Spring 2013

CSE2421 Systems 1
Introduction to Low-Level Programming and Computer Organization

Kitty Reeves

TWRF 8:00-8:55am
What’s next...

Computer Systems...
A Programmer’s Perspective
The role of the operating system

- Protect the computer from misuse
- Provide an abstraction for using the hardware so that programs can be written for a variety of different hardware
- Manage the resources to allow for reasonable use by all users and programs on a computer
The UNIX Operating System

- Developed in 1970s at Bell Labs
- Kernel written in C, also developed at the same time
  - C was developed for the purpose of writing UNIX and systems programming
- We are using a variant of UNIX named Linux
  - Other UNIX variants exist, such as Solaris, and the various BSDs (OpenBSD, NetBSD, FreeBSD, OSX)
Linux - OS

Linux: A True Story:

Week One
Hey, it's your cousin. I got a new computer but don't want Windows. Can you help me install "Linux"?
Sure.

Week Two
It says my Xorg is broken. What's an "Xorg"? Where can I look that up?
Hmm, lemme show you man pages.

Week Six
Due to auto-config issues, I'm leaving Ubuntu for Debian.
Uh or Gentoo. Uh oh.

Week Twelve
You haven't answered your phone in days. Can't sleep. Must compile kernel.
I'm too late.

Parents: Talk to your kids about Linux... before somebody else does.
A file is a sequence of bytes - not a magical container holding the bytes, but the bytes themselves.

How this information is treated depends on the context.

- The same sequence of bits can be used to represent a character, or an integer, or a floating-point number, or an instruction, or...

- It's all a matter of interpretation.

- % emacs hellob.c &
Why assembly language?

- Instruction based execution
  - Each program on a computer is a sequence of instructions written in machine language
  - Processor executes one instruction at a time in a program, then executes the next one in turn
  - To study code in this form, it's helpful to use assembly language rather than machine language code

```
gcc -S hellob.c
```
Chances are, you’ll never write programs in assembly

- Compilers are much better & more patient than you are

But: Understanding assembly is key to machine-level execution model

- Behavior of programs in presence of bugs
  - High-level language models break down

- Tuning program performance
  - Understand optimizations done/not-done by the compiler
  - Understanding sources of program inefficiency

- Implementing system software
  - Compiler has machine code as target
  - Operating systems must manage process state

- Creating / fighting malware
  - x86 assembly is the language of choice!
The compilation system... revisited

Type in program using an editor of your choice (file.c); plain text

\[ .c + .h = .i \] which is the “ultimate source code”? i.e. # includes expanded and #defines replaced

\[ .i \rightarrow .s \] which is assembler source code

\[ .s \rightarrow .o \] which is an object file; fragments of machine code with unresolved symbols i.e. some addresses not yet known (vars/subrs).

\[ .o + \text{library links} \rightarrow \text{a.out (default name)}; \text{resolves symbols, generates an executable.} \]
Declarations and the Preprocessor

#include

- provides ability to include declarations from other files
  - usually have the file name extension .h
- generally only include function prototypes, and type and variable declarations
- actual code is kept in a different file, usually with the extension .c
  - files with code are compiled individually to object code and then linked together to create a file with executable code

Two ways to use #include

- #include <stdio.h> for standard system files
- #include "swap.h" for user-written files

- Very rarely/never #include a .c file (link .c files)
Compilation stages

- **Preprocessor**
  - used to make sure the program parts see declarations they need (and other purposes too, e.g., macros)
  - directives begin with a # (pound sign)
  - do not end in a ; like C statements do

- **Translation**
  - makes sure individual parts are consistent within themselves
  - creates an object (.o) file

- **Linking**
  - joins one or more compiled object files together
    - includes matching function call to callee across separate files
    - and matching global variable use to where it’s defined
  - makes sure caller/callee consistent with each other
  - creates an executable file (the convention in Unix is no filename extension is used for executable files, although sometimes .x or .exe are used)
C programs must be compiled to be executed

Use the **gcc** program to build your programs

- invocation: **gcc** options source-files
- common options
  - **-ansi** enforces the ansi standards
  - **-g** enable debugging
  - **-Wall** warn about common things that may be problems
  - **-Werror** treat warnings as errors
  - **-o** filename place output in filename
  - **-c** only compile to object file, don't link (.o file created)
  - **-S** generate assembly code, don’t link (.s file created)

- a very simple compilation command
  - **gcc** file.c

- a simple compilation command
  - **gcc -g -Wall -Werror -ansi -o prog1 prog1.c tools.c**

- executable is run in UNIX by just typing its name (sometimes preceded by "./", like this: **./prog1**)

12
A look ahead to Chp. 3

Disassembler
- A tool that determines the instruction sequence represented by an executable program file

Unix command
- gcc –o hellob hellob.c
- objdump –D –t –s hellob
  -d, --disassemble
  • Display assembler contents of executable sections
  -D, --disassemble-all
  • Display assembler contents of all sections
  -S, --source
  • Intermix source code with disassembly
  -s, --full-contents
  • Display the full contents of all sections requested
  -t, --syms
  • Display the contents of the symbol table(s)
  -T, --dynamic-syms
  • Display the contents of the dynamic symbol table
Hardware Organization (big picture)

- **CPU**: Register File, ALU, Bus Interface
- **System bus**
- **Memory bus**
- **I/O bridge**
- **Main memory**
- **I/O bus**
- **Memory**
- **Display**
- **Disk Controller**
- **Disk*”
- **Expansion slots for other devices such as network adapters, video cards, etc.**
- **USB controller**
- **Mouse**
- **Keyboard**
- **Graphics adapter**
- **User**

- hellob executable stored on disk*

---

- **PC**
- **Bus Interface**
The OS sandwich

Runs the software and manages the hardware

RISC vs CISC
LOAD/STORE
ADDRESS BUS
ETC
DATA BUS
ADDRESSIBILITY
BIG/LITTLE ENDIAN
ALIGNMENT
ISA
PIPELINING

SOFTWARE
HARDWARE

Operating System
HW organization details

- **Bus**
  - Transfers one “word” at a time
    - Fundamental system parameter
    - Amount can fetch from memory at one time
    - Tends to be the size of the data bus

- **I/O Devices**
  - System’s connection to the external world
  - Transfers information back and forth between the I/O bus and an I/O device

- **Main Memory**
  - Temporary storage
  - Holds both the program and the data it manipulates
    - Von Neumann architecture
  - Is organized as a linear array of bytes each with its own unique address starting at zero
HW organization details (cont)

Processor (CPU)
- Interprets/executes instructions stored in main memory
- Updates the PC to point to the next instruction

PC
- Points at (contains the address of) some machine-language instruction in main memory

Register file
- Small storage device that consists of a collection of word-sized registers, each with their own name

ALU
- Computes new data and address values

ISA – instruction set architecture defines
- The processor state
- The format of the instructions
- The effect each instruction will have on the state
- Instructions:
Example... go from the bedroom to the kitchen
- In a studio apartment
- In a town house
- In a medium priced home
- In the White House

Example... find an address in computer memory
- When you have 10 bytes of memory
- When you have 10,000 bytes of memory

Cache
- Smaller and faster
- Temporary staging areas
- Goal \(\Rightarrow\) make the transfer/copy operations happen asap
  - It’s easier and cheaper to make processors run faster than it is to make main memory run faster!

Hierarchy
- The storage at one level serves as a cache for storage at the next lower level
- The farther you are away from the action, the longer it takes to make something happen
Abstraction

Provided by the OS

Process (chp. 8)
- The running of a program done by the processor
- Threads = multiple execution units
- Includes memory and I/O device (i.e. files abstraction)

Virtual Memory (chp. 9)
- Provides each process with the illusion that it has exclusive use of the main memory
- Program code and data
  - Includes files
  - Begins at same fixed address for all processes
  - Address space (chp. 7)

Files (chp. 10)
- Sequence of bytes
Address Space... a quick look

- An array of 8-bit bytes
- A pointer is just an index into this array

<table>
<thead>
<tr>
<th>ADDRESS SPACE</th>
<th>Description/info</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel virtual memory</td>
<td>Memory invisible to user code</td>
</tr>
<tr>
<td>User stack (created at run time)</td>
<td>Implements function calls</td>
</tr>
<tr>
<td>Memory mapped region for shared libraries</td>
<td>Ex. printf function</td>
</tr>
<tr>
<td>Run-time heap (created at run time by malloc/calloc)</td>
<td>Dynamic in size</td>
</tr>
<tr>
<td>Read/write data</td>
<td>Program (executable file)</td>
</tr>
<tr>
<td>Read-only code and data</td>
<td>Fixed size</td>
</tr>
</tbody>
</table>

32/64 bit starting address

Address 0

Notice symbolically drawn with memory “starting” at the bottom
What is a system?

“A collection of intertwined hardware and systems software that must cooperate in order to achieve the ultimate goal of running application programs”
Let’s check...

* see sizeck.c
* try –m32 option

ANSI rules

- Variables of type char are guaranteed to always be one byte.
- There is no maximum size for a type, but the following relationships must hold:
  - sizeof(short) <= sizeof(int) <= sizeof(long)
  - sizeof(float) <= sizeof(double) <= sizeof(long double)

<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1 byte</td>
<td>-128 to 127</td>
</tr>
<tr>
<td>unsigned char</td>
<td>1 byte</td>
<td>0 to 255</td>
</tr>
<tr>
<td>short</td>
<td>2 bytes</td>
<td>-32768 to 32767</td>
</tr>
<tr>
<td>unsigned short</td>
<td>2 bytes</td>
<td>0 to 65535</td>
</tr>
<tr>
<td>int</td>
<td>4 bytes</td>
<td>-2147483648 to 2147483647</td>
</tr>
<tr>
<td>unsigned int</td>
<td>4 bytes</td>
<td>0 to 4294967295</td>
</tr>
<tr>
<td>long</td>
<td>X</td>
<td>-2147483648 to 2147483647</td>
</tr>
<tr>
<td>unsigned long</td>
<td>4 bytes</td>
<td>0 to 4294967295</td>
</tr>
<tr>
<td>float</td>
<td>4 bytes</td>
<td>1.175494e-38 to 3.402823e+38</td>
</tr>
<tr>
<td>double</td>
<td>8 bytes</td>
<td>2.225074e-308 to 1.797693e+308</td>
</tr>
</tbody>
</table>
Every computer has a “word size”
- Nominal size of integer and pointer data
- Address space depends on word size
- \( 2^{\text{word-size-in-}\#\text{bits}} \)

Is it big enough?
- 64-bit high-end machines becoming more prevalent
- Portability issues – insensitive to sizes of different data types

<table>
<thead>
<tr>
<th># bytes</th>
<th># bits</th>
<th># of values ((2^{#\text{bits}}))</th>
<th>low</th>
<th>high</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
<td>256</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>16</td>
<td>65536</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>24</td>
<td>16777216</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>32</td>
<td>4294967296</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>40</td>
<td>1.09951E+12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>48</td>
<td>2.81475E+14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>56</td>
<td>7.20576E+16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>64</td>
<td>1.84467E+19</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>72</td>
<td>4.72237E+21</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>80</td>
<td>1.20893E+24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>88</td>
<td>3.09485E+26</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>96</td>
<td>7.92282E+28</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>104</td>
<td>2.02824E+31</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>112</td>
<td>5.1923E+33</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>120</td>
<td>1.32923E+36</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>128</td>
<td>3.40282E+38</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
What does this mean?
- abc
- 123
- 3.14
- 0x61

Representation (encode?)
- ASCII
- Simple Binary*
- One’s complement
- Two’s complement*
- Binary Coded Decimal
- Floating-point*

Limited number of bits to encode a value
Will there be a time that the value we want to encode does not fit? Yes! OVERFLOW
Need to be aware of the range of values that each limited number of bits will hold
Inaccuracies exist...

(see overflw.c)
Floating point

Google

“what every computer scientist should know about floating point”

- Squeezing infinitely many real numbers into a finite number of bits requires an approximate representation (rounding)

Overflow → +∞

Not associative

- Due to finite precision of the representation
  - A float has roughly seven decimal digits of precision
  - see floatpt.c
The machine level program generated has no information about data types.

- It’s the C compiler that maintains this type of information.

The different mathematical properties of integers vs floating-point arithmetic stem from the difference in how they handle the finite-ness of their representations.

- Integers – smaller values but more precise
- Real – wide range of values, but only approximately

Defines ranges of values.

Computer security vulnerabilities

Need to know/understand before progress to machine level programming (chp.3)
**Information Storage (details)**

- Byte = smallest addressable unit of memory
- Virtual memory = very large array of bytes
- Address = how byte of memory is uniquely identified
- Virtual address space = the set of all possible addresses

Reminder: no data typing at this level
Hexadecimal Notation (Hex)

- **Base 16**
- Useful in describing bit patterns
- Digits 0-9 and A-F
- In C
  - 0x or 0X prefix interpreted as hex value
  - Not case sensitive
  - Example
    - \( FA1D37B_{16} \) --> 0Xfa1d37b, 0xFA1D37B, 0xfA1d37B
- Easy to convert to/from hex, octal and binary

<table>
<thead>
<tr>
<th>DEC</th>
<th>HEX</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2^0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2^1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>2^2</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>2^3</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>F</td>
<td></td>
</tr>
</tbody>
</table>
FYI: “bit” stands for “binary digit”

Fact: $2^4 = 16$ and $2^3 = 8$

The power is the # of bits per hex/octal digit

Binary to Hex

Every 4 bits = 1 hex digit

Octal – base 8

Digits 0-7

Binary to Octal

Every 3 bits = 1 octal digit

Example: FA1D37B$_{16}$

Practice problem 2.1 (pg. 35)
When value $x_{10}$ is a power of 2

- $x = 2^n$ for some non-negative integer $n$, then if
  - $x = 16_{10} = 2^4 = 10000_2 = 10_{16}$
- Binary rep of $x \rightarrow 1$ followed by $n$ zeroes
- Hex rep of $x$ is $\rightarrow n = i + 4j$
  - Reminder: hex digit 0 in binary is 0000
  - Leading hex digit where $0 \leq i \leq 3$
    - 1 ($i=0$)
    - 2 ($i=1$)
    - 4 ($i=2$)
    - 8 ($i=3$)
  - Followed by $j$ hex 0s
- Examples:
  - $n=9 \rightarrow$ so $i = 1$ and $j = 2 \rightarrow x = (2)(00)$ hex i.e. $x = 200_{16}$
  - $n=6 \rightarrow$ so $i = 2$ and $j = 1 \rightarrow x = (4)(0)$ hex i.e. $x = 40_{16}$
  - See more practice problems 2.2 pg. 35
Convert to Decimal (why?)

- **Base ten (decimal):** digits 0-9
  - E.g., \(316_{10} = \)

- **Base eight (octal):** digits 0-7
  - E.g., \(474_8 = \)

- **Base 16 (hexadecimal):** digits 0-9 and A-F.
  - \(13C_{16} = \)

- **Base 2 (binary):** digits 0, 1
  - \(100110_2 = \)

In general, radix \(r\) representations use the first \(r\) chars in \(\{0...9, A...Z\}\) and have the form \(d_{n-1}d_{n-2}...d_1d_0\).

Summing \(d_{n-1} \times r^{n-1} + d_{n-2} \times r^{n-2} + ... + d_0 \times r^0\) converts to base 10.
Every base is base 10.

EXPLANATION
In general, $10_x = x_{10}$

- $10_2 = 2$
- $10_3 = 3$
- $10_4 = 4$
- $10_5 = 5$
- $10_6 = 6$
- $10_7 = 7$
- $10_8 = 8$
- $10_9 = 9$
- $10_{10} = 10$

http://cowbirdsinlove.com/43
Decimal to non-decimal base

Base Conversions

- Convert to base 10 by *multiplication of powers*
  
  \[
  10012_5 = ( \quad )_{10}
  \]

- Convert from base 10 by *repeated division*
  
  \[
  632_{10} = ( \quad )_8
  \]

- Converting base \(x\) to base \(y\): convert base \(x\) to base 10 then convert base 10 to base \(y\)
Convert from base 10

More Practice

123_{10} = (               )_3 and check

1234_{10} = (               )_{16} and check

Another way from decimal to base n

<table>
<thead>
<tr>
<th>n^8</th>
<th>n^7</th>
<th>n^6</th>
<th>n^5</th>
<th>n^4</th>
<th>n^3</th>
<th>n^2</th>
<th>n^1</th>
<th>n^0</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>128</td>
<td>64</td>
<td>32</td>
<td>16</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

- From LEFT TO RIGHT, ask “how many” and subtract
  - (219)_{10} = (               )_2 = (               )_{16}
#include <stdio.h>

int main()
{
    char buf[255] = {0};
    int x = 1234;
    printf("int as hex %x\n", x);
    sprintf(buf, "%x", x);
    printf("buf is %s\n", buf);
    return 0;
}
#include <stdio.h>
void main() //dec2bin.c
{
    long int m, no=0, a=1;
    int n, rem;
    printf("Enter any decimal number->");
    scanf("%d",&n);
    m=n;
    while(n!=0) {
        rem = n%2;
        no = no+rem*a;
        n = n/2;
        a = a*10;
    }
    printf("The value %ld in binary is->",m);
    printf("%ld",no);
} Compile and run it with the following input to determine the output:
1 =
2 =
4 =
8 =
16 =
32 =
64 =
128 =
256 =
1 with 15 zeroes =
1 with 20 zeroes =
Hex and Binary addition/subtraction

- Hex add first, then convert hex to binary and add
  - $A + 8 =$
  - $13 + F =$
  - $BEAD + 4321 =$

- Subtract in hex first, then convert each value to binary and subtract
  - $5CD2 - 2A0 =$
  - $3145 - 1976 =$
  - $A8D2 - 3DAC =$

→ carry/borrow 16 each time, since the next place is 16 times as large (see practice problems 2.4 pg 37)
Addressing and byte ordering

- Two conventions (for multi-byte objects)*
  - What is the address of the object?
  - What is the order of the bytes in memory?

- Typically
  - Multi-byte objects are stored contiguously
  - The address of the object is given as the smallest address of the bytes used

- 4-byte integer stored as hex value at address 0x100
- So &x = 0x100, and
- The 4 bytes of x would be stored at memory locations 0x100, 0x101, 0x102, 0x103

* Does not apply to characters because they are single byte values
Big Endian
- “The big end goes at byte zero”
- “big end” means the most significant byte of the given value

Little Endian
- “The little end goes at byte zero”
- “little end” means the least significant byte of the given value
- “Byte zero” means the smallest address used to store the given value

Example: hex/given value is 0x01234567
- What is the big end of the given value? \( \rightarrow \) 01
- What is the little end of the given value? \( \rightarrow \) 67
- What is the lower memory address i.e. byte zero? 0x100

<table>
<thead>
<tr>
<th>Byte order</th>
<th>0x100</th>
<th>0x101</th>
<th>0x102</th>
<th>0x103</th>
</tr>
</thead>
<tbody>
<tr>
<td>Big Endian</td>
<td>01</td>
<td>23</td>
<td>45</td>
<td>67</td>
</tr>
<tr>
<td>Little Endian</td>
<td>67</td>
<td>45</td>
<td>23</td>
<td>01</td>
</tr>
</tbody>
</table>
## Big/Little Endian

<table>
<thead>
<tr>
<th>Byte order</th>
<th>Big Endian</th>
<th>Little Endian</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>01</td>
<td>67</td>
</tr>
<tr>
<td>0x101</td>
<td>23</td>
<td>45</td>
</tr>
<tr>
<td>0x102</td>
<td>45</td>
<td>23</td>
</tr>
<tr>
<td>0x103</td>
<td>67</td>
<td>01</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Byte order</th>
<th>Big Endian</th>
<th>Little Endian</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>01</td>
<td>67</td>
</tr>
<tr>
<td>0x101</td>
<td>23</td>
<td>45</td>
</tr>
<tr>
<td>0x102</td>
<td>45</td>
<td>23</td>
</tr>
<tr>
<td>0x103</td>
<td>67</td>
<td>01</td>
</tr>
</tbody>
</table>

- There is no technological reason to choose one byte ordering convention over the other.
- Need to choose a convention and be consistent.
- Typically invisible to most application programmers as results are identical.
- What if transferring data, though?
- Need to know when looking at integer data in memory.
X86 is little endian

- Largely, for the same reason you start at the least significant digit (the right end) when you add—because carries propagate toward the more significant digits. Putting the least significant byte first allows the processor to get started on the add after having read only the first byte of an offset.
- After you've done enough assembly coding and debugging you may come to the conclusion that it's not little endian that's the strange choice—it's odd that we humans use big endian.
- A side note: Humans mostly read numbers and only sometimes use them for calculation. Furthermore we often don't need the exact numbers when dealing with large quantities - taking that into account - big endian is a sensible choice for humans.
- It reflects the difference between considering memory to always be organized a byte at a time versus considering it to be organized a unit at a time, where the size of the unit can vary (byte, word, dword, etc.)
When is byte ordering an issue?
1. Communications over a network between different machines
2. Representation of integer/real numeric data

3. Circumvention of normal type system
   - Using a “cast” to allow an object to be referenced according to a different data type from which it was created
     - Use and even necessary for system-level programming
   - Can cast such that the value is a sequence of bytes rather than an object of the original data type

```c
#include <stdio.h>
void main()
{
    int x = 0x12345678, i;
    unsigned char *xptr = &x;
    printf("the integer x is 0x%x\n",x);
    for (i = 0; i < 4; i++)
        printf("byte %d is %.2x\n",i+1, *(xptr+i));
}
```
(castex.c)
Systems can have different...

- word size
- byte sizes for each type
- endian-ness (for numeric values)
- representation of pointers
- character encoding schemes (ascii, ebcdic, unicode)
- instruction formats (again, just a sequence of bytes)
- ETC