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. 1:40 end

Name: Jonathan Chu Student ID: 104832220

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 m1 = ~(-1 << b); // lowest b bits
  long m2 = ~(-1 << (64-b)); // lowest 64-b bits
  long c = (a & m1) << (64-b);
  long d = (a >> b) & m2;
  return c | d;
}
```

11000000
00000011

Name: Jonathan Chu Student ID: 604832220 (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

```
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
idivq   %rsi
ret
```

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

C:
```
movq    %rdi, %rax
negq    %rsi
negq    %rax
cqto
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?

2a aquo and bquo generate the same machine code because they always produce the same final result. Although the processes differ (%rdx would be set to different values in each function since the signs of the arguments differ), the machine code ends up the same considering it is optimally compiled (-O2). This is not a bug because both functions produce the same result.

[illegible] Student ID: [illegible]

(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.

G

Name: Jonathan Chu Student ID: 004832220

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.

mask every other
" shift
use ^ or & to create result

```
bool is_strong(long a) {
  long m = 0xaaaaaaaaaaaaaaaa;
  long b = a & m;
  long c = (a<<1) & m;
  long d = (a & m)<<1;
  long e = a & (a<<1);
  long f = !((a>>63) ^ ((a<<1)>>63)); // 1 if same bit, 0 if diff bits
  long bc = !(b ^ c); // 0 if strong
  long de = !(d ^ e); // 0 if strong
  return bc | de | f;
}
```

7

Jonathan Chu

Student ID: 604832220

6 (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 c = a^b;
    long d = (~c & a)<<1;
    return c|d;
}
```

12

Name: Jonathan Chu Student ID: 004232220

(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.

```
switch:
  addq %rdi, %rsi
  cmpq $10, %rsi
  jg   .L2
  jge  .L1
  I = I + 2

.L1:
  J = J + 4;

.L2:
  k = k - 5
      ret
```

(V)

Name: Jonathan Chen Student ID: [redacted]

3. 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);
}
```

7

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

member	size	offset	reasoning
color	4 bytes	0 bytes	• color is of type int and is the 1st struct member
len	8 bytes	8 bytes	• len is of type long and is the 2nd member of a struct with alignment of 8 bytes, since its largest member takes 8 bytes
data	(len) bytes	16 bytes	• each element of data is of type char; data is 3rd member of struct with alignment = 8 bytes

3

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

The function needs to know how many bytes to allocate to copy the first two members.

Offsetof works by using the null pointer casted and taking advantage of bitwise operations.

4

Offsetof returns the number of bytes by which the specified member of the specified struct is offset, that is, how many bytes precede it in the struct.

Name: Jonathan Chen   Student ID: [illegible]

...c (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
```

Wrong order

CORRECTIONS

A: movl (%rdi), %rbp   +1

B: malloc

C: movl %eax, %r12

D: popq %r12
popq %r13   +3

(4)

e: Jonathan Chu Student ID: 80432270

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: Moving an 8 byte value into 4 bytes of a register could compromise the value of that pointer. ✓

B: Cannot jump to a nonexistent label ✓

C:

D: If values are popped off the stack in the wrong order, values within certain registers, %r12 and %rbx in this case, will not have the values we think they have, and if the program continued and we tried to use the values in those registers, there would be a big problem. ✓