CS33 Discussion 5

May 5, 2017

Agenda

- Midterm Review
- Floating Point

Midterm Review

Mean : 53.9

Median : 55

Q1

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.

Q1

```
long rcshi (long a, int b) {
  return ((unsigned long) a >> b) | (a << (63-b) << 1);
}
```

Can also use: ^,+ in place of |

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.

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?

Q2

a. A - aquo/bquo
   B - bquo/aquo
   C - brem
   D - arem

b. Behavior is undefined. Compiler chooses what to do.

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.

Q2c

bquo will change

Instead of trap, it will wrap around

Suppose, a = LONGMIN, b=-1

This will cause a trap without fwrapv

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.

Q3a

```
bool is_strong(long a){
        return <>;
}
```

<> can be any of the following:

- (a & (a<<1))!=0
- (a & ((unsigned long)a>>1))!=0
- mask1 = a& 0xAAAA; mask2 = a&0x5555; return !!(mask1 & (mask2<<1))
- mask1 = 0xAAAA; return!!((a&mask) &(a<<1)&mask)

These are most common approaches for this problem. There can be other ways to solve this as well.
Strong integers are numbers which have adjacent 1s in their binary representation. This can be simply done by left shifting the number and then taking AND with the original number. If it didnt have any adjacent 1s, then the result will be 0. Hence, its weak integer. For a strong integer, the result will be non-zero. Right shift can also be done. However, care should be taken to cast it into unsigned integer.

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 ‘-’.

Q3b

```
long weakadd(long a, long b){
	return (a|b|((a&b)<<1))^(a&b);
}
```

The main idea here is to understand that weak integers do not have adjacent 1 bits. Addition then can be simply done by OR. For the carry, result of AND is to be shifted left by 1 bit and the bit where AND gives 1 has to be 0 in result.

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.

Q4

```
        mov %rdi, %rax
        addq %rsi, %rax
        cmp $0, %rax
        jlt .L1
        je .L2
        jmp .L3
.L1 :   addq $2 , I
.L2 :   addq $4 , J
.L3 :   subq $5 , K
```

Note: Be careful that there is no break statement in the code.

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);
}
```

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

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

Q5

	color	len	data
Sizes:	4	8	?
Offsets:	0	8	16

Offset call is required to reach the data part after color and len. It tells how much memory is needed for color and len. Offset is needed so that the machine doesn’t access any padding bits in the struct.
Offsetof : it takes a struct and its data member, returns the offset from the beginning of the struct

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.

```
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
```

Q5c

A: movq (%rdi), rbp

B: call malloc

C: movq %rax, (%r12)

D: popq %r12
popq %r13

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.

Q5d

Error A: moving only 32 bits of (*ptr) into %ebp, will lose the higher bits of address. This would cause error in later instructions 0(%rbp), %edx where it tries to access the (**ptr) and it might crash or access wrong data.

Error B: Malloc shouldn't be a conditional jump and moreover the condition it's using is wrong and would almost never hold. So, malloc won't be called and new struct won't be allocated. This would cause a crash when it access memory out of bound.

Error C: moving half of the %eax to memory, would make the higher bits garbage and the caller function would access wrong data.

Error D: %r13 and %r12 will be swapped for caller function, because it's not following the stack push pop convention.

For further questions, contact:

1	Jin Wang
2 a,b	Shikhar Malhotra
2c	Mingda Li
3a	Aanchal Dalmia
3b	Nikita Tulpule
4	Swathi Patnaikuni
5a,b	Harshada Wadekar
5c	Anurag Pande
5d	Nazanin Farahpour

Floating Point

- So far, have worked with integer data types
  - signed: two's complement
  - unsigned
- Integers: 0, 1, 42, -101, 9001

How to represent a non-integer, like 0.5?

Answer: Floating Point Representation!

Floating Point (IEEE 754)

- Goal: Represent **rational** numbers with a **fixed** number of bits
  - float: 32 bits "single" precision
  - double: 64 bits "double" precision

Floating Point:

Single precision

Sign bit	"Exponent" Field	"Fraction" Field
1 bit	8 bits	23 bits

Two Types of FP

- There are two different "types" of a floating point number
- Case 1: "Normalized"
  - Common case. Represent large and moderately-small values.
- Case 2: "Denormalized"
  - Represent very small values (close to 0).

Case 1: Normalized

1. Normalized

Exp is not all 0's or all 1's

$$V = (-1)^s \times M \times 2^E$$

s = sign bit
M = 1 + f
E = e - Bias

Bias = $2^{k-1} - 1$ = $\begin{cases} 127 \text{ for single} \\ 1023 \text{ for double} \end{cases}$

k is # bits in exp

Case 1: Normalized

Bias = $2^{k-1} - 1$ = 127 for single / 1023 for double

k is # bits in exp

Single precision

0100 0010 0010 1000 0000 0000 0000 0000

What is V?

exp = 100 0010 0 = 0x84 = 8*16 + 4 = 132

f = 010 1000 0000 0000 0000 0000

= 0*2^(-1) + 1*2^(-2) + 0*2^(-3) + 1*2^(-4) = 0.3125

V = (-1)^0 * (1 + 0.3125) * 2^(132 - 127) = **42.0**

s = sign bit

M = 1 + f

E = e - Bias

$$V = (-1)^s \times M \times 2^E$$

Case 2: Denormalized

2. Denormalized

$$V = (-1)^s \times M \times 2^E$$

s = sign bit
M = f
E = 1 - Bias

Bias = $2^{k-1} - 1$ = 127 for single / 1023 for double

k is # bits in exp

Case 2: Denormalized

Bias $2^{k-1} - 1$ = 127 for single, 1023 for double

k is # bits in exp

1000 0000 0010 1100 0000 0000 0000 0000

exp = 000 0000 0 = 0
f = 010 1100 0000 0000
0000 0000

What is V?

= 1*2^(-2) + 1*2^(-4) + 1*2^(-5) = 0.34375
V = (-1)^1 * (0.34375) * 2^(1-127)
= -4.040761830951613e-39

s = sign bit
M = f
E = 1 - Bias

$$V = (-1)^s \times M \times 2^E$$

Case 3: Special Values

- Represent infinity, NaN via certain bit patterns

3a. Infinity

s	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

3b. NaN

s	1	1	1	1	1	1	1	1	$\neq 0$

Note: +Inf and -Inf are different (sign bit).

Case 3: Special Values

- Can represent 0.0 in two ways!
  - All bits 0, and sign bit is 1: -0.0
  - All bits 0, and sign bit is 0: +0.0

Example: Largest/Smallest

- Suppose we use an 8-bit floating-point format. There are 4 exponent bits, and 3 fraction bits.

What is the bias?

```
2^(4-1)-1 = 7
```

What is the smallest/largest positive value?

```
Smallest: 0 0000 001
Largest: 0 1110 111
```

How to represent 1.0?

```
0 0111 000
```

Rounding

- Floating point still can't represent every rational number exactly
  - Why? Finite number of bits (32, 64)
- IEEE standard defines several **rounding modes**

Four Rounding Modes

Mode	\$1.40	\$1.60	\$1.50	\$2.50	\$−1.50
Round-to-even	\$1	\$2	\$2	\$2	\$−2
Round-toward-zero	\$1	\$1	\$1	\$2	\$−1
Round-down	\$1	\$1	\$1	\$2	\$−2
Round-up	\$2	\$2	\$2	\$3	\$−1

Figure 2.36 **Illustration of rounding modes for dollar rounding.** The first rounds to a nearest value, while the other three bound the result above or below.

FP Operations

- After every operation on two floating point values, a round is performed.
  - Ex: (f+g) => round(f+g)

FP: Addition

- Addition commutes correctly
  - `(f+g) = (g+f)`
- Addition is generally *not* associative
  - `(3.14 + 1e10)-1e10 = 0.0`
  - `3.14 + (1e10-1e10) = 3.14`

FP: Multiplication

- Generally not associative
  - `(1e20*1e20)*1e-20 = +Inf`
  - `1e20*(1e20*1e-20) = 1e20`
- Does not distribute over addition

Exercise

Practice Problem 2.53

Fill in the following macro definitions to generate the double-precision values $+\infty$, $-\infty$, and 0:

```
#define POS_INFINITY
#define NEG_INFINITY
#define NEG_ZERO
```

You cannot use any include files (such as `math.h`), but you can make use of the fact that the largest finite number that can be represented with double precision is around 1.8×10^{308}.

Exercise

We assume that the value `1e400` overflows to infinity.

```
#define POS_INFINITY 1e400
#define NEG_INFINITY (-POS_INFINITY)
#define NEG_ZERO (-1.0/POS_INFINITY)
```

C FP: Casting Rules

- In C, exists float and double data types
  - Can cast to/from float types to integer types.
  - Can lose information due to rounding/truncation!

```
#include <stdio.h>
int main() {
    float v = 42.675;
    int foo = (int) v;
    printf("foo is: %d\n", foo);
    return 0;
}
```

```
$ gcc -o code code.c
$ ./code
foo is: 42
```

Casting Rules Quiz

	Exact conversion?	Can overflow/underflow?
int -> float		
int -> double		
float -> double		
double -> float		
double -> int		
float -> int		

Casting Rules Quiz

	Exact conversion?	Can overflow?
int -> float	No! Float can't repr very large ints.	No
int -> double	Yes	No
float -> double	Yes	No
double -> float	No	Yes
double -> int	No: rounded toward zero (1.99 -> 1, -1.99 -> -1)	Yes
float -> int	No: rounded toward zero	Yes

Practice Problem

Will the following evaluate to true or false? (x:int, f:float, d:double)

1. x == (int)(double)x
2. x == (int)(float)x
3. d == (double)(float)d
4. f == (float)(double)f
5. f == -(-f)
6. 1.0/2 == 1/2.0
7. d*d >= 0.0
8. (f+d)-f == d

Answers

1. Yes (double has greater precision and range)
2. No (when x is TMax)
3. No (when d is 1e40, it becomes infinity on right)
4. Yes (double has greater precision and range)
5. Yes (Sign bit is only changed)
6. Yes (First converted to float, and then divide)
7. Yes (may overflow to infinity)
8. No (f:1e20, d=1, f+d will round to 1e20, hence LHS will become 0)

Thank You