Lecture 11: More Logic Synthesis
CAD of VLSI systems (046880)

What we already learned:
- 2-level logic minimization

Today’s lesson:
- Multilevel minimization
  - Technology-independent
- Technology Mapping
  - Technology-dependent
Where we are….

Synthesis

- Functional
- Circuit (electrical, timing)
- Layout (geometrical)

Logical

- Architectural (behavioral, high-level)
- Sequential (FSMs, retiming)
- Combinational

Technology

- Multi-level
- Technology-dependent
- Technology-independent
- 2-level (SOP, POS)

Previous lecture

Next Semester
Why Multi-level?

- **2-level logic**
  - 2 gates between input and output
  - Impractical if too many product terms
    - e.g. N-bit adder, other arithmetic circuits
  - Impractical if large-fan-in
    - May be too slow
  - Many possible covers, as we have seen

- **Multi-level logic** (a.k.a. factored)
  - Any number of gates on a path
  - Most widely used in practice
  - More compact
    - Exploits common factors
  - Often faster, consumes less power
  - Advantages of complex gates
  - Even more degrees of freedom in design
    - Exact optimization is impractical (today)

\[
\begin{align*}
ac + bc + ad \\
a & \quad \quad \quad b & \quad \quad \quad c & \quad \quad \quad d \\
a & \quad \quad \quad c \quad \quad \quad b & \quad \quad \quad c \\
\end{align*}
\]
Representing multilevel logic

- These are circuits with the following restrictions:
  - Leaf cells have multiple inputs and a single output
  - Leaf cells have a combinational logic function
  - Pins at all levels are inputs or outputs (no INOUT)
    - Pins of the root called primary inputs and primary outputs
  - Each net has a single source terminal (no multiple drivers)
  - No “logic loops”

- With these restrictions:
  - A circuit can be represented by a directed graph.
    - What are the vertices? What are the edges?

- Note:
  - Since there are no loops in the circuit, this is a DAG (directed acyclic graph) with a topological order
  - This is a hybrid structural/behavioral representation
  - A real gate-level Implementation may differ because of fan-in limitation, library mapping.
Logic network “views”

- Why we need to discuss this?
  - For 2 level circuits the implementation (gates) directly follow the logic model
  - But for multi-level synthesis we have a choice…

- Merged View (“what you see is what you get”)
  - Every vertex in the logic network represent an actual gate/cell in the circuit implementation

- Separate the logic model view from the actual implementation
  - Logic Model: Each vertex can be an arbitrary logic function
    - Represented as SOP, or BDD, or a “factored form”
    - Factored forms are basically expressions with parentheses such as \((ad+b’c)(c+d’(e+ac’)) + (d+e)g\)
  - Gate Model: each vertex must be a single generic gate
    - E.g. 2-input NAND

- Pros and cons of each approach?
Multi-level synthesis: typical flow

1. Logic equations

2. Technology-independent optimization

3. Generic logic network

4. Technology-dependent optimization

5. Optimized logic network

- Factorization and Re-structuring
  - find common expressions
  - network transformations

- Cell-library binding and local transformations
  e.g. sizing, buffering

- Cell library, timing parameters
  layout parameters
How technology-independent optimization works?

Example:

\[ f_1 = abcd + abce + ab'cd' + ab'c'd' + a'c + cdf + abc'd'e' + ab'c'df' \]
\[ f_2 = bdg + b'dfg + b'd'g + bd'eg \]

Minimizing SOP (Espresso or alike):

\[ f_1 = bcd + bce + b'd' + a'c + cdf + abc'd'e' + ab'c'df' \]
\[ f_2 = bdg + dfg + b'd'g + d'eg \]

Factoring (find common sub expressions within the function):

\[ f_1 = c(a + b') + d'(b' + e) \]
\[ f_2 = g(b + f) + d'(b' + e) \]

Decomposition (find common sub expressions amongst the functions):

\[ f_1 = c(x + a') + ac'x' \]
\[ f_2 = gx \]
\[ x = d(b + f) + d'(b' + e) \]

Key problems:

- Find good common sub-functions (“factors”)
- Perform the “division”
Re-structuring

- Assume each vertex in the logic network is a SOP expression
- Examine the network and do sequences of re-structuring transformations such as those in previous example
Decomposition of a single function

- Divide the inputs into 2 sets A and B (possibly: disjoint sets)
  - A is called *free set*
  - B is called *bound set*
- Compose F as a new function \( f \) of fewer variables:
  - Variables in B
  - Some *functions* of the variables in A
- What is the difference between decomposition and factoring?
More re-structuring operations

- **Extract:** decompose several functions using a common sub-expression

- **Substitute:** like extraction, but use an existing vertex $v$ for the common sub-expression

- **Eliminate:** delete a vertex, replace it with its expression when needed downstream
  - This a “reverse action” why do it? Reduce logic levels, get rid of bad initial structure, start a new optimization

- **Simplify:** apply espresso on a vertex
  - result may be global (if an input signal disappears), or local (if it just saves literals internally in the node)
Factored Forms

- **Definition**: A SOP of SOPs with literals as leaves
- **Examples** of factored forms: \( x, \ y', \ abc', \ a+b, \ ((a+b)(c+a'+de) + f) \)
  - Any combination of parentheses, literals, AND/OR operations
- **Non-example**: \((a+b)'c\)
  - Only literals can be negated
- Can be represented as a tree
  - Each sub tree is called a **factor**
- **\( r(F) \)** Size of factored form = number of literals
- Factored forms terminology
  - **Equivalent** – if represent same logic function
  - **Syntactically Equivalent** – if factoring trees are isomorphic
  - **Maximally Factored** – if:
    - For every SOP there are no two syntactically equivalent factors in the products
    - For every POS there are no two syntactically equivalent factors in the sums
  - **Optimum** – if no other equivalent form has fewer literals (only \( x/x' \) impact)
Boolean Division

- \( f = p \cdot q + r \)
  - \( p \) is a boolean divisor of \( f \)
  - \( q \) is called quotient
  - \( r \) is called remainder (not necessarily contained in \( q \))

- \( f = p \cdot q \)
  - \( p \) is a boolean factor of \( f \)
  - It divides \( f \) evenly
  - \( p \) is a factor iff \( f \) is contained in \( p \)

- **Note:**
  - No notion of inverse here, such as in \( y = 1/x \) ...
  - Not real division
  - Too many divisors.... whenever \( f \cdot p \neq 0 \), \( p \) is a divisor of \( f \)!
The factorization problem

- Find a “good” divisor $p$ such that $f = p \cdot q + r$
  - Save literals
  - Good divisors should be common to many functions in the circuit
- Introduce $p$ as a new signal in the network
- Change the existing logic:
  - Express existing signals using $p$

- As a result of this work, cost of the network should be reduced:
  - Smaller area
    - Approximated by number of literals
  - Shorter delay
    - Approximated by counting logic levels on critical path
  - Testability
Why is it a hard problem? What to do?

- There are too many possible choices! Too many divisors! The challenge is to find a good divisor for the EXTRACT operation.

- We can limit ourselves to a subset: \textit{algebraic forms}
  - This is a heuristic approach
  - Easier to find and check than Boolean
  - Lower quality factorization, because Boolean properties are ignored, but we’ll save CPU time
  - Furthermore: limit ourselves to weak \textit{algebraic division}
Algebraic vs. Boolean Expressions

- Definition: **Algebraic Expression**
  
  $F$ is an Algebraic Expression if $F$ is SOP represented by a set of cubes $\{C_i\}$ and $\forall i \neq j: C_i \not\subseteq C_j$
  
  - i.e. no cube is contained in another

- Definition: **Expression Support** $\text{supp}(f)$
  
  Is the set of literals that $F$ explicitly depends on
  
  - E.g. $F=ab+cd$ $\text{supp}(f) = \{a,b,c,d\}$

- Definition: $F$ and $G$ are **Orthogonal Expressions**: if $\text{supp}(F) \cap \text{supp}(G) = \emptyset$
Algebraic Product and Division

**Definition: Algebraic Product**

F = PQ is *Algebraic Product* if P and Q are orthogonal (otherwise it is Boolean product)

- Example: \((a+b)(c+d) = ac + ad + bc + bd\) (algebraic)
- \((a+b)(a+c) = a+ab+ac+bc\) (Boolean)

**Definition: Division Operation**

Given F and P as SOP generate Q and R such that \(F = PQ + R\)

**Definition: Algebraic Division**

A division operation where PQ is Algebraic Product (otherwise it is Boolean division)

- Example: \(F = ad + ae + bcd + j\)
  - For divisor \(P = a\), \(Q = d+e\), \(R = bcd+j\)
  - For divisor \(P = a+bc\), \(Q=d\), \(R=ae+j\)
Weak Division

- Division F = PQ+R is Weak if
  - PQ is algebraic (P and Q depend on different variables)
  - **R has as few cubes as possible**
  - PQ+R is the same set of cubes as F

- It’s like division of polynomials in school-algebra
  - Treat a and a’ as if they were independent variables!

- Essence of the algorithm to find Q and R:
  - For each cube p_i of P, go through the cubes f_i of F and divide
  - Choose Q as the intersection of all those results

- Examples
  - F=ae + be + cde + ab, P=a+b
  - \( Q_a = F/a = e + b; \) \( Q_b = F/b = e + a \Rightarrow Q = Q_a \cap Q_b = e \)
Weak Division Algorithm

Given: F and P as set of cubes $f_i$ and $p_i$ respectively

\[
\text{WEAK\_DIV}(F, P) \{
\text{foreach } p_i \text{ in } P \{
L_{p_i} = \text{Literals}(p_i)
Q_i = \emptyset
\text{foreach } f_j \text{ in } F \{
L_{f_j} = \text{Literals}(f_j)
\text{if } L_{p_i} \subseteq L_{f_j} \{
Q_{ij} = (f_j \text{ with all } L_{p_i} \text{ deleted})
Q_i = Q_i \cup Q_{ij}
\}
\}
Q = \bigcap_i Q_i
R = F - PQ
\return (Q, R)
\}
\]

Example 1: divide F by P

F = $ac + ad + bc + bd + e$
P = $a + b$
p1 = \{a\}, Q_{11} = \{c\}, Q_{12} = \{d\}, Q_{13} = Q_{14} = \{\}, Q_1 = \{c, d\}$
p2 = \{b\}, Q_{21} = Q_{22} = \{\}, Q_{23} = \{c\}, Q_{24} = \{d\}, Q_{15} = \{\}, Q_2 = \{c, d\}$
Q = \{c, d\} (meaning cube $c + d$)
R = $(ac + ad + bc + bd + e) - (c+d)(a+b) = e$ (the - as in sets \)

Example 2:

F = $ad + aef + ab + b’cd + b’cef$
P = $a + b’c$
p1 = \{a\}, Q_1 = \{d, ef, b\}$
p2 = \{b’c\}, Q_2 = \{d, ef\}$
Q = \{d, ef, b\} \cap \{d, ef\} = \{d, ef\} = d + ef$
R = \{ab\} = ab
Cube-Free Expressions and Primary Divisors

- **Motivation:** Find common factor/divisors for multiple functions
  - Using WEAK_DIV we can find divisors and factors of functions
  - BUT THERE ARE TOO MANY OPTIONS!

- **Definition:** An expression $F$ is **cube-free** if there is no cube that is a factor of $F$
  - This is true if there is no literal that is common to all cubes. Why?
  - E.g. $ab+c$ is cube-free, $ab+ac$ is not, $abc$ is not

- **Definition:** **Primary divisors** of $F$ are the set of divisors $D(F) = \{F/c \mid c \text{ is a cube}\}$ (where / is weak div)
  - I.e. $D$ is the large set of the results of dividing $F$ by any possible cube
  - Example: for $F = (a+b+c)(d+e)f + bfg + h$
    - $df+ef (=F/a)$ is a primary divisor
    - $d+e+g (=F/bf)$ is a primary divisor
Kernels and Co-Kernels (Algebraic)

- Definition: \( \{K_f\} = \text{Kernels of } F \) are the \textit{cube-free primary-divisors} of \( F \)
- Definition: \( \text{Co-Kernel} \) is the cube ‘\( c \)’ used to obtain the Kernel by the \text{week_div} \( F/c \)
- Definition: A Kernel is said to be \textit{Level-0} if it has no kernels except itself
- Definition: A Kernel is \textit{Level-n} if it has at least one kernel of level \( n-1 \) but no higher (except itself)

Example: for \( F = (a+b+c)(d+e)f + bfg + h \)
- \( df+ef (=F/a) \) is \textit{not} \( K_F \) since it is a primary divisor but \textit{not} cube-free
- \( d+e+g (=F/bf) \) is a \( K_F \) since both primary divisor and cube free
Observation: Co-Kernels of $F$ must be intersections of literals of $\geq 2$ cubes
- Otherwise the resulting single cube divisor is not cube-free

```java
FindKernels(F) {
    K = {}; level = 0;
    foreach variable v of F {
        S = {cubes of F containing literal v};
        foreach possible sub group of S with $m > 1$ cubes {
            c = {a cube resulting by intersecting all $m$ cubes – intersecting variable wise!};
            if did not try c before {
                k = week_dev(F, c)
                (R, d) = FindKernels(k); /* R is a unique set of Kernels found below */
                K = K U R;
                if R != $\emptyset$ level = max(level, d+1)
            } // new c
        } // all groups
    } // all v’s
    if F is cube-free
        K = K U F; /* a function is always its own kernel if it is cube-free */
    return(K, level)
}
```
Example Finding All Kernels

- Alternative: find all potential co-kernels (at least 2 cubes), then try
- \( F = abc+abd'+bcd'+ade+cde+a'ce \)

1. Find all co-kernels by inspecting all variables

<table>
<thead>
<tr>
<th>Var</th>
<th>cubes</th>
<th>intersections</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>abc, abd', ade</td>
<td>a, ab</td>
</tr>
<tr>
<td>a'</td>
<td>a'ce</td>
<td>a'ce</td>
</tr>
<tr>
<td>b</td>
<td>abc, abd', bcd'</td>
<td>b, ab, bc, bd'</td>
</tr>
<tr>
<td>c</td>
<td>abc,bcd',cde</td>
<td>c,bc</td>
</tr>
<tr>
<td>d</td>
<td>ade</td>
<td>ade</td>
</tr>
<tr>
<td>d'</td>
<td>abd', bcd'</td>
<td>bd'</td>
</tr>
<tr>
<td>e</td>
<td>ade,cde, a'ce</td>
<td>e,ce,de</td>
</tr>
</tbody>
</table>

2. Now calculate the prime divisors

- Now for each divisible Kernel (level=?) run the above stages and mark it level when returning

<table>
<thead>
<tr>
<th>cube</th>
<th>primary divisor</th>
<th>kernel</th>
<th>level</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>bc+bd'+de</td>
<td>Yes</td>
<td>?</td>
</tr>
<tr>
<td>ab</td>
<td>c+d'</td>
<td>Yes</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>ac+ad'+cd'</td>
<td>Yes</td>
<td>?</td>
</tr>
<tr>
<td>bc</td>
<td>a+d'</td>
<td>Yes</td>
<td>0</td>
</tr>
<tr>
<td>bd'</td>
<td>a+c</td>
<td>Yes</td>
<td>0</td>
</tr>
<tr>
<td>c</td>
<td>ab+bd'+de+a'e</td>
<td>Yes</td>
<td>?</td>
</tr>
<tr>
<td>e</td>
<td>ad+cd+a'c</td>
<td>Yes</td>
<td>?</td>
</tr>
<tr>
<td>ce</td>
<td>d+a'</td>
<td>Yes</td>
<td>0</td>
</tr>
<tr>
<td>de</td>
<td>a+c</td>
<td>Yes</td>
<td>0</td>
</tr>
</tbody>
</table>
Finally: Finding Common Kernels

- Theorem: *Bryton/McMullen*
  - There is a common multi-cube divisor for $F$ and $G$ IFF there exists $K_f$ and $K_g$ and their intersection has $> 1$ cube. *That intersection IS the common divisor*
  - Intersection here means common cubes

- So we know what to do!
Common Kernels Example

- Given \( f = ae + be + cde + ab \)
- \( g = ad + ae + bd + be + bc \)

<table>
<thead>
<tr>
<th>K(f) Kernel</th>
<th>Co-Kernel</th>
</tr>
</thead>
<tbody>
<tr>
<td>a+b+cd</td>
<td>e</td>
</tr>
<tr>
<td>b+e</td>
<td>a</td>
</tr>
<tr>
<td>a+e</td>
<td>b</td>
</tr>
<tr>
<td>ae+be+cde+ab</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>K(g) Kernel</th>
<th>Co-Kernel</th>
</tr>
</thead>
<tbody>
<tr>
<td>a+b</td>
<td>d or e</td>
</tr>
<tr>
<td>d+e</td>
<td>a or b</td>
</tr>
<tr>
<td>c+d+e</td>
<td>b</td>
</tr>
<tr>
<td>ad+ae+bd+be+bc</td>
<td>1</td>
</tr>
</tbody>
</table>

- The only intersection of kernel with > 1 cubes is: a+b
- That is the only common divisor we can use for \( f \) and \( g \)!
Boolean Don’t Care Simplification

- Most powerful transformation
  - Simplify, meaning 2-level minimization of a sub-network

- Origin of don’t cares
  - Satisfiability don’t cares (SDC): Input patterns that can’t happen
    e.g. $x \text{ EXOR } (a+b) = 1$
  - Controllability don’t cares (CDC):
    Primary input patterns that can’t happen, propagated to our subnetwork inputs.
    E.g. $x_1'x_2'=1$ can’t happen $\Rightarrow$ CDC=$b'c'$
  - Observability don’t cares (ODC):
    Conditions that make $F$ insensitive to $v$.
    (e.g. Boolean derivative of $F$ wrt $v = 0$, propagated back, for any $z$)

- Propagation of CDC and ODC can be hard…
  Every network transformation changes the DC conditions…..
Cell-library binding (a.k.a. Technology Mapping)

- What do we start with?
  - Generic logic network, after technology-independent optimization.

- What are we trying to do?
  - Implement the logic by using cells from a given technology-specific library
    - Min. area for a given delay, or
    - Min. delay, with a constraint on area
  - Note: library-based design is beneficial, but is not the only way!
About libraries

- Mapping a logic design into “standard parts” became a common methodology in VLSI since the advent of CMOS VLSI in the late 80’s
  - In NMOS technology, individual device sizing was the norm
  - Standard-cell libraries enabled the ASIC industry. Why?

- Standard-cell libraries usually contain several combinational logic gates and sequential elements
  - Usually limited fan-in (e.g. no more than 4 inputs)
  - Including some ‘complex gates’ such as AOI
  - Cells come in a variety of sizes (e.g. S,M,L,X,…) to suit different capacitive loads
    - Inverters come in many sizes
  - Each cell is laid-out carefully, and is characterized
    - Delay, power, area,……
  - Cells are designed to fit a physical implementation
    - e.g. gate array, PLD, cell-based layout….. And associated CAD tools

- Library-based design is an excellent example of a successful methodology
  - Exploits abstraction, regularity, hierarchy, automation
Why library-binding is a hard problem?

- In this example:
  - Trivial binding:
  - more possible matches: using the ORAND gates
  - Even more possible solutions unrelated to the trivial binding (e.g. 2-level SOP for y and z)
  - This is a binate covering problem (for each selection we have to take care of what’s left as a consequence)

- In fact, we saw that even just proving that a bound network is equivalent to the unbound network was intractable (Tautology!)

- What to do?
  - Again: Use heuristics - reduce the problem complexity, solve a simpler problem, lose in optimality of solution
How library binding (usually) works?

- **Step 1**: convert the input logic network into a ‘standard form’
  - Decomposition into base functions
    - E.g. 2-input NANDs
  - This is called base-match

- **Step 2**: partition into sub-graphs
  - Smaller problems (divide and conquer)
  - Also: separate out latches and flip-flops
  - If sub-graphs are trees, covering is easier!

- **Step 3**: Solve the covering problem for each sub-graph
  - This is done optimally in linear time, but steps 1 and 2 determine quality of result

Cut multiple-fanout nets for partitioning into logic trees
Optimal tree covering by *Dynamic Programming*

Dynamic programming: A decision-sequence that makes the best decision on every sub-problem, and leads to an optimum decision overall. (Our problem follows this principle, fortunately)

**Example Library:**

<table>
<thead>
<tr>
<th>Cell</th>
<th>Pattern graphs</th>
</tr>
</thead>
<tbody>
<tr>
<td>inv</td>
<td><img src="image" alt="inv pattern" /> 2</td>
</tr>
<tr>
<td>nand2</td>
<td><img src="image" alt="nand2 pattern" /> 3</td>
</tr>
<tr>
<td>and2</td>
<td><img src="image" alt="and2 pattern" /> 4</td>
</tr>
<tr>
<td>or2</td>
<td><img src="image" alt="or2 pattern" /> 5</td>
</tr>
</tbody>
</table>

Library is described by *patterns* of NANDS. An inverter is a single-input NAND.

**Solution:**

- Cost of covering *u*: 3
- Cost of covering *t*: 4 (better than 2+3)
- Cost of covering *s*: 2
- Cost of covering *r*: 3+2+4, or 5+3 (better!)
Tree-Cover algorithm

Tree_cover (T, (V, E)) { 
    Set the cost of internal vertices to -1;
    Set the cost of leaf vertices to 0; /* input nets*/
    While (some vertex has negative weight) do {
        Select a vertex v in V whose children have all non-negative cost;
        M = set of all matching pattern-trees at vertex V ;
        /* use a recursive match algorithm */
        cost(v) = minimum over all possible matches, of:
            (cost of the matching pattern itself +
             cost of vertices matching its leaves)
    }
}
Polarity assignment during binding

- Trick:
  - Insert pairs of inverters between gates everywhere in the unbound network. It does not change the logic.
  - Add a fictitious cell into the library, whose cost is 0, equivalent to 2 inverters in series

- The covering algorithm will now work harder, but may find lower-cost solutions
  - Using some negated input signals or producing the complement of some sub-network function
Rule-based transformations
(vs. the algorithmic approach)

- So far, we looked at heuristic transformations, applied systematically to the whole network
- But it is also possible to apply *local transformations*, based on pattern identification and replacement, trying to optimize the circuit
  - Mimic the way an experienced designer would work
  - Usually, use a database of “rules” (*expert system*)
  - The earliest synthesis CAD systems used this approach
  - More flexible too (e.g. multi-output cells)
  - Interesting problems: control (rules about rules)
Summary

- Multilevel combinational logic synthesis
  - Logic transformations: Extract, Eliminate, Simplify
  - Factorization: Boolean/Algebraic divisors
  - Origin of Don’t Cares in logic networks
  - Cell library binding
  - Tree covering
  - Rule-based approach to circuit optimization