CSCI 2400, Spring 2014

Midterm Exam

Instructions:

- Make sure that your exam is not missing any sheets, then write your full name on the front. Put your name or student ID on each page.
- Write your answers in the space provided below the problem. If you make a mess, clearly indicate your final answer.
- You can use a *single* sheet of your own notes and the provided notes page. Please attach your single sheet of notes to your exam when you're done. You can not use a computer or calculator. Good luck! :(:

Problem	Page	Possible	Score
1	1	20	
2	2	13	
3	3	16	
4	4	15	
5	5	18	
6	7	18	
Total		100	0x64, 1100100

1. **[20 Points] We are running programs on a machine with the following characteristics:**

   - **Values of type int are 32 bits. They are represented in two's complement, and they are right shifted arithmetically. Values of type unsigned are 32 bits.**
   - **The random() function returns successive pseudo-random numbers in the range from 0 to 2^{31}– 1.**

We generate arbitrary values x, y, and z, and convert them to other forms as follows:/* Create some arbitrary values */ int x = random(); int y = random(); int z = random();

/* Convert to other forms */ unsigned ux =
(unsigned) x; unsigned uy = (unsigned) y;

For each of the following C expressions, you are to indicate whether or not the expression *always* yields 1. If so, circle "Y". If not, circle "N". You will be graded on each problem as follows:

- **If you circle no value, you get 0 points.**
- **If you circle the right value, you get 4 points.**
- **If you circle the wrong value, you get –2 points (so don't just guess wildly).**

https://robot.bolink.org/ebooks/Computer%20Systems%20-%20A%20Programmer%E2%80%99s%20Perspective%20Solutions%20Manual.pdf <- this is for old edition of book, I think

Expression	Always True?
(x<y) == (-x>-y)	Yes(1)
((x+y)<<4) + y-x == 17*y+15*x	Yes
~x+~y+1 == ~(x+y)	Yes
ux-uy == -(y-x)	Yes
(x >= 0) || (x < ux)	yes(5) (give an example if no)

(1) The solution manual entry to problem 2.54 has the counterexample x = Tmin, y = 0, but that can't happen here: x and y are nonnegative
(5) Remember x is nonnegative, so the first part always evaluates as 1. (x must be 0 or greater than 0)
*These question are tricky because It would always have to be true. So we have to find just one case to prove that it is not true.
Note: x,y are in the range of 0 to 2^31-1 x = T_max is a possibility but not x = any negative number

2. **[13 Points] Assume we are running code on a 5-bit machine using two's complement arithmetic for signed integers. Also assume that TMax is the maximum integer, TMin is the minimum integer. Fill in the empty boxes in the table below. The following definitions are used in the table:**

   int y = 9 int x =7;

   Note: You need not fill in entries marked with "–".

   Each blank space is 1 point.

   In the column labeled "Over/Under", you should indicate if an overflow (carry out of the highest bit) or underflow (borrow from the highest bit) occurred.

*T-min = 10000 = -16, -x = -7 = 11001, T-min - x = 101001 = -23 <- but this is underflow, so you must add 2^5 =32 or just truncate it to 5 bits: 1| 01001 -> 01001 = 9.

Expression	Decimal Representation	Hex Representation	Over/Under?
–	13	**0x0d**	**–**
–	**-7**	0x19 how you can haz hex in negative?? -see † for explanation	**–**
y	**9**	0x9	**–**
x+y	-16	-0x10	over
x-y	**-2**	0x1e	none
TMax + x	15+7 = -10	0x16 (-16 + 6)	over
TMin-x	9*	0x09*	under

† : to convert negative decimals to hex, split the binary values into groups of 4. For example, -7 ⟶ 11001, split into groups of 4, you get 0001 | 1001. Each of these binary values has a hex representation

, 0001 ⟶ 0x1 and 1001 ⟶ 0x9 so -7 ⟶ 11001 ⟶ 0x19 . Also: The first bit is negative (here the 5th bit) so if it's 1 this indicates *negative* 16 (plus whatever the other bits indicate).

3. **[16 Points] One of the most disgusting scenes in movie history is the dinner of Mr. Creosote, a grossly crass diner appearing in the Monty Python Movie, *"The Meaning of Life"*. Famously, after eating four dinners with a case of wine and a six crates of beer, Mr. Creosote is offered a "wafer thin mint" that leads to disastrous consequences.**

   In all the following questions, assume you have a 12-bit IEEE-compatible floating point format with a sign-bit, five exponent bits, six mantissa bits and an exponent bias of 15 (*i.e.* the normal IEEE bias, specified as $2^{5-1}-1$ for a 5-bit exponent)).

   Answer the following questions:

   (a) **[4 Points] Assume the mass M_C of Mr. Creosote is 236kg. Represent this as a binary floating point number.**

//How do you find the exponent value to be 0111 (7) ? thanks anonymous chameleon! <3 np :)

1. write the number 236 in binary: 11101100
2. add a floating point and count the number of decimals: 1.1101100 (7 places)
3. gives exponent of 7
4. you don't need the leading 1 in the floating point representation (it's assumed)
5. the exponent needs to be 22 = 10110 since with bias: 22-15 = 7

1. 110110______ x 2**0111______ //. why is there a leading 1?
//Should the signed bit be a 0? I thought having an S bit of 1 implied that the number was negative

In that example we have a leading 1 because it is normalized. You only have a leading zero when it is denormalized. Denormalized is when you have all zero in your exponent field of floating number. Notice the exponent field is 10110 in IEEE representation. There is a 1 in there, so there should be a leading 1 in the binary floating point example.
-thx anonymous hippo ;D

   (b) **[4 Points] Represent the mass of Mr. Creosote in kg using an IEEE 12-bit floating point number. Indicate the portions that are sign, exponent and mantissa by drawing lines between the labeled parts.**

*can someone explain how this works?

S*(2^e)*M (e = E -bias) (if S = 0 multiply by 1/ if S= 1 multiply by -1)

S = sign bit; E = exponent bits; M = mantissa bits

0	1	0	1	1	0	1	1	0	1	1	0
S	E	E	E	E	E	M	M	M	M	M	M

(c) **[4 Points]**

Determine the largest mass of the "wafer thin mint" M_{wtm} such that, when added to the mass M_C of Mr. Creosote, it does *not* change his mass when stored in the 12-bit IEEE format. In other words, determine the largest

M_{wtm} such that $M_{wtm} + M_C = M_C$. You should assume the standard "round to even" rounding mode. It may be useful to determine what values can be represented *precisely* around the value of M_C.

1 . _________0______ x 2**__-7_____

since we only have 6 mantissa bits, and the 6th bit is 0, putting a 1 at the 7th bit rounds to 0 (even)

(if we add 1 to 11101100 the result will be 11101101 and then if we shift it we will get 1.1101101*2^7 and since we have only 6 M places, the result will be 1.110110*2^7, so I think the largest number is 1)(please correct me if I'm wrong)

That might not round right. I'm not exactly sure, but that 1 might be carried over to the next bit with rounding resulting in 1.110111*2^7. Also, the representation would not be what is currently below. There would only end up being an exponent because the mantissa only has a single 1 somewhere in it. So there's an implied 1, and the stored mantissa is all zeroes, the sign bit is zero, and the exponent bits are whatever exponent we have plus 15 represented in binary. (The answer I got was 0|01110|000000)

(d) **[4 Points] What is the smallest possible non-zero positive mass that can be represented in this IEEE representation. Write down that mass below. = 2^(-21)**

0	0	0	0	0	0	0	0	0	0	0	1

4. **[15 Points] Consider the following assembly code for a C for loop:**

//is 0x8(%ebp) the first or second argument? 1st. it goes old ebp, return, first arg (each are 4 bytes)

```
loop:
        pushl %ebp
        movl %esp,%ebp
        movl 0x8(%ebp),%edx
        movl %edx,%eax
        addl 0xc(%ebp),%eax
        leal 0xffffffff(%eax),%ecx
        cmpl %ecx,%edx
        jae .L4
```

%edx initially holds start of string, %eax is given the end of the string (the terminator character) then is decremented so that it points to the actual last character. In "abcde" it points at e, Then does a comparison. If len is 1 or less, it jumps to the end of function

```
.L6:
        movb (%edx),%al
        xorb (%ecx),%al
        movb %al,(%edx)
        xorb (%ecx),%al
        movb %al,(%ecx)
        xorb %al,(%edx)
        incl %edx
        decl %ecx
        cmpl %ecx,%edx
        jb .L6
```

L6 swaps characters via double xor, compares addresses to make sure it still has things to swap

```
.L4:
        movl %ebp,%esp
        popl %ebp
        ret
```

Based on the assembly code above, fill in the blanks below in its corresponding C source code. (Note: you may only use the symbolic variables h, t and len in your expressions below — *do not use register names.*)

```
void loop(char *h, int len)
{ char *t;

    for (t = h+len-1; t > h; h++,t--) {   /* someone edit this if my t definition and t and h comparison is wrong */

        *h = *h^*t;  /*first time through for “abcde”, *h becomes a^e */

        *t = *t^*h; /* *t becomes (a^e)^e = a

        *h = *t^*h; /* becomes
    (a^e)^a = e */ }

    return;

    **XOR swapping algorithm

}
```

5. [18 Points] Consider the following C declaration:

```
    struct Node{
            char c;
            double value;
            struct Node* next;
            int flag;
            struct Node* left;
            struct Node* right;
    };

    typedef struct Node* pNode;

    /* NodeTree is an array of N pointers to Node structs */ pNode NodeTree[N];
```

A. [6 Points] Using the template below (allowing a maximum of 32 bytes), indicate the allocation of data for a Node struct. Mark off and label the areas for each individual element (there are 6 of them). Cross hatch the parts that are allocated, but not used (to satisfy alignment).

Assume the Linux alignment rules. Clearly indicate the right hand boundary of the data structure with a vertical line. //The address at which an item is stored must be a multiple of K, where K is the number of bytes required to store that item. (i.e. should not store an int (4 bytes) at address 3.)

X = empty, not used to satisfy alignment

c = char, v = value, n = next, f = flag, l = left, r = right

CHECK IF THIS IS RIGHT:

C	X	X	X	v	v	v	v	v	v	v	v	n	n	n	n	f	f	f	f
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

l	l	l	l	r	r	r	r	X	X	X	X
20	21	22	23	24	25	26	27	28	29	30	31

```
  0    1    2    3    4    5    6    7    8                9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
                                                                                                                              |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
```

Labeling the start of each element:

| c : 0 | value : 4 | next : 12 | flag : 16 | left : 20 | right : 24 |

B.[12 Points] For each of the four C references below, please indicate which assembly code section (labeled A – F) places the value of that C reference into register %eax. If no match is found, please write "NONE" next to the C reference.
The initial register-to-variable mapping for each assembly code section is:

%eax = starting address of the NodeTree array
%edx = i

References:

1. **pNode[i]->flag** D

2. **pNode[i]->left->left->c** N/A

3. **pNode[i]->next->next->flag** F

4. **pNode[i]->right->left->left** B

Linux/IA32 Assembly:

A.
```
sall $2, %edx
leal (%eax,%edx),%eax

movl 16(%eax),%eax
```

B.
```
sall $2,%edx   # multiply i by 4
leal (%eax,%edx),%eax
# add to start of array
movl (%eax),%eax
# deref : pNode[i]
movl 24(%eax),%eax
#pNode[i]->right
movl 20(%eax),%eax   #-> left
movl 20(%eax),%eax  #->left
```

C:
```
sall $2,%edx
leal (%eax,%edx),%eax

movl 20(%eax),%eax

movl 20(%eax),%eax
movsbl (%eax),%eax
```

D:
```
sall $2,%edx
leal (%eax,%edx),%eax

movl (%eax),%eax

movl 16(%eax),%eax
```

E:
```
sall $2, %edx
leal (%eax,%edx),%eax

movl (%eax),%eax

movl 16(%eax),%eax

movl 16(%eax),%eax

movl 20(%eax),%eax
```

F:
```
sall $2, %edx
leal (%eax,%edx),%eax

movl (%eax),%eax

movl 12(%eax),%eax

movl 12(%eax),%eax

movl 16(%eax),%eax
```

6. [18 Points] The next problem concerns the following C code:

```
/* copy string x to buf */ void foo(char
*x) {
    int buf[1]; strcpy((char *)buf, x);
}
```

```
void callfoo() {
    foo("abcdefghi");
}
```

Here is the corresponding machine code on a Linux/x86 machine:

```
080484f4 <foo>:
080484f4: 55                    pushl    %ebp
080484f5: 89 e5                 movl     %esp,%ebp
080484f7: 83 ec 18              subl     $0x18,%esp
080484fa: 8b 45 08              movl     0x8(%ebp),%eax
080484fd: 83 c4 f8              addl     $0xfffffff8,%esp
08048500: 50                    pushl    %eax
08048501: 8d 45 fc              leal     0xfffffffc(%ebp),%eax
08048504: 50                    pushl    %eax
08048505: e8 ba fe ff ff          call   80483c4 <strcpy>
0804850a: 89 ec                    movl  %ebp,%esp
0804850c: 5d                       popl  %ebp
 0804850d: c3                   ret

08048510 <callfoo>:
  08048510: 55                     pushl %ebp
  08048511: 89 e5                movl     %esp,%ebp
  08048513: 83 ec 08                subl  $0x8,%esp
  08048516: 83 c4 f4                addl     $0xfffffff4,%esp
  08048519: 68 9c 85 04 08          pushl    $0x804859c          # push string address
  0804851e: e8 d1 ff ff ff         call   80484f4 <foo>
  08048523: 89 ec                   movl %ebp,%esp
  08048525: 5d                       popl %ebp
  08048526: c3                       ret
```

This problem tests your understanding of the stack discipline and byte ordering. Here are some notes to help you work the problem:

- **strcpy(char *dst, char *src) copies the string at address src (including the terminating ’\0’ character) to address dst. It does not check the size of the destination buffer.**
- **Recall that Linux/x86 machines are Little Endian.**

- **You will need to know the hex values of the following characters:**

Character	**Hex value**	**Character**	**Hex value**
'a'	0x61	'f'	0x66
'b'	0x62	'g'	0x67
'c'	0x63	'h'	0x68
'd'	0x64	'i'	0x69
'e'	0x65	'\0'	0x00

Now consider what happens on a Linux/x86 machine when callfoo calls foo with the input string "abcdefghi".

(a) List the contents of the following memory locations immediately after strcpy returns to foo. Each answer should be an unsigned 4-byte integer expressed as 8 hex digits.

buf[0] = 0x64636261 //abcd in little endian

buf[1] = 0x68676665 //efgh in little endian

buf[2] = 0x08040069 // where do you get that 0804? It is part of the old ebp
// or should 0804 be 0d85 instead because its little endian?
I am confused about this as well. . It seems that the buffer is 12 bytes, because of these lines:
subl $0x8,%esp
addl $0xfffffff4,%esp
(0x8+0xfffffff4 = 12)
in that case, "abcdefghi" would not overflow the buffer… halp plz
I think the buffer starts 4 bytes
esp, but not sure ⟵ that sound

(b) Immediately before the ret instruction at address 0x0804850d executes, what is the value of the frame pointer register %ebp?

ebp = 0x68676665 (the old ebp is replaced and popped)

(c) Immediately after the ret instruction at address 0x0804850d executes, what is the value of the program counter register %eip?

eip = 0x08040069 (the return address is replaced and popped)