CS 354 Practice Exam 2

This is a practice exam. It has not been carefully edited (e.g., there may be minor errors).
This practice exam is designed to be harder than the exam you will be given. You should be able to complete every question on this exam. If you feel comfortable with this practice exam you should perform well on the real exam.
Come to class Monday November 11th with any questions on this exam (or from the class). This day will be a review day.

Test details: See http://pages.cs.wisc.edu/~cs354-1/exams.html
Nov 11th Tues 5:30 PM to 7:00 PM at
Van Vleck Room B102(Section 1), Room B130(Section 2)
You are allowed one (1) 8.5x11" sheet of notes (front and back)

Test topics:

- X86 functions
  - Calling convention
  - Caller vs callee registers
  - Initializing and restoring the stack
  - The call, leave, and return instructions
  - The program stack
  - Recursion in assembly
  - Stack smashing and security of the stack
- Memory hierarchy and caching
  - Relative performance of different memory technologies (SRAM, DRAM, Flash, Disk). I.e., the memory pyramid
  - Types of locality
  - Caches
    - Direct-mapped, Set-associative, Fully-associative
    - Determining the index, tag, and offset bits
    - Accessing caches
    - Hits and misses
  - Average memory access time (AMAT)
  - Writing cache conscious code, Strided access patterns
- Disks, I/O, and OS
  - Disk architecture and access time
  - Accessing files
  - Exceptions (interrupts, traps, faults, aborts)
  - Interacting with I/O devices (interrupts, DMA, etc.)
  - Privilege levels and privileged instructions
  - Re-entrant vs non-reentrant interrupts
  - Signals

Question 1: Cache design

Assume a cache with the following characteristics: $2^4 = B \quad b = 4$

1. A block/line size of 16 bytes
2. An associativity of two (2-way set associative)
3. Eight (8) total entries

	V	Tag	Data	V	Tag	Data
0	1	0xff 00				
01	1	0xff 00		1	0x80 00	
10	1	0x80 10				
11	1	0x80 10				

(A) Assume 16-bit addresses. Which bits correspond to the tag, index, and offset?

0000 0000 0000 0000

tag | index | offset

0x7010 → conflict miss

01

(B) Assume the following address stream. What is the state of the cache after the addresses have been accessed? Ignore the data that is stored/loaded. Fill in the above picture, except the data portion.

cold miss → 00

00 → hit, 01 → cold miss, → hit

0xff00, 0xff04, 0xff1c, 0x80a0, 0xff10,
0x8010, 0xff00, 0xff04, 0x80b0, 0xff14 → hit

a = 1010

cold miss 0001 | hit | hit | 1011 cold miss | 01

(C) What is the hit ratio for this address stream? (You can leave your answer as a fraction.)

$$\frac{\#hits}{\#accesses} \qquad 5/10 = 50\%$$

Question 2: Procedure Calls - Calling Conventions

Consider a new processor design called the x97 architecture, which has a total of 32 registers named r1 through r32 and the same instruction set as x86. The conventions for implementing procedures is much different in x97 when compared to the x86 architecture. In x97, the registers always contain the function arguments: starting with argument1 in r1, argument2 in r2, etc., with up to 16 arguments. In addition, r31 contains the return value, and r32 stores the value of the return address. The rest of the registers are "callee save".

2a) What exactly will be stored in the stack during a function call in the x97 architecture?
Is there still need for a stack?

local vars that don't fit in regs
local vars that are arrays/structs
vars that we need addresses of
extra args >16

~~Caller-saved~~ registers needed across call
caller-saved regs

2b. Fill in the empty slots in the diagram of the stack on the right side considering the C program on the left side is running on the x97 architecture.

X86 assembly program:

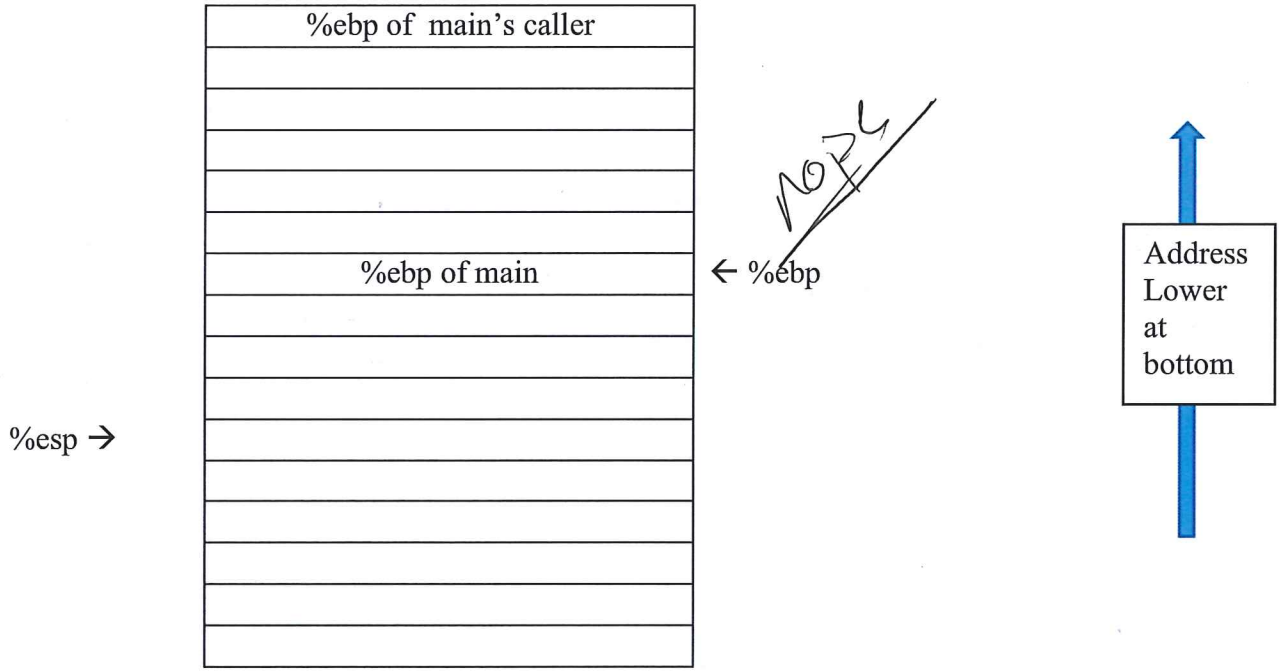
```
<f>:
push   %ebp
mov    %esp,%ebp
sub    $0x10,%esp
movl   $0x14,-0xc(%ebp) # Local var a
movl   $0x1e,-0x8(%ebp) # Local var b
movl   $0x28,-0x4(%ebp) # Local var c
mov    -0xc(%ebp),%eax  # mov a to eax
imul   -0x8(%ebp),%eax  # arith
mov    %eax,%edx        # arith
sar    $0x1f,%edx       # arith
idivl  -0x4(%ebp)       # arith
leave                   # return
ret

<main>:
push   %ebp
mov    %esp,%ebp
sub    $0x1c,%esp       #allocate space
movl   $0x3,-0xc(%ebp)  # local var a
movl   $0x4,-0x8(%ebp)  # local var b
movl   $0x5,-0x4(%ebp)  # local var c
mov    -0xc(%ebp),%eax  # set up arg 3
mov    %eax,0x8(%esp)
mov    -0x8(%ebp),%eax  # set up arg 2
mov    %eax,0x4(%esp)
mov    -0x4(%ebp),%eax  # set up arg 1
mov    %eax,(%esp)
call   8048394 <f>      # call function f
leave
ret
```

Corresponding X97 assembly program:

nope

2c) Fill in the slots in the stack layout of x97 below considering the instruction (sar $0x1f,%edx) in function f is being executed by the Processor.



2d) Write a short sequence of assembly instructions that emulates the "leave" instruction.

movl %ebp, %esp → stack pointer to beginning
popl %ebp → pop frame pointer

return?

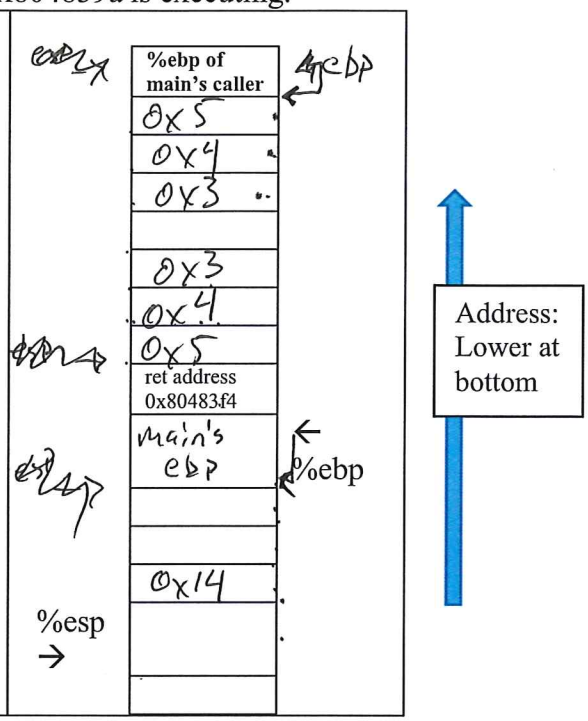
popl %eip → fixes stack (pop)
jumps to return address (addr in eip)

Question 3: Procedure Calls

3a) Fill in the empty slots in the diagram of the stack on the right side considering the C program on the left side is running and the instruction at address 0x804839a is executing.

```
08048394 <f>:
 8048394:   push   %ebp
 8048395:   mov    %esp,%ebp
 8048397:   sub    $0x10,%esp
 804839a:   movl   $0x14,-0xc(%ebp)  #local var a
 80483a1:   movl   $0x1e,-0x8(%ebp)  #local var b
 80483a8:   movl   $0x28,-0x4(%ebp)  #local var c
 80483af:   mov    -0xc(%ebp),%eax   # fetch a
 80483b2:   imul   -0x8(%ebp),%eax   #arith
 80483b6:   mov    %eax,%edx         #arith
 80483b8:   sar    $0x1f,%edx        #arith
 80483bb:   idivl  -0x4(%ebp)        #arith
 80483be:   leave                    # return
 80483bf:   ret

080483c0 <main>:
 80483c0:   push   %ebp
 80483c1:   mov    %esp,%ebp
 80483c3:   sub    $0x1c,%esp
 80483c6:   movl   $0x3,-0xc(%ebp) # Local var a
 80483cd:   movl   $0x4,-0x8(%ebp) # Local var b
 80483d4:   movl   $0x5,-0x4(%ebp) # Local var c
 80483db:   mov    -0xc(%ebp),%eax
 80483de:   mov    %eax,0x8(%esp)  #set up arg 3
 80483e2:   mov    -0x8(%ebp),%eax
 80483e5:   mov    %eax,0x4(%esp)  #set up arg 2
 80483e9:   mov    -0x4(%ebp),%eax
 80483ec:   mov    %eax,(%esp)     #set up arg 1
 80483ef:   call   8048394 <f>     # call f
 80483f4:   leave
 80483f5:   ret
```



3b) State one technique used by the compiler to detect and minimize the ill effects of stack smashing attacks during runtime.

3c) State one technique used by the Operating System to reduce the possibility of stack smashing attacks.

3d) State one precautionary measure that you as a programmer can take to avoid stack smashing attacks on the programs that you write.

scanfn scanf_s

validate user input

Check all array bounds

Question 4: Cache Organization

Assume the following is true for this question:
- The memory is byte addressable.
- Memory accesses are to 1-byte words (not 4-byte words).
- Physical addresses are 12 bits wide.
- The cache is 2-way set associative, with a 8-byte block size and 64 total cache lines.

In the following tables, **all numbers are in hexadecimal format**. The contents of the cache are as follows:

2 way set associative cache

All values are in hexadecimal notation

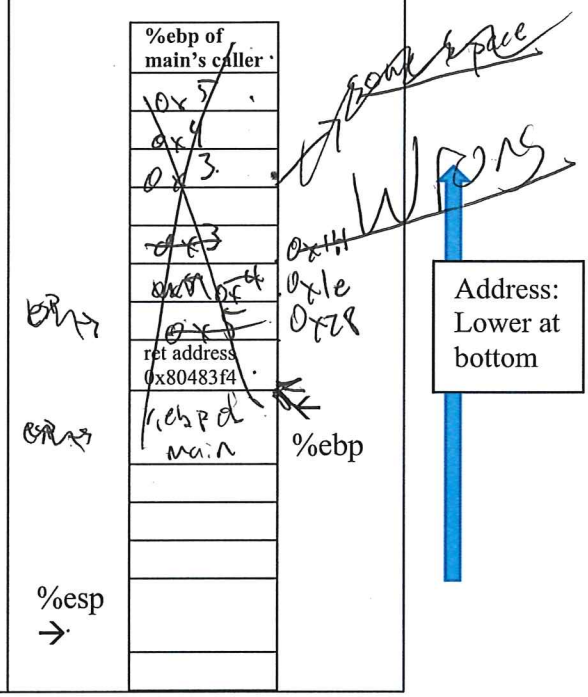
Index	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
0	2	0	34	29	34	29	39	AE	34	29	3	0	68	F2	34	29	39	AE	34	29
1	3	1	0D	8F	0D	8F	0C	3A	0D	8F	1	1	A4	DB	0D	8F	0C	3A	0D	8F
2	A	1	E2	4	E2	4	D2	4	E2	4	D	1	3C	A4	E2	4	D2	4	E2	4
3	3	0	AC	1F	AC	1F	B5	70	AC	1F	5	0	66	95	AC	1F	B5	70	AC	1F
4	0	1	60	35	60	35	19	57	60	35	1	1	8D	0E	60	35	19	57	60	35
5	A	1	B4	17	B4	17	67	DB	B4	17	6	1	DE	AA	B4	17	67	DB	B4	17
6	C	0	3F	A4	3F	A4	3A	C1	3F	A4	A	0	20	13	3F	A4	3A	C1	3F	A4
7	D	0	34	29	34	29	39	AE	34	29	9	0	68	F2	34	29	39	AE	34	29
8	3	1	0D	8F	0D	8F	0C	3A	0D	8F	0	1	A4	DB	0D	8F	0C	3A	0D	8F
9	A	1	E2	4	E2	4	D2	4	E2	4	D	1	3C	A4	E2	4	D2	4	E2	4
A	3	0	AC	1F	AC	1F	B5	70	AC	1F	5	0	66	95	AC	1F	B5	70	AC	1F
B	1	1	60	35	60	35	19	57	60	35	1	1	8D	0E	60	35	19	57	60	35
C	A	1	B4	17	B4	17	67	DB	B4	17	7	1	DE	AA	B4	17	67	DB	B4	17
D	C	0	3F	A4	3F	A4	3A	C1	3F	A4	2	0	20	13	3F	A4	3A	C1	3F	A4
E	1	0	3F	A4	3F	A4	3A	C1	3F	A4	6	0	20	13	3F	A4	3A	C1	3F	A4
F	A	1	34	29	34	29	39	AE	34	29	B	1	68	F2	34	29	39	AE	34	29
10	3	1	0D	8F	0D	8F	0C	3A	0D	8F	C	1	A4	DB	0D	8F	0C	3A	0D	8F
11	A	1	E2	4	E2	4	D2	4	E2	4	2	1	3C	A4	E2	4	D2	4	E2	4
12	E	0	AC	1F	AC	1F	B5	70	AC	1F	5	0	66	95	AC	1F	B5	70	AC	1F
13	2	1	60	35	60	35	19	57	60	35	C	1	8D	0E	60	35	19	57	60	35
14	A	1	B4	17	B4	17	67	DB	B4	17	7	1	DE	AA	B4	17	67	DB	B4	17
15	8	0	3F	A4	3F	A4	3A	C1	3F	A4	A	0	20	13	3F	A4	3A	C1	3F	A4
16	3	0	34	29	34	29	39	AE	34	29	3	1	68	F2	34	29	39	AE	34	29
17	4	1	0D	8F	0D	8F	0C	3A	0D	8F	A	1	A4	DB	0D	8F	0C	3A	0D	8F
18	A	1	E2	4	E2	4	D2	4	E2	4	4	1	3C	A4	E2	4	D2	4	E2	4
19	B	0	AC	1F	AC	1F	B5	70	AC	1F	0	0	66	95	AC	1F	B5	70	AC	1F
1A	7	1	60	35	60	35	19	57	60	35	7	1	8D	0E	60	35	19	57	60	35
1B	E	1	B4	17	B4	17	67	DB	B4	17	D	1	DE	AA	B4	17	67	DB	B4	17
1C	C	0	3F	A4	3F	A4	3A	C1	3F	A4	1	0	20	13	3F	A4	3A	C1	3F	A4
1D	A	0	3F	A4	3F	A4	3A	C1	3F	A4	C	0	20	13	3F	A4	3A	C1	3F	A4
1E	C	0	3F	A4	3F	A4	3A	C1	3F	A4	3	0	20	13	3F	A4	3A	C1	3F	A4
1F	F	0	0	FF	0	FF	B1	5F	0	FF	5	0	AC	96	0	FF	B1	5F	0	FF

Question 3: Procedure Calls

3a) Fill in the empty slots in the diagram of the stack on the right side considering the C program on the left side is running and the instruction at address 0x804839a is executing.

```
08048394 <f>:
 8048394:   push   %ebp
 8048395:   mov    %esp,%ebp
 8048397:   sub    $0x10,%esp
 804839a:   movl   $0x14,-0xc(%ebp)  #local var a
 80483a1:   movl   $0x1e,-0x8(%ebp)  #local var b
 80483a8:   movl   $0x28,-0x4(%ebp)  #local var c
 80483af:   mov    -0xc(%ebp),%eax   # fetch a
 80483b2:   imul   -0x8(%ebp),%eax   #arith
 80483b6:   mov    %eax,%edx         #arith
 80483b8:   sar    $0x1f,%edx        #arith
 80483bb:   idivl  -0x4(%ebp)        #arith
 80483be:   leave                    # return
 80483bf:   ret

080483c0 <main>:
 80483c0:   push   %ebp
 80483c1:   mov    %esp,%ebp
 80483c3:   sub    $0x1c,%esp
 80483c6:   movl   $0x3,-0xc(%ebp) # Local var a
 80483cd:   movl   $0x4,-0x8(%ebp) # Local var b
 80483d4:   movl   $0x5,-0x4(%ebp) # Local var c
 80483db:   mov    -0xc(%ebp),%eax
 80483de:   mov    %eax,0x8(%esp)  #set up arg 3
 80483e2:   mov    -0x8(%ebp),%eax
 80483e5:   mov    %eax,0x4(%esp)  #set up arg 2
 80483e9:   mov    -0x4(%ebp),%eax
 80483ec:   mov    %eax,(%esp)     #set up arg 1
 80483ef:   call   8048394 <f>     # call f
 80483f4:   leave
 80483f5:   ret
```



3b) State one technique used by the compiler to detect and minimize the ill effects of stack smashing attacks during runtime.

canary, static bounds checking

3c) State one technique used by the Operating System to reduce the possibility of stack smashing attacks.

ASLR, ADI (application data integrity)

3d) State one precautionary measure that you as a programmer can take to avoid stack smashing attacks on the programs that you write.

check all array bounds

use sqnfn / scanf_s

Question 4: Cache Organization

Assume the following is true for this question:
- The memory is byte addressable.
- Memory accesses are to 1-byte words (not 4-byte words).
- Physical addresses are 12 bits wide.
- The cache is 2-way set associative, with a 8-byte block size and 64 total cache lines.

In the following tables, **all numbers are in hexadecimal format**. The contents of the cache are as follows:

2 way set associative cache

All values are in hexadecimal notation

Index	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
0	2	0	34	29	34	29	39	AE	34	29	3	0	68	F2	34	29	39	AE	34	29
1	3	1	0D	8F	0D	8F	0C	3A	0D	8F	1	1	A4	DB	0D	8F	0C	3A	0D	8F
2	A	1	E2	4	E2	4	D2	4	E2	4	D	1	3C	A4	E2	4	D2	4	E2	4
3	3	0	AC	1F	AC	1F	B5	70	AC	1F	5	0	66	95	AC	1F	B5	70	AC	1F
4	0	1	60	35	60	35	19	57	60	35	1	1	8D	0E	60	35	19	57	60	35
5	A	1	B4	17	B4	17	67	DB	B4	17	6	1	DE	AA	B4	17	67	DB	B4	17
6	C	0	3F	A4	3F	A4	3A	C1	3F	A4	A	0	20	13	3F	A4	3A	C1	3F	A4
7	D	0	34	29	34	29	39	AE	34	29	9	0	68	F2	34	29	39	AE	34	29
8	3	1	0D	8F	0D	8F	0C	3A	0D	8F	0	1	A4	DB	0D	8F	0C	3A	0D	8F
9	A	1	E2	4	E2	4	D2	4	E2	4	D	1	3C	A4	E2	4	D2	4	E2	4
A	3	0	AC	1F	AC	1F	B5	70	AC	1F	5	0	66	95	AC	1F	B5	70	AC	1F
B	1	1	60	35	60	35	19	57	60	35	1	1	8D	0E	60	35	19	57	60	35
C	A	1	B4	17	B4	17	67	DB	B4	17	7	1	DE	AA	B4	17	67	DB	B4	17
D	C	0	3F	A4	3F	A4	3A	C1	3F	A4	2	0	20	13	3F	A4	3A	C1	3F	A4
E	1	0	3F	A4	3F	A4	3A	C1	3F	A4	6	0	20	13	3F	A4	3A	C1	3F	A4
F	A	1	34	29	34	29	39	AE	34	29	B	1	68	F2	34	29	39	AE	34	29
10	3	1	0D	8F	0D	8F	0C	3A	0D	8F	C	1	A4	DB	0D	8F	0C	3A	0D	8F
11	A	1	E2	4	E2	4	D2	4	E2	4	2	1	3C	A4	E2	4	D2	4	E2	4
12	E	0	AC	1F	AC	1F	B5	70	AC	1F	5	0	66	95	AC	1F	B5	70	AC	1F
13	2	1	60	35	60	35	19	57	60	35	C	1	8D	0E	60	35	19	57	60	35
14	A	1	B4	17	B4	17	67	DB	B4	17	7	1	DE	AA	B4	17	67	DB	B4	17
15	8	0	3F	A4	3F	A4	3A	C1	3F	A4	A	0	20	13	3F	A4	3A	C1	3F	A4
16	3	0	34	29	34	29	39	AE	30	29	3	1	68	F2	34	29	39	AE	34	29
17	4	1	0D	8F	0D	8F	0C	3A	0D	8F	A	1	A4	DB	0D	8F	0C	3A	0D	8F
18	A	1	E2	4	E2	4	D2	4	E2	4	4	1	3C	A4	E2	4	D2	4	E2	4
19	B	0	AC	1F	AC	1F	B5	70	AC	1F	0	0	66	95	AC	1F	B5	70	AC	1F
1A	7	1	60	35	60	35	19	57	60	35	7	1	8D	0E	60	35	19	57	60	35
1B	E	1	B4	17	B4	17	67	DB	B4	17	D	1	DE	AA	B4	17	67	DB	B4	17
1C	C	0	3F	A4	3F	A4	3A	C1	3F	A4	1	0	20	13	3F	A4	3A	C1	3F	A4
1D	A	0	3F	A4	3F	A4	3A	C1	3F	A4	C	0	20	13	3F	A4	3A	C1	3F	A4
1E	C	0	3F	A4	3F	A4	3A	C1	3F	A4	3	0	20	13	3F	A4	3A	C1	3F	A4
1F	F	0	0	FF	0	FF	B1	5F	0	FF	5	0	AC	96	0	FF	B1	5F	0	FF

32 sets = 2^5 → index

4 byte line = 2^3 → offset

4a. The box below shows the format of a physical address. Indicate (in the diagram) the bits that are used to determine the following: O(the block offset within the cache line), I (the cache index), and T (the cache tag).

Bit #	11	10	09	08	07	06	05	04	03	02	01	00
Use	T	T	T	T	I	I	I	I	I	O	O	O

4b. Now, for the following 12 bit addresses, perform lookup on the cache and fill in the table:

Address	Address in binary form												Tag (hex)	Index (hex)	Offset (hex)	Hit?	Word Returned (if known)
	11	10	9	8	7	6	5	4	3	2	1	0					
0x3B6	0	0	1	1	1	0	1	1	0	1	1	0	3	0x16	6	H	34
0xD3A	1	1	0	1	0	0	1	1	1	0	1	0	D	0x07	2	M	
0xABC	1	0	1	0	1	0	1	1	1	1	0	0	A	17	4	H	0C
0x97A	1	0	0	1	0	1	1	1	1	0	1	0	9	0F	2	M	

4c. State one advantage and one disadvantage of a Set associative cache when compared to a direct mapped cache.

\+ fewer conflict miss

\- more complicated logic

4e. The following table gives the parameters for a number of different caches, where m is the number of physical address bits, C is the cache size (number of data bytes), B is the block size in bytes, and E is the number of lines per set. For each cache, determine the number of cache sets (S), tag bits (t), set index bits (s), and block offset bits (b).

Cache	m	C	B	E	S	t	s	b
1	32	1024	4	256	1	30	0	2
2	16	1024	4	1	256	6	8	2
3	12	256	8	8	4	7	2	3
4	32	512	32	4	8	24	3	5

(FA)

Question 5: Interrupts and Exceptions

When a system starts running, the operating system is the first piece of code to execute, and it informs the CPU of the location of various trap and interrupt handlers via a special instruction; on x86, the ldt (or load descriptor table) instruction is used. This way, the CPU knows what code to run when one of these exceptional events occurs.

5a) On x86, the ldt instruction can only be executed when the CPU is in privileged mode. What is privileged mode?

mode where all insts can be executed
used only by OS.

5b) Why does the CPU need to be in privileged mode for ldt to run? What bad things could happen otherwise?

user could install any interrupt handlr & mess with other users / system data / code

5c) What happens when a normal user application, such as a program you wrote and compiled, tries to execute a privileged instruction such as ldt?

privilege fault / exception

5d) What are the four types of exceptions and what distinguishes one from the other?

interrupts → async. outside event faults → possibly recoverable

traps → user asked for it aborts → unrecoverable

5e) What is the difference between a re-entrant and non re-entrant interrupt handler? In what types of scenarios might a re-entrant interrupt handler be better?

re-entrant can handle an interrupt during interrupt handlr

better for interactive platforms

Question 6: Caches and Disks (mixed set of questions)

6a) State any one victim selection policy that can be used to pick the victim to be replaced in a set associative cache in order to make space for a new block of contents fetched from memory?

LRU, NMRU, Random, PLRU, 1000's of others

6b) What components make up the positioning time during an access to data on a magnetic disk?

seek, rotate, read

6c) What is the need for sophisticated flash translation layer (FTL) for Solid State Drives?

wear leveling, block erasure

6d) What is memory mapped I/O?

writing to a certain address forwards data to I/O

6e) What is Direct Memory Access (DMA) ?

data move to/from memory/I/O device by dedicated H/W not CPU

6f) What is the AMAT for a cache that has a hit rate of 20%, a miss latency of 10 cycles, and a hit latency of 3 cycles?

$3 + 0.2 \cdot 10 = 5$ cycles

6g) What are the three kinds of cache misses? Explain each one.

6h) Explain one way you may restructure your code to be more cache friendly.

spatial locality

6i) Assume there is a disk with a 5ms average seek time, a spin rate of 6000 RPM, and a transfer rate of 50 MB/s.

6i-1) What is the average rotational latency for this disk?

6i-2) What is the average total access time to read a file of 512 KB (0.5 MB) including positioning time?

6i-3) After you read this 512 KB file, you immediately want to re-read the first 5 KB. How long does this take (exactly)?

i) $\frac{6000 \text{ RPM}}{60 \text{ s/min}} = 100 \text{ rot/s} = .01 \text{ s/rot} = 10 \text{ ms}$

avg $= 10 \text{ ms}/2 = 5 \text{ ms}$

2) seek + rot + read

$5 \text{ ms} + 5 \text{ ms} + \frac{512 \text{ kB}}{50 \text{ MB/s}}$

$10 \text{ ms} + \frac{1}{100} \text{ s}$

20 ms

3) seek + rot + read

0 $10 \text{ ms} + \frac{5/12 \text{ MB}}{50 \text{ MB}}$

$10 \text{ ms} + \frac{.001}{10} \text{ s}$

10.1 ms

6s → cold → never touched this data before

Capacity → not enough room for working set

~~would be~~ have seen this before, but not in cache (not conflict)

Conflict → would have been a hit if FA cache another set has room or a line that will not be re-referenced

→ see Mark Hill's thesis