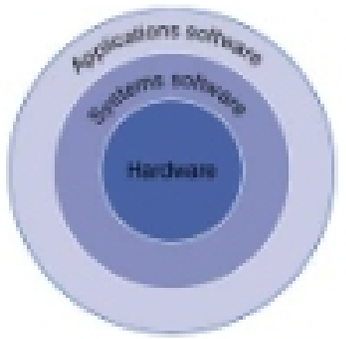
Lecture 8: Program and Compilation

- Program Memory Layout
- From C to MIPS
- C v.s. Java

Below Your Program

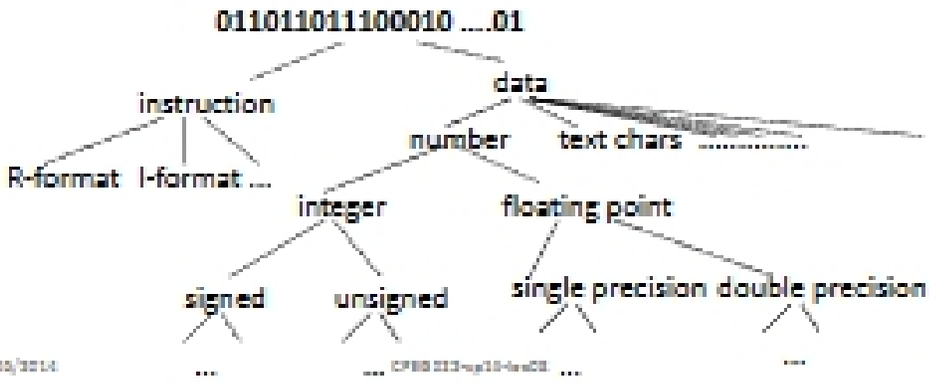


- Application software
  - Written in high-level language
- System software
  - Compiler: translates HLL code to machine code
  - Operating System: service code
    - Handling input/output
    - Managing memory and storage
    - Scheduling tasks & sharing resources
- Hardware
  - Processor, memory, I/O controllers

Code v.s. Data

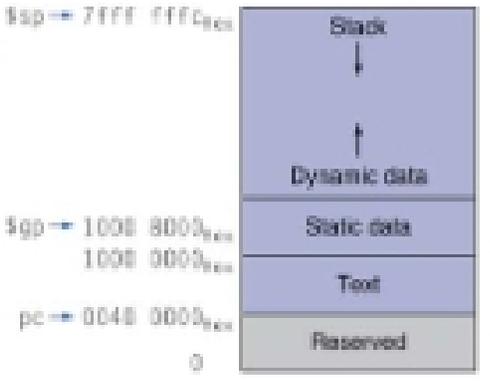
- Instructions are represented as numbers and, as such, are indistinguishable from data
- Programs are stored in alterable memory (that can be read or written to just like data)

Bits everywhere... What do they mean?



Memory Layout for a Program

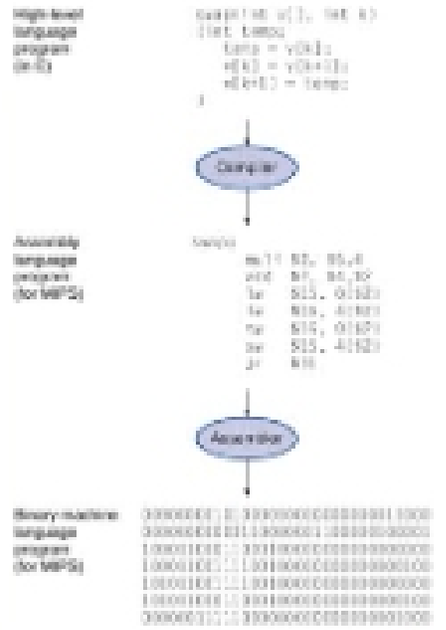
- Text: program code
- Static data: global variables
  - e.g., static variables in C, constant arrays and strings
  - $gp initialized to address allowing ±offsets into this segment
- Dynamic data: heap
  - E.g., malloc in C, new in Java
- Stack: automatic storage



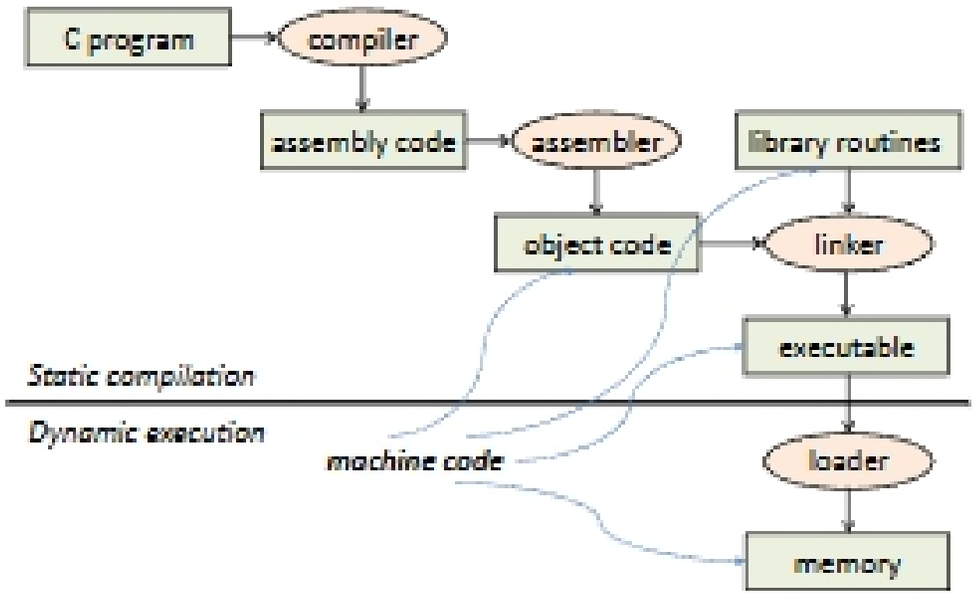
Now we know how a program looks like.
But how is it generated?

Levels of Program Code

- High-level language
  - Level of abstraction closer to problem domain
  - Provides for productivity and portability
- Assembly language
  - Textual representation of instructions
- Hardware representation
  - Binary digits (bits)
  - Encoded instructions and data



The C Code Translation Hierarchy



Assembly vs. Machine Language

- Assembly provides convenient symbolic representation
  - Much easier than writing down numbers
- Machine language is the underlying reality
  - Used by the processor at run time
- Assembly can provide 'pseudo-instructions'
  - e.g., "move $t0, $t1" exists only in assembly would be implemented using "add $t0,$t1,$zero"
- When considering performance you should count real instructions

Producing an Object Module

- Assembler (or compiler) translates program into machine instructions
- It provides information for building a complete program from the pieces
  - Header: described contents of object module
  - Text segment: translated instructions
  - Static data segment: data allocated for the life of the program
  - Relocation info: for contents that depend on absolute location of loaded program
  - Symbol table: global definitions and external refs
  - Debug info: for associating with source code

Linking Object Modules

- Produces an executable image
  - Merges segments
  - Resolve labels (determine their addresses)
  - Patch location-dependent and external refs
- Could leave location dependencies for fixing by a relocating loader
  - But with virtual memory, no need to do this
  - Program can be loaded into absolute location in virtual memory space

Loading a Program

- Load from image file on disk into memory
  1. Read header to determine segment sizes
  2. Create virtual address space
  3. Copy text and initialized data into memory
  4. Set up arguments on stack
  5. Initialize registers (including $sp, $fp, $gp)
  6. Jump to startup routine
     - Copies arguments to $a0, … and calls main
     - When main returns, do exit syscall

Dynamic Linking

- Only link/load library procedure when it is called
  - Requires procedure code to be relocatable
  - Avoids image bloat caused by static linking of all (transitively) referenced libraries
  - Automatically picks up new library versions
  - May go through a number of indexing at run time