MASTERNOTES

COMPUTER ARCHITECTURE

YOUNG, FALL 2013

**- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -**

***---CHAPTER 1---***

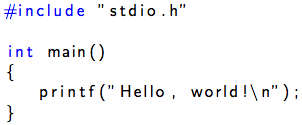
**- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -**

**INFORMATION IS BITS**

★**GENERAL:**

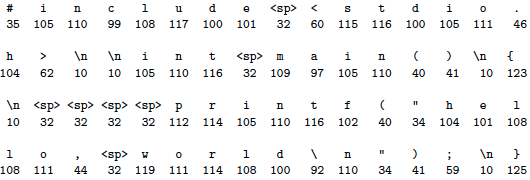
* all information in a system (disk files, programs, data, etc.) is represented as a series of bits
* each ASCII character is a byte (8 bit chunks)

★**HELLO.C PROGRAM:**

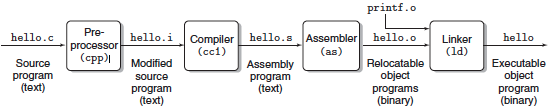


* hello.c is a ***text file***
* ***text files:*** files such as helo.c that consist of exclusively ASCII characters
* ***binary files:*** all other files that are not text files

★**MACHINE REPRESENTATION OF HELLO.C PROGRAM:**



★**STEPS IN COMPILATION SYSTEM:**

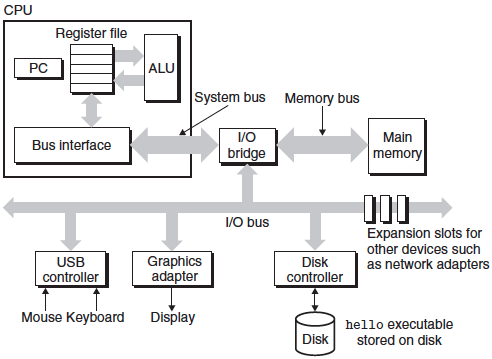


* **preprocessing phase:** modifies the original C program according to the directives that begin with the # character.
  + If system defined the file, use < >
  + If you defined the file, use " "
  + ***example:*** #include <stdio.h>
    - command in line 1 of hello.c tells the preprocessor to read the contents of the system header file stdio.h and insert it directly into the program text. The result is another C program, typically with the .i suffix.
    - "go find the directory file and use it in program"
    - this is not actually in C and that's why semicolon isn't used
  + ***example:*** #define VARIABLE 5
    - defines constants in directory
    - acts like static variables
    - these lines are not actually in C
    - convention: uppercase names
* **compilation phase:** the compiler translates the text file hello.i into the text file hello.s, which contains an assembly-language program.
  + each statement in an assembly-language program exactly describes one low-level machine-language instruction in a standard text form.
  + useful because it provides a common output language for different compilers for different high-level languages.
  + ***example:*** C compilers and Fortran compilers both generate output files in the same assembly language.
* **assembly phase.** Next, the assembler translates hello.s into machine language instructions, packages them in a form known as a relocatable object program, and stores the result in the object file hello.o.
  + The hello.o file is a binary file whose bytes encode machine language instructions rather than characters.
  + If we were to view hello.o with a text editor, it would appear to be gibberish.
* **linking phase:** Notice that our hello program calls the printf function, which is part of the standard C library provided by every C compiler.
  + The printf function resides in a separate precompiled object file called printf.o, which must somehow be merged with our hello.o program.
  + The linker handles this merging.
  + The result is the hello file, which is an executable object file (or simply executable) that is ready to be loaded into memory and executed by the system.

★**REASONS TO KNOW HOW COMPILATION WORKS:**

* **optimizing program performance:**
  + it's good to know about efficiency from the ground up
  + ***example:*** switch or if-else statements
* **understanding link-time errors**
  + some of the worst bugs are related to linker problems
  + ***example:*** difference between static and global variables
* **avoiding security holes**
  + ***example:*** buffer overflow vulnerabilities account for the majority of security holes in network and internet serves
  + first step to secure programming: understand consequences of he way data and control information are stored on program stack; it makes a difference

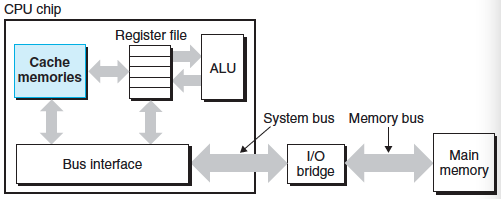
★**HARDWARE ORGANIZATION OF A SYSTEM:**

****

* **buses:** electrical conduits that carry bytes of information back and forth between components
  + **words:** systems are designed to transfer these fixed-sized chunks of bytes
  + number of bytes in word is a system parameter that varies across systems
  + most systems today have either 4 bytes (32 bits) or 8 bytes (64 bits)
* **I/O devices:** system's connection to the external world
  + ***example:*** keyboard and mouse for input, display for output, disk drive for long-term storage of program
  + the program initially is on the disk
  + each I/O device is connected to the I/O bus by either a ***controller***or an***adapter***, whose job is to transfer information back and forth between the I/O bus and the I/O device
    - **controller:** chip sets in the device or the motherboard
    - **adapter:** card that plugs into the slot of the motherboard
    - **motherboard:** system's main circuit board
* **main memory:** temporary storage device that holds both a program and the data it manipulates while the processor is executing the program
  + physically, consists of DRAM (dynaic random access memory)
  + memory is organized as a linear array of bytes, each of whch has its own unique address (array index) starting at 0
* **processor:** the central processing unit (CPU)
  + engine that interprets instructions stored in main memory
  + **program counter (PC):** the core of the processor that is a word-sized storage device
    - points to / contains the address of some machine-level instruction in main memory
  + repeatedly executes the instruction pointed at by the PC from the time the power is on and until the power is off
  + reads instruction then performs some simple operations (of which there are only a few) that revolve around ***main memory***, ***register file***, and ***ALU*:**
    - **register file:** small storage device that consists of a collection of word-sized registers with its own unique name
    - **ALU (arithmetic/logic unit):** computes new data and address values
  + ***example*** ***simple operations:***
    - **load:** copy a byte or a word from main memory into a register, overwriting the previous contents of the register.
    - **store:** copy a byte or a word from a register to a location in main memory, overwriting the previous contents of that location.
    - **operate:** copy the contents of two registers to thealu, perform an arithmetic operation on the two words, and store the result in a register, overwriting the previous contents of that register.
    - **jump:** extract a word from the instruction itself and copy that word into the program counter (pc), overwriting the previous value of the pc.

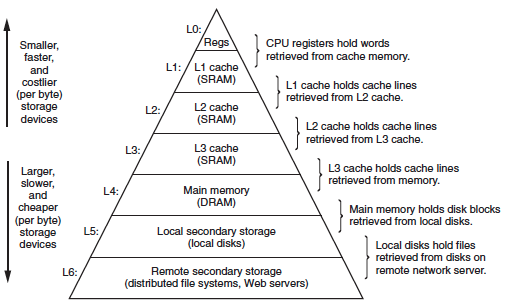
★**CACHES MATTER:**

* exploit this to improve performance
* system spends a lot of time moving information from one place to another
* ***example:*** (hello.c program) disk 🡪 copied to main memory 🡪 copied to processor
* all this copying slows down the "real work" of the program
  + it's our goal to make these copy operations run as fast as possible
  + faster devices cost more – easier/cheaper to make processors run faster than the main memory
* **processor-memory gap:** the difference between operation times between the processor's memory and any other storage (disk, register file) since the main memory is a large memory and is slower
  + **cache memories:** smaller, faster storage devices that temporary hold data that the processor is likely to need in the near future
    - can be accessed almost as fast as register files
    - **locality:** by accessing data in localized regions, we can get the goods of both a very large memory and a very fast one by



★**HEIRARCHY OF STORAGE DEVICES:**

* ***example:*** inserting a smaller, faster storage device (cache) between the processor and a larger, slower device (main memory)
* top to bottom hierarchy – slower, larger, less costly on the bottom
* main idea: storage at one level serves as a cache for storage at the next lower level

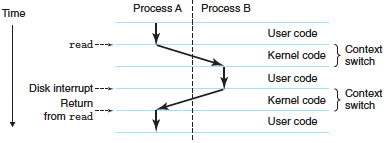


★**OPERATING SYSTEM MANAGES THE HARDWARE:**

* **operating system:** layer of software between the application program and the hardware
  + all attemps by an application program to use the hardware (keyboard, display, disk, main memory) must go through the operating system
* **two primary purposes of OS:**
  + 1) protect the hardware from misuse by runaway applications
  + 2) provide application with simple, uniform mechanisms for manipulating complicated low-level hardware devices

★**PROCESSES:**

* **process:** operating system's abstraction for running a program
  + multiple processes can run concurrently on the same system
  + **context switching:** instructions of one process are interleaved with the instructions of another process such that the operating system can transfer control from the current process to some new process.
    - Saves context of current process, restores context of new process, then passes control to the new process, wherein the new process picks up exactly where it left off.
    - ***example:*** two processes



* + **multicore processors:** can execute several programs simultaneously
  + **unicore system (traditional):** can only execute one program at a time
  + context: state information that the process needs in order to run that the operating system keeps track of.
    - current PC values, register file, contents of main memory
  + **system call:**
    - passes control to the OS (same process as in context switching but with the OS)

★**THREADS:**

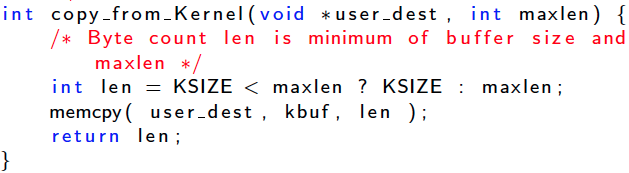
* stuff here

**- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -**

**SLIDES #1: GREAT REALITIES**

★**(1) INTS ARE NOT INTEGERS; FLOATS ARE NOT REALS**

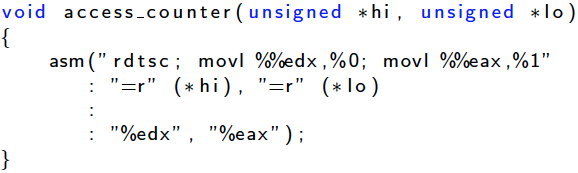
* arithmetic program arithmetic
* ***example:*** is for floats and ints?
  + Ints are not integers
  + ***float:***
  + ***int:*** not necessarily correct for ints because large numbers are not representable by int data type
* ***example:*** is for floats and ints?
  + Floats are not reals
  + ***float:*** since the small number 3.14 gets lost in the large float
  + ***int:*** yes for unsigned and signed ints
* ***example:* code security:**

****

* + a negative or a positive int (that is, you can be passed a signed or unsigned int) can be passed to maxlen but the compiler won't tell you if the data types don’t match
  + this is because it makes C run much faster when it's run without a bunch of checks like in java 🡪 assumes that you know what you’re doing

★**(2) YOU'VE GOT TO KNOW ASSEMBLY:**

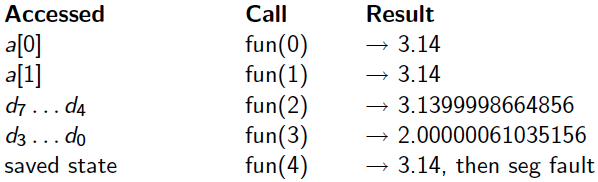
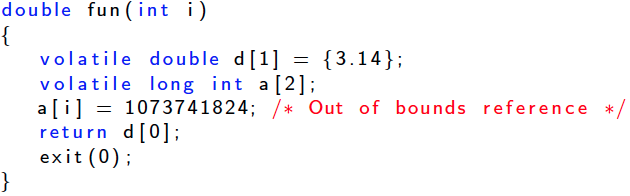
* ***example:* diving down to assembler level:**



* + you can write in assembly within C
  + reason to do so: you cn acess machine resourcs hat you can't get with plain C such as the machine level counter

★**(3) MEMORY MATTERS**

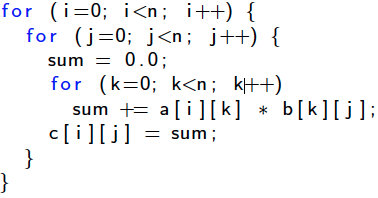
* memory is not unounded
  + it must be allocated and managed
  + applications are memory dominated
  + you have to "free up" memory once you are done using it
* ***example:* memory referencing bug:**

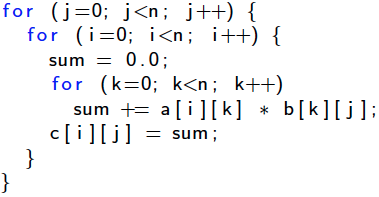


* + run on little endian machine
  + a is a pointer to the front of the array
  + you can call a[2], a[7], a[10000], but the C compiler wont tell you that it is wrong
  + in this case, a[2] is an out of bounds reference and d[0] is stored in memory directory after a so it messes up the data stored in d[0]

|  |  |  |  |
| --- | --- | --- | --- |
| a[0] | a[1] | d[0] | … |

* ***example:* memory performance:**





* + matrix multiplication
  + bottom one takes 20x longer than the top one
  + c is **row major** order so changing it to **column order** makes a huge difference
  + it affects the # times you are accessing the cache

★**(4) THERE'S MORE TO PERFORMANCE THAN ASYMPTOTIC COMPLEXITY**

* **asymptotic complexity:** time complexity, big O notation, worst case analysis, etc.
* you have to optimize at multiple levels: algorithm, data representations, procedures, and loops
* must understand the system to optimize performance
  + how programs compiled and executed
  + improve performance without destroying code modularity and generality
  + measure program performance and identify bottlenecks

★**(5) COMPUTERS DO MORE THAN EXECUTE PROGRAM**

* I/O system, communicate with other computers over networks, etc.

**- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -**

***---CHAPTER 2---***

**- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -**

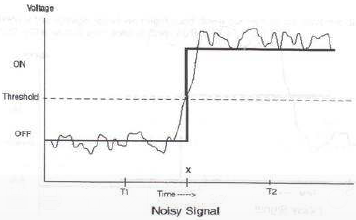
**REPRESENTING DATA**

★**INFORMATION STORAGE:**

* most computers use chunks of 8 bits (bytes) as the smallest addressable unit of memory
* **virtual memory:** the way machine level programs view memory as array of bytes
  + conceptually a very large array of bytes
  + implemented with hierarchy of different memory types: SPRAM, DRAM, disk, etc.
* **address:** each byte is identified by this unique number
* **virtual address space:** set of all possible addresses; conceptual image presented to the machine-level program (it's not actual a physical thing)

★**WHY BINARY AND NOT BASE 10 ON COMPUTERS:**

* **binary:**

****

* + easy to store
  + reliably transmitted even in noisy, inaccurate wires
* **base 10:**
  + difficult to store 10 values
  + difficult to transmit
  + messy to implement digital logic functions (addition, multiplication, etc.)

★**BYTE-ORIENTED MEMORY ORGANIZATION:**

* byte = 8 bits
* nibble = 4 bits
* byte addressable: each byte has an address
* if 8 bits, then possible possible addresses
* binary range in base 2 = 00000000 to 11111111
* decimal range in base 10 = 0 to 255

★**ENCODING BYTE VALUES:**

* **hexademical:**
  + more convident to write bit patterns in base 16
  + range from 00 to FF
  + in C, constants starting with 0x or 0X is recognized as hexademical

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **HEX** | **DEC** | **BINARY** | **HEX** | **DEC** | **BINARY** |
| **0** | **0** | **0000** | **8** | **8** | **1000** |
| **1** | **1** | **0001** | **9** | **9** | **1001** |
| **2** | **2** | **0010** | **A** | **A** | **1010** |
| **3** | **3** | **0011** | **B** | **B** | **1011** |
| **4** | **4** | **0100** | **C** | **C** | **1100** |
| **5** | **5** | **0101** | **D** | **D** | **1101** |
| **6** | **6** | **0110** | **E** | **E** | **111** |
| **7** | **7** | **0111** | **F** | **F** | **1111** |

* + not case sensitive

★**CONVERSIONS:**

* **binary to octal**: split into chunks of 3
* **binary to decimal:** evaluate the magnitude (skip the first binary to the left and then just figure it out by math).
* **decimal to binary:** divide the give decimal number by 2 and write down the remainders. then list the remainders from bottom to top.
* **binary to hexadecimal:** splitting it into groups of 4 bits each. Note, however, that if the total number of bits is not a multiple of 4, you should make the leftmost group be the one with fewer than 4 bits, effectively padding the number with leading zeros. Then you translate each group of 4 bits into the corresponding hexadecimal digit.
  + ***example:***

Macintosh HD:Users:jungyoon:Desktop:Screen Shot 2013-09-03 at 12.18.05 AM.png

* **hexadecimal to binary:**
  + ***example:***

**Macintosh HD:Users:jungyoon:Desktop:Screen Shot 2013-09-03 at 12.19.01 AM.png**

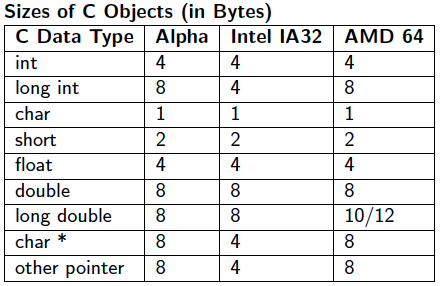
* **hexadecimal to decimal:**
* **decimal to hexadecimal:** divide the give decimal number by 16 and write down the remainders. then list the remainders from bottom to top

★**WORDS:**

* **word size:** every computer has a normal size of integer and pointer data
  + **maximum size of virtual address:** virtual addresses can range from 0 to , giving the program access to at most bytes
  + ***example:*** 32-bit word size = 4 GB virtual address space
    - addresses of successive words differ by 4 bytes
    - ***example:*** first address is at byte 0, second address is at byte 4

★**DATA SIZES:**

* computers/compilers support multiple data formats using different ways to encode data such as integers and floating point
  + ***example:*** machines have instructions for representing integers as 2, 4, and 8 byte quantities
* **sizes of C numberic data types:** #bytes allocated varies with machine and compiler



* + note that pointers use machine's full word size

★**ADDRESSING AND BYTE ORDERING:**

* **big endian:** least significant bute has highest address
  + sun, powerpc macintosh

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Address** | **0x100** | **0x101** | **0x102** | **0x103** |
| **Value** | **01** | **23** | **45** | **67** |

* + ***example:*** 4-byte representation of 0x01234567
* **little endian:** least significant byte has lowest address
  + alpha, intel macintosh, pc

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Address** | **0x100** | **0x101** | **0x102** | **0x103** |
| **Value** | **67** | **45** | **23** | **01** |

* + ***example:*** 4-byte representation of 0x01234567
* where byte order becomes important:
  + binary data are connumictated over a network between different machines (could be using different byte ordering)
  + looking at byte sequence representing integer data
  + programs are written that circumvent the normal type system (casting)
* sdf

★**REPRESENTING STRINGS:**

* here

★**REPRESENTING CODE:**

* here