UNIVERSITY OF CALIFORNIA, LOS ANGELES UCLA

BERKELEY • DAVIS • IRVINE • LOS ANGELES • RIVERSIDE • SAN DIEGO • SAN FRANCISCO SANTA BARBARA • SANTA CRUZ

CS 33
Midterm

All answers must be written on the answer sheet (last page of the exam).

All work should be written directly on the exam, use the backs of pages if needed.

This is an open book, open notes quiz – but you cannot share books or notes. An ASCII table is on the second to last page if you need it.

I will follow the guidelines of the university in reporting academic misconduct – please do not cheat.

NAME: SOLUTIONS

ID: 8675309

Problem 1: _____

Problem 2: _____

Problem 3: _____

Total: _____

1. ***C If You Can Solve This (28 points)***: The following problem assumes the following declarations:

```
int x = rand();
int y = rand();
int z = rand();
float f= foo();  // f is not NaN
unsigned ux = (unsigned) x;
unsigned uy = (unsigned) y;
```

For the following C expressions, circle either Y or N (but not both).

	Always True?	
a. ((float) x + f) - f == (float) x	Y	[N]
Let x=1 and f=1e20, rounding will cause the lhs to evaluate to zero.		
b. (ux*uy) == (x*y)	[Y]	N
Mixed integers will be implicitly casted to unsigned		
c. ((x&8) | y) == y) ⇒ (x<<28)>0	Y	[N]
Let x =0xFFFFFFFF and y = 0x8		
d. (x^y)^x + z == y+z	[Y]	N
Property of xor, an example is in the inplace swap function from lecture and text		

Note that "⇒" represents an *implication*. A ⇒ B means that you assume A is true, and your answer should indicate whether B should be implied by A – i.e. given that A is true, is B always true?

2. ***This Problem Bytes (21 points):*** Consider the following 8 bits:

   10111100

   We will interpret these bits in three different ways (assume the above is in big endian form):

   a. An 8-bit unsigned integer

      $2^7+0^6+2^5+2^4+2^3+2^2+0^1+0^0=128+32+16+8+4=188$

   b. An 8-bit two's complement integer

      $-2^7+0^6+2^5+2^4+2^3+2^2+0^1+0^0=-128+32+16+8+4=-68$

   c. The 8-bit floating point format we covered in class

      1 0111 100
      s e f

      $(-1)^s Mx2^E$, M =1.f, E=e-bias
      bias=7
      $M=1.100=2^0+(1/2^1)=1.5$
      E=0111-7=0

      $10111100 = -1.5x2^0=-3/2$

2. ***This Problem Bytes (21 points):*** Consider the following 8 bits:

   10101100

   We will interpret these bits in three different ways (assume the above is in big endian form):

   a. An 8-bit unsigned integer

      $2^7+0^6+2^5+0^4+2^3+2^2+0^1+0^0=128+32+8+4=172$

   b. An 8-bit two's complement integer

      $-2^7+0^6+2^5+0^4+2^3+2^2+0^1+0^0=-128+32+8+4=-84$

   c. The 8-bit floating point format we covered in class

      1 0101 100 — $(-1)^s M x 2^E$, M =1.f, E=e-bias
      s e f — bias=7
      $M=1.100=2^0+(1/2^1)=1.5$
      $E=0101-7=-2$

      $10101100 = -1.5x2^{-2}=-3/8$

2. ***This Problem Bytes (21 points):*** Consider the following 8 bits:

   10011100

   We will interpret these bits in three different ways (assume the above is in big endian form):

   a. An 8-bit unsigned integer

      $2^7+0^6+0^5+2^4+2^3+2^2+0^1+0^0=128+16+8+4=156$

   b. An 8-bit two's complement integer

      $-2^7+0^6+0^5+2^4+2^3+2^2+0^1+0^0=-128+16+8+4=-100$

   c. The 8-bit floating point format we covered in class

      1 0011 100 — $(-1)^s M x 2^E$, M =1.f, E=e-bias
      s e f — bias=7
      $M=1.100=2^0+(1/2^1)=1.5$
      $E=0011-7=-4$

      $10011100 = -1.5x2^{-4}=-3/32$

2. ***This Problem Bytes (21 points):*** Consider the following 8 bits:

   10110100

   We will interpret these bits in three different ways (assume the above is in big endian form):

   a. An 8-bit unsigned integer

      $2^7+0^6+2^5+2^4+0^3+2^2+0^1+0^0=128+32+16+4=180$

   b. An 8-bit two's complement integer

      $-2^7+0^6+2^5+2^4+0^3+2^2+0^1+0^0=-128+32+16+4=-76$

   c. The 8-bit floating point format we covered in class

1	0110	100
s	e	f

      $(-1)^s Mx2^E$, M =1.f, E=e-bias
      bias=7
      $M=1.100=2^0+(1/2^1)=1.5$
      E=0110-7=-1

      10110100 = $-1.5x2^{-1}=-3/4$

3. ***And This One is a Pain in My Big Endian (51 points):*** Here's your chance to show your bomb lab skills. Below we show the disassembled function *func0()* – compiled on an ia32 machine. The function reads one integer, using scanf(). Your job is to ***provide the input to scanf() that will result in this function returning the value 42, 23, 0, 83.***

   To help you with this task – we provide part of the C code for the function below:

```
int func0 (int j, int k)
{
  int a;
  int i;

  a=0; // a will be modified by the switch statement

  scanf("%d", &i); // THE INPUT YOU ARE FIGURING OUT

  switch(i){
    …
  }

  return a;
}
```

   where the …'s above represent missing code you need to figure out.

   In addition to the input that would result in this function returning a 42, 23, 0, 83, please fill in the blanks we have on the answer key with the requested intermediate values that would help you answer this question. The next two pages contain everything you need for this problem.

Here's the function from objdump:

```
08048470 <func0>:
 8048470:       55                      push   %ebp
 8048471:       89 e5                   mov    %esp,%ebp
 8048473:       83 ec 28                sub    $0x28,%esp
 8048476:       8d 45 f4                lea    -0xc(%ebp),%eax
 8048479:       89 5d f8                mov    %ebx,-0x8(%ebp)
 804847c:       8b 5d 08                mov    0x8(%ebp),%ebx    first arg j
 804847f:       89 75 fc                mov    %esi,-0x4(%ebp)
 8048482:       8b 75 0c                mov    0xc(%ebp),%esi    second arg k
 8048485:       89 44 24 04             mov    %eax,0x4(%esp)
 8048489:       c7 04 24 64 86 04 08    movl   $0x8048664,(%esp)
 8048490:       e8 07 ff ff ff          call   804839c <__isoc99_scanf@plt>
 8048495:       83 7d f4 07             cmpl   $0x7,-0xc(%ebp)
 8048499:       76 15                   jbe    80484b0 <func0+0x40>
default 804849b:       89 d8           mov    %ebx,%eax
  0,5   804849d:       89 f1           mov    %esi,%ecx
 804849f:       8b 5d f8                mov    -0x8(%ebp),%ebx
 80484a2:       8b 75 fc                mov    -0x4(%ebp),%esi
 80484a5:       d3 e0                   shl    %cl,%eax
 80484a7:       89 ec                   mov    %ebp,%esp         a=j<<k
 80484a9:       5d                      pop    %ebp
 80484aa:       c3                      ret
 80484ab:       90                      nop
 80484ac:       8d 74 26 00             lea    0x0(%esi,%eiz,1),%esi
 80484b0:       8b 45 f4                mov    -0xc(%ebp),%eax
 80484b3:       ff 24 85 74 86 04 08    jmp    *0x8048674(,%eax,4) jump table addr
 80484ba:       8d b6 00 00 00 00       lea    0x0(%esi),%esi
   7    80484c0:       89 f0           mov    %esi,%eax
 80484c2:       8b 75 fc                mov    -0x4(%ebp),%esi
 80484c5:       21 d8                   and    %ebx,%eax         a=j&k
 80484c7:       8b 5d f8                mov    -0x8(%ebp),%ebx
 80484ca:       89 ec                   mov    %ebp,%esp
 80484cc:       5d                      pop    %ebp
 80484cd:       c3                      ret
 80484ce:       66 90                   xchg   %ax,%ax
   1    80484d0:       8d 04 1e        lea    (%esi,%ebx,1),%eax a=j+k
 80484d3:       8b 5d f8                mov    -0x8(%ebp),%ebx
 80484d6:       8b 75 fc                mov    -0x4(%ebp),%esi
 80484d9:       89 ec                   mov    %ebp,%esp
 80484db:       5d                      pop    %ebp
 80484dc:       c3                      ret
 80484dd:       8d 76 00                lea    0x0(%esi),%esi
   2    80484e0:       89 d8           mov    %ebx,%eax
 80484e2:       8b 5d f8                mov    -0x8(%ebp),%ebx
 80484e5:       29 f0                   sub    %esi,%eax         a=j-k
 80484e7:       8b 75 fc                mov    -0x4(%ebp),%esi
 80484ea:       89 ec                   mov    %ebp,%esp
 80484ec:       5d                      pop    %ebp
 80484ed:       c3                      ret
 80484ee:       66 90                   xchg   %ax,%ax
   3    80484f0:       89 d8           mov    %ebx,%eax
 80484f2:       89 f1                   mov    %esi,%ecx
 80484f4:       d3 f8                   sar    %cl,%eax          a=j>>k
 80484f6:       09 f3                   or     %esi,%ebx
 80484f8:       8b 75 fc                mov    -0x4(%ebp),%esi   a+=j|k
```

```
80484fb:       01 d8                   add    %ebx,%eax
80484fd:       8b 5d f8                mov    -0x8(%ebp),%ebx
8048500:       89 ec                   mov    %ebp,%esp
8048502:       5d                      pop    %ebp
8048503:       c3                      ret
8048504:       8d 74 26 00             lea    0x0(%esi,%eiz,1),%esi
8048508:       31 c0                   xor    %eax,%eax
804850a:       eb ea                   jmp    80484f6 <func0+0x86>
804850c:       8d 74 26 00             lea    0x0(%esi,%eiz,1),%esi
8048510:       89 f0                   mov    %esi,%eax
8048512:       8b 75 fc                mov    -0x4(%ebp),%esi
8048515:       31 d8                   xor    %ebx,%eax
8048517:       8b 5d f8                mov    -0x8(%ebp),%ebx
804851a:       89 ec                   mov    %ebp,%esp
804851c:       5d                      pop    %ebp
804851d:       c3                      ret
```

4 6 a=j^k

And here is some gdb interaction which should prove useful:

```
(gdb) break *0x8048470
Breakpoint 1 at 0x8048470
(gdb) run
Starting program

Breakpoint 1, 0x08048470 in func0 ()
(gdb) p $esp
$1 = (void *) 0xffffd3dc
(gdb) p $ebp
$2 = (void *) 0xffffd408
(gdb) x/32x 0xffffd3dc
0xffffd3dc:     0x08048572      0x00000030      0x0000001a      0x0000000a
0xffffd3ec:     0x006fdff4      0x080485b0      0x080483b0      0x006fdff4
0xffffd3fc:     0x00000000      0x080485b0      0x00000000      0xffffd488
0xffffd40c:     0x00581ce6      0x00000003      0xffffd4b4      0xffffd4c4
0xffffd41c:     0xf7ffd428      0x080483b0      0xffffffff      0x00567fc4
0xffffd42c:     0x0804827e      0x00000001      0xffffd470      0x00557a05
0xffffd43c:     0x00568ab0      0xf7ffd708      0x006fdff4      0x00000000
0xffffd44c:     0x00000000      0xffffd488      0xf67726a3      0xb9e6515c
(gdb) x/16x 0x8048660
0x8048660 <__dso_handle>:       0x00000000      0x74006425      0x69747365      0x2020676e
0x8048670 <__dso_handle+16>:    0x000a6425      0x0804849b      0x080484d0      0x080484e0
0x8048680 <__dso_handle+32>:    0x080484f0      0x08048508      0x0804849b      0x08048510
0x8048690 <__dso_handle+48>:    0x080484c0      0x3b031b01      0x00000020      0x00000003
```

breakpoints stop before executing the current line/instr, stack setup for func0 has not been executed.

ret addr arg1:j arg2:k These values may be different

addr 0x8048674 jump table

0 1 2 3 4 5 6 7

Hint – don't forget that gdb reverses byte ordering within each 4-byte chunk. So in the following dump:

```
(gdb) x/4x 0x00111110
0x111110:   0x33221100  0x77665544  0xBBAA9988  0xFFEEDDCC
```

This prints out 16 bytes of memory starting at address 0x111110. In this example, the 16 bytes of memory starting at 0x111110 would contain, in order from lowest address (0x111110) to highest address (0x11111F):

00112233445566778899AABBCCDDEEFF

So address 0x111110 contains the byte 0x00, address 0x111111 contains the byte 0x11, address 0x111112 contains the byte 0x22, and so on. So in terms of just the least significant hex place of the address, gdb is actually printing out addresses in the following order:

3 2 1 0 7 6 5 4 B A 9 8 F E D C

This is useful when reading words, but can be confusing for other values.

Return Val	42	23	0	83
j	0x30 = 48	0x41 = 65	0x22 = 34	0x36 = 54
k	0x1a = 26	0x2a = 42	0x10 = 16	0x1d = 29
default addr	0x0804849b			
case	6	2	7	1