UCLA Computer Science 33 (Fall 2016)
Midterm 1, 100 minutes, 100 points, open book, open notes.
Put your answers on the exam, and put your name and student ID at the top of each page.

Name: [redacted] Student ID: [redacted]

1	2 a+b	2 c	3 a	3 b	4	5 a+b	5 c	5 d	total

1111
1110
0001

1 (11 minutes). In a circular shift, bits are not discarded when they fall off the end of a word; instead, they are reintroduced on the other end. Write a 64-bit function 'long rcshi (long a, int b);' for GCC on the x86-64 that returns the result of circularly shifting A right by B bits, where you can assume 0 <= B < 64. For example, rcshi (0xbaddeadbeefadded, 16) == 0xddedbaddeadbeefa.

16 bits -> 4 bits

```
long rcshi (long a, long b) {
    long mask = (a & ~(-1 << b));   // bits have b bits
    long temp = a >> b;             // make room for bits
    long g = mask << (64-b);        // move saved bits into right spot
    return (g | temp);              // replace bits
}
```

Name: Timothy Margono Student ID: 004 800 078 (10)

2. On the x86-64, the 'cqto' instruction sets rdx to zero or to -1 depending on whether rax's sign bit is 0 or 1, respectively, and the 'idivq X' instruction divides the 128-bit signed integer ((2**64)*(long)rdx + ((unsigned long)rax)) by the 64-bit signed integer X and puts the signed quotient into rax and the signed remainder into rdx. 'idivq X' traps if X is zero, or if integer overflow occurs when attempting to fit the quotient into rdx.

With that in mind, consider the following C code 735

```
long aquo (long a, long b) { return a / b; }
long arem (long a, long b) { return a % b; }
long bquo (long a, long b) { return -a / -b; }
long brem (long a, long b) { return -a % -b; }
```

When compiled for x86-64 by 'gcc -O2 -S', the compiler translates this source code into the following four functions, where the order and names of the functions have been changed. (Notice that A and B are identical.)

```
A:
        movq    %rdi, %rax
        cqto                    # extend rdx
        idivq   %rsi
        ret
B:
        movq    %rdi, %rax
        cqto
        idivq   %rsi
        ret
```

```
C:
        movq    %rdi, %rax
        negq    %rsi            # negate
        negq    %rax            # negate
        cqto                    # extend rdx
        idivq   %rsi
        movq    %rdx, %rax
        ret
D:
        movq    %rdi, %rax
        cqto
        idivq   %rsi
        movq    %rdx, %rax
        ret
```

2a (8 minutes): Label each machine-code functions (A, B, C, D) with the C-language function (aquo, arem, bquo, brem) or functions that it corresponds to. 8

2b (5 minutes): Explain why two of the C-language functions generate exactly the same machine code, even though mathematically they are different functions. Why isn't this a bug in the C compiler? 2

-a/-b and a/b will have the same machine code as long as a signed number, and the idivq is signed division.

Name: Timothy Thompson Student ID: 004 800 078

2c (10 minutes): Suppose we also use '-fwrapv', i.e., we compile with 'gcc -O2 -S -fwrapv'. Which part of the machine code for aquo, arem, bquo, and brem would you expect to change, and why? Give an example call showing why the given machine code would be incorrect if '-fwrapv' had generated it.

I would expect to change the two legs of instructions in the machine code for brem. (Think)
If a or b is the min. signed long, then taking the negative of it returns (Think)

7

Name: Timothy Margono Student ID: 004 800 078

3. In this problem, your answers must use only the integer operations ^, &, |, ~, !, ==, !=, <, >, <=, >=, <<, and >> along with any integer constants that you find useful. Your answers should contain only straight-line code, i.e., no conditional expressions (?:), conditional statements, or loops. You may assume that your code is compiled with -fwrapv. Minimize the number of operations that you use in your answers.

3a (10 minutes). Define a "strong integer" to be an integer whose binary representation contains two adjacent 1 bits, and a "weak integer" to be an integer that is not strong. Write a C function 'bool is_strong (long a);' that returns 1 if if A is a strong integer, and 0 otherwise.

0011

```
bool IS_strong (long a){
    int mask1 = 0x33333333;
    int mask2 = 0x66666666;
    int mask3 = 0x99999999;
    int mask4 = 0xBBBBBBBB;
    int mask5 = 0xCCCCCCCC;
    int mask6 = 0xDDDDDDDD;

    long check1 = (a & mask1) | (a & (mask1 << 1));
    long check2 = (a & mask2) | (a & (mask2 << 1));
    long check3 = (a & mask3) | (a & (mask3 << 1));
    long check4 = (a & mask4) | (a & (mask4 << 1));
    long check5 = (a & mask5) | (a & (mask5 << 1));
    long check6 = (a & mask6) | (a & (mask6 << 1));
    long final = (check1 | check2 | check3 | check4 |
                  check5 | check6);

    return (final != 0);
}
```

(2)

Name: Timothy Murgano Student ID: 004 800 078

3b (12 minutes). Write a C function 'long weakadd (long a, long b);' that returns the sum of two weak integers A and B. If the result would not fit in 'long', yield the low-order 64 bits of the correct mathematical answer. Remember, your implementation is limited to the operators mentioned on the previous page; in particular, it cannot use '+' or '-'.

$-a = \sim a + 1$

```
long weakadd (long a, long b) {

return a (-) b;

}
```

(0)

Name: Timothy Morgan Student ID: 004 800 078

4 (12 minutes). The programming language Fortran, introduced in 1957 and still widely used in big scientific applications, has an arithmetic IF that uses a three-way branch, in which the numbers are labels of statements to go to depending on whether the if-expression is negative, zero, or positive. For example, assuming all quantities are 32-bit integers, this Fortran code:

```
      if (M + N) 10, 20, 30
10    I = I + 2
20    J = J + 4
30    K = K - 5
```

acts like this C code, where each "..." stands for 2**31 - 4 case labels:

```
switch (M + N) {
  case -1: case -2: case -3: ...
    I = I + 2;
  case 0:
    J = J + 4;
  case 1: case 2: case 3: ...
    K = K - 5;
}
```

in that it adds 2 to I if M + N < 0, adds 4 to J if M + N <= 0, and always subtracts 5 from K. Show how the above code can be translated to x86-64 machine code that uses only one comparison instruction, as opposed to the two or more comparisons that one might normally expect. Assume that M is in %rdi, %N is in %rsi, that I, J and K are global variables residing in RAM, and that -fwrapv is being used.

```
reg %rsi
cmpl %rsi, %rdi
js .L1
je .L2
jmp .L3

.L1  addl $2, 8(%rsp)   // I = I + 2
.L2  addl $4, 16(%rsp)  // J = J + 4
.L3  subl $5, 24(%rsp)  // K = K - 5
```

12

Name: Timothy Margono Student ID: 004 800 078

5. Consider the following x86-64 C program, which uses a flexible array member:

```
#include <stddef.h>
#include <stdlib.h>
#include <string.h>
struct cs {
  int color;
  long len;
  char data[];
};

/* Set *P to be a newly allocated string that has the old *P's
   color, but has only the bytes in the old *P starting at OFFSET
   and continuing for LEN bytes.  Return the address of the newly
   allocated string's data.  */
char *
substr (struct cs **ptr, long offset, long len) {
  struct cs *old = *ptr;
  int a = alignof (struct cs);
  struct cs *new
    = malloc (offsetof (struct cs, data) + len);
  new->color = old->color;
  new->len = len;
  *ptr = new;
  /* The standard function memcpy (A, B, C)
     copies C bytes from B to A and returns A.  */
  return memcpy (new->data, old->data + offset, len);
}
```

Timothy Margono

6

5a (3 minutes). Give the sizes and offsets of each member of 'struct cs', and why they have the values they do.

color: offset 0, 4 bytes. color is an int.

len: offset 8, 8 bytes. len is a long.

data: offset 16, size unknown. The size of the array is unknown.

2

5b (5 minutes). Explain why the offsetof call is needed, how it works, and what it returns.

The offset call is needed to find the correct amount of memory to assign for the struct. It works by casting the nullptr to type cs* and following it to data member data. The address of it is then taken and casted to (size_t) (like this: (size_t) &((cs*)0)->data) It will return the offset of the data array in memory.

4

Name: Timothy Runyan Student ID: 00 4 800 078

5c (12 minutes). Compiling the 'substr' function on the x86-64 might yield the following code, except that four faults (errors in the machine code) have been deliberately introduced. These faults are labeled A, B, C and D below. Fix each of the faults by correcting the machine code.

p 216

```
substr:
        pushq   %r13
        movq    %rsi, %r13
        pushq   %r12
        movq    %rdi, %r12
        pushq   %rbp
        pushq   %rbx
        movq    %rdx, %rbx
        subq    $8, %rsp
A:      movl    (%rdi), %ebp
        leaq    23(%rdx), %rdi
        andq    $-8, %rdi
B:      jle     malloc
        movl    0(%rbp), %edx
        leaq    16(%rbp,%r13), %rsi
        movq    %rbx, 8(%rax)
        leaq    16(%rax), %rdi
        movl    %edx, (%rax)
        movq    %rbx, %rdx
C:      movl    %eax, (%r12)
        addq    $8, %rsp
        popq    %rbx
        popq    %rbp
D:      popq    %r13
        popq    %r12
        jmp     memcpy
```

ptr → %rdi

offset → %rsi

len → %rdx

movq (%rdi), %rbp +3

Call malloc +3

23+len = rdi

→ 8 rdi

ropq %r12 +3

9

Name: Timothy Rogers Student ID: 004 800078

6

5d (12 minutes). For each fault in (5c), give an example of what can go wrong in a C program if all the other faults are fixed but that fault remains unfixed. Be as precise as you can in your example.

A) Program won't compile as incorrect suffix may be used. pointers are 8 bytes and need the suffix q. Also the register %ebp needs to be changed to its 64-bit version %rbp

It will work but truncate

+1 ✓

B) malloc is a function and it called every time subfor is run. So malloc has a chance to not call malloc and the right amount of storage might never be allocated.

✓

C)

D) The stack follows a Last in, first out structure. The wrong value would be popped and put in register %r13.

+2

for the ✓ caller function

175