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: Xiaohe Yang Student ID: 504040737

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

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.

```
long rcshi( long a, int b) {
    long mask1=~(-1 << b) ;        // 000.. 00 11... 11 (n)
    long mask2= (1 << (64-b)) ;    // 00...0 11.. 11111 (n)
    long shift = mask2 & (a >> b) ; // prevent sign extension
    long toAdd =(mask1 & a) << (64-n) ;
        // get truncated bits, shift to the left most

    return toAdd | shift ;

}
```

Name: Xiaohe Yang Student ID: 504640737 (4)

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.~~ rax

With that in mind, consider the following C code

```
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:   aquo/arem                   C:   brem
     movq    %rdi, %rax               movq    %rdi, %rax
     cqto                             negq    %rsi
     idivq   %rsi                     negq    %rax
     ret                              cqto
B:   aquo/arem                        idivq   %rsi
     movq    %rdi, %rax               movq    %rdx, %rax
     cqto                             ret
     idivq   %rsi                D:   bquo
     ret                              movq    %rdi, %rax
                                      cqto
                                      idivq   %rsi
                                      movq    %rdx, %rax
                                      ret
```

giving marks for 1
it should be aquo / bquo

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. A,B A,B D C (4)

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?

(0) Because it's two's complement.

Those are synonymous, just like jz and je.

Name: Xiaohe Yang Student ID: 504640737

2c (10 minutes): Suppose we also use '-fwrapv', i.e., we compile with 'gcc -O2 -S -fwrapv' (wraps around). 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.

```
pushq  %rbp
pushq  %rbx
subq   $8   %rsp
movq   %rdi  %rbp
movq   %rsi  %rdi
call brem
movq   %rax  %rbp
popq   %rbx
popq   %rbq
addq   $8   %rsp
ret
```

Since the register we are currently using are caller-saved, they don't survive from calls. So we need to save the address of the main program so we can resume from here later after return. Also registers like %rdi and %rsi we can't do arithmic on them, so move to stack.

3

Name: Xiaohe Yang Student ID: 504640737

A 10
B 11
C 12

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.

010101 01100000
101010

```
bool is_strong (long a) {
    long m1 = 0xAAAA AAAA AAAA AAAA // 101010...
    long m2 = 0X5555 5555 5555 5555 // 0101....
    long indicator1 = a & m1;
    long indicator2 = a & m2;

    int r=((indicator1<<1) & indicator2) | ((indicator2 << 1) & indicator1);
                    // only left shift to prevent sign extension
    return !!r; // convert to one-bit
}
```

10

Name: Xiaohe Yang Student ID: 504640737

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 '-'.

```
long weakadd (long a, long b) {
    long m1 = 0xAAAA AAAA AAAA AAAA; // 1010 1010...
    long m2 = 0x5555 5555 5555 5555 // 0101 0101..
    long carry = (a & b) << 1 ;
    return (a ^ m1) ^ (a ^ m2) ^ carry ;

}
```

(1)

0101010
0100101

1010100
01010010

^ ^ m

Add

Name: Xiaohe Yang Student ID: 504640137

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~~ 64-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~~ $2^{63}-4$ 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.

```
switch: addq   %rdi, %edx
        addq   %rsi, %edx
        jz     .L1          // M+N = 0
        cmpq   $0  %edx
        jg     .L2          // M+N > 0
        addq   $2  4(%rsp)
.L1:    addq   $4  8(%rsp)
.L2:    subq   $5  12(%rsp)
```

(12)

Name: Xiaohe Yang Student ID: 504640737

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+a-1) & ~(a-1)
  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);
}
```

(6)

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

	color	len	data[ ]
size	4 ✓	8 ✓	1 x size of data ✓
offset	0 ✓	8 ✓	16 ✓

alignment = 8

3

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

Because the size of the char array is variable, the offsets will be different for different size.

the offsets of struct cs, and data size.

3 It returns the ~~total~~ offset of struct cs, and data size.

It adds the sizes, and adds buffers depending on alignment.. (8)

Xiaohe Yang Student ID: 504640737

(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.

```
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
```

Name: Xiaohe Yang Student ID: 504640737

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.

0

310