CSO: Recitation 5

Goktug Saatcioglu

10.03.2019

1 Representation

Values in programs are represented as bytes in memory. 1 byte = 8 bits = 2 hexadecimal values = 0 to 255 in base-10 decimal.

- Hexadecimal is base 16 representation. One hexadecimal digit encodes 4 bits. It is 0 to 9 followed by A to F. When writing hexadecimal, to make it clear that we are writing hexadecimal, we use the notation $0 \times \ldots$.
- Bytes can be ordered in two ways in memory. The first is Big Endian meaning the most significant byte has the lowest address and the second is Little Endian meaning the most significant byte has the highest address.
- For example, if we have a variable x such that $x = 0 \times 01234567$ then its most significant byte is 01. Address x is given by $\&x$ 0×100. If the architecture is big endian then we store x as `|01|23|45|67|` where `|01|` is stored at address `0x100` and `23` is stored in address `0x101` and so on. Alternatively, if the architecture is little endian then we store x as `|67|45|23|01|` where `|01|` is stored at address `0x103` and `23` is stored in address `0x102` and so on.
-

1.1 Representing Strings

Recall that strings in C are represented as arrays of characters. Each character is encoded using the ASCII standard. We have a standard 7-bit encoding of character set where Character `'0'` has code `0x30` and digit `i` has code `0x30+i`. Furthermore, strings should be null-terminated so that we know when to stop reading more from the memory (recall that we only allocate enough space for the string and if we do not null terminate we will end up with undefined behavior if we keep reading). Regardless of whether we have little endianness of big endiannes, the representation of the string will be the same at the array level. Thus, byte ordering is not an issue. We see that: Byte ordering is an issue for a single data item. An array is a group of data items.

2 Boolean Algebra

We have bit-wise operations `&` (AND), `|` (OR), `~` (NOT) and `^` (XOR). Interpret `1` as true and `0` as false. We can use bit-wise operations together to achieve other bit-wise operations. For example, `~(A&B)` (NAND), `~(AB)`— (NOR) and `~(A^B)` (XNOR). It is technically enough to have either a NAND or a NOR to implement all Boolean algebra. Can you see why?

2.1 Lifting operations to bit-vectors

Operations on a bit-vector (i.e. a set of bytes) (e.g. an integer is a bit vector of 4 bytes) is done bit-wise. So if we have `01101001` and `01010101` the bit-wise operations work bit-by-bit starting for the least significant bit of the two vectors. Bit operations can be used on any "integral" data type (long, int, short, char, unsigned).

2.2 Comparing to logical operations

Recall that the logical operations are `&&` (Logical AND), `||` (Logical OR) and `!` (Logical NOT). `0` is interpreted as being false and everything else is interpreted as being true. The result of logical operations are always either `0` or `1`. Examples

- `0x41 = 0xBE`

- ` 0x00 = 0xFF`
- `0x69 & 0x55 = 0x41`
- `0x69 | 0x55 = 0x75`
- `!0x41 = 0x00`
- `!0x00 = 0x01`
- `!!0x41 = 0x01`
- `0x69 && 0x55 = 0x01`
- `0x69 || 0x55 = 0x01`
- `p && *p` (prevents null pointer access using short circuiting)

The type bool did not exist in standard C89/90. It was introduced in C99 standard. You may need to use the following switch with gcc: `gcc {std=c99 ....` Then you may `#include <stdbool.h>` in your C code and use the type `bool`.

2.3 Shift operations

- Left Shift: x << y
  - Shift x left by y positions
  - Throw away extra bits on left
  - Fill with 0's on right
- Right Shift: x >> y
  - Shift x right y positions
  - Throw away extra bits on right
  - type 1: Logical shift: Fill with 0's on left
  - type 2: Arithmetic shift: Replicate most significant bit on right
- Undefined Behavior: Shift amount < 0 or $\geq$ size of x

2.4 Integer Representation

We have two types of integers: (1) signed (can be +,- or 0) and (2) unsigned (can be + or 0). To convert an unsigned bit-vector x with w bits we do

$$B2U(x) = \sum_{i=0}^{w-1} x_i 2^i.$$

An unsigned integer represents values from 0 to 2^{n-1} where n is the number of bits we use for our binary integer. Furthermore, unsigned binary arithmetic is just like normal grade-school arithmetic so, for example, for addition we can do addition with carry to compute the result of any addition (i.e. add from left-to-right, propagate a carry).

2.4.1 Signed integer representation

There are many ways of representing negative numbers in binary. We need to take into account the balance of the representation, the number of zeros in the representation, and the ease of doing arithmetic operations. The options are:

- Sign magnitude: use the most significant bit to represent the sign. The rest of the bits are used to represent the number. In this case, we have two zeros: $+0$ and -0. Furthermore, the representation is balanced meaning we have $3, 2, 1, +0, -0, -1 - 2, -3$. Doing operations is a little complicated but manageable.
- One's complement: take the complement (bit-wise not) of a positive number to represent its negative counterpart. Again, we have two zeros: $+0$ and -0. Furthermore, the representation is balanced meaning we have $+0, 1, 2, 3, -3, -2, -1, -0$. Doing arithmetic is easy but our representation has redundancies.
- Two's complement: the the one's complement of a positive number and then add 1. Now we have only one 0 and the representation is no longer balanced meaning we have $+0, 1, 2, 3, -4, -3, -2, -1$. Doing arithmetic is again easy and we no longer have redundancies. Thus, this is the best option.
  - The big idea is: For each positive number (X), assign value to its negative (-X), such that X + (-X) = 0 with "normal" addition, ignoring carry out (at the very end).

If n bits represent 2^n possible values then with 2's complement we assign 2^{n-1} of them to positive values (starting with 0) and 2^{n-1} of them to negative values. If the most significant bit (MSB) is 0 then the integer is positive. If the most significant bit (MSB) is 1 then the integer is negative. Range of an n-bit number: -2^{n-1} through $2^{n-1} - 1$. Notice that the most negative number -2^{n-1} has no positive counterpart.

2.4.2 Converting Binary (2's C) to Decimal

- If MS bit is one (i.e. number is negative), take two's complement to get a positive number.
- Get the decimal as if the number is unsigned (using power of 2s).
- If original number was negative, add a minus sign.

You can `#include <limits.h>` to get the maximum and minimum value for each data type.

3 Type casting

There are two types of casting:

- Explicit casting between signed and unsigned int tx, ty;

```
unsigned ux, uy;
tx = (int) ux;
uy = (unsigned) ty;
```

- Implicit casting also occurs via assignments and procedure calls

```
tx = ux;
uy = ty;
```

To convert an unsigned integer to a signed integer we do the following:

- Keep the bit presentation
- Re-interpret

Result: bit-pattern may not change but the numerical value may change.

3.1 Casting Rules

If there is a mix of unsigned and signed in single expression, signed values implicitly cast to unsigned (Including comparison operations ¡, ¿, ==, ¡=, ¿=). The rules are as follows:

`int -> unsigned int -> long -> unsigned long -> long long -> unsigned long long -> float -> double -> long double.`

Consider the following program:

```
#include <stdio.h>

int main() {
  int i=-7;
  unsigned j = 5;

  if(i > j) printf("Surprise!\n");

  return 0;
}
```

The condition evaluates to true because we are comparing a signed int to an unsigned one. We convert the signed `i` to unsigned which becomes `9` and the condition ends up evaluating the condition to true. (+/-16 is the conversion value for 4 bit signed/unsigned integers)

4 Bit Hacks

One of the advantages of knowing about bits and bit-level boolean logic is the ability to use bit-wise boolean operations to implement interesting and possibly faster versions of common operations. Let's start with swapping. How can we swap two integers without using a temporary variable? Normally, we swap two values by using a temporay variable `tmp`. However, this isn't actually necessary. We can use binary xor. Consider the folllowing C function.

```
void swap(int* x, int* y) {
  *x = *x ^ *y;
  *y = *x ^ *y;
  *x = *x ^ *y;
}
```

How can we reason about the above code? We will use the following facts:

- $A \oplus B = B \oplus A$ (commutativity)
- $(A \oplus B) \oplus C = A \oplus (B \oplus C)$ (associativity)
- $A \oplus 0 = A$ (identity element)
- $A \oplus A = 0$ (inverse element)

Then, it must be that $(A \oplus B) \oplus A = B$ which is exactly what our swap function is computing. There is one corner cases here, can you see what it is? (Consider the case where $A = B$ and our rule for the inerse element, what happens?)

Let's try something else. How can we check whether a number is even or odd by only using bit-wise operations? Consider the following C function.

```
int is_even(int x) {
  return !(x & 1);
}
```

The reason this works is because for us to know whether a number is even or odd we only need to check the last bit. If the last bit is zero then the number must be even and otherwise odd. So we use 1 to make sure our result is only 0 or 1 since bit-wise operators work bit-by-bit so that we can exract the last bit of x. The not is necessary as 0 is logical false and we want to return true if the number is even (not false).

Now another task: how can we manually take the 1's and 2's complement of a number using only bit-wise operators? Consider the following C program.

```c
int ones_complement(int x) {
  return ~x;
}
int twos_complement(int x) {
  return ones_complement(x) + 1;
}
```

This one should have fairly straightforward.

Next, let's check whether a number is a power of 2. Consider the following C program.

```c
int is_power_of_2(int x) {
  return x && !(x & (x-1));
}
```

How does this "hack" work? Well a power of 2 must only have a single bit set. Then, if we do $x - 1$ and x was a power two all bits to the left of the bit that was 1 should become 1. For example, $16_2 = 00010000$ and $(16 - 1)_2 = 15_2 = 00001111$ such that `x&(x-1)` will become 0. If x is not a power of 2 then x and $x - 1$ will share at least one bit that is 1 such that we get 1. The left hand side of the logical and takes care of the case when $x = 0$.

Finally, how can we check whether the n-th bit of a number is set to 1. Consider the following C program.

```c
int is_n_bit_set(int x, int n) {
  return !(x & (1 << n))
}
```

Try explaining the logic behind this program yourself. If you see what's going on here then you should also be able to do other stuff such as setting/unsetting the n-th bit of a number. As a challenge you can also try toggling the n-th bit of a number. This means that if the bit is 1 then it becomes 0 and if it is 0 then it becomes 1. You must do this without actually checking whether the bit was set. (Hint: XOR)