(*Lec 6*) 2-Level Minimization: Basics & Algs

What you know

- Some useful data structures to represent Boolean functions
  - Cube lists: represent a SOP form of a function
  - BDDs: represent function itself in canonical form
- A few important algorithms & applications
  - URP-style cubelist tautology, ITE on BDDs
  - Use of BDDs to see if 2 different networks or FSMs are same
- A new *way of thinking* about Boolean functions
  - Divide & conquer algorithms on data structures

What you don't know

- Algorithms to simplify (minimize) a Boolean function(s)
- Starting point is the classical 2-level form, sum of products

Copyright Notice

Where Are We?

- **Moving on to real logic synthesis--for 2-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

Handouts

- **Physical**
  - **Lecture 6 -- 2 Level Logic Minimization**
  - **Paper 1 – dynamic variable ordering for BDDs**
- **Electronic**
  - **HW3 will be on the web site this weekend**
- **Reminders**
  - **HW2 extended – Friday 5pm, my office (3105) or Lyz Knight's (3107)**
  - **Project 1 deadline will also get pushed back a little…**

Readings

- DeMicheli has a lot of relevant stuff
  - He actually worked on this stuff as a grad student at Berkeley
- Read this in Chapter 7
  - 7.1 Intro: take a look.
  - 7.2 Logic optimization principles: read 7.2.1-7.2.3 as background
  - 7.3 Ops on 2-level logic covers: read it but don't worry about 7.3.2
  - 7.4 Algorithms for logic minimization: read it, but focus on Expand and Reduce and the ESPRESSO minimizer.
  - 7.5-7.6 Skip.
  - 7.7 Perspectives: read it.
- Read this in Chapter 2
  - 2.5.3 Satisfiability and cover: gives some background about how people really solve covering problems of the type we talk about here

Readings

- If you are feeling especially macho here:
  - Robert K. Brayton, Gary D. Hachtel, Curtis T. McMullen, Alberto Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publishers, 1984.
  - The bible of ESPRESSO heuristics, from the designers of the algorithms and authors of the various early code implementations
  - Lots of good details.
  - Not for the timid.
  - (Knowing some APL would also help explain the mysterious notation.)
  - Another good one is: Gary Hachtel and Fabio Somenzi, *Logic Synthesis and Verification Algorithms*, Kluwer Academic Publishers, 1996.
  - We use some examples from both of these books.

2-Level Minimization

- What we assume you've seen is minimization by...
  - Boolean algebra
    - Pro: easy, just algebra
    - Con: hard to know when you have a good answer, hard with lots of vars & functions
  - Karnaugh Maps
    - Pro: easy, visual
    - Con: too complex past about 6-7 variables
  - Quine McCluskey
    - Pro: systematic algorithm, fairly easy
    - Con: complexity scales exponentially

- What's new here?

2-Level Minimization

- A little history...
  - 1950s: Classical approaches
    - Quine McCluskey approach showed that you could minimize things exactly, but complexity wasn't very good
    - Dropped from attention...

  - 1970s, early 80s: Heuristic approaches
    - Don't go after exact optimum solutions, just *good* solutions
    - Lots of progress, lots of attention
    - Most famous: ESPRESSO from Berkeley

  - 1980s-90s: New exact approaches
    - Now have good data structre (BDDs) to do complicated things
    - Clever new approaches to "exact minimization" that tended not to go exponential on practical test cases

2-Level Minimization: Focus

Current state of affairs

- Everybody uses BDDs for everything, everywhere...
- ..*except one place*: Heuristic 2-level ESPRESSO minimization
  - ESPRESSO hacks on cubelists
  - ESPRESSO is many, fairly complex heuristics
  - ESPRESSO is called in the inner loop of many other optimization tasks now, that need a fast, good, 2-level minimization as part of a bigger design task
- There are also several clever new exact algorithms
  - ...that use BDDs for the data structures
  - Tend to be slower than ESPRESSO, but guarantee the exact best answer possible

What will we look at...?

- A quick review of basics of 2-level logic minimization
- A quick tour of the ESPRESSO strategy, with details for just a few of the ESPRESSO heuristics

2-Level Minimization: Background

Exactly what is the goal here?

- Input: a truth table (with lots of don't cares)
- Output: minimized sum-of-products expression
- Minimum means, usually
  - Fewest product terms (each term == an AND gate, conceptually)
  - Fewest literals (each literal is a gate input)

cd \ ab	00	01	11	10
00	1	1		
01	1	1	1	1
11			1	1
10	1	1	1	1

OK, but not the best...

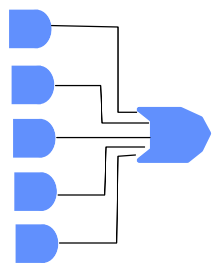
cd \ ab	00	01	11	10
00	1	1		
01	1	1	1	1
11			1	1
10	1	1	1	1

...best you can do.

2-Level Minimization

- Reminder: what's a "literal"
  - **Appearance of a variable in true or complemented form in an SOP expression for a function**
  - **A primary input on a gate**

f = ab' + ab'c + abc + bcd + bcd' has [ ] literals

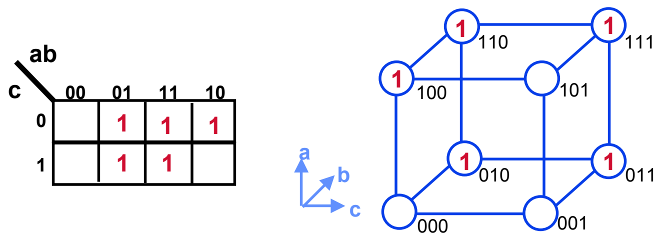


2-Level Minimization

- What is simplification really all about?
  - ***Generate*** **component pieces of the solution**
  - ***Select*** **which of these pieces are in the best solution**
- Component pieces = *Prime Implicants*
  - **Products terms with some specific properties**
- Pick which pieces you need = *Covering Problem*
  - **You don't want all of them, which ones are needed?**
- Need to review some terminology…

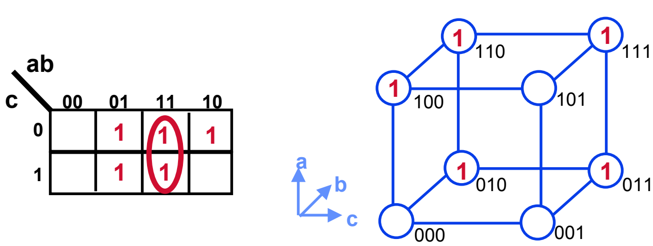
2-Level Minimization: Terms

- Aside:
  - Most useful to think of all the terms in cube-space, or on a Kmap
- What are component pieces of solution?
  - Term: Implicant
    - An implicant is any product term contained in your function
    - ...when the implicant is 1 ==> your function is 1
    - ...anything you can circle in a Kmap
    - ...any cube you can identify on a cube-plot of your function



2-Level Minimization: Terms

- What are component pieces of solution?
  - Term: Prime Implicant (PI)
    - An implicant with the property that if you remove any literal, it stops being an implicant
    - ...a circle in a Kmap you cannot make any bigger
    - ...a cube not contained in any other cube in your function



2-Level Minimization: Mastering Terminology

Aside

- Remember: a "cube" is just a "product term"
- Keep in mind how all the different "views" of what a product term is *relate* to each other for simplification...

A product term with________ literals = A Kmap group that circles ________ 1s = A cube that covers _____ vertices (minterms) = An AND gate with _______ input wires

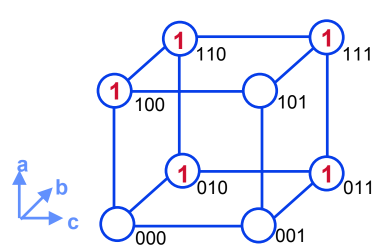
= ***A Good Thing***

2-Level Minimization: Terms

What are component pieces of solution?

- Term: Essential Prime Implicant
  - If there is a row of the truth table...
  - ...or a 1 in the Kmap
  - ...or a vertex of the cube where f==1
  - ..that is covered by *exactly one PI*, this PI is called essential

c \ ab	00	01	11	10
0		1	1	1
1		1	1	



2-Level Minimization: PIs

- **Big theorem from 50s**
  - **Due to Quine (of ...& McCluskey)**
    - **Sort of makes sense**
    - **If not, you could always take an implicant of function, expand it to become a Prime, and buy back a few literals**

c \ ab	00	01	11	10
0		1	1	1
1		1	1	

- **Consequence**
  - **All we really need to work with is the PIs, that's what the solution to 2-level minimization will be made out of**

2-Level Minimization: Aside

cd \ ab	00	01	11	10
00			1	1
01	1	1		
11			1	1
10	1	1		

- **Sometimes** ______ **is essential**

cd \ ab	00	01	11	10
00			1	1
01		1	1	
11	1	1		
10	1			1

- **Sometimes** ______ **is essential**

- **In general, a *mix* of essential and inessential PIs**

2-Level Minimization

- **First systematic minimization: Quine McCluskey**
  - Generate all the PIs
    - Simple, exhaustive pairwise comparision technique
    - Start with each minterm, try to see how far you can "grow it"
  - Transform into a **covering** problem

cd \ ab	00	01	11	10
00		1	1	
01		1	1	
11	1		1	1
10				

	m3	m4	m5	m11	m12	m13	m15
bc' = P1		I	I		I	I	
abd = P2						I	I
acd = P3				I			I
b'cd = P4	I			I			

a b c d
bc' = – I 0 – so covers:

2-Level Simplification: QM

- **What's a "*covering problem*"?**
  - Somebody gives you a matrix with 0s and 1s in it
  - You must pick a set of rows...
    - ...that guarantees that each column is "covered"
    - ..means there is a row with a "1" in that column in your set
  - ...that minimizes some cost, ie, each row "costs" something, so want to choose the "cheapest" rows to cover all the columns

	m3	m4	m5	m11	m12	m13	m15
bc' = P1		1	1		1	1	
abd = P2						1	1
acd = P3				1			1
b'cd = P4	1			1			

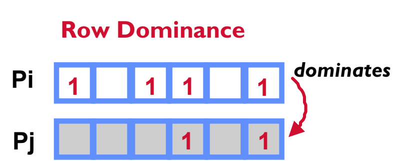
2-Level Minimization: Coverings

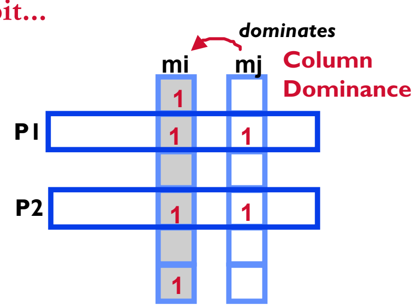
- How do you solve a covering problem?
  - With difficulty -- it's exponentially hard in general
  - But there are tricks to help, to exploit problem structure
- Reduction techniques
  - Try to make the covering matrix smaller
  - Example: find the essential PIs:
    - Look for **columns** with a single 1 in them
    - ...the **row** of that single 1 is an essential PI
    - Cross out the row (it must be in solution, no need to search for it) and all columns with 1s in this row (these are covered minterms, no need to try to cover them elsewhere)

	i	j	k	l
P1	1		1	1
P2		1	1	
P3		1		1

2-Level Minimization: Dominance Relations

- Also row, col patterns to exploit...





Pi has a 1 in every place Pj does, and a few more

Keep Pi, kill Pj. Why?

Pi covers all the same minterms and it's bigger so it's cheaper. So we'd never pick Pj. Kill Pj row.

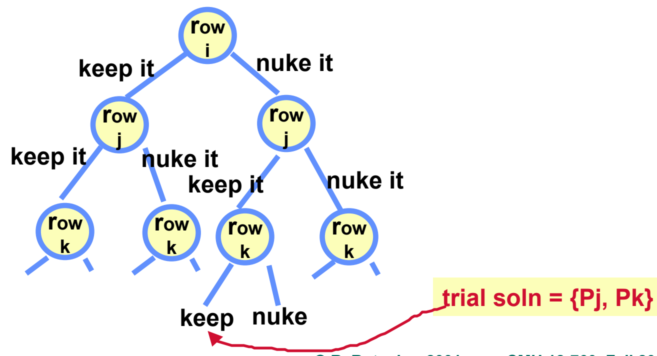
mi has a 1 in every place mj does, and a few more

Cross out col **mi** ignore it. Why?

If you pick PIs to cover mj, you will also cover mi, so stop worrying about mi, just try to cover mj.

2-Level Minimization: Covering

- Does this always work? NO!
  - Sometimes you go till you cannot reduce further and you still have a nontrivial table you cannot just read answer off of
  - Now what? Do Combinatorial Search
    - Explore a search tree, each child of each node is a different decision you need to try
    - Techniques to prune search quickly: branch & bound



2-Level Minimization: QM

- What's wrong with this approach?

#1 PI enumeration is very slow
  - You build them up from minterms, exhaustively checking each evolving implicant against others to see if you can expand till prime
  - Why is this a problem
    - A "nasty" problem has zillions of primes

#2 Exact covering using this exhaustive search is very slow
  - You already have a zillion PIs
  - Doing an exhaustive search to get exact right set of PIs to cover the minterms is exponential in number of PIs…
  - …and number of PIs is itself enormous in general

2-Level Minimization: Strategies

- So, what do people actually do?
  - Heuristic minimization
    - Don't generate all the PIs explicitly, then do exact cover
    - Instead, generate some cover of the function, then iteratively improve it

1 Generate *some cover* of function f

2 Iteratively *improve, reshape* this cover

3 if (it's still getting better) goto 2

4 clean up final answer, quit

2-Level Minimization: Covers of F

- Reminder: function vs *cover* of function
  - We've been sloppy so far here not to distinguish these
  - A function != cover

- *Cover* of a function
    - In SOP style, this is what set of implicants (product terms) you will actually use to implement your function
    - ...it's the set of groupings circled in your Kmap
    - ...it's what gets implemented as AND gates in real hardware
    - ...it's what a cubelist represents (each cube==product)
    - ...it's NOT what a BDD represents!

2-Level Minimization: Cover vs Function

- Remember
  - BDDs represent the function itself, not the gate implementation
  - Cubelists represent only a particular implementation

A function...

x_1	x_2	x_3	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

3 different covers of the function...

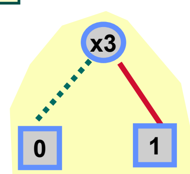
= x1'•x2'•x3 + x1'•x2•x3 + x1•x2'•x3 + x1•x2•x3

= x1'•x3 + x1•x3

= x3

3 different cubelist representations

x1 x2 x3	x1 x2 x3	x1 x2 x3
[11 11 01]	[10 11 01]	[10 10 01]
	[01 11 01]	[10 01 01]
		[01 10 01]
		[10 01 01]



Only 1 BDD for all of these

2-Level Minimization: Focus

- What do people *really* do today?
- Heuristic minimization *a la* ESPRESSO
  - About only place left people still use cubelists
  - Lots of URP algorithms
  - We'll look a few in detail, not all
- Exact minimization
  - Slower than ESPRESSO, but exact optimum answer
  - Actually, the tricks are to avoid generating PIs explicitly, and to avoid generating ones you know early won't be in the final cover
  - Data structures are actually BDDs here, usually
  - We won't talk about these (though there are some very elegant algorithms here, and lots of interesting current work...)

2-Level Minimization: ESPRESSO Heuristics

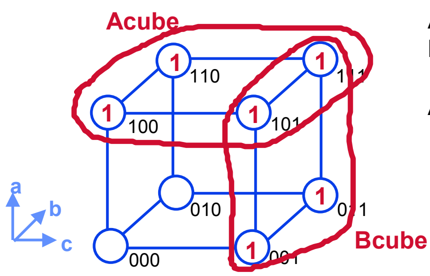
- What we just reviewed here
  - 2-level minimization "basics"
    - Minimum solutions are made out of *prime implicants*
    - Finding the best set of PIs is intrinsically a *covering problem*
    - It's usually too expensive to generate all PIs and search exhaustively for the best cover
- What you don't know (yet)
  - Heuristics that avoid the explosion-of-PIs problem
  - ESPRESSO: most successful heuristic, from IBM / Berkeley
    - The "reduce-expand-irredundant" loop
  - Some more basic tools for doing this
    - More operators on covers of functions represented as cubelists
    - More useful properties of covers of Boolean functions

2-Level Minimization: PCN Revisited

- Positional Cube Notation (PCN)
  - Recall basics: for each cube (product term)
    - 1 slot per variable, 2-bits per slot
    - 01 == var 10 == var' 11== var not in product 00==*void*
  - Cube list represents a cover of a function f
    - Just a list of cubes, one cube for each product in SOP cover
- Useful properties
  - Reasonably small: n vars => 2n bits/cube
  - Already saw that it's fairly simple to do some things
    - Cofactor, etc
- There are, in fact, a bunch of *other* operators...

2-Level Minimization: PCN

- One nice reason for the bit-oriented format
  - Boolean ops on cubes -- AND, OR, etc -- actually meaningful
- Operator: Cube Intersection
  - Regard each cube as a set of (appropriately adjacent) minterms
  - Cube intersection ==

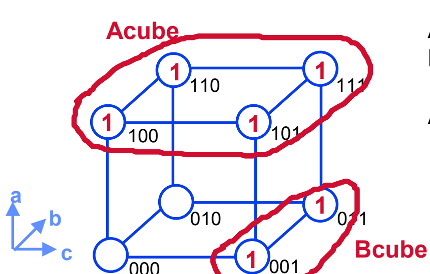


Acube = [01 11 11] = a
Bcube = [11 11 01] = c

AND =

2-Level Minimization: PCN

- What happens when cubes don't intersect?
  - One of the fields in the AND is void == 00
  - 00 means "nuke this cube, it can't exist!"

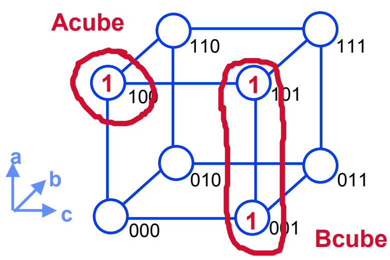


Acube = [01 11 11] = a
Bcube = [10 11 01] = a'c

AND =

2-Level Minimization: PCN

- OK, so what does bitwise OR do...?
- Operator: Supercube
  - Supercube(Acube,Bcube) = ?
    - Another cube...
  - Supercube = bitwise OR

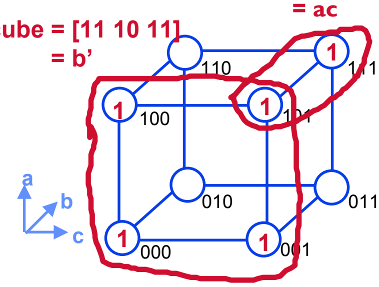


Acube = [01 10 10] = ab'c'
Bcube = [11 10 01] = b'c

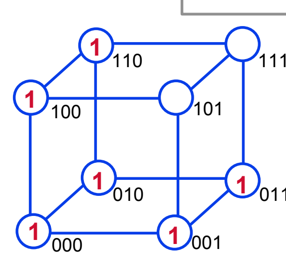
OR =

2-Level Minimization: PCN

- More sophisticated ops: Acube # Bcube (called 'sharp')
  - In English
  - Say we "sharp off" parts of Acube that are in Bcube (see HW)



Acube # Bcube =



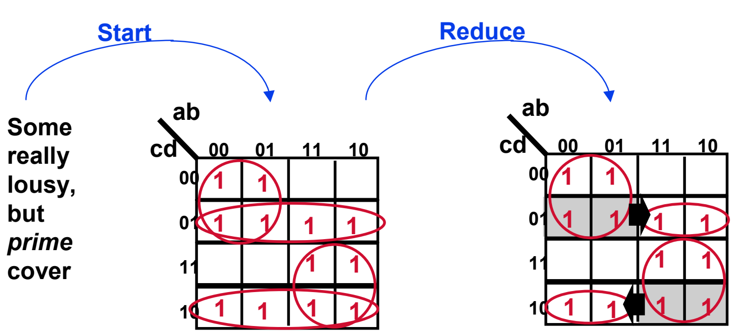
Heuristic 2-Level Minimization

- OK, where are we?
  - We have a data structure: Cubelists in PCN
    - Good for representing covers of functions in SOP form
  - We have several useful operators
    - Intersect, supercube, # [*this is a HW problem*]
  - We have URP as a basic algorithm style for attacking things, ie, Recursive divide & conquer based on Shannon factorization for
    - Tautology, cube containment in a cover [*this is also a HW problem...*]

- What don't we have?
  - Overall strategy for heuristic minimization
  - == ESPRESSO

Real 2-Level Synthesis: Basic Style

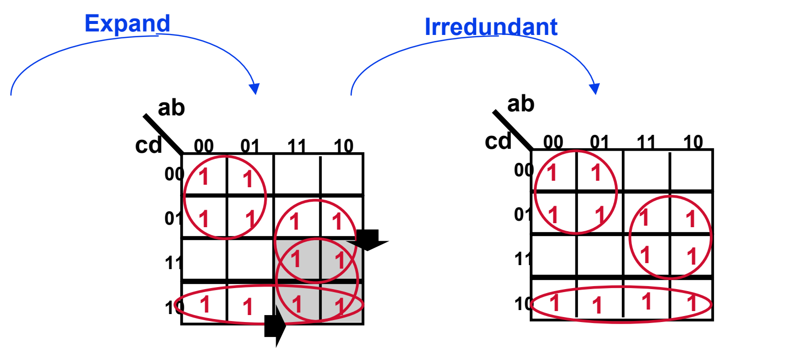
- ESPRESSO *style*
  - Start with any prime cover...
  - Repeat these operations: reduce, expand, irredundant



4 PIs

4 implicants,
but now maybe *not* prime
cover has been *reshaped*

Real 2-Level Synthesis



4 PIs,
one is *redundant*

3 PIs, now none redundant

- Need more terminology to explain strategy here…

Properties of Covers

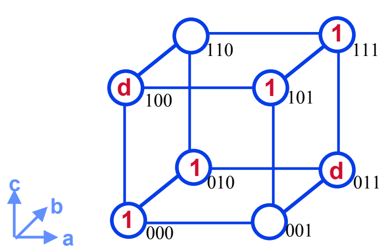
- What do we start with?
  - Truth table with don't cares
  - Represent as sets of minterms; standard names for these:

a b c	f
0 0 0	1
0 0 1	0
0 1 0	1
0 1 1	d
1 0 0	d
1 0 1	1
1 1 0	0
1 1 1	1

F^{ON} = ON-set = { } = where f == 1

F^{OFF} = OFF-set = { } = where f == 0

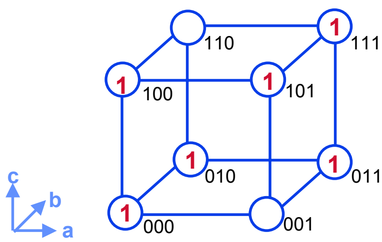
F^{DC} = Don't care set = { } = where f == d



  - Ultimately, we manipulate cubelist-style covers for F^{ON}, F^{OFF}, F^{DC}

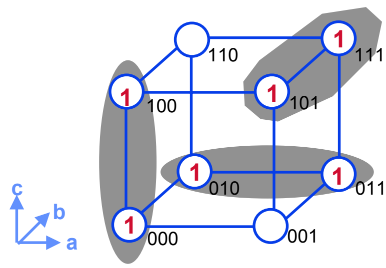
Properties of Covers

- **Types of covers**
  - **Minimum: has fewest *PIs*, and among all covers with same number of PIs, this one has the fewest *literals***
  - **This is the best you can do…**



ON-set of f
(skip don't cares for now…)

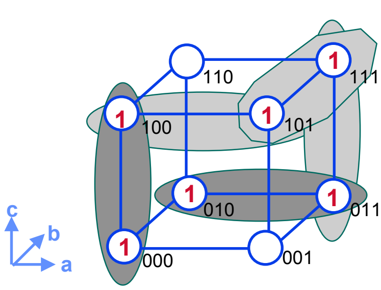
c \ ab	00	01	11	10
0	1	1		1
1		1	1	1



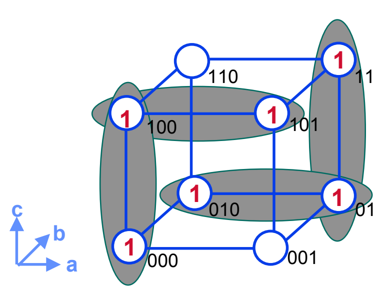
Minimum cover of f

Properties of Covers

- **Types of covers**
  - **Minimal Irredundant: this cover not a proper *super*set of any other cover**
  - **In English: can't remove any cube and still have a cover**
  - **Not as good as a minimum cover, "weaker" statement about quality of the cover of the function.**



Redundant cover



Minimal irredundant cover
(but it's not *minimum*)

Properties of Covers

- **Hierarchy of "goodness" in covers**
  - **Prime cover better than nonprime cover**
  - **Irredundant is better than an arbitrary Prime cover**
  - **Minimum is better than Irredundant**
  - **Think about these like this:**

- **Why are we doing this?**
  - ***Minimum* is hard to get...**
  - **....but we can *aim* for Minimal Irredundant**
  - **If we get lucky we'll get a Minimum; if not, we're probably close.**

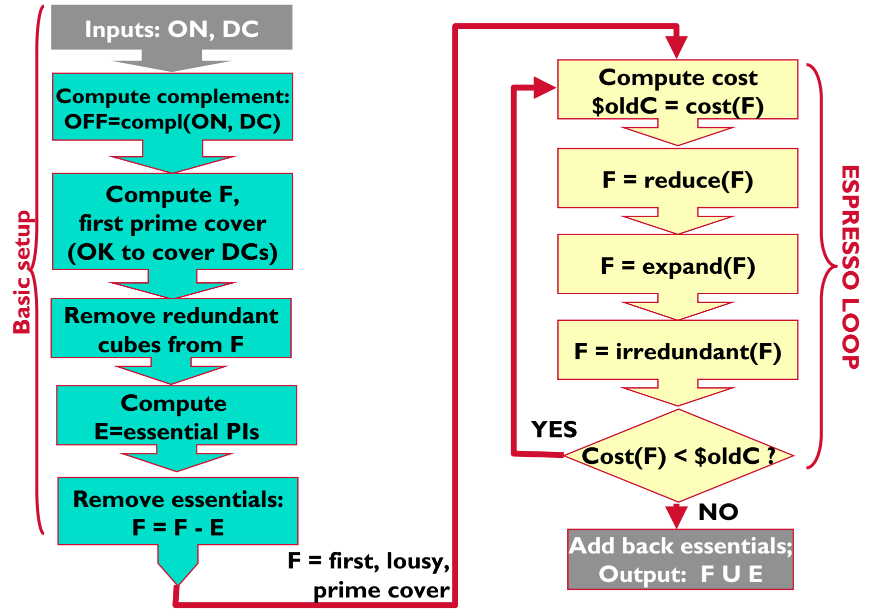
ESPRESSO Loop: Details of the Strategy

- **Iteratively reshapes a cover**
  - **A (somewhat) simplified version of algorithm**

```
ESPRESSO (F^ON, F^DC ) {
  F^OFF = complement(F^ON U F^DC);     // get cover of OFF-set
  F = expand(F^ON, F^OFF);             // get first cubelist cover of function f...
                                       //    ...OK to cover some don't cares
  F = irredundant(F^ON, F^DC)          // get rid of redundant cubes from expand()
  E = essentials(F, F^DC);             // find essential primes, remember them
  F = F - E;                           // take essentials out of F, we don't need
                                       //     to try to look later for covers of these

 // ESPRESSO loop
  do {
    $C = cost(cubelist for F);         // count literals and cubes
    F = reduce(F, F^DC);               // shrink this cover...
    F = expand(F, F^OFF);              //...then regrow some PIs--maybe improve
    F = irredundant(F, F^DC);          //  get rid of redundant cubes in F
  } while( cost(cubelist for F) < $C)  // ...ie, while things are getting better
  return( F U E );                     // put back essential PIs
}
```

ESPRESSO-Loop: the “Big Picture” View



ESPRESSO Loop: What Will We Cover ...?

1. How you do *complement*, why you need it.

```
ESPRESSO (F^ON, F^DC ) {
   F^OFF = complement(F^ON U F^DC);
   F = expand(F^ON, F^OFF);

   F = irredundant(F^ON, F^DC)
   E = essentials(F, F^DC);
   F = F - E;

   do {
      $C = cost(cubelist for F);
      F = reduce(F, F^DC);
      F = expand(F, F^OFF);
      F = irredundant(F, F^DC);
   } while( cost(cubelist for F) < $C)
   return( F U E );
}
```

Skip this one. (essentials)

2. Simplified version of how *expand* works

3. Just mention what *reduce* does, not how.

4. Just mention what *irred*. does, not how.

ESPRESSO Ops: Complement

- Input
  - Cube list F that covers function f (we'll ignore the don't cares)
- Output
  - Cube list that covers complement of function F, i.e., a cover of F'
- Why do we need it?
  - We will use it in expand() -- this tells us where we *cannot* expand cubes
- Strategy
  - A lot like URP tautology, only with different rules

Complement

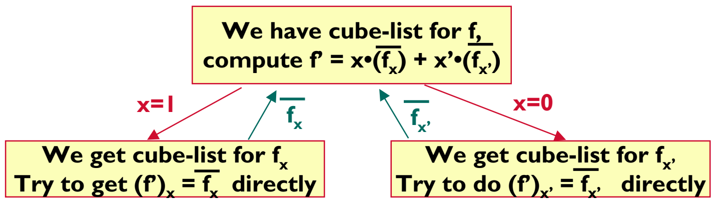
- Basically same as URP tautology
  - Critical observation

$$f' = x \cdot (f_x) + x' \cdot (f_{x'})$$

  - Just like with tautology, try to complement the function if you can, if you can't -- you cofactor and try on the simpler pieces
- As before, we need
  - Splitting rule: which variable do we pick to cofactor?
  - Termination rule: how do we actually complement at the leaves?

URP Complement

- There are again several useful termination rules



- Examples
  - If f_x or $f_{x'}$ == all don't care cube (==1) then return complement ==“0”
  - If every cube in f_x or $f_{x'}$ has one variable in same polarity, say “y”, eg

yzw = y•(zw + z' + wv) => complement is $y' + \overline{(zw + z' + wv)}$
yz'
ywv => return [y' + URPcomplement(zw + z' + wv)]

URP Complement: Role of Unateness

- Turns out unateness helps again
  - Splitting rule: Pick most not-unate variable
  - Try to get the leaves to be unate -- why?
  - We actually did this on HW1:

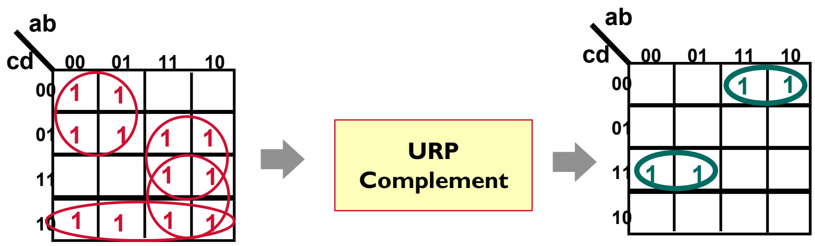
Positive unate in x: $\overline{f} = \overline{f_x} + \overline{x}\,\overline{f_{\overline{x}}}$

Negative unate in x: $\overline{f} = x\,\overline{f_x} + \overline{f_{\overline{x}}}$

Get to eliminate the x variable entirely, which is a little simpler

URP Complement

- What it does
  - It gives us a cube-list cover of the OFF set, which expand() uses



We start with a cover of the ON set...

We get a cover of the OFF set... it may not be very good (minimal) but that's OK

ESPRESSO Ops: Expand

- Input
  - Cube list F that covers function f
- Output
  - Cube list that covers f, with each implicant as big as possible, i.e. *prime*
- Strategy

```
assign each cube Ci a priority weight wi
for (each cube Ci in priority order from small to large) {
    determine which vars in cube Ci we can remove;
    remove these vars to make the cube a bigger expanded cube
}
```

Aside: What Does "Expanding A Cube" Mean?

Suppose we start with this

zw \ xy	00	01	11	10
00			1	1
01			1	1
11			1	
10			1	

cube = xyz'w'
= [01 01 10 10]

zw \ xy	00	01	11	10
00			1	1
01			1	1
11			1	
10			1	

cube = xz'w'
= [01 11 10 10]
= y -> don't care
= "raised y in cube"

zw \ xy	00	01	11	10
00			1	1
01			1	1
11			1	
10			1	

cube = xz'
= [01 11 10 11]
= w -> don't care
= "raised w in cube"

Expanding Cubes

- NOTE: we cannot keep expanding this cube forever...
  - We eventually make a cube that covers 0s as well as 1s
  - Such a cube is called infeasible -- it tries to cover a cube in OFF set.

zw \ xy	00	01	11	10
00			1	1
01			1	1
11			1	
10			1	

cube = z'
= [11 11 10 11]
= raised x
= INFEASIBLE

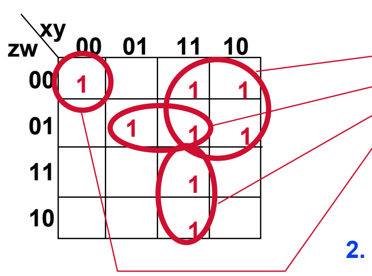
zw \ xy	00	01	11	10
00			1	1
01			1	1
11			1	
10			1	

cube = x
= [01 11 11 11]
= raised z
= INFEASIBLE

Expand: Ordering Cubes

- **First problem: what *order* to expand cubes?**
  - Order obviously makes a difference in final answer!
- **Strategy**
  - Weight the cubes
  - Sort the cubes on weight number
  - Expand cubes in this sorted order: *light to heavy*
- **Idea**
  - Cube is "*light*" if it is *unlikely* to be covered by other cubes
    - Light cubes cover fewer minterms...
    - ...don't have so many don't cares in PCN slots
    - Example: (ab'cd) is lighter than (ac)
  - Heuristic:
    - Add up all the 1s in each column of cube cover; big num means lots of vars in this polarity (or don't cares)
    - Look for cubes that have few 1s in these dense columns

Expand: Weighting Cubes




```
xz'       01 11 10 11
yz'w      11 01 10 01
xyz       01 01 01 11
x'y'z'w'  10 10 10 10
------------------------
sum       [          ]
```

1. First do per column per bit sums

2. Transpose this vector

4 rows x 8 columns

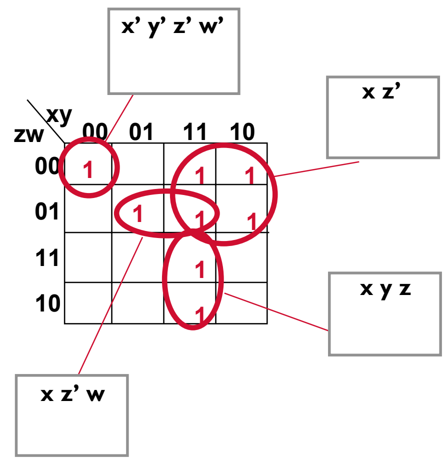
```
01 11 10 11       2
11 01 10 01   *   3   =   0*2+1*3+1*2+1*3+1*3+0*1+1*3+1*3       [ ]
01 01 01 11       2       1*2+1*3+0*2+1*3+1*3+0*1+0*3+1*3   =
10 10 10 10       3       0*2+1*3+0*2+1*3+0*3+1*1+1*3+1*3
                  3       1*2+0*3+1*2+0*3+1*3+0*1+1*3+0*3
                  1
                  3
                  3
```

3. Do Matrix multiply

4. Result = weights

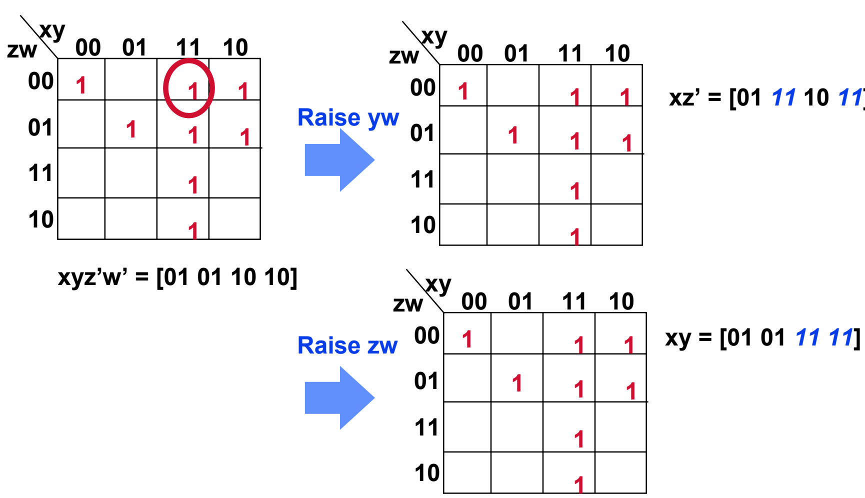
Expand: Weighting Cubes

- Small num = *light* cube
  - These cubes have vars where others have don't cares or vars of opposite polarity
  - **Sort** by ascending weight
  - **Expand** in ascending order



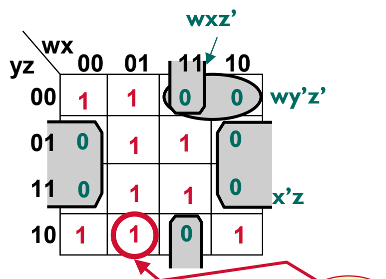
Doing Expand on a Cube: Which Vars to Raise?

- Given a cube to expand, which vars do we turn into don't cares? May be several different possible answers... Called "raising" the variables.



Expand: the Blocking Matrix

- **Turn this into yet-another-covering problem**
  - Make a small binary matrix called the "blocking matrix"
  - One row for each variable in the cube you are trying to expand
  - One column for each cube in the cover of the OFF set
  - Put a "1" in the matrix if the cube variable (row) != polarity of the var in the cube (column) of the OFF cover; else "0". If don't care, it's a "0"



f = w'x'z' + xz + x'yz' + w'xyz'

$\overline{f}$ = x'z + wxz' + wy'z'

expand this cube

cubes in $\overline{f}$ cover (may be a lot...)

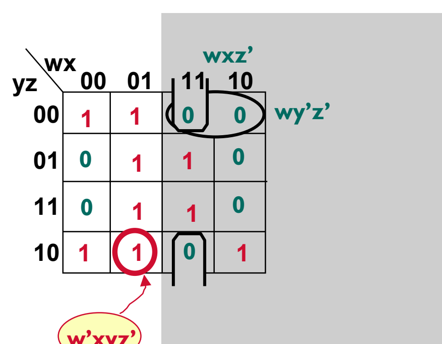
vars in expand cube

What "Blocking" Means

- **Just look at first row, where var = w' in expand cube**

cubes in f' cover

vars in expand cube	x'z	wxz'	wy'z'
w'	0	1	1
x			
y			
z'			



- Blocking matrix captures idea of which cubes of OFF set will BLOCK you from raising the variable
- It's not just "if we raise this one w' var we will hit this one OFF cube" but all the stuff in the OFF set you *could* hit if you raise *other* vars, too.

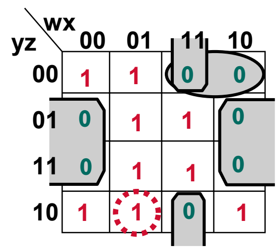
Solving the Right Covering Problem Here

- What "cover" do we want here?

cubes in f' cover (rows: vars in expand cube)

vars in expand cube	x'z	wxz'	wy'z'
w'	0	1	1
x	1	0	0
y	0	0	1
z'	1	0	0

2 solutions

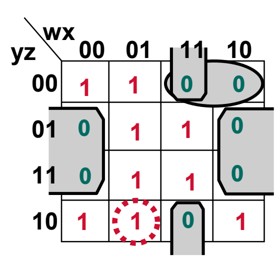


Why the Blocking Matrix Works

- It guarantees no "expanded" parts of your cube get blocked
  - You pick rows -- vars -- to keep that cover the columns
  - So, variables you keep all mutually **DO NOT HIT** any cubes in OFF set
  - When you AND these vars, the single product term -- bigger cube -- you get also **DOES NOT HIT** any of the cubes in OFF set

cubes in f' cover (rows: vars in expand cube)

vars in expand cube	x'z	wxz'	wy'z'
w'	0	1	1
x	1	0	0
y	0	0	1
z'	1	0	0



Solution is:
raise x, y
keep: w'z'
as bigger cube

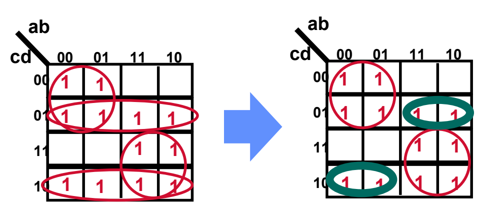
Solving the Covering Task on Blocking Matrix

- Use fast, non-optimal heuristics
  - **Need to do this quick, since you do it a lot--for every cube being expanded in the cover of function f inside expand()**
- Use simple, greedy heuristics…
  - **...ie, at each step, pick the row with the most 1s in it, etc**
  - **Also, can use some simple essential / dominance rules like from Q-M**
    - **Gotta pick the row associated with a column with a single 1 in it**
    - **Can do simple row and col dominance tricks to reduce the size**
  - **What you DON'T do is aggressive search with backtracking--no time**

ESPRESSO Ops: Reduce

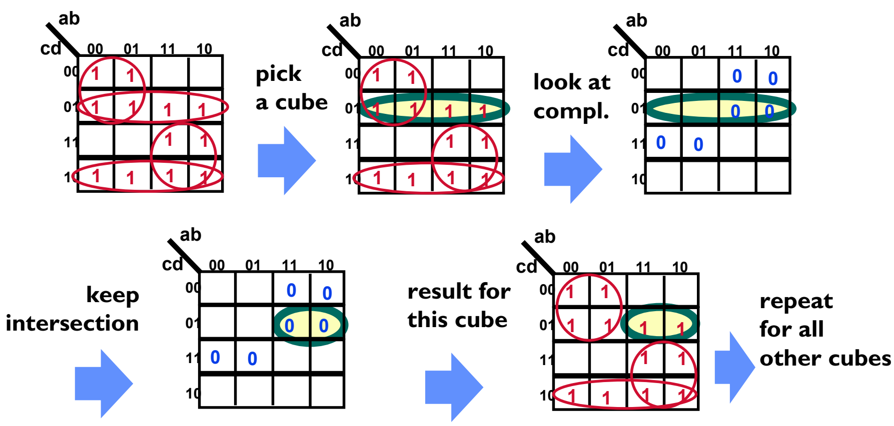
- Input
  - **Cube list F that covers function f**
- Output
  - **Cube list that covers f, with each implicant reduced--maybe not *prime*–so that no implicants overlap on any minterms**
- Strategy

```
for (each cube C in F, now from heavy to light) {
    intersect C with the rest of the cover F
    remove from C the minterms covered elsewhere
    find the biggest cube that covers this "reduced C"
    replace C with this reduced cube
}
```



Reduction

- **Basic trick**
  - Pick a cube, and remove it from your current cover of f…
  - …carefully intersect it with the complement of the *rest* of this cover
  - …minterms in this intersection are what you want to keep
  - …repeat on next cube



© R. Rutenbar 2001 CMU 18-760, Fall 2001 63

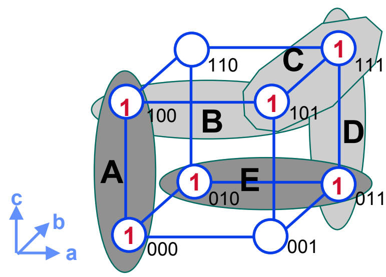
ESPRESSO Ops: Reduce

- **Basic strategy**
  - Weight cubes like expand() does…
  - …but now process heavy to light
    - Heavy cube covers *lots* of minterms…
    - …so better chances for reduction of heavy cubes first
  - Process cubes one at a time, in this heavy-to-light order
- **What does *Reduce* do…?**
  - Starts with a prime cover…
  - *…and shrinks individual primes in it*
  - You still get a cover of the function but it's probably not prime anymore
  - Big idea: this is a good starting point to do expand again, to regrow cubes in a different direction

© R. Rutenbar 2001 CMU 18-760, Fall 2001 64

ESPRESSO Ops: Irredundant

- Need a little terminology, *again*
  - Look at a little example cover F = {cubes A, B, C, D, E}
- Now, for a particular *fixed* cover…



Redundant cover

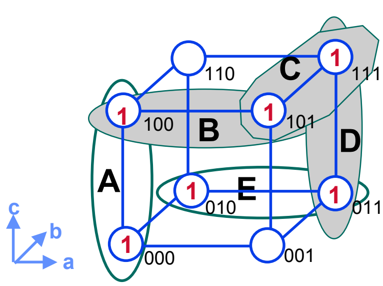
Relatively essential PIs = [ ]
These cover minterms not covered by another cube *in this particular cover*

Totally redundant PIs = [ ]
These are cubes in F covered by the relatively essential PIs -- nuke them

Partially redundant PIs = [ ]
These are what's left over of F

ESPRESSO Ops: Irredundant

- Goal of irredundant() here is what??
  - Know we need to keep {A, E}, "relatively essential" in *this* cover
  - Which of {B,C,D} can we nuke and not uncover any minterms?



What's covered by partially redundant PIs = {B,C,D} (relatively essentials unshaded for clarity)

What can we get rid of and not uncover any minterms of f?

Expected answer:

ESPRESSO Ops: Irredundant

- What irredundant does
  - It chooses which of these partially redundant PIs to get rid of to reduce the size of the cover
- How ESPRESSO does NOT do it
  - Cube by cube, ie, like expand() and reduce(), which use cube weighting
  - You could go thru the cubes in order, and ask "can I get rid of this cube? is it covered by the rest of the cubes?"
  - It works, but not too well
- How ESPRESSO does it
  - Yet another covering problem
  - You get a matrix of 0s and 1s and you do a heuristic cover on it
  - Turns out this "more global" view of the problem, which looks at all the cubes simultaneously, gives much better answers

How Well Does All This Work...?

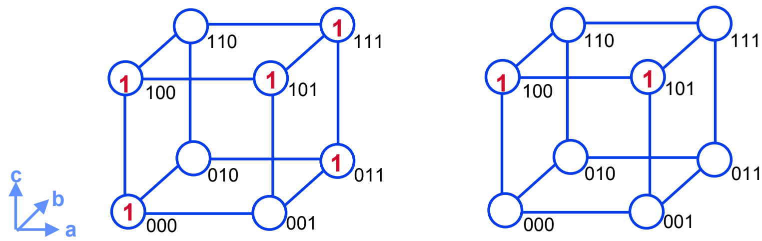
- Fabulous
  - Everybody uses ESPRESSO. Really fast, really robust
- Where does ESPRESSO spend its time?

Complement	14%	(big if there are lots of cubes in cover)
Expand	29%	(depends on of size of complement)
Irredundant	12%	
Essentials	13%	
Reduce	8%	
Various optimizations	22%	(special case, "last gasp" optimizations)

- How fast?
  - Usually does less than 5 expand-reduce-irredundant loop iterations; often converges in just 1-2 iterations.
  - Example result: minimized SOP with 3172 terms, 23741 literals, in roughly 16 CPU seconds on a ~10 MIP machine (in 1984...)

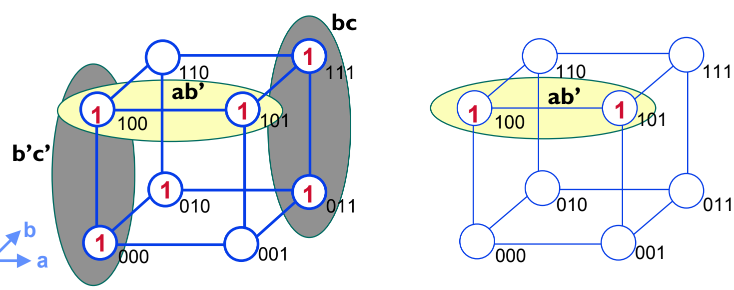
ESPRESSO: Multiple Output Functions

- We've totally avoided one big point so far...
  - In real world, want to minimize a set of functions over same input
  - f1(x,y,z), f2(x,y,z), f3(x,y,z), ... fk(x,y,z)
  - Want to try to share product terms among these functions



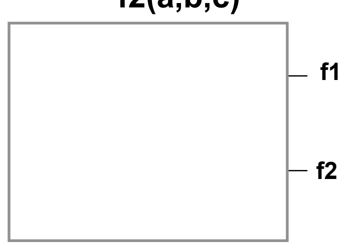
f1(a,b,c) f2(a,b,c)

Multiple Function Minimization



f1(a,b,c) f2(a,b,c)

Good solution has just 3 cubes, one of them shared between f1 and f2



ESPRESSO: Multiple Function Min

- Trick
  - Transform the multiple function problem into a single new function
  - Messy part: it's now a function with non-binary variables!
  - Called a *multi-valued* function
- There are generalizions to handle this...
  - PCN, Shannon expansion, URP algorithms, unateness, etc, all can be generalized to apply to this case
  - All the old algorithms work, they just get a lot messier inside
  - This is the way ESPRESSO really handles multiple functions simultaneously
  - De Micheli has some stuff about this...
  - ...but even he tends to avoid all the details

Summary

- Espresso does heuristic 2-level minimization
  - Avoids enumerating all PIs then doing a covering problem
  - Basic strategy is Reduce-Expand-Irredundant
    - Reduce: take a prime cover and shrink each cube so no minterms covered by more than 1 cube; done cube-by-cube
    - Expand: take a cover and make all the cubes prime; used to reshape a cover after reducing it; done cube-by-cube
    - Irredundant: take a prime cover and get rid of big set of redundant cubes to make better cover; not cube-by-cube, a covering problem
  - Repeat: Iteratively improve the cover...until can't make it any better
- How good is it?
  - Great, usually only a few cubes away from minimum
  - Fast, even for big things
  - It set the standard for 2-level minimization