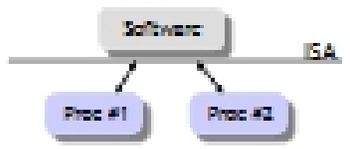
Lecture 8: Running a Program

(Compiling, Assembling, Linking, Loading)

(CPEG323: Intro. to Computer System Engineering)

Instruction Set Architecture (ISA)

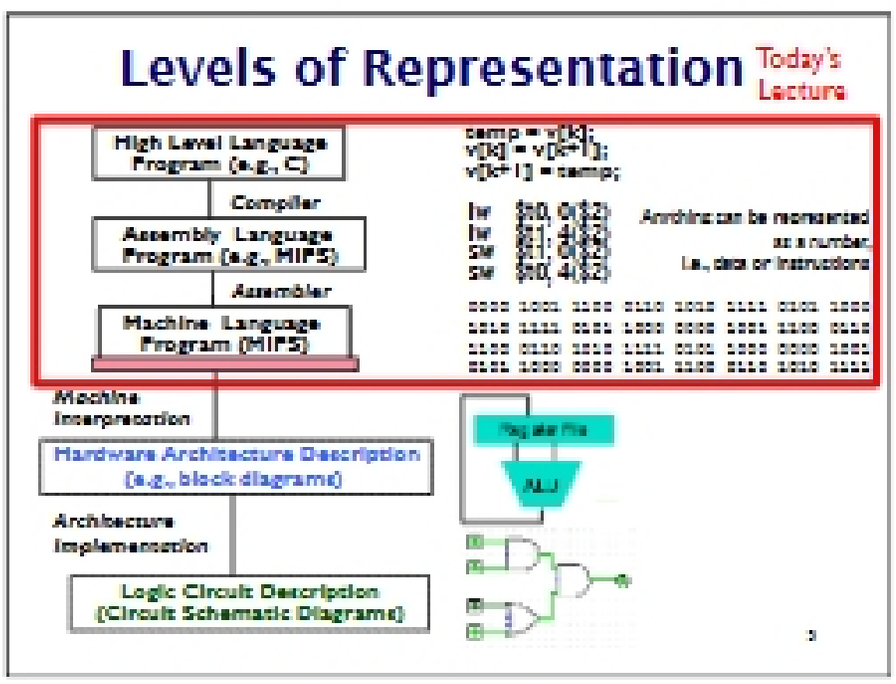
- The ISA is an abstraction layer between hardware and software
  - Software does not need to know how the processor is implemented
  - Processors that implement the same ISA appear equivalent



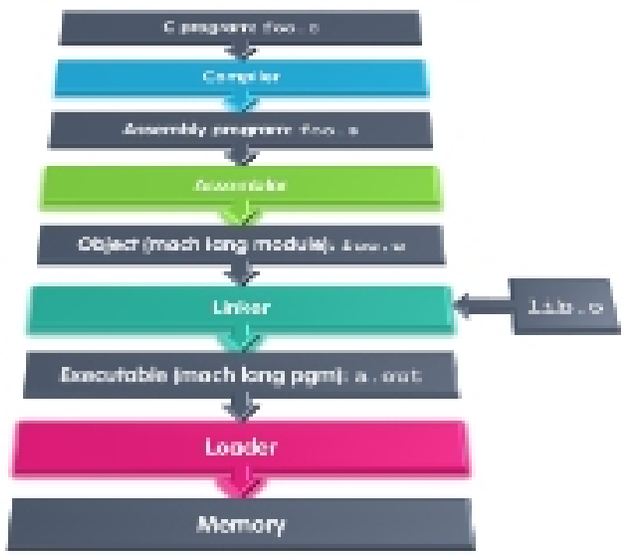
- An ISA enables processor innovation without changing software
  - This is how Intel has made billions of dollars
- Before ISAs, software was re-written/re-compiled for each new machine

Levels of Representation

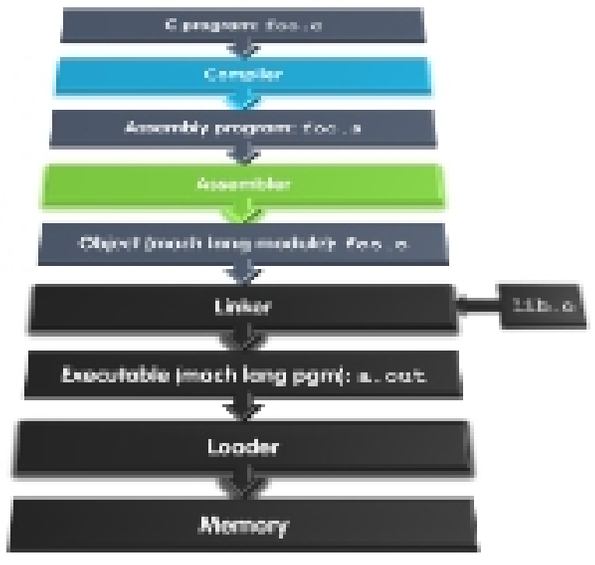
Today's Lecture



Steps to Starting a Program



Running a program



Compiler

- Input: High-Level Language Code (e.g., C such as foo.c)
- Output: Assembly Language Code (e.g., foo.s for MIPS)
- Note: Output *may* contain pseudoinstructions
- Pseudoinstructions: instructions that assembler understands but not in machine (last lecture) For example:
- mov $s1,$s2 ⇒ or $s1,$s2,$zero

Assembler

- Input: Assembly Language Code (e.g., foo.s for MIPS)
- Output: Object Code (e.g., foo.o for MIPS)
  - Reads and uses directives
  - Replace Pseudoinstructions
  - Produce Machine Language
  - Create Object File

Assembler Directives

- Give directions to assembler, but do not produce machine instructions

  .text: Subsequent items put in user text segment (machine code)

  .data: Subsequent items put in user data segment (binary rep of data in source file)

  .globl sym: declares global symbol and can be referenced from other files

  .asciiz str: Store the string str in memory and null-terminate it

  .word w1…wn: Store the n 32-bit quantities in successive memory words

Pseudoinstruction Replacement

- Problem:
  - when breaking up a pseudoinstruction, the assembler may need to use an extra register.
  - If it uses any regular register, it will overwrite whatever the program has put into it.

Pseudo:	Real:
ble $t0,100,loop	slti $at,$t0,101 bne $at,$0,loop

- Solution:
  - Reserve a register ($1, called $at for "assembler temporary")

Producing Machine Language (1/3)

- Simple Case
  - Arithmetic, Logical, Shifts, and so on.
  - All necessary info is within the instruction already.
- What about Branches?
  - PC-Relative
  - So once pseudoinstructions are replaced by real ones, we know by how many instructions to branch.
- So these can be handled.

Producing Machine Language (1/3)

- "Forward Reference" problem
  - Branch instructions can refer to labels that are "forward" in the program:

```
        or   $v0, $0,  $0
L1:     slt  $t0, $0,  $a1
        beq  $t0, $0,  L2
        addi $a1, $a1, -1
        j    L1
L2:     add  $t1, $a0, $a1
```

  - Solved by taking 2 passes over the program.
    - First pass remembers position of labels
    - Second pass uses label positions to generate code

Producing Machine Language (3/3)

- What about jumps (j and jal)?
  - Jumps require absolute address.
  - So, forward or not, still can't generate machine instructions without knowing the position of instructions in memory.
- What about references to data?
  - la gets broken up into lui and ori
  - These will require the full 32-bit address of the data.
- These can't be determined yet, so we create two tables…

Symbol Table

- List of "items" in this file that may be used by other files.
- What are they?
  - Labels: function calling
  - Data: anything in the .data section; variables which may be accessed across files

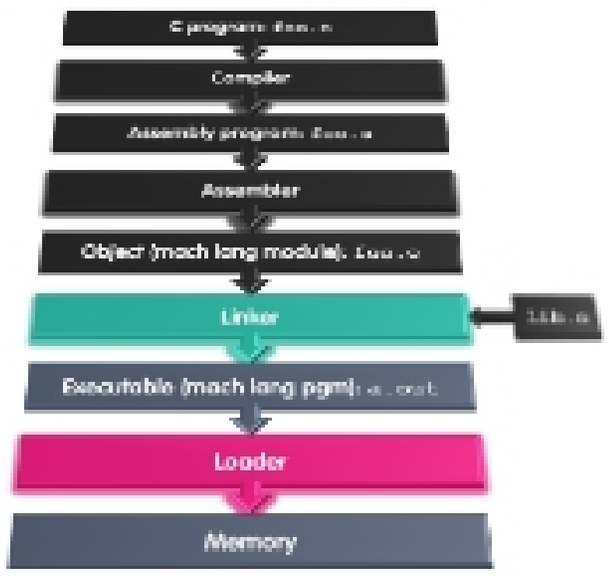
Relocation Table

- List of "items" this file needs the address of later.
- What are they?
  - Any label jumped to: j or jal
    - internal
    - external (including lib files)
  - Any piece of data that references an address
    - such as the la instruction

Object File Format

- object file header: size and position of the other pieces of the object file
- text segment: the machine code
- data segment: binary representation of the data in the source file
- relocation information: identifies lines of code that need to be "handled"
- symbol table: list of this file's labels and data that can be referenced
- debugging information

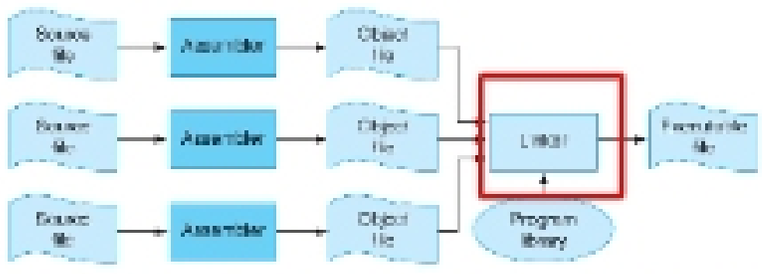
Where Are We Now?



Linker

- Input: Object Code files, information tables (e.g., foo.o,libc.o for MIPS)
- Output: Executable Code (e.g., a.out for MIPS)
- Combines several object (.o) files into a single executable ("linking")
- Enable separate compilation of files
  - Changes to one file do not require recompilation of whole program

The purpose of a linker



- The linker is a program that takes one or more object files and assembles them into a single executable program.
- The linker resolves references to undefined symbols by finding out which other object defines the symbol in question, and replaces placeholders with the symbol's address.