

Computer Science 2021
Spring 2016
Midterm Exam 1
February 29th, 2016
Time Limit: 50 minutes, 3:35pm-4:25pm

- This exam contains 7 pages (including this cover page) and 5 questions. Once we tell you to start, please check that no pages are missing.
- Before starting the exam, you can fill out your name and other information of this page, but don't open the exam until you are directed to start. Don't put any of your answers on this page.
- You may use any textbooks, notes, or printouts you wish during the exam, but you may not use any electronic devices: no calculators, smart phones, laptops, etc.
- You may ask clarifying questions of the instructor or TAs, but no communication with other students is allowed during the exam.
- Please read all questions carefully before answering them. Remember that we can only grade what you write on the exam, so it's in your interest to show your work and explain your thinking.
- Students often find that the quiz questions vary in difficulty. Your best strategy is usually to skim over all the questions, and then start working on the ones that look easiest. We also suggest that you leave time at the end to attempt every question, since we can't give you any partial credit if you leave a question blank.
- By signing below you certify that you agree to follow the rules of the exam, not to share exam material with other students before their exams, and that the answers on this exam are your own work only.

The exam will end promptly at 4:25pm. Good luck!

Your name (print): _____

Your UMN email/X.500: _____@umn.edu

Row letter: _____ Seat number: _____

Sign and date: _____

Question	Points	Score
1	20	
2	15	
3	28	
4	21	
5	16	
Total:	100	

1. (20 points) Binary integers.

Imagine that our system uses a **5-bit** integer representation, and does addition and subtraction using the rules for 5-bit, two's complement arithmetic. For each of the expressions in the left column, fill in the following columns with the result, expressed in either decimal or in binary. When converting to decimal, you should treat the value as either signed or unsigned according to the rules of C, where TMin and TMax are signed, as are plain constants, but a constant ending in U is unsigned.

Expression	Decimal Representation	Binary Representation
$-T_{\text{Max}} - 1U$		
$-T_{\text{Min}}$		
$-T_{\text{Max}} - T_{\text{Min}}$		
$T_{\text{Max}} + T_{\text{Max}}$		
$T_{\text{Min}} - 1U$		

2. (15 points) Multi-dimensional arrays.

In the source code below, M, N, and L are constant integers declared with `#define` and used to set the dimensions of two three-dimensional arrays. The function `copyandsub` reads from one array and writes to the other. Based on the assembly language code for `copyandsub` shown below, fill in the numeric values of M, N, and L. Remember that arrays in C are stored in *row-major order*.

```
#define M _____

#define N _____

#define L _____

int array1[M][N][L];
int array2[L][N][M];
void copyandsub(int i, int j, int k)
{
    array1[i][j][k] = array2[k][j][i] - M;
}
```

Suppose the above code generates the following assembly code:

```
copyandsub:
    leaq    (%rdx,%rdx,4), %rax
    leaq    (%rax,%rax,4), %rcx
    salq    $3, %rcx
    leaq    (%rsi,%rsi,4), %rax
    addq    %rax, %rax
    addq    %rcx, %rax
    addq    %rdi, %rax
    movl    array2(,%rax,4), %eax
    subl    $10, %eax
    imulq   $600, %rdi, %rcx
    leaq    (%rsi,%rsi), %r8
    salq    $5, %rsi
    subq    %r8, %rsi
    leaq    (%rcx,%rsi), %rdi
    addq    %rdi, %rdx
    movl    %eax, array1(,%rdx,4)
    ret
```

3. (28 points) Assembly language.

On the left is assembly code for a function with a loop and a jump table. On the right is an incomplete skeleton for corresponding C code. Fill in the blanks in the C code so that it has the same behavior. You don't need to declare any new variables, just use the ones we declared. (This code does do something useful, though we have made it more complicated than necessary.)

<pre> func1: movq %rdi, %rax jmp .L2 .L3: shrq %rdi addq \$1, %rax shrq %rax .L2: cmpq \$2, %rdi ja .L3 movq %rax, %rdx subq %rdi, %rdx cmpq \$5, %rdx ja .L4 jmp *.L6(,%rdx,8) .L6: .quad .L10 .quad .L7 .quad .L8 .quad .L5 .quad .L8 .quad .L5 .L4: movl \$5, %eax ret .L7: movl \$7, %eax ret .L8: movl %rdi, %rax ret .L10: movl \$12, %eax .L5: ret </pre>	<pre> int func1(unsigned long x) { unsigned long f, c; int result; f = _____; c = _____; while (_____ >= _____) { f = _____; c = _____; } switch (_____) { default: result = _____; break; case _____: result = _____; break; case _____: result = _____; break; case _____: case _____: result = _____; break; case _____: case _____: result = _____; break; } return result; } </pre>
--	--

4. (21 points) Floating point.

Consider the following two 9-bit floating point representations based on the IEEE floating-point format with a sign bit:

1. Format A has 1 sign bit, 4 exponent bits and 4 fraction bits
2. Format B has 1 sign bit, 5 exponent bits and 3 fraction bits

Below are some bit patterns in Format A. Your job is to convert each to the closest value in Format B. If necessary, you should apply the *round-to-nearest, ties-to-even* rounding rule. In addition, give the values of the numbers represented by the patterns. Give these as whole numbers (e.g. 17), fractions (e.g. 11/16), or special values (NaN, $\pm\infty$).

Format A		Format B	
Bit Pattern	Value	Bit Pattern	Value
1 1000 1011			
0 0000 0100			
1 1111 0000			

5. (16 points) Sweet sixteen.

The following weird union type contains five different structs within it. The `n` field of `s1` (i.e., `the_u.f1.n`) is at offset 16 from the start of the union. For each of the other four structures, fill in the blank with which part in that structure starts at the same offset 16. Each answer should be either a field name like `f`, or an element within an array field, like `a[0]`. (You may find it helpful to figure out the offsets of each field within the structures, or to draw pictures of their layouts, so we've left some extra space after each.)

```
union u {

    /* At offset 16:  n    */
    struct s1 { long l; long m; long n; } f1;

    /* At offset 16: _____ */
    struct s2 { int ia[4]; long la[2]; } f2;

    /* At offset 16: _____ */
    struct s3 { short s[10]; double d; } f3;

    /* At offset 16: _____ */
    struct s4 { char *cp; short *sp; int *ip; long *lp; } f4;

    /* At offset 16: _____ */
    struct s5 { short s; int i; short t; int j; short u; int k; } f5;

} the_u;
```

This extra page has some tables of useful information for your reference.
x86-64 assembly language (AT&T format):

Instructions	
add X, Y	add $X + Y$ and store in Y , set flags based on result
cmp X, Y	compute $Y - X$, and set flags only based on result
lea X, Y	compute address of X and store in Y
imul X, Y, Z	compute $X \times Y$ and store in Z
ja L	jump to L if unsigned greater than
jmp L	jump to L , always
jmp $*X$	jump to address X
mov X, Y	copy value X to location Y
sal X, Y	shift Y left by X positions
shr Y	logical shift Y right by one position
shr X, Y	logical shift Y right by X positions
sub X, Y	compute $Y - X$ and store in Y , set flags based on result
Addressing modes	
(R)	mem[reg[R]]
D(R)	mem[D + reg[R]]
D(R,S)	mem[D + reg[R] * S]
Size suffixes	
b	8-bit byte
w	16-bit value
l	32-bit value
q	64-bit value
Calling conventions	
Argument registers	%rdi, %rsi, %rdx, %rcx, %r8, %r9
Return value	%rax

Sizes of basic C types on x86-64:

Type	Size (bytes)	Alignment
char	1	1
short	2	2
int	4	4
long	8	8
float	4	4
double	8	8
pointer	8	8