Karnaugh map

- Karnaugh map (K-map) allows viewing the function in a picture form
- Containing the same information as a truth table
- But terms are arranged such that two neighbors differ in only one variable
- It is easy to identify which terms can be combined
- Example: A map with 3 variables

Example: 1, 2, 3, and 4 variables maps are shown below

Farmer’s example and truth tables
Farmer's example truth tables and K-maps

- With three variables, we do not want W and G or G and C to be equal (both 0 or both 1) at the same time.
- With four variables, it is not a problem if F is also the same.

<table>
<thead>
<tr>
<th>W</th>
<th>G</th>
<th>C</th>
<th>A1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

K-map for 5-variables functions

F(A,B,C,D,E) = \( \sum \{2,5,7,8,10,13,15,17,19,21,23,24,29,31\} \)

\[ F(A,B,C,D,E) = CE + AB'E + BC'D'E' + A'C'DE' \]

K-map for 6-variable functions

\[ G(A,B,C,D,E,F) = \sum \{2,8,10,18,24,26,34,37,42,45,50,53,58,61\} \]

\[ G(A,B,C,D,E,F) = D'EF' + ADE'F + A'CD'F' \]

K-map Example for Adder functions

\[ S(A,B,C) = \sum m(1,2,4,7) \]

\[ Cout(A,B,C) = \sum m(3,5,6,7) \]

\[ S = A'B'C + A'B'C' + AB'C' + ABC \]

\[ Cout = BC + AC + AB \]

Minimization of POS Forms

\[ f = \Pi M(4, 5, 6) \]

POS minimization of \( f = x_1 x_2 + x_1' x_2' + (x_1 + x_2) \)
Minimization of 4-Var. Function in POS Form

\[ f = \prod M(0, 1, 4, 8, 9, 12, 15) \]

\[ x_1 x_2 x_3 x_4 \]

POS minimization of \( f = \prod M(0, 1, 4, 8, 9, 12, 15) \)

Simplification of ‘g’ in 7-segment display

\[ g = Z'Y'X'W' + ZY'X'W + ZY'XW' + ZYX'W + ZYW + Z'YW' \]

\[ XW \]

Minimization of Product-of-Sums Forms

\[ g = ZY'X'W' + ZYX'W + ZYW + Z'YW' \]

SOP:

\[ XY \]

Minimization of 4-Var. Function in POS Form

\[ f = Z'Y'X'W' + ZY'X'W + ZY'XW' + ZYX'W + ZYW + Z'YW' \]

\[ XY \]

Simplification of ‘a’ in POS Form

\[ a = Z'Y'X'W' + ZY'X'W + ZY'XW' + ZYX'W + ZYW + Z'YW' \]

\[ XY \]

Simplification of ‘a’ in 7-segment display

\[ a = Z'Y'X'W' + ZY'X'W + ZY'XW' + ZYX'W + ZYW + Z'YW' \]

\[ XY \]
Don’t care condition is input combination that will never occur.
So the corresponding output can either be 0 or 1.
This can be used to help simplifying logic functions.
Example: \( F(A,B,C,D) = \sum m(1,3,7,11,15) + \sum D(0,2,5) \)

K-map with Don’t Care Conditions

Examples

- Simplify the following function considering:
  - the sum-of-products form
  - the product-of-sums form

\[ \begin{array}{c|cccc} \hline AB & 00 & 01 & 10 & 11 \\ \hline CD & & & & \\ \hline 00 & d & I & I & d \\ 01 & 0 & d & I & 0 \\ 11 & 0 & 0 & 1 & 0 \\ 10 & 0 & 0 & 1 & 0 \\ \hline \end{array} \]

\[ F = CD + A'B' \]

1-bit building blocks to make n-bit circuit

- Design a 1-bit circuit with proper “glue logic” to use it for n-bits
  - It is called a bit slice
  - The basic idea of bit slicing is to design a 1-bit circuit and then piece together n of these to get an n-bit component

- Example:
  - A half-adder adds two 1-bit inputs
  - Two half adders can be used to add 3 bits
  - A 3-bit adder is a full adder
  - A full adder can be a bit slice to construct an n-bit adder

\[ \begin{array}{c|c|c|c} \hline A & B & S \\ \hline 0 & 0 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ \hline \end{array} \]

\[ \begin{array}{c|c|c} \hline A & B & w \\ \hline 0 & 0 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 1 & 1 \\ \hline \end{array} \]

Multiple Function Unit Design

- Design a unit that can do more than one function
- In that case, we can design a function unit for each operation like ADD, SUB, AND, OR, ....
- And then select the desired output
- For example, if we want to be able to perform ADD and SUB on two given operands A and B, and select any one
- Then the following set up will work

Design of a SUB Unit

- An n-bit subtract unit can be designed in the same way as an n-bit adder
- One bit subtract unit: It has two inputs A and B (B is subtracted from A) and a borrow (w)
- Outputs: R (primary), W borrow (cascading)
- Borrow from one stage is fed to the next stage
- Truth table is shown
- An n-bit subtract circuit is shown

Full adder & multi-bit ripple-carry adder

- Two half adders can be used to add 3 bits
- n-bit adder can be built by full adders
- n can be arbitrary large

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>w (in)</th>
<th>R</th>
<th>w (out)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
ADD/SUB unit design – First Idea

- Separate ADD and SUB units are expensive
- We can simplify the design
- A - B is the same as adding negation of B to A
- How to negate?
  - 2’s complement (i.e., take 1’s complement and add 1)
  - Adding 1 is also expensive
  - It needs an n-bit adder in general
  - However, we only need to add two bits in each stage
    - In the first stage, we need to add 1’s complement of LSB and 1
    - In other stages, we need to add carry output of previous bit to 1’s complement of current bit
- We select B or negation of B depending on the requirement
- We add A to the selected input to obtain the result

ADD/SUB unit design – Better Idea

- 2’s complement generation of B is expensive
- We can take 1’s complement of B and add it to A
- Then we need to add 1 to the result
- This can be done by setting input carry=1 to n-bit adder
- For add, we simply add B to A with input carry = 0
- Selection signal ADD/SUB = 0 for ADD and 1 for SUB
- The block diagram is shown below
- MUX and inverter can be replaced by XOR (B, ADD/SUB)

One-Bit Adder

- Takes three input bits and generates two output bits
- Multiple bits can be cascaded

ADD/SUB unit design – Better Idea Example

Example: Find A-B
A=0101, B=0110
1’s comp. of B
(i.e., neg (B))
= 1001
2’s comp. of B
= 1010

ADD/SUB unit design – Better Idea Operation

Example: Find A-B
A=0101, B=0110
1’s comp. of B
(i.e., neg (B))
= 1001
2’s comp. of B
= 1010

Adder Boolean Algebra

- A B CI CO S
- 0 0 0 0 0
- 0 0 1 0 1
- 0 1 0 0 1 C = A.B + A.CI+ B.CI
- 0 1 1 1 0
- 1 0 0 0 1 S = A.B.CI + A'.B'.CI+A'.B.CI'+A.B'.CI'
- 1 0 1 1 0
- 1 1 0 1 0
- 1 1 1 1 1
Detecting Overflow

- No overflow when adding a positive and a negative number
- No overflow when signs are the same for subtraction
- Overflow occurs when the value affects the sign:
  - overflow when adding two positives yields a negative
  - or, adding two negatives gives a positive
  - or, subtract a negative from a positive and get a negative
  - or, subtract a positive from a negative and get a positive
- Consider the operations $A + B$, and $A - B$
  - Can overflow occur if $B$ is 0?
  - Can overflow occur if $A$ is 0?

Problem: ripple carry adder is slow

- Is a 32-bit ALU as fast as a 1-bit ALU?
- Is there more than one way to do addition?
  - two extremes: ripple carry and sum-of-products

Carry-look-ahead adder

- An approach in-between our two extremes
- Motivation:
  - If we didn’t know the value of carry-in, what could we do?
  - When would we always generate a carry? $g_i = a_i + b_i$
  - When would we propagate the carry? $p_i = a_i + b_i$
- Did we get rid of the ripple?

\[
\begin{align*}
    c_1 &= g_0 + p_0c_0 \\
    c_2 &= g_1 + p_1c_1 \\
    c_3 &= g_2 + p_2c_2 \\
    c_4 &= g_3 + p_3c_3 \\
    c_5 &= g_4 + p_4c_4 \\
    c_6 &= g_5 + p_5c_5 \\
    c_7 &= g_6 + p_6c_6 \\
    c_8 &= g_7 + p_7c_7 \\
    c_9 &= g_8 + p_8c_8 \\
    c_{10} &= g_9 + p_9c_9 \\
    c_{11} &= g_{10} + p_{10}c_{10} \\
    c_{12} &= g_{11} + p_{11}c_{11} \\
    c_{13} &= g_{12} + p_{12}c_{12} \\
    c_{14} &= g_{13} + p_{13}c_{13} \\
    c_{15} &= g_{14} + p_{14}c_{14} \\
    c_{16} &= g_{15} + p_{15}c_{15}
\end{align*}
\]

Feasible! Why?

A 4-bit carry look-ahead adder

- Generate $g$ and $p$ term for each bit
- Use $g$’s, $p$’s and carry in to generate all $C$’s
- Also use them to generate block $G$ and $P$
- CLA principle can be used recursively

Use principle to build bigger adders

- A 16-bit adder uses four 4-bit adders
- It takes block $g$ and $p$ terms and cin to generate block carry bits out
- Block carries are used to generate bit carries
  - could use ripple carry of 4-bit CLA adders
  - Better: use the CLA principle again!

Delays in carry look-ahead adders

- 4-Bit case
  - Generation of $g$ and $p$: 1 gate delay
  - Generation of carries (and $G$ and $P$): 2 more gate delay
  - Generation of sum: 1 more gate delay
- 16-Bit case
  - Generation of $g$ and $p$: 1 gate delay
  - Generation of block $G$ and $P$: 2 more gate delay
  - Generation of block carries: 2 more gate delay
  - Generation of bit carries: 2 more gate delay
  - Generation of sum: 1 more gate delay
- 64-Bit case
  - 12 gate delays