C + x86 Assembly : Representing and Manipulating Information

By Amanda Falke
September 2014
Abstractmachines at gmail

Questions which I try to answer from :
Bryant and O'Halloran's *"Computer Systems: A Programmer's Perspective"*

Format: Questions in **bold,** and *answers* follow.

2.59 ◆◆ Write a C expression that will yield a word consisting of the least significant byte of x, and the remaining bytes of y. For operands x = 0x89ABCDEF and y =0x76543210, this would give 0x765432EF.

ANSWER EXPLANATION: AND with a 1 preserves. AND with a 0 destroys. We will AND with a 1 for information we want, and AND with a 0 for information we do not want. Combining these operations on both bit vector variables, and then adding those two bit vector variables, produces our desirable, masked answer.

ANSWER FOR 2.59:

```
int n  = ( x & 0x000000FF) + ( y & 0xFFFFFF00);
```

2.61 ◆◆
Write C expressions that evaluate to 1 when the following conditions are true, and to 0 when they are false. Assume x is of type int.
A. Any bit of x equals 1.
B. Any bit of x equals 0.
C. Any bit in the least significant byte of x equals 1.
D. Any bit in the most significant byte of x equals 0.
Your code should follow the bit-level integer coding rules (page 120), with the additional restriction that you may not use equality (==) or inequality (!=) tests.

A: ANSWER EXPLANATION: An AND with 1 and x would return 1 if [bit of x] is one. Note that the BOOL data type is not native to c, so we will return an int data type.

THE ANSWER FOR A:

```
int n = x && 1 || 0;
```

// ALTERNATE ANSWER:
int n = !!(x);

B: ANSWER EXPLANATION: An XOR with 1 and x would return 1 if [bit of x] is zero. Similarly, a 1, AND'ed with a complemented 0 would return 1 (1 && NOTZERO == 1).

THE ANSWER FOR B:

int n = ~x && 1;

// Alternate answer:
int n = !!(~x);

C: ANSWER EXPLANATION: To return a 1 or true value if LS Byte of x equals 1: check for IF any of bits in the least significant byte (meaning the rightmost 8 bits) are zero. Hence, we will AND the least significant byte with 1111 1111, which will only return a 1 if 1. This is because 1 & 1 is 1. See AND truth table. Note that we assume 32 bit LITTLE ENDIAN machine byte order.

THE ANSWER FOR C:

int n = !!(x && oxFF); // do a LOGICAL AND on 8 bits with the least significant bytes.

// note that oxFF = 1111 1111, so this is the same as **int n = !!(x & 11111111);**

D: *ANSWER EXPLANATION: To return a 1 or true value if MS Byte of x equals 0: Assuming IA32, with 32 bit integers, the most significant byte (meaning the 8 bits that are on the leftmost side of the bit array or bit string you may have) are followed by 24 more bits (because 8 bits + 24 bits = 32 bits).. Hence, we will assume 32 bits for this answer, and have 8 bits in the most significant byte that will return a true value if the most significant byte has zeros. The bitwise solution would be to XOR these bits with a 1 to return a true value; the logical solution would be to AND bits with COMPLEMENTED. Note that little endian byte order, which is assumed for x86 IA32, puts the least significant values in the lowest order (leftmost) bit places.* BIG ENDIAN WOULD BE: int n = 1111 1111 0000 0000 0000 0000 0000 0000 && !x); LITTLE ENDIAN IS: // int n = 0000 0000 0000 0000 0000 0000 1111 1111 && !x;

THE ANSWER FOR D:

int n = (~x && ox000000FF); // LITTLE ENDIAN

2.66 ◆◆
Write code to implement the following function:
/*
*** Generate mask indicating leftmost 1 in x. Assume w=32.**
*** For example 0xFF00 -> 0x8000, and 0x6600 --> 0x4000.**
*** If x = 0, then return 0.**
*/
int leftmost_one(unsigned x);
Your function should follow the bit-level integer coding rules (page 120), except that you may assume that data type int has w = 32 bits.
Your code should contain a total of at most 15 arithmetic, bit-wise, and logical operations.
Hint: First transform x into a bit vector of the form [0 . . . 011 . . . 1].

ANSWER EXPLANATION: Solution: bit smearing. This "smears" the high order bits over lower order bits. This solution is chosen because the 32-_builtin_clzll(v) function cannot be used per requirements. Bit smearing uses the BITWISE OR operator |= to create a mask. At the end of the algorithm, function returns an XOR with first bit smearing shift (highest order bit).

```
int leftmost_one( unsigned int x )
{
x |= x >> 1;
x |= x >> 2;
x |= x >> 4;
x |= x >> 8;
x |= x >> 16;

return x ^= x >> 1; // bitwise XOR
}
```

2.76 ◆◆
Suppose we are given the task of generating code to multiply integer variable x by various different constant factors K. To be efficient, we want to use only the operations +, -, and <<. For the following values of K, write C expressions to perform the multiplication using at most three operations per expression.
A. K = 17:
B. K = −7:
C. K = 60:
D. K = −112:

A - D: *ANSWER EXPLANATION: Note that one left shift will multiply a bit vector by POWERS OF TWO for EACH LEFT SHIFT. For example, to multiply by 16, you would left shift four times. 2^4 = 16. Note also the twos complement paradigm: that the number 10000000 is equal to -128, and that the number 10000001 is equal to -127. That is, the "bits other than the MSB" add to positive values, and the MSB indicates the negative value by the bit place. As put by the CS tutors, in twos complement, the non-MSB bits "dig you out of negative, towards positive." The answers to this question use shifting algorithms for multiplication for two's complement numbers.*

FINAL SOLUTIONS: FOR A where K = 17: shift x to the left 4 times and add 1. each shift is a power of 2. 16 is 2 to the 4, hence we have four left shifts. A left shift is multiplication.

THE ANSWER FOR A: K = 17:.
x = x << 4; // 2^4 = 16, so this multiplies x by 16.
x = x + 1;

THE ANSWER FOR B: K = -7: Note that -7 is 1001
x = x << 3; // this is 1000, or x^3 = 8. In two's complement, 1000 is -8.
x = x + 1; // -8 + 1 is -7.

THE ANSWER FOR C: K = 60. Note that 60 is 0011 1100
x = x << 6; // this is 64
x = x - 4; // 60

THE ANSWER FOR D: K = -112: Note that -12 is 1111 0100
x = x << 7; // this is 128, 10000000, or x^7 = 128. In two's complement, 1000000 is -128.
x = x + 16; // -128 + 16 is -112.

2.86 ◆◆
Consider a 16-bit floating-point representation based on the IEEE floating-point format, with one sign bit, seven exponent bits (k = 7), and eight fraction bits (n = 8). The exponent bias is 2^7–1 – 1= 63.
Fill in the table that follows for each of the numbers given, with the following instructions for each column:
Hex: The four hexadecimal digits describing the encoded form.
M: The value of the significand. This should be a number of the form x or x/y , where x is an integer, and y is an integral power of 2.
E: The integer value of the exponent.

V : The numeric value represented. Use the notation x orx × 2z, where x and z are integers.

description	hex	m (8 fraction bits)	e	v
-0	0000	00000000	00000..	0
Smallest value -2	???	??	??	??
512	0x44000000	0000000	.00000100	512 in IEEE 754
largest denormalized	????	????	??	????
infinity	um, infinity??			
0x3BB0	0x3BB0		0.0036430	

2.90 ◆◆
Around 250B.C., the Greek mathematician Archimedes proved that 223/71 <π < 22/7 .
Had he had access to a computer and the standard library <math.h>, he would have
been able to determine that the single-precision floating-point approximation of π has the hexadecimal representation 0x40490FDB. Of course, all of these are just approximations, since π is not rational.
A. What is the fractional binary number denoted by this floating-point value?

ANSWER:
3.141593 in decimal is equal to **011.00100100**00111111011100

B. What is the fractional binary representation of 22/7 ? Hint: See Problem 2.82.

ANSWER:
22/7 = 3.14285714

3 in base 10 is equal to 011 in base 2, so **011.00100100**100...

C. At what bit position (relative to the binary point) do these two approximations to π diverge?

ANSWER:
After the 8th fractional decimal place. For clarification, note which parts are BOLDED in the answers above to parts A and parts B. Thanks! :)

7.9 ◆
Consider the following program, which consists of two object modules:

```
1 /* foo6.c */
2 void p2(void);
3
4 int main()
5 {
6 p2();
7 return 0;
8 }
```

```
1 /* bar6.c */
2 #include <stdio.h>
3
4 char main;
5
6 void p2()
7 {
8 printf("0x%x\n", main);
9 }
```

When this program is compiled and executed on a Linux system, it prints the string
"0x55\n" and terminates normally, even though p2 never initializes variable main.

Can you explain this?

ANSWER: Rule #1 of the LINKER is that "multiple symbols are not allowed," and Rule #2 of the LINKER is that "strong symbols are preferred over weak symbols." The variable main is uninitialized, and hence, variable main is a weak symbol. The function main() is a strong symbol because it is initialized. The LINKER actually does indeed allow for multiple symbols, so long as there is a strong symbol that takes precedence over the weak symbol. The result of these operations are that the main; is ignored, while strong symbol main() is linked by linker. The uninitialized main is completely ignored, because it's weak!

7.15 ◆◆
Performing the following tasks will help you become more familiar with the various tools for manipulating object files.
A. How many object files are contained in the versions of libc.a and libm.a on your system?

ANSWER:

The answer is to view object files in an archive using the ar command, option t:
ar -t libc.a

Computer response: no such files.

B. Does gcc -O2 produce different executable code than gcc -O2 -g?

ANSWER:
Yes, it does produce different executables; with or without gdb is the difference.

Using the -g tag on the command line generates symbolic information for the gdb debugger as well as many error messages.

C. What shared libraries does the gcc driver on your system use?

ANSWER:

The answer is to view object files in an archive using the ar command, option t:
ar -t libc.a

Computer response: no such files.