University of California at Berkeley
College of Engineering
Department of Electrical Engineering and Computer Science

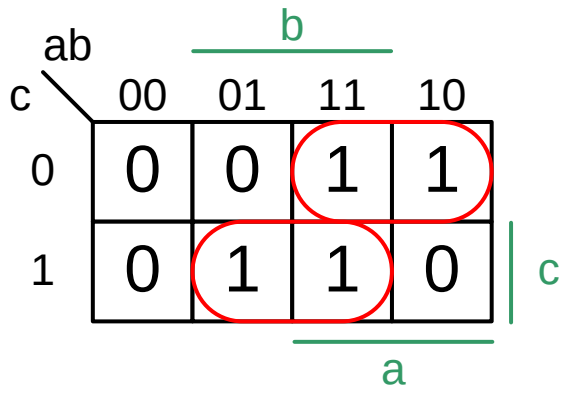
EECS150, Spring 2010

Homework 10 Solutions: Combinational Logic

1. This circuit was a mux, given by the truth table:

a	b	c	Out
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

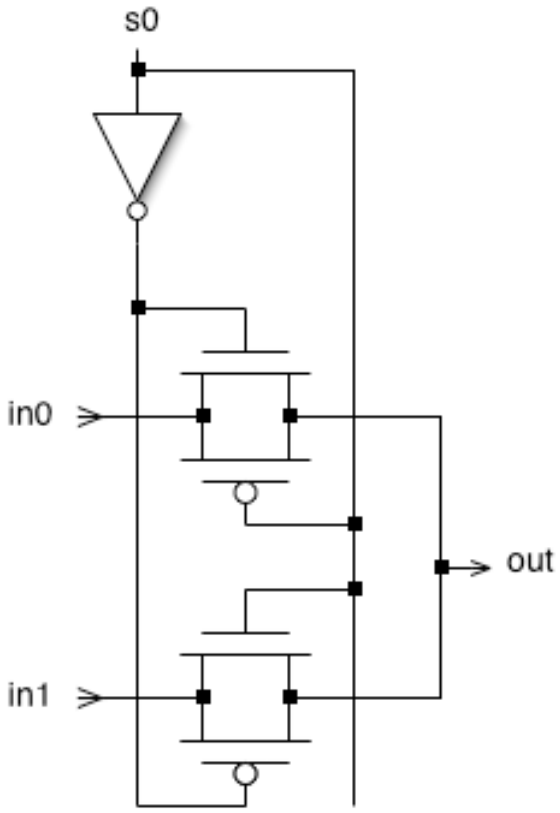
From this, we construct the minimized K-map:



And from the minimized K-map, we derive the optimal logic expression:

$$Out = a\bar{c} + bc$$

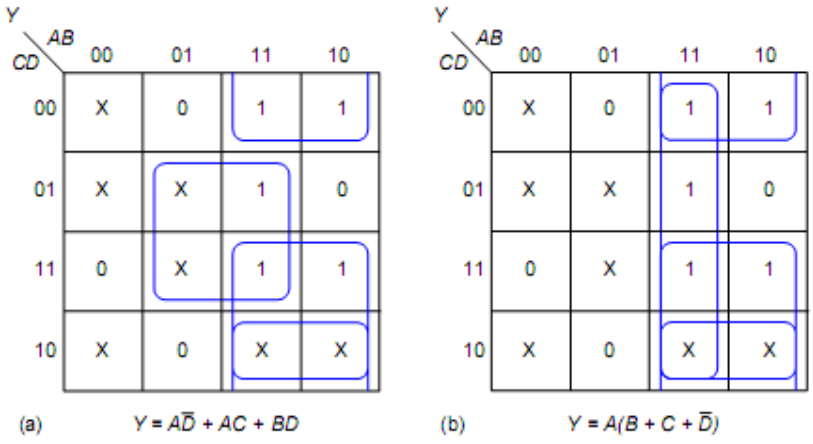
Can we optimize this circuit further? Yes we can - with transistor-level optimizations. In the case of the mux, we can use the pass-gate approach shown in lecture:



Transistor-level optimizations usually yield a more optimal solution relative to K-map optimizations because K-maps deal at the logic gate level. The lower down (in terms of abstractions), we optimize, the better result we are going to get.

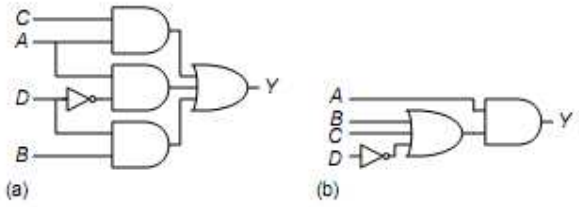
2. DDCA 2.19

Two different ways of arriving at a minimal boolean expression for this truth table are shown here:



Note that the X's (don't cares) are allowed to take on either a zero or a one. We can let specific X's be 1's if we can use them to grow a bigger covering in our K-Map (remember, the bigger the covering, the more minimal it is). We let all the other X's be 0's so we don't have to make new coverings just to accomodate them (the fewer the coverings, the more minimal the implementation).

Once we have our minimized boolean expressions, building the circuit is trivial (again, two different answers):

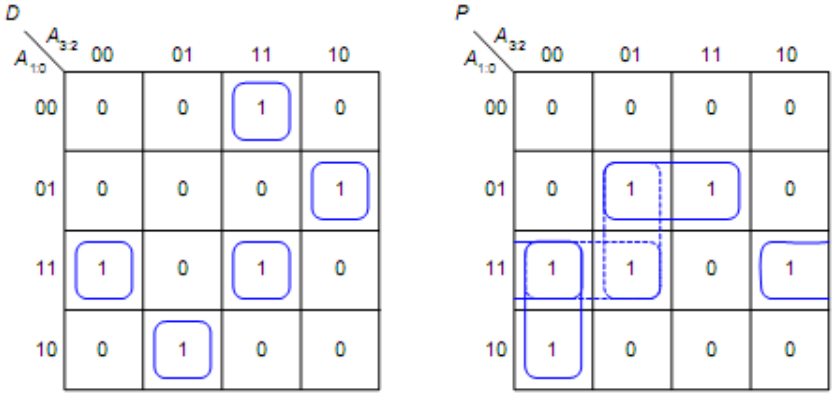


3. DDCA 2.24

We can capture the exact behavior of this circuit by first describing its function using a truth table:

Decimal Value	A_3	A_2	A_1	A_0	D	P
0	0	0	0	0	0	0
1	0	0	0	1	0	0
2	0	0	1	0	0	1
3	0	0	1	1	1	1
4	0	1	0	0	0	0
5	0	1	0	1	0	1
6	0	1	1	0	1	0
7	0	1	1	1	0	1
8	1	0	0	0	0	0
9	1	0	0	1	1	0
10	1	0	1	0	0	0
11	1	0	1	1	0	1
12	1	1	0	0	1	0
13	1	1	0	1	0	1
14	1	1	1	0	0	0
15	1	1	1	1	1	0

Now that we have our truth table, we can go ahead and minimize its logic by entering it into a K-Map:

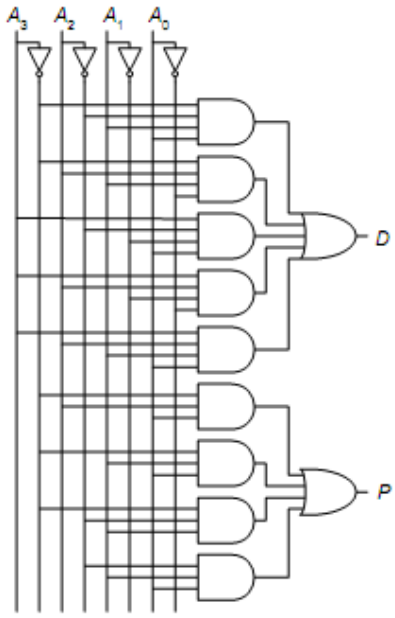


Note that there are two different ways to minimize the boolean expression for P.

From our K-Map, we arrive at the following two boolean equations:

$$D = \overline{A_3}\ \overline{A_2}\ A_1\ A_0 + \overline{A_3}\ A_2\ A_1\ \overline{A_0} + A_3\ \overline{A_2}\ \overline{A_1}\ A_0 + A_3\ A_2\ \overline{A_1}\ \overline{A_0} + A_3\ A_2\ A_1\ A_0$$
$$P = \overline{A_3}\ A_2\ A_0 + \overline{A_3}\ A_1\ A_0 + \overline{A_3}\ \overline{A_2}\ A_1 + \overline{A_2}\ A_1\ A_0 \text{ or}$$
$$P = \overline{A_3}\ A_1\ A_0 + \overline{A_3}\ \overline{A_2}\ A_1 + \overline{A_2}\ A_1\ A_0 + A_2\ \overline{A_1}\ A_0$$

With our boolean expressions, we can implement our circuit in the standard fashion:



Note that the circuit implements only the first minimized expression for P.

4.

$$\begin{aligned} F_{two} &= ac + ad + bc + bd + e \\ &= a(c+d) + b(c+d) + e \\ F_{three} &= (a+b)(c+d) + e \end{aligned}$$

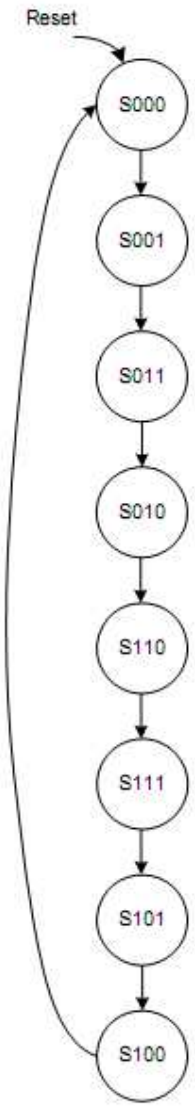
Each 2-input gate requires 2 transistors per input = 2 transistors.

```
2 Level AND/OR Logic:
    4x 2-input AND gates
    4x 2-input OR gates (
Total Cost:         4*(4 + 4) = 32 transistors
Worst Case Delay:   1 AND gate + 3 OR gates = 4 gate delays

3 Level Logic:
1x  2-input AND gates
    3x 2-input OR gates
Total Cost:         4*(1 + 3) = 16 transistors
Worst Case Delay:   1 AND gate + 2 OR gates = 3 gate delays
```

5. DDCA 3.24

The first thing we do in any FSM design problem is translate the problem statement into the state transition diagram for the FSM we are building. In the case of the gray code counter we are building, the state transition diagram is shown here:



Once we have the state transition diagram, we pick a convenient encoding for all the states. In this case, we will pick the state encoding to be just the gray codes themselves (i.e. state S011 has an encoding of `3'b011`). This has the advantage that the output of this FSM is just the current state (i.e. the output of the FSM is just the output of the current state register). The next state transition table is shown here:

S_2	S_1	S_0	NS_2	NS_1	NS_0
0	0	0	0	0	1
0	0	1	0	1	1
0	1	0	1	1	0
0	1	1	0	1	0
1	0	0	0	0	0
1	0	1	1	0	0
1	1	0	1	1	1
1	1	1	1	0	1

The output logic truth table is shown here:

S_2	S_1	S_0	Out_2	Out_1	Out_0
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	0
0	1	1	0	1	1
1	0	0	1	0	0
1	0	1	1	0	1
1	1	0	1	1	0
1	1	1	1	1	1

6. Repeated DDCA 3.24

   The only thing that changes between this problem and the last is how we encode our states. Using one-hot encoded states, the following table shows the new state encodings for each state:

State	**Encoding**
S000	00000001
S001	00000010
S011	00000100
S010	00001000
S110	00010000
S111	00100000
S101	01000000
S100	10000000

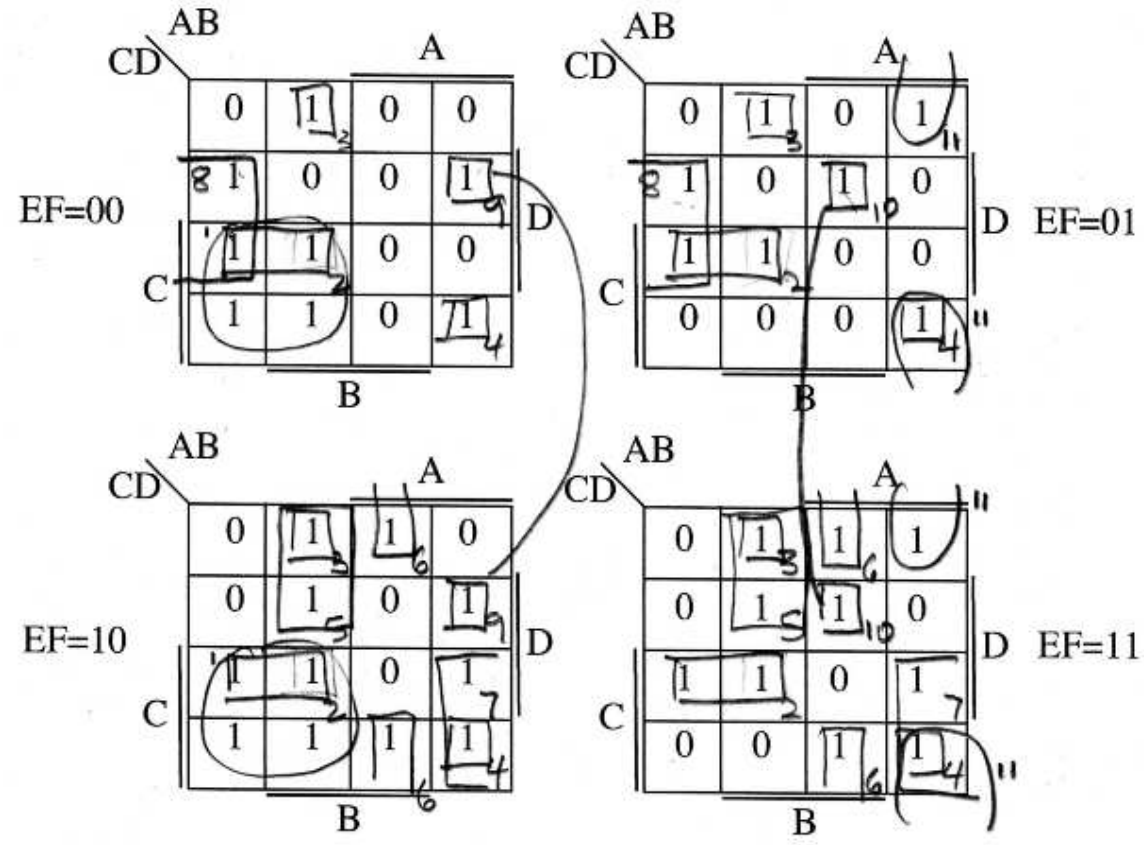
   This particular encoding has the advantage that the next state function is just a left shift of the current state (and that the top bit, when shifted off to the left, wraps around to the bottom bit) and can be implemented with just a simple shift register. The following table shows the next state transition table with one-hot encoded states:

State	Next State
00000001	00000010
00000010	00000100
00000100	00001000
00001000	00010000
00010000	00100000
00100000	01000000
01000000	10000000
10000000	00000001
All else	xxxxxxxx

However, since our current state is no longer just the gray code itself, our output function will not just be the current state. We will need a circuit that converts our one-hot encoded states to their respective gray code values. This circuit implements the behavior described by the truth table:

State	Output
00000001	000
00000010	001
00000100	011
00001000	010
00010000	110
00100000	111
01000000	101
10000000	100
All Else	xxx

7. This problem appeared on midterm 1 of CS150 spring 2007. The minimal covering that they came up with is shown here:



The expression corresponding to this minimal covering was:

$G = \overline{A}\,C\,\overline{F} + \overline{A}\,C\,D + \overline{A}\,B\,\overline{C}\,D + A\,\overline{B}\,C\,\overline{D} + \overline{A}\,B\,\overline{C}\,E + A\,B\,\overline{D}\,E + A\,\overline{B}\,C\,E + \overline{A}\,\overline{B}\,D\,\overline{E} + A\,\overline{B}\,\overline{C}\,D\,\overline{F} + A\,B\,\overline{C}\,D\,F + A\,\overline{B}\,\overline{D}\,F$

8. The last Bear in line can see every skullcap but his own, and so he calls out Red if the Red skullcaps in front of him have odd parity, White if the Red skullcaps in front have even parity. Hes got a 50To see this, consider the Bear immediately in front of the last Bear. The parity computed by the last Bear is the parity of all the caps in front of him XORd with his own cap. He knows the parity of all the caps (he just heard the last Bear yell it out) and he knows the parity of the caps in front. From this, he can deduce his own color, from the following table:

	Global Parity Odd	*Global Parity Even*
Parity in Front Odd	*White*	*Red*
Parity in From Even	*Red*	*White*

Now consider the third Bear from the back in line. He knows three things: the parity of the caps in front of him, the parity of all the caps (except for the cap of the last Bear, which were ignoring), and the cap of the Bear behind him. And he knows the following equation must hold: Parity of all caps = my cap is Red $\oplus$ Parity of caps ahead $\oplus$ Cap behind is Red So this yields the following table:

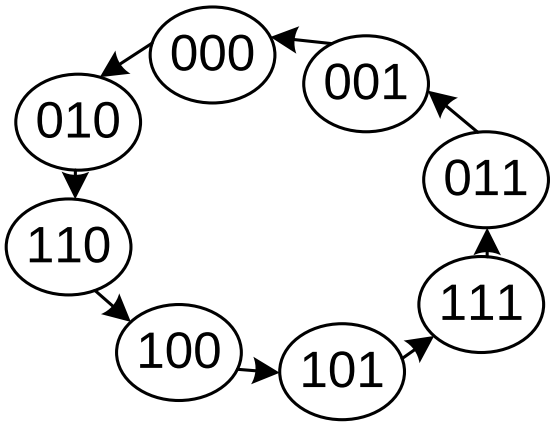
	Global Parity Odd	Global Parity Even
Parity in Front Odd, Cap Behind is Red	White	Red
Parity in Front Odd, Cap Behind is Red	Red	White
Parity in Front Even, Cap Behind is Red	White	Red
Parity in Front Even, Cap Behind is White	Red	White

This is of course our familiar three-variable parity function, with some change of names and variables! Now consider the general case. The kth Bear from the end waits until he hears all k-1 Bears behind him call out their caps. From this, he knows three things: first, the global parity (from the original call; two, the parity behind him (hes kept track as hes heard the Bears behind him call out their colors correctly), and, three, the parity of caps in front of him. And he knows he must maintain the following invariant: Parity of all caps = my cap is Red $\oplus$ Parity of caps ahead $\oplus$ Parity of caps behind Which gives the following table:

	Global Parity Odd	Global Parity Even
Parity in Front Odd, Parity Behind Even	White	Red
Parity in Front Odd, Parity Behind Odd	Red	White
Parity in Front Even, Parity Behind Odd	White	Red
Parity in Front Even, Parity Behind Even	Red	White

Which again is the familiar parity function. Using these observations and this table, the Bears are able to defeat the Evil Cardinal. They then captured his Axe and, to prevent further violence of this type, mounted it on a wooden plaque where it remains today.

9. The bubble-arc diagram for this state machine is given below:

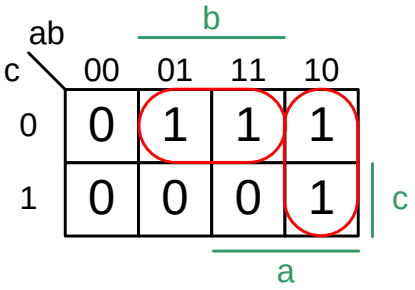


From this, we can construct the truth table for this FSM. Like other counters, which transition unconditionally based on their current state and always to a known next state, the current state can be thought of as the inputs:

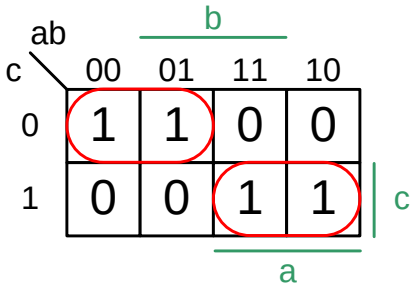
cs0	cs1	cs2	ns0	ns1	ns2
0	0	0	0	1	0
0	0	1	0	0	0
0	1	0	1	1	0
0	1	1	0	0	1
1	0	0	1	0	1
1	0	1	1	1	1
1	1	0	1	0	0
1	1	1	0	1	1

With three next state bits, we need $3\times$ 3 variable K-Maps (let a, b, c equate to $cs_{0,1,2}$):

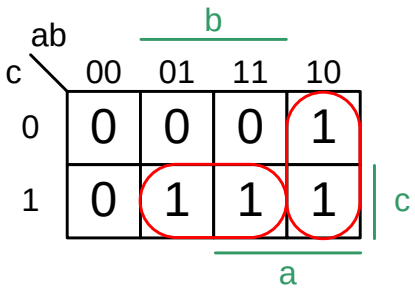
ns0



ns1



ns2



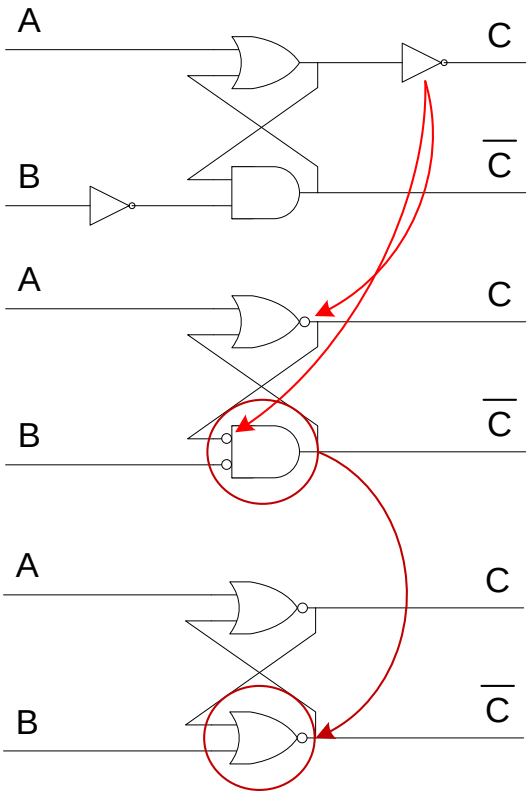
Which yield the following minimized boolean expressions (equate $x[2:0]$ to $ns_{0,1,2}$):

$$ns_0 = b\bar{c} + a\bar{b}$$

$$ns_1 = \bar{a}\bar{c} + ac$$

$$ns_2 = bc + a\bar{b}$$

10. This circuit is, in fact, an S-R latch. We can transform the circuit using bubble-pushing techniques as follows:



Push the inverter back to the top OR gate's output. In order to maintain the same logic function, we need another inverter at the top input of the AND gate.

Use bubble-pushing/De Morgan's laws to transform the AND into a NOR.

And the corresponding wave diagram (which follows from standard S-R latch behavior) is: