Optimization and Tradeoffs

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

Transforming G1 to G2 represents a tradeoff: Some criteria better, others worse.
Optimization and Tradeoffs

Introduction

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.

Optimizations

All criteria of interest are improved, while others are worsened

Tradeoffs

Some criteria of interest are improved, while others are worsened

Optimization and Tradeoffs

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

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

\[ F = xyz + xyz' + x'y'z' + x'y'z \]

\[ F = xy(z + z') + x'y'(z + z') \]

\[ F = xy*1 + x'y'*1 \]

\[ F = xy + x'y' \]

Optimization and Tradeoffs

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
    - \( c + d + c + d + d + d + d + d \)

- Sometimes after combining terms, can combine resulting terms

For \( F \), clearly two opportunities

\[ F = x'y'z + xyz + xyz' + x'y'z' \]

\[ F = xy(z + z') + x'y'(z + z') \]

\[ F = xy*1 + x'y'*1 \]

\[ F = xy + x'y' \]

Optimization and Tradeoffs

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

\[ F = x'y'z + xyz + xyz' + x'y'z' \]

\[ F = xy(z + z') + x'y'(z + z') \]

\[ F = xy*1 + x'y'*1 \]

\[ F = xy + x'y' \]

Optimization and Tradeoffs

Karnaugh Maps for Two-Level Size Minimization

K-map

Notice not in binary order

For \( F \), clearly two opportunities

Top left circle is shorthand for \( x'y' + x'y = x(y + z) = x' + x \)

Draw circle, write term that has all the literals except the one that changes in the circle

Circle \( xy \), \( x = 1 \) in both cells of the circle, but \( y \) changes \( z = 3 \) in one cell, \( 0 \) in the other

Minimized function: OR the final terms

\[ F = x'y' + xy \]

Easier than all that algebra.
Optimization and Tradeoffs
Karnaugh Maps for Two-Level Size Minimization

- 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

\[ G = xy'z' + xyz + xz + xz' \]
\[ G = x(y'z' + yz + z + z') \]
\[ G = x(y'z' + yz + z + z') \]
\[ G = x(y'z' + yz + z + z') \]
\[ G = x(y'z' + yz + z + z') \]

- Circles can cross left/right sides
  - Remember, edges are adjacent
  - 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

- 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

\[ H = x'y'z + xy'z + xz' + xz \]
\[ (xy) \text{ appears in all combinations} \]
\[ I = x'y'z + xy'z + xz' + xz \]
\[ I = x'y'z + xy'z + xz' + xz \]
\[ I = x'y'z + xy'z + xz' + xz \]
\[ I = x'y'z + xy'z + xz' + xz \]
\[ I = x'y'z + xy'z + xz' + xz \]
\[ I = x'y'z + xy'z + xz' + xz \]
\[ I = x'y'z + xy'z + xz' + xz \]
\[ I = x'y'z + xy'z + xz' + xz \]
\[ I = x'y'z + xy'z + xz' + xz \]
Optimization and Tradeoffs
Karnaugh Maps for Two-Level Size Minimization

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:
\[ G = a + a'b'c' + b\cdot c' + bc' \]
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: \[ G = a + c' \]

Optimization and Tradeoffs
Karnaugh Maps: Don’t Care Input Combinations

- What if particular input combinations can never occur?
  - e.g., Minimize \( F = xy'z' \), given that \( xy'z' \) (\( xyz=000 \)) can never be true, and that \( x'y'z \) (\( xyz=101 \)) can never be true
  - So it doesn’t matter what \( F \) outputs when \( xy'z \) or \( x'y'z \) 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

Example: Minimize:
\[ F = a'b'c' + abc' + a'b'c \]
- Given don’t cares: \( a'b'c, abc \)

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
Optimization and Tradeoffs
Karnaugh Maps: Don’t Care Input Combinations

Example:
- 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

<table>
<thead>
<tr>
<th>2,3,4, detector</th>
<th>2,3,4, detector</th>
</tr>
</thead>
<tbody>
<tr>
<td>G</td>
<td>G</td>
</tr>
</tbody>
</table>

With don’t cares:
- F = y + z'

Optimization and Tradeoffs
Automating Two-Level Logic Size Minimization

Minimizing by hand
- Is hard for functions with 5 or more variables
- May not yield minimum cover depending on order we choose
- Is error prone

Minimization thus typically done by automated tools
- Exact algorithm: finds optimal solution
- Heuristic: finds good solution, but not necessarily optimal

Optimization and Tradeoffs
Basic Concepts Underlying Automated Two-Level Logic Minimization

Definitions
- On-set: All minterms that define when F=1
- Off-set: All minterms that define when F=0
- Implicant: Any product term (minterm or other) that when 1 causes F=1
  - On K-map, any legal (but not necessarily largest) circle
  - Cover: Implicant xy covers minterms xyz and xyz'
- Expanding a term: removing a variable (like larger K-map circle)
  - xyz → xy is an expansion of xyz

Definitions (cont)
- Essential prime implicant: The only prime implicant that covers a particular minterm in a function’s on-set
  - Importance: We must include all essential PIs in a function’s cover
  - In contrast, some, but not all, non-essential PIs will be included

Note: We use K-maps here just for intuitive illustration of concepts; automated tools do not use K-maps.
Optimization and Tradeoffs
Automated Two-Level Logic Minimization Method

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Determine prime implicants</td>
</tr>
<tr>
<td>2</td>
<td>Add essential prime implicants to the function’s cover</td>
</tr>
<tr>
<td>3</td>
<td>Cover remaining minterms with non-essential prime implicants</td>
</tr>
</tbody>
</table>

- Steps 1 and 2: Exact

Optimization and Tradeoffs
Example of Automated Two-Level Minimization

1. Determine all prime implicants
2. Add essential Pls to cover
   - Italicized 1s are thus already covered
   - Only one uncovered 1 remains
3. Cover remaining minterms with nonessential Pls
   - Pick among the two possible Pls

Optimization and Tradeoffs
Problem with Methods that Enumerate all Minterms or Compute all Prime Implicants

- Too many minterms for functions with many variables
  - Function with 32 variables:
    - $2^{32} = 4$ billion possible minterms.
    - Too much compute time/memory
  - Too many computations to generate all prime implicants
    - Comparing every minterm with every other minterm, for 32 variables, is ($4$ billion)$^2 = 1$ quadrillion computations
    - Functions with many variables could requires days, months, years, or more of computation - unreasonable

Optimization and Tradeoffs
Solution to Computation Problem

- Solution
  - Don’t generate all minterms or prime implicants
  - Instead, just take input equation, and try to “iteratively” improve it
  - Ex: $F = abcd + abcd + jkln$  
    - Note: 15 variables, may have thousands of minterms
    - But can minimize just by combining first two terms:  
      - $F = abcd + jkln = abcd + jkln$
Optimization and Tradeoffs
Two-Level Minimization using Iterative Method

- Method: Randomly apply “expand” operations, see if helps
  - Expand: remove a variable from a term
    - Like expanding circle size on K-map
    - e.g., Expanding $x'z$ to $z$ legal, but expanding $x'y'$ to $y'$ not legal, in shown function
    - After expand, remove other terms covered by newly expanded term
  - Keep trying (iterate) until doesn’t help

Ex:

\[ F = abdefgh + abdefgh' + jklmnop \]
\[ F = abdefg + abdefgh' + jklmnop \]
\[ F = abdefg + jklmnop \]

Optimization and Tradeoffs
Multi-Level Logic Optimization – Performance/Size Tradeoffs

- We don’t always need the speed of two level logic
- Multiple levels may yield fewer gates
  - Example
    - \[ F_1 = ab + acd + ace \]
    - Solution
      - \[ F_2 = ab + ac(d + e) = a(b + c(d + e)) \]
    - General technique: Factor out literals – \[ xy + xz = x(y+z) \]

\[ F_1 = ab + acd + ace \]
\[ F_2 = a(b+c(d+e)) \]
22 transistors
2 gate delays
18 transistors
3 gate delays

Optimization and Tradeoffs
Multi-Level Example: Non-Critical Path

- Critical path: longest delay path to output
- Optimization: reduce size of logic on non-critical paths by using multiple levels

\[ F_1 = (a+b)c + defg + efg \]
\[ F_2 = (a+b)c + (d+e)fg \]