Machine Language Level Organization

Major components:
- memory
- central processing unit (CPU)
- registers
- the fetch/execute cycle
  (the *hardware process*)

Memory

Dynamic random access memory (DRAM)
- slow and cheap (relative to registers and cache)
- each location in memory has a unique address
- the smallest addressable unit of memory may be a byte, or some larger quantity
- cache memory is an optimization

Data and instructions are both stored in memory
- *stored program computer*
Central Processing Unit

Control
- decodes instructions and manages CPU’s internal resources

Registers
- general-purpose registers available to user processes
- special-purpose registers directly managed in fetch/execute cycle
- other registers may be reserved for use of operating system
- very fast and expensive (relative to memory)
- hold all operands and results of arithmetic instructions (on RISC systems)
- save bits in instruction representation

Data path or arithmetic/logic unit (ALU)
- operates on data

Stored Program Concept

Instructions are collections of bits
Programs are stored in memory, to be read or written just like data

Memory for data, programs, compilers, editors, etc.

Fetch & Execute Cycle
Instructions are fetched and put into a special register
Bits in the register “control” the subsequent actions
Fetch the “next” instruction and continue
Of course, on most systems several programs will be stored in memory at any given time.

On most contemporary systems instructions of only one of those will be executed at any given instant.

The operating system will rapidly switch among the eligible processes, producing the illusion that several programs are executing at the same time.

Sometimes called the hardware process… executes continuously.

Steps:
- fetch an instruction from memory to the instruction register
- increment the program counter register (by the instruction length)
- decode the instruction (in the control unit)
- fetch operands, if any, usually from registers
- perform the operation (in the data path); this may modify the PC register
- store the results, usually to registers
1) fetch an instruction from memory to the instruction register
2) increment the program counter register (by the instruction length)
3) decode the instruction (in the control unit)
4) fetch operands, if any, usually from registers
5) perform the operation (in the data path); this may modify the PC register
6) store the results, usually to registers

But, how is all of this driven?

Machine language:
- registers store collections of bits
- all data and instructions must be encoded as collections of bits (binary)
- bits are represented as electrical charges (more or less)
- control logic and arithmetic operations are implemented as circuits, which are driven by the movement of electrical charges
- so, the instructions directly manipulate the underlying hardware (cool, huh?)

The collection of all valid binary instructions is known as the machine language.
- what’s valid depends on the design of the hardware, especially the control circuitry
- must be formally specified
- machine language is not human-friendly
### Assembly Language

More human-friendly syntax:
- expressed in text, not in binary
- instructions are identified by (more-or-less) mnemonic names
- instruction operands may include registers, memory locations, or...

Aspects of assembly language:
- unlike high-level languages, each instruction is extremely simple, so assembly language programs are much longer than corresponding high-level language programs
- assembly language must be translated into machine language in order to be executed
- assembly language is not usually any more portable across different hardware platforms that is machine language
- most assembly languages are quite similar... from a certain point of view

### Typical Program Translation

We'll be a little concerned with what happens here.

We'll be more concerned with what happens here.

For the most part, we'll be concerned with writing assembly language, how it's translated into machine language, and how the machine language is executed.
The process of loading is nontrivial:
- the machine language program won't contain the right values for addresses because there's usually no way to know just where a program will be loaded in memory
- either the loader must change the addresses to the correct physical values…
- or there must be some efficient way to manage the substitution/calculation of the correct addresses as the instructions are executed