UC Berkeley
Teaching Professor
Dan Garcia

CS61C

Great Ideas in Computer Architecture

(a.k.a. Machine Structures)

UC Berkeley
Professor
Bora Nikolić

Running a Program – CALL (Compiling, Assembling, Linking, and Loading)

Interpretation vs Translation

Great Idea #1: Abstraction (Levels of Representation/ Interpretation)

High Level Language Program (e.g., C)

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

Compiler

Assembly Language Program (e.g., RISC-V)

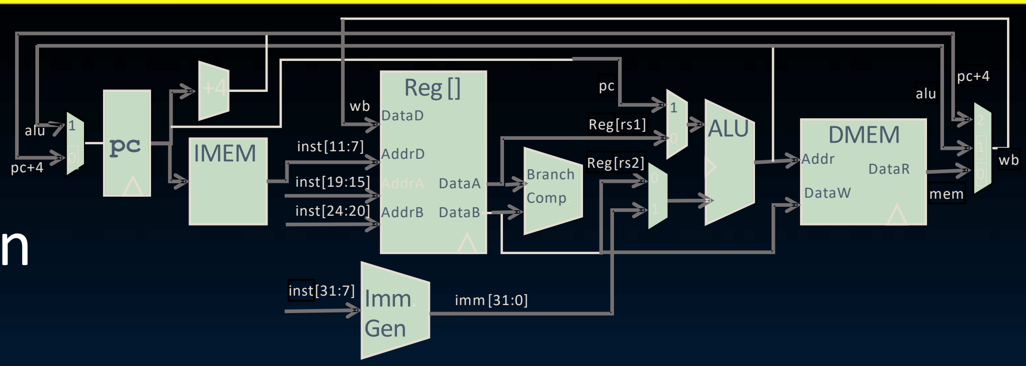
```
lw    x3, 0(x10)
lw    x4, 4(x10)
sw    x4, 0(x10)
sw    x3, 4(x10)
```

Assembler

Machine Language Program (RISC-V)

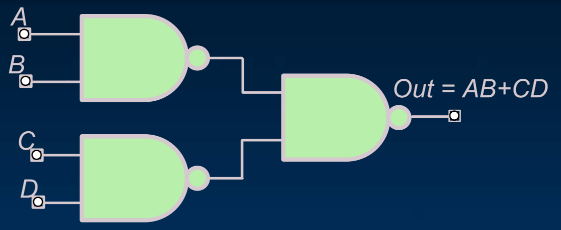
```
1000 1101 1110 0010 0000 0000 0000 0000
1000 1110 0001 0000 0000 0000 0000 0100
1010 1110 0001 0010 0000 0000 0000 0000
1010 1101 1110 0010 0000 0000 0000 0100
```

Hardware Architecture Description (e.g., block diagrams)



Architecture Implementation

Logic Circuit Description (Circuit Schematic Diagrams)



Garcia, Nikolić

Language Execution Continuum

- Interpreter is a program that executes other programs

Java bytecode

Python Java C++ C Assembly Machine code

⟷

Easy to program — Difficult to program

Inefficient to interpret — Efficient to interpret

- Language translation gives us another option
- When to choose? In general, we
  - interpret a high-level language when efficiency is not critical
  - translate to a lower-level language to increase performance

Garcia, Nikolić

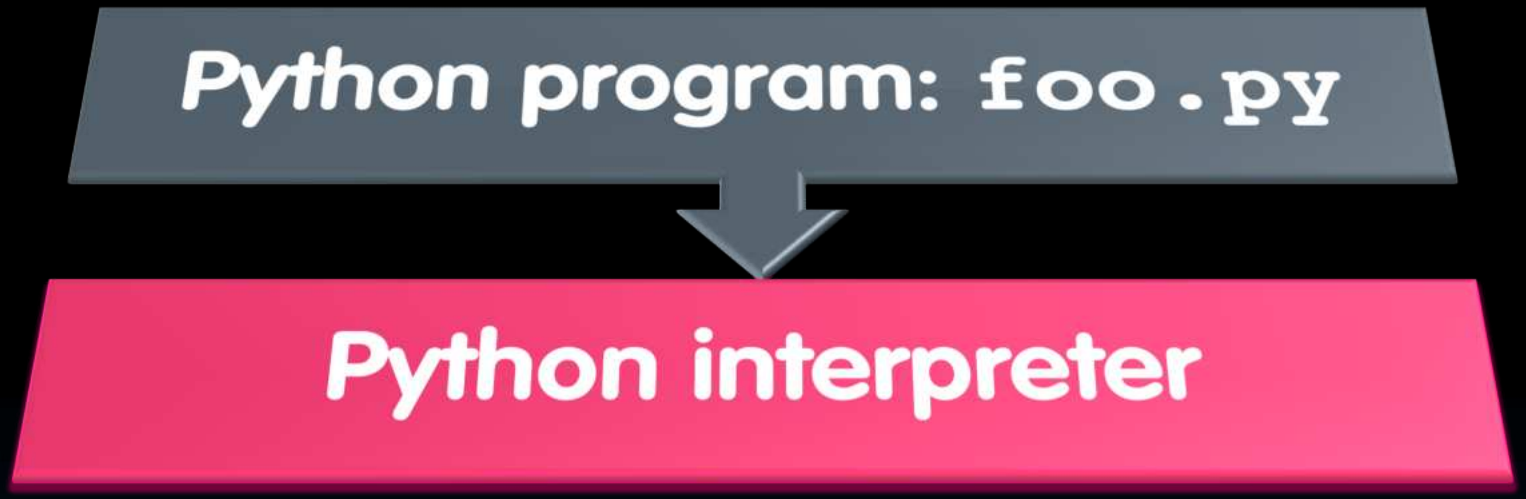
Interpretation vs Translation

- 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

Garcia, Nikolić

Interpretation (1/2)

- For example, consider a Python program **`foo.py`**



- Python interpreter is just a program that reads a python program and performs the functions of that python program

Garcia, Nikolić

Interpretation (2/2)

- WHY interpret machine language in software?
- Eg., VENUS RISC-V simulator useful for learning/debugging
- Eg., Apple Macintosh conversion
  - Switched from Motorola 680x0 ISA to PowerPC (before x86)
  - Could require all programs to be re-translated from high level language
  - Instead, let executables contain old and/or new machine code, interpret old code in software if necessary (emulation)

Garcia, Nikolić

Interpretation vs. Translation? (1/2)

- Generally easier to write interpreter
  - …you did it in CS61A!
- Interpreter closer to high-level, so can give better error messages (e.g., VENUS)
  - Translator reaction: add extra information to help debugging (line numbers, names)
- Interpreter slower (10x?), code smaller (2x?)
- Interpreter provides instruction set independence: run on any machine

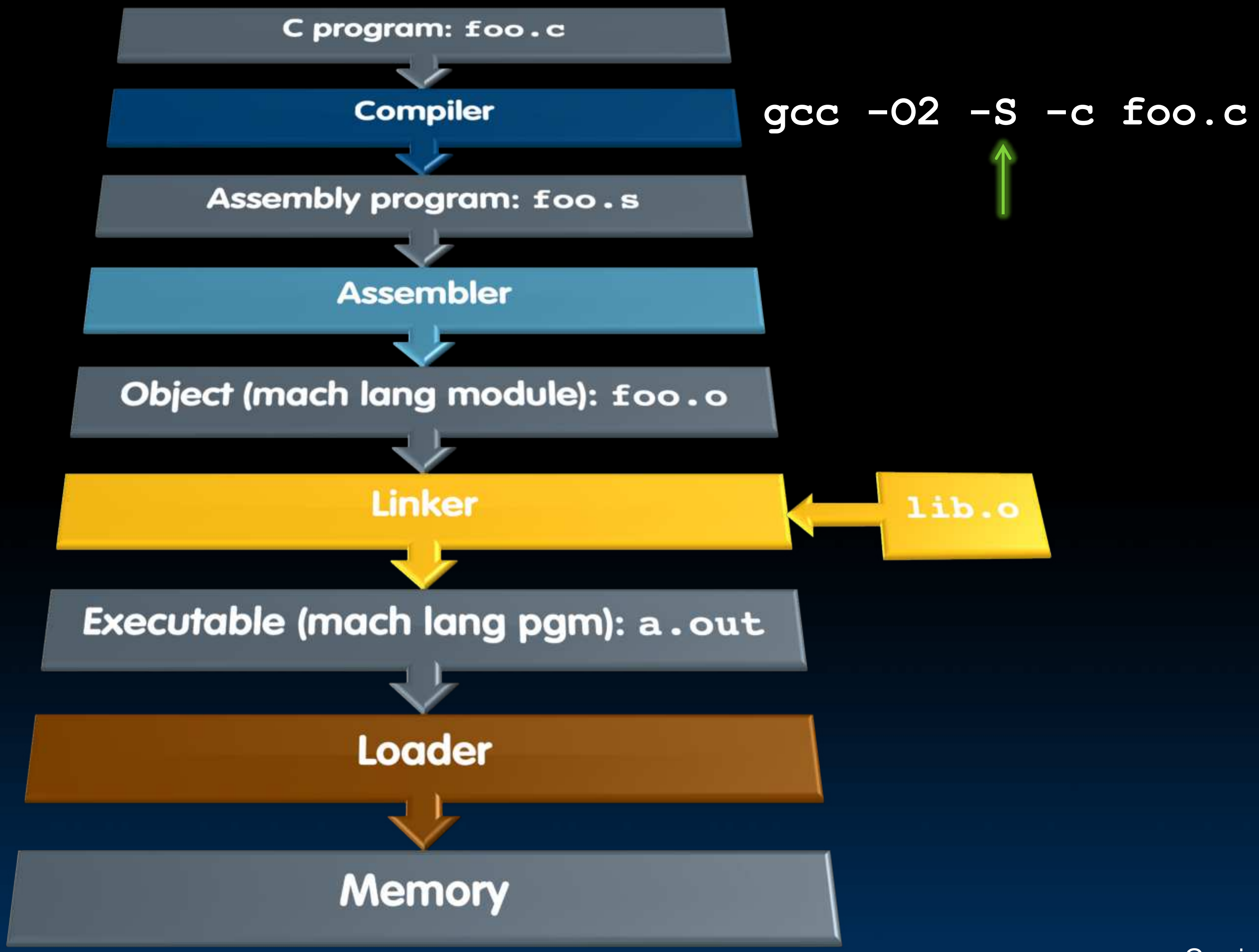
Interpretation vs. Translation? (2/2)

- Translated/compiled code almost always more efficient and therefore higher performance:
  - Important for many applications, particularly operating systems
- Translation/compilation helps "hide" the program "source" from the users:
  - One model for creating value in the marketplace (e.g., Microsoft keeps all their source code secret)
  - Alternative model, "open source", creates value by publishing the source code and fostering a community of developers.

☑

Compiler

Steps in Compiling and Running a C Program

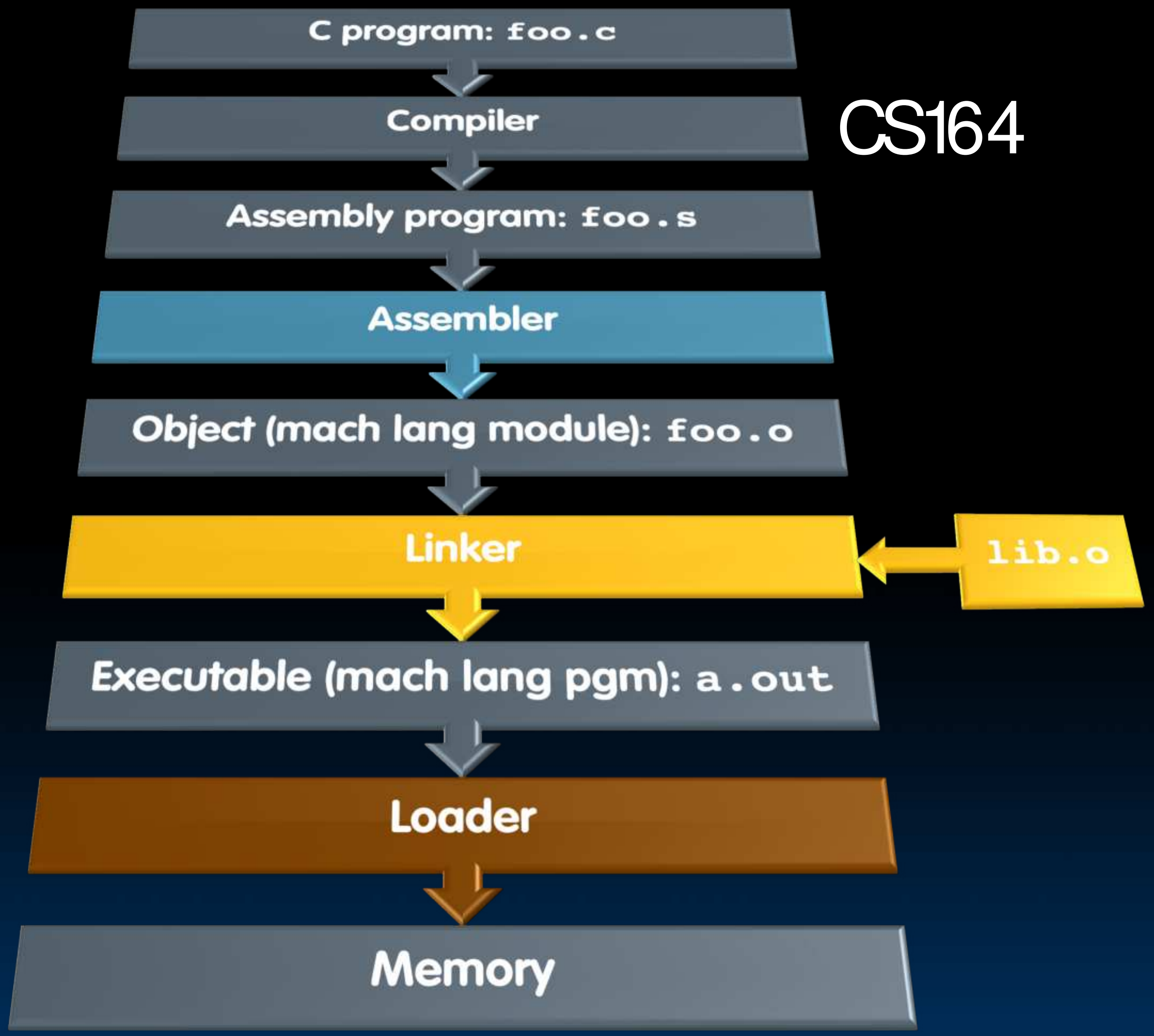


Compiler

- Input: High-Level Language Code (e.g., `foo.c`)
- Output: Assembly Language Code (e.g., `foo.s` for RISC-V)
- Note: Output *may* contain pseudo-instructions
- Pseudo-instructions: instructions that assembler understands but not in machine For example (copy the value from `t2` to `t1`):
  - `mv t1,t2` → `addi t1,t2,0`

Assembler

Where Are We Now?



Assembler

- Input: Assembly Language Code (includes pseudo ops)
  (e.g., `foo.s` for RISC-V)
- Output: Object Code, information tables (true assembly only)
  (e.g., `foo.o` for RISC-V)
- Reads and Uses Directives
- Replace Pseudo-instructions
- Produce Machine Language
- Creates Object File

Garcia, Nikolić

Assembler Directives (See *RISCV Reader, Chapter 3*)

- 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 (source file data in binary)
`.globl sym`:	Declares `sym` global and can be referenced from other files
`.string str`:	Store the string `str` in memory and null-terminate it
`.word w1…wn`:	Store the *n* 32-bit quantities in successive memory words

Garcia, Nikolić

Pseudo-instruction Replacement

- Assembler treats convenient variations of machine language instructions as if real instructions

Pseudo:	Real:
`mv t0, t1`	`addi t0,t1,0`
`neg t0, t1`	`sub t0, zero, t1`
`li t0, imm`	`addi t0, zero, imm`
`not t0, t1`	`xori t0, t1, -1`
`beqz t0, loop`	`beq t0, zero, loop`

`la t0, str`	`lui t0, str[31:12]`
	`addi t0, t0, str[11:0]` **OR**
	`auipc t0, str[31:12]`
	`addi t0, t0, str[11:0]`

DON'T FORGET:
sign extended immediates +
Branch imms count halfwords)
STATIC Addressing
PC-Relative Addressing

Garcia, Nikolić

Producing Machine Language (1/3)

- Simple Case
  - Arithmetic, Logical, Shifts, and so on
  - All necessary info is within the instruction already
- What about Branches and Jumps?
  - PC-Relative (e.g., **`beq/bne`** and **`jal`**)
  - So once pseudo-instructions are replaced by real ones, we know by how many instructions to branch/jump over
- So these can be handled

Garcia, Nikolić

Producing Machine Language (2/3)

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

```
      addi t2,  zero,  9    # t2 = 9
L1:   slt  t1,  zero,  t2   # 0 < t2? Set t1
      beq  t1,  zero,  L2   # NO! t2 <= 0; Go to L2
      addi t2,  t2, -1      # YES! t2 > 0; t2--
      j L1                  # Goto L1
L2:
```

3 words back (6 halfwords)

3 words forward (6 halfwords)

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

Garcia, Nikolić

Producing Machine Language (3/3)

- What about PC-relative jumps (**`jal`**) and branches (**`beq`**, **`bne`**)?
  - **`j offset`** *pseudo instruction* expands to **`jal zero, offset`**
  - Just count the number of instruction *half-words* between target and jump to determine the offset: *position-independent code (PIC)*
- What about references to static data?
  - **`la`** gets broken up into **`lui`** and **`addi`** (use **`auipc/addi`** for PIC)
  - These 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

Garcia, Nikolić

Relocation Table

- List of "items" whose address this file needs
- What are they?
  - Any absolute label jumped to: **`jal, jalr`**
    - Internal
    - External (including lib files)
    - Such as the **`la`** instruction E.g., for **`jalr`** base register
  - Any piece of data in static section
    - Such as the **`la`** instruction E.g., for **`lw/sw`** base register

Garcia, Nikolić

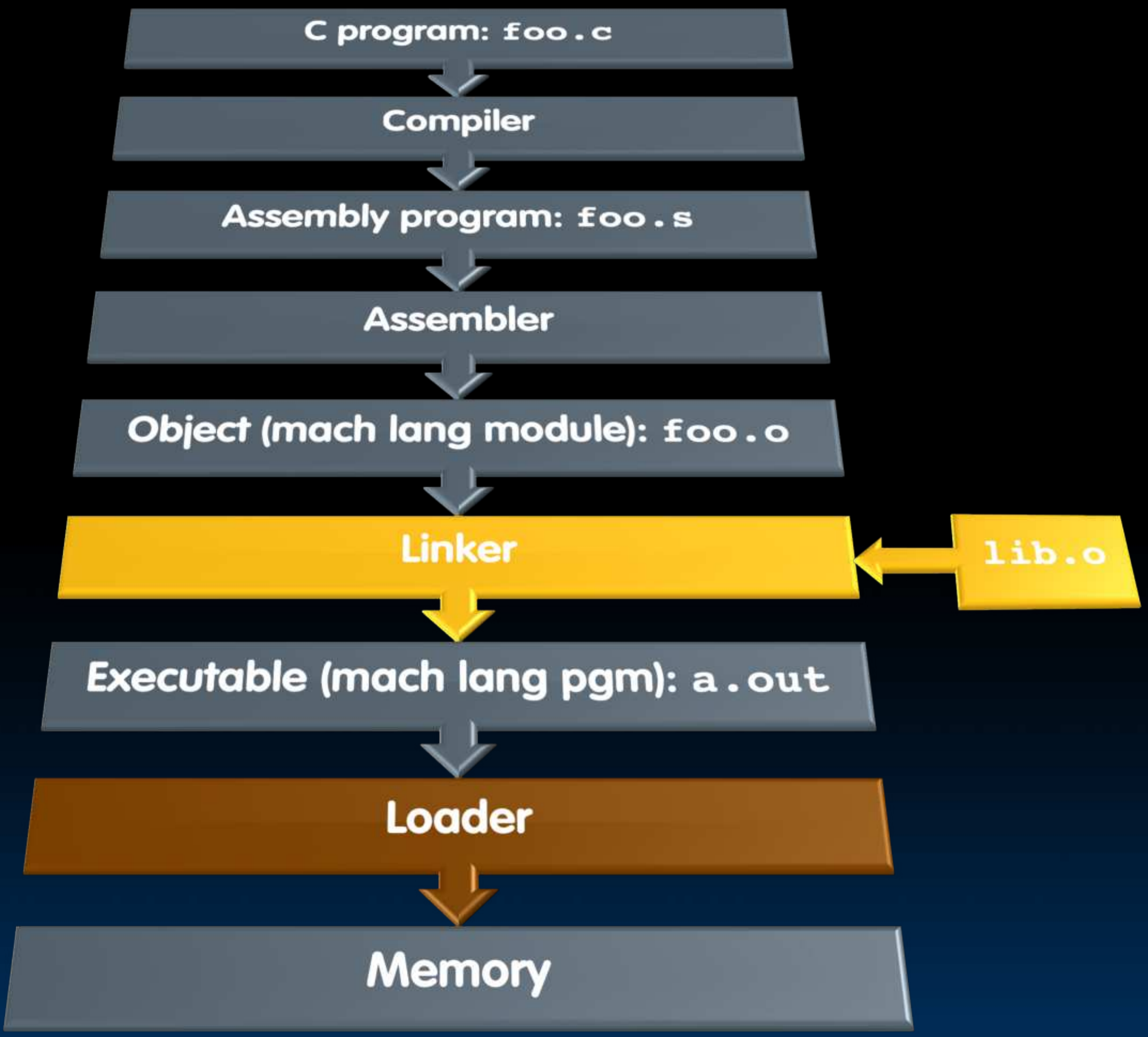
Object File Format

- 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 static data in the source file
- relocation information: identifies lines of code that need to be fixed up later
- symbol table: list of this file's labels and static data that can be referenced
- debugging information
- A standard format is ELF (except MS)
  http://www.skyfree.org/linux/references/ELF_Format.pdf

Garcia, Nikolić

Linker

Where Are We Now?

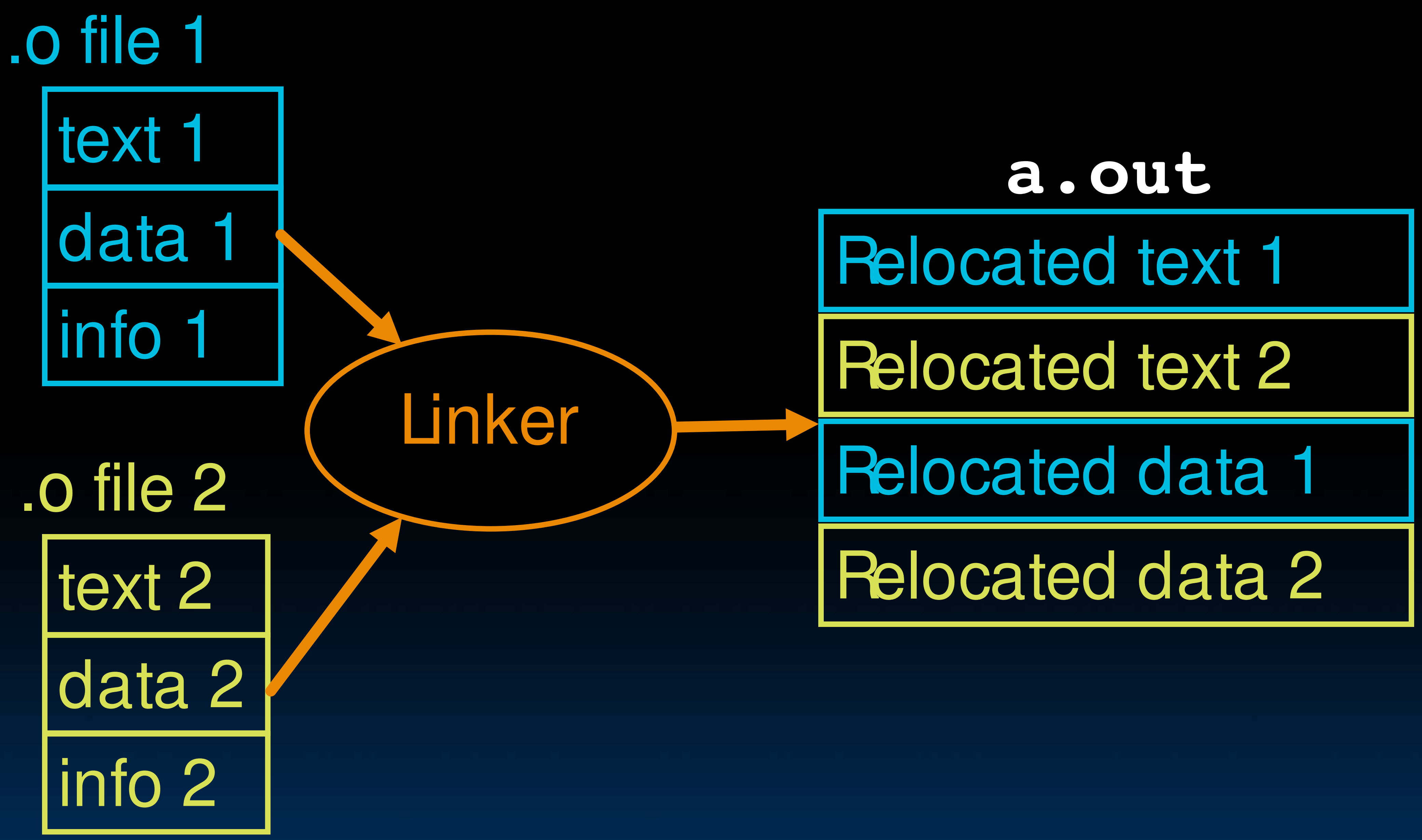


Garcia, Nikolić

Linker (1/3)

- Input: Object code files, information tables (e.g., `foo.o`, `libc.o` for RISC-V)
- Output: Executable code (e.g., `a.out` for RISC-V)
- Combines several object (`.o`) files into a single executable ("linking")
- Enable separate compilation of files
  - Changes to one file do not require recompilation of the whole program
    - Linux source > 20 M lines of code!
  - Old name "Link Editor" from editing the "links" in jump and link instructions

Linker (2/3)



.o file 1
text 1
data 1
info 1
.o file 2
text 2
data 2
info 2
Linker
a.out
Relocated text 1
Relocated text 2
Relocated data 1
Relocated data 2

Linker (3/3)

- Step 1: Take text segment from each `.o` file and put them together
- Step 2: Take data segment from each `.o` file, put them together, and concatenate this onto end of text segments
- Step 3: Resolve references
  - Go through Relocation Table; handle each entry
  - I.e., fill in all absolute addresses

Garcia, Nikolić

Four Types of Addresses

- PC-Relative Addressing (`beq`, `bne`, `jal`; `auipc/addi`)
  - Never need to relocate (PIC: Position-Independent Code)
- Absolute Function Address (`auipc/jalr`)
  - Always relocate
- External Function Reference (`auipc/jalr`)
  - Always relocate
- Static Data Reference (often `lui`/`addi`)
  - Always relocate

Garcia, Nikolić

Absolute Addresses in RISC-V

- Which instructions need relocation editing?
  - J-format: jump/jump and link

xxxxx	rd	jal

  - I-,S- Format: Loads and stores to variables in static area, relative to global pointer

xxx	gp		rd	lw

xx	rs1	gp		x	sw

  - What about conditional branches?

xx	rs1	rs2		x	beq bne

  - PC-relative addressing preserved even if code moves

Garcia, Nikolić

Resolving References (1/2)

- Linker assumes first word of first text segment is at address `0x10000` for RV32
  - (More later when we study "virtual memory")
- 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

Garcia, Nikolić

Resolving References (2/2)

- To resolve references:
  - Search for reference (data or label) in all “user” symbol tables
  - If not found, search library files (e.g., for `printf`)
  - Once absolute address is determined, fill in the machine code appropriately
- Output of linker: executable file containing text and data (plus header)

Static vs. Dynamically Linked Libraries

- What we've described is the traditional way: statically-linked approach
  - Library is now part of the executable, so if the library updates, we don't get the fix (have to recompile if we have source)
  - Includes the entire library even if not all of it will be used
  - Executable is self-contained
- Alternative is dynamically-linked libraries (DLL), common on Windows & UNIX platforms

Garcia, Nikolić

Dynamically Linked Libraries (1/2)

- Space/time issues
  + Storing a program requires less disk space
  + Sending a program requires less time
  + Executing two programs requires less memory (if they share a library)
  – At runtime, there's time overhead to do link
- Upgrades
  + Replacing one file (**`libXYZ.so`**) upgrades every program that uses library "**`XYZ`**"
  – Having the executable isn't enough anymore

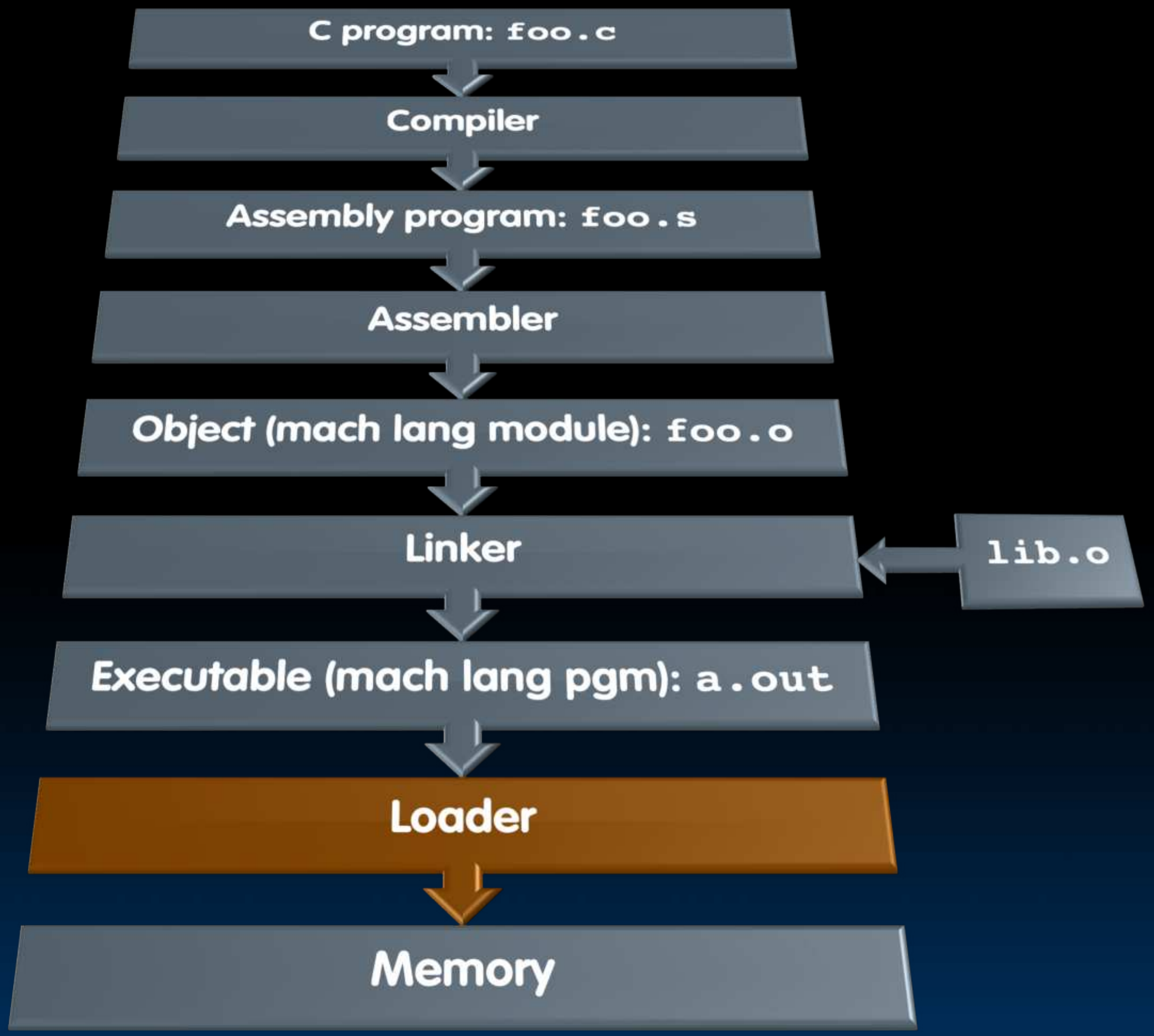
Overall, dynamic linking adds quite a bit of complexity to the compiler, linker, and operating system. However, it provides many benefits that often outweigh these

Dynamically Linked Libraries (2/2)

- Prevailing approach to dynamic linking uses machine code as the "lowest common denominator"
  - Linker does not use information about how the program or library was compiled (i.e., what compiler or language)
  - Can be described as "linking at the machine code level"
  - This isn't the only way to do it ...

Loader

Where Are We Now?



Garcia, Nikolić

Loader Basics

- Input: Executable Code (e.g., `a.out` for RISC-V)
- Output: (program is run)
- 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

Garcia, Nikolić

Loader … What Does It Do?

- Reads executable file's header to determine size of text and data segments
- Creates new address space for program large enough to hold text and data segments, along with a stack segment
- Copies instructions + data from executable file into the new address space
- Copies arguments passed to the program onto the stack
- Initializes machine registers
  - *Most registers cleared, but stack pointer assigned address of 1st free stack location*
- Jumps to start-up routine that copies program's arguments from stack to registers & sets the PC
  - *If main routine returns, start-up routine terminates program with exit system call*

Example

Example C Program: Hello.c

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

Compiled Hello.c: Hello.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
```


Garcia, Nikolić

Assembled `Hello.s`: Linkable `Hello.o`

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

2c: 00008067 jalr ra
```


Garcia, Nikolić

Linked Hello.o: a.out

```
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

   101c0: 000215b7 lui  a1,0x21
   101c4: a1c58593 addi a1,a1,-1508
<string2>
   101c8: 288000ef jal  ra,10450      # <printf>
   101cc: 00c12083 lw   ra,12(sp)
   101d0: 01010113 addi sp,sp,16
   101d4: 00000513 addi a0,0,0
   101d8: 00008067 jalr   ra
```

LUI/ADDI Address Calculation in RISC-V

Target address of <string1> is **0x00020 A10**
Instruction sequence **LUI 0x00020**, **ADDI 0xA10**
doesn't quite work because immediates in RISC-V are sign extended
(and 0xA10 has a 1 in the high order bit)!

0x00020 000 + 0xFFFFF A10 = 0x0001F A10

(Off by 0x00001 000)

So we get the right address if we calculate it as follows:

(0x00020 000 + 0x00001 000) + 0xFFFFF A10 = 0x00020 A10

What is 0xFFFFF A10?

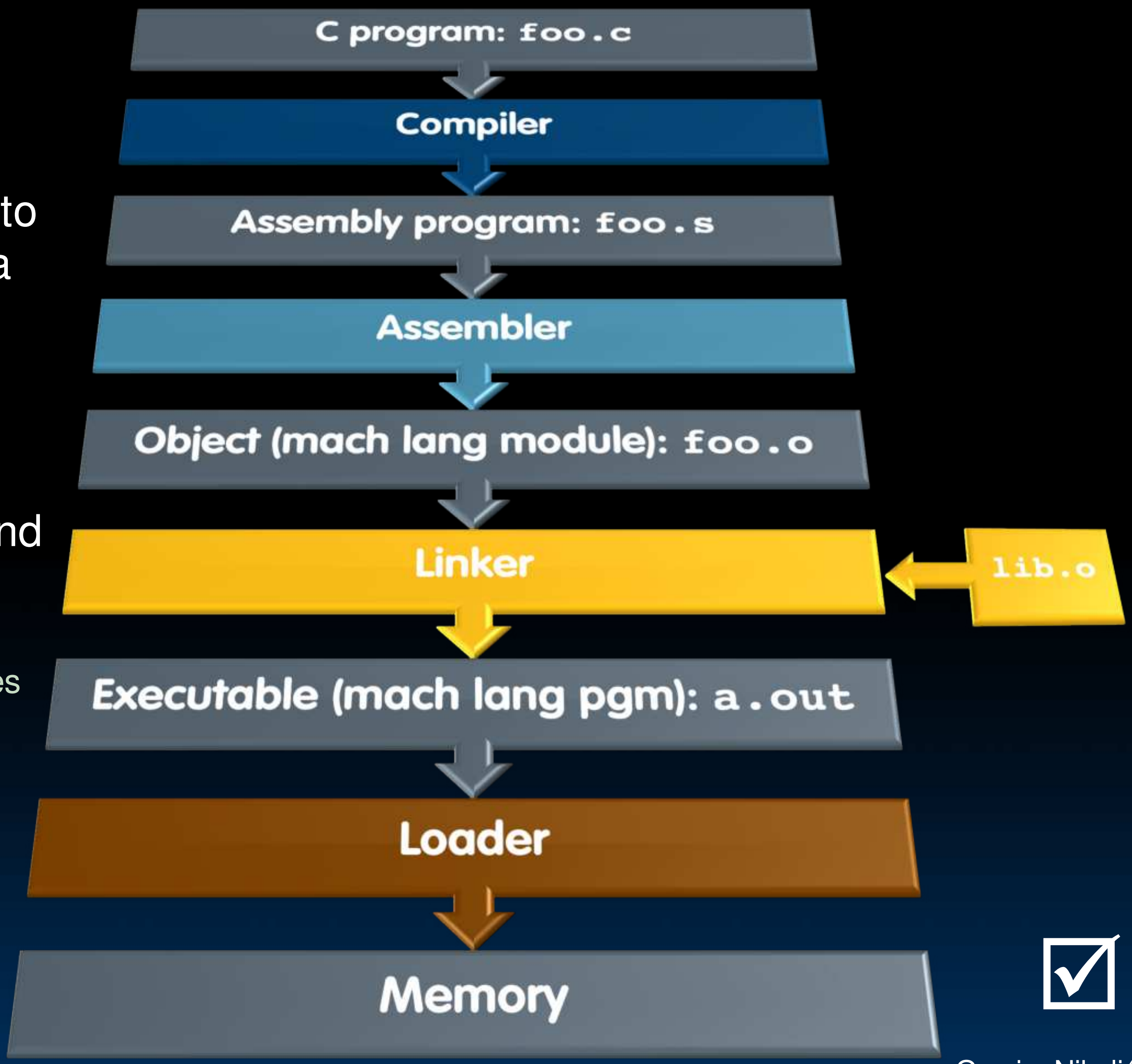
Twos complement of 0xFFFFF A10 = 0x00000 5EF + 1 = 0x00000 5F0 = 1520_{ten}
So 0xFFFFF A10 = -1520_{ten}

Instruction sequence **LUI 0x00021**, **ADDI -1520**
calculates **0x00020 A10**

Garcia, Nikolić

And In Conclusion, …

- 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



Garcia, Nikolić