

Lecture 1

Intro, Number Rep

The value of the i th digit d in any number base is $d \times \text{Base}^i$ where i starts 0 and increases from right to left.

Signed Magnitude

Similar to unsigned numbers but the most significant bit (first bit) represents if the number is positive or negative, this bit is called the *sign bit*.

The issue with this representation is that there are two representations for the number 0, and operations are slow since there is additional work required to handle the sign bit.

Two's Complement

The convention to represent signed numbers is called **Two's Complement**. The left most bit is the sign bit, and $1111 \dots 1111_{\text{two}}$ is the most negative number. The advantage twos complement has over *sign and magnitude* representation is that there is only one zero.

Negation Shortcut

Simply flip bits, then add 1 to the result. This reason this shortcut works is because the sum of a number and its inverted representation must be $1111 \dots 1111_{\text{two}}$ (*there is no carries*).

Now since $x + \bar{x} = -1$, we have $\bar{x} + 1 = -x$

Sign Extension Shortcut

- for positive 16 \rightarrow 32 bit binary numbers, just add 16 zeros in the most significant bit
- For negative 16 \rightarrow 32 bit binary numbers, copy the sign bit (which is 1) 16 times, placing it on the left of the number.

This works because positive numbers have an infinite number of leading zeros and negative numbers have an infinite number of leading ones.

One's Complement

A representation in which the negative of a none's complement is found by inverting each bit. So now $11 \dots 110_{\text{two}}$ is equal to -1. This representation is similar to twos complement but has two 0s

Biased Notation

A notation that represents the most negative value by $00 \dots 000_{\text{two}}$ and the most positive value by $11 \dots 111_{\text{two}}$, with zero typically having the value $10 \dots 000_{\text{two}}$, thereby biasing the number such that the number plus the bias has a nonnegative representation.

In simpler words, bias notation is like unsigned representation but has a shift (a bias), shifting the range of values on the unsigned number line to the left, allowing representation of negative numbers.

With this new system, to interpret a binary string in bias notation, we evaluate the number as if it was unsigned, then add the bias (the bias is usually negative). Note we do this because we shift the numbers to the left by a bias.

Example 1.1: Interpreting a Stored Binary

Assume we have a -127 bias with an 8-bit number.

To read $0b0000\ 1001$, we treat it as if it was unsigned, which gives us 9. Then we add the bias to get our value -118 :

$$9 + (-127) = -118$$

To store a decimal number as a binary string in bias notation, we first subtract the bias (bias is negative so basically add) then store the resulting number as an unsigned binary.

Lemma 1.2

Subtracting a bias will never give us a negative number.

This is because the range of numbers is from $[-B, 2^n - 1 - B]$, where B is the magnitude of the bias. So a number x , will always be greater than or equal to $-B$, and obviously $x - (-B) \geq 0$.

Questions

- is right shift and left shift division and multiplication by 2 respectively?

Chapter 0: Introduction

The arguments to functions are passed by copying the value of the argument, and it is impossible for the called function to change the actual argument in the caller. When desired to achieve "call by reference," a pointer may be passed explicitly, and the function may change the object to which the pointer points.

Array name are passed as the location of the array origin, so array arguments are effectively call by reference.

C is not a strongly-typed language. It is relatively permissive about data conversion, although it will not automatically convert data types with the wild abandon of PL/I

Chapter 2: Types, Operators, and Expressions

A **string constant** is a sequence of zero or more characters surrounded by double quotes. Underneath, a string is an array whos elements are single characters. The compiler automatically places the null character `\0` at the end of each string, so the program can conveniently find the end.

Bitwise Operations

In C, a leading 0 on an int constant implies *octal*. A leading 0x indicates *hexa-decimal*.

The bitwise AND operator `&` is used to turn off bits.

The bitwise OR operator `|` is used to turn on bits.

The one's operator `~` (one's complement) is used to flip bits.

2.1 Memory Model Review

Memory works very similar to an array. You can think of most version of memory you work with conceptually to be one very long array,

2.1.1 The Stack

The stack is memeory that is automatically allocated and freed by the system, and grows from top-down. **What is the stack used for?**

- Anything considered "Temporary"

- This includes: local variables, local constants, arguments to functions, local information about a function call.

Stack Frames, function calls

- Everytime a function is called, a new "stack frame" is allocated on the stack
- Stack Frame Includes:
 - return "instruction" address (who called me?), arguments, and space for other local variables
- When function ends, stack frame is tossed off the stack; automatically frees memory for future stack frames.

2.1.2 The Heap

Questions

- What is an example of local information about a function call?

06/22/23

Lecture 3

C Pointers, Arrays, Memory Management

06/27/23

Lecture 4

C Memory

Stack grows down, i.e, starts af FF and decreases.

The stack memory is destroyed as you return from functions

The heap memory does not, you are responsible for destorying (freeing) the memory when you don't need it anymore.

Chapter 5: Pointers and Arrays

Definition 4.1: Pointer

A pointer is a variable that contains the address of another variable. The convention to naming pointer variables is the prefix them with `p_<name>`.

An pointer that points to an `int` is declared as follows:

```
int *p_x;
```

It says that the combination of `*p_x` is an `int`, that is, if the derefencing operator is used on `p_x`, it is equivalent to a variable of type `int`.

- The unary operator `&` gives the *address* of an object.
- The dereference operator `*` used in the context as an operand to an address, access that address to fetch the contents to which it points to.

Pointers and Syntax

- pointers can occur on the left side of assignments. That is `p_x` points to `x`, then `*p_x = 1` sets `x` to 1.
- Normally a pointer can point to only one type, however, `void *` is a type that can point to anything. (use sparingly)

Arrays

Arrays and pointers have a strong relationship. The name of an array is actually an address location to the zeroth element in the collection.

So saying `p_arr = arr` is equivalent to saying `p_arr = &arr[0]`

Moreover, indexing into an array is actually using pointer arithmetic under the hood. For example: doing `arr[3]` is equivalent to `*(a + 3)`.

Address Arithmetic

The only difference between an array name and a pointer is that a pointer is a variable, but an array name is a **constant**. Constructions like `a = p_arr` or `a++` or `p = &a` are illegal.

Memory Organization

Memory is just a large, single-dimensional array with the, that is byte-addressable, where the address acting as the index to that array. 8 bits = 1 byte, and 4 bytes = 1 word.

Type Sizes: Assume we are working with a 32-bit system.

- `int` occupy 4 bytes, aka 32 bits.
- `char` occupy 1 byte
- *pointers* occupy 4 bytes. This is because memory address are 32 bits long in a 32-bit system (why?), therefore the pointer values, the address that the pointer points to, are also 32 bits long.

Definition 4.2: Endianness Systems

Endianness affects how a group of bytes are stored and read in memory.

Little-Endian: In a little-endian system, memory is stored with the most significant byte at the highest address.

Big-Endian: In a big-endian system, memory is stored with the most significant byte at the lowest address.

Example 4.3

For the program below assume, the memory is drawn as such in little endian system.

```
int main() {
    int x[2]; // address of x is: 0x1000 0000
    x[0] = -2, x[1] = 44513;
    char y[] = "ADEL"; // address of y is: 0x1000 0010
    char *c y;
}
```

The corresponding memory model is as follows

Address	Byte 0	Byte 1	Byte 2	Byte 3
0x1000 0000	0xFE	0xFF	0xFF	0xFF
0x1000 0004	0xE1	0xAD	0x00	0x00
0x1000 0008	-	-	-	-
0x1000 000C	-	-	-	-
0x1000 0010	0x41	0x44	0x45	0x4C
0x1000 0014	0x00	-	-	-
0x1000 0018	0x10	0x00	0x00	0x10

Note: While ints get stored in reverse, character arrays or strings are stored in increasing memory addresses.

- type declaration tells compiler how many bytes to fetch on each access through pointer.
-

Dynamic Allocation

- malloc: memory Allocation
- calloc: cleared allocation
- realloc: re-allocation

68 6c 70 74 78 7c 80 84 88 8c 90

Questions

- is there double pointers, a pointer that points to another pointer? **Yes, you write a function to increment a pointer. Here it will accept a double point as an argument**
- Is there null in C? Can you initialize a pointer to be null? **Yes, C has a null keyword, NULL**
- It is said that the difference between arrays and pointers is that array names is a constant, not a variable. Can you have a pointer to a constant? Why is `p = &arr` illegal?

06/27/23

Lecture 5

Floating Point

Just as we can show decimal numbers in scientific notation, we can also show binary numbers in scientific notation:

$$1.0_{\text{two}} \times 2^{-1}$$

To keep a number in normalized form, we need a base that allows us to shift the binary point left or right to have on nonzero digit to the left of the decimal point. Only base 2 fulfills this since multiplication by 2 is a left shift and division is a right shift.

Definition 5.1: Floating point

Computer arithmetic that represents numbers in which the binary point is not fixed, i.e., it floats. The floating-point representation is as such:

31	30	23	22	0
s	exponent			fraction/mantissa

The scientific notation in binary form of the float is a singly nonzero digit to the left of the binary point (**normalized form**) as such:

$$1. \underbrace{xxxxxxxx}_{\text{mantissa/significand}}_{\text{two}} \times 2^{yyyy \rightarrow \text{exponent}}$$

Where $1.xxxxxxxxx$ is the significand ($0 < S < 1$), and $yyyy$ is called the exponent.

To pack more numbers in the significand, IEEE 754 makes the leading 1-bit of a normalized binary numbers implicit. So now, the mantissa is 1 + 23-bit significand field. Therefore, the simple RISC-V floating number as such:

$$(-1)^S \times (1 + F) \times 2^E$$

Where S is the sign of the floating-point number, F involves the value in the fraction field, generally a number between 0 and 1, also called the *mantissa*. And E comes from the exponent field.

Note however, float comparisons are hard with this form. The most negative exponent is all 1s, and 1_{ten} is a single 1 bit in the least significant bit. The negative number looks larger. Therefore we introduce bias notation to shift the numbers where now the most negative number is $00 \dots 00_{\text{two}}$ and the most positive as $11 \dots 11_{\text{two}}$. IEEE 754 uses a bias of 127 for single precision, and 1023 for double precision.

Biased exponent means that the value represented by a floating-point number is really:

$$(-1)^S \times (1 + F) \times 2^{E - \text{Bias}}$$

Example 5.2

What is the decimal equivalent of the following IEEE 754 single-precision binary floating point number?

31	30		23	22		0
1		1000 0001			111 0000 ... 0000	

$$= (-1)^1 \times 1.111 \times 2^{129-127} \quad (1)$$

$$= -1 \times 1.111 \times 2^2 \quad (2)$$

$$= -1 \times 111.1 \quad \text{shift the decimal point by 2} \quad (3)$$

$$= -1 \times (4 + 2 + 1 + \frac{1}{2}) \quad (4)$$

$$= -7.5 \quad (5)$$

Step Size

Because we have a fixed # of bits, we cannot represent all numbers. **Step size** is the spacing between consecutive floats with a given exponent.

What we really are asking for is what is the next representable number after y? before y?

The next step size to why is just adding y to the smallest bit in the significand times the same exponent.

$$y + ((0.0\dots001) \times 2^{(E-Bias)})$$

Note we multiply by the same exponent because y decimal was shifted, so we also need to shift the smallest bit decimal to the same position.

Note: the bigger the exponent, the bigger the step size since it's shifted more. And the smaller the exponent, the smaller the step size

Representing Zero

Note: Zero has no normalized representation (there is always an implicit 1 in the significand)