Computer Science 2021 Section 010
Fall 2018
Midterm Exam 2
November 16th, 2018
Time Limit: 50 minutes, 3:35pm-4:25pm

- This exam contains 8 pages (including this cover page) and 4 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 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

Sign and date: ______________________________

Question	Points	Score
1	24	
2	31	
3	25	
4	20	
Total:	100	

1. (24 points) Unexpected instructions.

   In the left column are 12 x86-64 instructions. In the right column are 12 C statements containing `long` variables named after registers. Match each instruction with the C code that describes its effect, assuming that each variable is held in the register with the same name. But don't expect to be able to tell just by comparing instruction names with the C operators.

   Each choice on the right should be used the same number of times it appears: i.e., choices A, G, and I are used twice each, and the others once each.

(a) ____ `lea (%rax, %rax, 8), %rdx`

(b) ____ `and $0, %rdx`

(c) ____ `lea 0(%rdx), %rdx`

(d) ____ `lea (%rax, %rdx), %rdx`

(e) ____ `mov %rdx, %rdx`

(f) ____ `sar $63, %rdx`

(g) ____ `shl $2, %rdx`

(h) ____ `add %rdx, %rdx`

(i) ____ `lea (,%rdx, 4), %rdx`

(j) ____ `xor %rdx, %rdx`

(k) ____ `lea -1(%rax, %rax), %rdx`

(l) ____ `lea (%rax, %rax, 2), %rdx`

A. `rdx = 0`

A. `rdx = 0`

B. `rdx += rax`

C. `rdx = 2*rax - 1`

D. `rdx = 3*rax`

E. `rdx = 9*rax`

F. `rdx = rdx << 1`

G. `rdx = 4*rdx`

G. `rdx = 4*rdx`

H. `rdx = rdx < 0 ? -1 : 0`

I. `/* does nothing */`

I. `/* does nothing */`

2. (31 points) Machine code and arrays.

   Below is the assembly code for a function named `check`. On the next page, along with another copy of the assembly, is the outline of C code for the function, with many parts left blank. Fill in the blanks in the C code to create a function that does the same thing as the assembly code, in other words what might have been the source code compiled to create this assembly code. Use the macro `SIZE` in code related to the size of the arrays, so that the source code could also be used if the array size changed.

   Note that in several places the structure of the assembly code does not directly match the source code: for instance the source code has one parameter and four local variables, whereas the assembly code uses a different number of registers. There are multiple possible correct answers, based on different ways of writing the same functionality.

```
check:
    pushq   %rbx
    movl    $100, %edx
    movq    %rdi, %rbx
    xorl    %esi, %esi
    movq    $flags, %rdi
    call    memset
    xorl    %eax, %eax
.L4:
    movq    (%rbx,%rax,8), %rdx
    cmpq    $99, %rdx
    jbe     .L2
.L6:
    xorl    %eax, %eax
    jmp     .L3
.L2:
    incq    %rax
    movb    $1, flags(%rdx)
    cmpq    $100, %rax
    jne     .L4
    movl    $100, %edx
    xorl    %eax, %eax
.L5:
    orb     $0x80, flags(%rax)
    decq    %rdx
    movq    (%rbx,%rax,8), %rax
    jnz     .L5
    xorl    %eax, %eax
.L7:
    cmpb    $0x81, flags(%rax)
    jne     .L6
    incq    %rax
    cmpq    $100, %rax
    jne     .L7
    movl    $1, %eax
.L3:
    popq    %rbx
    ret
```

This code does something meaningful, but you don't need to understand the meaning to complete the question.
The `memset` function is declared as:
`void *memset(void *s, int c, size_t n);`
It fills in an `n`-byte memory area pointed to by `s` with the byte value `c`, and returns the pointer `s`.
The instruction `incq` adds 1 to its operand, while the instruction `decq` subtracts 1. Both instructions set the condition code flags based on the result.

```
check:
    pushq   %rbx
    movl    $100, %edx
    movq    %rdi, %rbx
    xorl    %esi, %esi
    movq    $flags, %rdi
    call    memset
    xorl    %eax, %eax
.L4:
    movq    (%rbx,%rax,8), %rdx
    cmpq    $99, %rdx
    jbe     .L2
.L6:
    xorl    %eax, %eax
    jmp     .L3
.L2:
    incq    %rax
    movb    $1, flags(%rdx)
    cmpq    $100, %rax
    jne     .L4
    movl    $100, %edx
    xorl    %eax, %eax
.L5:
    orb     $0x80, flags(%rax)
    decq    %rdx
    movq    (%rbx,%rax,8), %rax
    jnz     .L5
    xorl    %eax, %eax
.L7:
    cmpb    $0x81, flags(%rax)
    jne     .L6
    incq    %rax
    cmpq    $100, %rax
    jne     .L7
    movl    $1, %eax
.L3:
    popq    %rbx
    ret
```

```
#define SIZE 100
unsigned char flags[SIZE];
long check(long a[SIZE]) {
  long i1, i2, i3, j;

  memset(_______, ________, _______);

  for (i1 = 0; ______ SIZE; i1++) {

    if (_____ < 0 || ______________)
      return 0;

      ______________ = 1;
  }
  j = 0;

  for (________; _________; ________) {

    ______________________;

    j = ________;
  }

  for (i3 = 0; ________; _________) {

    if (_____________________)

      return _______;
  }

  return _______;
}
```

3. (25 points) Stack layout.

For this question, consider the following C code, with some of the corresponding assembly code to the right:

```
long f1(long x) {
    long l1 = 1;
    long l2 = 2;
    long l3 = x;
    return l1 + l2 + l3;
}

void f2(long x) {
    char local[8];
    if (x & 1) {
        local[0] = 0;
    }
    printf("You won ");
    if (local[0])
        printf("%.8s\n", local);
    else
        printf("nothing.\n");
}

int main(void) {
    long input;
    /* ... */
    f1(input);
    f2(input);
    /* ... */
}
```

```
f1:
    subq    $0x28, %rsp
    movq    %rdi, 0x10(%rsp)
    movq    $1, 0x20(%rsp)
    movq    $2, 0x18(%rsp)
    movq    0x10(%rsp), %rax
            # here (a)
    addq    $3, %rax
    addq    $0x28, %rsp
    ret

f2:
    subq    $0x28, %rsp
    movq    %rdi, %rax
    andl    $1, %eax
    testq   %rax, %rax
    je      .L4
    movb    $0, 0x10(%rsp)
            # here (b)
    .L4:    # ...
    ret

main:
    # ... input in %rbx
    # here %rsp is 0x7fffffff0030
    mov     %rbx, %rdi
    call    f1
    mov     %rbx, %rdi
    call    f2
```

Comments containing `...` indicate places where we have left out code to simplify what you have to look at.

Assume that before the call to `f1`, the value of the stack pointer `%rsp` is `0x7fffffff0030`. The questions on the next page will ask you about the contents of the stack during the execution of `f1` and `f2`. For those questions:

- Fill in a box with an exact numeric value if you know it, in either decimal or hexadecimal with `0x`.
- If you know some but not all digits of a number, you can write question marks in place of the unknown digits.
- Write "Return address for `function`" for the return address of a call to a function `function`.
- If there is no way to know the contents of a location from the information we have provided, write "unknown" or leave the box blank.

The `printf` format specifier `%.8s` is like a regular "`%s`", except that it will never print more than 8 characters of the string.

(a) Suppose that the value of the variable `input` is 7. Fill in the blanks in the following table to show the contents for the stack at the point labeled "`here (a)`" inside `f1`:

Address	Contents
`0x7fffffff0030`	Return address for `main`
`0x7fffffff0028`	
`0x7fffffff0020`	
`0x7fffffff0018`	
`0x7fffffff0010`	
`0x7fffffff0008`	
`0x7fffffff0000`	

(b) Still for a run with `input` equal to 7, fill in the following table to show the contents of the stack at the point labeled "`here (b)`" inside `f2`:

Address	Contents
`0x7fffffff0030`	Return address for `main`
`0x7fffffff0028`	
`0x7fffffff0020`	
`0x7fffffff0018`	
`0x7fffffff0010`	
`0x7fffffff0008`	
`0x7fffffff0000`	

(c) Suppose that you wanted the program to print the message:

`You won $9999999`

What value would `input` need to have to make that happen? (Hint: the final page of the exam has an ASCII table.) Write the 64-bit integer value in hexadecimal:

4. (20 points) Structure shuffling.

   Each of the parts of this question gives you the names and types of the fields of a C `struct`. Choose an order for the fields, so that layout of the structure on x86-64 has the requested property. Give the order by listing the names of the fields in your order; we have shown one answer as an example. Optionally (e.g., for partial credit), we have also left space where you could draw the structure layout.

   Example fields: `short s; char c; float f;`
   Example property: fields are in order of increasing size
   Order: `c`, `s`, `f`

   (a) Fields: `short s; int i; char c; double d; unsigned char uc;`
   Property: structure requires no padding

   Order:

   (b) Fields: `char a, b, c; int i, j, k;`
   Property: total size (including padding) is 24 bytes

   Order:

   (c) Fields: `float *fp; short s; float f; int i; unsigned short us;`
   Property: `s` starts at offset 10

   Order:

   (d) Fields: `short s; char c; float f; double d; int *ip`
   Property: every field offset divisible by 8

   Order:

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

Addressing modes	
(R)	mem[reg[R]]
D(R)	mem[D + reg[R]]
D(B,I,S)	mem[D + reg[B] + reg[I] * S] omitted D, B, or I treated as 0 omitted S treated as 1
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

ASCII/hex table:

```
0 NUL    10 DLE    20        30 0    40 @    50 P    60 `    70 p
1 SOH    11 DC1    21 !      31 1    41 A    51 Q    61 a    71 q
2 STX    12 DC2    22 "      32 2    42 B    52 R    62 b    72 r
3 ETX    13 DC3    23 #      33 3    43 C    53 S    63 c    73 s
4 EOT    14 DC4    24 $      34 4    44 D    54 T    64 d    74 t
5 ENQ    15 NAK    25 %      35 5    45 E    55 U    65 e    75 u
6 ACK    16 SYN    26 &      36 6    46 F    56 V    66 f    76 v
7 BEL    17 ETB    27 '      37 7    47 G    57 W    67 g    77 w
8 BS     18 CAN    28 (      38 8    48 H    58 X    68 h    78 x
9 HT     19 EM     29 )      39 9    49 I    59 Y    69 i    79 y
A LF     1A SUB    2A *      3A :    4A J    5A Z    6A j    7A z
B VT     1B ESC    2B +      3B ;    4B K    5B [    6B k    7B {
C FF     1C FS     2C ,      3C <    4C L    5C \    6C l    7C |
D CR     1D GS     2D -      3D =    4D M    5D ]    6D m    7D }
E SO     1E RS     2E .      3E >    4E N    5E ^    6E n    7E ~
F SI     1F US     2F /      3F ?    4F O    5F _    6F o    7F DEL
```