A Computer Science 33 (Fall 2016)
erm 1, 100 minutes, 100 points, open book, open notes.
your answers on the exam, and put your name and

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 moved = a >> b;
    long move = a << (64+~b+1);
    long mask = (1 << 63) >> b;
    return (mask & move) | (~mask & moved);
}
```

Signs Unreadable

In 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
```
movq    %rdi, %rax
cqto
idivq   %rsi
ret
```

B: bquo
```
movq    %rdi, %rax
cqto
idivq   %rsi
ret
```

C: brem
```
movq    %rdi, %rax
negq    %rsi
negq    %rax
cqto
idivq   %rsi
movq    %rdx, %rax
ret
```

D: arem
```
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. ✓ A: aquo B: bquo C: brem D: arem

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?

$\frac{a}{b} = \frac{-a}{-b}$ b/c the negatives cancel, so are equivalent

(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.

-fwrapv is a flag that allows wrapping around for overflow

3

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

1011 0101 | 1001 0100 | 1010 0101 | 1100 0110 | 0110 0011

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.

```
bool is_strong(long a){
    long res = a & (a>>1);
    return !!res;
}
```

cast to unsigned long.

(7)

0000
0001
0011
0010
0111
0101
0110
0100

(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 o = (a & b)<<1;
    long or = a | b;
    long and = a & b;
    long res = (o | or) & ~and;
    return res;
}
```

(12)

0010 0010
0100

0101 0101
o: 1010 or: 0101
and: 001
res: 101

0101 0001
o: 0010 and 0001
or: 0111
res: 0110

0101 0010 and 0000
o: 0000
or: 0101
res: 0111

1001 0101
and: 0001
o: 0010
or: 1101
res: 1110

-7 + 5
-2

-8 + 4 + 2

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;
}
```

-1, does all 123
0, does 23
1, does 3

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.

— because it always subtracts 5 from k, we only need to determine if M+N==0, then J=J+4, else do J=J+4 and I=I+2

```
.L1
movl %rdi, %rax
addl %rsi, %rax
test %rax, %rax
"k=k-5" → ? machine instructions ?
je .L2
"I=I+2"
ret
.L2
"I=I+2"
"J=J+4"
ret
```

(8)

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);
}
```

4

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

int color size = 4 bytes b/c need 2^{32} to range int
offset is 0 b/c is first
long len size = 8 bytes b/c need 2^{64} to range long
offset is 4 bytes b/c alignment is 8 and int already used 4 → 0-3

1

data[]? 8

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

- needed to help allocate correct amount of memory. helps calculate how much memory the "int color; long len" require (includes alignment)
- offset of (struct cs, data) returns how much the header data needs
  len is how much the array needs

3

- returns the size_t of how much memory it needs in bytes
- it works by finding address of data and start of struct and get the diff

(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.

D: not popping off stack correctly, S= stack will break

(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
```

%r13 → %r12 or 12

%r12 → %r13 or 13

+3

(3)