What you've seen so far...
- 2-level minimization a la ESPRESSO
  - Manipulates (reshapes) SOP covers of functions
  - Heuristic: REDUCE - EXPAND - IRREDUNDANT

What's left?
- Multi-level minimization, where final form of logic network is not just 2-level SOP AND-OR form

What do we need?
- New, more general model of logic networks
- New operators: forms of division for Boolean functions
- New heuristic minimization strategies to use this model + operators
### Where Are We?

- **Moving on to real logic synthesis**—for **multi-level** stuff

<table>
<thead>
<tr>
<th>M</th>
<th>T</th>
<th>W</th>
<th>Th</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aug</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
</tr>
<tr>
<td>Sep</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td>Oct</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>29</td>
<td>30</td>
<td>31</td>
<td>1</td>
</tr>
<tr>
<td>Nov</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
</tr>
<tr>
<td>Thanksg</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
</tr>
<tr>
<td>Dec</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
</tbody>
</table>

- **Introduction**
- Advanced Boolean algebra
- JAVA Review
- Formal verification
- 2-Level logic synthesis
- **Multi-level logic synthesis**
- Technology mapping
- Placement
- Routing
- Static timing analysis
- Electrical timing analysis
- Geometric data structs & apps

### Readings

- DeMicheli has a lot of relevant stuff
  - Again, he worked on some of this at Berkeley and at IBM
- **Read this in Chapter 8**
  - 8.1 Intro: take a look.
  - 8.2 Models and Transforms—this is about the “Boolean network model”
  - 8.3 The Algebraic Model -- how people do factoring for complex Boolean logic networks
Why Multi-Level Forms

- 2-level too restrictive: specific area vs delay tradeoff
  - Area = gates + literals (wires), i.e., things that take space on a chip
  - Delay = max levels of logic gates required to compute function
  - 2-level is minimum gate delay possible, but usually worst on area

Why Multi-Level?

- Rarely see 2-level designs for really big things, mostly for pieces of bigger things
  - Even smallish things routinely done as multi-level
Real MultiLevel Example

- ...and this is a pretty small design, done by Synopsys DesignCompiler

---

Boolean Logic Network Model

- Need more sophisticated model of these networks
- New model: Boolean Logic Network
  - Idea: it's a netlist of connected components, like a logic diagram, but now individual components can be arbitrary Boolean functions

Ordinary gate netlist

```
<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

x

y

Same circuit as a Boolean logic network, x, y are now Boolean functions

```
| primary inputs | internal vertices | primary outputs |
```

© R. Rutenbar 2001, CMU 18-760, Fall 2001

Page 4
It’s just a graph, with:

- Primary inputs (usually vars)
- Primary outputs (stuff network creates for other logic to consume)
- Intermediate nodes that are themselves represented as Boolean functions...all in SOP form

Now what?

- Look at some operators that one can use to manipulate these networks
- Some are fairly simple structural operations on graphs
- Some will require entirely new operators (like division)
- Our derivation follows DeMicheli closely, sections 8.1 and 8.2

Consider example from De Micheli

Let’s look at some operations on this network...

\[
\begin{align*}
p &= ce + de \\
q &= a + b \\
r &= p + a' \\
s &= r + b' \\
t &= ac + ad + bc + bd + e \\
u &= q'c + qc' + qc \\
v &= a'd + bd + c'd + ae' \\
w &= v \\
x &= s \\
y &= t \\
z &= u
\end{align*}
\]

Network Quality measure = \(\sum_{\text{nodes}} (\text{literals})\) =
Reminder: Boolean Network Model

- Remember what this picture means
  - It’s a graph
  - Has primary inputs and outputs
  - Internal nodes mean “here is an SOP-form Boolean function”
  - Edges mean “here are signals going into/out of these functions”
  - #literals = count up all lits in every SOP equation in every Boolean node

As gates it looks like this...

Operations on Boolean Network

- What’s the overall goal here?
  - Simplify the network – reduce total number of literals
  - Optimize timing – reduce delay from input to output thru gates, wires

- 3 basic types of operations
  - Add new network nodes: this is related to factoring—take “big” nodes and factor them into more, better, smaller nodes
  - Remove network nodes: take nodes that are “too small” and substitute them back into the fanout nodes that they feed
  - Simplify network nodes: no change in # of nodes, just simplify insides

- A big set of possible operators in real implementations
  - Look at just a couple of examples...
**Network Ops: Elimination**

- **Reducing #nodes: Elimination**
  - Removes an internal vertex by replacing it (adding its SOP expression) into all the other vertices it feeds.
  - Note: eliminate vertex for $r$ requires substituting $(p+a')$ in $s$ node.

```
a                   v=a'd+bd+c'd+ae'
b p=ce+de           s=p+a'+b'
c     t=ac+ad+bc+bd+e  
d q=a+b
```

$\Sigma \text{lits } =$

**Network Ops: Extraction**

- **Adding nodes: Extraction**
  - Create a new vertex that represents a common subexpression for $\geq 2$ vertices, and add it to network.
  - Substitute the output of the new vertex for common parts elsewhere.
  - Note that: $p = (c+d)e$ and $t=(c+d)(a+b) + e$, so extract $c+d$.

```
a                   v=a'd+bd+c'd+ae'
b  p=ke           r=p+a' s=r+b'
c     t=ka+kb+e
```

$\Sigma \text{lits } =$
**Network Ops: Simplification**

▲ Simplifying a node: 2-Level Simplification

- Run a 2-level minimizer (ESPRESSO!) at a vertex -- see if the SOP cover of the vertex gets simpler
  - Note -- if you don't eliminate any vars, it's a local transformation
  - If you actually eliminate a var, it's global -- changes the network
  - Note: note $u = q'c + qc' + qc = q+c$

**Network Ops: Iterative Improvement**

▲ Sort of like ESPRESSO loop

- Iteratively apply these (and other) ops to network to try to improve it
- Usually count literals (all wires into each node of the network) or count (gates + literals)
- Our example can simplify to this by applying these (and other) ops:
Network Ops: Scripts

What do people really use to do multi-level optimization?
- Programs like MIS II, SIS, HSIS, VIS (from Berkeley)
- Commercial tools from Synopsys, Synplify, Cadence, Avanti

What do multilevel synthesis tools look like?
- Use Boolean network model
- Provide collections of network operators
- Users invoke scripts that run a sequence of these ops on their design

What's a script look like...

Here is a “famous” script originally from MIS II tool

The so-called “rugged” script
- A sequence of network ops...

```
sweep; eliminate -1
simplify -m nocomp
eliminate - 1

sweep; eliminate 5
simplify -m nocomp
resub -a

fx
resub -a; sweep

eliminate -1; sweep
full_simplify -m nocomp
```
Running Real Logic Synthesis: SIS

SIS is a Berkeley multi-level synthesis tool

\texttt{/afs/ece/class/ee760/sis} is the binary for IBM and SUN

\begin{verbatim}
UC Berkeley, SIS Development Version (compiled 2-Nov-95 at 6:54 PM)
sis>
\end{verbatim}

Command prompt
Type "help" to get a list of all commands

Rugged Ops: Sweep

\textbf{Sweep ...}

- Eliminates all single-input vertices
- Eliminates vertices with a constant function (ie, ==0, ==1 always)
- Sort of a basic “clean up” op

- \texttt{sweep}; eliminate -1
- \texttt{simplify -m nocomp}
- \texttt{eliminate -1}

- \texttt{sweep}; eliminate 5
- \texttt{simplify -m nocomp}
- \texttt{resub -a}
- \texttt{fx}
- \texttt{resub -a; sweep}
- \texttt{eliminate -1; sweep}
- \texttt{full_simplify -m nocomp}
Sweep Examples

Sweep examples

\[ a \]
\[ F = a \]
\[ G = F \]
\[ H = F \]
\[ Q = a + a' \]

Running sweep in SIS

\textbf{SIS session}

\begin{verbatim}
sis> read_eqn sweep.eqn
sis> print
  F = a
  \{G\} = F
  \{H\} = F
  \{Q\} = a + a'
sis> sweep
sis> print
  \{Q\} = a + a'
  \{G\} = a
  \{H\} = a
\end{verbatim}

UNIX file: sweep.eqn

\[ F = a; \]
\[ G = F; \]
\[ H = F; \]
\[ Q = a + a'; \]

Change in total literal count:
Aside: SIS Syntax

For a typical eqn format input file

- + means OR
- * means AND
- " " (a space) also means AND
- ' (one apostrophe) means NOT (on a literal)
- ( ) used for grouping
- != means EXOR
- == means EXNOR
- !( ) means NEGATE the contents of the parens
- F (a capital letter) usually means a function, output of a network node
- x (a small letter) usually means a primary input to the overall network

SIS “print” output

- {G} means G is a primary output of the network (nobody else eats it)
- [31] means SIS creates a new Boolean network node during simplification, and it gives you a number in brackets as an ID.

Network Ops: Eliminate

Eliminate <threshold>...

- Eliminates all nodes in the network whose "value" is less than or equal to threshold.
- Value of node
  - Number of times the node is used in the factored form for each of its fanout nodes
  - Number of lits saved by NOT eliminating the node
- Eliminates node by collapsing it into its fanout nodes
- "-1" means eliminate nodes only used once elsewhere in network

```bash
sweep; eliminate -1 simplify -m nocomp
```

```bash
sweep; eliminate 5 simplify -m nocomp resub -a
```

```bash
fx resub -a; sweep
```

```bash
eliminate -1; sweep
```

```bash
full_simplify -m nocomp
```
“Value” of Elimination

Scenario
- We have a vertex that has \( L \) literals in it; it feeds \( N \) other vertices
- What happens if we eliminate it? What is “value” of this?
- Answer is: change in total number of literals in design

\[
\begin{align*}
G_1 &= F + \ldots \\
G_2 &= F + \ldots \\
&\quad \vdots \\
G_N &= F + \ldots \\
\end{align*}
\]

Total literals before =

We eliminate vertex \( F \)

\[
\begin{align*}
G_1 &= (L \text{ literals}) + \text{stuff} \ldots \\
G_2 &= (L \text{ literals}) + \text{stuff} \ldots \\
&\quad \vdots \\
G_N &= (L \text{ literals}) + \text{stuff} \ldots \\
\end{align*}
\]

Total literals after =

Change = value =

Eliminate Examples

Eliminate -1

\[
\begin{align*}
F &= ab \\
G &= F+x \\
\text{eliminate} \\
G &= ab+x
\end{align*}
\]

Eliminate 5

\[
\begin{align*}
F &= abc \\
G_1 &= F+d \\
G_2 &= F+ef \\
G_3 &= F+gh \\
G_4 &= F+de \\
\Sigma \text{lits} = \\
G_1 &= abc +d \\
G_2 &= abc +ef \\
G_3 &= abc +gh \\
G_4 &= abc +de \\
\Sigma \text{lits} =
\end{align*}
\]
Running eliminate in SIS

SIS session

```
sis> read_eqn elim.eqn
sis> print
  F = a b c
  {G1} = F + d
  {G2} = F + e f
  {G3} = F + g h
  {G4} = F + de
sis> eliminate 1
sis> print
  F = a b c
  {G1} = F + d
  {G2} = F + e f
  {G3} = F + g h
  {G4} = F + de

UNIX file: elim.eqn
```

No change. Why?
Cost to eliminate F node is +5 literals.
But, we set threshold to +1 literal, so—eliminate won't do anything here. Cost is too high.

Running eliminate in SIS

SIS session continued

```
sis> eliminate 3
sis> print
  F = a b c
  {G1} = F + d
  {G2} = F + e f
  {G3} = F + g h
  {G4} = F + de
sis> eliminate 5
sis> print
  {G1} = a b c + d
  {G2} = a b c + e f
  {G3} = a b c + g h
  {G4} = a b c + de
```

Now it does it.

No change. Why? Same reason.
Cost to eliminate F node is +5 literals.
But, we set threshold to +3 literals, so—eliminate won't do anything here. Cost is too high.
Network Ops: Simplify

- **simplify**
  - Run ESPRESSO on each node
  - Minimize SOP 2-level form of each
  - `-m nocomp` says don’t try to compute the full offset for each node-- makes it run faster

- **full_simplify**
  - Same as simplify, but uses a larger set of don’t cares...
  - ...works harder to try to get a better (smaller SOP) answer

```plaintext
sweep; eliminate -1
simplify -m nocomp
eliminate -1
sweep; eliminate 5
simplify -m nocomp
resub -a
fx
resub -a; sweep
eliminate -1; sweep
full_simplify -m nocomp
```

Simplify Examples

Simplify

- \( F = a + a'b + c \) simplify \( G_1 = a + b + c \)
- \( G = a + a' \) simplify \( 1 \)

Goal is just to “clean up” insides of each node in the Boolean network
Network Ops: Resub

- **Resub -a**
  - Substitute each node in the network into each other node in the network
  - In other words, for each pair of nodes S, T, checks if S is a factor of T, or if T is a factor of S
  - Tries to use both the true and complemented form of the output of each node it tries to substitute
  - Loops until network stops getting “better”, i.e., literal count stops decreasing
  - “-a” means that *algebraic division* is how it checks to see if one node can substitute (divide) into another
  - (We talk about *algebraic division* next -- don’t worry...)

```plaintext
sweep; eliminate -1
simplify -m nocomp
eliminate -1
sweep; eliminate 5
simplify -m nocomp
resub -a
fx
resub -a; sweep
eliminate -1; sweep
full_simplify -m nocomp
```

Resub Example

**Resub example 1**

- F = ab
- G = ab + c
- H = ab + e

**Resub example 2**

- F = ab
- G = ab + c
- H = a' + b' + cd

Note: F was complemented
Running resub in SIS

SIS session

sis> read_eqn resub.eqn
sis> print
  {F} = a b
  {G} = a b + c
  {H} = a b + e
sis> resub -a
sis> print
  {F} = a b
  {G} = (F) + c
  {H} = (F) + e

UNIX file: resub.eqn

F = a b ;
G = a b + c ;
H = a b + e ;

Network Ops: Fx

Fx

- Extracts common subexpressions that are either
  - A single cube (eg, b'cd)
  - A double cube (eg, ab + b'cd)
- Result is a new nodes in the network that represent these common "factors" removed
- Note that after you get these factors, you run "resub" to see which ones are worth keeping
  - ...ie, if it made the network worse to factor them out, resub will put the factors back into the fanout nodes
**fx Example**

F = ab + c + x

G = abx + cx + d

H = ab + d

Fx will consider several potential factors: ab, ab+c, then decide which ones are worth extracting.

**Running fx in SIS**

**SIS session**

```nim
sis> read_eqn fx.eqn
sis> print
{F} = a b + c + x
{G} = a b x + c x + d
{H} = a b + d
sis> fx
sis> print
{F} = [31] + x
{G} = [31] x + d
{H} = a b + d
[31] = a b + c
```

**UNIX file: fx.eqn**

```nim
UNIX file: fx.eqn
F = a b + c + x;
G = a b x + c x + d;
H = a b + d;
```

F = N + x

G = Nx + d

H = ab + d
\textbf{resub \(!= \text{fx}\)}

- \textit{fx} tries to find \textbf{NEW} common factors
  - \textit{It adds} nodes to the network \textit{to do this}
  - \textit{Tries to find good (usable) common subexpressions}

- \textit{resub} uses what is \textbf{already} in network
  - \textit{It CANNOT} go find \textit{or} "extract" \textit{new factors}
  - \textit{It just looks at what nodes are already} around in network
  - \textit{It} tries to \textit{use these} to substitute one node \textit{into another} \textit{to save literals}

- \textbf{So….}:
  - \textit{Do fx first:} \textbf{create} \textit{a bunch of good-looking common factors}
  - \textit{Do resub next:} \textbf{try} \textit{to use these factors} \textit{to improve network}

\textbf{Rugged Script}

- \textbf{Now it’s possible to go back and \textit{really} read the script}
  - \textbf{It should make sense…}
    - \textbf{4 major phases of simplification}
    - \textbf{Goes from easy optimizations \textit{to harder}, more expensive ones}
    - \textbf{Uses ESPRESSO} \textit{to do each individual} \textit{node}
    - \textbf{Uses algebraic division} \textit{to find good common subexpressions}
    - \textbf{Tracks literal count} \textit{to judge quality of network}

\begin{itemize}
  \item \texttt{sweep; eliminate -1 simplify -m nocomp}
  \item \texttt{eliminate -1}
  \item \texttt{sweep; eliminate 5 simplify -m nocomp resub -a}
  \item \texttt{fx resub -a; sweep}
  \item \texttt{eliminate -1; sweep}
  \item \texttt{full_simplify -m nocomp}
\end{itemize}
Multilevel Synthesis: What’s Left?

Factoring: how do we really do it?
- Operators we don’t have are those related to factoring out (extracting) common subexpressions from multiple vertices
  - Allow us to do the substitution, decomposition, extraction ops
  - (Simplification op is just ESPRESSO on 1 vertex)
  - We need this to be able to do the “fx” factoring

New model of Boolean functions: Algebraic model
- Yet another way of thinking about Boolean functions that allows us easily to do several division-like operations
- Term “algebraic” comes from pretending that Boolean expressions behave like polynomials of real numbers, not like Boolean algebra
- Big new Boolean operator: algebraic division

Algebraic Model

Idea: keep just those rules (axioms) that work for polynomials of reals AND Boolean algebra, dump rest

<table>
<thead>
<tr>
<th>Real numbers</th>
<th>Boolean algebra</th>
</tr>
</thead>
<tbody>
<tr>
<td>a•b = b•a</td>
<td>a•b = b•a</td>
</tr>
<tr>
<td>a+b = b+a</td>
<td>a+b = b+a</td>
</tr>
<tr>
<td>a•(b+c) = (a•b)•c</td>
<td>a•(b+c) = (a•b)•c</td>
</tr>
<tr>
<td>a+(b+c) = (a+b)+c</td>
<td>a+(b+c) = (a+b)+c</td>
</tr>
<tr>
<td>a•1 = a</td>
<td>a•1 = a</td>
</tr>
<tr>
<td>a+0 = 0</td>
<td>a+0 = 0</td>
</tr>
<tr>
<td>a+0 = a</td>
<td></td>
</tr>
</tbody>
</table>

SAME

| a+a' = 1              | a•a' = 0                |
| a•a = a              | a+a = a                 |
| a+1 = 1              | a+(b•c) = (a+b)•(a+c)    |

NOT ALLOWED
Algebraic Model

In English
- Only get to use algebra rules from real numbers
- A variable and its complement are treated as totally unrelated

Idea
- Boolean functions represented / manipulated as SOP expressions
- Each product term in such an expression is just a set of variables
- The expression itself is just a set of these products (cubes)

Algebraic Division

Model for factoring
- Given function $f$ we want to factor like this:

$$f = d \cdot q + r$$

- divisor
- quotient
- remainder (if $r=0$, then we say the quotient is a factor)

- (just like regular numbers, e.g., $15 = 7 \cdot 2 + 1$)
- Boolean example
Algebraic Division

Example

\[ f = ac + ad + bc + bd + e \quad \text{want} \quad f = d \cdot q + r \]

<table>
<thead>
<tr>
<th>Divisors (d)</th>
<th>Quotient (q)</th>
<th>Remainder (r)</th>
<th>Factor?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ac + ad + bc + bd + e</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a + b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c + d</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

 Turns out there is a very nice algorithm for this

**Inputs**
- A Boolean expression \( A \) and a divisor (to divide by) \( D \), represented as sets of cubes (and each cube a set of literals)

**Output**
- Quotient \( q = A/D = \) cubes in quotient, or 0 if none
- Remainder \( r = \) cubes in remainder, or 0 if \( D \) was a factor
- ie, figures out \( q, r \) so that \( A = D \cdot q + r = D \cdot (A/D) + r \)

**Strategy**
- Cubewise walk thru cubes in divisor \( D \), trying to divide them into \( A \)
- ...being careful to track which cubes do divide into \( A \)
Algorithm

AlgebraicDivision( A, D) { /* divide D into A */
  for ( each cube d in divisor D ) {
    let C = { cubes in A that contain this product term “d” };  
    if ( C is empty ) {  
      return ( quotient = 0, remainder = A);  
    }  
    let C = cross out literals of cube “d” in each cube of C;    
    if ( d is the first cube we have looked at in divisor D )  
      let Q = C;  
    else Q = Q \ C;  
  }
  R = A - ( Q * B );
  return ( quotient = Q, remainder = R);
}

Example:

Cube xyzw contains product term “yz”

Example:

Suppose C = xyz + yzw +pqyz and d = “xy”. Then crossing out all the “xy” parts yields
z + y + pq

Algebraic Division: Example

A/D:  A = axc + axd + axe + bc + bd + e      D = ax + b

Easiest way manually is to make this table:
one row per cube in A, one column per cube in D, bottom row to evolve Quotient Q and, when done, remember to get remainder

<table>
<thead>
<tr>
<th>A cube</th>
<th>D cube: ax</th>
<th>D cube: b</th>
</tr>
</thead>
<tbody>
<tr>
<td>axc</td>
<td>axc</td>
<td></td>
</tr>
<tr>
<td>axd</td>
<td>axd</td>
<td></td>
</tr>
<tr>
<td>axe</td>
<td>axe</td>
<td></td>
</tr>
<tr>
<td>bc</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bd</td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Q</td>
<td>Q</td>
<td></td>
</tr>
</tbody>
</table>

Remainder R = A – Q*D

R = (axc + axd + axe + bc + bd + e) – [(ax+b)*( ]

Algebraic Division: Warning

- Remember the basic model assumptions
  - Cannot do any “boolean” simplification, only “algebraic”
- So what?
  - OK, suppose you have this

\[
A = ab'c' + ab + ac + bc \quad B = ab + c' \quad \text{want } A / B
\]

- You must transform it to something like this...

- Because you MUST treat the true and compl forms of var as different

One More Constraint: Redundant Cubes

- To do \( A/D \), we need function \( A \) not to have redundant cubes
  - Redundant meaning formally minimal with respect to single-cube containment, ie, “completely covered by other cubes in SOP cover”

\[
F = a + ab + bc \quad \text{is redundant}
D = a \quad \text{is the divisor; we want to do } F/D
\]

now: compute \( F / D \), ie, \( F / a \)
use our algebraic division algorithm...
Multilevel Synthesis Models: Where are We?

Given Boolean $A$, $D$, you can compute $A = Q*D + R$ easily

- This is great—but it's still not enough
- Real problem: I give you $n$ functions $F_1$, $F_2$, … $F_n$, and want to find a set of good common divisors $d_i$

How to find?

- Case 1: divisors $d$ that are just 1 cube (1 product term), e.g., $d = ab$
- Case 2: “bigger” multiple-cube divisors, e.g., $d = ab + c'd + e$

New Idea: Kernels

Where to look for multiple cube divisors? *Kernels*

- *Kernel* of a Boolean expression $f$ is:

- *Co-kernel* of $f$ is:

$$f = d \cdot q + r$$
Kernels

Cube-free means...?

- Means you cannot factor out a single cube (product term) divisor that leaves no remainder
- Technically -- has no one cube that is a factor of expression
- So, you divide expression \( f \) by a cube, look at result, if you can pull out a cube -- any cube -- with 0 remainder, it's not a kernel

Expression \( f \) \( f = d \cdot q + r \) Cube-free?

<table>
<thead>
<tr>
<th>( f )</th>
<th>( f = d \cdot q + r )</th>
<th>Cube-free?</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( a + b )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( ab + ac )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( abc + abd )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( ab + acd + bd )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Kernels of expression \( f \) denoted \( K(f) \)

- Look at example \( f = abc + abd + bcd \)

<table>
<thead>
<tr>
<th>Divisor cube ( d )</th>
<th>( f = d \cdot q + r )</th>
<th>Is it a Kernel of ( f )?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 ( (1)(abc+abd+bcd)+0 )</td>
<td>No, has cube = b as factor</td>
<td></td>
</tr>
<tr>
<td>( a )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( b )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( c )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( d )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( ab )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( ac )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( ad )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( bc )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( bd )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( cd )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( abc )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Kernels

- What don’t we know yet?
  - Why we should care about kernels
  - If we should care, how to find them

- Why you should care:
  **Theorem: Brayton & McMullen**

  Expressions $f$, $g$ have a common multiple-cube divisor $d$ if and only if

  - There are kernels $k_1 \in K(f)$, $k_2 \in K(g)$ such that $d \in k_1 \cap k_2$
  - $|k_1 \cap k_2| \geq 2$

Kernel Theorem

- OK, let’s try that in English...
  - Start with expressions $f$ and $g$
  - Look at sets of kernels of each $K(f)$, $K(g)$
  - Since $k_1$ is a kernel of $f$, $k_2$ is a kernel of $g$, we know that

- Remember: $k_1$, $k_2$ are cube-free, they have to be multi-term SOP expressions lacking a common factorable cube
Kernels

- So if we substitute back into \( f, g \)

- ...but we can rewrite this, pulling out \( k_1 \cap k_2 = (X + Y + ... ) \)

- ...but now it's clear that \( k_1 \cap k_2 = (X + Y + ... ) \)
  is a common, multiple-cube divisor! It's a nice, big common factor!

That was NOT a Proof!!
- ...it was just an example, but it illustrates what's going on

Why is Brayton/McMullen so important?
- It's a necessary and sufficient condition

There is a common multiple-cube divisor for your functions \( f, g \) \iff You can find kernels in \( f \), and in \( g \) such that intersection of kernels gives expression with \( >=2 \) cubes;
  ...that intersection is your divisor

- It's hugely practical: the only place to look for multiple-cube factors is in intersections of the kernels of your functions. There's no place else.
Consider this \( f, g \)

\[
\begin{align*}
\text{\( f = ae + be + cde + ab \)} & \quad \text{\( g = ad + ae + bd + be + bc \)} \\
\end{align*}
\]

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

Intersecting these 2 kernels: \( (a+b+cd) \times (a+b) = (a+b) \)

\( (a+b) \) is a divisor we can consider for both \( f, g \)

So, they are quite useful, but how to get them?

- Another recursive algorithm (are we surprised...?)
- There are 2 more useful properties of kernels we need to see first...

Start with a function \( f \) and a kernel \( k_1 \) in \( K(f) \)

\[
\begin{align*}
\text{\( f = cube1 \cdot k_1 + remainder1 \)} \\
\end{align*}
\]

First: a new, interesting question: what about \( K(k_1) \)??

- \( k_1 \) is a perfectly nice Boolean expression, so it's got its own kernels
- Do these kernels have anything interesting to say about \( K(f) \)??
**Kernels**

- Look at $K(k_1)$
  - Suppose $k_2$ is a kernel in $K(k_1)$, then we know
  - Substitute this in for $k_1$ in original expression for $f$
  - Neat trick: cube1$\cdot$cube2 is itself just another single cube, so rewrite to emphasize this fact:

**Kernel Hierarchy**

- So, what does this say?
  - $k_2$ is itself a kernel of function $f$!
  - There is a hierarchy of kernels, each inside the next, up the hierarchy

- Terminology
  - A kernel $k$ in $K(f)$ is a level 0 kernel if it has no kernels inside it except itself
    - In English: only cube you can pull out is ‘1’ and get a cube-free quotient as the result
  - A kernel $k$ in $K(f)$ is a level $i$ kernel if it contains only kernels of level $< i$, and just one kernel at level $i$ which is itself
    - In English: a level-1 kernel only has level-0 kernels inside it. A level-2 kernel only has level-1 kernels in it, etc…
2nd useful result [Brayton et al]

Co-kernels of a Boolean expression in SOP form correspond to intersections of 2 or more of its cubes in this SOP form.

**NOTE:** Intersections here means specifically that we regard a cube as a set of literals, and look at common subsets of literals

- Note: this is not like “AND” for products.

**Example**

\[ ace + bce + de + g \]

\[ ace \cap bce = ce \implies ce \text{ is a potential co-kernel} \]

\[ ace \cap bce \cap de = e \implies e \text{ is a potential co-kernel} \]

**How do we use these 2 results?**

- Find the kernels recursively –
  - Whenever we find one, call kernel() routine on it, so find (if any) lower level kernels inside

- Use algebraic division to divide function by potential co-kernels, to generate recursive calls…
  - …but be smart: co-kernels are intersections of the cubes
  - …if there’s at least 2 cubes, then look at the intersection \( C \) of the literals in those cubes and use the result as our co-kernel cube
Kernel Algorithm

Algorithm is then...

\[
\text{FindKernels( expression } F) \{ \\
K = \text{null}; \\
\text{for ( each variable } x \text{ in } F \} \{ \\
\text{if ( there are at least 2 cubes in } F \text{ that have variable } x \} \{ \\
\text{let } S = \{ \text{cubes in } F \text{ that have variable } x \text{ in them} \}; \\
\text{let } c = \text{cube that results from intersection of all cubes in } S, \\
\text{this will be the product of just those literals} \\
\text{that appear in each of these cubes in } S; \\
K = K \cup \text{FindKernels( } F / c \}; \\
\} \\
K = K \cup F; \\
\text{return( } K \} \\
\}
\]

algebraic division, but
simpler since it always
just divides by exactly
1 cube, a simple product term

Function \( F \) is always its
own kernel, with
trivial cokernel = 1

Kerneling Example

To start, divide \( f \) by each of the variables, and use to recurse

- We're looking for co-kernels with ONE variable in them
- But—be smart, it cannot be a cokernel unless its in at least 2 cubes

\( f = ace + bce + de + g \)
Kernel Hierarchy, Example Revisited

With this algorithm, overall recursion tree looks like this

\[ f = ace + bce + de + g \]

Kernel Hierarchy

With this algorithm...

- Can find all the kernels (and cokernels too)

Problem

- Will revisit same kernel multiple times

Solution

- Trick: remember which variables you already tried in the cokernels
- Problem: kernel you get for cokernel \(abc\) is same as for \(cba\), but current algorithm doesn't know this and will find same kernel for both cubes
- A little extra book keeping solves this -- see De Michelli pp 367-369
Using Kernels and Co-Kernels

What good are these?

Exactly the right component pieces for...

- Extraction of a single-cube divisor from multiple expressions
- Extraction of a multiple-cube divisor from multiple expressions

When you want a single-cube divisor: go looking for co-kernels
When you want a multiple-cube divisor: go looking for kernels

Multilevel Synthesis Models: Summary

- Boolean network model
  - Like a gate network, but each node in network is an SOP form
  - Supports many operations to add, reduce, simplify nodes in network

- Algebraic model & algebraic division
  - Simplified Boolean functions to behave like polynomials of real numbers
  - Lets you divide one Boolean function by another
  - function \( f = (\text{divisor } d) \ast (\text{quotient } q) + \text{remainder } r \)

- Kernels / Co-kernels of a function
  - Kernel = cube-free quotient got by dividing by a single cube
  - Intersections of kernels of 2 functions \( f, g \) are where all the interesting multiple-cube common subexpressions are to be found
  - Strong theorem here: Brayton-McMullen

- Still have to figure out what the right common factors are to have, given all this machinery...