计算机系统导论期中考试　　2012 年 11 月 7 日

姓名：　　学号：　　成绩：

Problem A -- Multiple Choice (18 pts, 1pt or 2pts each)

1. Let int x = −31/8 and int y = −31 >> 3. What are the values of x and y? **(m-f10-a 1-5) C**
   a) x = −3, y = −3
   b) x = −4, y = −4
   c) x = −3, y = −4
   d) x = −4, y = −3

2. For which values can X not be equal to Z in the code below (circle all that apply) **(f-f09 1-B) A&D**
   int X = CONSTANT;
   float Y = X;
   int Z = Y;
   a) For large positive values of CONSTANT (e.g.,>1,000,000,000)
   b) For large negative values of CONSTANT (e.g.,>-100)
   c) For small positive values of CONSTANT (e.g.,<100)
   d) For small negative values of CONSTANT (e.g.,<-1,000,000,000)
   e) None of the above (i.e., X==Z in all of these cases)

3. Consider the following code, what is the output of the printf? **(m-f10-a 1-1) B**
   int x = 0x15213F10 >> 4;
   char y = (char) x;
   unsigned char z = (unsigned char) x;
   printf("%d, %u", y, z);
   a) -241, 15
   b) -15, 241
   c) -241, 241
   d) -15, 15

4. After executing the following code, which of the variables are equal to 0? **(m-f11 1-3) CD or CDE**
   a) unsigned int a = 0xffffffff;
   b) unsigned int b = 1;
   c) unsigned int c = a + b;
   d) unsigned long d = a + b;
   e) unsigned long e = (unsigned long)a + b

5. How does x86 assembly store the return value when a function is finished? **(f-f10 1-4) B**
   a) The ret instruction stores it in a special retval register.
   b) By convention, it is always in %eax.
   c) It is stored on the stack just above the (%ebp) of the callee.
   d) It is stored on the stack just above all the arguments to the function

6. Which of the following is FALSE concerning x86-64 architecture? **(f-f10 1-9) D**
   a) A double is 64 bits long.
   b) Registers are 64 bits long.

c) Pointers are 64 bits long.
d) Pointers point to locations in memory that are multiples of 64 bits apart.

7. Denormalized floating point numbers are **(m-f10-a 1-18) A**
   a) Very close to zero (small magnitude)
   b) Very far from zero (large magnitude)
   c) Un-representable on a number line
   d) Zero.

8. What is the minimum (most negative) value of a 32-bit two's complement integer? **(m-f11 1-1) C**
   a) -2^{32}
   b) $-2^{32} + 1$
   c) -2^{31}
   d) $-2^{31} + 1$

9. What is the difference between the mov and lea instructions? **(m-f11 1-2) B**
   a) lea dereferences an address, while mov doesn't.
   b) mov dereferences an address, while lea doesn't.
   c) lea can be used to copy a register into another register, while mov cannot.
   d) mov can be used to copy a register into another register, while lea cannnot.

10. Which of the following 8 bit floating point numbers (1 sign, 3 exponent, 4 fraction) represent NaN? **(m-s11 1-2) B**
    a) 1 000 1111
    b) 0 111 1111
    c) 0 100 0000
    d) 1 111 0000

11. *%rsp* is 0xdeadbeefdeadd0d0. What is the value in *%rsp* after the following instruction executes? pushq %rbx **(m-s11 1-3) D**
    a) 0xdeadbeefdeadd0d4
    b) 0xdeadbeefdeadd0d8
    c) 0xdeadbeefdeadd0cc
    d) 0xdeadbeefdeadd0c8

12. Consider an int *a and an int n. If the value of %ecx is a and the value of %edx is n, which of the following assembly snippets best corresponds to the C statement return a[n]? **(m-s11 1-1) C**
    a) ret (%ecx,%edx,4)
    b) leal (%ecx,%edx,4),%eax
       ret
    c) mov (%ecx,%edx,4),%eax
       ret
    d) mov (%ecx,%edx,1),%eax
       ret

13. What is the C equivalent of mov 0x10 (%rax,%rcx,4), %rdx **(2pts) (m-s11 1-8) C**
    a) rdx = rax + rcx + 4 + 10
    b) *(rax + rcx + 4 + 10) = rdx
    c) rdx = *(rax + rcx*4 + 0x10)
    d) rdx = *(rax + rcx + 4 + 0x10)

14. In what section of an ELF binary are initialized variables located? **(2pts) (e2b-s11) B**
    a) .symtab
    b) .data
    c) .bss
    d) .text

15. A system uses a two-way set-associative cache with 16 sets and 64-byte blocks. Which set does the byte with the address 0xdeadbeef map to? **(2pts) (e2-f10) B**
    a) Set 7
    b) Set 11
    c) Set 13
    d) Set 14

Problem B -- Bits & Bytes, Int, and FP (20 pts)

1. For each of the following propositions, write in all comparisons that make it always true among the four possibilities:

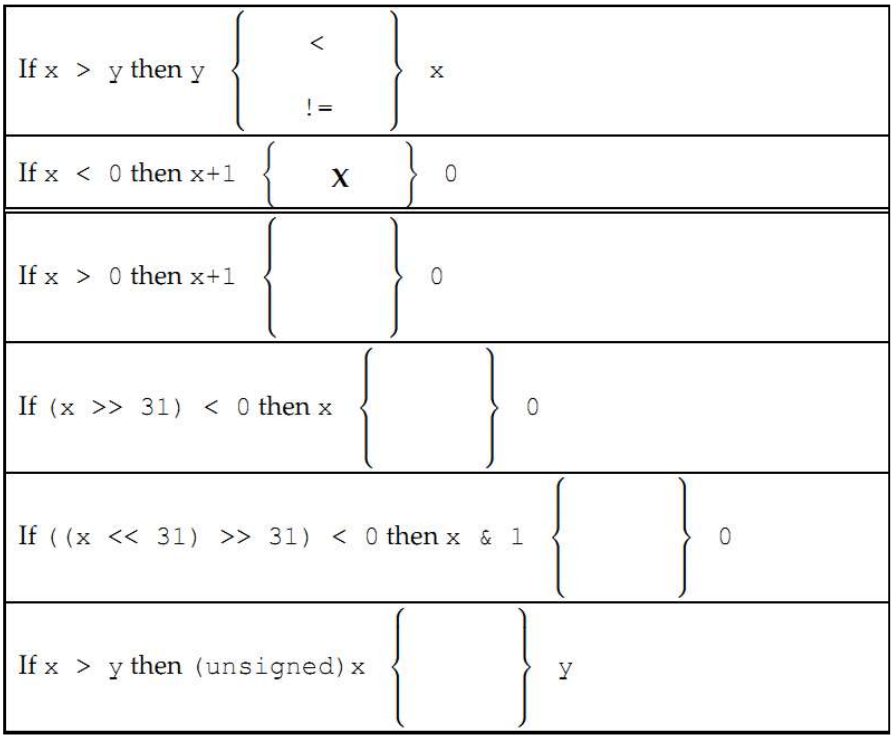
   < > == !=

   If none are guaranteed to hold, please indicate that explicitly by marking it with an X. We have filled in the first two for you as examples. Assume the variables are declared with

   int x, y;

   and initialized to some unknown values. You may assume that int's are 32 bits wide, char's are 8 bits wide and that right shift is arithmetical on signed numbers and logical on unsigned numbers. **(m-s07 1) (4pts)**

   (1) !=
   (2) <
   !=
   (3) >
   !=
   (4) !=



2. Floating point encoding. Consider the following 5-bit floating point representation based on the IEEE floating point format. This format does not have a sign bit – it can only represent nonnegative numbers.

• There are k = 3 exponent bits. The exponent bias is 3.

• There are n = 2 fraction bits.

Recall that numeric values are encoded as a value of the form $V = M \times 2^E$, where E is the exponent after biasing, and M is the significant value. The fraction bits encode the significand value M using either a denormalized (exponent field 0) or a normalized representation (exponent field nonzero). The exponent E is given by E = 1 − Bias for denormalized values and E = e − Bias for normalized values, where e is the value of the exponent field exp interpreted as an unsigned number.

Below, you are given some decimal values, and your task it to encode them in floating point format. In addition, you should give the rounded value of the encoded floating point number. To get credit, you must give these as whole numbers (e.g., 17) or as fractions in reduced form (e.g., 3/4). Any rounding of the significand is based on round-to-even, which rounds an unrepresentable value that lies halfway between two representable values to the nearest even representable value.

(8pts)

Value	FP bits	Rounded value
3	100 10	3 normalized, exact
9	110 00	8 normalized, round down, because 10 is odd
3/16	000 11	3/16 exact, denorm
15/2	110 00	8 normalized, round up, change exponent

Value	Floating Point Bits	Rounded value
9/32	001 00	1/4
3		
9		
3/16		
15/2		

3. Conversion from float to int is a notoriously expensive operation. Not all processors do this in hardware. A clever technique uses the normalization step in double-precision floats. Imagine if you could normalize a (non-negative) floating point number so that the low-order significant bit represents 2^0. Then the significant would be an integer. **(8 pts) (f-f09 7)**

Assume a double where, after taking the exponent into account, the low-order bit represents 1 (2^0). What does the "implied leading 1" of the significant represent?
2^{52}

Now, assume a double x is known to be in the range 0<=x<232. To force the low-order bit of x to represent 1(20), you should add by ___**2^{52}**_____?

After this operation, you can store the double to memory and read the (circle one:)

- *Upper 32 bits* (the end with the sign bit == 0), or
- ***Lower 32 bits* (the other end)**

as an unsigned integer to get the integer value of the original double. This algorithm will round (circle one)

- *toward zero,*
- *up,*
- *down,*
- ***to the nearest even.***

Hints:

double-precision significant has 52 bits
double-precision exponent has 11 bits
double-precision sign-bit is (of course) 1 bit

Problem C Machine Programming (25 pts)

1. The function below is hand-written assembly code for a sorting algorithm. Fill in the blanks on the next page by converting this assembly to C code **(m-s09-4) (13pts)**

```
    .globl mystery_sort     # exports the symbol so other .c files
                            # can call the function

    mystery_sort:
            jmp     loop1_check

    loop1:
            xor     %rdx, %rdx
            mov     %rsi, %rcx
            jmp     loop2_check

    loop2:
            mov     (%rdi, %rcx, 8), %rax
            cmp     %rax, (%rdi, %rdx, 8)
            jg      loop2_check
            mov     %rcx, %rdx

    loop2_check:
            dec     %rcx
            test    %rcx, %rcx
            jnz     loop2

            dec     %rsi
            mov     (%rdi, %rsi, 8), %rax
            mov     (%rdi, %rdx, 8), %rcx
            mov     %rcx, (%rdi, %rsi, 8)
            mov     %rax, (%rdi, %rdx, 8)

    loop1_check:
            test    %rsi, %rsi
            jnz     loop1

            ret
```

```
void mystery_sort (long* array, long len)
{
 long a, b, tmp;
  while(len>0)
 {
    a = 0;
  for(b=len-1; b>0; b--)
  {
   if(array[b]>array[a])
   {
     a=b;
   }
  }
  len--;
  tmp = array[len];
  array[len] = array[a];
  array[a] = tmp;
```

```
void mystery_sort (long* array, long len)
{
  long a, b, tmp;

  while (_____ > _____)
  {

    a = _____;

    for (b = _____; b > _____; b--)
    {

      if (array[_____] > array{_____])
      {

         _____ = _____;
      }
    }

    len--;

    tmp = array[_____];

    array[_____] = array[_____];

    array[_____] = tmp;
  }
}
```

2. Below is some assembly code to a famous algorithm. Please briefly read the code then answer the questions on the following page. **(m-s09-6) 12pts**

```
0000000000400498 <mystery>:
  400498:  41 b8 00 00 00 00  mov    $0x0,%r8d
  40049e:  eb 22              jmp    4004c2 <mystery+0x2a>
  4004a0:  89 c8              mov    %ecx,%eax
  4004a2:  c1 e8 1f           shr    $0x1f,%eax
  4004a5:  01 c8              add    %ecx,%eax
  4004a7:  d1 f8              sar    %eax          ; arith shift right 1 bit
  4004a9:  42 8d 0c 00        lea    (%rax,%r8,1),%ecx
  4004ad:  48 63 c1           movslq %ecx,%rax
  4004b0:  8b 04 87           mov    (%rdi,%rax,4),%eax
  4004b3:  39 d0              cmp    %edx,%eax
  4004b5:  7d 05              jge    4004bc <mystery+0x24>
  4004b7:  41 89 c8           mov    %ecx,%r8d
  4004ba:  eb 06              jmp    4004c2 <mystery+0x2a>
  4004bc:  39 d0              cmp    %edx,%eax
  4004be:  7e 10              jle    4004d0 <mystery+0x38>
  4004c0:  89 ce              mov    %ecx,%esi
  4004c2:  89 f1              mov    %esi,%ecx
  4004c4:  44 29 c1           sub    %r8d,%ecx
  4004c7:  85 c9              test   %ecx,%ecx
  4004c9:  7f d5              jg     4004a0 <mystery+0x8>
  4004cb:  b9 ff ff ff ff     mov    $0xffffffff,%ecx
  4004d0:  89 c8              mov    %ecx,%eax
  4004d2:  c3                 retq
```

a) Please write a single line of C code to represent the instruction lea (%rax,%r8,1),%ecx (Use C variables named rax,r8, and ecx, you can ignore types).

ecx = rax + r8

b) Please write a single line of C code to represent the instruction mov (%rdi,%rax,4),%eax (Use C variables named rdi and rax, you can ignore types).

eax = *(rdi + rax) // rdi is int *

eax = *(rdi + 4 * rax) // rdi is char *

c) Commonly found in assembly is the leave instruction; why is that instruction not in this code?

no stack to return from, no push %ebp

d) You learned about two different architectures in class, IA32 and x86 64. What architecture is this code written for and what major downside would occur from using the other architecture

x86_64

32-bit has less registers than 64-bit mode, so local variables would need to be stored on the stack. Stack accessing takes extra instructions and extra time.

Problem D Stack (10 pts)

Stack discipline. Consider the following C code and assembly code for a recursive function . **(m-f11-5) 10pts**

```
int gcd(int a, int b)
{
    if(!b)
    {
        return a;
    }

    return gcd(b, a % b);
}
```

```
0x08048394 <+0>:    push   %ebp
0x08048395 <+1>:    mov    %esp,%ebp
0x08048397 <+3>:    sub    $0x10,%esp
0x0804839a <+6>:    mov    0x8(%ebp),%eax
0x0804839d <+9>:    mov    0xc(%ebp),%ecx
0x080483a0 <+12>:   test   %ecx,%ecx
0x080483a2 <+14>:   je     0x80483b7 <gcd+35>
0x080483a4 <+16>:   mov    %eax,%edx
0x080483a6 <+18>:   sar    $0x1f,%edx
0x080483a9 <+21>:   idiv   %ecx
0x080483ab <+23>:   mov    %edx,0x4(%esp)
0x080483af <+27>:   mov    %ecx,(%esp)
0x080483b2 <+30>:   call   0x8048394 <gcd>
0x080483b7 <+35>:   leave
0x080483b8 <+36>:   ret
```

Imagine that a program makes the procedure call gcd(213, 18). Also imagine that prior to the invocation, the value of %esp is 0xffff1000—that is, 0xffff1000 is the value of %esp immediately before the execution of the call instruction.

A. Note that the call gcd(213, 18) will result in the following function invocations: gcd(213, 18), gcd(18, 15), gcd(15, 3), and gcd(3, 0). Using the provided code and your knowledge of IA32 stack discipline, fill in the stack diagram with the values that would be present immediately before the execution of the leave instruction for gcd(15, 3). Supply numerical values wherever possible, and cross out each blank for which there is insufficient information to complete with a numerical value.

Hints: The following set of C style statements describes an approximation of the operation of the instruction idiv %ecx, where '/' is the division operator and '%' is the modulo operator:

%eax = %eax / %ecx

%edx = %eax % %ecx

Also, recall that leave is equivalent to movl %ebp, %esp; popl %ebp

Part A:

unknown	0xffff1008
18	0xffff1004
213	0xffff1000
unknown	0xffff0ffc
unknown	0xffff0ff8
unknown	0xffff0ff4

unknown	0xffff0ff0
15	0xffff0fec
18	0xffff0fe8
0x080483b7	0xffff0fe4
0xffff0ff8	0xffff0fe0
unknown	0xffff0fdc
unknown	0xffff0fd8
3	0xffff0fd4
15	0xffff0fd0
0x080483b7	0xffff0fcc
0xffff0fe0	0xffff0fc8
unknown	0xffff0fc4
unknown	0xffff0fc0
0	0xffff0fbc
3	0xffff0fb8
unknown	0xffff0fb4
unknown	0xffff0fb0

B. What are the values of %esp and %ebp immediately before the execution of the ret instruction for gcd(15, 3)?

esp: 0xffff0fcc
ebp: 0xffff0fe0

Problem E -- The Hit or Miss Question (15 pts)

Given a 32-bit Linux system that has a 2-way associative cache of size 128 bytes with 32 bytes per block. Long longs are 8 bytes. For all parts, assume that table starts at address 0x0.

```
int i;

int j;

long long table[4][8];

for (j = 0; j < 8; j++) {

        for (i = 0; i < 4; i++) {

                table[i] [j] = i + j;

        }

}
```

A. This problem refers to code sample 1. In the table below write down in each space whether that element's access will be a hit or a miss. Indicate hits with a 'H' and misses with a 'M'.

	0	1	2	3	4	5	6	7
0	M	M	M	M	M	M	M	M
1	M	M	M	M	M	M	M	M
2	M	M	M	M	M	M	M	M
3	M	M	M	M	M	M	M	M

What is the miss rate of this code sample?

1

```
int i;

int j;

int table[4][8];

for (j = 0; j < 8; j++) {

        for (i = 0; i < 4; i++) {

                table [i] [j] = i + j;

                table[j][i]?

        }

}
```

B. This problem refers to code sample above. In the table below write down in each space whether that element's access will be a hit or a miss. Indicate hits with a 'H' and misses with a 'M'.

	0	1	2	3	4	5	6	7
0	**M**	**H**	**H**	**H**	**H**	**H**	**H**	**H**
1	**M**	**H**	**H**	**H**	**H**	**H**	**H**	**H**
2	**M**	**H**	**H**	**H**	**H**	**H**	**H**	**H**
3	**M**	**H**	**H**	**H**	**H**	**H**	**H**	**H**

What is the miss rate of this code sample?

1/8

C. One code sample performs better than the other. Why is this?

The second sample performs better than the first because it makes better use of the cache. In the first, heavy conflict results in a large number of evictions. In the second, the entire array fits in the cache, and the only misses are cold misses.

Problem F – Linking (12 pts)

For each of the following code snippets, write down all symbols in the resulting object files from compilation. Write whether it is a weak global, strong global, or local variable, and what section of the final compiled ELF binary the variable will go into. Fill in the value if you have enough information to determine the value.

A.

main.c

```
#include <stdio.h>

int x;
int y;
int z = 0;

int main() {
  printf("%x\n", x);
  printf("%x\n", y);
  x = 0xdeadbeef;
  printf("%x\n", x);
  printf("%x\n", y);
  return 0;
}
```

foo.c

```
short x = 5;
short y = 2;
```

File	Symbol	Strength / scope	Value	ELF Section
Main.o	X	Weak global	-	.data
	Y	Weak global	-	.data
	Z	Strong global	0	.data
	Main	Strong global	-	.text
foo.o	X	Strong global	5	.data
	Y	Strong global	2	.data