UC Berkeley
Teaching Professor
Dan Garcia

CS61C

Great Ideas in Computer Architecture

(a.k.a. Machine Structures)

UC Berkeley
Lecturer
Justin Yokota

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

Pseudoinstructions

- **Pseudoinstructions: Convenient variations of instructions understood by the assembler, but not by the machine.**

Pseudoinstruction	Real instruction(s)
`mv rd, rs1`	`addi rd, rs1, 0`
`not rd, rs`	`xori ??, ??, ??`

`li rd, imm`	`# <= 12-bit signed imm`
	`addi ??, ??, ??`
	
	`# >12-bit immediate`
	`lui ??, ??, ??`
	`addi ??, ??, ??`

`j Label`	`jal ??, ??`
`jr rs1`	`jalr ??, ??`
`la rd, label` (address `addr`)	
`call label` (address `addr`)	

What real machine instructions do these pseudoinstructions translate to?

https://cs61c.org/fa22/pdfs/resources/reference-card.pdf

Pseudoinstructions

Check out the green card for the full list of pseudoinstructions!

- **Pseudoinstructions: Convenient variations of instructions understood by the assembler, but not by the machine.**

Pseudoinstruction	Real instruction(s)	
`mv rd, rs1`	`addi rd, rs1, 0`	
`not rd, rs`	`xori rd, rs, -1`	
`li rd, imm`	`# <= 12-bit signed imm`	`# >12-bit immediate`
	`addi rd, x0, imm`	`lui rd, imm[31:12]`
		`addi rd, rd,imm[11:0]`
`j Label`	`jal x0, label`	
`jr rs1`	`jalr x0, rs1`	
`la rd, label` (address `addr`)	`# absolute address`	`# PC-relative address`
	`lui rd, addr[31:12]`	`auipc rd, addr[31:12]`
	`addi rd, rd, addr[11:0]`	`addi rd, rd, addr[11:0]`
`call label` (address `addr`)	`auipc ra, addr[31:12]`	
	`jalr ra, addr[11:0]`	

Compilers often use `call` for `jal far_away`, e.g., in a different file.

Translating and Running a Program: CALL

- Pseudoinstructions
- Translating and Running a Program: CALL
- CALL: Compiler
- CALL: Assembler
- CALL: Linker
- CALL: Loader
- Example: Hello, World!
- Program Translation vs. Interpretation (recorded)

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

How do we translate programs from a high-level language into machine code?

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)

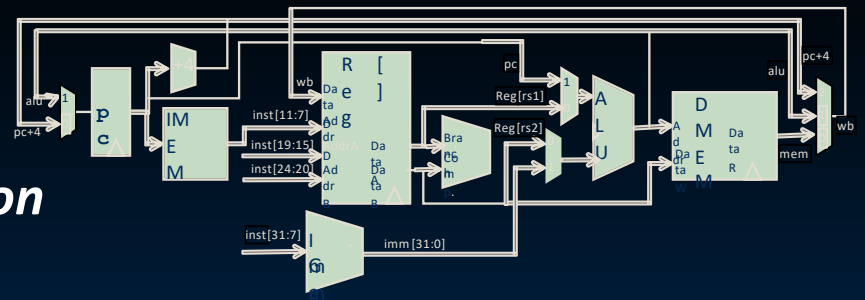
```
lw    x3, 0(x10)
lw    x4, 4(x10)
sw    x4, 0(x10)
sw    x3, 4(x10)
```

Assembler

Machine Language Program (RISC-V)

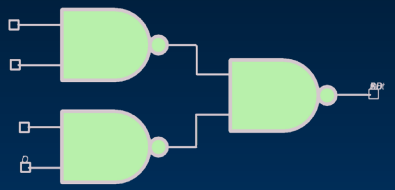
```
1000 1101 1110 0010 0000 0000 0000 0000
1000 1110 0001 0000 0000 0000 0000 0100
1010 1110 0001 0010 0000 0000 0000 0000
1010 1101 1110 0010 0000 0000 0000 0100
```

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



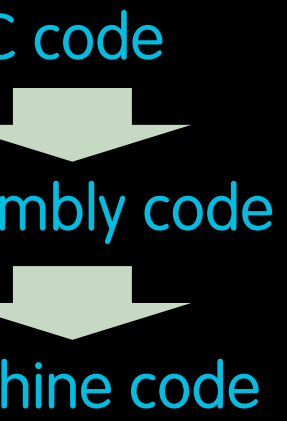
Architecture Implementation

Logic Circuit Description (Circuit Schematic Diagrams)



How Do We Run a C Program?

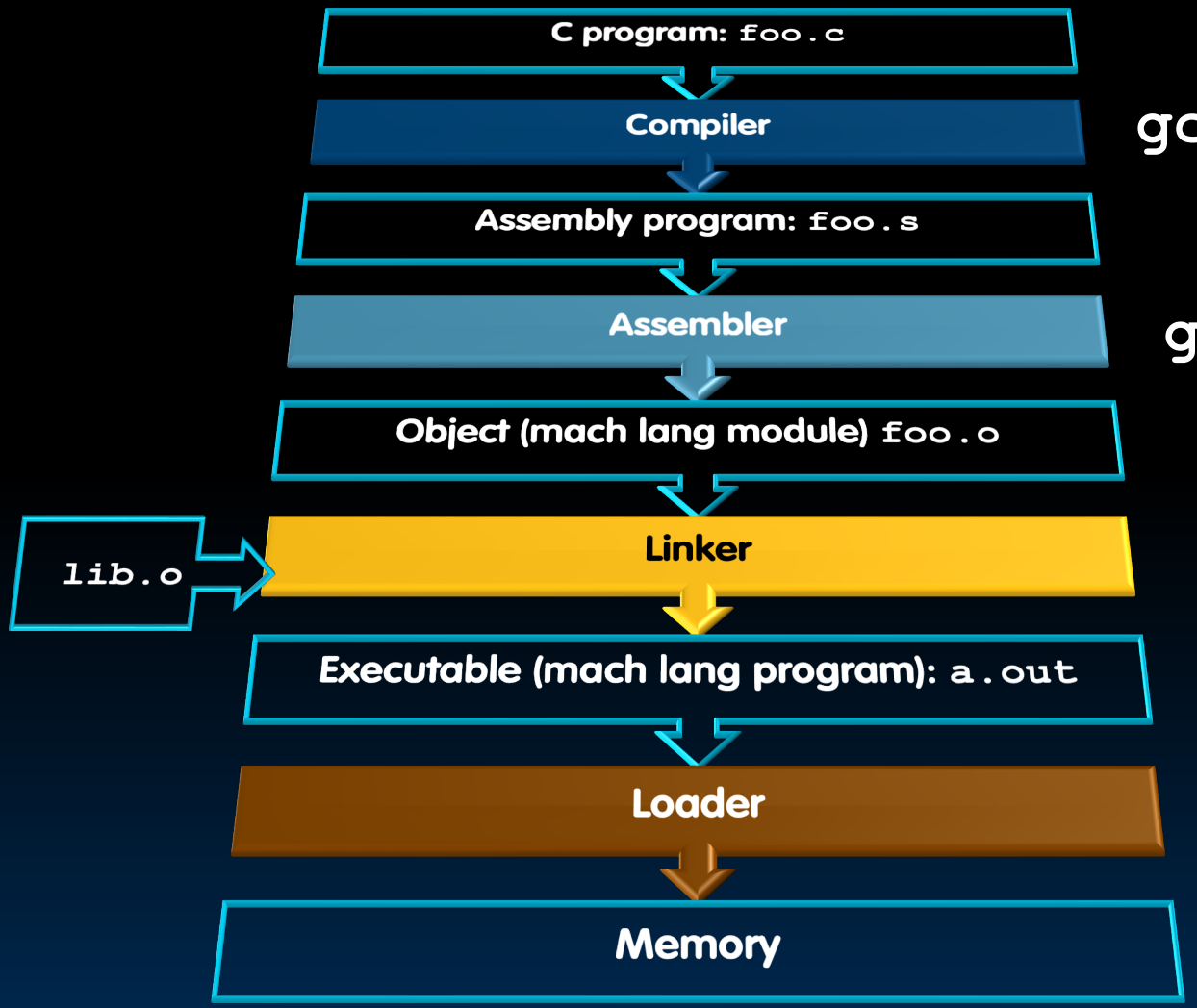
- Translator: **Converts a program from the source language to an equivalent program in another language**
  - Translating/compiling to lower-level languages almost always means higher efficiency, higher performance.



- **Contrast with Interpreter: Directly executes a program in the source language**
  - Note: C programs/RISC-V can also be interpreted!
  - Example: Venus RISC-V simulator useful for learning/debugging

Check out the last recorded section of this lecture, which discusses translation vs. interpretation in more detail!

Steps in Compiling and Running a C Program



(optimize compilation; off: -O0)

```
gcc  -S -O2 foo.c
```

```
gcc  -c foo.s
```

```
gcc foo.o
```

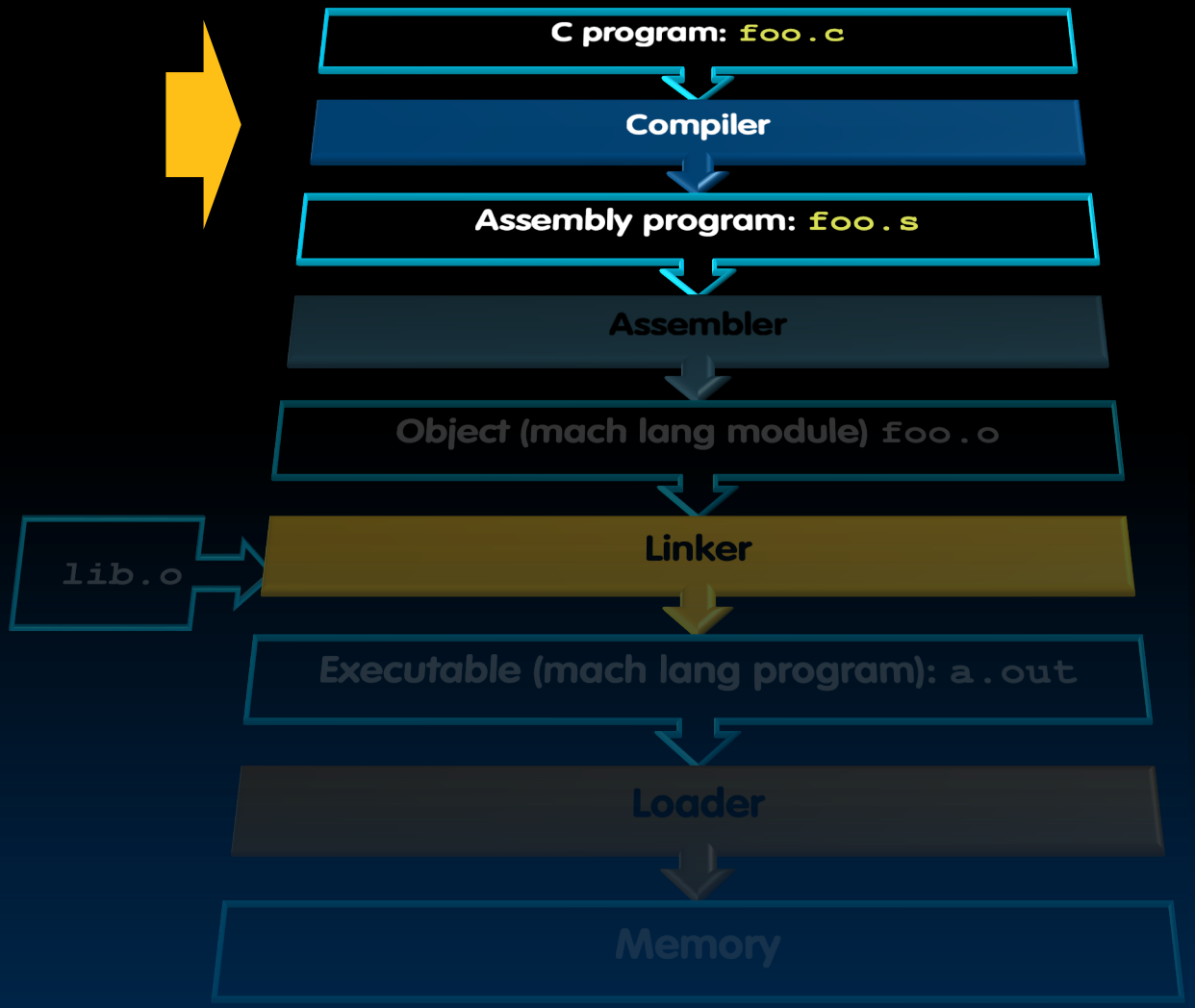
```
./a.out
```

"Compiling C code to binary" generally/colloquially refers Steps 1-3, i.e., translating a C program to an executable.

CALL: Compiler

- Pseudoinstructions
- Translating and Running a Program: CALL
- CALL: Compiler
- CALL: Assembler
- CALL: Linker
- CALL: Loader
- Example: Hello, World!
- Program Translation vs. Interpretation (recorded)

Compiler



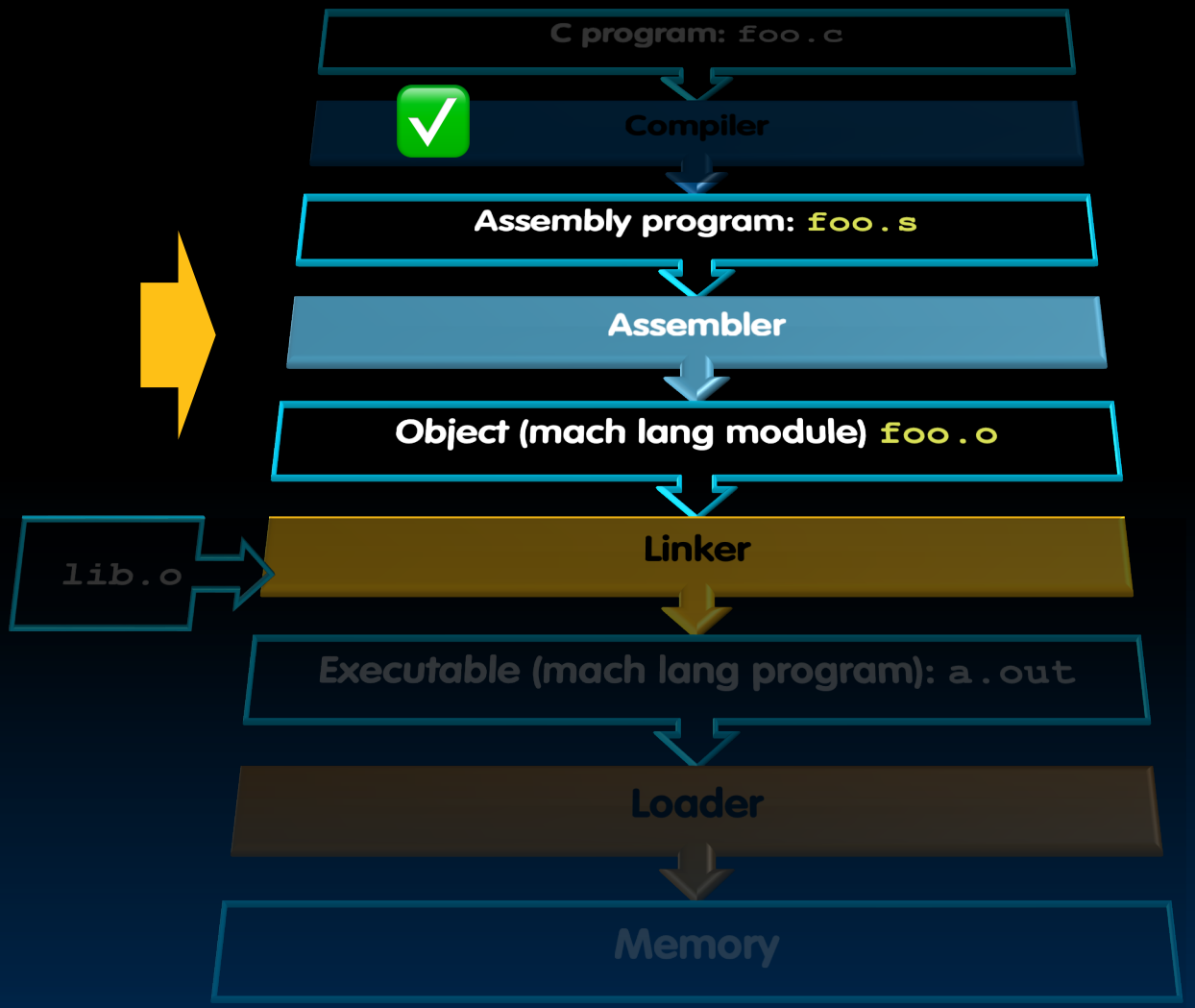
- **Input: High-level Language Code (e.g., `foo.c`)**
- **Output: Assembly Language Code (e.g., `foo.s` for RISC-V)**
- **Note: Output *may* contain pseudoinstructions!**
  - e.g., `mv`, `li`, `call`, `j`, etc.

Take CS164 to build your own compiler!

CALL: Assembler

- Pseudoinstructions
- Translating and Running a Program: CALL
- CALL: Compiler
- CALL: Assembler
- CALL: Linker
- CALL: Loader
- Example: Hello, World!
- Program Translation vs. Interpretation (recorded)

Assembler



- **Input: Assembly Language Code (e.g., `foo.s` for RISC-V)**
  - Includes pseudoinstructions!
- **Output: Machine Language Module,** object file **(e.g., `foo.o` for RISC-V)**
  - Object Code (machine language)
  - Information for linking and debugging
    - Symbol Table, Relocation Information, Data Segment, etc.
- **Reads and uses** directives
- **Replaces pseudoinstructions with true assembly**
  - Then, produce machine language code

Directives

- Directives **give directions to the assembler:**
  - Often generated by the compiler (previous stage)
  - Directives do not produce machine instructions! Rather, they inform how to build different parts of the *object file*.

`.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

Object File Format (1/3)

1. Object File Header: **size and position of other pieces of the object file**
2. Text Segment: **machine code**
3. Data Segment: **binary representation of static data in the source file**
4.
5.
6.

Producing Machine Code

- **Simple case:**
  - Arithmetic, Logical, Shifts, etc.
  - All necessary info is within the instruction already!

```
add x18 x18,x10
addi x19,x19,-1
```

- **PC-Relative Branches and Jumps:**
  - e.g., `beq`/`bne`/etc. and `jal`
  - *Position-Independent Code (PIC)*:
    - Once pseudoinstructions are replaced with real ones, all PC-relative addressing can be computed
  - Determine the offset to encode by counting the number of *half-word* instructions between current instruction and target instruction

```
j Label
```

```
jal x0, Label
```

Computing PC-Relative Addresses: Two-Pass

- **Can we compute all offsets in a single pass? No…**
- **"Forward Reference" problem:**
  - Branches, PC-relative jumps can refer to labels that are "forward" in the program.
  - Precise positive offsets are unknown in the first pass!

```
        addi t2, zero, 9     # t2 = 9;
Loop:   slt  t1, zero, t2    # t1 = (0 < t2);
        beq  t1, zero, Exit  # t2 <= 0; goto Exit
        addi t2, t2, -1      # t2--;
        j Loop               # go back to Loop
Exit:   …
```

3 words back (byte offset –12)

3 words forward (byte offset +12)

- **Instead, take two passes over the program.**
  - Pass 1: Remember positions of labels (store in *symbol table*).
  - Pass 2: Use label positions to generate machine code.

What about Other References?

- **References to other files?**
  - e.g., `call`-ing `strlen` from the C `string` library
- **References to static data?**
  - e.g., `la` gets broken up until `lui` and `addi`
  - These require knowing the full 32-bit address of the data

- **These can't be determined yet, so the Assembler jots them down in two tables:** Relocation Information **and** Symbol Table**.**

Object File Format (2/3)

1. Object File Header: **size and position of other pieces of the object file**
2. Text Segment: **machine code**
3. Data Segment: **binary representation of static data in the source file**
4. Symbol Table: **List of file's labels, static data that can be referenced by other programs**
5. Relocation Information: **Lines of code to fix later (by Linker)**
6.

Symbol Table

- **List of "items" in this file**
- **Instruction** Labels
  - Used to compute machine code for PC-relative addressing in branches, function calling, etc.
  - `.global` directive: labels can be referenced by other files.
- Data: **anything in the `.data` section**
  - Global variables may be accessed/used by other files.

Relocation Information

- List of "items" whose address this file needs
- Any external label jumped to
  - External label (including lib files): `jal ext_label`
- Any piece of data in static section
  - e.g., `la` instruction (for `lw`/`sw` base register)

Object File Format (3/3)

Common Standard Format: ELF (UNIX)
http://www.skyfree.org/linux/references/ELF_Format.pdf

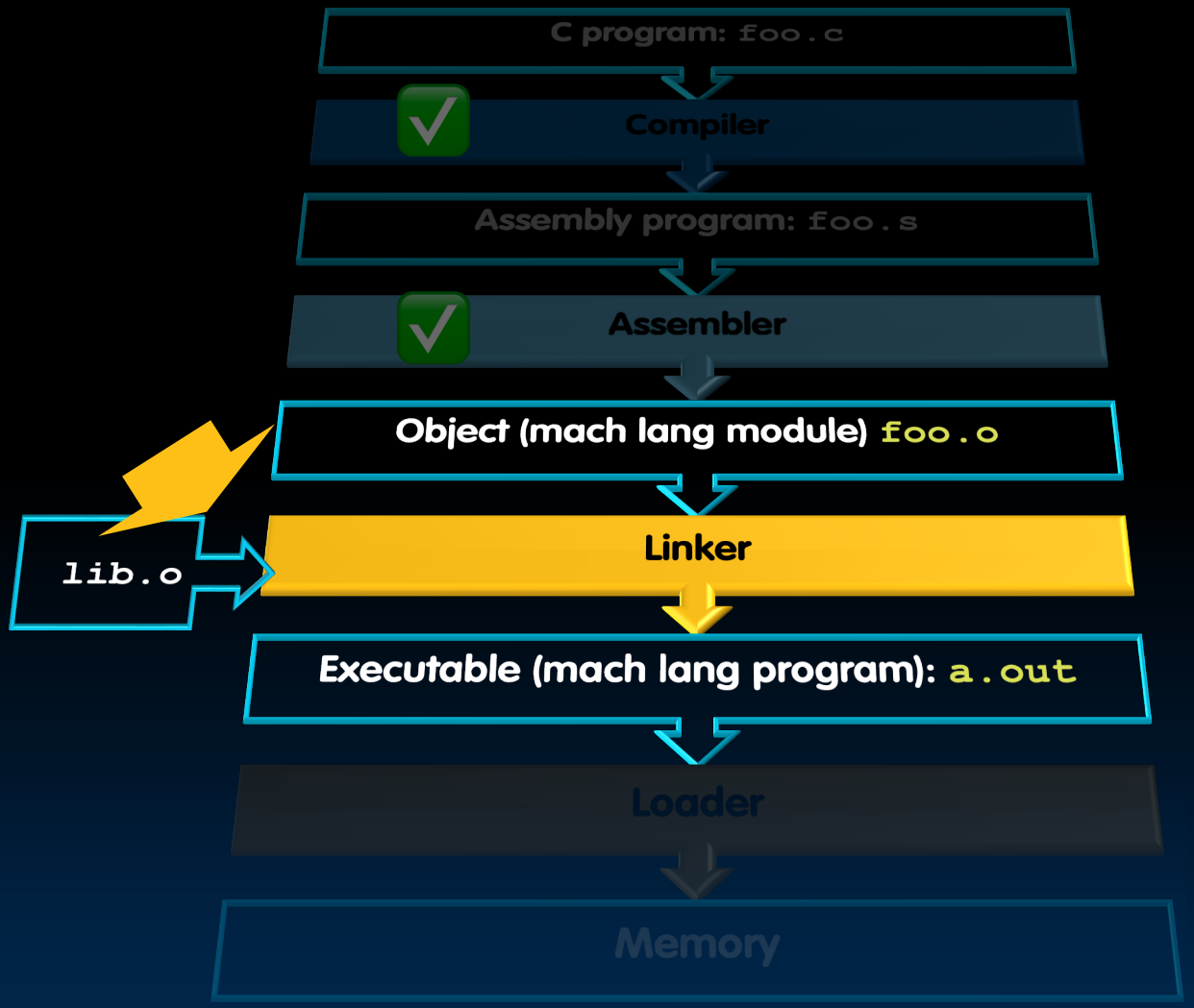
1. Object File Header: **size and position of other pieces of the object file**
2. Text Segment: **machine code**
3. Data Segment: **binary representation of static data in the source file**
4. Symbol Table: **List of file's labels, static data that can be referenced by other programs**
5. Relocation Information: **Lines of code to fix later (by Linker)**
6. Debugging Information

Bonus: check out
`objdump foo.o`
(Note: hive machines use x86-64 ISA)

CALL: Linker

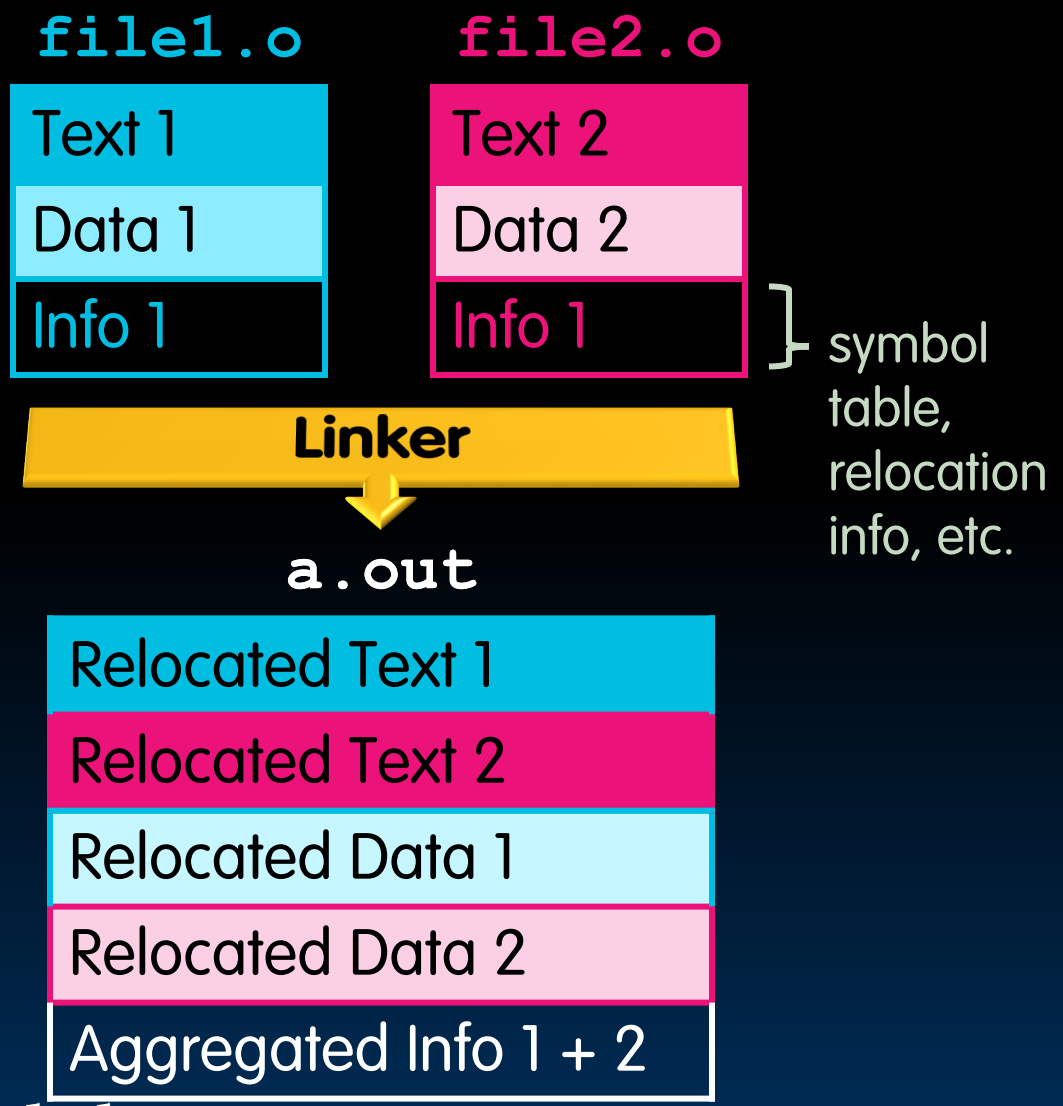
- Pseudoinstructions
- Translating and Running a Program: CALL
- CALL: Compiler
- CALL: Assembler
- CALL: Linker
- CALL: Loader
- Example: Hello, World!
- Program Translation vs. Interpretation (recorded)

Linker



- **Input: Object files (e.g., `foo.o`, `lib.o` for RISC-V)**
  - Text/Data segments, Info Tables per file
- **Output: Executable machine code, (e.g., `a.out` for RISC-V)**
- **The Linker enables separate compilation of files.**
  - Changes to one file does not require recompilation of the entire program.
    - (e.g., Linux source >20M lines of code)
  - Old name: "Link Editor" from editing the "links" in jump-and-link instructions.

Linker Patches Together Multiple Object Modules



1. **Put together text segments from each `.o` file.**
2. **Put together data segments from each `.o` file, then concatenate this onto the end of Step 1's segment.**
3. **Resolve references.**
   - Go through *Relocation Table* and handle each entry, i.e., fill in all *absolute addresses*.

Which Addresses Need Relocating?

- **PC-Relative Addressing**
  `beq`, `bne`, `jal`, `auipc/addi`, etc.
  - Never relocate
    - Position-independent code (PIC)

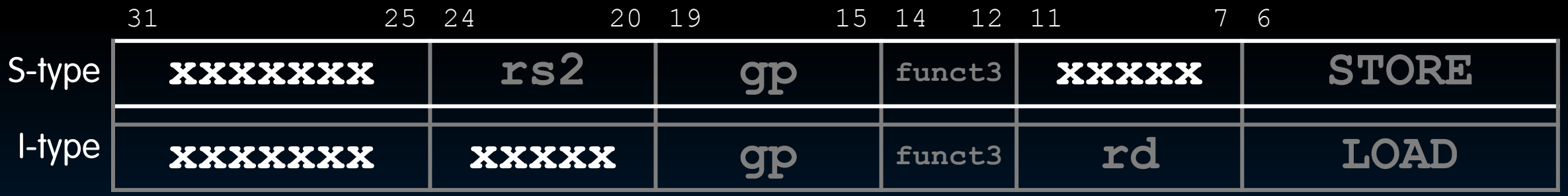
- **External Function Reference**
  usually `jal` or `auipc/jalr`
  - Always relocate
    - Addresses were unknown at time of assembling

- **Static Data Reference**
  `lw`, `sw`, `lui/addi`
  - Always relocate
    - Data segment has relocated

Which Instructions Need Relocation Editing?

- **J-Format (only when jumping externally!)**

- **Loads and stores using `gp` to access `.data` variables**
  - Global pointer (`gp`), is a pointer to the data/static segment.



- **`lui, addi; auipc/jalr` (if jumping externally)**
- **Again, conditional branches (B-Type) don't need editing!**
  - PC-relative addressing *preserved* even if text is relocated

Resolving References (1/2)

- **For RV32, linker assumes first text segment starts at address `0x10000`.** (More later: virtual memory)
- **Linker knows:**
  - Length of each text/data segment
  - Ordering of text and segments
- **Linker calculates:**
  - Absolute address of each label to be jumped to and of each piece of data referenced.

Resolving References (2/2)

- **For RV32, linker assumes first text segment starts at address `0x10000`.** (More later: virtual memory)
- **Linker knows:**
  - Length of each text/data segment
  - Ordering of text and segments
- **Linker calculates:**
  - Absolute address of each label to be jumped to and of each piece of data referenced.
- **Linker resolves 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 determined, fill in machine code appropriately.
- **Linker Output:**
  - Executable containing text and data (+ header/debugging info)

Static vs. Dynamic Linking

- **So far, we've described the traditional way:** statically-linked libraries.
  - The library is now part of the executable – if the library updates, we won't get the fix unless we recompile the user program.
  - The executable includes the entire library, even if not all of it is used by the user program.
  - Pro: The Executable is self-contained!

✅ **The Alternative:** dynamically-linked libraries **(DLL), common on Windows & UNIX Platforms.**

Dynamically-Linked Libraries (DLLs)

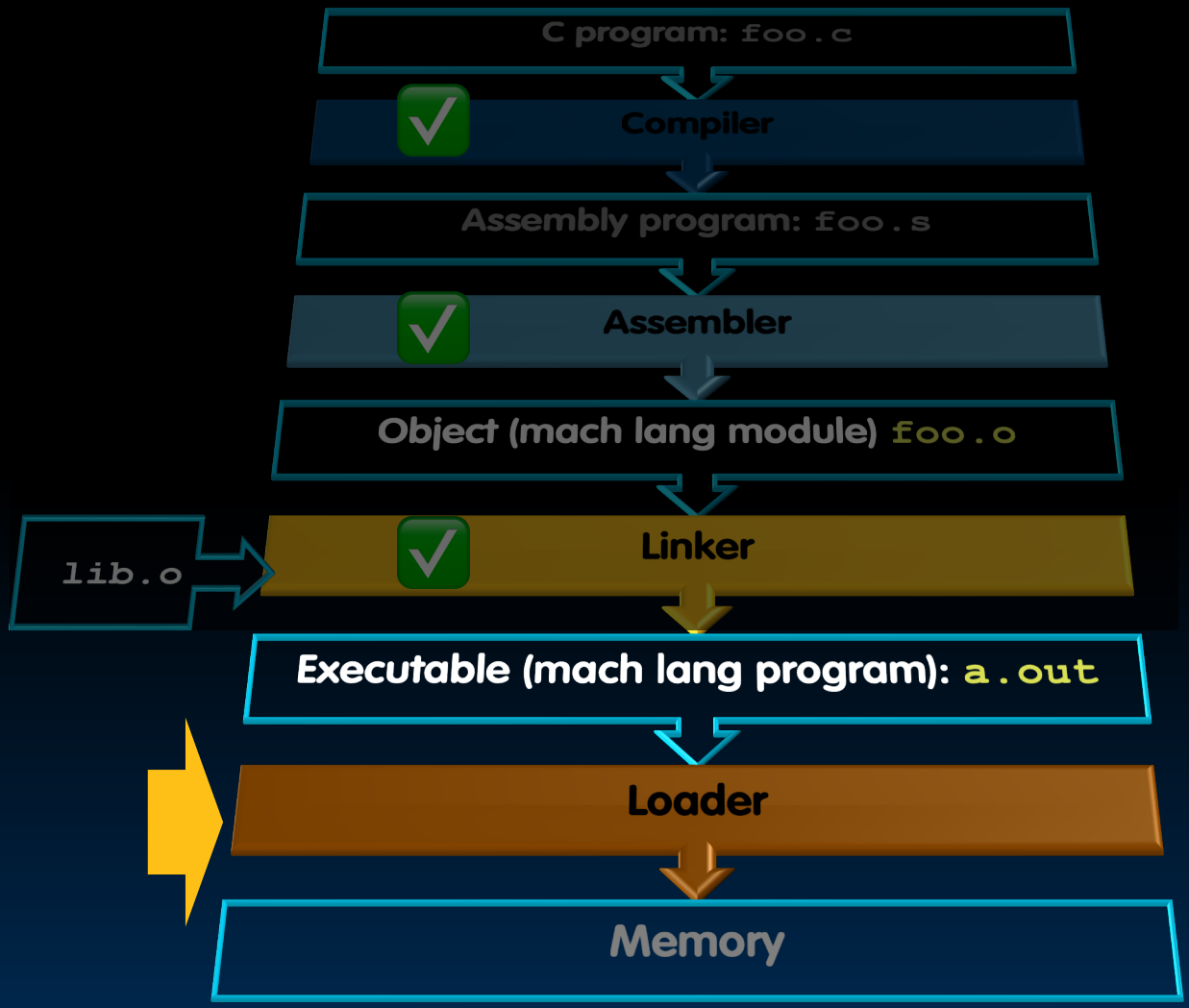
- **Space vs. Time**
  - Storing a program requires less disk space; less time to send a program.
  - Executing two programs requires less memory (if they share a library)
  - ⚠ Runtime overhead: extra time to dynamically link!
- **Reasonable handling of upgrades**
  - Replacing `libXYZ.so` upgrades every program using library `XYZ`.
  - ⚠ Having the program executable isn't enough anymore!
- **Prevailing approach to DLL uses machine code as the "lowest common denominator."**
  - "Linking at the machine code level," i.e., linker does not know what compiler/what language compiled from.
  - Not the only the way to do it…

Overall dynamic linking adds complexity to compiler, linker and OS. However, it provides many benefits that often outweigh its complexities.

CALL: Loader

- Pseudoinstructions
- Translating and Running a Program: CALL
- CALL: Compiler
- CALL: Assembler
- CALL: Linker
- CALL: Loader
- Example: Hello, World!
- Program Translation vs. Interpretation (recorded)

Loader



- Input: Executable Code (e.g., `a.out` for RISC-V)
- Output: (program is run)
- Executable files are stored on disk.
- When an executable is run, loader loads it into memory and start it running.

In reality: Loader is the operating system (OS)!

Loading is one of the OS tasks. Take CS162 for more!

Loader…What Does It Do?

- **Load program into a newly created address space:**
  - Read executable's file header for sizes of text, data segments.
  - Create new address space for program large enough to hold text and data segments, along with a stack segment.
    - Copy instructions, data from executable file into new address space.
    - Copy arguments passed to the program onto the stack.
- **Initialize machine registers**
  - Most registers cleared; stack pointer (`sp`) assigned address of first free stack location
- **Jump to start-up routine, which does the following:**
  - Copy program arguments from stack to registers, set PC
  - If main routine returns, terminate program with exit system call.

Peer Instruction

- At what point in the process are all the machine code bits determined for the following assembly instructions?

1. `add x6, x7, x8`

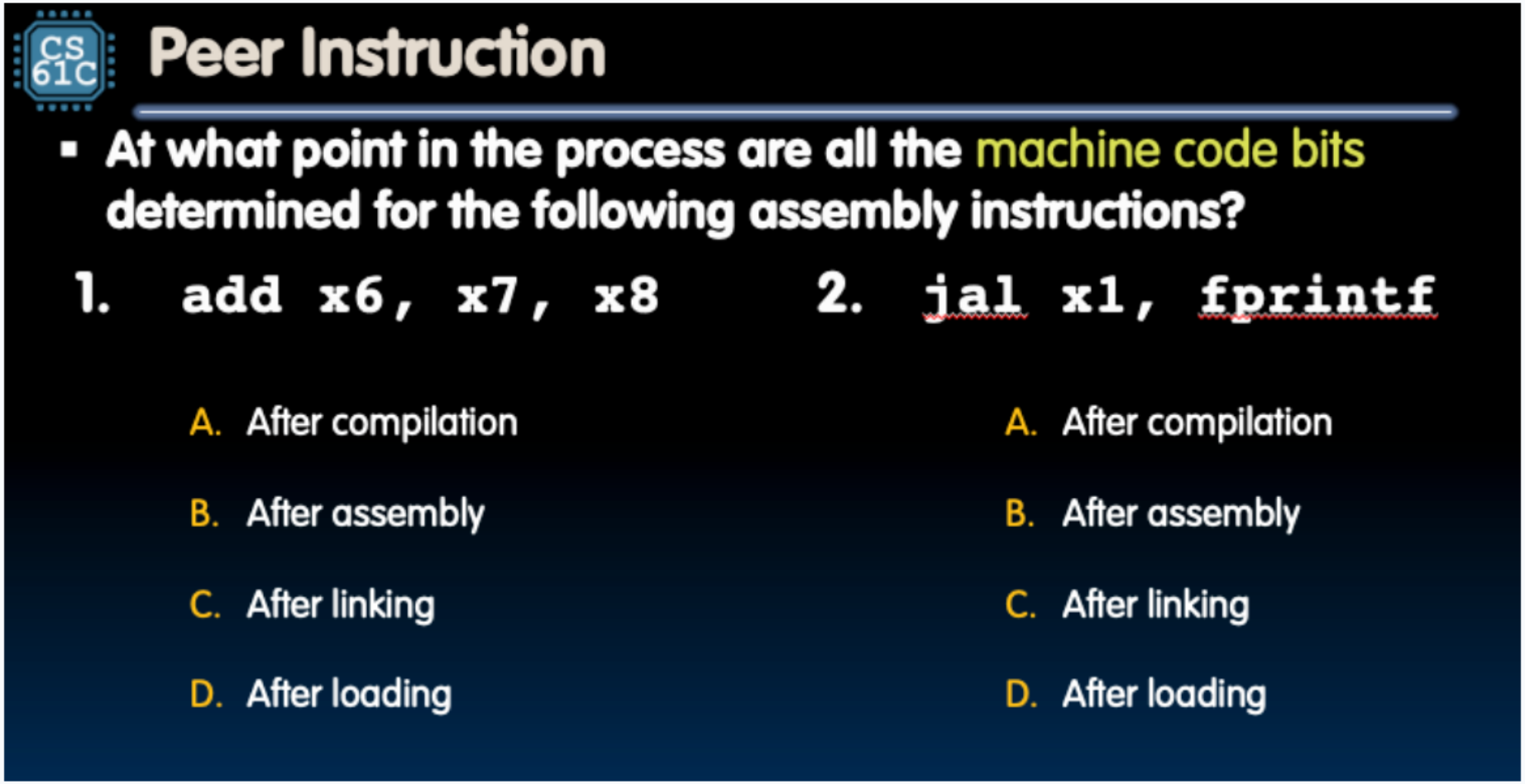
A. After compilation
B. After assembly
C. After linking
D. After loading

2. `jal x1, fprintf`

A. After compilation
B. After assembly
C. After linking
D. After loading

pollev.com/yanl

At what point in the process are all the machine codes determined?



Peer Instruction

- At what point in the process are all the machine code bits determined for the following assembly instructions?

1. `add x6, x7, x8`

A. After compilation
B. After assembly
C. After linking
D. After loading

2. `jal x1, fprintf`

A. After compilation
B. After assembly
C. After linking
D. After loading

Example: Hello, World!

- Pseudoinstructions
- Translating and Running a Program: CALL
- CALL: Compiler
- CALL: Assembler
- CALL: Linker
- CALL: Loader
- Example: Hello, World!
- Program Translation vs. Interpretation (recorded)

Example C Program: hello.c

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

Compile: hello.c → hello.s

```
.text
   .align 2
   .globl main
main:
   addi sp,sp,-4
   sw   ra,0(sp)
   la   a0, str1
   la   a1, str2
   call printf
   lw   ra,0(sp)
   addi sp,sp,4
   li   a0,0
   ret
.section .rodata
   .balign 4
str1:
   .string "Hello, %s!\n"
str2:
   .string "world"
```

Directive: Enter text section
Directive: Align code to 2^2 bytes
Directive: Declare global symbol **main**

Label for start of **main**

Allocate stack frame, save return address

Pseudoinstructions: load addresses of **str1**, **str2**

Pseudoinstruction: call function **printf**

Restore return address, deallocate stack frame

Return with value 0

Directive: Enter read-only data section
Directive: Align data section to 4 bytes

Label for **str1**
Directive: null-terminated string

Label for **str2**
Directive: null-terminated string

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

Assemble: hello.s → hello.o

Text segment

```
00000000 <main>:
0:  ff010113 addi sp,sp,-16
4:  00112623 sw ra,12(sp)
8:  00000537 lui a0,0x0
c:  00050513 addi a0,a0,0
10: 00005b7 lui a1,0x0
14: 00058593 addi a1,a1,0
18: 00000097 auipc ra,0x0
1c: 000080e7 jalr ra,0
20: 00c12083 lw ra,12(sp)
24: 01010113 addi sp,sp,16
28: 00000513 addi a0,a0,0
2c: 00008067 jalr rac
```

la, call pseudo-instructions replaced

Address placeholders

Object file (Assembler output)

```
        … ff010113 00112623 00000537
00050513 000005b7 00058593 00000097
000080e7 00c12083 01010113 00000513
00008067 …
```

Symbol Table

Label	Address (in module)	type
main:	0x00000000	global text
str1:	0x00000000	local data
str2:	0x0000000c	local data

Relocation Information

Address	type	Dependency
0x00000008	lui	%hi(str1)
0x0000000c	addi	%low(str1)
0x00000010	lui	%hi(str2)
…	…	…

Link : 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 # 20a10 <str1>
   101c0: 000215b7 lui  a1,0x21
   101c4: a1c58593 addi a1,a1,-1508 # 20a1c <str2>
   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
```

How did the linker compute these immediates?

`lui`/`addi` address calculation, redux

```
101b8: 00021537 lui  a0,0x21
101bc: a1050513 addi a0,a0,-1520 # 20a10 <str1>
```

- **Recall that `addi` extends sign. To create the 32-bit immediate `0x20A10` (address of `str1`):**
  - The lower 12-bit `addi` immediate has negative sign bit.
  - *Add 1* to the upper 20-bit `lui` immediate.

```
  0x00021000
+ 0xFFFFFA10
```

```
  0x00021          0x000
+ 0xFFFFF -1     + 0xA10
  0x00020          0xA10
```

concat

`0x00020A10`

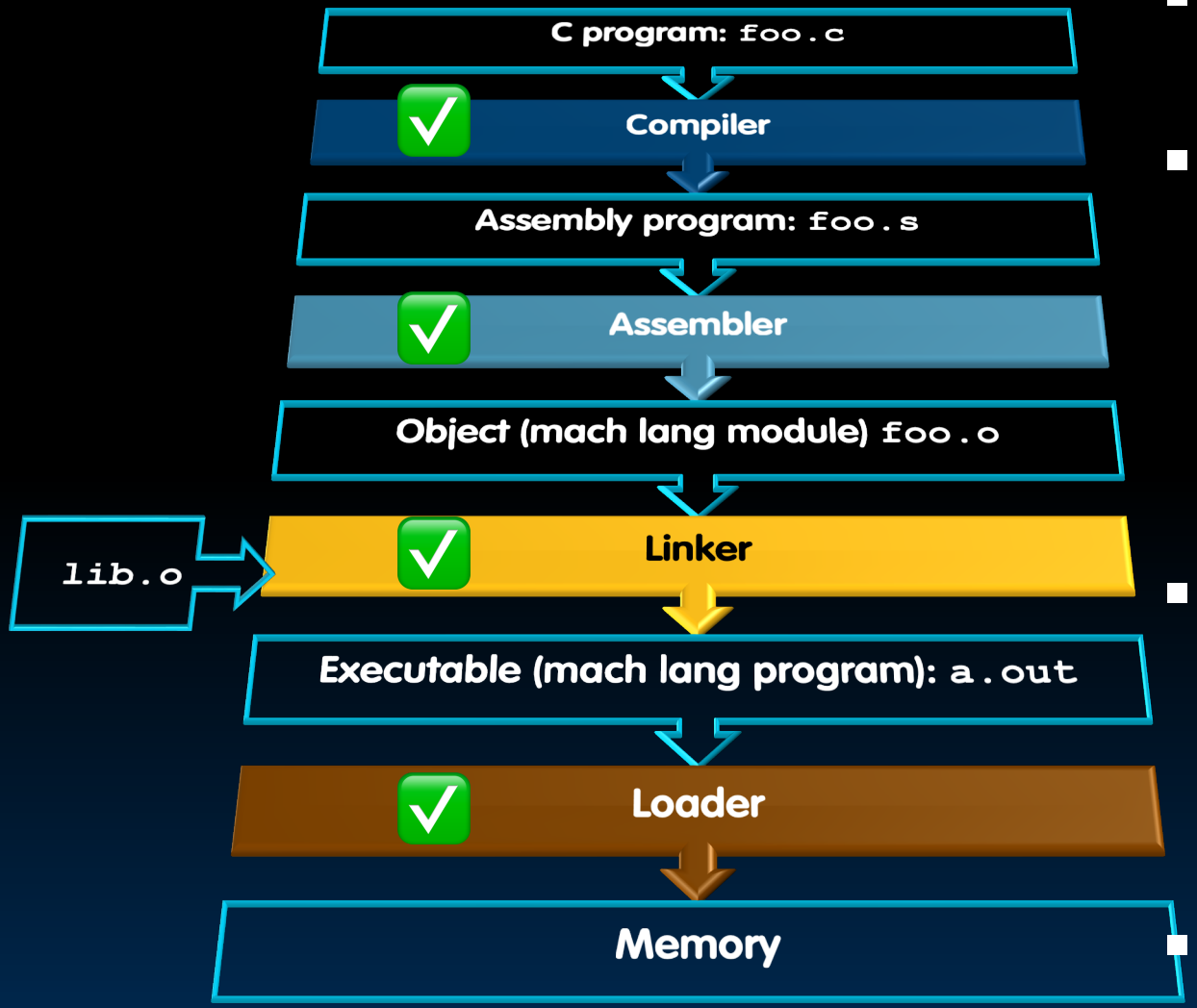
- **Lower 12-bit immediate -1520:**

Two's complement of `0xFFFFFA10`:

$$\texttt{0x000005EF} + 1 = \texttt{0x5F0} = 1520_{ten}$$

So $\texttt{0xFFFFFA10} = -1520_{ten}$

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, creates checklist for the linker (relocation table).
  - 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.

Program Translation vs Interpretation (Recorded)

Recording:
https://youtu.be/DFEe0uRAm7o

- Pseudoinstructions
- Translating and Running a Program: CALL
- CALL: Compiler
- CALL: Assembler
- CALL: Linker
- CALL: Loader
- Example: Hello, World!
- Program Translation vs. Interpretation (recorded)

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

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

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

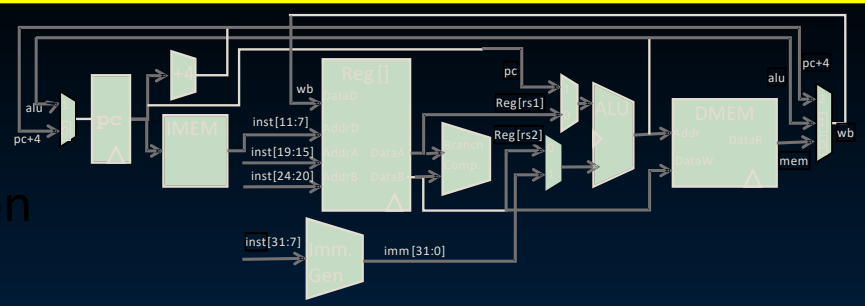
```
lw      x3, 0(x10)
lw      x4, 4(x10)
sw      x4, 0(x10)
sw      x3, 4(x10)
```

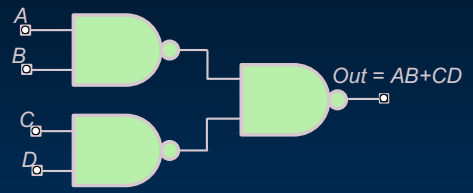
Machine Language Program (RISC-V)

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

Architecture Implementation

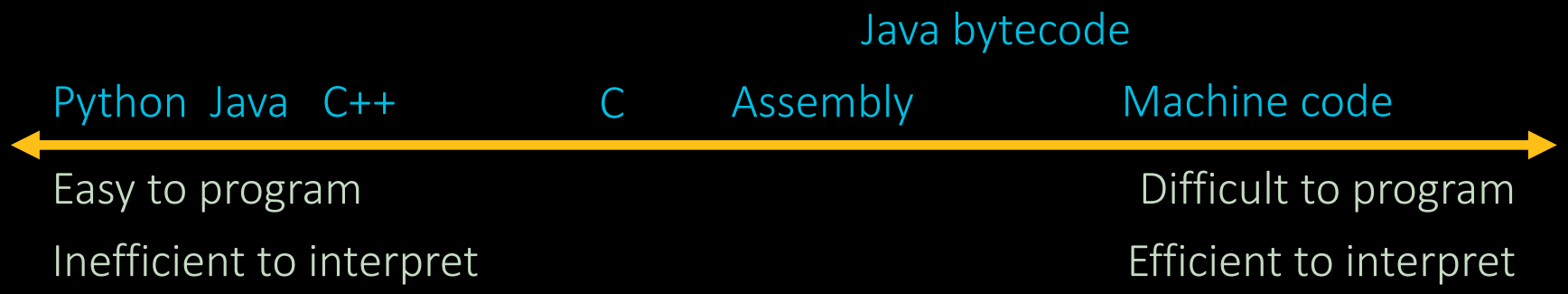
Logic Circuit Description (Circuit Schematic Diagrams)





Language Execution Continuum

Interpreter is a program that executes other programs



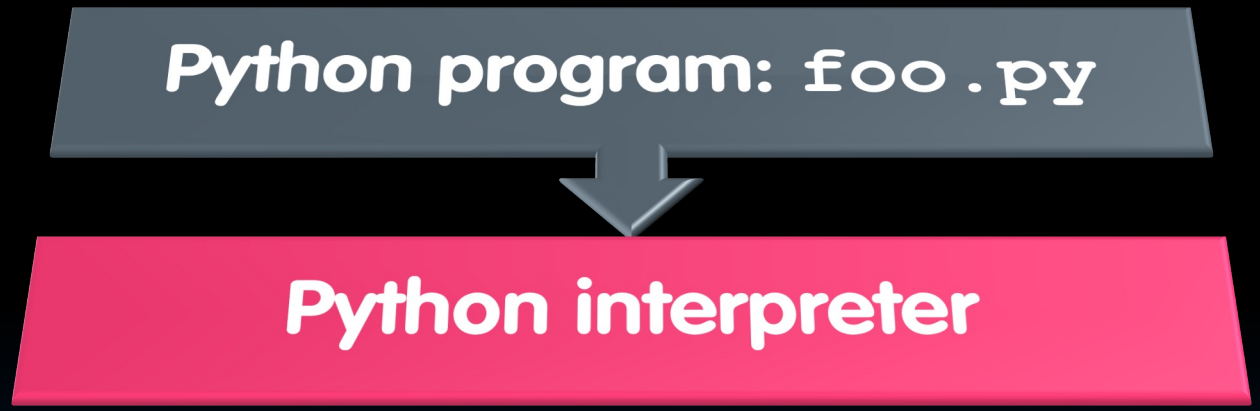
- **Language translation gives us another option**
- **When to choose? In general, we**
  - interpret a high-level language when efficiency is not critical
  - translate to a lower-level language to increase performance

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

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

Interpretation (2/2)

- WHY interpret machine language in software?
- E.g., VENUS RISC-V simulator useful for learning/debugging
- E.g., 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)

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.