Technology Independent Logic Optimization

Prof. Kurt Keutzer
EECS
University of California
Berkeley, CA

Thanks to R. Rudell, S. Malik

Multi-level logic optimization

• Context
• Motivation
• What to do?
• How to do it!
RTL Design Flow

Logic Optimization

Perform a variety of transformations and optimizations
- Structural graph transformations
- Boolean transformations
- Mapping into a physical library

smaller, faster
less power
Logic Optimization

Reduce to Combinational Optimization
**Representation: Boolean Network**

A Boolean network is designated $\eta=(\hat{y}, H)$ where:
- $\hat{y} = (y_1, K, y_{n+m+r})$ is a vector of variables
- $H = (H_1, K, H_{n+m+r})$ is a vector of functions

$y_1, K, y_n$ are the primary input variables
$y_{n+1}, K, y_{n+r}$ are the intermediate variables
$y_{n+m+1}, K y_{n+m+r}$ are the primary output variables

$y_i = H_i(y_1, K, y_{n+m+r})$

A Boolean network has an associated graph which shows the function dependencies; i.e., the edge $(i, j)$ is present if $y_i \in \text{sup}(H_j)$.

**Combinational Logic Optimization**

**Input:**
- Initial Boolean network
- Timing characterization for the module
  - input arrival times and drive factors
  - output loading factors
- Optimization goals
  - output required times
- Target library description

**Output:**
- Minimum-area net-list of library gates which meets timing constraints
  
  *A very difficult optimization problem!*
Modern Approach to Logic Optimization

Divide logic optimization into two subproblems:

- Technology-independent optimization
  - determine overall logic structure
  - estimate costs (mostly) independent of technology
  - simplified cost modeling
- Technology-dependent optimization (technology mapping)
  - binding onto the gates in the library
  - detailed technology-specific cost model

Orchestration of various optimization/ transformation techniques for each subproblem

Formats

Boolean Network, Boolean equations

Generic library – technology independent

- Has standard functions – ND2, ND4, AOI22, pos-edge-FF
- only an estimate of timing

Actual technology library

- Represents logic functions and their physical characteristics of a library cells offered by a particular silicon vendor – e.g. TSMC
- E.g. captured in .lib file
- Contains complete logical, timing information
2-Level => Implementation

- We know that any logic function can be implemented in Truth-Table form, so can we build a general circuit structure that can implement a simplified ROM form directly?
- How about an even more compact form?
2-Level => Programmable Logic Arrays (PLAs)

- We know that any logic function can be implemented in Truth-Table form, so can we build a general circuit structure that can implement a simplified ROM form directly?

4 inputs

3 outputs

6 product terms

2-Level => Programmable Logic Arrays (PLAs)

- Does this representation have any limitations?

4 inputs

3 outputs

6 product terms
Multi-level logic optimization

- Context
- Motivation
- What to do?
- How to do it!

Multilevel Combinational Circuits

Motivation: There are many functions that are too “expensive” to implement in two-level form

Try 16-bit adder \( \Rightarrow \) 32 input lines and \( 2^{16} \) product terms!
Two-Level versus Multilevel

2-Level:

\[ f_1 = AB + AC + AD \]
\[ f_2 = \overline{AB} + \overline{AC} + \overline{A}E \]

6 product terms which cannot be shared.
24 transistors in static CMOS

Multi-level:

Note that \( B + C \) is a common term in \( f_1 \) and \( f_2 \)

\[ K = B + C \]
\[ f_1 = AK + AD \]
\[ f_2 = \overline{AK} + \overline{A}E \]

3 Levels
20 transistors in static CMOS
not counting inverters

Tech.-Independent Optimization

- Involves:
  - Minimizing two-level logic functions.
  - Finding common subexpressions.
  - Substituting one expression into another.
  - Factoring single functions.

Factored versus Disjunctive forms

\[ f = ac + ad + bc + bd + a\overline{e} \]

sum-of-products or disjunctive form

\[ f = (a + b)(c + d) + a\overline{e} \]

factored form

Naturally leads to implementation in multi-level or complex gates
Decomposition

G is a divisor of F if \( F = G \times H + R \) and \( H \) is non-empty

Searching for divisors which are common to many functions in the network

Decomposition:
- identify divisors which are common to several functions
- introduce common divisor as a new node
- re-express existing nodes using the new divisor

Technology-independent measure of cost to measure goodness
- area cost: total number of literals
- delay cost: levels of logic on the critical path

Algebraic vs. Boolean Methods

Algebraic techniques view equations as polynomials and attempt to factor equations or “divide” them

- Do not exploit Boolean identities e.g., \( a \overline{a} = 0 \)

In algebraic substitution (or division) if a function \( f = f(a, b, c) \) is divided by \( g = g(a, b) \), \( a \) and \( b \) will not appear in \( f / g \)

- Algebraic division: \( O(n \log n) \) time
- Boolean division: unmanageable number of divisors
Comparison of factorization

\[ f = a \bar{b} + a \bar{c} + b \bar{a} + b \bar{c} + c \bar{a} + c \bar{b} \]

Algebraic factorization procedures

\[ f = a(\bar{b} + \bar{c}) + \bar{a}(b + c) + b \bar{c} + c \bar{b} \]

Boolean factorization produces

\[ f = (a + b + c)(\bar{a} + \bar{b} + \bar{c}) \]

Comparison

Substitution is the factoring of one node in the Boolean Network (e.g. \( l \)) by another (e.g. \( r \))

Algebraic substitution of \( l \) into \( r \) fails

Boolean substitution yields results

\[ l = (b \bar{f} + \bar{b} f)(a + e) + \bar{a} e (b \bar{f} + bf) \]
\[ r = (b \bar{f} + \bar{b} f)(\bar{a} + \bar{e}) + a e (b \bar{f} + bf) \]

After resub:

\[ r = a(\bar{e} \bar{l} + el) + \bar{a}(\bar{e}l + e \bar{l}) \]
\[ l = a(e r + \bar{e} \bar{r}) + \bar{a}(\bar{e} r + e \bar{r}) \]
Algebraic Decomposition

Algebraic approximation (informal definition)
- simplify Boolean function using two-level minimization
- treat result as a polynomial; i.e.,
  \[ x_i \text{ and } \bar{x}_i \text{ are different variables} \]
  \[ \text{i.e., } x_i \cdot \bar{x}_i \neq 0 \text{ and } x_i \cdot x_i \neq x_i \]
- identify common divisors as algebraic divisors of the polynomials

Motivation
- manipulating polynomials is fast (linear time algorithms)
- # algebraic divisors still exponential, but usually manageable
- loss of optimality, but experimentally shows good results
- interleave Boolean simplification procedures to improve results

Techniques
- single-cube algebraic divisors (common-cube decomposition)
- multiple-cube algebraic divisors (kernel decomposition)

Decomposition?

\[ F = a \, d \, e + b \, d \, e + c \, d \, e + f \]
\[ G = b \, g + c \, g + d \, g + a \, e \, f \]
\[ H = a \, e \, g + b \, c \]

How can this logic be further simplified?
Common Cube Decomposition

Finds algebraic divisors which are single cubes

"Common cubes" are easy to detect

Greedy algorithm:

– enumerate all maximal common cubes
– select cube which saves the most literals
– add node to the network and re-express affected nodes
– repeat until no common cubes remain

References: [Dietmeyer-69], [Brayton-82], [Rudell-89]

\[
\begin{align*}
F &= \text{a} \text{d} \text{e} + \text{b} \text{d} \text{e} + \text{c} \text{d} \text{e} + \text{f} \\
G &= \text{b} \text{g} + \text{c} \text{g} + \text{d} \text{g} + \text{a} \text{e} \text{f} \\
H &= \text{a} \text{e} \text{g} + \text{b} \text{c}
\end{align*}
\]

\[
\begin{align*}
F &= \text{X} \text{d} + \text{b} \text{d} \text{e} + \text{c} \text{d} \text{e} + \text{f} \\
G &= \text{b} \text{g} + \text{c} \text{g} + \text{d} \text{g} + \text{X} \text{f} \\
H &= \text{X} \text{g} + \text{b} \text{c} \\
X &= \text{a} \text{e}
\end{align*}
\]

Common Cube Decomposition

Does reduce logic

Safe, as an optimization (never changes logical value)

Used industrially in early 80’s

Can we do better?
Kernel Decomposition

cube-free: an expression is cube-free if no literal appears in every cube

- $a + b + c$ is a kernel of $F$
- $b + c$ is not a kernel because it is contained in $a + b + c$

Kernel Decomposition

Kernels are useful because:
- if $f$ and $g$ share a common multiple-cube divisor, then the intersection of some kernel from $f$ and some kernel from $g$ yields a common multiple-cube divisor [Brayton-82].
- practical algorithms exist to find intersections of kernels [Rudell-89].
- kernel decomposition finds solutions which are difficult to find using only common-cube decomposition
Finding the right decomposition

Cube and kernel factorizations are heuristics – and depend on the initial "structure" of the Boolean network

We wish to "hill climb" try to find some new structures

How?

Selective Collapsing

"Collapse" nodes into their fanout to increase the size of each node

\[
\begin{align*}
  f &= a h + b h' + c d \\
  g &= b h \\
  h &= a c + d'
\end{align*}
\]

collapse \( h \)

\[
\begin{align*}
  f &= a (a c + d') + b (a' d' + c' d') + c d \\
  g &= b (a c + d')
\end{align*}
\]

simplify

\[
\begin{align*}
  f &= a c + a d' + a' b d' + b c' d' + c d \\
  g &= a b c + b d'
\end{align*}
\]

Goals:
- remove bad initial structure
- reduce logic level depth
- expose further optimization opportunities
Summary of Typical Recipe

Selective-collapse
Simplify: Two-level minimization at Boolean network node
Structuring/Algebraic decomposition
Local Boolean optimizations

Tree-covering for gate selection
Load-buffering for fanout-tree construction
Local transformation improvement of circuit structure

Restructure and iterate if timing constraints not met

Multi-level logic optimization

- Context
- Motivation
- What to do?
- How to do it!
A Boolean network is designated \( \eta = (\vec{y}, H) \) where:
- \( \vec{y} = (y_1, K, y_{n+m+r}) \) is a vector of variables
- \( H = (H_1, K, H_{n+m+r}) \) is a vector of functions

\( y_1, K, y_n \) are the primary input variables
\( y_{n+1}, K, y_{n+r} \) are the intermediate variables
\( y_{n+m+1}, K y_{n+m+r} \) are the primary output variables

\[ y_i = H_i (y_1, K y_n) \]

A Boolean network has an associated graph which shows the function dependencies; i.e., the edge \((i, j)\) is present if \( y_i \in \text{sup}(H_j) \).

---

**Weak (or Algebraic) Division**

**Definition:** The support of \( f \), denoted \( \text{sup}(f) \) = \{ set of all variables \( v \) that occur in \( f \) as \( v \) or \( \overline{v} \) \}

**Example:** \( f = A \overline{B} + \overline{C} \)
\[ \text{sup}(f) = \{ A, B, C \} \]

**Definition:** we say that \( f \) is orthogonal to \( g \), \( f \perp g \) if \( \text{sup}(f) \cap \text{sup}(g) = \emptyset \)

**Example:** \( f = A + B \quad g = C + D \)
\[ \therefore f \perp g \text{ since } \{ A, B \} \cap \{ C, D \} = \emptyset \]
Weak Division - 2

We say that \( g \) divides \( f \) weakly if there exist \( h, r \) such that \( f = gh + r \) where \( h \neq \phi \) and \( g \perp h \)

Example: \( f = ab + ac + d \)
\[
g = b + c
\]
\[
f = a(b + c) + d \quad h = a \quad r = d
\]

We say that \( g \) divides \( f \) evenly if \( r = \phi \)

The quotient \( f/g \) is the largest \( h \) such that \( f = gh + r \)
\[i.e., \quad f = (f/g)g + r\]

Computing \( f/g \)

Given \( f = \{ c_i \}, \ g = \{ a_i \} \) i.e., lists of cubes
\[
h_i = \{ b_j \mid a_i b_j \in f \} \forall i
\]
i.e., all the multipliers of the cube \( a_i \) in \( g \)
that produce elements of \( f \) are in \( h_i \)

Theorem: \( f/g = \bigcap_{i=1}^{\| g \|} h_i = h_1 \cap h_2 \ldots h_{\| g \|} \)
Weak Division Example

\[
f = abc + abde + abh + bcd \\
g = c + de + h
\]

Theorem says \( f / g = f / c \cap f / de \cap f / h \)

\[
f / c = ab + bd \\
f / de = ab \\
f / h = ab
\]

\[
f / g = (ab + bd) \cap ab \cap ab = ab
\]

Re-express \( f = ab(c + de + h) + bcd \)

Time complexity: \( O(|f| |g|) \)

Types of Algebraic Divisors

Define divisors of \( f \) as the set
\[
D(f) = \{ g \mid f / g \neq \phi \}
\]

Define primary divisors of \( f \) as
\[
P(f) = \{ f / c \mid c \text{ is a cube} \}
\]

Example: \( f = abc + abde \)
\[
f / a = bc + bde \quad \text{is a primary divisor}
\]

Every divisor of \( f \) is contained in a primary divisor. If \( g \) divides \( f \), then \( g \subseteq p \in P(f) \)

\( g \) is termed “cube-free” if the only cube dividing \( g \) evenly is 1.
Kernels and Divisors

Define the kernels of $f$ as
$$K(f) = \{ k \mid k \in P(f), \text{ k is cube-free} \}$$

Example: $f = abc + abde$
$$f/a = bc + bde \text{ is a primary divisor}$$
but is not cube-free since $b$ is a factor
$$f/a = b(c + de)$$
$$f/ab = c + de \text{ is a kernel}$$
$ab$ is the co-kernel

The co-kernel of a kernel is not unique.

Examples

Consider
$$f = acd + bcd + ae + be$$

**DIVISOR TYPE**

<table>
<thead>
<tr>
<th>DIVISOR TYPE</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f/a$</td>
<td>$cd + e$</td>
</tr>
<tr>
<td>$f/c$</td>
<td>$ad + bd$</td>
</tr>
<tr>
<td>$f/cd$</td>
<td>$a + b$</td>
</tr>
<tr>
<td>$f/e$</td>
<td>$a + b$</td>
</tr>
</tbody>
</table>
Examples

Consider
\[ f = acd + bcd + ae + be \]

**DIVISOR TYPE**

\[ \frac{f}{a} = cd + e \] Primary/kernel

\[ \frac{f}{c} = ad + bd \] Primary/not a kernel (d)

\[ \frac{f}{cd} = a + b \] Primary/kernel

\[ \frac{f}{e} = a + b \] Primary/kernel

Common Divisors and Kernels

Goal of multi-level logic optimizer is to find common divisors of two (or more) functions \( f \) and \( g \)

**Theorem (Brayton, McMullin):** \( f \) and \( g \) have a non-trivial common divisor \( d \) (\( d \neq \text{cube} \)) if and only if there exist kernels \( k_f \in K(f), k_g \in K(g) \) such that \( k_f \cap k_g \) is non-trivial, i.e., not a cube

\[ \therefore \] can use kernels of \( f \) and \( g \) to locate common divisors
Algorithm to find All Kernels

Kernels(f)

\[ K = \text{Kernel1}(\emptyset, f / c_f) \]
\[
\text{if } (f \text{ is cube-free })
\]
\[
\quad \text{return}\left( f \cup K \right) ;
\]

Kernel1( j, g )

\[
R = g ;
\]
\[
/* n = \text{number of literals } */
\]
\[
\text{for } ( i = j + 1; i \leq n; i = i + 1 ) \{
\]
\[
\quad \text{if } ( l_i \text{ in one or no terms } ) \text{ continue } ;
\]
\[
\quad c_e = \text{Max. literal cube evenly dividing } g / l_i ;
\]
\[
\quad \text{if } ( l_k \text{ not in } c_e \text{ for all } k \leq i )
\]
\[
\quad \quad R = R \cup \text{Kernel1}(i, (g / l_i) / c_e)
\]
\[
\}
\]
\[
\text{return}\left( R \right) ; \quad \{\text{Presume total ordering on literals}\}
\]

Kerneling Example

\[ f = \text{abcd + abce + adgh + aegh + abde + acdeg + beh} \]

\[ \text{co-kernel} \quad \text{kernel} \]
\[
1 \quad a(bc + gh)(d + e) + ade(b + cg) + beh
\]
\[
a \quad (bc + gh)(d + e) + de(b + cg)
\]
\[
ab \quad c(d + e) + de
\]
\[
abc \quad d + e
\]
\[
abd \quad c + e
\]
\[
ac \quad b(d + e) + deg
\]
\[
acd \quad b + eg
\]
\[
ace \quad b + dg
\]
\[
ad \quad b(c + e) + g(ce + h)
\]
\[
ade \quad b + cg
\]
\[
adg \quad ce + h
\]
**Kerneling Illustrated**

\[ \text{abcd + abce + adfg + aefg + adbe + acdef + beg} \]

**Pruning Condition and Example**

If the largest cube factor \( c_e \) contains an already selected literal, then terminate current branch.

All kernels that would be found by continuing have already been produced.

\[ f = abc(d + e)(k + l) + agh + m \]

\[ f / a = bc(d + e)(k + l) + gh \]

\[ f / ab = c(d + e)(k + l) \]

\[ f / ac = b(d + e)(k + l) \]

\[ b \text{ already selected} \]
Where are we?

What have we considered?
What have we not considered?

Orchestration of Optimization Techniques

Technology-independent:
- two-level minimization
- selective collapsing
- algebraic decomposition
- restructuring for timing
- redundancy removal
- transduction
- global-flow

Technology-dependent:
- tree covering
- load buffering
- rule-based mapping
- signature analysis
- inverter phase assignment
- discrete sizing
Logic optimization - summary

Current formulation of logic synthesis and optimization is the most common techniques for designing integrated circuits today

Has been the most successful design paradigm 1989 - present

Almost all digital circuits are touched by logic synthesis

- Microprocessors (control portions/random glue logic ~ 20%)
- Application specific standard parts (ASSPs)- 20 - 90%
- Application specific integrated circuits (ASICs) - 40 - 100%

Real logic optimization systems orchestrate optimizations

- Technology independent
- Technology dependent
- Application specific (e.g. datapath oriented)

Extra slides
Computing $f/g$

Given $f = \{ c_i \}$, $g = \{ a_i \}$

1) Encode cubes $a_i \in g$ with unique integer codes, by assigning a unique bit position for every literal in $\text{sup}(g)$
   e.g., $g = ab + e$
   \[
   \begin{array}{c}
   110 \\
   001
   \end{array}
   \]

2) Encode cubes $c_j \in f$ similarly
   e.g., $f = abc + abd + de$
   \[
   \begin{array}{ccc}
   110 & 110 & 001
   \end{array}
   \]

3) Sort $\{ a_i, c_j \}$ by their codes
   e.g., $ab, abc, abd, e, de$
   \[
   \begin{array}{c}
   110 \\
   001
   \end{array}
   \]
   \[
   h_1 = c + d \\
   h_2 = d
   \]

Strong (or Boolean) Division

Given a function $f$ to be strong divided by $g$

Add an extra input to $f$ corresponding to $g$, namely $G$ and obtain function $h$ as follows

\[
\begin{align*}
  h_{DC} &= G\overline{G} + \overline{G}g \\
  h_{ON} &= f_{ON} - h_{DC} \\
  h_{OFF} &= f_{ON} + h_{DC}
\end{align*}
\]

Minimize $h$ using two-level minimizer
Decomposition - details

\[ F = \begin{cases} 
  f_1 &= AB + AC + AD + AE + \overline{A}BCDE \\
  f_2 &= \overline{AB} + \overline{AC} + \overline{AD} + \overline{AF} + \overline{A}BCDF 
\end{cases} \]

**Factor F**

\[ F = \begin{cases} 
  f_1 &= A(B + C + D + E) + \overline{ABCDE} \\
  f_2 &= \overline{A}(B + C + D + F) + \overline{ABCDF} 
\end{cases} \]

**Extract common expression**

\[ G = \begin{cases} 
  g_1 &= B + C + D \\
  f_1 &= A(g_1 + E) + \overline{A}Eg_1 \\
  f_2 &= \overline{A}(g_1 + F) + \overline{A}Fg_1 
\end{cases} \]