(Lec 7) Multi-Level Minimize I: Models & Methods

- 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

Copyright Notice

Where Are We?

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

	M	T	W	Th	F	
Aug	27	28	29	30	31	1
Sep	3	4	5	6	7	2
	10	11	12	13	14	3
	17	18	19	20	21	4
	24	25	26	27	28	5
Oct	1	2	3	4	5	6
	8	9	10	11	12	7
	15	16	17	18	19	8
	22	23	24	25	26	9
	29	30	31	1	2	10
Nov	5	6	7	8	9	11
	12	13	14	15	16	12
Thnxgive	19	20	21	22	23	13
	26	27	28	29	30	14
Dec	3	4	5	6	7	15
	10	11	12	13	14	16

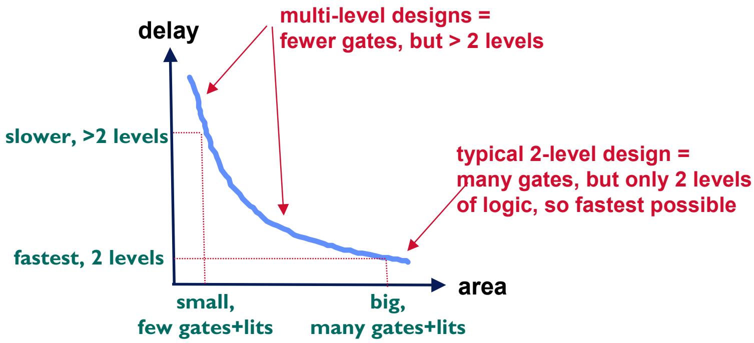
Introduction
Advanced Boolean algebra
JAVA Review
Formal verification
2-Level logic synthesis
Multi-level logic synthesis
Technology mapping
Placement
Routing
Static timing analysis
Electrical timing analysis
Geometric data structs & apps

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**

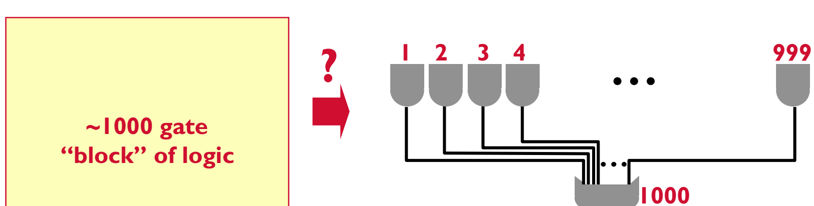
Why Multi-Level Forms

- 2-level too restrictive: specific area vs delay tradeoff
  - **Area = gates + literals (wires), ie, 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?

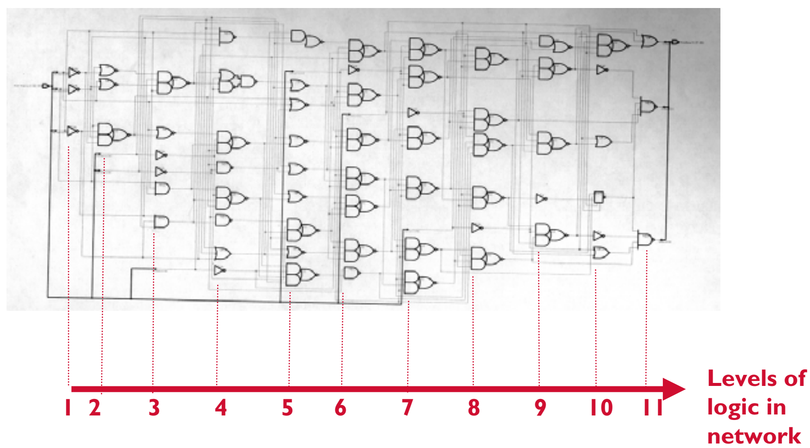
- Rarely see 2-level designs for really big things, mostly for pieces of bigger things
  - **Even smallish things routinely done as multi-level**



This is usually NOT going to be the preferred logic network structure…

Real MultiLevel Example

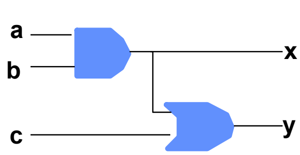
- ...and this is a pretty *small* design, done by Synopsys DesignCompiler



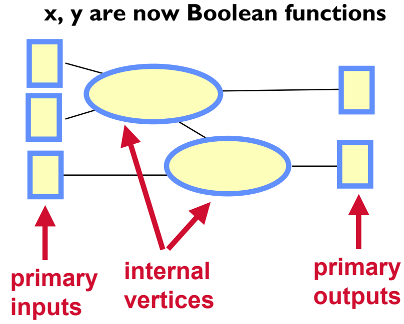
Boolean Logic Network Model

- 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 func's

Ordinary gate netlist



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



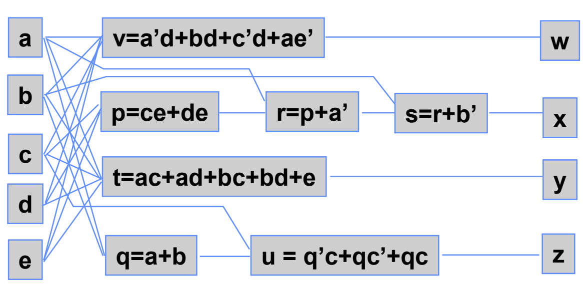
Boolean Logic Networks

- 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

Boolean Logic Networks

- Consider example from De Micheli
  - Let's look at some operations on this network…

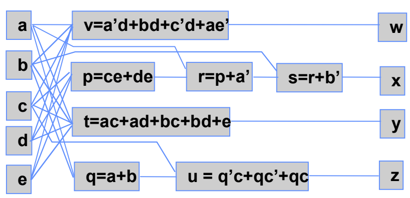
p = ce + de
q = a + b
r = p + a'
s = r + b'
t = ac + ad + bc + bd + e
u = q'c + qc' + qc
v = a'd + bd + c'd + ae'
w = v
x = s
y = t
z = u



Network Quality measure = $\sum_{nodes}$ (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 means "here are signals going into/out of these functions"
  - #literals = count up all lits in every SOP equation in every Boolean node



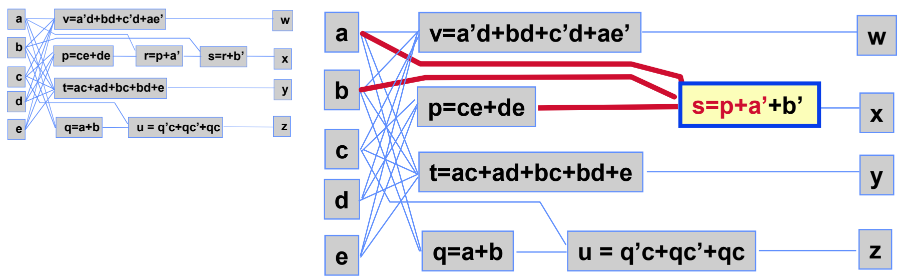
As gates it looks like this...

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

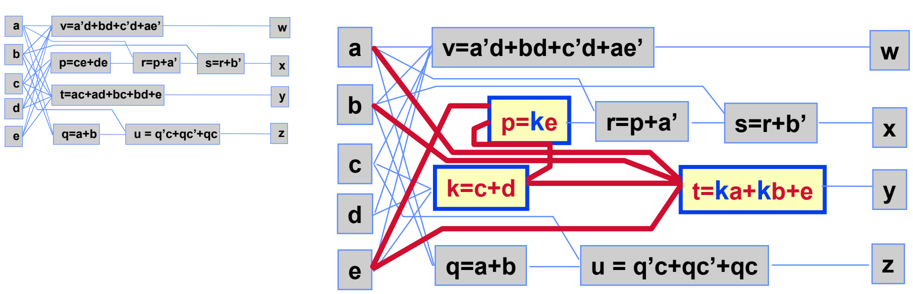
- **Reducing #nodes: Elimination**
  - **Removes** an internal vertex by replacing it (adding its SOP expression) into all the other vertices it feeds
  - Note: eliminate vertex for **r** requires substituting **(p+a')** in **s** node



Σlits =

Network Ops: Extraction

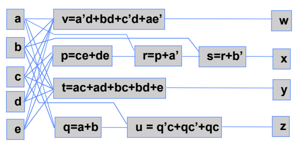
- **Adding nodes: Extraction**
  - **Create** a new vertex that represents a **common subexpression** for >= 2 vertices, and add it to network
  - Substitute the output of the new vertex for common parts elsewhere
  - Note that: p = (c+d) e and t=(c+d)(a+b) + e, so extract c+d

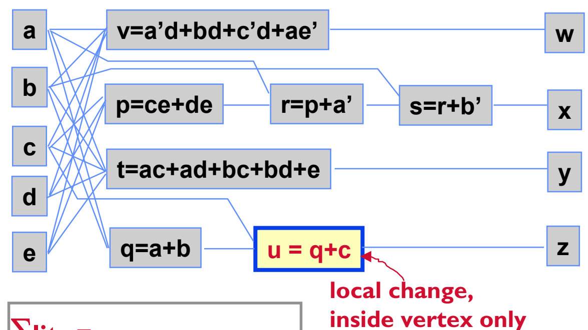


Σlits =

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





Σlits =

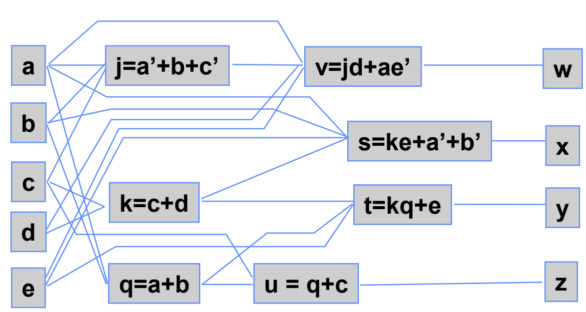
Network Ops: Iterative Improvement

- 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:

Literals

Before:

After:



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...?

Scripts

- 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
  - /afs/ece/class/ee760/sis is the binary for IBM and SUN

```
UC Berkeley, SIS Development Version (compiled 2-Nov-95 at 6:54 PM)
sis>
```

Command prompt
Type “help” to get a list of all commands

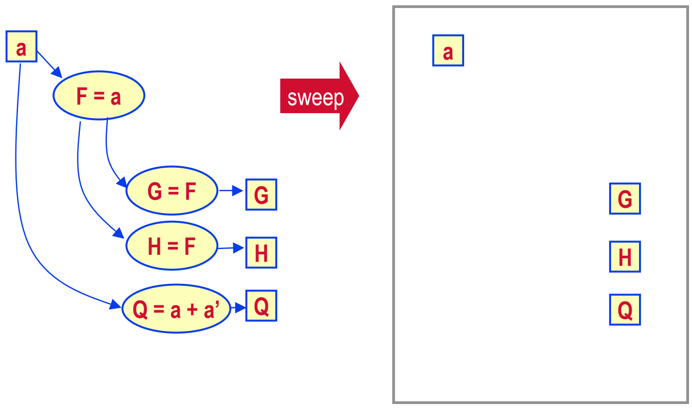
Rugged Ops: Sweep

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

```
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
```

Sweep Examples

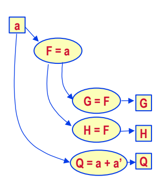
Sweep examples



Running sweep in SIS

SIS session

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



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.

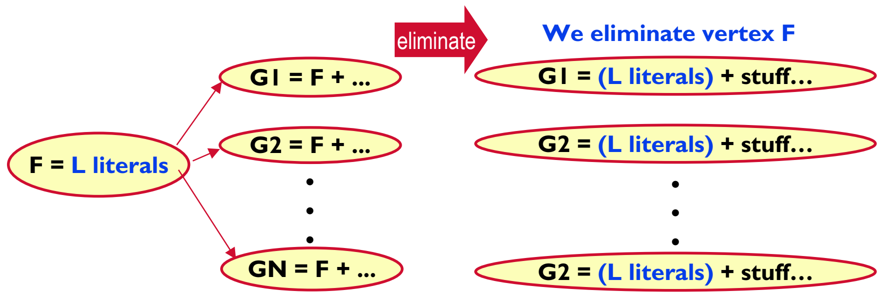
Network Ops: Eliminate

- 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

```
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
```

"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**



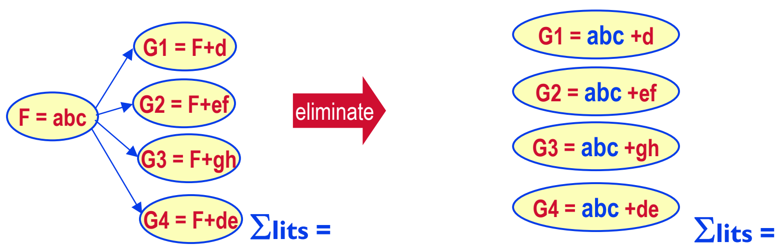
Total literals before =

Total literals after =
Change = value =

Eliminate Examples

Eliminate -1

Eliminate 5



Running eliminate in SIS

- SIS session

UNIX file: elim.eqn

```
F = a b c ;
G1 = F + d ;
G2 = F + e f ;
G3 = F + g h ;
G4 = F + de ;
```

```
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
```

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
```

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.

```
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
sis>
```

Now it does it.

G1 = abc +d
G2 = abc +ef
G3 = abc +gh
G4 = abc +de

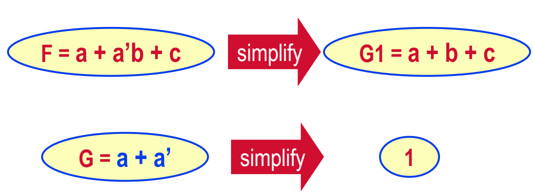
Network Ops: Simplify

- 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

```
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



Goal is just to "clean up" insides of each node in the Boolean network

Network Ops: Resub

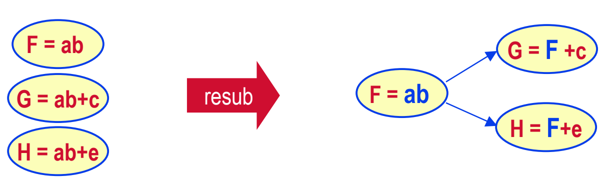
Resub -a

- 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", ie, 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...)

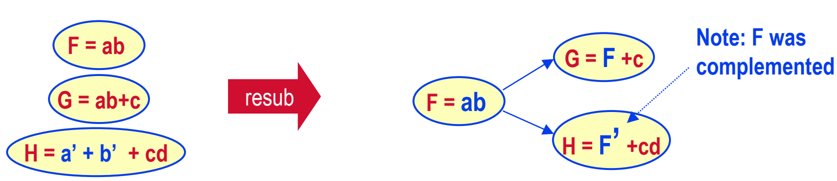
```
sweep; eliminate -1
simplify -m nocomp
eliminate -1

sweep; eliminate 5
simplify -m nocomp
resub -a

fx
resub -a; sweep

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


Resub Example

Resub example 1



Resub example 2



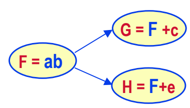
Running resub in SIS

SIS session

UNIX file: resub.eqn

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

```
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
```



Network Ops: Fx

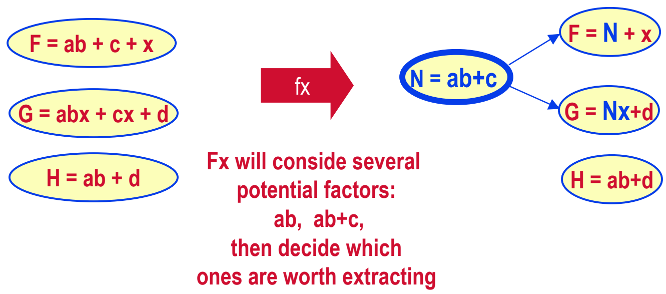
Fx

- 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

```
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
```

fx Example

fx example



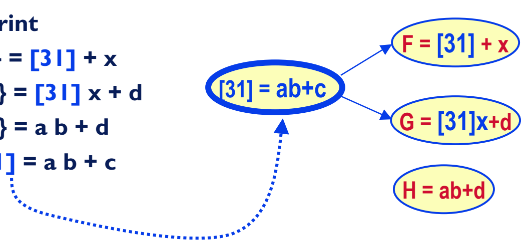
Running fx in SIS

- SIS session

```
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

```
F = a b + c + x;
G = a b x + c x + d ;
H = a b + d ;
```



resub != fx

- fx tries to find NEW common factors
  - It adds nodes to the network to do this
  - Tries to find good (usable) common subexpressions
- resub uses what is *already* in network
  - It CANNOT go find or "extract" new factors
  - It just looks at what nodes are already around in network
  - It tries to use these to substitute one node into another to save literals
- So….
  - Do fx first: create a bunch of good-looking common factors
  - Do resub next: try to use these factors to improve network

Rugged Script

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

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

```
sweep; eliminate 5
simplify -m nocomp
resub -a
```
First round of "easy" factoring

```
fx
resub -a; sweep
```
Second round of "aggressive" factoring

```
eliminate -1; sweep
full_simplify -m nocomp
```
Optimize each node aggressively

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

Real numbers		Boolean algebra	
a•b = b•a	SAME	a•b = b•a	
a+b = b+a		a+b = b+a	
a•(b•c) = (a•b)•c		a•(b•c) = (a•b)•c	
a+(b+c) = (a+b)+c		a+(b+c) = (a+b)+c	
a•(b+c) = a•b + a•c		a•(b+c) = a•b+a•c	
a•1 = a a•0 = 0		a•1 = a a•0 = 0	
a+0 = a		a+0 = a	
X	NOT ALLOWED	*a+a' = 1*	*a•a' = 0*
		a•a = a	*a+a = a*
		a+1 = 1	
		a+(b•c) = (a+b)•(a+c)	

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:

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

divisor (d)

quotient (q)

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

- (just like regular numbers, eg, 15 = 7 • 2 + 1)
- Boolean example

Algebraic Division

- Example

f = ac + ad + bc + bd + e   want f = d • q + r

Divisors (d)	*Quotient (q)*	*Remainder (r)*	*Factor?*
ac+ad+bc+bd+e			
a+b			
c+d			
a			
b			
c			
d			
e			

Algebraic Division

- 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•q+ r = D•(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

Algebraic Division Algorithm

- Algorithm

bugfix

```
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)
}
```

bugfix

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

A cube	D cube: ax C = …	D cube: b C = …
axc	axc	
axd	axd	
axe	axe	
bc		
bd		
e		
	Q =	Q =

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

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   B = ab + c'   want A / B

  - You must *transform* it to something like this…

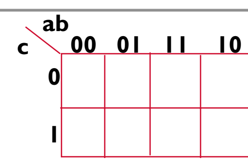
  - Because you **MUST** treat the true and compl forms of var as *different*

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"

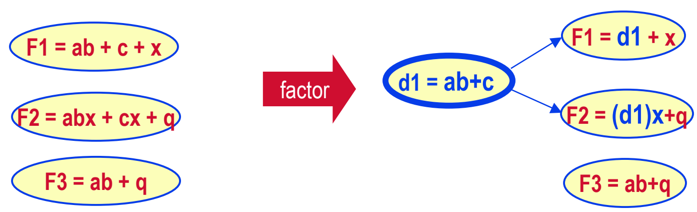
F = a + ab + bc   is redundant
D = a   is the divisor; we want to do F/D

now: compute F / D, ie, F / a
use our algebraic division algorithm…



Multilevel Synthesis Models: Where are We?

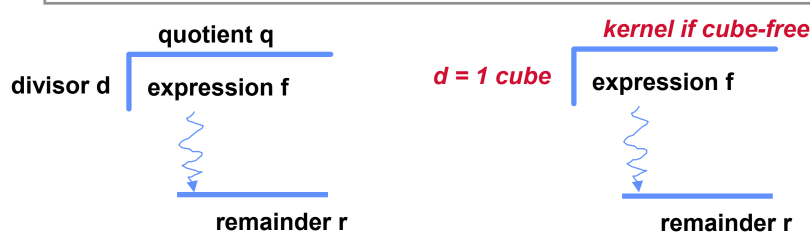
- Given Boolean A, D, you can compute A = Q*D + R easily
  - This is great—but its still not enough
  - Real problem: I give you n functions F1, F2, … Fn, and want to find a set of good *common divisors* di



- How to find?
  - Case 1: divisors d that are just 1 cube (1 product term), eg, d = ab
  - Case 2: "bigger" multiple-cube divisors, eg 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•q + r   f = d•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*q+r*	*Cube-free?*
a		
a+b		
ab + ac		
abc + abd		
ab + acd + bd		

Kernels

- Kernels of expression *f* denoted *K(f)*
  - Look at example f = abc + abd + bcd

Divisor cube d	*f= d • q + r*	*Is it a Kernel of f?____*
1	(1)(abc+abd+bcd)+0	No, has cube = b as factor
a		
b		
c		
d		
ab		
ac		
ad		
bc		
bd		
cd		
abc		
...		

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

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 k1 is a kernel of f, k2 is a kernel of g, we know that
  - Remember: k1, k2 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 $k1 \cap k2$ = (X + Y + ...)
- ...but now it's clear that $k1 \cap k2$ = (X + Y + ...) is a common, multiple-cube divisor! It's a nice, big common factor!

Kernels

- 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.

Kernels: Example

- Consider this f, g

f = ae + be + cde + ab

g = ad + ae + bd + be + bc

K(f) Kernel	Co-kernel
a+b+cd	e
b+e	a
a+e	b
ae+be+cde+ab	1

K(g) Kernel	Co-kernel
a+b	d or e
d+e	a or b
d+e+c	b
ad+ae+bd+be+bc	1

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

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

Kernels

- 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 k1 in *K(f)*

f = cube1 • k1 + remainder1

- First: a new, interesting question: *what about K(k1) ??*
  - k1 is a perfectly nice Boolean expression, so its got its own kernels
  - Do these kernels have anything *interesting* to say about K(f)...?

Kernels

- Look at *K(* k1 *)*
  - Suppose k2 is a kernel in *K(* k1 *)*, then we know
  - Substitute this in for k1 in original expression for f
  - Neat trick: cube1•cube2 is itself just another *single cube*, so rewrite to emphasize this fact:

Kernel Hierarchy

- So , what does this say?
  - k2 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…

Kernel Hierarchy

- 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 ∩ bce = ce => ce *is a potential co-kernel*

ace ∩ bce ∩ de = e => e *is a potential co-kernel*

Kernel Hierarchy

- 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…

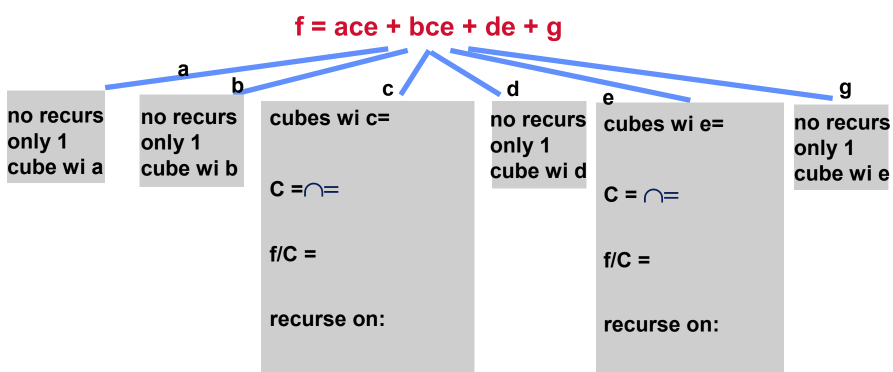
```
FindKernels( expression F) {
   K = null;
   for ( each variable x in  F  ) {
      if ( there are at least 2 cubes in F that have variable x ) {
         let S = { cubes in F that have variable x in them };
         let c = cube that results from intersection of all cubes in S,
                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

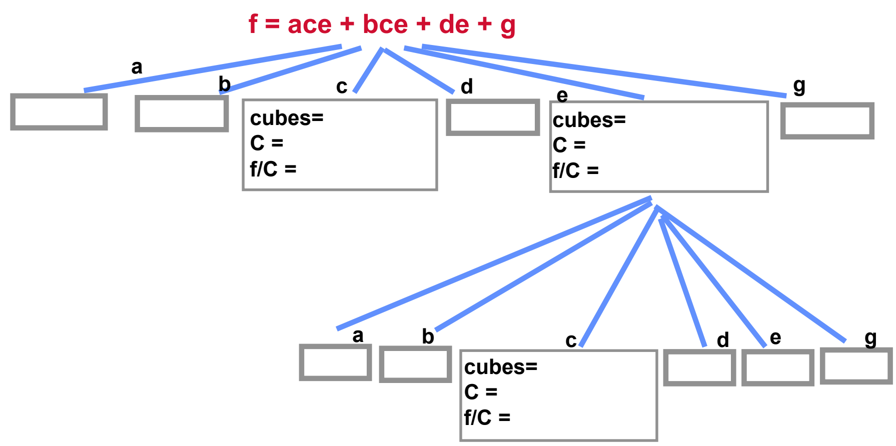
Kerneling Example

- 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



Kernel Hierarchy, Example Revisited

- With this algorithm, overall recursion tree looks like this

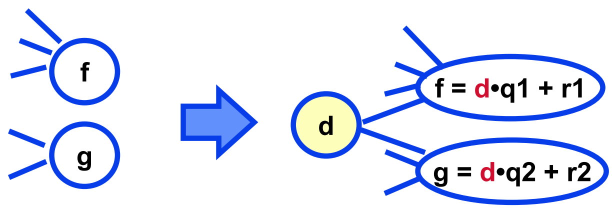


Kernel Hierarchy

- 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 = (divisor d)• (quotient q) + 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...