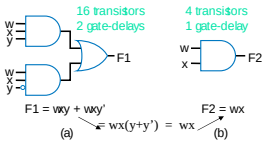
ECE 274 - Digital Logic

Lecture 19

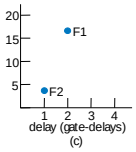
- Lecture 19 – Optimization
  - Two-Level Minimization
    - Karnaugh Maps

Introduction

- We now know how to build digital circuits
  - How can we build ***better*** circuits?
- Let's consider two important design criteria
  - ***Delay*** – the time from inputs changing to new correct stable output
  - ***Size*** – the number of transistors
  - For quick estimation, assume
    - Every gate has delay of "1 gate-delay"
    - Every gate *input* requires 2 transistors
    - Ignore inverters



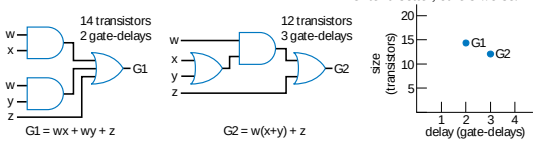
Transforming F1 to F2 represents an ***optimization***: Better in all criteria of interest



Introduction

- Tradeoff
  - Improves some, but worsens other, criteria of interest

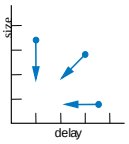
Transforming G1 to G2 represents a ***tradeoff***: Some criteria better, others worse.

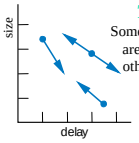


Introduction

Optimizations

All criteria of interest are improved (or at least kept the same)





Tradeoffs

Some criteria of interest are improved, while others are worsened

- We obviously prefer optimizations, but often must accept tradeoffs
  - You can't build a car that is the most comfortable, and has the best fuel efficiency, and is the fastest – you have to give up something to gain other things.

Combinational Logic Optimization and Tradeoffs

- Two-level size optimization using algebraic methods
  - Goal: circuit with only two levels (ORed AND gates), with minimum transistors
    - Though transistors getting cheaper (Moore's Law), they still cost something
- Define problem algebraically
  - Sum-of-products yields two levels
    - F = abc + abc' is sum-of-products; G = w(xy + z) is not.
  - Transform sum-of-products equation to have fewest literals and terms
    - Each literal and term translates to a gate input, each of which translates to about 2 transistors (see Ch. 2)
    - Ignore inverters for simplicity

Example

F = xyz + xyz' + x'y'z' + x'y'z

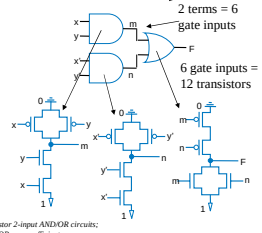
F = xy(z + z') + x'y'(z + z')

F = xy*1 + x'y'*1

F = xy + x'y'

4 literals + 2 terms = 6 gate inputs

6 gate inputs = 12 transistors



Assuming 4-transistor 2-input AND/OR circuits; in reality, only NAND/NOR are so efficient.

Algebraic Two-Level Size Minimization

- Previous example showed common algebraic minimization method
  - (Multiply out to sum-of-products, then)
  - Apply following as much possible
    - ab + ab' = a(b + b') = a*1 = a
    - "Combining terms to eliminate a variable"
      - (Formally called the "Uniting theorem")
  - Duplicating a term sometimes helps
    - Note that doesn't change function
      - c + d = c + d + d = c + d + d + d + d
      - ...
  - Sometimes after combining terms, can combine resulting terms

F = xyz + xyz' + x'y'z' + x'y'z

F = xy(z + z') + x'y'(z + z')

F = xy*1 + x'y'*1

F = xy + x'y'

F = x'y'z' + x'y'z + x'yz

F = x'y'z' + x'y'z + x'y'z + x'yz

F = x'y'(z+z') + x'z(y'+y)

F = x'y' + x'z

G = xy'z' + xy'z + xyz + xyz'

G = xy'(z' +z) + xy(z+z')

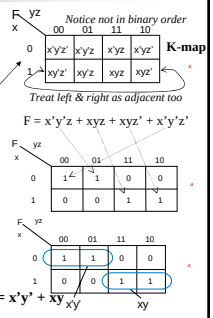
G = xy' + xy *(now do again)*

G = x(y'+y)

G = x

Karnaugh Maps for Two-Level Size Minimization

- Easy to miss "seeing" possible opportunities to combine terms
- ***Karnaugh Maps (K-maps)***
  - **Graphical** method to help us find opportunities to combine terms
  - Minterms differing in one variable are *adjacent* in the map
  - Can clearly see opportunities to combine terms – look for adjacent 1s
    - For F, clearly two opportunities
    - Top left circle is *shorthand* for x'y'z'+x'y'z = x'y'(z'+z) = x'y'(1) = x'y'
    - Draw circle, write term that has all the literals except the one that changes in the circle
      - Circle xy, x=1 & y=1 in both cells of the circle, but z changes (z=1 in one cell, 0 in the other)
    - Minimized function: OR the final terms



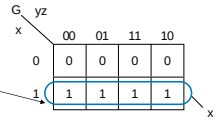
Easier than all that algebra:

F = xyz + xyz' + x'y'z' + x'y'z
F = xy(z + z') + x'y'(z + z')
F = xy*1 + x'y'*1
F = xy + x'y'

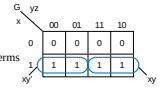
K-maps

- Four adjacent 1s means two variables can be eliminated
  - Makes intuitive sense – those two variables appear in all combinations, so one *must* be true
  - Draw one big circle – *shorthand* for the algebraic transformations above

G = xy'z' + xy'z + xyz + xyz'
G = x(y'z'+ y'z + yz + yz') (must be true)
G = x(y'(z'+z) + y(z+z'))
G = x(y'+y)
G = x

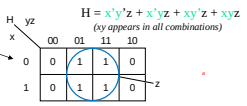


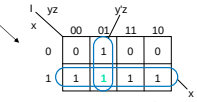
Draw the biggest circle possible, or you'll have more terms than really needed

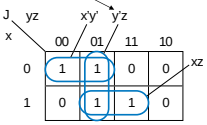


K-maps

- Four adjacent cells can be in shape of a square
- *OK* to cover a 1 twice
  - Just like duplicating a term
    - Remember, c + d = c + d + d
- No *need* to cover 1s more than once
  - Yields extra terms – not minimized



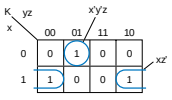


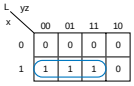


The two circles are shorthand for:
I = x'y'z + xy'z' + xy'z + xyz + xyz'
I = x'y'z + xy'z + xy'z' + xy'z + xyz + xyz'
I = (x'y'z + xy'z) + (xy'z' + xy'z + xyz + xyz')
I = (y'z) + (x)

K-maps

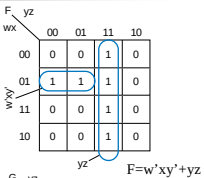
- Circles can cross left/right sides
  - Remember, edges are adjacent
    - Minterms differ in one variable only
- Circles must have 1, 2, 4, or 8 cells – 3, 5, or 7 not allowed
  - 3/5/7 doesn't correspond to algebraic transformations that combine terms to eliminate a variable
- Circling all the cells is OK
  - Function just equals 1



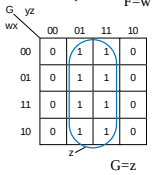


K-maps for Four Variables

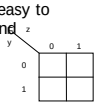
- Four-variable K-map follows same principle
  - Adjacent cells differ in one variable
  - Left/right adjacent
  - Top/bottom also adjacent
- 5 and 6 variable maps exist
  - But hard to use
- Two-variable maps exist
  - But not very useful – easy to do algebraically by hand



F=w'xy'+yz



G=z



Two-Level Size Minimization Using K-maps

General K-map method

1. Convert the function's equation into sum-of-products form
2. Place 1s in the appropriate K-map cells for each term
3. Cover all 1s by drawing the fewest largest circles, with every 1 included at least once; write the corresponding term for each circle
4. OR all the resulting terms to create the minimized function.

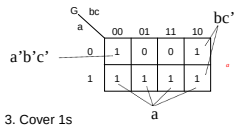
Example: Minimize:

G = a + a'b'c' + b*(c' + bc')

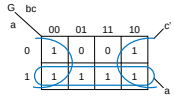
1. Convert to sum-of-products

G = a + a'b'c' + bc' + bc'

2. Place 1s in appropriate cells



3. Cover 1s

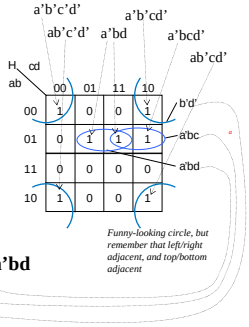


4. OR terms: **G = a + c'**

Two Level Size Minimization Using K-maps
– Four Variable Example

- Minimize:
  - H = a'b'(cd' + c'd') + ab'c'd' + ab'cd' + a'bd + a'bcd'

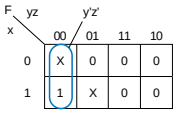
1. Convert to sum-of-products:
   - H = a'b'cd' + a'b'c'd' + ab'c'd' + ab'cd' + a'bd + a'bcd'
2. Place 1s in K-map cells
3. Cover 1s
4. OR resulting terms



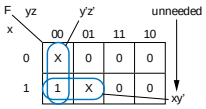
H = b'd' + a'bc + a'bd

Don't Care Input Combinations

- What if particular input combinations can never occur?
  - e.g., Minimize F = xy'z', given that x'y'z' (xyz=000) can *never* be true, and that xy'z (xyz=101) can *never* be true
  - So it doesn't matter what F outputs when x'y'z' or xy'z is true, because those cases *will never occur*
  - Thus, make F be 1 or 0 for those cases *in a way that best minimizes the equation*
- On K-map
  - Draw Xs for don't care combinations
    - Include X in circle ONLY if minimizes equation
    - Don't include other Xs



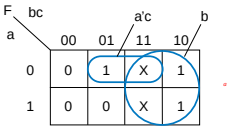
Good use of don't cares



Unnecessary use of don't cares; results in extra term

Minimization Example using Don't Cares

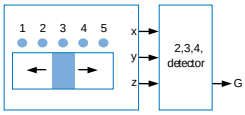
- Minimize:
  - F = a'bc' + abc' + a'b'c
  - Given don't cares: a'bc, abc
- Note: Use don't cares with caution
  - Must be *sure* that we really don't care what the function outputs for that input combination
  - If we do care, even the slightest, then it's probably safer to set the output to 0



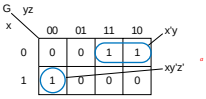
F = a'c + b

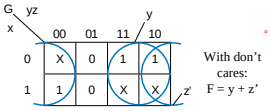
Minimization with Don't Cares Example: Sliding Switch

- Switch with 5 positions
  - 3-bit value gives position in binary
- Want circuit that
  - Outputs 1 when switch is in position 2, 3, or 4
  - Outputs 0 when switch is in position 1 or 5
  - Note that the 3-bit input can never output binary 0, 6, or 7
    - Treat as don't care input combinations



Without don't cares: F = x'y + xy'z'





With don't cares: F = y + z'

Design Challenge

- Design Challenge
  - Perform two-level logic size optimization for the function *F(a,b,c) = ab'c + abc + a'bc + abc'* using (a) algebraic methods, (b) a K-map. Express the answers as sum-of-products.
  - Due:
    - Next Lecture (Wednesday, November 16)
  - Extra Credit *(Homework)*
    - 2 points