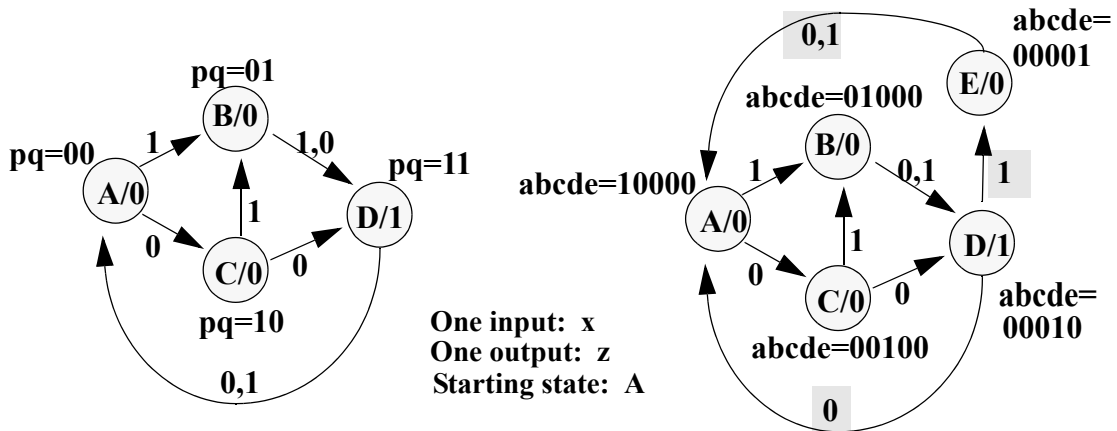


CMU Fall'01 18-760 VLSI CAD

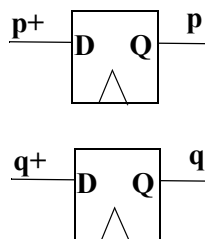
[120 pts] Homework 3. Out Tue Oct 2, Due TUE. Oct. 23, in class (v3)

1. Sequential FSM Verification [20 pts]

Show how to use **kbdd** to formally check the equivalence of these two example state machines. As you can see, one machine has 4 states, and the other has 5 states, and a few suspicious (shaded) transitions. Are they equivalent? Show how to use **kbdd** to do the reachability set computations, one set at a time, for the cross-product machine. The goal here is to understand step by step how you do this mechanically, as a sequence of symbolic ops using a BDD package. Hand in a printout of your **kbdd** run to show how you did this. **Explain** your answer



Encoded pq machine

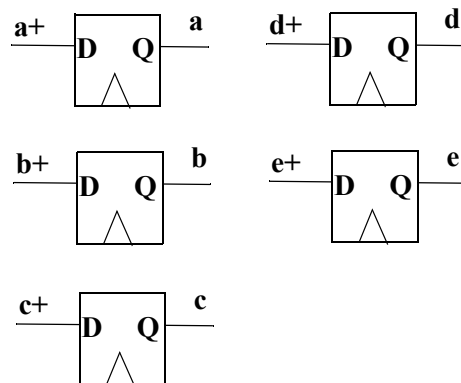


Implementation equations

$$\begin{aligned} p+ &= q'x' + p'q \\ q+ &= q'x + p'q + pq'x' \\ z &= pq \end{aligned}$$

...with or without this x' var, it is correct

1-hot abcde machine



Implementation equations

$$\begin{aligned} a+ &= dx' + e & z &= d \\ b+ &= ax + cx \\ c+ &= ax' \\ d+ &= b + cx' \\ e+ &= dx \end{aligned}$$

bugfix--this one is OK

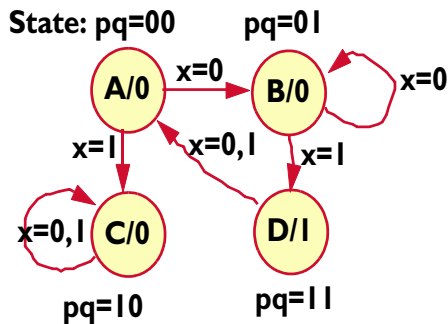
2. FSM Reachability Analysis [10 pts]

In the lecture about FSM verification, we introduced the concept of *reachability analysis*, where we represent as a Boolean function the set of all states that our machine can visit from a known start state in 0 clock ticks (we called it R_0), 1 tick (R_1), 2 ticks (R_2), and so on. This is actually a *forward* reachability analysis: you pick a state in which the FSM starts and then you go forward in time to see what states you can reach. It turns out that you can also go *backwards* in time. You pick an “end state” and you go *backwards* to see which sets of *prior* states could have got you into that state. It turns out that this has one very nice technical advantage: you don’t need to build the BDD for the transition relation

$$\delta(\text{old state, input, next state}) = 0 \text{ or } 1$$

In practice it is often true that δ can make a huge BDD.

Consider again our simple machine from the lecture. Let $R_0 = \{B\}$ be our arbitrarily chosen end



- $R_0 = \text{end state} = \{B\}$
- $R_1 = \text{prior states 1 tick before} = \{A, B\}$
- $R_2 = \text{prior states 2 ticks before} = \{A, B, D\}$
- $R_3 = R_2$

state. Then we expect the set of states prior to R_0 —the states we could have been in one clock tick in the *past*, and still reached state B one tick *later*—to be $\{A, B\}$, which we call R_1 . In this notation, R_k denotes the Boolean function that represents the set of all states that could reach our end state $\{B\}$ in not more than k clock ticks. It turns out that there is another quantification sort of formula to figure out how to compute these sets. For our particular little FSM, that formula is:

$$R_{-(k+1)}(p,q) = R_{-(k)}(p,q) + (\exists x)R_{-(k)}[p^+(p,q,x), q^+(p,q,x)]$$

where $R_k(p,q)$ is the Boolean function for a set of reachable states, and (p,q) is the bit pattern representing a state in our FSM. $p^+(p,q,x)$ and $q^+(p,q,x)$ are the next-state equations from the FSM. If you start in state (p,q) and see input x , then you go to state (p^+, q^+) .

For this problem, do the following:

- Explain briefly *why* this formula works. In English, what is going on here? The key is to look at each term in the equation, and the quantification operation, and understand why this formula gives us a new Boolean equation that represents the states our FSM could be in one clock tick earlier.
- Assume that $R_0(p,q) = \{\text{state B}\} = p \cdot q$. Then, show how to use this formula to compute $R_1(p,q)$. Does the result make sense?

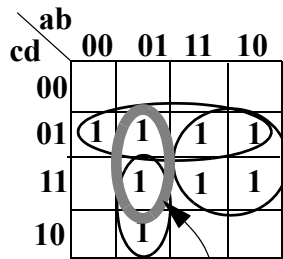
3. URP Algorithms: Cube Containment [10 pts]

A natural question to ask in any attack on logic simplification is this one:

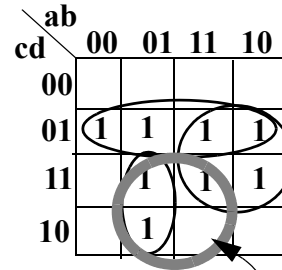
given a particular cover of a function F represented as a set of cubes $\{C_1, C_2, \dots, C_k\}$, and another cube S , does the cover of F actually **contain** cube S ?

Two examples are shown below:

cover = $\{c'd, ad, a'bc\}$



This cube $S=a'bd$ is contained in the cover



This cube $S=bc$ is NOT contained in the cover

It turns out there is a very simple way to do this via a URP algorithm. We first need one bit of notation:

Given a cover of $F = \{C_1, C_2, \dots, C_k\}$ and another cube S

The *cofactor* of cover F with respect to cube S , written F_S , is the set of cubes that result by cofactoring each cube C_i with respect to the variables in the product term represented by cube S .

For example, if we represent F with the SOP cover $ab + bc + bc'd + a'b'd$ and let cube $S=a'c$. Then F_S is the set of cubes we get by setting $a=0$ and $c=1$ in this list. The result is $F_S = 0 + b + 0 + b'd = b + b'd = b+d$. This leads us to our big result:

a cover $F = \{C_1, C_2, \dots, C_k\}$ **contains** cube S if and only if $F_S = 1$.

This is a very practical result, since it means we can test if a cover F **contains** a particular cube S by computing the cube list for cofactor F_S and then calling our URP tautology algorithm on F_S .

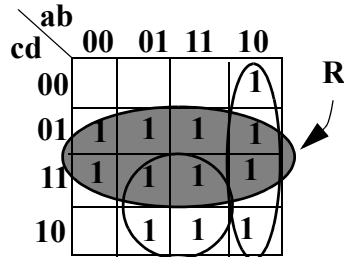
Do this: prove the IF part of this result: **If** F contains cube S , **then** $F_S = 1$.

Hint: You can look in DeMicheli but you can't copy that proof, which is too terse. Pretend F is just another SOP Boolean algebra equation. LHS = SOP form. RHS = "1". What can you try to do to both sides of this equation (it involves term S) to get to this result? The answer is not very much algebra.

4. ESPRESSO Ops: Reduce [10 pts]

Consider the cover of the new function $G(a,b,c,d)$ shown below. We want to apply the REDUCE operator to the single shaded cube R in the map. To illustrate how REDUCE actually does its job, do the following:

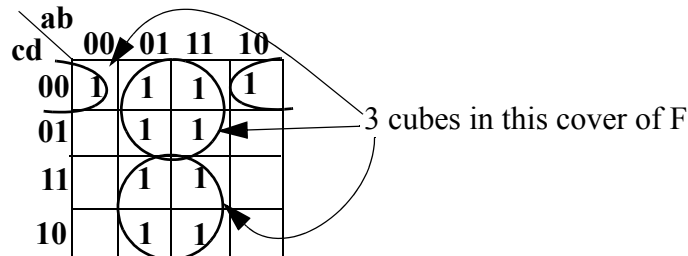
- Write the PCN cube list for the starting cover (3 cubes) shown below for G .
- Write the PCN cube list for the cover that has the single cube R removed: $G - \{R\}$.



- Show a Kmap for the complement of this reduced cover $G - \{R\}$. Draw the 1's in this map and circle a reasonable-looking cover by eye, i.e., just draw a reasonable cover of the OFF set by eye.
- In PCN notation (write a cube list) and on the Kmap, intersect each cube in this complement with the original R cube.
- Take this new intersected cubelist and do the supercube operator (bitwise OR) across all the cubes (if there is more than one cube). Draw the resultant single cube on the Kmap.
- Comment in a sentence why this result makes *sense* as “the reduction of cube R ” in the ESPRESSO algorithm.

5. ESPRESSO Operations: Complement, Expand-Irredundant [20 pts]

Consider the function $F(a,b,c,d)$ shown on the Kmap below. An intermediate nonprime cover with 3 cubes is given below.



Answer the following, showing your work for each step:

- **Complement [2 pts]:** By eye, pick a decent (try for minimal) cover of the off-set of the function in the above Kmap. Draw the Kmap for this cover.

- **Cube ordering [4 pts]:** Determine and list the order in which the cubes in the cover of the ON set given above will get expanded using the cube weighting heuristic. Show the weight computation. Redraw the Kmap above and show the weight for each cube.
- **Expand [12 pts]:** In order from lightest to heaviest, show how to expand each cube. In particular, for each cube: build and show the blocking matrix; extract by eye a decent looking cover for the matrix; show the resulting expanded cube. Draw a Kmap with each of the new expanded cubes circled.
- **Irredundant [2 pts]:** by eye, pick an irredundant set of cubes from the expanded cubes you just generated. **Draw** the final Kmap covering that uses these cubes.

6. Algebraic Division and Kerneling [20 pts]

Consider these 2 functions:

$$F = abrs+abrt+abd+abe+abu+ghrs+ghrw+ghd+ghe+ghu+dp+eq+rstuw$$

$$G = a+b+cd$$

Do this:

- Show how to do the division F/G using the algebraic division algorithm from the lecture notes. Show the work involved, and give the final quotient and remainder expressions.
- Use the recursive algorithm from class notes, and show how to find all the kernels and co-kernels in function F . Show the recursion tree like in the class notes.

7. Kerneling [10 pts]

Since kerneling is an important and frequent operation in multilevel synthesis, there are variants of the basic algorithm that trade some quality for more speed. Let's consider one such option, presented here in pseudocode form:

```

quickKernel( SOP cover of function F ) {
  if( every literal in F appears just once )
    return F;
  let x = any literal that appears more than once in F;
  let G = F/x // ie, cross out x in each term where it appears,
              // and remove those other terms that did not have any x's in them
  // make G "cube free", ie, find any cube that is common to ALL products
  // in the equation for G, and then get rid of it
  let c = first product term in SOP form for G;
  for( each product P in SOP form G )
    let c = literals common to product P and product c
  let G = remove common cube c (if it is non-empty) from each product term in G
  return quickKernel( G )
}

```

NOTE: different phrasing for this step, maybe easier to see what's happening. Ex: if $P = xyzw$ and $c=xw$, this step just yields $c = xw$

Do this:

- **Show** the result of running this algorithm on the expression for F from the previous problem. Show the chain of recursive calls and what gets passed into and out of the chain of calls. For the step that says “pick a literal that appears more than once in F”, choose the first available literal in **alphabetical** order.
- This algorithm finds exactly **one** kernel of the function F. It also always finds a particular **kind** of kernel -- what “kind” of kernel does it find? (**Hint:** kernels have *levels*...)

8. Comparing 2-level and multi-level synthesis [20 pts]

Let's try to get a feel for the difference in approach and results between two-level and multi-level minimization, as exemplified by 2 real synthesis tools: **Espresso** (for 2-level minimization) and **SIS** (for multilevel minimization). We are going to be working with a simple combinational function. The computation we want to implement operates on two 4-bit unsigned numbers and produces a 4-bit unsigned number:

$$s[3:0] = (a[3:0] + b[3:0]) \text{ MOD } 13$$

Keep in mind that you (conceptually) do the 4-bit add first, then compute the mod-13 remainder (a number between 0 and 12) as the result. If this was a C program, it would something like:

```
unsigned int a, b s;
s = (a + b)%13;
s = s & 0x0000000f;
```

1. Create an Espresso file for these output functions. We suggest you do this with a simple program in your favorite language, but feel free to do it by hand if you prefer.

ESPRESSO has a simple input and output format (comments are in italics, you don't type in these comments):


```
.i 4           there are 4 inputs
.o 1           there is 1 output
.ilb w x y z   names of inputs
.ob f         names of outputs
0-11 1        a line of the truth table,
              "-" means don't care on this input
01-1 1        more truth table
1011 1        ditto
1111 -        explicitly declaring that f is
              don't care for this input pattern
0110 0        more truth table
.e           done
```

If you put this in a file called *input*, you can run this by typing:

```
espresso <input >output
```

and file *output* will contain this:

```
.i 4
.o 1
.ilb w x y z
.ob f
.p 2           result SOP form has 2 product terms in it
--11 1        one product term is: yz
01-1 1        the other term is: w'xz
.e
```

 *corrected this comment*

ESPRESSO's output format is the same as the input, and you just read the truth table lines in the output as specifying product terms you OR together to get the final SOP solution.

The resultant file for this “mod 13” function should look like the following (except for the *italics comments*):

```

      8 inputs
.i 8  ←
.o 4  ← 4 inputs
.ilb a0 a1 a2 a3 b0 b1 b2 b3 ← names of input vars
.ob s0 s1 s2 s3 ← names of output vars
.p 256 ← doesn't matter for input, output tells # of products
00000000 0000 ← (0+0)mod 13 = 0
...
11000011 0010 ← (12+3)mod 13 = 2
...
11111111 0100 ← (15+15)mod 13 = 4
.e ← end of file

```

Minimize the logic (assume you put it in a file called **mod13.pla**) by running Espresso:

```
/afs/ece/class/ee760/bin/espresso <mod13.pla > mod13.out
```

Assuming you just implemented the result as a two-level structure, and you had up to **8-input** AND and OR gates (and inverters) answer these questions:

- **How many gates** are required to implement the product outputs? (If you need an input inverted, just count 1 inverter, for simplicity).
- **How many literals** in all? (You may want to write a little program to count).
- Include your espresso **output file** as well.

2. Now, minimize the logic by running SIS:

```
/afs/ece/class/ee760/bin/sis
```

To read in the file (which is in *programmable logic array*, “pla” format), type the following at the sis> prompt:

```
read_pla mod13.pla
```

To verify that it read it in properly, you can look at the SOP form for the function by typing “print”.

We are going to use the well-known script called “rugged” that tends to give reasonably good answers. A script is simply a recipe or series of operations performed on the multi-level network. To run this script, you can either type in the commands below directly at the sis> prompt, or put them in a file and type “source <file>”. Here is the “rugged” script (same as from the lecture notes):

```

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
```

To see the resultant multi-level network, type “print”. Using the UNIX script utility, or simply by cut-and-paste, print out this resultant network and hand it in. As in part 2, answer:

- **How many gates** would be required to implement it, assuming each node was done as a standard two-level structure AND-OR form, and you have up to 8-input gates? (Be careful to count the inverters this time, since they may appear inside individual nodes in the Boolean network.)
 - **How many literals** in all?
3. Comment briefly on the differences in the results between Espresso and SIS.