

Note: Keep in mind that this review is designed as a practice exam & is not intended to be comprehensive. While studying for the exam, it would be prudent to consider variations on the questions presented here. You should also include the homework questions when reviewing for the exam. Finally, it is possible to get strong clues for answering some of the questions in the review by referring to other questions. In general, you should not depend on in the actual exam.

1. Consider the following Boolean function: $F = AB + (AC)'$

a. Write out the truth table for the function (note you may have more grid than you really need)

C	B	A	F	AB	+	AC	'						
0	0	0	1		1		1						
0	0	1	1		1		1						
0	1	0	1		1		1						
0	1	1	1	1	1		1						
1	0	0	1		1		1						
1	0	1	0		0	1							
1	1	0	1		1		1						
1	1	1	1	1	1	1							

b. Determine the function number.

c. Write out the function in canonical sum of min terms form.

$C'B'A' + C'B'A + C'BA' + C'BA + CB'A' + CBA' + CBA$

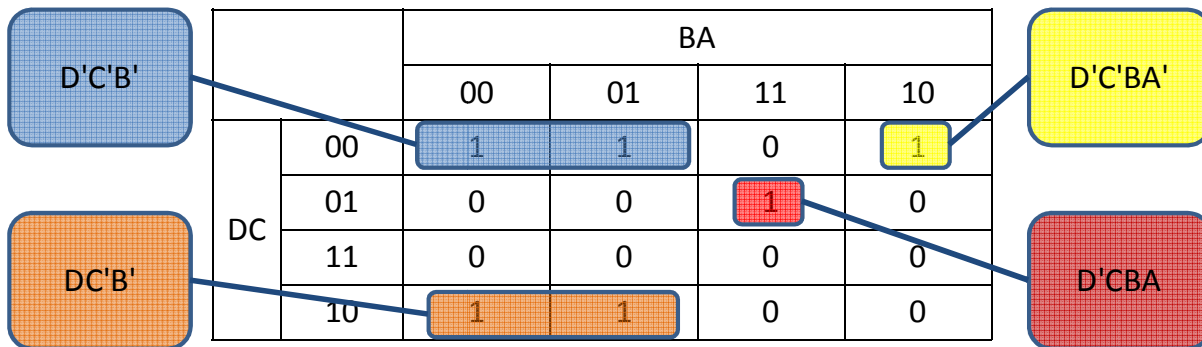
d. Without introducing new variables, produces an alternate but equivalent function.

$AB + A' + C'$ (NOTE: there are other possible answers – use a truth table to verify your answer)

2. Briefly explain what purpose(s) expressing a function in canonical sum of min terms form serves.

There are an infinite # of alternate forms of a given function. Using canonical sum of min terms provides a standard way for us to express a given function in an unambiguous manner because the canonical form of a function is unique.

3. Consider the following Karnaugh map:



a. Write the original function as a sum of minterms

$$D'C'B'A + D'C'B'A + D'CBA + D'CBA + D'C'BA' + DC'B'A' + DC'B'A$$

b. Fill in the corresponding reduction for each indicated grouping.

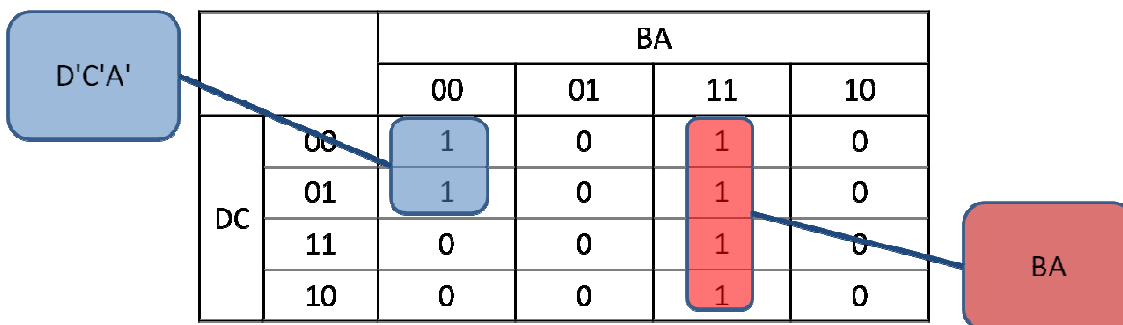
c. Write the corresponding reduced function.

$$D'C'B' + DC'B' + D'C'BA' + D'CBA$$

4. Consider the following function:

$$F = D'C'B'A' + D'CB'A' + D'C'BA + D'CB'A + DCBA + DC'BA$$

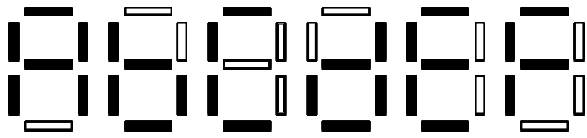
a. Use a Karnaugh map to reduce the function. Clearly and unambiguously indicate your reduction groupings.



b. Write out the reduced function

$$F = D'CA' + BA$$

5. Suppose we want to modify a seven segment LED display to include hex values as follows:



HANDY LOOKUP TABLE:							
0000	0	0100	4	1000	8	1100	C
0001	1	0101	5	1001	9	1101	D
0010	2	0110	6	1010	A	1110	E
0011	3	0111	7	1011	B	1111	f

Write out the decode function for segment 0:

segment 0 is on for symbols: 0,2,3,5,6,7,8,9,A,C,E,F (note: C was missing before; I added it here, but you could have left it out & dropped term DCB'A' from the function)

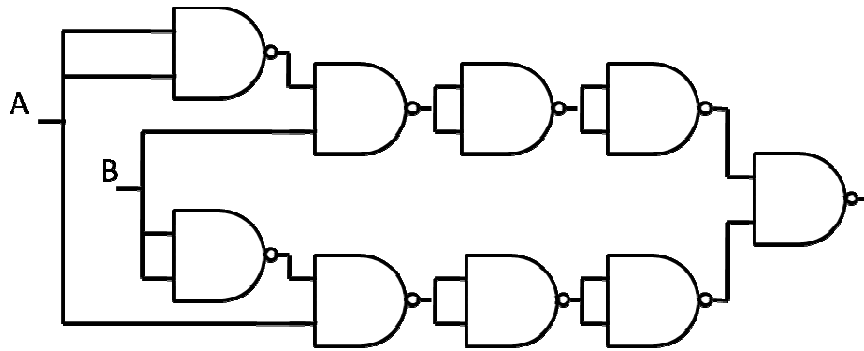
$S_0 = D'C'B'A' + D'C'BA' + D'C'BA + D'C'BA + D'CB'A + D'CBA' + D'CBA + DC'B'A' + DC'B'A + DC'BA' + DCBA' + DCBA$

6. Implement equivalents for the following function using only NAND gates:

a. XOR

B	A	XOR
0	0	0
0	1	1
1	0	1
1	1	0

So our function can be written:
 $f = BA' + AB$



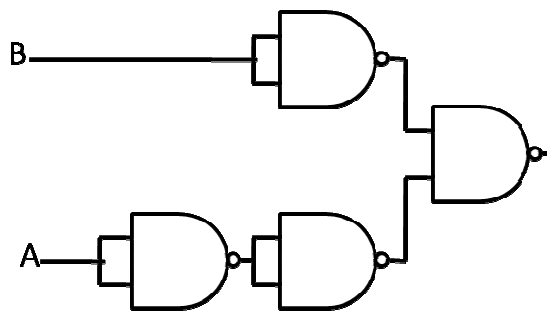
Note: we could reduce this circuit further by removing redundant NAND gates

b. $A \leq B$

B	A	$A \leq B$
0	0	1
0	1	0
1	0	1
1	1	1

So our function can be written:
 $f = B'A' + BA' + BA$

we could shorten to the following:
 $f = B + A'$



Note: we could reduce this circuit further by removing redundant NAND gates

7. Implement a 2bit by 2 bit multiplier (you may use AND, OR, NOT & NAND gates)

Start by building the truth table. We'll use DC as a two bit number & BA as a two bit number & let ZYXW be the 4 bit result. For clarity, I've added the base 10 equivalent for the table

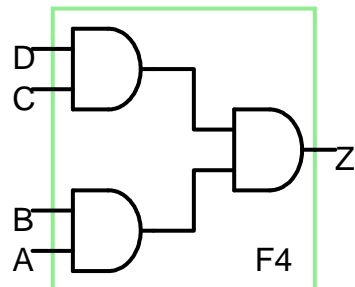
DC	BA	ZYXW	
00	00	0000	0 x 0 = 0
00	01	0000	0 x 1 = 0
00	10	0000	0 x 2 = 0
00	11	0000	0 x 3 = 0

01	00	0000	1 x 0 = 0
01	01	0001	1 x 1 = 1
01	10	0010	1 x 2 = 2
01	11	0011	1 x 3 = 3

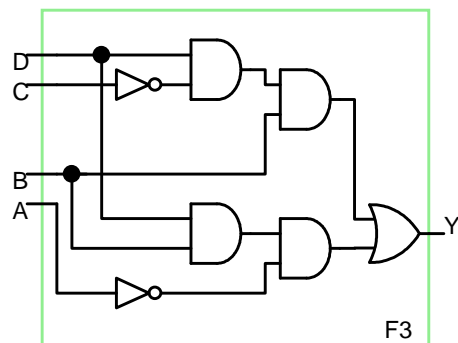
10	00	0000	2 x 0 = 0
10	01	0010	2 x 1 = 2
10	10	0100	2 x 2 = 4
10	11	0110	2 x 3 = 6

11	00	0000	3 x 0 = 0
11	01	0011	3 x 1 = 3
11	10	0110	3 x 2 = 6
11	11	1001	3 x 3 = 9

Here's the circuit for just Z; we'll call it F4 when we use it again later:



Here's the circuit for just Y; we'll call it F3 when we use it again later:



Looking at the truth table we have the following functions:

change this

$$W = D'CB'A + D'CBA + DCB'A + DCBA$$

$$X = D'CBA' + D'CBA + DC'B'A + DCB'A + DCBA'$$

$$Y = DC'BA' + DC'BA + DCBA'$$

$$Z = DCBA$$

Note that W, X & Y look unpleasant. It's worth the time to reduce them (you could use a K-Map or inspection) to the following:

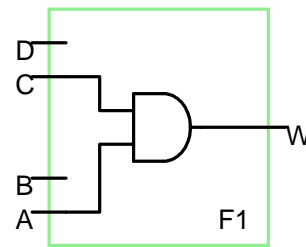
$$W = AC$$

$$X = D'CB + DB'A + CBA'$$

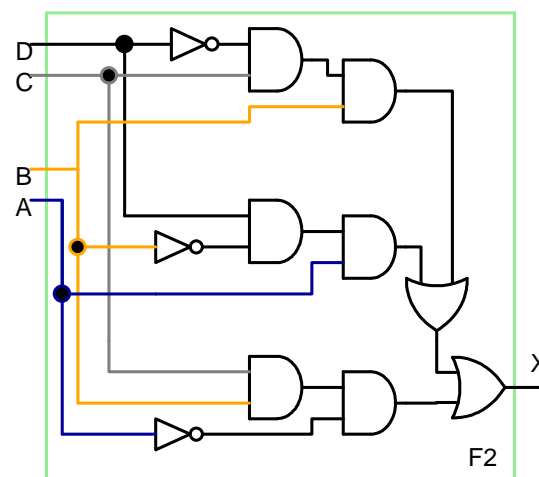
$$Y = DC'B + DBA'$$

Next, we'll build a circuit for each output

Here's the circuit for just W; we'll call it F1 when we use it again later:

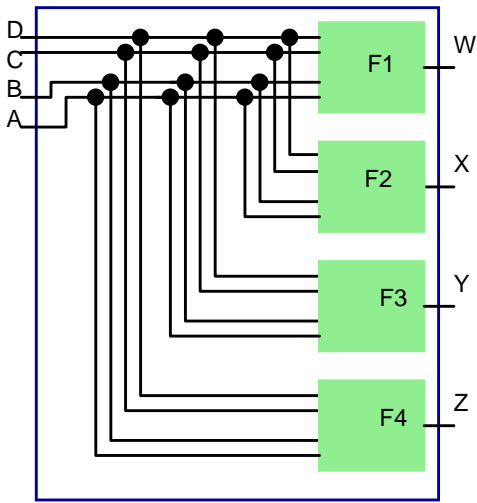


Here's the circuit for just X; we'll call it F2 when we use it again later:

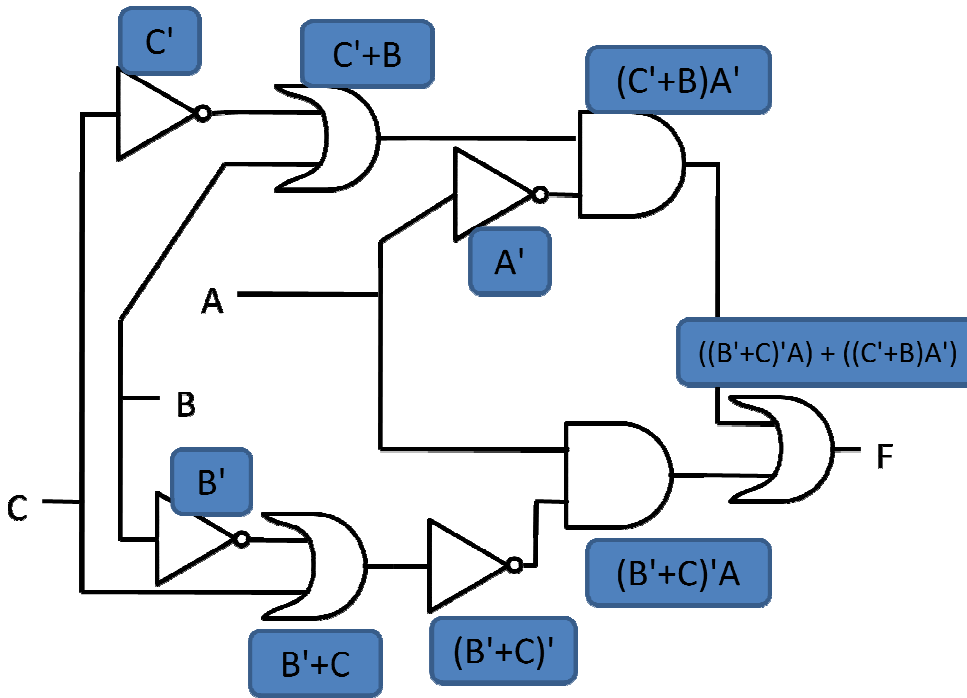


Finally, we can treat each circuit like a black box & link them together. Note: it is very likely we could have built this circuit with fewer gates, but the solution would be less intuitive.

Note: this question is significantly harder than one I'd ask on an exam.



8. Write out the formula represented by the following circuit:



final answer is $((B'+C)A) + ((C'+B)A')$

9. What information does a stack frame contain? Why do we use it to pass arguments to procedures?

Contents: non-register arguments to pass to the procedure, return address, local variables created by the procedure, registers saved by the procedure.

We pass arguments along it if:

- we need to use the C calling convention
- we need to allow a variable number of arguments (determined at runtime)

Note: this is from advanced procedures & won't be on exam 3

10. Compare & contrast C calling convention with STDCALL convention.

C Calling convention: allows a variable # of arguments & the caller is responsible for stack clean up
 STDCALL: procedure is responsible for stack clean up, it uses slightly less code & is used by the Windows API

Note: this is from advanced procedures & won't be on exam 3

11. Write some assembly code for a procedure which sums the contents of an array of WORDs as indicated below. Note: only output registers should be modified by the procedure.

Inputs:

Offset to Array	EDX
Array length	ECX

Outputs:

Sum	EAX
Flags	Flags

```
.code
main PROC
    call Clrscr
    mov EDX, OFFSET arrayX
    mov ECX, LENGTHOF arrayX
    call DumpRegs
    call SumArray
    call DumpRegs
    exit
main ENDP

; input
; EDX : offset to array of WORDs to sum up
; ECX : length of array to sum up
; output
; EAX : sum of the array
; FLAGS : set as needed
SumArray PROC
    ; unless stated otherwise, we should preserve the registers that aren't outputs
    push ECX
    push ESI

    ; zero out the total & move address into ESI
    mov EAX, 0
    mov ESI, EDX

    ; loop until we've summed the whole thing
lp:
    add AX, [ESI]; note we're adding WORDs, not DWORDs so use AX rather than EAX
    add ESI, 2
    loop lp

    ; restore (in reverse order!) registers we've messed with & return
    pop ESI
    pop ECX

    ret
SumArray ENDP

END main
```

12. Consider the following code listing, the right most column indicates the line's location in memory. Assume the stack starts out empty & diagram the stack (including stack pointer) where indicated. Note you may have more rows than you really need.

```

Proc1 PROC
00401030  push     eax
           ; A) Diagram the stack here
00401031  call    Proc2 (401036h)
           Proc1 ENDP

Proc2 PROC
00401036  push     ebx
           ; B) Diagram the stack here
00401037  call    Proc3 (40103Ch)
           Proc2 ENDP

Proc3 PROC
0040103C  pop      ebx
0040103D  push     ecx
           ; C) Diagram the stack here
0040103E  push     0
00401040  call    ExitProcess
(401072h)

Proc3 ENDP

main PROC
00401045  mov     eax, 0Ah
0040104A  mov     ebx, 0Bh
0040104F  mov     ecx, 0Ch
00401054  mov     edx, 0Dh
00401059  call   Proc1 (401030h)
0040105E  push   0
00401060  call   ExitProcess
(401072h)

```

A)

00401059
0000000A <- top of stack ptr

B)

00401059
0000000A
00401031
0000000B <- top of stack ptr

C)

00401059
0000000A
00401031
0000000B
0000000C <- top of stack ptr

13. List the individual actions performed by the CALL instruction.

push the current address (contents of EIP) onto the stack
 move the address of the procedure into the instruction pointer (EIP)