Information Storage: Recap

Hexadecimal Notation

Decimal	0	1	2	3	4	5	6	7
Binary	0000	0001	0010	0011	0100	0101	0110	0111
Hexadecimal	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7
Decimal	8	9	10	11	12	13	14	15
Binary	1000	1001	1010	1011	1100	1101	1110	1111
Hexadecimal	0x8	0x9	0xA	0xB	0xC	0xD	0xE	0xF

Patterns:

- 2^n = $100...0_2$ (1 followed by n × 0s), e.g., 1, 2, 4, & 8.
- 2^n-1 = $11...1_2$ (n × 1s), e.g., 0, 1, 3, 7, & 15 (0xF).
- $11...10_2$ = 2 × $11...1_2$, (for the same strings of 1s), e.g., 6 (0110_2) = 2 × 3 (0011_2), 10 (1010_2) = 2 × 5 (0101_2), 14 (1110_2, 0xE) = 2 × 7 (0111_2), & 12 (1100_2, 0xC) = 2 × 6 (0110_2)

These patterns cover all hexadecimal digits except 5, 9, 0xB, & 0xD. But 5 = 4 + 1, 9 = 8 + 1, 0xB = 0xA + 1, & 0xD = 0xC + 1.

Representations

Bit patterns in memory or registers can represent:

- Integers
- Floating point numbers
- Characters
- Strings
- Machine instructions (i.e., code)
- Addresses
- Bit vectors

Types

C declaration	**C declaration**	**Bytes**
Signed	**Unsigned**	**x86_64**

char	unsigned char	1*
short	unsigned short	2
int	unsigned int	4
long	unsigned long	4
int32_t	uint32_t	4
int64_t	uint64_t	8
char *		8
float		4
double		8

Comments

- I have only shown the sizes for the machines on which we will do the labs.
- sizeof(type) is a C expression that returns the size of the given type in a number of bytes (e.g., sizeof(unsigned short)).
- Only the size of char is always going to be 1, the others will depend on the compiler, two different compilers on the same machine may choose different representations (i.e., sizes for the built-in types).
- The only thing the C standard promises is sizeof(char) ≤ sizeof(short) ≤ sizeof(int) ≤ sizeof(long).

Integers

- Finite, not very large, but precise
- Associative law holds
- Signed or unsigned
    - Two's complement representation

Floating Numbers

- Finite, much larger, but imprecise
- Associative law doesn't hold
- Representation like scientific notation (e.g., 6.022×10^{23})
  - Usually a sign, mantissa, and biased-exponent
  - Base 2 or 10
  - Also bit patterns reserved for $-\infty$, $+\infty$, and NaN (not a number, e.g., 0/0)

Characters

- char (or unsigned char) type
- ASCII character encoding
  - **https://en.wikipedia.org/wiki/ASCII** (https://en.wikipedia.org/wiki/ASCII)
  - man ascii
  - Originally based on the English alphabet (i.e., can't do Greek letters)
    - Unicode extends ASCII to include all other alphabets

Strings

- Array of characters terminated with the null (having value 0) character
- Always stored contiguously in memory

Machine Instructions

- Code executed by the microprocessor

Addresses

- For instance char *p
- Beware of pointer arithmetic

Bit vectors

- String of bits

Alignment

The compiler usually likes to assign an address to a variable according to its size (more precisely according to the size of the variable's base type). For instance, for a variable of a basic type, the compiler will assign an address that is a multiple of that type's size. Say, you declare "long l;", the compiler will almost always select an address for "l" that is a multiple of 8 (assuming the size of long is 8). When we talk about caches, we will understand the reason behind this behavior.

Byte ordering

A multi-byte programming object (e.g., a long) is usually stored in memory as a contiguous sequence of bytes, with the address of the object given by the smallest address of the bytes used.

For ordering the bytes representing an object, there are two common conventions:

- Little endian: store the object with the least significant byte first and most significant byte last.
- Big endian: store the object with the most significant byte first and least significant byte last.

The machines in our lab are based on the Intel/AMD processors which are little endian. IBM and Oracle processors are big endian. ARM can be either, although Android and IOS set it to little endian.

Byte ordering usually does not matter. It matters in the following cases:

1. Data communication over the network.
2. Looking at byte sequences representing integer data (e.g., output of the disassembler).
3. Code that circumvent the type system (e.g., casting).