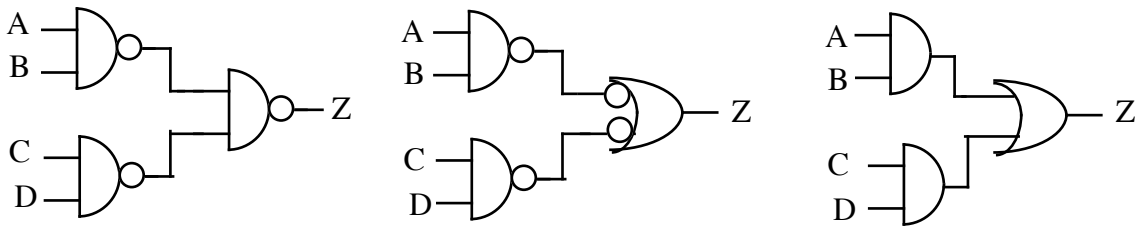
Solutions for 3824 Midterm Exam 3/4/04

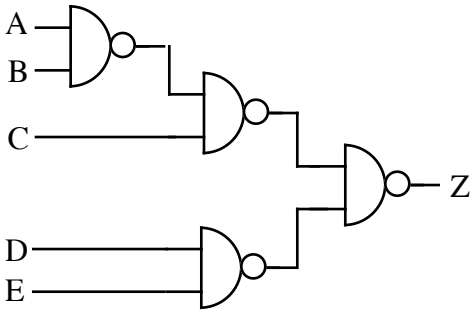
1. Truth table.

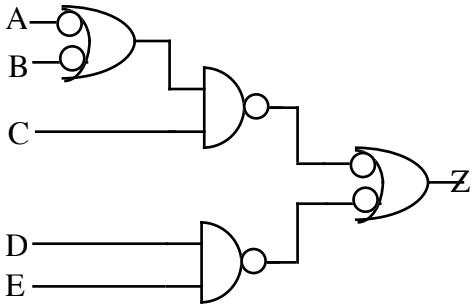
A	B	C	Z
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

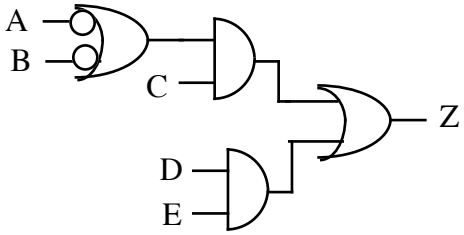
2. When analyzing a circuit built with NAND-gates, the easiest approach is first to convert the circuit to one with AND-gates, OR-gates and inverters, as we did in class. A NAND-gate can be thought of as an OR-gate with inverted inputs. Bubbled outputs and bubbled inputs can be cancelled. So the circuit below can easily be converted to an AND-OR circuit as shown. We can then easily see that the corresponding logic expressions is Z=AB+CD



For the circuit of this problem, shown below, we can easily find the equivalent circuit shown on the right. We can then transform the right-hand two levels to get the circuit below.







Now we can easily write an expression for the third circuit:
$Z=(\bar{A}+\bar{B})C+DE=\bar{A}C+\bar{B}C+DE$
This is a great deal easier than analyzing the original circuit directly to get the hideous expression"
Z=(((AB)'C)'(DE)')' and then using the deMorgan laws to reduce it to the same SOP expression.

3. Find a minimal sum-of-products (SOP) expression for the function described by:

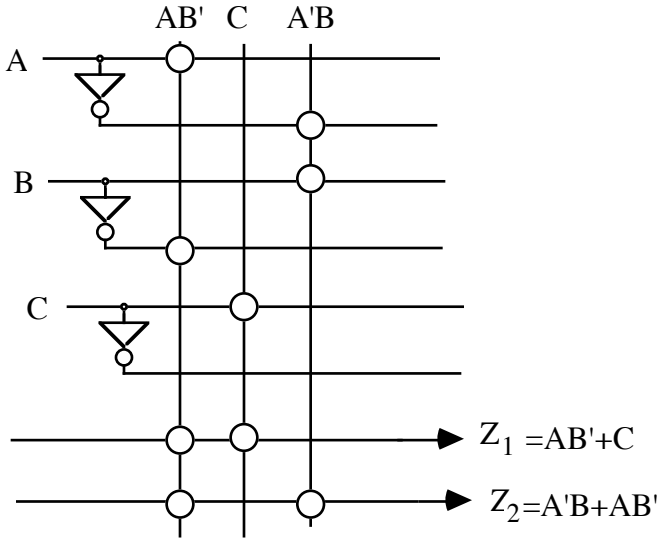
$Z=AB+\bar{A}C+BCD$
First use the consensus theorem ($PQ+\bar{P}R=PQ+\bar{P}R+QR$) to introduce the "assassin" term, BC:
$Z=AB+\bar{A}C+BC+BCD$
Then use BC and the absorption theorem (P+PQ=P) to eliminate BCD. Finally, use the consensus theorem again, in the other direction, to liquidate the assassin, obtaining:
$Z=AB+\bar{A}C$.

4. Find a simple product-of-sums (POS) expression for the function described by $Z=A\bar{B}+C(D+\bar{E})$.
The best approach is to use "add-out" version of the distributive law, (P+QR=(P+Q)(P+R)). Since we are looking for a POS form do NOT use the other form of the distributive law, i.e., do NOT multiply out, since this would take us further from our goal. The first step is to treat $A\bar{B}$ as P, C as Q, and $(D+\bar{E})$ as R, which leads to:
$Z=A\bar{B}+C(D+\bar{E})=(A\bar{B}+C)(A\bar{B}+D+\bar{E})$.
Next add out the parenthesized terms (treating $A\bar{B}$ as QR in both cases, with C playing the role of P in one case and $D+\bar{E}$ doing so in the other case), which gives us:
$Z=(A+C)(\bar{B}+C)(A+D+\bar{E})(\bar{B}+D+\bar{E})$.

Unfortunately a number of students used the incorrect procedure in the text (which I warned against several times) to laboriously produce an invalid result: a COMPLEMENTED POS expression instead of a POS expression.

5. Find a minimal SOP expression for the complement of $Z=(A+\bar{B})(C+\bar{D}E)$.
This can be solved in one step by simply interchanging AND and OR, and complementing all literals (do NOT add or delete overbars). Pay careful attention to the use of parens to enforce proper precedence when necessary. This gives us:
$Z=\bar{A}B+\bar{C}(D+\bar{E})$
Multiplying out yields the desired SOP expression, $Z=\bar{A}B+\bar{C}D+\bar{C}\bar{E}$.
We could also do this in smaller steps, using the deMorgan laws alternately:
Z=(A+B')'+(C+D'E)'= A'B+C'(D'E)'=A'B+C'(D+E')=A'B+C'D+C'E').

6. Program the PLA below (by putting circles around intersections) to realize the functions described by the expressions,
$Z1=A\bar{B}+C$ and $Z2=A\Theta B$.
Remember that an important advantage of the PLA is that the same product term can be fanned out to several OR-gates, as was done with the $A\bar{B}$ term below.



7. Simplify the following expression: The result should be in the same form as the original expression, using XOR, AND, and, if necessary, NOT. Note that an inverter is cheaper than an XOR-gate. $Z=A\oplus\bar{B}\oplus 0\oplus CD\oplus E\oplus 1\oplus\bar{A}$

Use the following identities for XOR: $X\oplus 0=X$, $X\oplus X=0$, $X\oplus 1=\bar{X}$

These reduce the expression, by getting rid of the 0, canceling out the A's, and canceling out negations in pairs. Note that there is one 1, and two negated variables. This should leave one negation, which should be in the form of one complemented variable (it doesn't matter which one). The result is $Z=\bar{B}\oplus CD\oplus E$

8. The flow table for a 3-state, 2-input Moore type sequential machine is shown below,
along with an encoding (in terms of y1 and y2) of the states.

	AB 00	01	11	10	Z	y1	y2
1	1	2	-	-	0	0	0
2	-	3	-	1	0	0	1
3	2	-	-	2	1	1	1

Using the don't cares as best you can to simplify the logic, generate logic expressions for Y1, Y2, and Z.

The expression for Z, a function only of y1 and y2 is easy to find. At first we might write Z=y1y2, but then, on second thought we could see that the y2 terms unnecessary, since y1 is 1 only for row 3 (the 10 state is not used.) So clearly Z=y1.

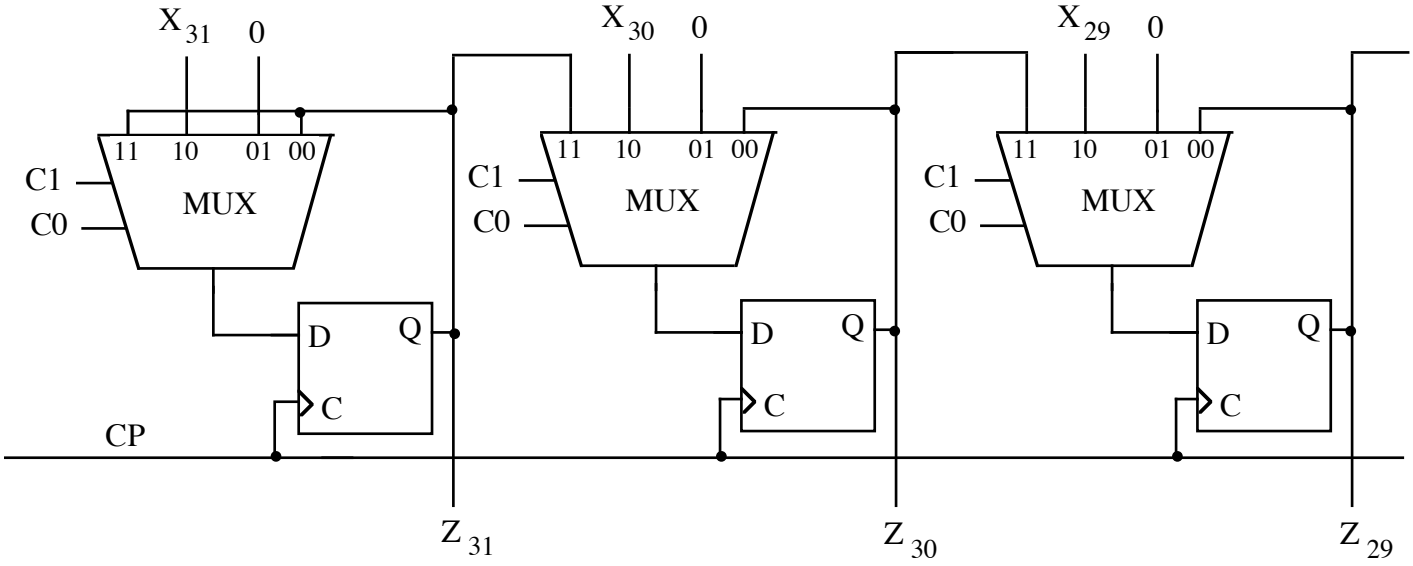
Below is the Y-matrix. Y1=1 wherever the next-state entry is 3, and Y2=1 wherever the next-state entry is 2 or 3,

	AB 00	01	11	10	y1	y2
1	00	01	--	--	0	0
2	--	11	--	00	0	1
3	01	--	--	01	1	1
					Y1	Y2

If we ignored the don't cares (including the y1y2=10 row) we would get Y1=$\bar{A}B\bar{y}1y2$, Y2=$\bar{A}B\bar{y}1+\bar{B}y1y2$.

But if we take them into account these expressions reduce to:
Y1=By2, Y2=B+y1.

9. Below are the leftmost 3 stages of a 32-bit register. For each state of the control variables C1 and C0, specify in the answer box what function the register performs.

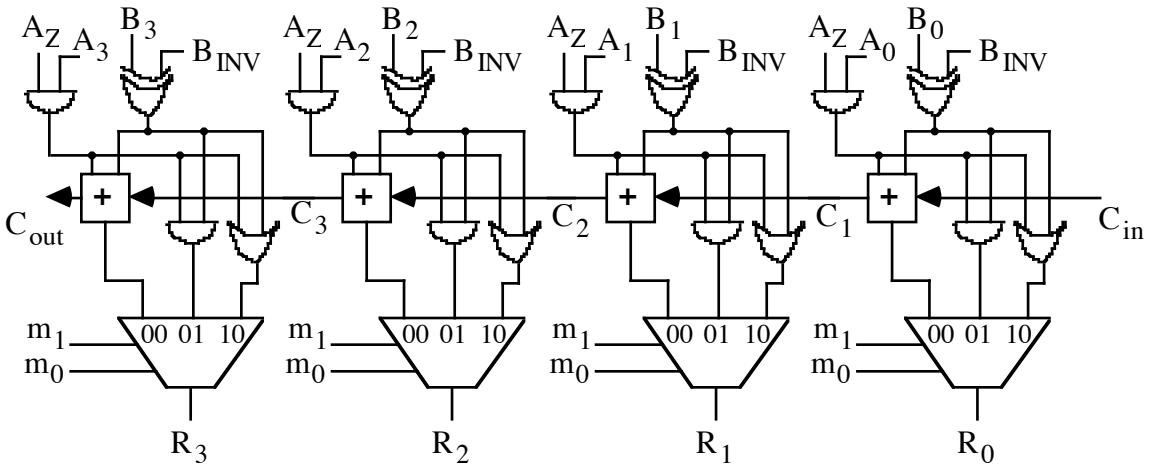


ANSWERS

C1 C0	FUNCTION
00	No change, hold previous values
01	clear to 0
10	Load register from X
11	shift right arithemetic

10. Below is a diagram of a 4-bit ALU that can perform the following 5 operations: addition, subtraction, AND, OR, and negate B (output is to be -(B)). Overflow detection is not shown. The MUX input labels are to be read as m1m0. As usual assume two's complement is used.

In the table below the diagram, specify, for each of these operations (listed in rows), what the values of the signals (listed in columns) should be. Indicate don't cares by dashes where appropriate.



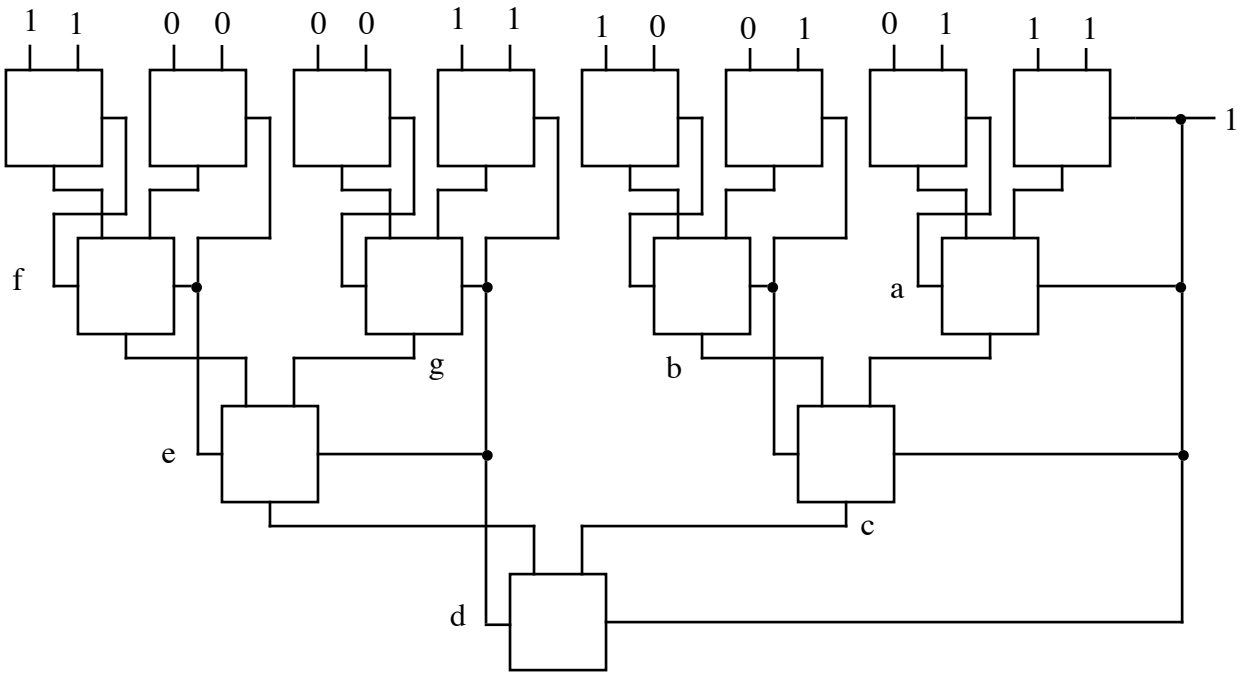
	m1	m0	Cin	Binv	Az
ADD	0	0	0	0	1
SUB	0	0	1	1	1
OR	1	0	-	0	1
AND	0	1	-	0	1
negate B	0	0	1	1	0

11. (a) Convert the two's complement number 1101 1010 to the decimal equivalent.
Since this is a negative number, we first find the two's complement. Easiest way is to is to start at right end, and copy unchanged all bits up to and including the first 1 encountered. Then invert all bits to the left of the first 1. This gives us: 0010 0110. Now convert this to decimal. For such a simple example, any number of methods will work fine. A method that works very well for bigger problems that must be solved by hand computation is to convert first to octal--very easily done by inspection, and then go from octal to decimal using the multiply and add algorithm. In this case the octal equivalent is: 46 and the decimal equivalent is 8x4+6=38. So the answer is -38.
(b) Convert the decimal number -113 to the 8-bit two's complement equivalent.
First convert the positive number 113 to binary. Again, octal is a good intermediate step. Converting 113 to octal is best done by repeated division, generating the octets as the remainders, starting at the least significant end. We get

```
  8 | 113    remainder  1
      -----
      8|14     6
         ---
           1     remainder  1.
```

So the number is 161, which we easily convert, one octet at a time, to the 8-bit binary number
01110001. To get the negative, we again find the TC, which is: 10001111.

12. An 8-bit FCLA adder is shown below, with a particular pair of numbers to be added. Specify the values of the signals, c (carry), p, and g, as appropriate, that should appear at locations labeled a-g. Note that there is a carry-in of 1 from the least significant bit end.



ANSWERS

	c	p	g
a	1		
b		1	0
c		0	1
d	1		
e	0		
f	0		
g		0	0

13. The MIPS program segment shown below implements the for statement:
for(i=0; i≤n; i++) a[i]=a[i]+1

$a0 contains the address of a[0], the first element of an n+1-element array of words.
$a1 contains n.
Five of the instructions in the MIPS segment are incomplete. For each of them, write the complete instruction in the answer box.

```
            sll $t2, $a1, 2
            add $t2, $t2, $a0
            addi $t0, $a0, -4
LOOP: addi      (a)
            lw $t1   (b)
            addi      (c)
            sw        (d)
            bne       (e)
```

The complete program segment is:

```
            sll $t2, $a1, 2
            add $t2, $t2, $a0  #$t2 get the address of the last member of the array a[n]
            addi $t0, $a0, -4
LOOP: addi      $t0, $t0, 4         (a)
            lw $t1   $t1, 0($t0)        (b)
            addi      $t1, $t1, 1         (c)
            sw        $t1, 0($t0)         (d)
            bne       $t0, $t2, LOOP  (e)
```

14. Write a short MIPS program segment that implements the pseudo-instruction bodd $a0, LOC. This instruction causes a branch to LOC if the number in $a0 is odd.

There are a number of good solutions. All depend on determining if the LSB of the number is a 1, in which case the number is odd. One approach is to capture that bit by a masking technique, i.e., use another word that has a 1 only in the LSB position. If we AND $a0 with that word, the result will be a 1 iff the LSB of $a0 is a 1, else the result will be a 0. We can then use bne to make the decision. The segment is:

```
andi $at, $a0, 1
bne $at, $zero, LOC
```

Another interesting, and equally good, solution, found by a student in the class, is to OR $a0 with 1. The result will be different from $a0 if $a0 has a 0 in the LSB position. A beq will then do the job.

```
ori $at, $a0, 1
beq $at, $a0, LOC
```

Another good solution, found by several students, is to shift $a0 left by 31 positions, so that all bits will be 0 except for the leftmost bit, which will be a 1 if the LSB of $a0 is 1 and otherwise 0. So a bne will make the decision for us.

```
sll $at, $a0, 31
bne $at, $zero, LOC
```

Another solution based on shifting is not quite as good, as it requires 3 instructions. We shift $a0 right one position and then shift left by one position. If the LSB=0, there will be no change. But if LSB=1, then the 1 will be lost as a result of the right shift.

```
sra $at, $a0, 1
sll $at, $a0, 1
bne $at, $a0, LOC
```

Some students chose to use the division instruction. This entails dividing $a0 by 2 and then testing the remainder to see if it is 1. Of course a right shift does such a division more efficiently. Division is a slow operation and furthermore, we must first put 2 in a register as there is no divide immediate instruction. Also, we have to move the remainder into a register. So this is a significantly less desirable solution. The program segment is:

```
ori $at,$zero, 2
div $a0, $at
mfhi $at
bne $at, $zero, LOC
```