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

AB \ C	00	01	11	10
0	1	0	1	1
1	1	1	1	0

A	B	C	F
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

1

Location of Min-terms in K-maps

x_1	x_2	x_3	
0	0	0	m_0
0	0	1	m_1
0	1	0	m_2
0	1	1	m_3
1	0	0	m_4
1	0	1	m_5
1	1	0	m_6
1	1	1	m_7

(a) Truth table

x_3 \ x_1x_2	00	01	11	10
0	m_0	m_2	m_6	m_4
1	m_1	m_3	m_7	m_5

(b) Karnaugh map

$\mathbf{m_2 + m_6 = x_1' x_2 x_3' + x_1 x_2 x_3'}$
$\mathbf{= x_2 x_3'}$

2

Simplification using K-map

- Groups of '1's of size 1x1, 2x1, 1x2, 2x2, 4x1, 1x4, 4x2, 2x4, or 4x4 are called prime implicants (p.159 in textbook).

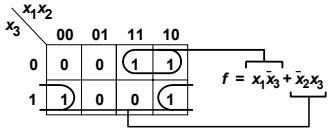
AB \ C	00	01	11	10
0	1	0	1	1
1	1	1	1	0

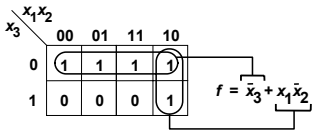
AB \ C	00	01	11	10
0	1	0	1	1
1	1	1	1	0

- A '1' in the K-map can be used by more than one group
- Some rule-of-thumb in selecting groups:
  - Try to use as few group as possible to cover all '1's.
  - For each group, try to make it as large as you can (i.e., if you can use a 2x2, don't use a 2x1 even if that is enough).

3

Examples of 3-Variable K-map





4

Karnaugh maps with up to 4 variables

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

A	0	1
0	0	
1	1	

B \ A	0	1
0	1	0
1	0	1

C \ AB	00	01	11	10
0	1	1	1	1
1	1	0	1	0

CD \ AB	00	01	11	10
00	1	1	1	1
01	1	0	1	0
11	0	1	1	0
10	0	1	1	0

- What if a function has 5 variables?

5

Farmer's example and truth tables

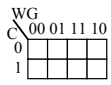
6

Farmer's example truth tables and K-maps

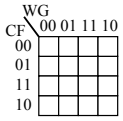
- With three variables, we do not want W and G or G and C to be equal (both 0 or both 1) at the same time
- With four variables, it is not a problem if F is also the same

W	G	C	A1
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

A1



A2



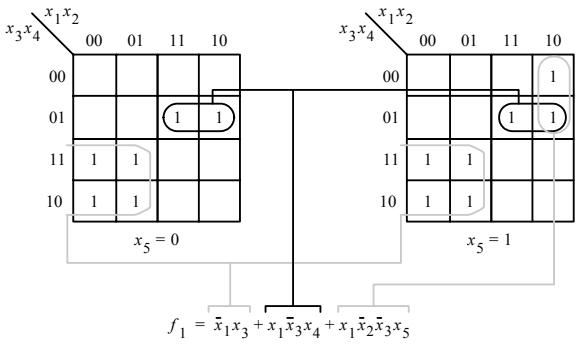
A1 =

A2 =

W	G	C	F	A2
0	0	0	0	0
0	0	0	1	1
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	1
1	1	1	1	0

7

K-map for 5-variables functions

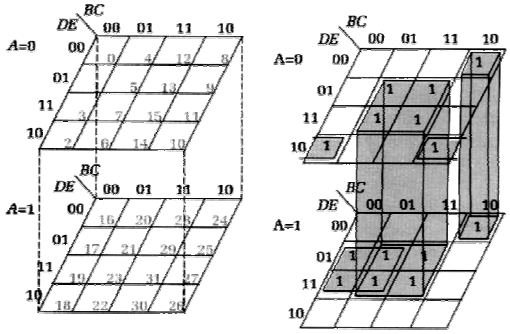


8

K-map for 5-variables functions

F(A,B,C,D,E) = Σ m(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'

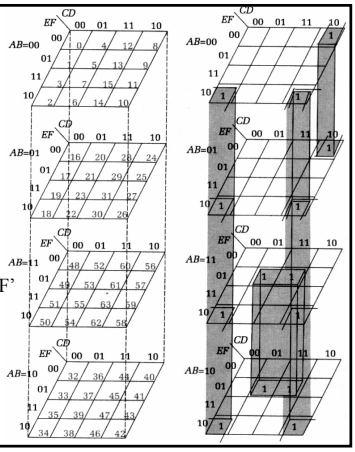


9

K-map for 6-variable functions

G(A,B,C,D,E,F)
= Σ m(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'



Lec 9

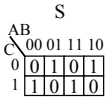
K-map Example for Adder functions

A	B	C	S	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

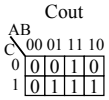
$$S(A,B,C) = \sum m(1,2,4,7)$$

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

S



Cout

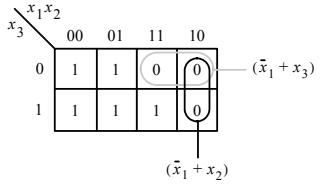


S = A'B'C + A'BC' + AB'C' + ABC

Cout = BC + AC + AB

11

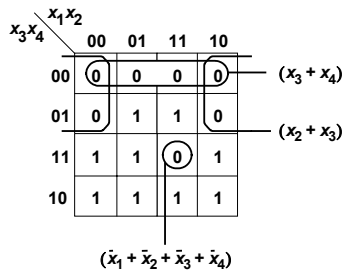
Minimization of POS Forms



POS minimization of $f = \Pi M(4, 5, 6)$

12

Minimization of 4-Var. Function in POS Form



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

13

7-segment display

14

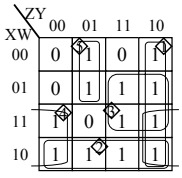
Simplification of 'g' in 7-segment display

g= Z'YX'W' +ZY'X'W'
+Z'YX'W +ZYX'W +ZY'X'W
+ Z'Y'XW +ZYXW +ZY'XW
+ Z'Y'XW' +Z'YXW' +ZYXW' +ZY'XW'

XW \ ZY	00	01	11	10
00	0	1	0	1
01	0	1	1	1
11	1	0	1	1
10	1	1	1	1

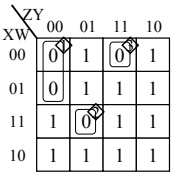
15

Minimization of Product-of-Sums Forms



g=ZY' ①
+XW' ②
+ZW ③
+Y'X ④
+Z'YX' ⑤

Cost = 22
(5 AND gates,
1 OR gates
16 inputs)



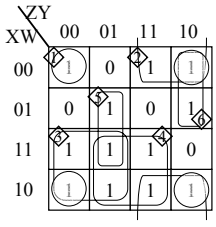
g=(Z+Y+X) ①
.(Z+Y'+X'+W') ②
.(Z'+Y'+X+W) ③

Cost = 18
(3 OR gates,
1 AND gates
14 inputs)

16

Simplification of 'a' in 7-segment display

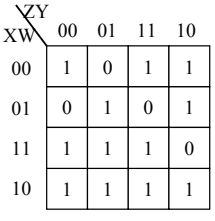
a=Z'Y'X'W' +ZYX'W' +ZY'X'W'
+Z'YX'W +ZY'X'W
+ Z'Y'XW +Z'YXW +ZYXW
+ Z'Y'XW' +Z'YXW' +ZYXW' +ZY'XW'



a=Y'W' ①
+ZW' ②
+Z'X ③
+YX ④
+Z'YW ⑤
+ZY'X' ⑥

17

Simplification of 'a' in POS Form



SOP: a=Y'W'
+ZW'
+Z'X
+YX
+Z'YW
+ZY'X'

Cost =
(AND gates,
OR gates
inputs)

POS:

Cost =
(AND gates,
OR gates
inputs)

18

K-map with Don't Care Conditions

- ***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)=Σ m(1,3,7,11,15)+ Σ D(0,2,5)**

AB \ CD	00	01	11	10
00	d	1	1	d
01	0	d	1	0
11	0	0	1	0
10	0	0	1	0

F = CD+A'B'

AB \ CD	00	01	11	10
00	d	1	1	d
01	0	d	1	0
11	0	0	1	0
10	0	0	1	0

F = CD+A'D

d: Don't Care Condition

19

Examples

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

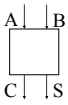
AB \ CD	00	01	11	10
00	1	0	0	1
01	1	1	1	d
11	0	d	1	1
10	0	0	0	1

AB \ CD	00	01	11	10
00	1	0	0	1
01	1	1	1	d
11	0	d	1	1
10	0	0	0	1

20

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**



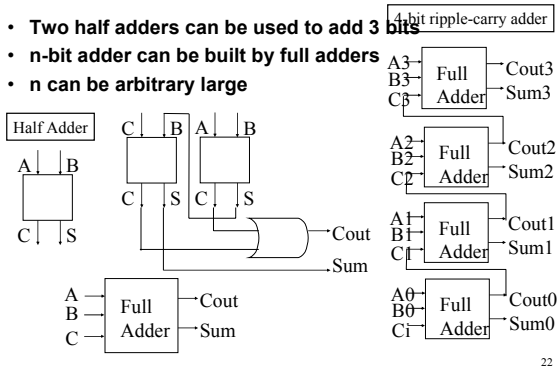
A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Half Adder

21

Full adder & multi-bit ripple-carry adder

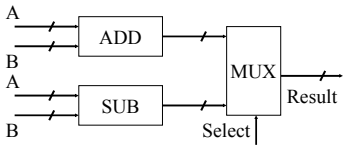
- **Two half adders can be used to add 3 bits**
- **n-bit adder can be built by full adders**
- **n can be arbitrary large**



22

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**

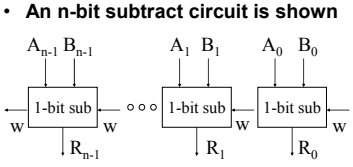


23

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**

A	B	w (in)	R	w(out)
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		



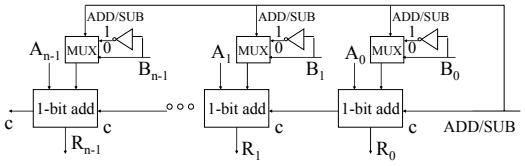
24

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)



ADD/SUB unit design – Better Idea Example

Example: Find A-B
A=0101, B=0110

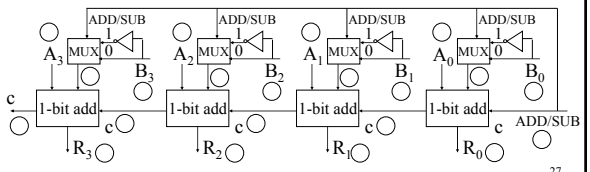
1's comp. of B
(i.e., neg (B))
=
2's comp. of B
= =

First Idea:
A+(comp(B)+1)

0101
+ ____ ←2's

Better Idea:
A+comp(B)+**1**

1
0101
+ ____ ←1's



ADD/SUB unit design – Better Idea Operation

Example: Find A-B
A=0101, B=0110

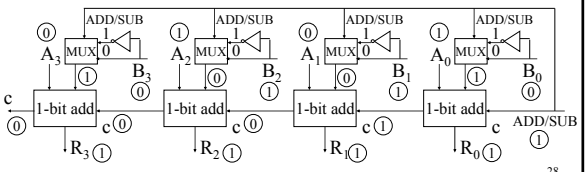
1's comp. of B
(i.e., neg (B))
=1001
2's comp. of B
=1001+1=1010

First Idea:
A+(neg(B)+1)

0000
0101
+ 1010← 2's
1111

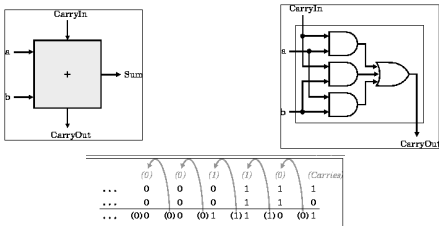
Better Idea:
A+neg(B)+**1**

0001**1**
0101
+ 1001←1's
1111



One-Bit Adder

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



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

Can you see the ripple? How could you get rid of it?

$c_1 = b_0c_0 + a_0c_0 + a_0b_0$
$c_2 = b_1c_1 + a_1c_1 + a_1b_1 \quad c_2 =$
$c_3 = b_2c_2 + a_2c_2 + a_2b_2 \quad c_3 =$
$c_4 = b_3c_3 + a_3c_3 + a_3b_3 \quad c_4 =$

Not feasible! Why?

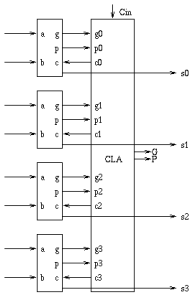
Carry-look-ahead adder

- 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?

$c_1 = g_0 + p_0c_0$
$c_2 = g_1 + p_1c_1 \quad c_2 = g_1 + p_1g_0 + p_1p_0c_0$
$c_3 = g_2 + p_2c_2 \quad c_3 = g_2 + p_2g_1 + p_2p_1g_0 + p_2p_1p_0c_0$
$c_4 = g_3 + p_3c_3 \quad c_4 = g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0 + p_3p_2p_1p_0c_0$

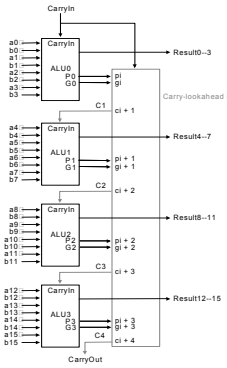
Feasible! Why?

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