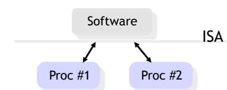
Lecture 8: Running a Program

(CPEG323: Intro. to Computer System Engineering)

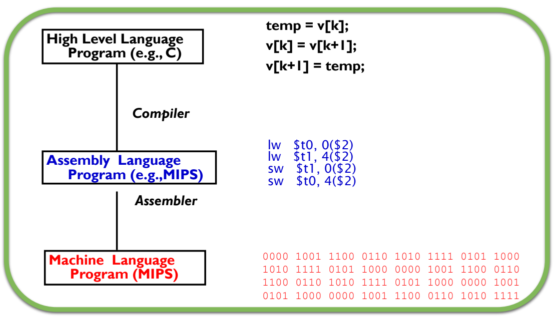
Instruction Set Architecture (ISA)

- 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 appears equivalent

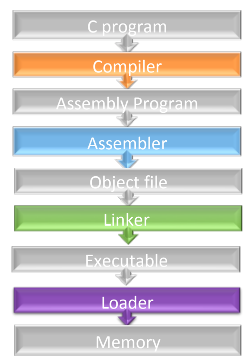


- An ISA enables processor innovation without changing software.

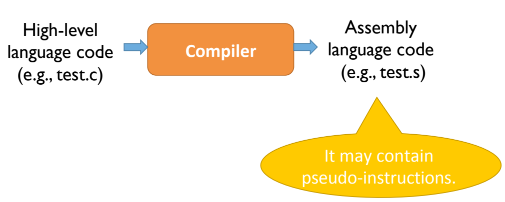
Levels of Program Code



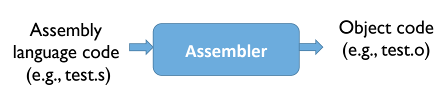
Steps to Starting a Program



Compiling



Assembling



- Reads and uses directives
- Replace Pseudoinstructions
- Produce Machine Language
- Create Object File

Assembler (1): Read and Use 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

Assembler (2): Pseudo-Instruction Replacement

- Pseudoinstructions:
  - MIPS "instructions" that are convenient for an assembly programmer to use
  - Get translated by the assembler into real instructions

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

- When breaking up a pseudoinstruction, the assembler may need to use an extra register.
- Reserve a register ($1, called $at for "assembler temporary")

Assembler (3): Producing Machine Language

- It is easy when all necessary information is within the instruction.
  - E.g., add $t1, $t2, $t3
- What about branches?
- What about jumps?

Assembler (3): Producing Machine Language - Branches

- Branches require a relative address.
- So once pseudo-instructions are replaced by real ones, we know by how many instructions to branch.

```
     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.
  - In the first pass, expand pseudo instruction and remember position of labels
  - In the second pass, uses label positions to generate relative addresses for branch

Assembler (3): Producing Machine Language - Jumps

- Jumps require **absolute address** of instructions.
- We still can't generate machine instruction without knowing the position of instructions in memory.
- So, we create two tables
  - **Symbol table**, which stores a list of items in this file that may be used by other files.
    - Labels: function calling
    - Data: anything in the .data section; variables which may be accessed across files
  - **Relocation table**, which stores a list of items this file needs the address of later.
    - Any label jumped to: j or jal
    - Any piece of data that references an address (e.g., la)

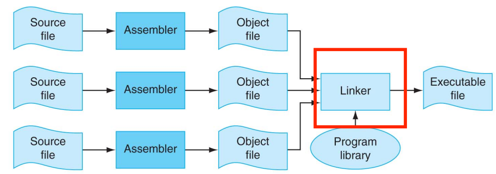
Assembler (4): Create Object Files

- 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

Linking

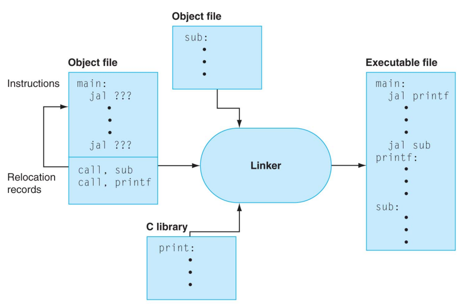
- Combines several object (.o) files into a single executable
  - Take text segment from each .o file and put them together
  - Take data segment from each .o file, put them together and put to the end of text segments.
  - Resolve references, i.e., fill in all absolute addresses
    - Go through relocation table; handle each entry
- 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.

Linker stitches file together



Linking - Resolving References

- Linker *assumes* first word of first text segment is at address 0x00000000.
- Linker knows:
  - length of each text and data segment
  - ordering of text and data segments
- Linker calculates:
  - absolute address of each label to be jumped to (internal or external) and each piece of data being referenced
- To resolve references:
  - search for reference (data or label) in all "user" symbol tables
  - if not found, search library files (for example, for printf)
  - once absolute address is determined, fill in the machine code appropriately
- Output of linker: executable file containing text and data (plus header)

Loading

- Executable files are stored on disk.
- When one is run, loader's job is to load it into memory and start it running.
- In reality, loader is the operating system (OS), loading is one of the OS tasks

Loading – 5 steps

1. Allocates memory for the program's execution.
2. Copies the text and data segments from the executable into memory.
3. Copies program arguments (*e.g.*, command line arguments) onto the stack.
4. Initializes registers: sets $sp to point to top of stack, clears the rest.
5. Jumps to start routine, which:
   1) copies main's arguments off of the stack
   2) jumps to main.

Summary: Running a Program

- **Compiler** converts a single HLL file into a single assembly language file.
- **Assembler** removes pseudo-instructions, converts what it can to machine language, and creates a checklist for the linker (relocation table). A .s file becomes a .o file.
  - Does 2 passes to resolve addresses, handling internal forward references
- **Linker** combines several .o files and resolves absolute addresses.
  - Enables separate compilation, libraries that need not be compiled, and resolves remaining addresses
- **Loader** loads executable into memory and begins execution.

Interpretation vs. Translation

- How do we run a program written in a source language?
  - Interpreter: Directly executes a program in the source language
  - Translator: Converts a program from the source language to an equivalent program in another language

Compiler vs. Interpreter Advantages

Compilation:
- Faster Execution
- Single file to execute
- Compiler can do better diagnosis of syntax and semantic errors, since it has more info than an interpreter (Interpreter only sees one line at a time)
- Can find syntax errors *before* run program
- Compiler can optimize code

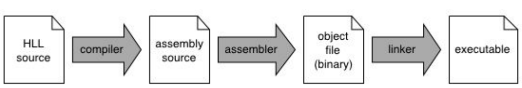
Interpreter:
- Easier to debug program
- Faster development time

Compiler vs. Interpreter Disadvantages

Compilation:
- Harder to debug program
- Takes longer to change source code, recompile, and relink

Interpreter:
- Slower execution times
- No optimization
- Need all of source code available
- Source code larger than executable for large systems
- Interpreter must remain installed while the program is interpreted

The compilation process



- To produce assembly code: `gcc –S test.c`
  - produces `test.s`
- To produce object code: `gcc –c test.c`
  - produces `test.o`
- To produce executable code: `gcc test.c`
  - produces `a.out`

Reading

- 5th edition: 2.12-2.13, A.1-A.7