

CSE 351 Midterm - Winter 2017

February 08, 2017

Please read through the entire examination first, and make sure you **write your name and NetID on all pages!** We designed this exam so that it can be completed in 50 minutes.

There are 5 problems for a total of 100 points. There is one extra credit problem worth 10 extra points if you have time and feel adventurous :). The point value of each problem is indicated in the table below. Write your answer neatly in the spaces provided. If you need more space, you can write on the back of the sheet where the question is posed, but please make sure that you indicate clearly the problem to which the comments apply. If you have difficulty with part of a problem, move on to the next one. They are independent of each other.

The exam is CLOSED book and CLOSED notes (no summary sheets, no calculators, no mobile phones, no laptops). Please do not ask or provide anything to anyone else in the class during the exam. Make sure to ask clarification questions early so that both you and the others may benefit as much as possible from the answers.

Good Luck!

Name: _____

Student ID: _____

Section: _____

Problem	Max Score	Score
1. Number Representation	20	
2. Addresses	10	
3. Assembly and C	30	
4. Pointers and Values	20	
5. Procedures	20	
TOTAL	100	
<i>Extra Credit</i>	10	

1. Number Representation (20 points)

Integers

- (a) Assuming unsigned integers, what is the result when you compute $UMAX+1$?

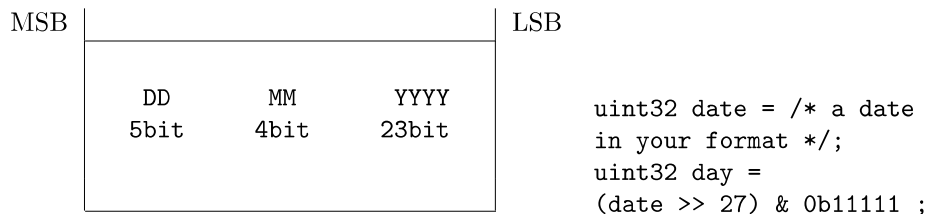
UMIN

- (b) Assuming two's complement signed representation, what is the result when you compute $TMAX+1$?

TMIN (or $(-TMAX - 1)$)

- (c) How would you encode a date with the format $\langle \text{day}=\text{DD} \rangle \langle \text{month}=\text{MM} \rangle \langle \text{year}=\text{YYYY} \rangle$ in a 32 bit word in a way that is easy to extract the day, month and year with masks? Write a C expression that extracts the day only.

(multiple answers accepted)



Floating Point

- (d) Floating point is an approximation of real numbers. What are the components of a floating point number? And what do you get when you add a very large floating point number with a very small floating point number? Why? Please be concise :) Floating point numbers have sign, mantissa, and exponent components.

Ans: When you add a very large floating-point number with a very small one, the resulting answer will be the same as the very large number. This is because of the limited number of mantissa bits – after normalization the bits representing the very small number will be truncated away.

Casting and Pointers

- (e) Given the following code:

```
float f = 5.0;
int i = (int) f;
int j = *((int *)&f);
```

Does $i==j$ return true or false? Explain concisely.

Ans: $i==j$ will return false. i holds the estimate of 5.0 as an integer, whereas j holds the bit-pattern representation of 5.0 in floating-point, which is not the

Name:

NetID:

2. Addresses (10 points)

The table below represents a chunk of memory on a **little-endian** machine with 64-bit words. For example, address `0x27` contains byte `0xAB`.

Word Address	+0	+1	+2	+3	+4	+5	+6	+7
<code>0x0000000000000000</code>	77	66	55	44	33	22	11	00
<code>0x0000000000000008</code>								
<code>0x0000000000000010</code>								
<code>0x0000000000000018</code>	03	00	00	00	00	00	00	00
<code>0x0000000000000020</code>								0xAB

(a) Write word `0x0011223344556677` at address `0x00`.

(b) In location `0x18`, write a pointer that points to a location that stores value `0x44`.

3. Assembly and C (30 points)

Consider the following x86-64 assembly, (partially blank) C code, and memory listing. Addresses and values are 64-bit.

foo:	movl \$0, %eax	int foo(long *p) {	int result = 0;	Address	Value
		while (p != NULL) {	while (p != NULL) {	0x1000	0x1030
L1:	testq %rdi, %rdi	// cast p, then deref	p = *(long**)p;	0x1008	0x1020
	je L2	result = result + 1;	result = result + 1;	0x1010	0x1000
	movq (%rdi), %rdi	}	}	0x1018	0x0000
	addl \$1, %eax	return result;	return result;	0x1020	0x1030
	jmp L1	}	}	0x1028	0x1008
				0x1030	0x0000
				0x1038	0x1038
L2:	ret			0x1040	0x1048
				0x1048	0x1040

(a) Given the assembly of `foo`, fill in the blanks of the C version.

(b) Trace the execution of the call to `foo((long*)0x1000)` in the table to the right. Show which instruction is executed in each step until `foo` returns. In each space, place the **the assembly instruction** and the values of the appropriate registers **after that instruction executes**. You may leave those spots blank when the value does not change. You might not need all steps listed on the table.

Instruction	%rdi (hex)	%eax (decimal)
movl	0x1000	0
testq		
je		
movq	0x1030	
addl		1
jmp		
testq		
je		
movq	0x0	
addl		2
jmp		
testq		
je		
ret		

(c) Briefly describe the value that `foo` returns and how it is computed. Use only variable names from the C version in your answer.

It returns the depth of the pointer chain from `p` by counting how many times it can be dereferenced before it's `NULL`.

4. Pointers and Values (20 points)

Consider the following variable declarations:

```
int x;
int y[11] = {0,1,2,3,4,5,6,7,8,9,10};
int z[][5] = {{210, 211, 212, 213, 214}, {310, 311, 312, 313,314}};
int aa[3] = {410, 411, 412};
int bb[3] = {510, 511, 512};
int cc[3] = {610, 611, 612};
int *w = {aa, bb, cc};
```

Variable	Address
aa	0x000
bb	0x100
cc	0x200
w	0x300
x	0x400
y	0x500
z	0x600

- (a) Fill in the table below with the address, value, and type of the given C expressions. Answer N/A if it is not possible to determine the address or value of the expression. The first row has been filled in for you.

C Expression	Address	Value	Type (int/int*/int**)
x	0x400	N/A	int
&x	0x400	N/A	int
y[0]	0x500	0	int
*(y+1)	0x504	1	int
*(z[0]+1)	0x604	211	int
w[1]	0x308	0x100	int*

5. Procedures (20 points)

Below is a simple program that calls a function `bigbig` with several arguments. A portion of the result of `objdump` is shown beside it.

```
int main() {
    return bigbig(1, 2, 3, 4, 5, 6, 7);
}

main:
100000f20:
100000f25:
100000f2a:
100000f2f: ... omitted ...
100000f34:
100000f3a:
100000f40:
100000f42: callq 0x100000f00 <bigbig>
100000f47: ... omitted ...
100000f48: retq
```

Suppose you ran `gdb` on this program, running the command `break bigbig`. Draw the state of the stack and registers below at the time of the breakpoint (right **before** the instruction at `0x100000f00` executes).

Assume that, before doing any setup for the call to `bigbig`, `%rsp = 0x1ffefff8` and all other relevant registers are set to 0. If a value on the stack in the given table is still unknown after execution, you may leave that box blank.

Stack Address	Value
0x1ffefff8	
0x1ffefff0	0x7
0x1ffeffe8	0x100000f47
0x1ffeffe0	
0x1ffeffd8	

Reg	Value	Reg	Value
%rax		%r8	5
%rbx		%r9	6
%rcx	4	%r10	
%rdx	3	%r11	
%rsi	2	%r12	
%rdi	1	%r13	
%rsp	0x1ffeffe8	%r14	
%rbp		%r15	

Name:

NetID:

Extra Credit (10 points)

What does the code below compute?

```
int mystery(unsigned x) {
    x = (x & 0x55555555) + ((x >> 1) & 0x55555555);
    x = (x & 0x33333333) + ((x >> 2) & 0x33333333);
    x = (x & 0x0F0F0F0F) + ((x >> 4) & 0x0F0F0F0F);
    x = (x & 0x00FF00FF) + ((x >> 8) & 0x00FF00FF);
    x = (x & 0x0000FFFF) + ((x >>16) & 0x0000FFFF);
    return x;
}
```

It's POPCOUNT, which counts the number of set bits in x .

References

Powers of 2:

$$\begin{array}{ll}
 2^0 = 1 & \\
 2^1 = 2 & 2^{-1} = 0.5 \\
 2^2 = 4 & 2^{-2} = 0.25 \\
 2^3 = 8 & 2^{-3} = 0.125 \\
 2^4 = 16 & 2^{-4} = 0.0625 \\
 2^5 = 32 & 2^{-5} = 0.03125 \\
 2^6 = 64 & 2^{-6} = 0.015625 \\
 2^7 = 128 & 2^{-7} = 0.0078125 \\
 2^8 = 256 & 2^{-8} = 0.00390625 \\
 2^9 = 512 & 2^{-9} = 0.001953125 \\
 2^{10} = 1024 & 2^{-10} = 0.0009765625
 \end{array}$$

Hex/decimal/binary help:

Decimal	Hexadecimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

Name:

NetID:

Assembly Code Instructions:

push	push a value onto the stack and decrement the stack pointer
pop	pop a value from the stack and increment the stack pointer
call	jump to a procedure after first pushing a return address onto the stack
ret	pop return address from stack and jump there
mov	move a value between registers and memory
lea D(base, index, scale), dest	compute effective address (does not load) and place in register dest. (dest = D + (base + (index * scale)) when scale is 1,2,4,8)
add	add src (1 st operand) to dst (2 nd) with result stored in dst (2 nd)
sub	subtract src (1 st operand) from dst (2 nd) with result stored in dst (2 nd)
and	bit-wise AND of src and dst with result stored in dst
or	bit-wise OR of src and dst with result stored in dst
sar	shift data in the dst to the right (arithmetic shift) by the number of bits specified in 1 st operand
jmp	jump to address
je/jne	conditional jump to address if zero flag is / is not set
js/jns	conditional jump to address if sign flag is / is not set
cmp	subtract src (1 st operand) from dst (2 nd) and set flags, discard result
test	bit-wise AND src and dst and set flags, discard result

Register map for x86-64:

Note: all registers are caller-saved except those explicitly marked as callee-saved, namely, `rbx`, `rbp`, `r12`, `r13`, `r14`, and `r15`. `rsp` is a special register.

<code>%rax</code>	Return Value	<code>%r8</code>	Argument #5
<code>%rbx</code>	Callee Saved	<code>%r9</code>	Argument #6
<code>%rcx</code>	Argument #4	<code>%r10</code>	Caller Saved
<code>%rdx</code>	Argument #3	<code>%r11</code>	Caller Saved
<code>%rsi</code>	Argument #2	<code>%r12</code>	Callee Saved
<code>%rdi</code>	Argument #1	<code>%r13</code>	Callee Saved
<code>%rsp</code>	Stack Pointer	<code>%r14</code>	Callee Saved
<code>%rbp</code>	Callee Saved	<code>%r15</code>	Callee Saved