Minimization of Combinational Circuits (6.2)

Notes:

- I will post a set of slides that does this very formally later. They are _really_ good. Look them over.
- The text does a pretty good job here (though not as good/formal as the slides).

One good question is "how do we minimize combinational circuits?" And the answer to that question is more complex than you'd think. Part of the problem is it isn't clear what it means to minimize. Do we mean to use the fewest number of gates? If so, we could end up taking something like the canonical sum-of-products and making a really, really, long path. That would greatly slow the clock.

We *probably* mean to reduce the number of gates as much as possible while keeping the delay bounded by some value. In this class, we'll just look at finding the minimal sum-of-products and product-of-sums forms for a given logic equation. This is called *two-level logic* because the worst-case path through the circuit will consist of exactly two gate delays.

Motivational Example #1

Consider the logic statement F(a,b,c)=ab'c+abc+abc'. You'll notice that we are using three 3-input AND gates and one three input OR gate to implement this logic (let us assume that all literals are available, so a and a' can both be used and don't require a NOT gate). What can you do to reduce the complexity of this circuit while keeping ourselves to two levels of logic?

First notice that ab'c+abc can be combined (using the "combining rule" from the first lecture of the year). We can also

combine abc+abc'. So we get: ___________________________ as a simplified version of this. As you've likely noticed, it isn't always easy to find pairs that can combine, and sometimes those pairs are well hidden inside of various terms.

Motivational Example #2

Consider the following: a'b'c+a'cd'+bcd. It turns out that equation can be reduced to a'c+bcd. How the heck can we figure that out? One trick is to drop back to the canonical sum-of-products form. That gets us:

____

You might notice that all four the terms that have a'c in them are true. So we can combine those four into a'c. That leaves abcd. It can be combined with a'bcd to make bcd. It turns out we can't combine any more and this point and so we've succeeded!

We got the right answer this way, but it wasn't obvious. Heck, it wasn't obvious that we are done. So the question is, how do we make it obvious? How can we quickly see what terms combine and prove to ourselves that we are done? What we'd like is an easy way to visualize exactly

What we are looking for

Say there are 3 variables: a, b and c. A given term (say abc) can be combined with three other terms (a'bc, ab'c and abc' in this case). We want a way to see those connections quickly.

Karnaugh Maps

Consider ab’c+abc+abc’ from above. Let’s write the truth table in a rather odd way:

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

Notice that box which represents abc is adjacent to the three different terms it can be combined with. In this case we can quickly see how the combining rule can be applied. We end up with two product terms: _________ and _________

Now consider

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

In this case, we can combine all four terms into a single term: ____________________

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

And notice we “wrap around” the edges. So this combines to be: ____________________

Informal Algorithm

What we are going to do is circle groups of “1s” that are rectangles[1] where each side is a power of 2 in length. The first K-map on this page had a 2x1 and a 1x2 rectangle, the second a 1x4 and the third a 1x2. We never circle a rectangle that is part of a larger legal rectangle. We only circle enough rectangles so that every 1 is circled.

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

Notice that in the K-map above there are three rectangles we could circle, but two of them cover all the “1s”. Let’s practice a bit.

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

____

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

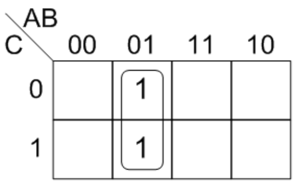
____

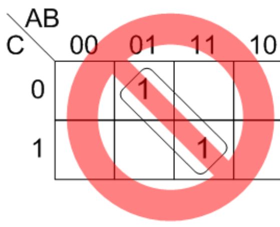
ab/c	00	01	11	10
0	1	1	1	1
1	1	1	0	1

____

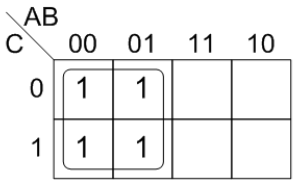
[1] Recall a square is special case of a rectangle.

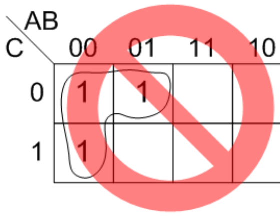
- K-map "rules"
  - Only circle adjacent cells (remember edges are adjacent!)



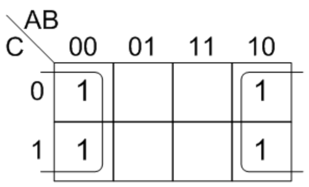


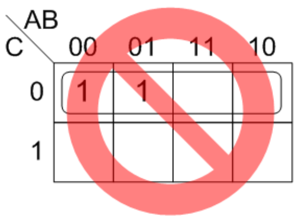
  - Only circle groups that are powers of 2 (1, 2, 4, 8, …)





  - Only circle 1-cells





An interesting example

Circle all rectangles:

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

____

Answer 1

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

____

Answer 2

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

____

Terminology (p308-310)

Notice that we are finding sum-of-products solutions.

- Recall that a ***minterm*** is a product term that includes all of the functions variables exactly once.
- The ***on-set*** of a function is the set of minterms that define when the function should evaluate to 1 (the minterms that have a 1 in the truth table.)
  - The ***off-set*** is the set of minterms that evaluate to zero.
- An ***implicant*** of a function is a product term that evaluates to 1 only in places that function evaluates to 1. (The on-set of an implicant of a function is a subset of the on-set of the function.)
  - Graphically, in a K-map an implicant is: ____
- An implicant ***covers*** those minterms that appears in its on-set.
  - What is the on-set of the function F(a,b)=a? _________________________
  - What minterms does that function cover? __________________________
- Removing a variable from a term is known as ***expanding*** the term. This is the same as expanding the size of a circle on a K-map.

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

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

- ***Prime implicant***: ____
- ***Essential one***[2]: ____
- ***Essential prime implicant***: ____

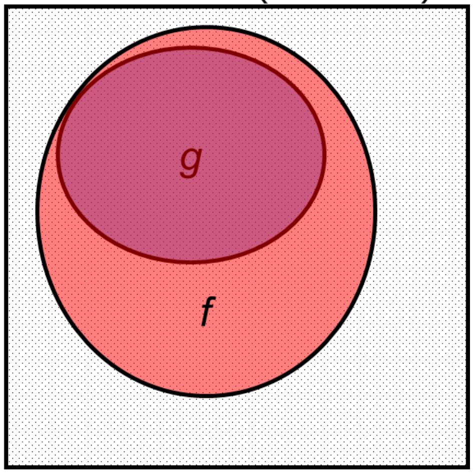
ab/c	00	01	11	10
0				
1				

[2] This term isn't used by our text, they skip from prime implicant directly to essential prime implicant.

🔵 = input combinations for which *g* outputs 1

🔴 = input combinations for which *f* outputs 1

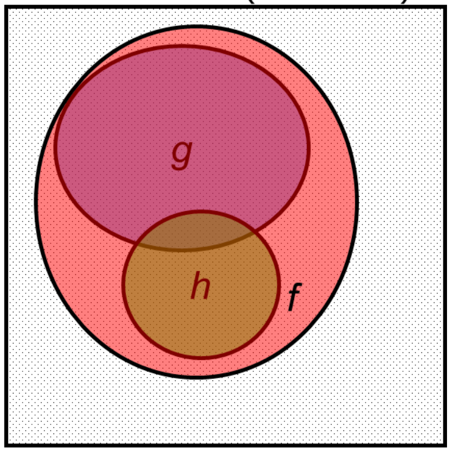
Space of all 2^n input combinations (minterms)



f* covers *g
if *f*=1 whenever *g*=1

$$f \geq g$$

Space of all 2^n input combinations (minterms)



🔵 = input combinations for which *g* outputs 1

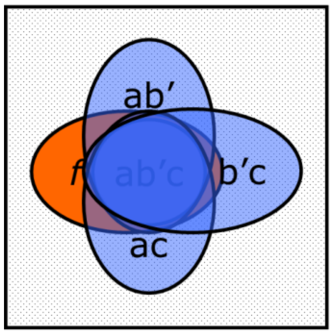
🔴 = input combinations for which *f* outputs 1

🟢 = input combinations for which *h* outputs 1

If *g* is a product term & *g*≤*f*,
Then *g* is an **implicant** of *f*.

- Removing a literal from any product term (any implicant) makes it cover twice as many minterms.
  - Removing a literal "grows" the term
  - ex. 3 variables: ab'c covers 1 minterm

$$\left.\begin{array}{l}\text{ab'}\\ \text{ac}\\ \text{b'c}\end{array}\right\}\text{each cover 2 minterms}$$



ab'c is an implicant of *f*.

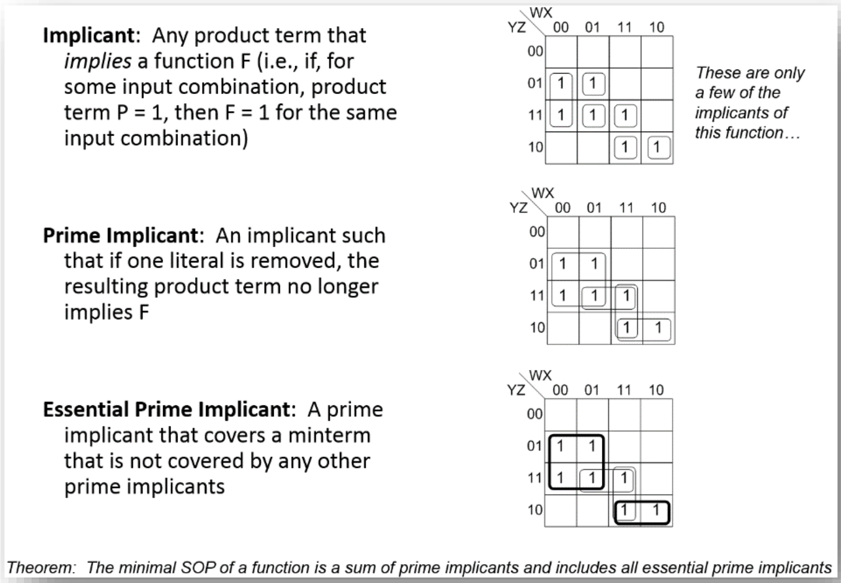
Any way of removing a literal makes ab'c no longer imply *f*.
So ab'c is a **prime implicant** of *f*.

More Formal Algorithm

- Identify all prime implicants
- Identify all essential ones.
- Circle all essential prime implicants
- Cover the remaining minterms using a minimal number of remaining prime implicants.

Notice that there may be more than one solution. Also notice that the last step is a bit vague ☺

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



Implicant: Any product term that *implies* a function F (i.e., if, for some input combination, product term P = 1, then F = 1 for the same input combination)

These are only a few of the implicants of this function…

Prime Implicant: An implicant such that if one literal is removed, the resulting product term no longer implies F

Essential Prime Implicant: A prime implicant that covers a minterm that is not covered by any other prime implicants

Theorem: The minimal SOP of a function is a sum of prime implicants and includes all essential prime implicants

4-variable

ab/cd	00	01	11	10
00	0	1	1	0
01	0	1	1	0
11	0	0	1	1
10	0	0	0	0

And some practice with these:

ab/cd	00	01	11	10
00	1	1	1	1
01	0	1	1	0
11	0	0	1	0
10	1	1	0	0

ab/cd	00	01	11	10
00	1	1	0	1
01	0	0	0	0
11	0	0	0	1
10	1	0	0	1

ab/cd	00	01	11	10
00	0	1	0	0
01	1	1	1	0
11	0	1	1	1
10	0	1	0	0

What's left?

- Don't cares
- 5+ variable
- Product-of-sums
- Programmable techniques
- Lots of practice.
- More context. Remember this is only for 2-level logic…

Don’t cares:

Note: I’m leaving the zeros blank to make things more readable!

$$F = \sum_{W,X,Y,Z}(1,5,7,11,15) + d(8,12,13,14)$$

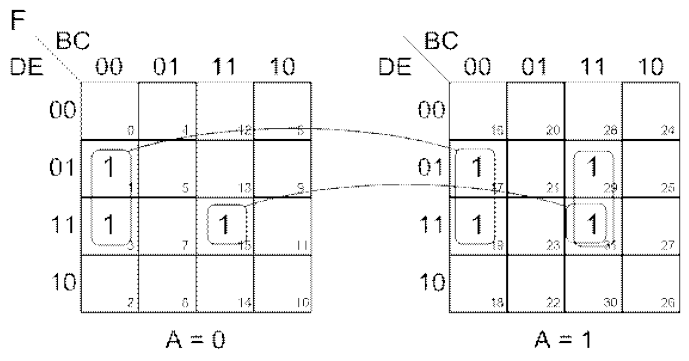
F

YZ \ WX	00	01	11	10
00			d	d
01	1	1	d	
11		1	1	1
10			d	

- **Use d cells to make prime implicants as large as possible.**
- **No PI should include only d’s**
- **Only 1-cells should be considered when finding the minimal cover set.**

5-variable

$$F = \sum_{A,B,C,D,E}(1,3,15,17,19,29,31)$$



$$F = \overline{B}\,\overline{C}E + BCDE + ABCE$$

Product of Sums:

Let F be:

ab/cd	00	01	11	10
00		1	1	1
01	1	1	1	1
11	1	1		
10	1	1		

Then F' is:

ab/cd	00	01	11	10
00	1			
01				
11			1	1
10			1	1

$F \rightarrow F'$

Find minimal SoP for F'.
Use deMorgans.

Programmable techniques

- Later in the semester, time allowing.

More context

Basically, just remember that this doesn't find the "minimal" solution. If finds the minimal sum-of-products. It's not even clear how we would measure "minimal" over all. Least delay? Least number of gates? Least "gate inputs"? Recall we did a "least delay" solution for GA1. This would have helped find a nice starting point, but wouldn't have solved the problem.

But this technique does let us find the minimal two-level solution (SoP or PoS). Which is pretty cool.

Questions:

1. Why isn't this a great technique for a computer? Why is it good for people?

2. What might be easier for a computer?

3. Can you define all the terms we've seen?
   - On-set, Off-set?
   - Implicant, prime implicant?
   - Essential one, essential prime implicant?
   - Cover?

4. Why is the instance to the right tricky?

ab/cd	00	01	11	10
00			1	
01	1	1	1	
11		1	1	1
10		1		

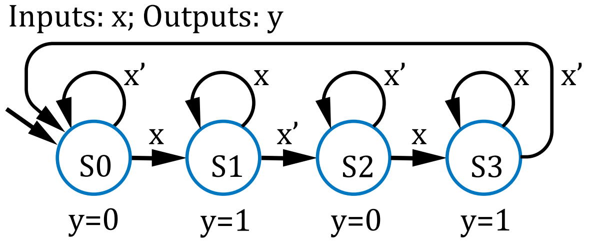
State minimization

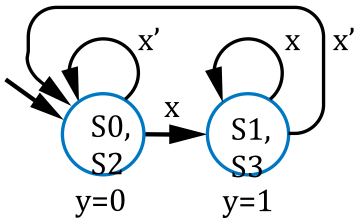
One thing to be aware of is that the 2nd edition of our text somehow got *worse* at covering this material. I've posted the relevant 6 pages from the 1st edition on the website, you should take a look at it.

Motivational Example

Consider the following two machines. Are they equivalent?

Inputs: x; Outputs: y





Say x were 0, 1, 1, 0, 0. What would be the output of the larger one? The smaller one? Are they *always* the same? How can you be sure?

What makes states equivalent?

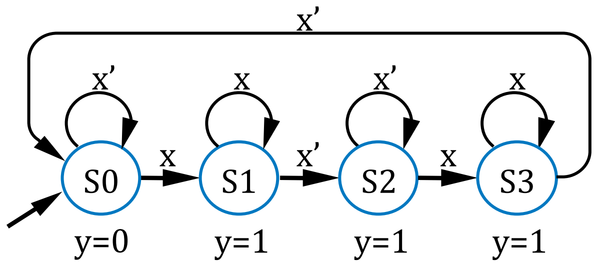
Two states are equivalent if

1. They assign the same values to outputs,

   e.g. S0 and S2 both assign y to 0,S1 and S3 both assign y to 1
2. AND, for all possible sequences of inputs, the FSM outputs will be the same starting from either state

   Let's look at this example. Are any states equivalent?

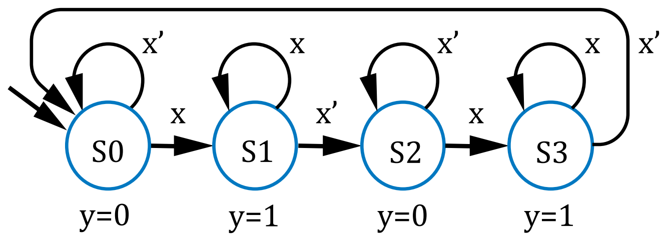
Inputs: x; Outputs: y



Let's use an implication table.

S1			
S2			
S3			
	S0	**S1**	**S2**

Inputs: x; Outputs: y



Which states can't be equivalent? Any with differing outputs! Cross those out.

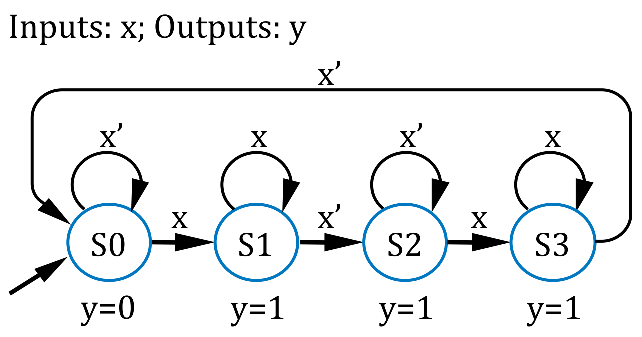
Now we'll follow the following algorithm. Do steps 2&3.

	Step	Description
1	*Mark state pairs having different outputs as nonequivalent*	States having different outputs obviously cannot be equivalent.
2	*For each unmarked state pair, write the next state pairs for the same input values*	
3	*For each unmarked state pair, mark state pairs having nonequivalent next-state pairs as nonequivalent. Repeat this step until no change occurs, or until all states are marked.*	States with nonequivalent next states for the same input values can't be equivalent. Each time through this step is called a ***pass***.
4	*Merge remaining state pairs*	Remaining state pairs must be equivalent.

What does this look like after step 4?

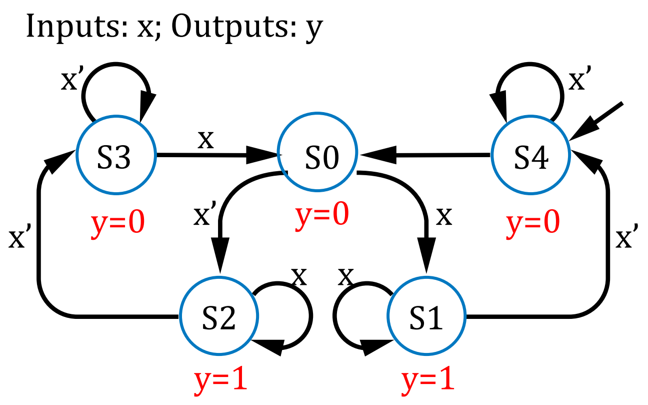
Another Example

S1			
S2			
S3			
	S0	**S1**	**S2**



Now let's do a final, less trivial, example:

S				
S				
S				
S				
	S	S	S	S



Step	Description
1 *Mark state pairs having different outputs as nonequivalent*	States having different outputs obviously cannot be equivalent.
2 *For each unmarked state pair, write the next state pairs for the same input values*	
3 *For each unmarked state pair, mark state pairs having nonequivalent next-state pairs as nonequivalent. Repeat this step until no change occurs, or until all states are marked.*	States with nonequivalent next states for the same input values can't be equivalent. Each time through this step is called a ***pass***.
4 *Merge remaining state pairs*	Remaining state pairs must be equivalent.