System I

RISC-V Assembly

Bo Feng, Lei Wu, Rui Chang

Zhejiang University

Disclaimer

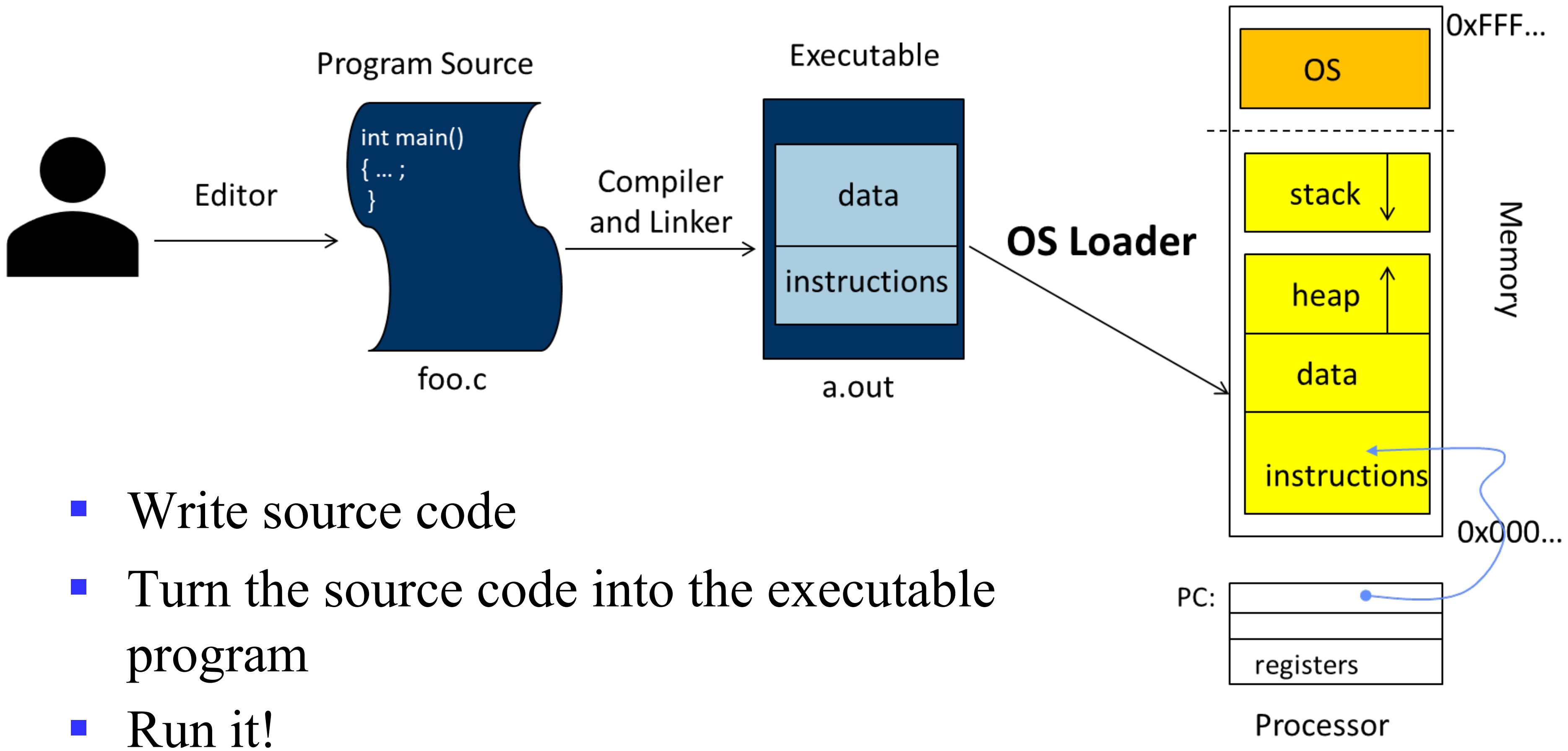
- **Many images and resources used in this lecture are collected from the Internet, and they are used only for the educational purpose. The copyright belong to the original owners, respectively.**

- **Part of slides credit to**
  - **David A. Patterson and John L. Hennessy. Computer Organization and Design RISC-V Edition: The Hardware Software Interface, 1st Edition.**
  - **John L. Hennessy and David A. Patterson. Computer Architecture: A Quantitative Approach, 6th Edition.**
  - **Andrew Waterman and David A. Patterson. The RISC-V Reader: An Open Architecture Atlas.**
  - **CSCE 513, Prof. Yonghong Yan @ University of North Carolina at Charlotte**
  - **CENG3420, Bei Yu @ The Chinese University of Hong Kong**
  - **CS 3410, Prof. Hakim Weatherspoon @ Cornell University**
  - **CS 162, Prof. Sam Kumar @ UC Berkeley**
  - **CSc 453, Prof. Saumya Debray @ University of Arizona**

Overview

- RISC-V ISA
- RISC-V Assembly Language
  - Some basic concepts
  - From source code to a running program

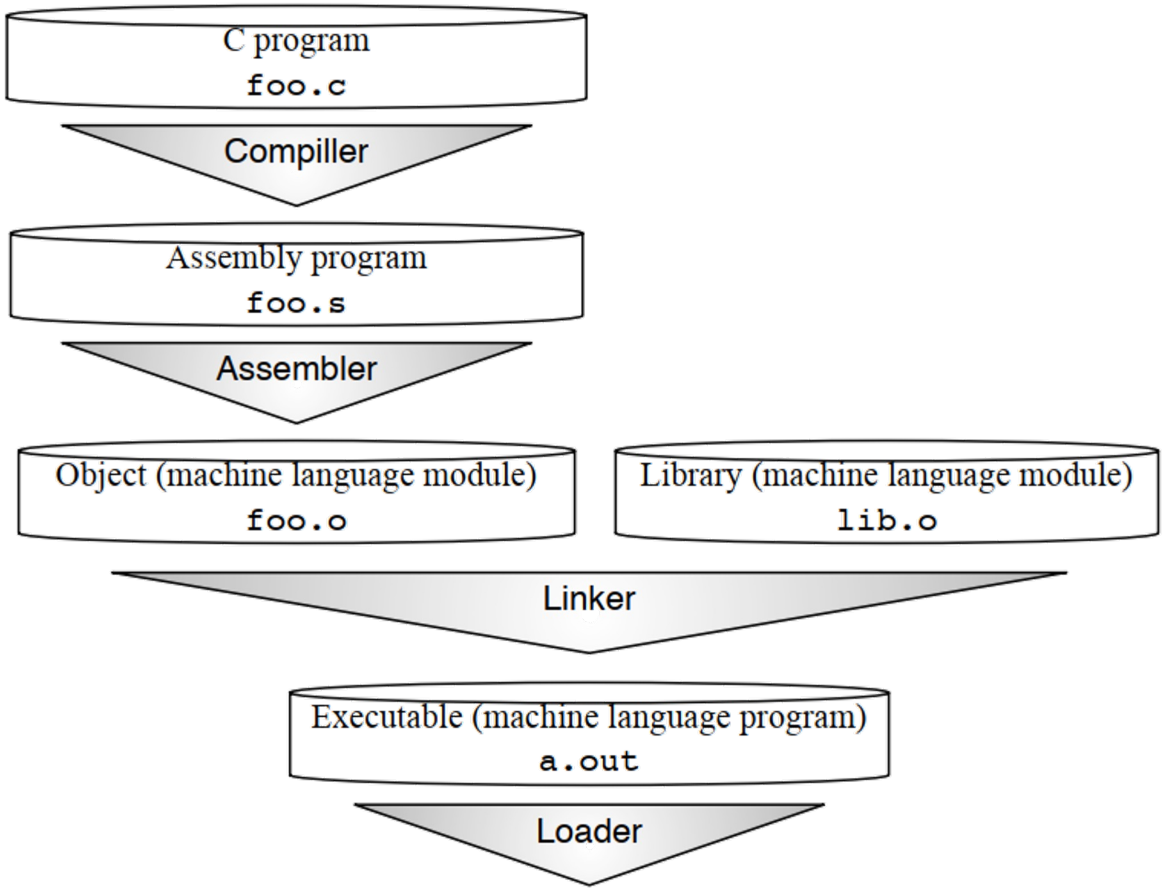
From Source Code to A Running Program



- Write source code
- Turn the source code into the executable program
- Run it!

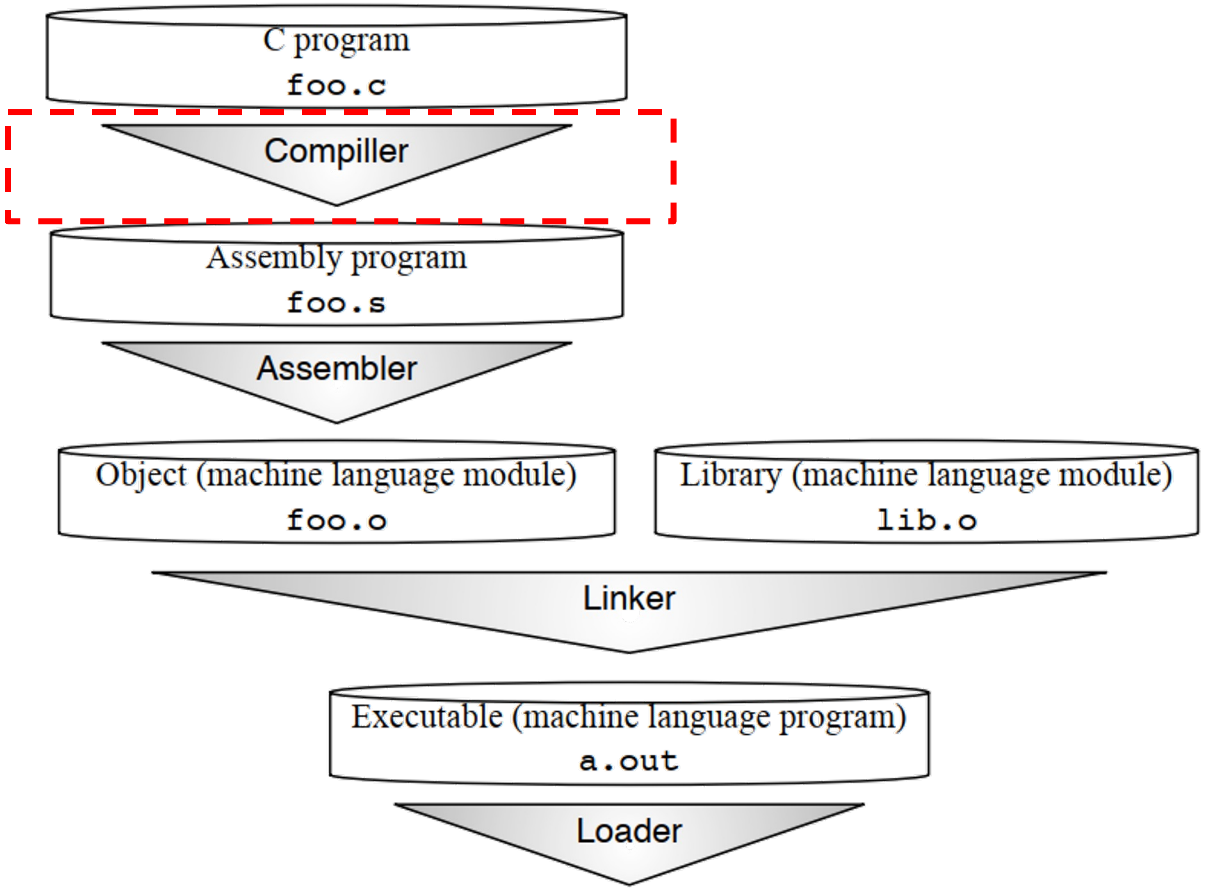
From C Source Code to A Running Program

- Steps of translation from C source code to a running program. These are the logical steps, although some steps are combined to accelerate translation.


C program
foo.c
Compiller
Assembly program
foo.s
Assembler
Object (machine language module)
foo.o
Library (machine language module)
lib.o
Linker
Executable (machine language program)
a.out
Loader

Compiler: *.c -> *.s

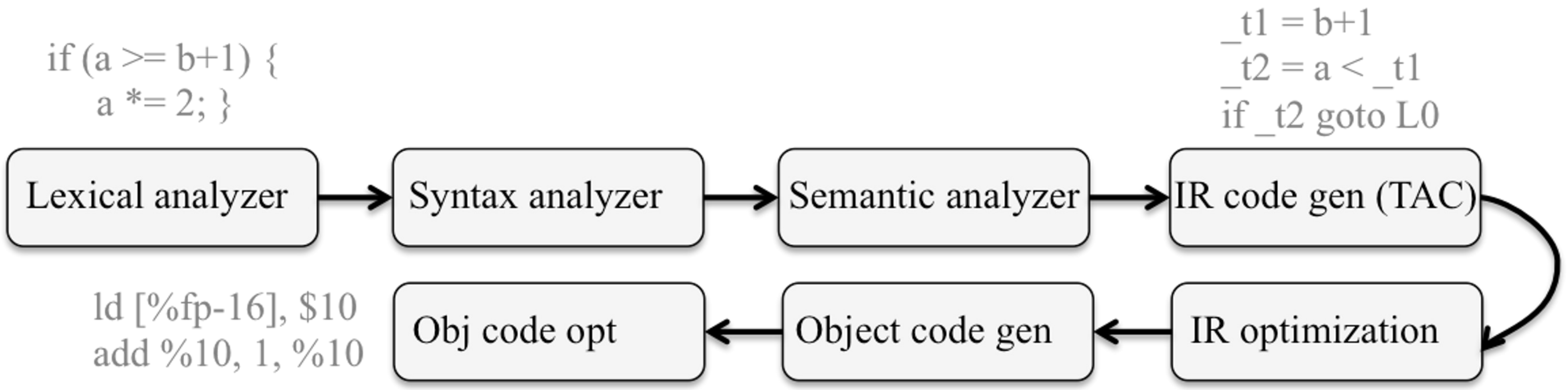
- Steps of translation from C source code to a running program. These are the logical steps, although some steps are combined to accelerate translation.


C program
foo.c
Compiller
Assembly program
foo.s
Assembler
Object (machine language module)
foo.o
Library (machine language module)
lib.o
Linker
Executable (machine language program)
a.out
Loader

Compiler

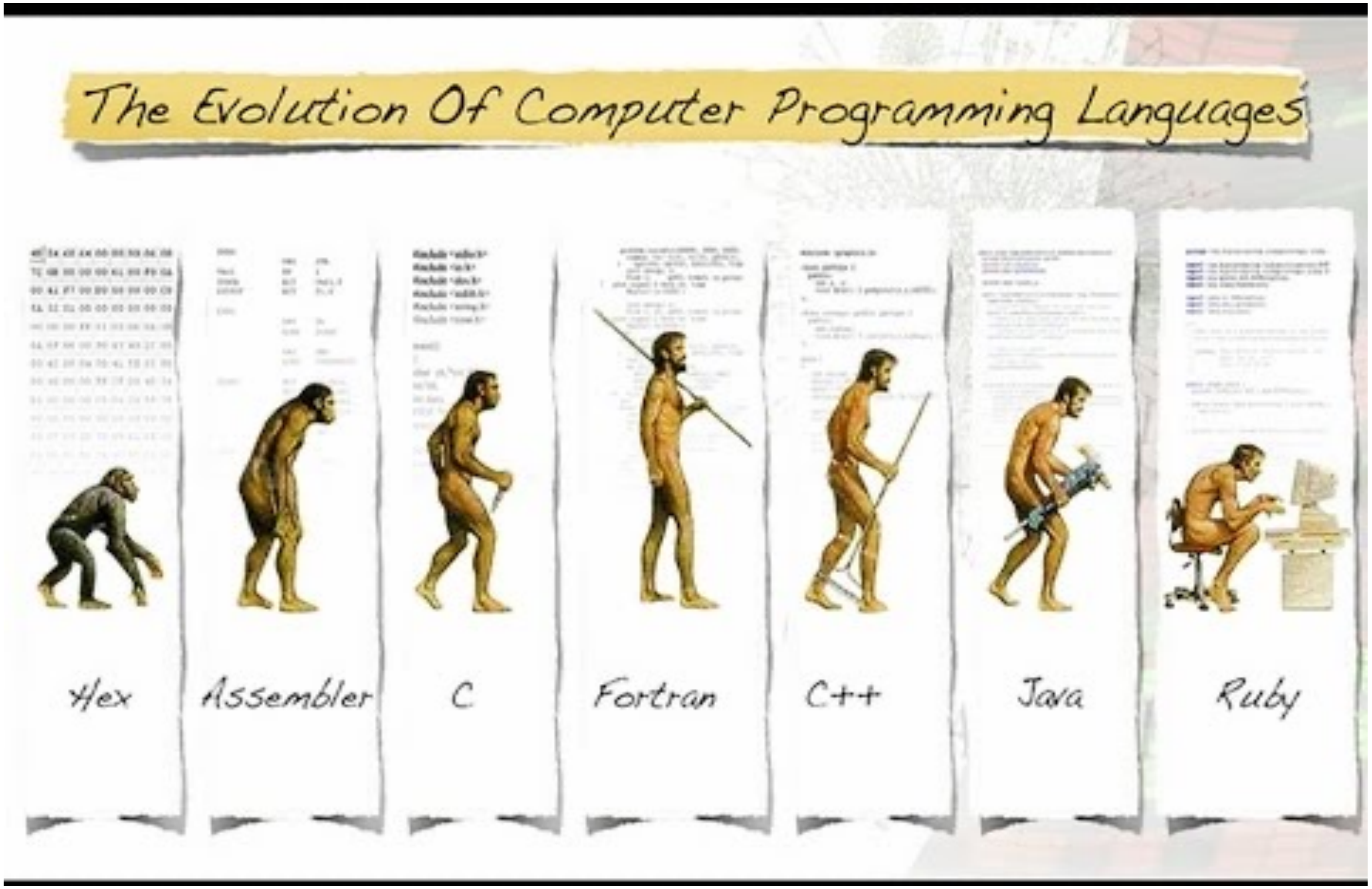
- Preprocessor (*.c -> *.i)
  - Expands all macro definitions and include statements (and anything else starting with a #) and passes the result to the actual compiler.

- Compiler (*.i -> *.s)



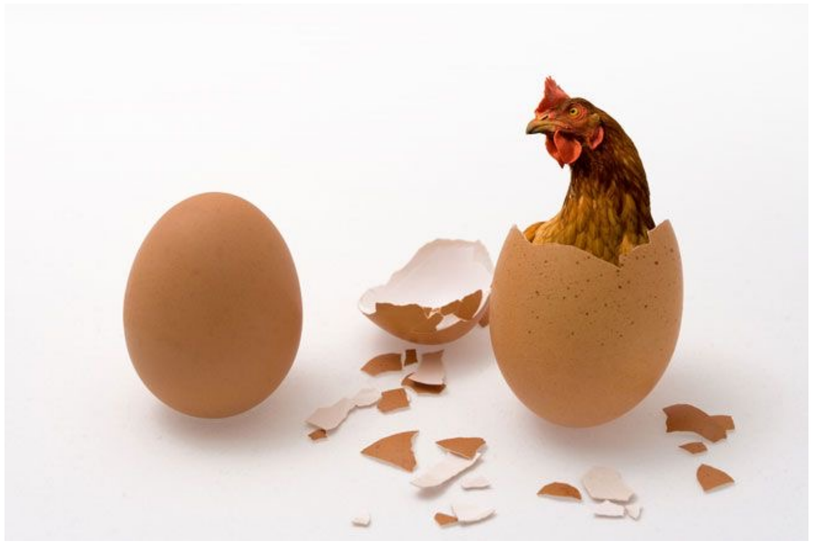
https://people.cs.pitt.edu/~xianeizhang/notes/Linking.html

Evolution of Programming Languages



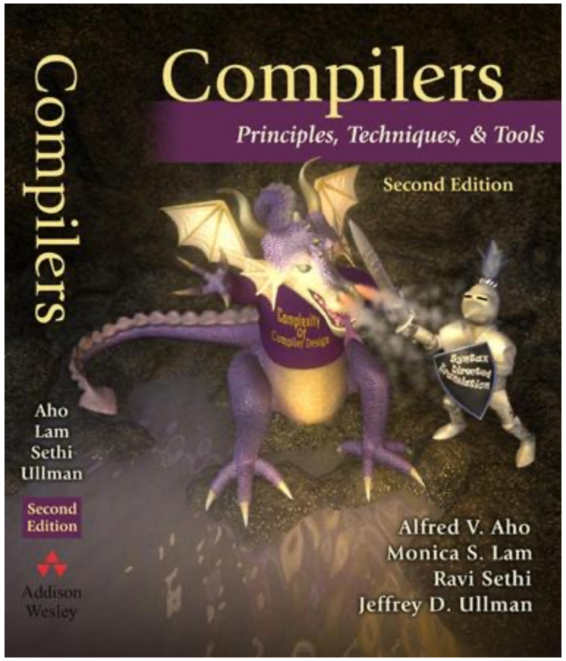
Bootstraping: Self-Compiling Compilers

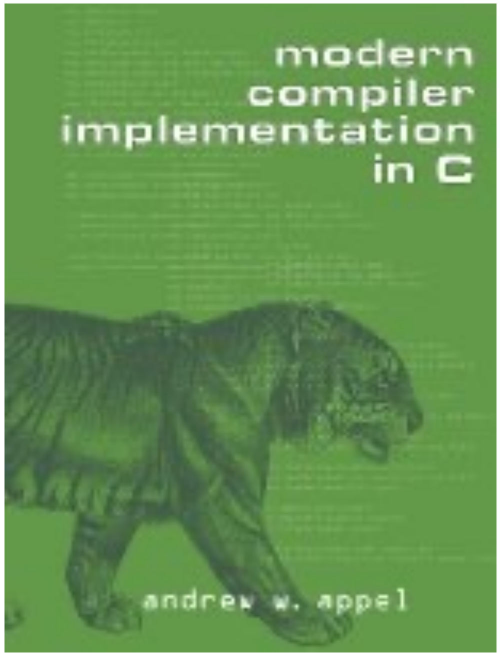
- A compiler (or assembler) written in the source programming language that it intends to compile.
- An initial core version of the compiler (the bootstrap compiler) is generated in a different language (which could be assembly language); successive expanded versions of the compiler are developed using this minimal subset of the language.
- The problem of compiling a self-compiling compiler has been called the chicken-or-egg problem in compiler design, and bootstrapping is a solution to this problem

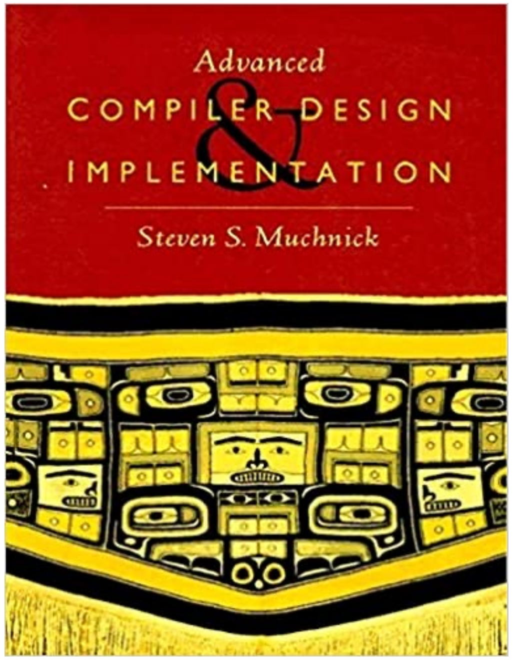


Reflections on Trusting Trust

NOT Be Detailed in This Class…



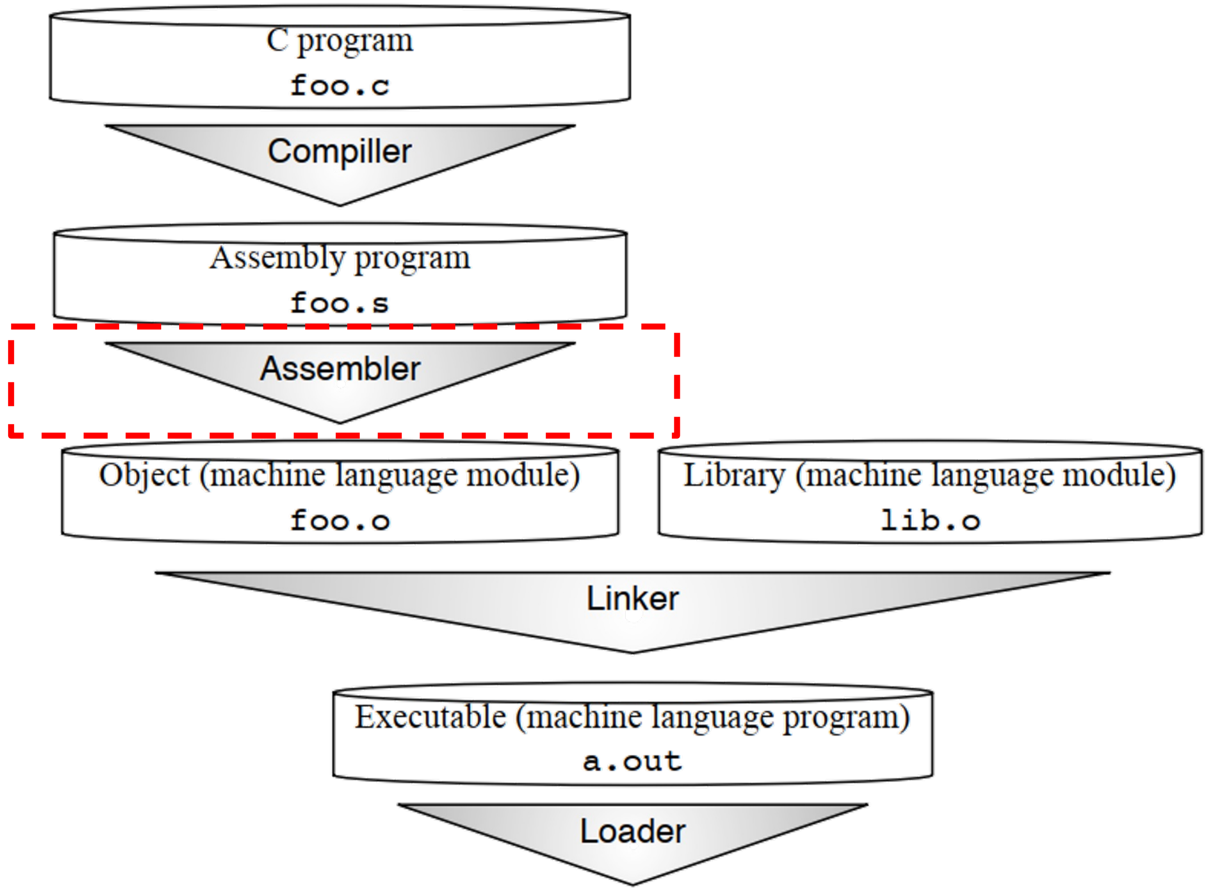




Because You Deserve More!

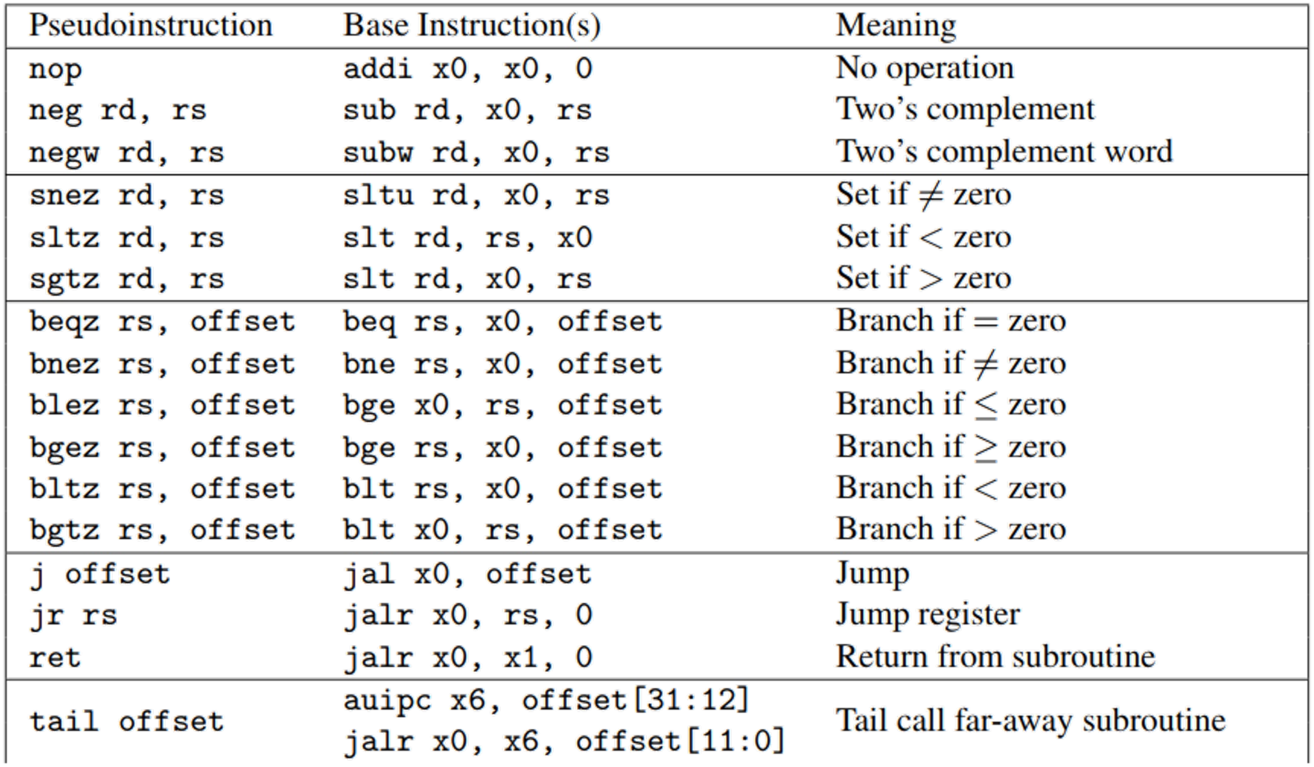
Assembler: *.s -> *.o

- Steps of translation from C source code to a running program. These are the logical steps, although some steps are combined to accelerate translation.


C program
foo.c
Compiller
Assembly program
foo.s
Assembler
Object (machine language module)
foo.o
Library (machine language module)
lib.o
Linker
Executable (machine language program)
a.out
Loader

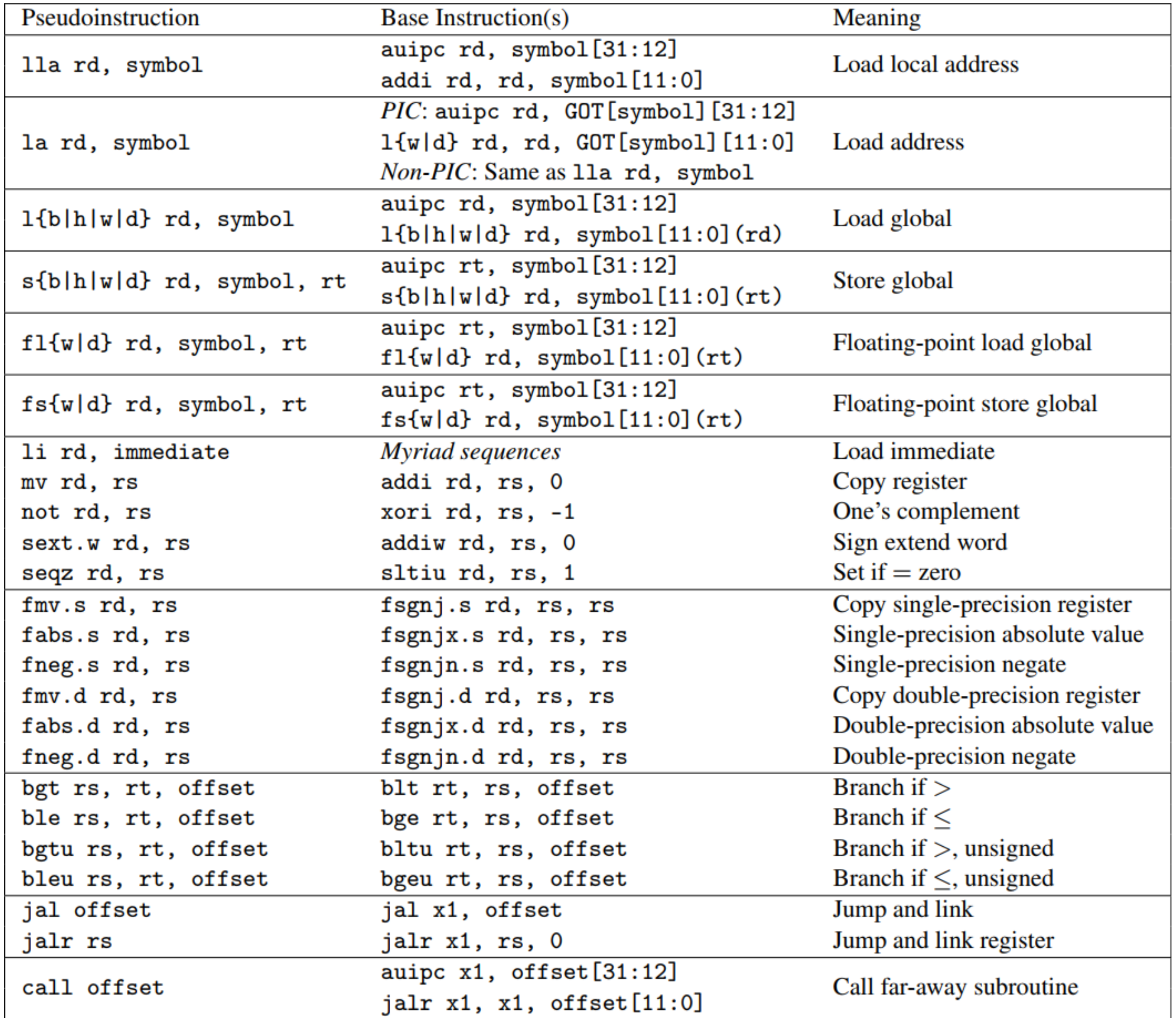
Assembler

- Not simply to produce object code from the instructions that the processor understands, but to extend them to include operations useful for the assembly language programmer or the compiler writer. This category, based on clever configurations of regular instructions, is called ***pseudo-instructions***.



Pseudoinstruction	Base Instruction(s)	Meaning
`nop`	`addi x0, x0, 0`	No operation
`neg rd, rs`	`sub rd, x0, rs`	Two's complement
`negw rd, rs`	`subw rd, x0, rs`	Two's complement word
`snez rd, rs`	`sltu rd, x0, rs`	Set if $\neq$ zero
`sltz rd, rs`	`slt rd, rs, x0`	Set if $<$ zero
`sgtz rd, rs`	`slt rd, x0, rs`	Set if $>$ zero
`beqz rs, offset`	`beq rs, x0, offset`	Branch if $=$ zero
`bnez rs, offset`	`bne rs, x0, offset`	Branch if $\neq$ zero
`blez rs, offset`	`bge x0, rs, offset`	Branch if $\leq$ zero
`bgez rs, offset`	`bge rs, x0, offset`	Branch if $\geq$ zero
`bltz rs, offset`	`blt rs, x0, offset`	Branch if $<$ zero
`bgtz rs, offset`	`blt x0, rs, offset`	Branch if $>$ zero
`j offset`	`jal x0, offset`	Jump
`jr rs`	`jalr x0, rs, 0`	Jump register
`ret`	`jalr x0, x1, 0`	Return from subroutine
`tail offset`	`auipc x6, offset[31:12]` `jalr x0, x6, offset[11:0]`	Tail call far-away subroutine

Assembler



Pseudoinstruction	Base Instruction(s)	Meaning

`lla rd, symbol`	`auipc rd, symbol[31:12]`
	`addi rd, rd, symbol[11:0]`
		Load local address

`la rd, symbol`	*PIC*: `auipc rd, GOT[symbol][31:12]` `l{w|d} rd, rd, GOT[symbol][11:0]` *Non-PIC*: Same as `lla rd, symbol`	Load address

`l{b|h|w|d} rd, symbol`	`auipc rd, symbol[31:12]`
	`l{b|h|w|d} rd, symbol[11:0](rd)`
		Load global

`s{b|h|w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]`
	`s{b|h|w|d} rd, symbol[11:0](rt)`
		Store global

`fl{w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]`
	`fl{w|d} rd, symbol[11:0](rt)`
		Floating-point load global

`fs{w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]`
	`fs{w|d} rd, symbol[11:0](rt)`
		Floating-point store global

`li rd, immediate`	*Myriad sequences*	Load immediate
`mv rd, rs`	`addi rd, rs, 0`	Copy register
`not rd, rs`	`xori rd, rs, -1`	One's complement
`sext.w rd, rs`	`addiw rd, rs, 0`	Sign extend word
`seqz rd, rs`	`sltiu rd, rs, 1`	Set if = zero
`fmv.s rd, rs`	`fsgnj.s rd, rs, rs`	Copy single-precision register
`fabs.s rd, rs`	`fsgnjx.s rd, rs, rs`	Single-precision absolute value
`fneg.s rd, rs`	`fsgnjn.s rd, rs, rs`	Single-precision negate
`fmv.d rd, rs`	`fsgnj.d rd, rs, rs`	Copy double-precision register
`fabs.d rd, rs`	`fsgnjx.d rd, rs, rs`	Double-precision absolute value
`fneg.d rd, rs`	`fsgnjn.d rd, rs, rs`	Double-precision negate
`bgt rs, rt, offset`	`blt rt, rs, offset`	Branch if $>$
`ble rs, rt, offset`	`bge rt, rs, offset`	Branch if $\leq$
`bgtu rs, rt, offset`	`bltu rt, rs, offset`	Branch if $>$, unsigned
`bleu rs, rt, offset`	`bgeu rt, rs, offset`	Branch if $\leq$, unsigned
`jal offset`	`jal x1, offset`	Jump and link
`jalr rs`	`jalr x1, rs, 0`	Jump and link register

`call offset`	`auipc x1, offset[31:12]`
	`jalr x1, x1, offset[11:0]`
		Call far-away subroutine

“Hello World” in C

```
#include <stdio.h>
int main()
{
   printf("Hello, %s\n", "world");
   return 0;
}
```

Input: in Assembly Language (*.s)

```
    .text                         # Directive: enter text section
    .align 2                      # Directive: align code to 2^2 bytes
    .globl main                   # Directive: declare global symbol main
  main:                           # label for start of main
    addi sp,sp,-16                # allocate stack frame
    sw   ra,12(sp)                # save return address
    lui  a0,%hi(string1)          # compute address of
    addi a0,a0,%lo(string1)       #   string1
    lui  a1,%hi(string2)          # compute address of
    addi a1,a1,%lo(string2)       #   string2
    call printf                   # call function printf
    lw   ra,12(sp)                # restore return address
    addi sp,sp,16                 # deallocate stack frame
    li   a0,0                     # load return value 0
    ret                           # return
    .section .rodata              # Directive: enter read-only data section
    .balign 4                     # Directive: align data section to 4 bytes
  string1:                        # label for first string
    .string "Hello, %s!\n"        # Directive: null-terminated string
  string2:                        # label for second string
    .string "world"               # Directive: null-terminated string
```

Assembler Directives

- `.text`—Enter text section.
- `.align 2`—Align following code to 2^2 bytes.
- `.globl main`—Declare global symbol “main”.
- `.section .rodata`—Enter read-only data section.
- `.balign 4`—Align data section to 4 bytes.
- `.string ‘‘Hello, %s!\n’’`—Create this null-terminated string.
- `.string ‘‘world’’`—Create this null-terminated string.

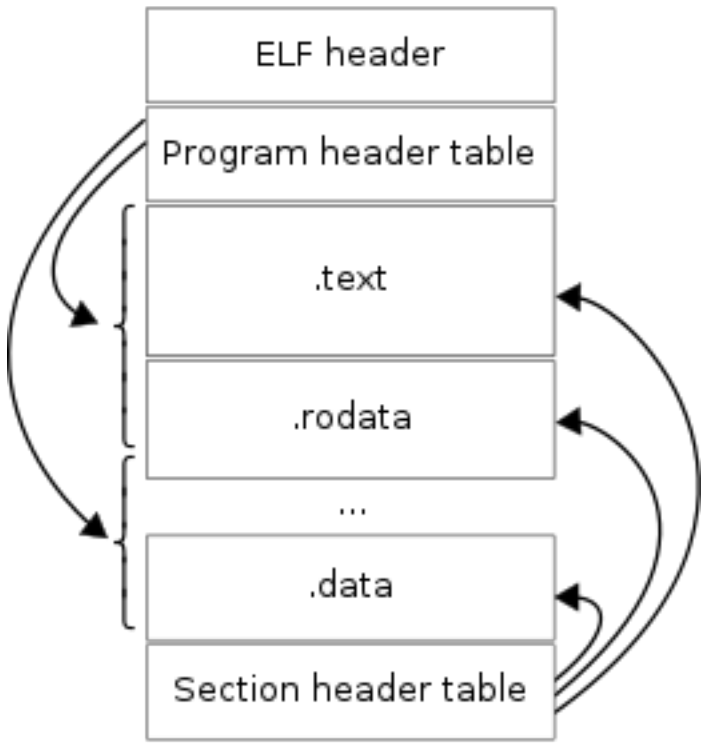
Output: in RISC-V Machine Language (*.o)

- The assembler produces the object using the **Executable and Linkable Format** (ELF, formerly named ***Extensible Linking Format***) standard format.

```
00000000 <main>:
  0: ff010113   addi  sp,sp,-16
  4: 00112623   sw    ra,12(sp)
  8: 00000537   lui   a0,0x0
  c: 00050513   mv    a0,a0
 10: 000005b7   lui   a1,0x0
 14: 00058593   mv    a1,a1
 18: 00000097   auipc ra,0x0
 1c: 000080e7   jalr  ra
 20: 00c12083   lw    ra,12(sp)
 24: 01010113   addi  sp,sp,16
 28: 00000513   li    a0,0
 2c: 00008067   ret
```

Executable and Linkable Format (ELF)

- ELF is a common standard file format for executable files, object code, shared libraries, and core dumps.
- The standard binary file format for Unix and Unix-like systems on x86 processors by the 86open project.
- By design, the ELF format is flexible, extensible, and cross-platform. This has allowed it to be adopted by many different operating systems on many different hardware platforms.

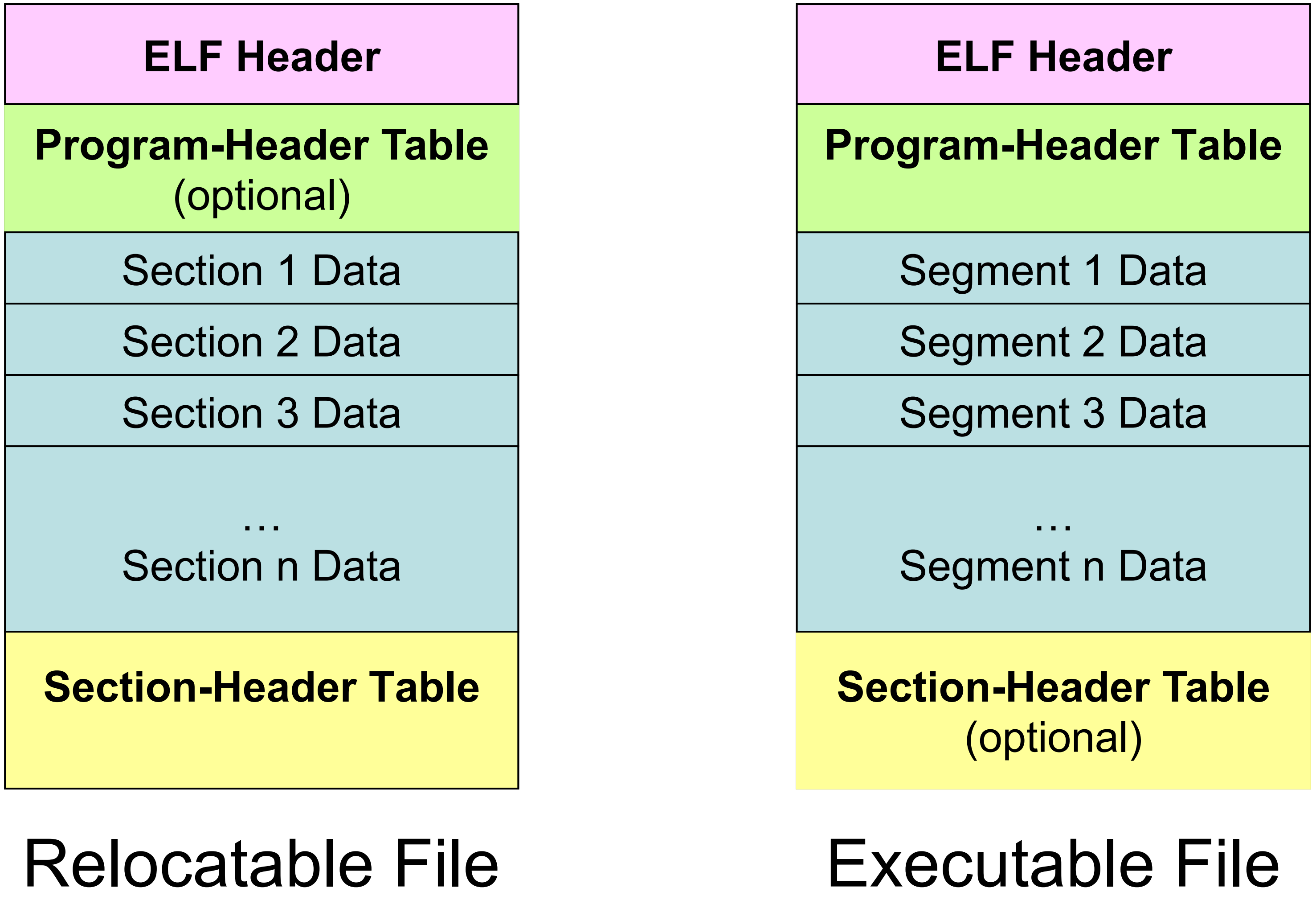


An ELF file has two views: the program header shows the *segments* used at run time, whereas the section header lists the set of *sections*.

ELF Object Files

- Created by the assembler and the linker, object files are binary representations of programs intended to be executed directly on a processor.
  - Programs that require other abstract machines, such as shell scripts, are excluded.
- There are three main types of object files:
  - A **relocatable** file holds code and data suitable for **linking** with other object files to create an executable or a shared object file.
  - An **executable** file holds a program suitable for **execution**; the file specifies how ***exec*** creates a program's process image.
  - A **shared object** file holds code and data suitable for linking in two contexts. First, the linker processes the shared object file with other relocatable and shared object files to create another object file. Second, the dynamic linker combines it with an executable file and other shared objects to create a process image.

Relocatable vs. Executable



Relocatable Object File

- As assembler's output
  - Binary machine code, but **NOT executable**
    - E.g., .o for Linux, while .obj for Windows
  - May refer to external symbols
    - Need a symbol table
  - Each object file has its own address space
    - Addresses will need to be fixed later

Symbols and References

- Global labels: Externally visible “exported” symbols
  - Can be referenced from other object files
  - Exported functions, global variables

- Local labels: Internal visible only symbols
  - Only used within this object file
  - Static functions, static variables, loop labels, …

Issues Need to be Resolved

- How does the Assembler resolve local references?
- How does the Assembler resolve external references?

How to Resolve Local References?

- Handle forward references
  - Two-pass assembly
    - Do a pass through the whole program, allocate instructions and lay out data, thus determining addresses
    - Do a second pass, emitting instructions and data, with the correct label offsets now determined
  - One-pass (or backpatch) assembly
    - Do a pass through the whole program, emitting instructions, emit a 0 for jumps to labels not yet determined, keep track of where these instructions are
    - Backpatch, fill in 0 offsets as labels are defined

Format of Relocatable Object File

- Header
  - Size and position of pieces of file
- Text Section
  - Instructions
- Data Section
  - Static data (local/global vars, strings, constants)
- Debugging Information
  - Line number -> code address map, etc.
- Symbol Table
  - External (exported) references
  - Unresolved (imported) references

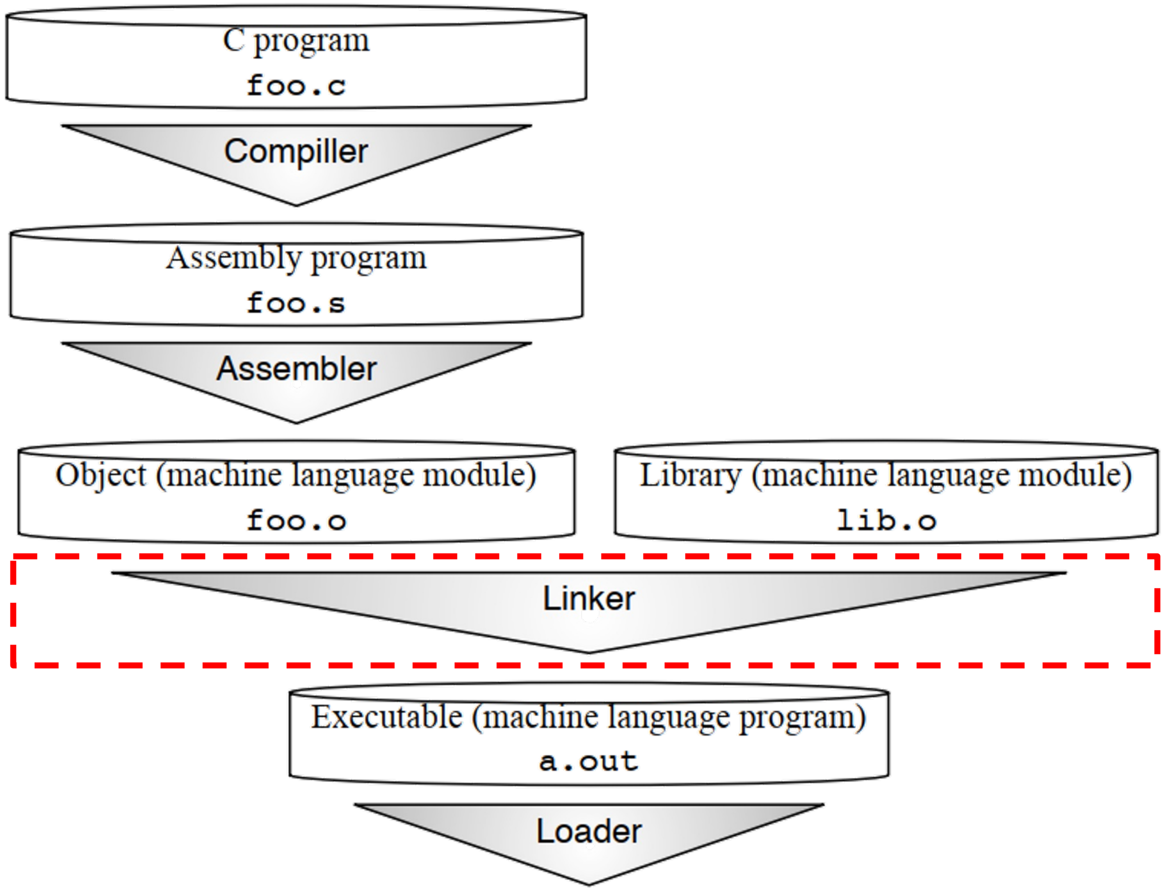
Commonly Used Sections

- Text (.section .text)
  - Contain code (instructions)
- Read-Only Data (.section .rodata)
  - Contains pre-initialized constants
- Read-Write Data (.section .data)
  - Contains pre-initialized variables
- BSS (.section .bss)
  - Contains un-initialized data
  - http://www.faqs.org/faqs/unix-faq/faq/part1/section-3.html

- Useful Tools: **Objdump & Readelf**

Linker: (multiple) *.o -> a.out

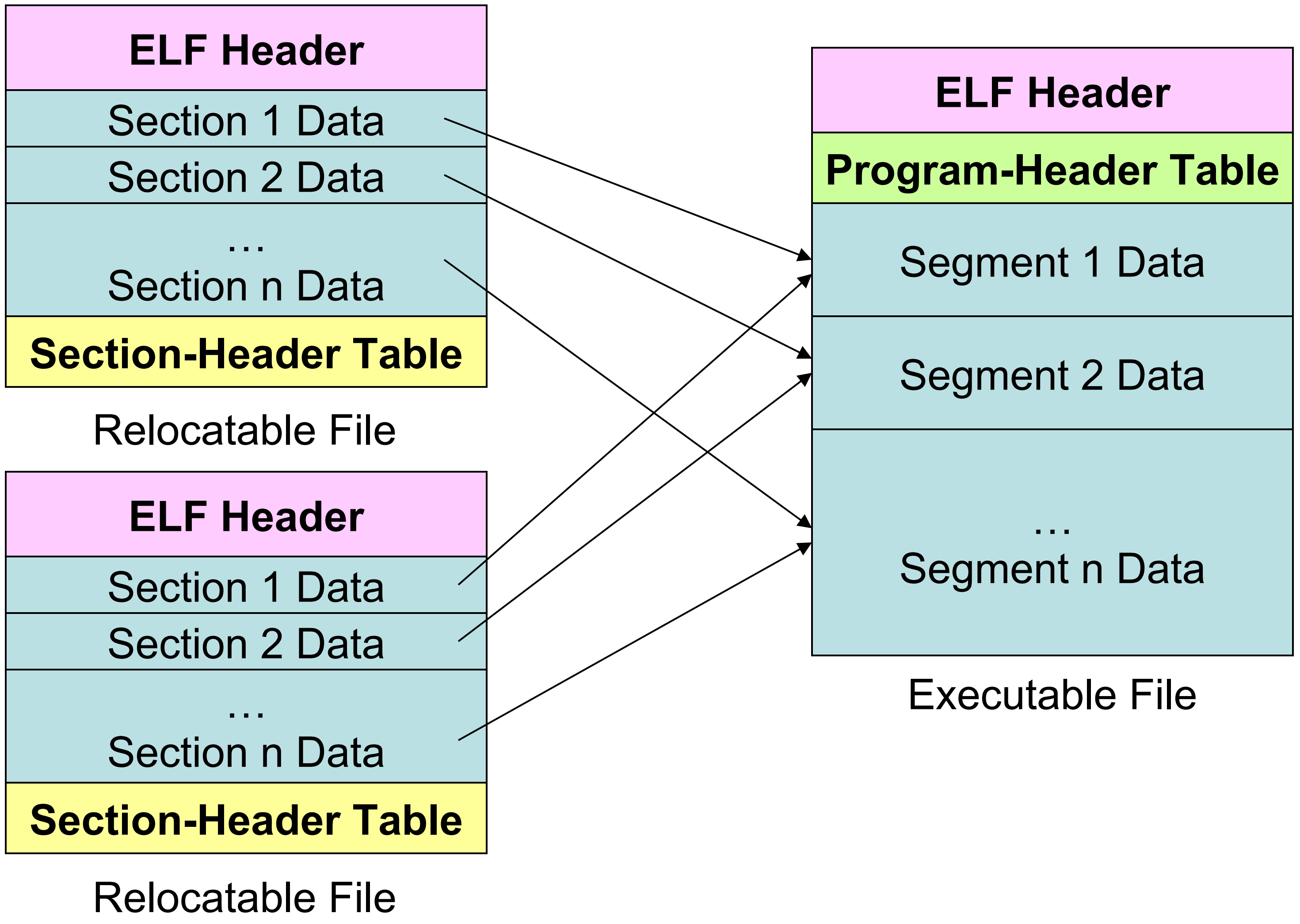
- Steps of translation from C source code to a running program. These are the logical steps, although some steps are combined to accelerate translation.


C program
foo.c
Compiller
Assembly program
foo.s
Assembler
Object (machine language module)
foo.o
Library (machine language module)
lib.o
Linker
Executable (machine language program)
a.out
Loader

Linker

- Rather than compile all the source code every time one file changes, the linker allows individual files to be compiled and assembled separately.
- Why separate compile/assemble and linking steps?
  - Separately compiling modules and linking them together obviates the need to recompile the whole program every time something changes
    - Need to just recompile a small module
    - A linker coalesces object files together to create a complete program
- The linker "stitches" the new object code together with existing machine language modules, such as libraries.

Role of Linker



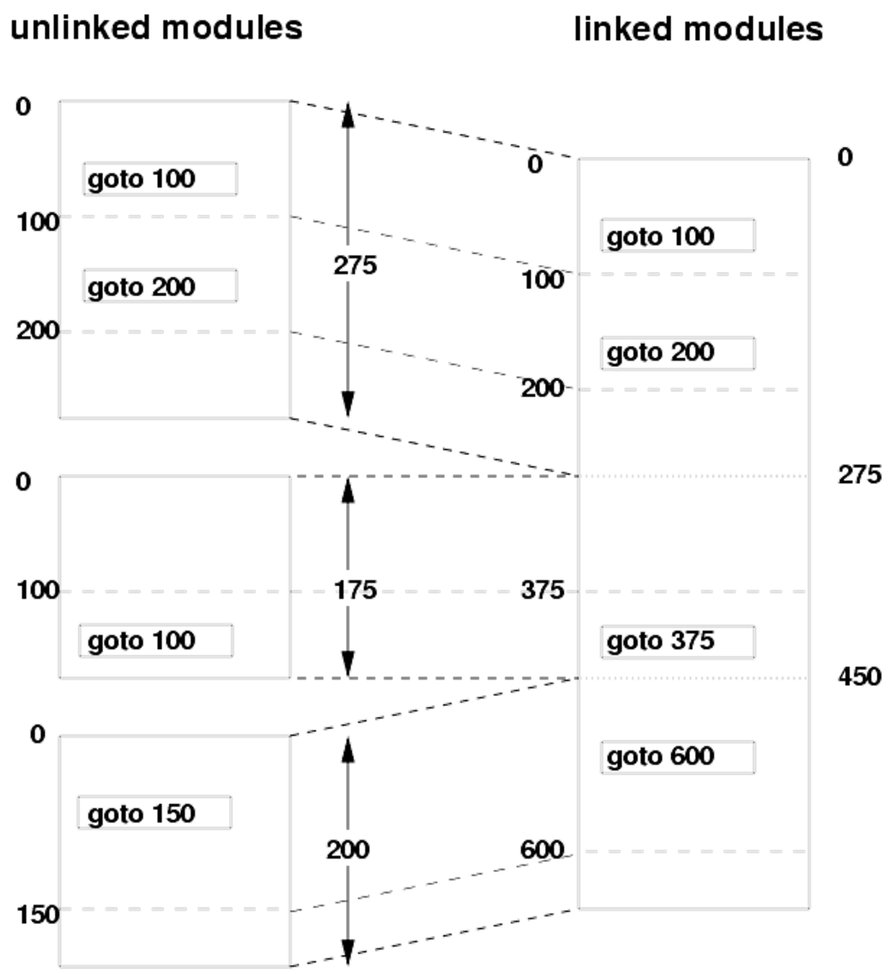
Issues Need to be Resolved

- How does the linker combine separately compiled files?
- How does linker resolve unresolved references?
- How does linker relocate data and code segments

- To combine object files into an executable file
  - Relocate each object's text and data segments
  - Resolve as-yet-unresolved symbols
  - Record top-level entry point in executable file

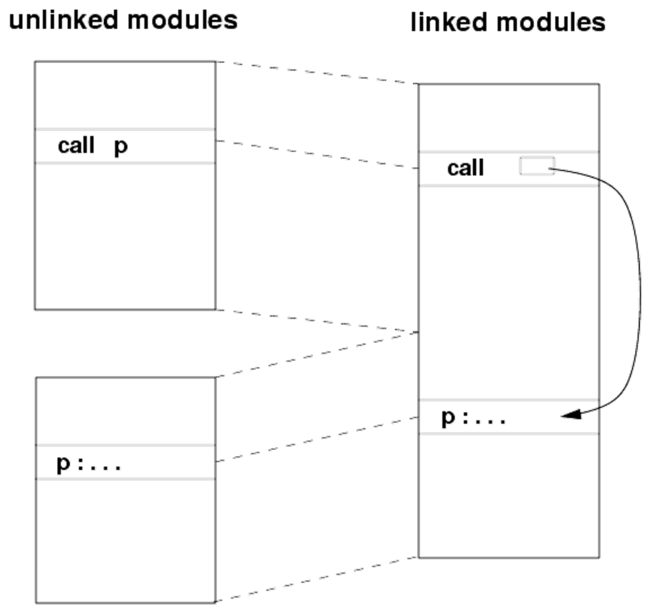
Linker Functions 1: Fixing Addresses

- Addresses in an object file are usually relative to the start of the code or data segment in that file.
- When different object files are combined:
  - The same kind of segments (text, data, read-only data, etc.) from the different object files get merged.
  - Addresses have to be "fixed up" to account for this merging.
  - The fixing up is done by the linker, using information embedded in the executable for this purpose ("relocations").



Linker Function 2: Symbol Resolution

- Suppose:
  - Module B defines a symbol x;
  - Module A refers to x.
- The linker must:
  - Determine the location of x in the object module obtained from merging A and B; and
  - Modify references to x (in both A and B) to refer to this location.



- Each linkable module contains a symbol table, whose contents include:
  - Global symbols defined (maybe referenced) in the module.
  - Global symbols referenced but not defined in the module (these are generally called externals).
  - Segment names (e.g., text, data, rodata).
  - These are usually considered to be global symbols defined to be at the beginning of the segment.
  - Non-global symbols and line number information (optional), for debuggers.

Format of Executable Object File

- Header
  - **Location of main entry point**
- Text Segment
  - Instructions
- Data Segment
  - Static data (local/global vars, strings, constants)
- **Relocation Information**
  - Instructions and data that depend on actual addresses
  - Linker patches these bits after relocating segments
- Symbol Table
  - Exported and imported references
- Debugging Information

Various Executable Object Files

- Unix/Linux
  - a.out
  - COFF: Common Object File Format
  - ELF: Executable and Linking Format
  - …
- Windows
  - PE: Portable Executable
- All support both executable and other object files

Output: a.out

- In addition to the instructions, each object file contains a symbol table that includes all the labels in the program that must be given addresses as part of the linking process. This list includes labels to data as well as to code.

```
000101b0 <main>:
    101b0: ff010113 addi sp,sp,-16
    101b4: 00112623 sw   ra,12(sp)
    101b8: 00021537 lui  a0,0x21
    101bc: a1050513 addi a0,a0,-1520 # 20a10 <string1>
    101c0: 000215b7 lui  a1,0x21
    101c4: a1c58593 addi a1,a1,-1508 # 20a1c <string2>
    101c8: 288000ef jal  ra,10450 <printf>
    101cc: 00c12083 lw   ra,12(sp)
    101d0: 01010113 addi sp,sp,16
    101d4: 00000513 li   a0,0
    101d8: 00008067 ret
```

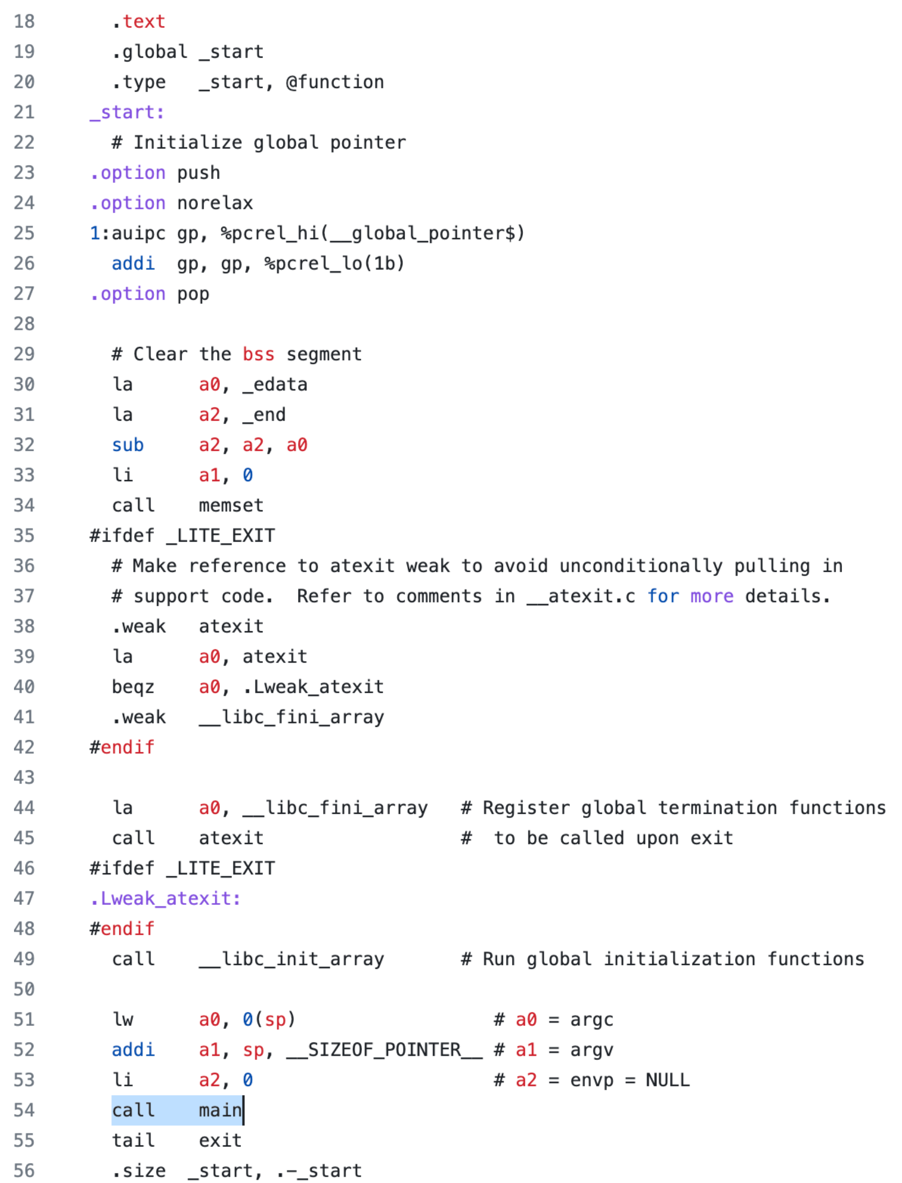
Actual Entry Point: _start and crt0

- For most C and C++ programs, the true entry point is not ***main***, it's the ***_start*** function, which initializes the program runtime and invokes the program's *main* function.

- Conventionally, it is implemented as ***crt0*** (also known as ***c0***), a set of execution startup routines linked into a C program that performs any initialization work required before calling the program's *main* function.

- *crt0* generally takes the form of an object file called ***crt0.o***, often written in assembly language (***crt0.s***), which is automatically included by the linker into every executable file it builds. "crt" stands for "C runtime", and the zero stands for "the very beginning".

An Example of crt0.s



```
  .text
  .global _start
  .type   _start, @function
_start:
  # Initialize global pointer
.option push
.option norelax
1:auipc gp, %pcrel_hi(__global_pointer$)
  addi  gp, gp, %pcrel_lo(1b)
.option pop

  # Clear the bss segment
  la      a0, _edata
  la      a2, _end
  sub     a2, a2, a0
  li      a1, 0
  call    memset
#ifdef _LITE_EXIT
  # Make reference to atexit weak to avoid unconditionally pulling in
  # support code.  Refer to comments in __atexit.c for more details.
  .weak   atexit
  la      a0, atexit
  beqz    a0, .Lweak_atexit
  .weak   __libc_fini_array
#endif

  la      a0, __libc_fini_array   # Register global termination functions
  call    atexit                  #  to be called upon exit
#ifdef _LITE_EXIT
.Lweak_atexit:
#endif
  call    __libc_init_array       # Run global initialization functions

  lw      a0, 0(sp)                  # a0 = argc
  addi    a1, sp, __SIZEOF_POINTER__ # a1 = argv
  li      a2, 0                      # a2 = envp = NULL
  call    main
  tail    exit
  .size  _start, .-_start
```

Quiz

```
#include <stdio.h>
#include <stdlib.h>


#define ITEM_NUM 16
static int ITEM_SIZE = 4;
```

Where does the assembler place the following symbols in the object file that it creates?

A. Text Section
B. Data Section
C. Exported reference in symbol table
D. Imported reference in symbol table
E. None of the above

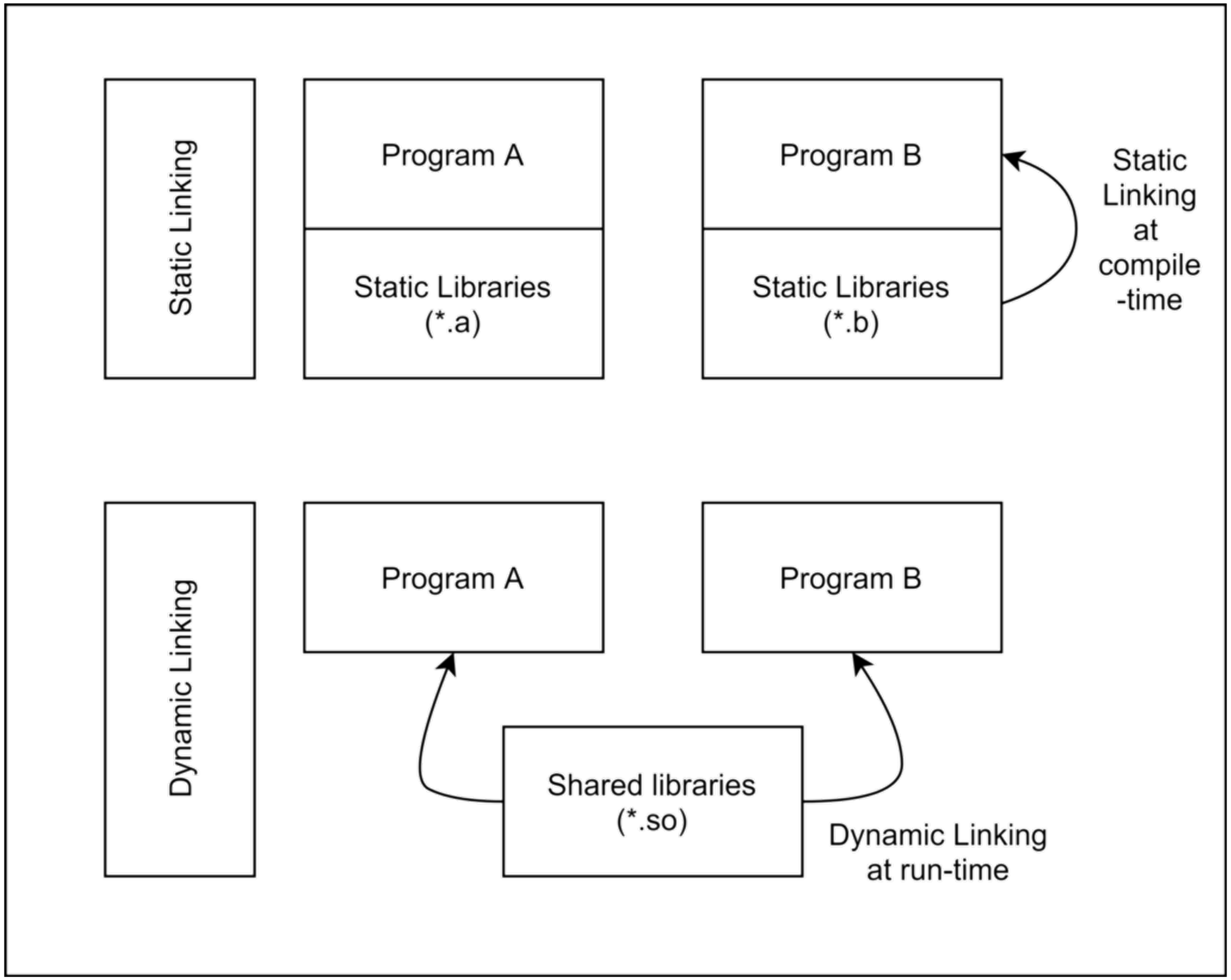
```
int main() {
    size_t buf_size = ITEM_NUM * ITEM_SIZE * sizeof(char);
    char* heap_buf = (char *) malloc(buf_size);
    if (heap_buf) {
        printf("Succeed to allocate: %d!\n", buf_size);
    }
    free(heap_buf);
    return 0;
}
```

Q1: ITEM_NUM
Q2: ITEM_SIZE
Q3: malloc

Static vs. Dynamic Linking

- Static linking
  - All potential library code is linked and then loaded together ***before execution***.
  - Such libraries can be relatively large, so linking a popular library into multiple programs wastes memory. Moreover, the libraries are bound when linked—even when they are updated later to fix bugs—forcing the statically-linked code to use the old, buggy version.
- Dynamic linking
  - The desired external function is ***loaded and linked to the program only after it is first called***; if it's never called, it's never loaded and linked. Every call after the first one uses a fast link, so the dynamic overhead is only paid once.
  - Each time a program starts it links in the current version of the library functions it needs, which is how it can get the newest version. Furthermore, if multiple programs use the same dynamically linked library, the code in the library need appear only once in memory.
  - Instead of jumping to the real function, it jumps to a short (three-instruction) ***stub function***.

Static vs. Dynamic Linking



Static Libraries

- Static Library: Collection of object files
  (think: like a zip archive)

- Q: Every program contains the entire library?
- A: No, the linker picks only object files needed to resolve undefined references at link time

- E.g., libc.a contains many objects:
  - printf.o, fprintf.o, vprintf.o, sprintf.o, snprintf.o, …
  - read.o, write.o, open.o, close.o, mkdir.o, readdir.o, …
  - rand.o, exit.o, sleep.o, time.o, ….

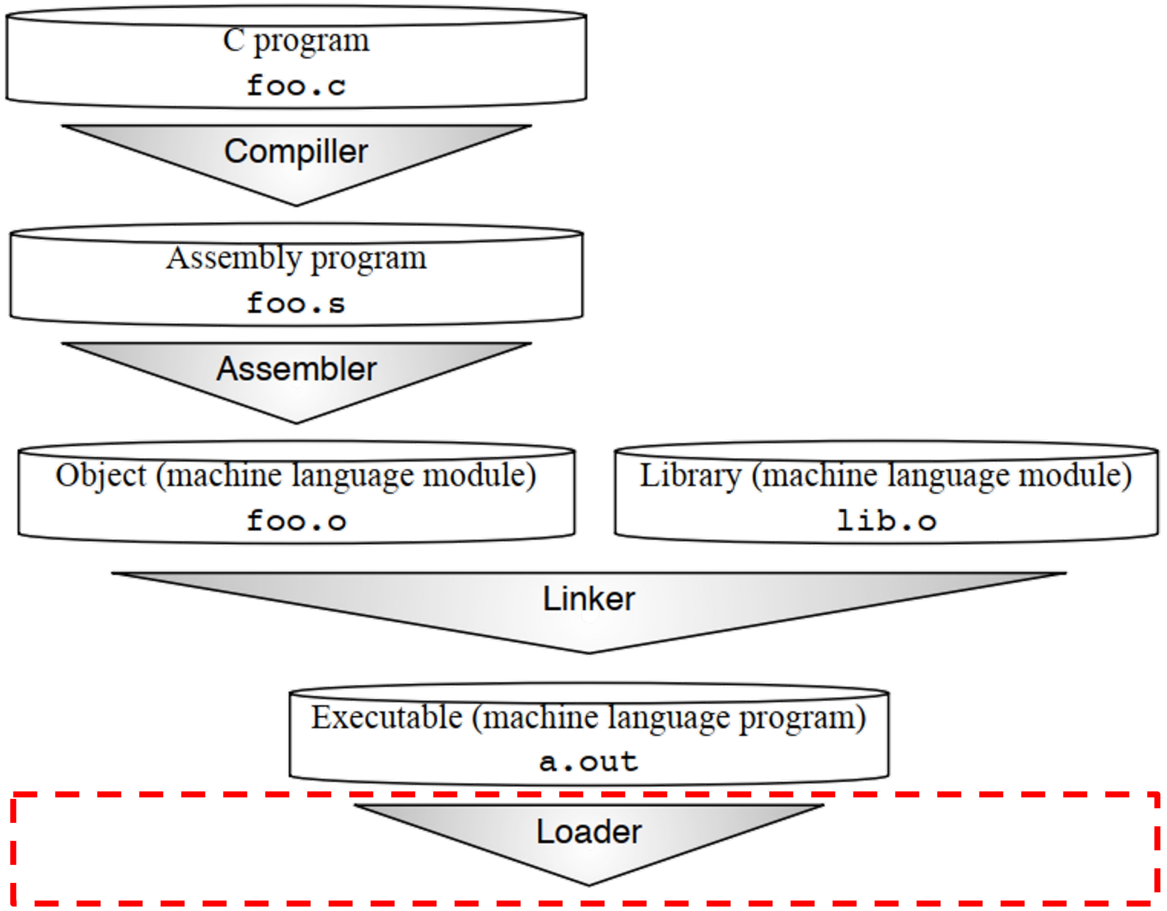
Shared Libraries

- Q: Every program contains parts of same library?
- A: No, they can use shared libraries
  - Executables all point to single shared library on disk
  - **Final linking (and relocations) done by the loader**

- Optimizations:
  - Library compiled at fixed non-zero address
  - Jump table in each program instead of relocations
  - Can even patch jumps *on-the-fly*

Loader: Run a Program

- Steps of translation from C source code to a running program. These are the logical steps, although some steps are combined to accelerate translation.



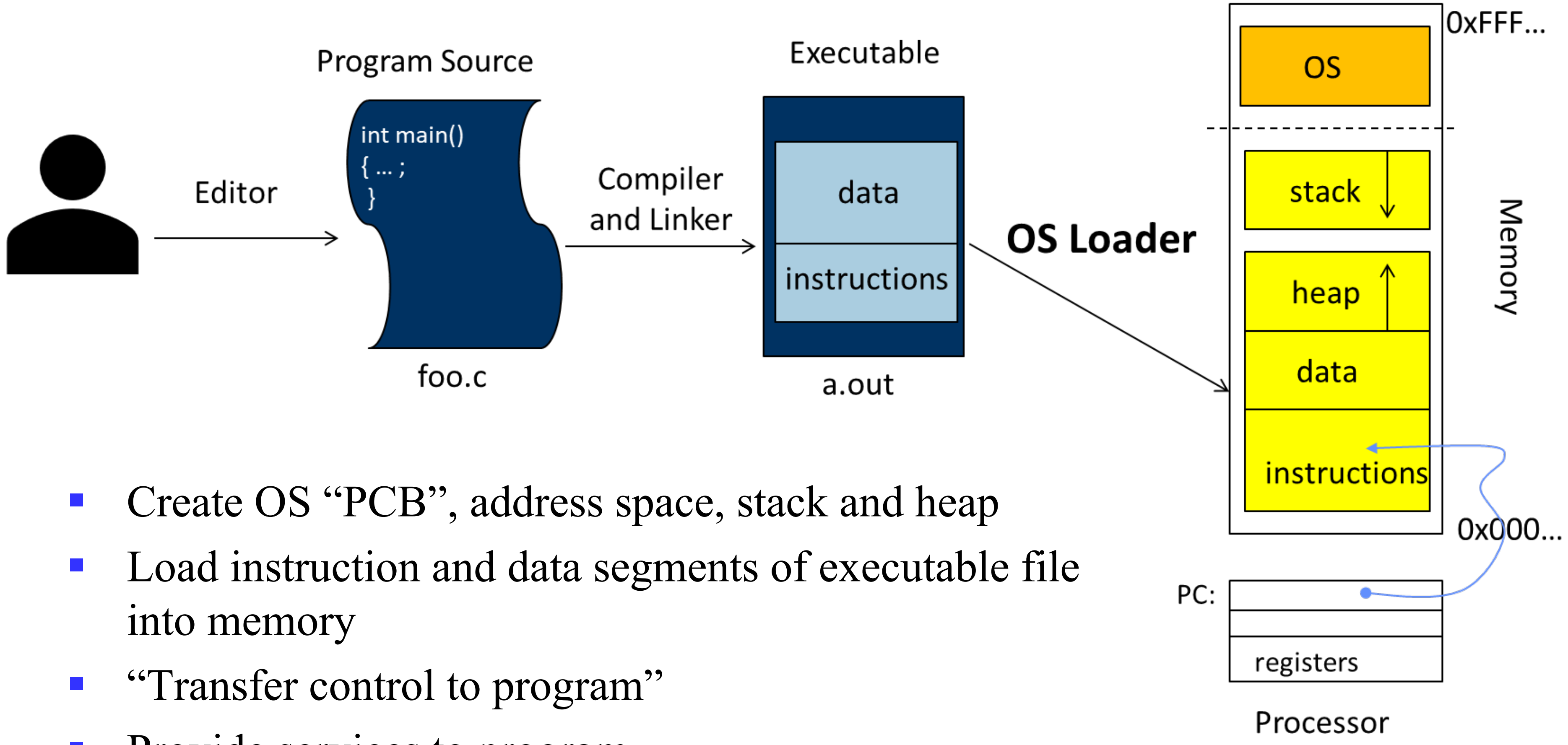
C program
foo.c
Compiller
Assembly program
foo.s
Assembler
Object (machine language module)
foo.o
Library (machine language module)
lib.o
Linker
Executable (machine language program)
a.out
Loader

Loader

- A program like the one in a.out is an executable file kept in the computer's storage. When one is to be run, the loader's job is to load it into memory and jump to the starting address.
  - Initializes registers, stack, arguments to first function
  - Jumps to entry-point

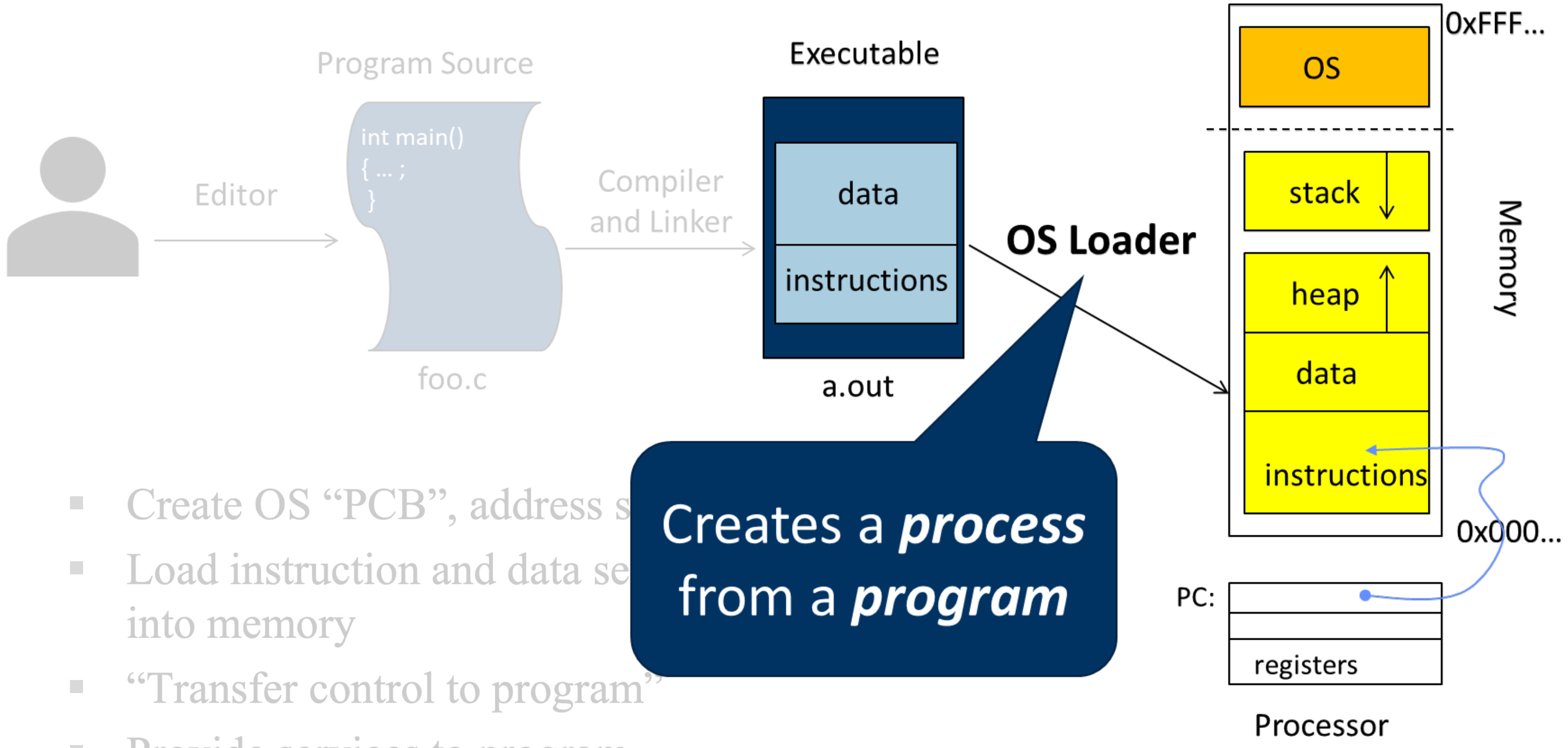
- The "loader" today is the operating system; stated alternatively, loading a.out is one of many tasks of an operating system.

How to Run a Program?



- Create OS "PCB", address space, stack and heap
- Load instruction and data segments of executable file into memory
- "Transfer control to program"
- Provide services to program
- While protecting OS and program

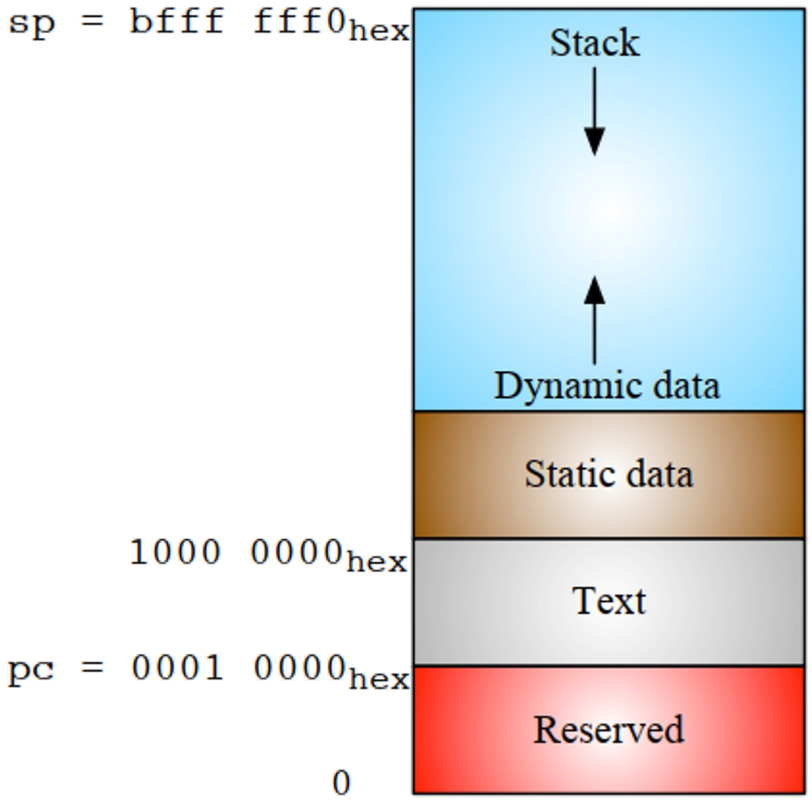
How to Run a Program?



- Create OS "PCB", address s
- Load instruction and data se into memory
- "Transfer control to program"
- Provide services to program
- While protecting OS and program

Memory Layout

- RV32I allocation of memory to program and data. The high addresses are the top of the figure, and the low addresses are the bottom.
- In this RISC-V software convention, the stack pointer (sp) starts at 0xbffffff0 and grows down toward the Static data.
- The text (program code) starts at 0x00010000 and includes the statically-linked libraries.
- The Static data starts immediately above the text region; in this example, we assume that address is 0x10000000.
- Dynamic data, allocated in C by malloc(), is just above the Static data. Called the heap, it grows upward toward the stack. It includes the dynamically-linked libraries.



Loading Statically-linked Programs

- Programs are usually loaded at a fixed address in a fresh address space (so can be linked for that address).
- In such systems, loading involves the following actions:
  - Determine how much address space is needed from the object file header;
  - Allocate that address space;
  - Read the program into the segments in the address space;
  - Zero out any uninitialized data (".bss" segment) if not done automatically by the virtual memory system.
  - Create a stack segment;
  - Set up any runtime information, e.g., program arguments or environment variables.
  - Start the program executing.

Loading Dynamically-linked Programs

- Loading is a little trickier for dynamically-linked programs. Instead of simply starting the program, the operating system starts ***the dynamic linker***. It in turn starts the desired program, and then handles all first-time external calls, copies the functions into memory, and edits the program after each call to point it to the correct function.
  - GOT & PLT

Position-Independent Code (PIC)

- If the load address for a program is not fixed (e.g., shared libraries), we use position independent code.
- Basic idea: separate code from data; generate code that doesn't depend on where it is loaded.
- PC-relative addressing can give position-independent code references.

PIC (cont'd): ELF Files

- ELF executable file characteristics:
  - Data pages follow code pages;
  - The offset from the code to the data does not depend on where the program is loaded.
- The linker creates a ***global offset table*** (**GOT**) that contains offsets to all global data used.
- If a program can load its own address into a register, it can then use a fixed offset to access the GOT, and thence the data.

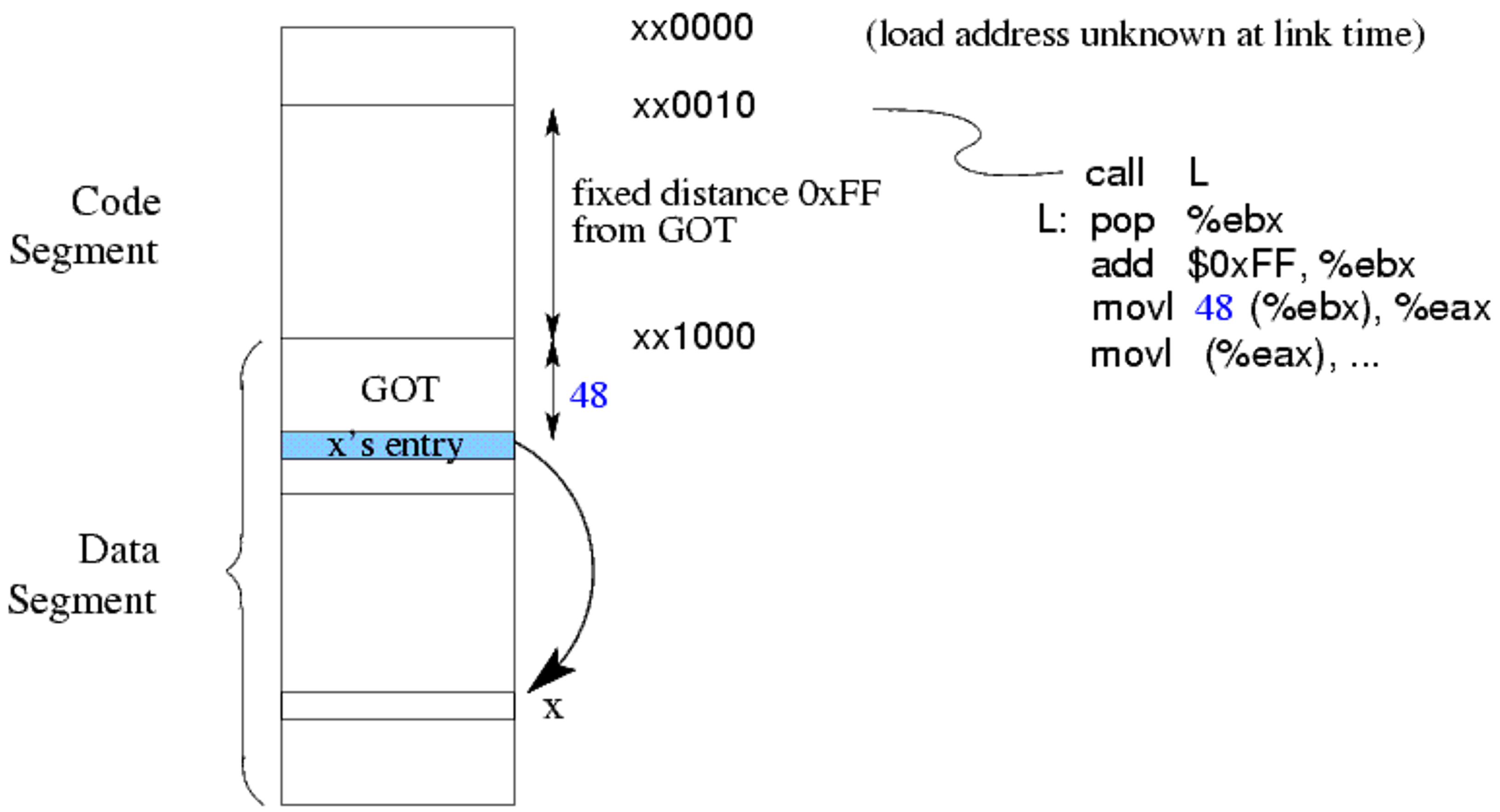
PIC (cont'd): Code on ELF

- Code to figure out its own address (x86):

```
        call  L   /* push address of next instruction on stack */
L:      pop %ebx  /* pop address of this instruction into %ebx */
```

- Accessing a global variable x in PIC:
  - GOT has an entry, say at position k, for x. The dynamic linker fills in the address of x into this entry at load time.
  - Compute "my address" into a register, say %ebx (above);
  - %ebx += offset_to_GOT;   /* fixed for a given program */
  - %eax = contents of location k(%ebx) /* %eax = addr. of x */
  - access memory location pointed at by %eax;

PIC on ELF: Example


xx0000
(load address unknown at link time)
xx0010
fixed distance 0xFF from GOT
xx1000
48
Code Segment
Data Segment
GOT
x's entry
x

```
    call  L
L:  pop   %ebx
    add   $0xFF, %ebx
    movl  48 (%ebx), %eax
    movl  (%eax), ...
```

Based on Linkers and Loaders, by J. R. Levine (Morgan Kaufman, 2000)

Shared Libraries

- Have a single copy of the library that is used by all running programs.
- Saves (disk and memory) space by avoiding replication of library code.
- Virtual memory management in the OS allows different processes to share “read-only” pages, e.g., text and read-only data.
  - This lets us get by with a single physical-memory copy of shared library code.

Shared Libraries: cont’d

- At link time, the linker:
  - Searches a (specified) set of libraries, in some fixed order, to find modules that resolve any undefined external symbols.
  - Puts a list of libraries containing such modules into the executable.
- At load time, the startup code:
  - Finds these libraries;
  - Maps them into the program’s address space;
  - Carries out library-specific initialization.
- Startup code may be in the OS, in the executable, or in a special dynamic linker.

Dynamic Linking

- Defers much of the linking process until the program starts running.
- Easier to create, update than statically linked libraries.
- Has higher runtime performance cost than statically linked libraries:
  - Much of the linking process has to be redone each time a program runs.
  - Every dynamically linked symbol has to be looked up in the symbol table and resolved at runtime.

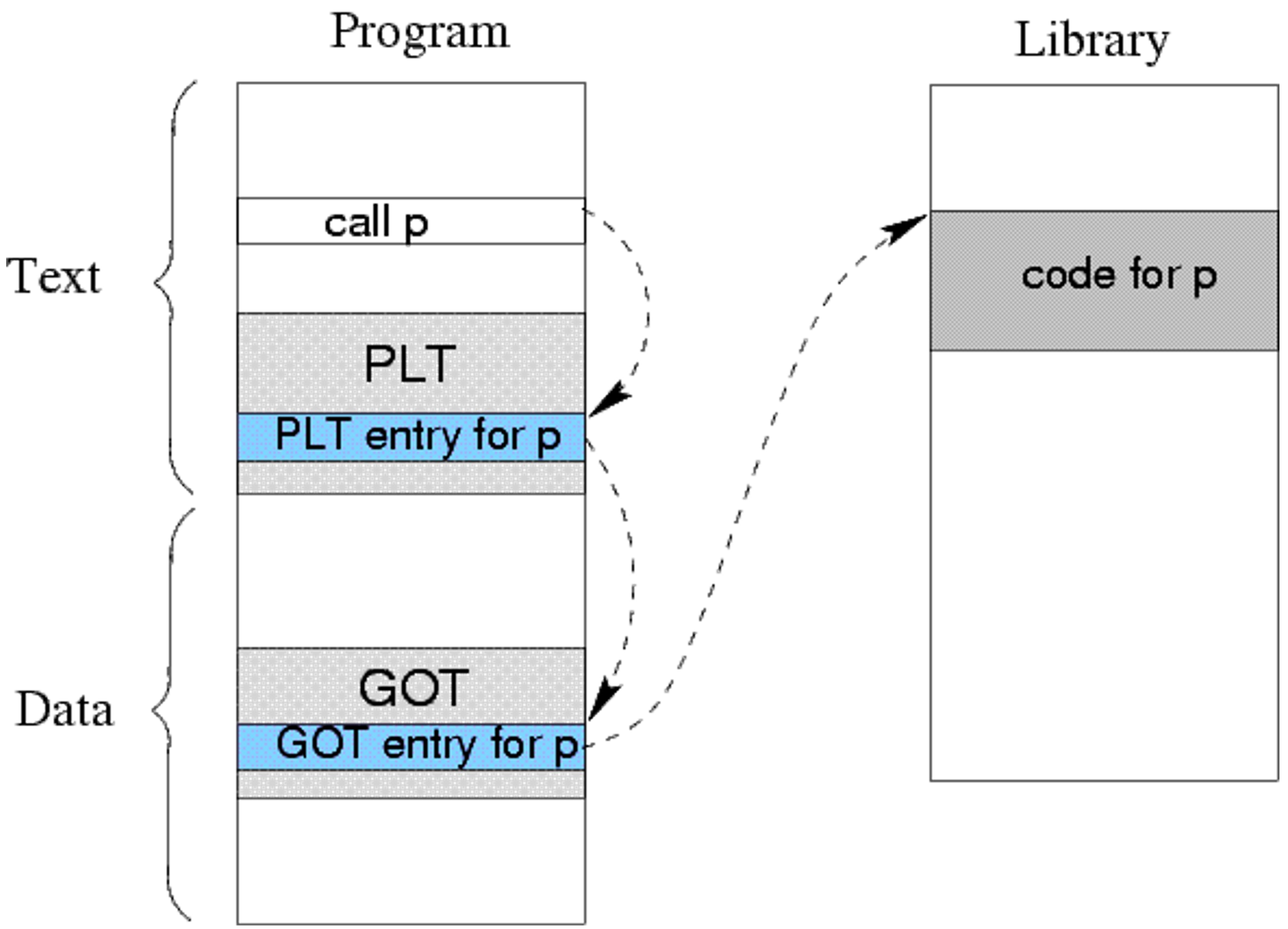
Dynamic Linking: Basic Mechanism

- A reference to a dynamically linked procedure *p* is mapped to code that invokes a *handler*.
- At runtime, when *p* is called, the handler gets executed:
  - The handler checks to see whether p has been loaded already (due to some other reference);
  - If so, the current reference is linked in, and execution continues normally.
  - Otherwise, the code for *p* is loaded and linked in.

Dynamic Linking: ELF Files

- ELF shared libraries use PIC (position independent code), so text sections do not need relocation.
- Data references use a GOT:
  - Each global symbol has a relocatable pointer to it in the GOT;
  - The dynamic linker relocates these pointers.
- We still need to invoke the dynamic linker on the first reference to a dynamically linked procedure.
  - Done using a ***procedure linkage table*** (**PLT**);
  - PLT adds a level of indirection for function calls (analogous to the GOT for data references).

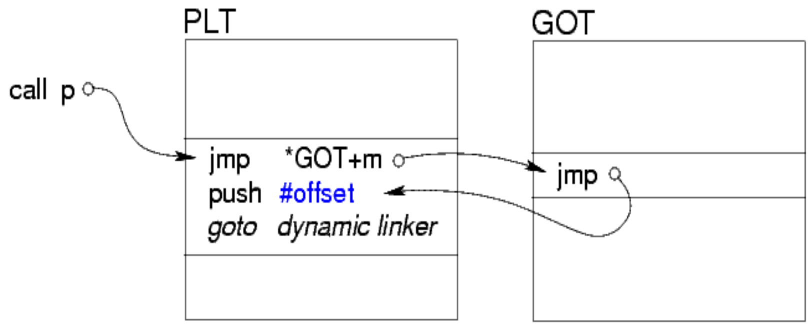
ELF Dynamic Linking: PLT and GOT



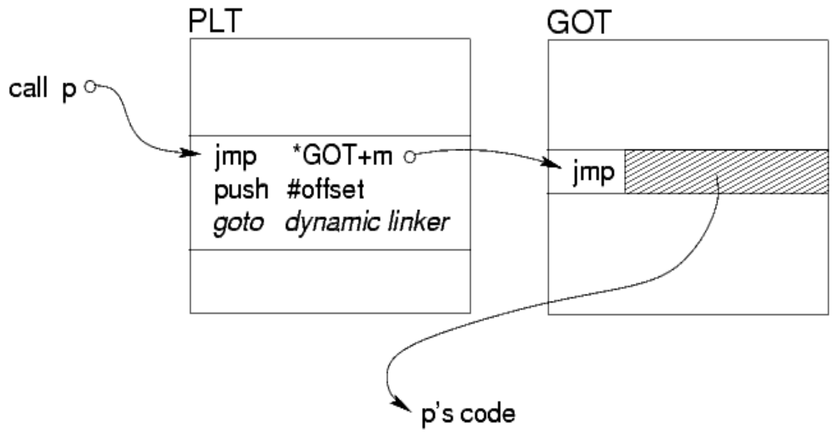
ELF Dynamic Linking: Lazy Linkage

- Initially, GOT entry points to PLT code that invokes the dynamic linker.
  - **Offset** identifies both the symbol being resolved and the corresponding GOT entry.
- The dynamic linker looks up the symbol value and updates the GOT entry.
- Subsequent calls bypass dynamic linker, go directly to callee.
- This reduces program startup time. Also, routines that are never called are not resolved.

Before:



After:



PLT
GOT
call p
jmp *GOT+m
push #offset
goto dynamic linker
jmp
p's code