

This shows that \( yz = xyz + x'y'z \)

But we can also see that \( xz = xyz + xy'z \)

So we can construct the circuit using either \( yz + xy'z \) or \( xz + x'y'z \)
Implicants

• To simplify a sum-of-products expansion, we use a K-map to identify blocks of minterms we can combine
• In a function with 3 variables:
  – blocks of 2 adjacent cells represent pairs that can be combined in the product of 2 literals
  – 2x2 and 4x1 blocks represent minterms that can be combined into a single literal
  – a block consisting of all 8 cells represents 1
• The product of literals corresponding to a block of 1s in a K-map is called an implicant of the function
Prime implicants

• If a block of 1s in a K-map is not contained in a larger block of 1s representing the product of fewer literals, the block is a prime implicant.

• The goal in using K-maps is to identify the largest possible blocks of 1s, but we must include blocks representing isolated 1s; such a block is an essential prime implicant.

• We can express the sum of products as the sum of prime implicants; as in the previous example, there may be more than one way to do this.
Example 4

- 3-variable K-map; read across as:
  \( a'b'c' \) (0), \( a'b'c \) (0), \( a'bc \) (1), \( a'bc' \) (0),
  \( ab'c' \) (0), \( ab'c \) (0), \( abc \) (1), \( abc' \) (1)

- Note how each term listed above differs from the one next to it by exactly one variable
Minimization

• Find set of ovals that covers all the ones:

(a) $a'bc + abc = bc$
(b) $abc + abc' = ab$
(c) $x = bc + ab$
K-maps & Sigma notation

Illustration at left shows how a K-map for a 3-variable expression corresponds to the numbered rows of truth table used to create sigma notation.

Easier to remember if you know binary equivalents of decimal labels & 2 general rules:

- adjacent bit strings differ by single bit
- 1st digit of top row cells always 0, bottom row always 1
Examples

$\Sigma(0,3,4,7)$ has K-map:

<table>
<thead>
<tr>
<th></th>
<th>$b'c'$</th>
<th>$b'c$</th>
<th>$bc$</th>
<th>$bc'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a'$</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$a$</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

$b'c' + bc$

$\Sigma(1,3,4,6)$ has K-map:

<table>
<thead>
<tr>
<th></th>
<th>$b'c'$</th>
<th>$b'c$</th>
<th>$bc$</th>
<th>$bc'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a'$</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$a$</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

$a'c + ac'$
One more example

\[ \Sigma(0,2,4,6,7) \]

<table>
<thead>
<tr>
<th></th>
<th>b’c’</th>
<th>b’c</th>
<th>bc</th>
<th>bc’</th>
</tr>
</thead>
<tbody>
<tr>
<td>a’</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>a</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Couple of different ways to group – but we should always pick the largest grouping.
In this case, it involves both ends – we can see that neither a nor b matters if c’ is present.
Leads to minimized expression: c’ + abc
K-map for 4 variables

(a) Decimal labels for the minterms in the Karnaugh map.

(b) The regions where the variables are 1.
K-maps & don’t care conditions

• It isn’t always necessary to process all possible input combinations, since some are never expected to be present
• Such input combinations are called don’t care conditions, since we don’t care about the outputs they’d produce should they ever be present
K-maps & don’t care conditions

• With don’t care condition present, you can arbitrarily choose either 1 or 0 for output

• Choice of output (1 or 0) for don’t care conditions can aid in minimization

• Sigma notation for don’t care conditions:
  \[ \Sigma(x,y,z) + d(a,b) \] where x,y,z,a and b all represent lines in the function’s truth table

• We represent don’t care conditions in a K-map with Xs
Example

• K-map for $X(a,b,c) = \Sigma(2,4,6) + d(0,7)$

<table>
<thead>
<tr>
<th></th>
<th>X</th>
<th></th>
<th></th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>X</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• With d.c. conditions, since we don’t care, we can choose to include, or not include, boxes with X designations in K-maps
  ✓ X is “wildcard” condition – can be treated as either 1 or 0
  ✓ In K-map above, if minterm 0 is treated as 1 and 7 as 0, get $\Sigma(0,2,4,6) = c'$
K-maps: summary

• To minimize a Boolean function in n variables, we draw a K-map of appropriate size
• Place 1s in all cells corresponding to minterms of the sum-of-products expansion
• Identify all prime implicants:
  – look for blocks of $2^k$ clustered cells containing 1s (where $n > k > 1$) - these correspond to product of n-k literals
Summary continued

• Once all prime implicants identified, find the smallest possible subset that covers all the 1s in the K-map
• Begin by selecting essential prime implicants
• Add additional implicants to ensure that all 1s are covered
• Since K-maps are graphical tool, they are difficult to automate
Quine-McCluskey method

• Provides procedure for simplifying sum-of-products that can be mechanized
• Doesn’t rely on visual inspection (as K-maps do)
• Can be used for Boolean functions in any number of variables (K-maps get awkward beyond 5 or 6)
• Major disadvantage: algorithm is exponential
Quine-McCluskey method

• Two parts:
  – Find terms that are candidates for inclusion in minimal expansion as Boolean sum of Boolean products
  – Determine which of these terms should actually be used
Quine-McCluskey method: step 1

• Express each minterm in n variables as a bit string of length n
  – use a 1 in the ith position if $x_i$ occurs
  – use a 0 in the position if $x_i'$ occurs

• For example, in the function represented by the sum-of-products $xyz + xy'z + x'yz' + x'y'z$, the bit strings are: 111, 101, 010, 001
Quine-McCluskey method: step 2

• Group the bit strings according to the number of 1s in them, as illustrated in the table below:

<table>
<thead>
<tr>
<th>Minterm</th>
<th>Bit string</th>
<th># of 1s</th>
</tr>
</thead>
<tbody>
<tr>
<td>xyz</td>
<td>111</td>
<td>3</td>
</tr>
<tr>
<td>xy'z</td>
<td>101</td>
<td>2</td>
</tr>
<tr>
<td>x'y'z'</td>
<td>010</td>
<td>1</td>
</tr>
<tr>
<td>x'y'z</td>
<td>001</td>
<td>1</td>
</tr>
</tbody>
</table>
Quine-McCluskey method: step 3

• Determine all products in n-1 variables that can be formed by taking the Boolean sum of minterms in the expansion
  – minterms that can be combined are represented by bit strings that differ in exactly one position
  – Use strings to represent the products with:
    • 1 in the ith position if \(x_i\) occurs
    • 0 in the ith position if \(x_i'\) occurs
    • a dash if there is no literal involving \(x_i\) in the product
Quine-McCluskey method: step 3

For our ongoing example, we get:

<table>
<thead>
<tr>
<th>Minterm</th>
<th>Bit string</th>
<th>Combining Term</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>xyz</td>
<td>111</td>
<td>1,2</td>
<td>xz</td>
</tr>
<tr>
<td>xy'z</td>
<td>101</td>
<td>2,4</td>
<td>y'z</td>
</tr>
<tr>
<td>x'yz'</td>
<td>010</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x'y'z</td>
<td>001</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Quine-McCluskey method: step 4

• Determine all products in \( n-2 \) variables that can be formed by taking the Boolean sum of products in \( n-1 \) variables

• Products in \( n-1 \) variables that can be combined have bit strings that:
  – have a dash in exactly the same position
  – differ in exactly one position

• Continue combining Boolean products into products in fewer variables as long as possible

• For our example, we have gone as far as possible in one step
Quine-McCluskey method: step 5

• Find all the Boolean products that arose that were not used to form a Boolean product in one fewer literal
• In our example, there is one of these: $x'y'z'$, bit string 010, as well as the two candidate products represented by strings 1-1 and -01
Quine-McCluskey method: step 6

• Now we form a table showing which terms are covered by which products; there is a row for each candidate product, and a column for each original term; a table for the example is shown on the next slide

• Every minterm must be covered by at least one product

• Each essential prime implicant must be included; once these are found, we can simplify the table by eliminating the columns for minterms covered by this prime implicant

• We continue to identify essential prime implicants and eliminate redundant ones until we reach a point where the table does not change; then a backtracking procedure is used to find the optimal solution
Quine-McCluskey method: step 6

<table>
<thead>
<tr>
<th></th>
<th>xyz</th>
<th>xyz'</th>
<th>x'yz'</th>
<th>x'y'z</th>
</tr>
</thead>
<tbody>
<tr>
<td>x'yz'</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xy</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>y'z</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

We place an X in a position if the original term in the sum-of-products expansion was used to form the candidate product.

When there is only one X in a column, the product corresponding to the row this X is in must be used.

Thus, for this example all of the candidate products must be used, and the simplified sum-of-products is:

\[ x'y'z' + xy + y'z \]