

# CS61C: Great Ideas in Computer Architecture (aka Machine Structures)

Lecture 9: CALL (Compiler, Assembler, Linker, Loader)

Instructors: Rosalie Fang, **Charles Hong**, Jero Wang

# Announcements

- Topic-specific exam prep sections begin.
  - RISC-V: Thursday (today!) and Friday this week, 5-7pm in Cory 540AB (same content in both sections)
- Project 2 part A due Friday, July 7th, 11:59 PM PT

## Last Time

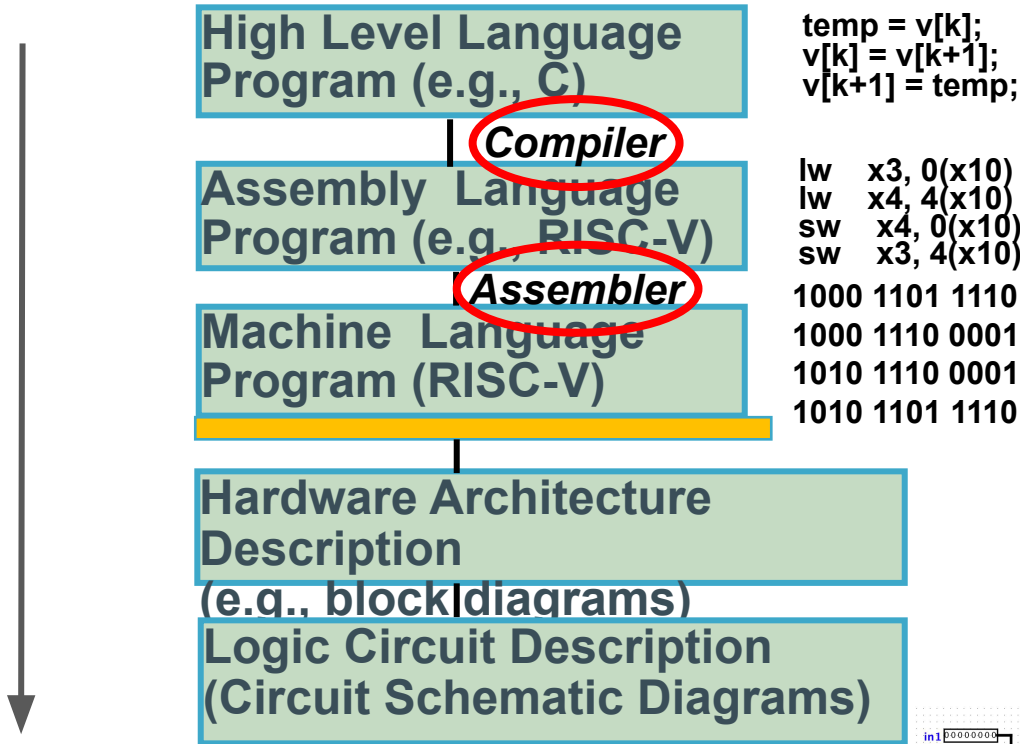
- Calling Convention (cont.)
- Translating RISC-V to binary

# Agenda

- Translation, Compilation, and Interpretation
- CALL process
  - C Pre-Processing
  - Compilation
  - Assembly
  - Linking
  - Loading

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

CS61C

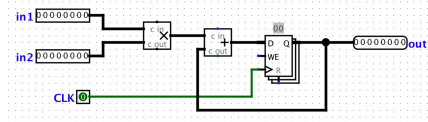
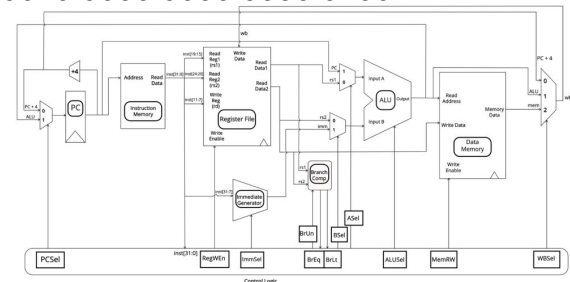


```
temp = v[k];  
v[k] = v[k+1];  
v[k+1] = temp;
```

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

```
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
```

Anything can be represented  
as a number,  
i.e., data or instructions



## So far...

- How do we run a program written in a source language?
  - So far, we've looked at C, RISC-V, and RISC-V but in binary
  - Each of these levels have brought us closer to something a machine can understand
    - C: effectively English, compiled into RISC-V
    - RISC-V: slightly less clear English but still readable, translated into RISC-V machine code
    - RISC-V machine code: 0s and 1s, readable by humans if we put a lot of effort and time into it, but is in theory, manageable by the system!
    - Why in theory?
      - Need to handle multiple files, libraries, addresses, etc.

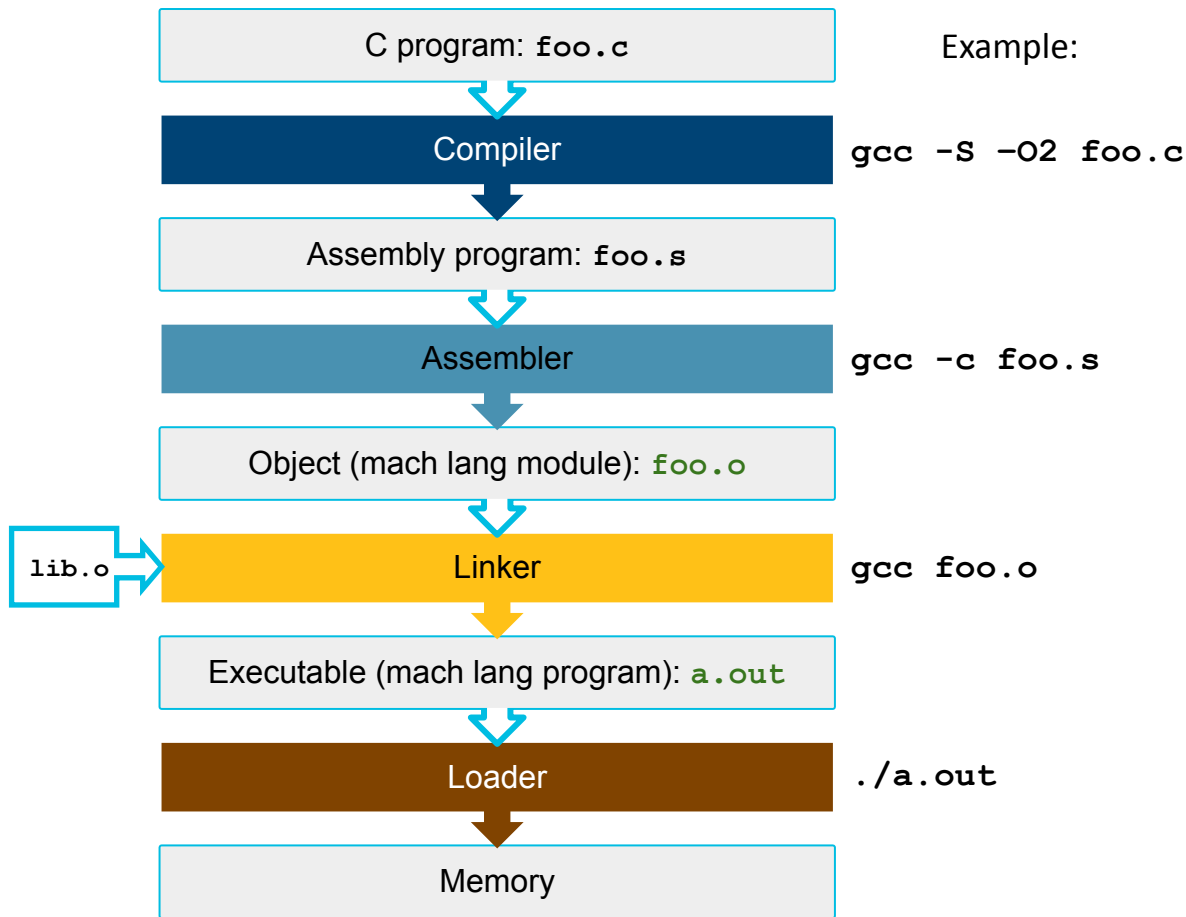
# Overview

Compilation/Compiler

Assembling/Assembler

Linking/Linker

Loading/Loader



C: Compiler

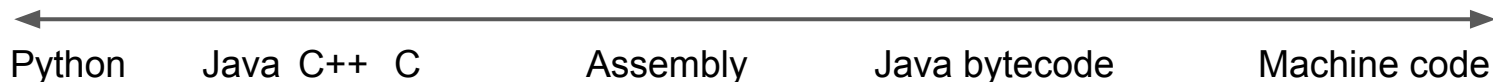


# Language Execution Continuum

- **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
  - Compile to a lower-level language to increase performance

Easy to program  
Inefficient to interpret

Difficult to program  
Efficient to interpret

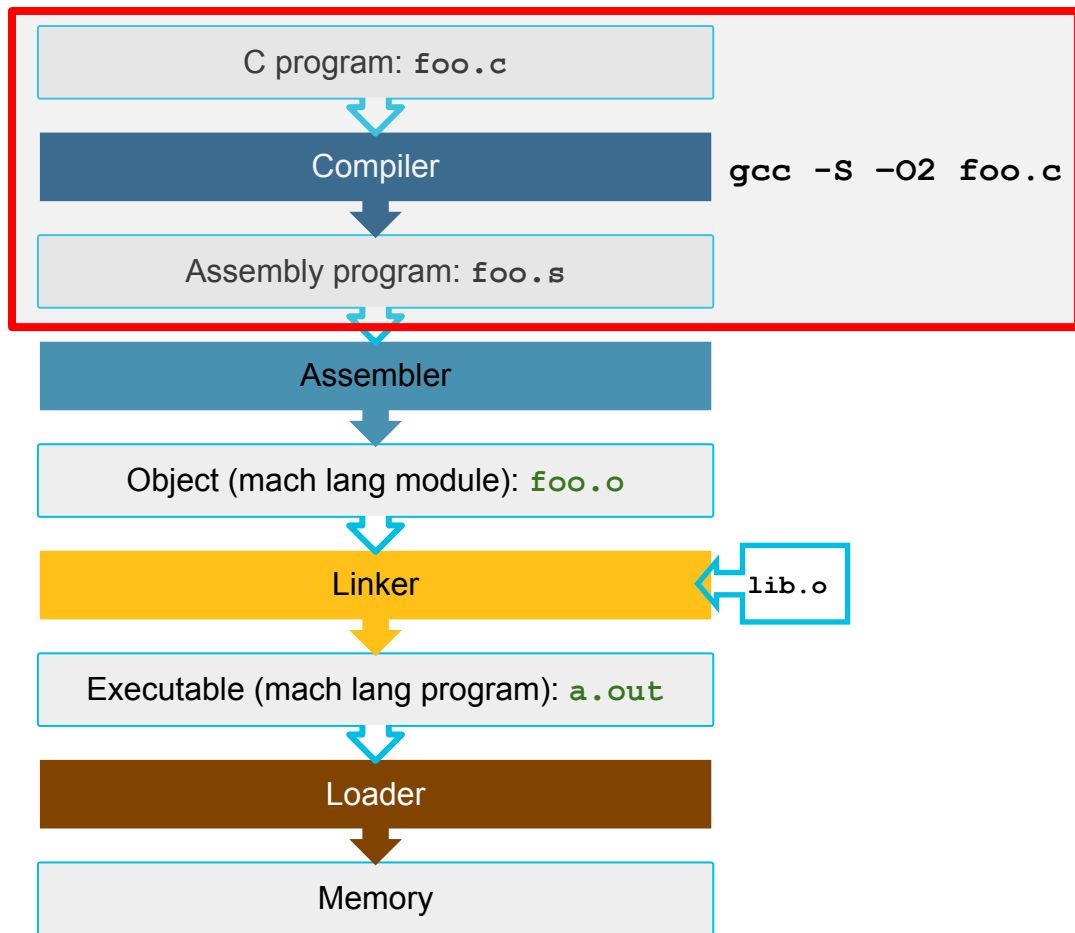


# Compiler/Compilation

NOTE: **Preprocessor** runs before(ish) compiler and handles directives such as:

- `#define` statements
- `#include` statements
- Conditional statements (`#if`, `#endif`, `#ifdef`, `#else`, etc...)
- Macros (objects/data, functions, etc...)

**Output:** pure C file

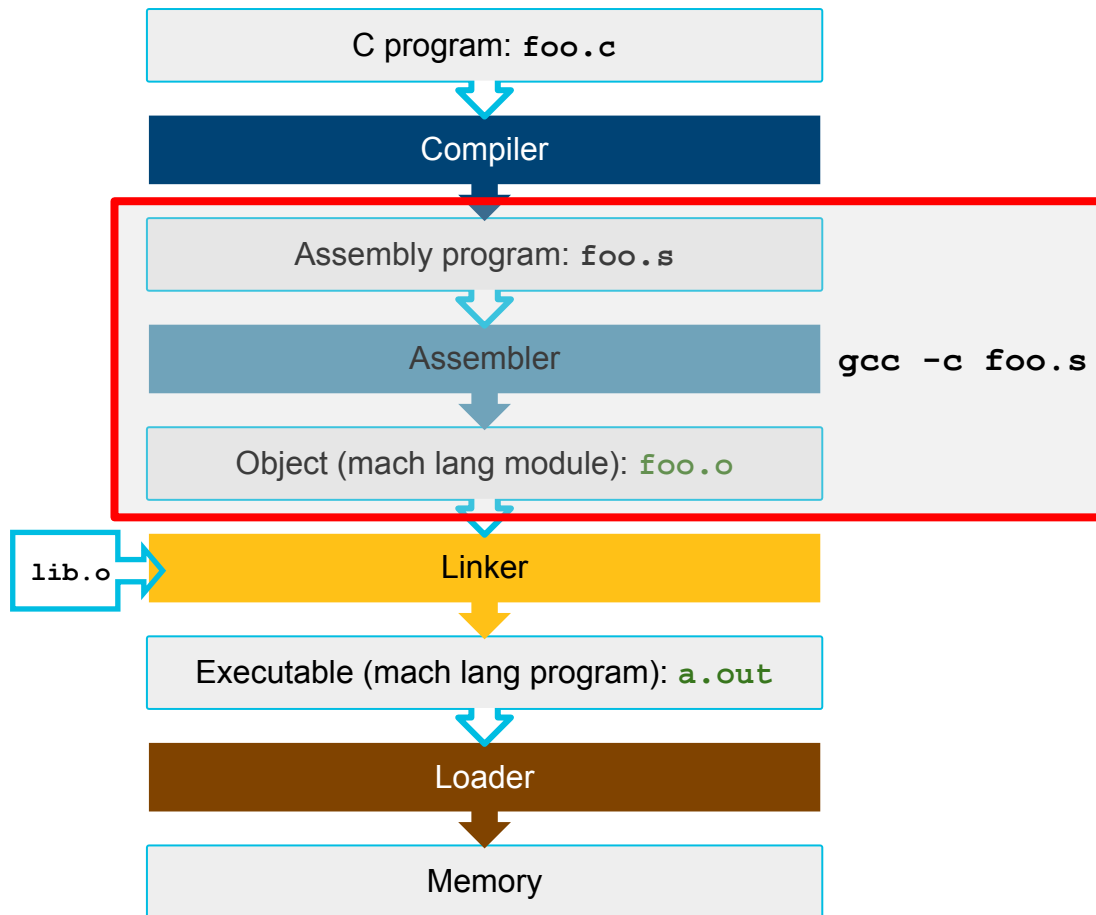


# Compiler/Compilation

- Input/Output: C/low-level language  $\Rightarrow$  RISC-V/other assembly language
- Optimization occurs at this stage
  - Bottleneck for compilation difficulties
  - CS 164!
- Output will have:
  - Labels
  - Pseudoinstructions
  - Fully relative addressing

A: Assembler

# Assembler/Assembly



# Assembler/Assembly

- Input/Output: Optimised RISC-V/assembly (\*.s)  $\Rightarrow$  object file (\*.o)

# \*.o (object) files

- File header
  - Size + position of later components
- Text segment
  - Machine code
- Data segment
  - Binary representation of all static data
- Symbol table
  - Filled with labels and their relative addresses
- Relocation table
  - Filled with values that need to be located (both internally and externally to native file)
- Debugging information

# Assembler Directives

- These are keywords of sorts, that give directions to assembler, but do not produce any machine instructions (binary output in the object file)
- `.text`: subsequent items put in user text segment (machine code)
  - This usually encompasses all of the code and will be the last segment of most assembly files
- `.data`: subsequent items put in user data segment (source file data in binary)
  - This is where static and global variables across the source file are usually defined
- `.globl sym`: Declares symbol global across multiple files 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
- More directives can be found defined [here](https://www.rowleydownload.co.uk/arm/documentation/gnu/as/RISC_002dV_002dDirectives.html)
  - [https://www.rowleydownload.co.uk/arm/documentation/gnu/as/RISC\\_002dV\\_002dDirectives.html](https://www.rowleydownload.co.uk/arm/documentation/gnu/as/RISC_002dV_002dDirectives.html)



# Symbol Table

- List of “items” in this file that may be used by the file that generated it, or other files
- Native file uses it to replace forward references
  - **Forward references:** labels that are used by an instruction before it's been defined in the code
- Other files use it to replace their own external references
  - **External references:** labels that are used by an instruction that are **not** defined in that same file
  - External references in C could constitute calling functions that need to import a predefined C header file or your own definitions
    - `#include <stdio.h>`
    - `#include "yourown.h"`
- What generally goes into a symbol table?
  - Labels: function calling
  - Data: anything in the .data section
  - Variables which may be accessed across files

# Relocation Table

- List of “items” whose address this file needs, including labels that may've been defined locally or externally
- What do these labels/items include?
  - Any absolute label jumped to: jal, jalr
  - Internal labels sometimes
  - External labels always (including lib files)
  - la instructions, when defined by LUI
  - Any piece of data in static section
- Information used during linking stage

# 2-Pass Assembler Example

file\_a.s

func\_name:

addi sp, sp, -4

beq a0, x0, loop

# <8 instructions here>

jal ra, malloc

loop: bneq t0, x0, done

# <3 instructions here>

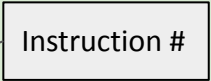
j loop

done: ret

Where are the label definitions?

Where are the label references?

## 2-Pass Assembler Example



```
file_a.s
0 func_name:
(0)    addi sp, sp, -4
1      beq a0, x0, loop
...    # <8 instructions here>
10     jal ra, malloc
11 loop: bneq t0, x0, done
...    # <3 instructions here>
15     j loop
16 done: ret
```

Where are the label definitions?

Where are the label references?

## 2-Pass Assembler Example

file\_a.s

```
0 func_name:
(0)    addi sp, sp, -4
1      beq a0, x0, loop
      # <8 instructions here>

10     jal ra, malloc
11 loop: bneq t0, x0, done
      # <3 instructions here>

15     j loop
16 done: ret
```

Where are the label definitions?

func\_name  $\Rightarrow$  instr. 0

loop  $\Rightarrow$  instr. 11

done  $\Rightarrow$  instr. 16

Where are the label references?

## 2-Pass Assembler Example

file\_a.s

```
0 func_name:
(0)    addi sp, sp, -4
1      beq a0, x0, loop
      # <8 instructions here>

10     jal ra, malloc

11 loop: bneq t0, x0, done
      # <3 instructions here>

15     j loop

16 done: ret
```

Where are the label definitions?

func\_name  $\Rightarrow$  instr. 0

loop  $\Rightarrow$  instr. 11

done  $\Rightarrow$  instr. 16

Where are the label references?

loop  $\Rightarrow$  instr. 1      done  $\Rightarrow$  instr. 11

malloc  $\Rightarrow$  instr. 10      loop  $\Rightarrow$  instr. 15

# 2-Pass Assembler Example

```
file_a.s
0 func_name:
    addi sp, sp, -4
1    beq a0, x0, loop
    # <8 instructions here>
10   jal ra, malloc
11 loop: bneq t0, x0, done
    # <3 instructions here>
15   j loop
16 done: ret
```

Symbol Table for File A

Label Name	Relative Addressing

Relocation Table for File A

Label Name	State

# 2-Pass Assembler Example

```
file_a.s
0 func_name:
    addi sp, sp, -4
1    beq a0, x0, loop
    # <8 instructions here>
10   jal ra, malloc
11 loop: bneq t0, x0, done
    # <3 instructions here>
15   j loop
16 done: ret
```

## First Pass

Symbol Table for File A

Label Name	Relative Addressing
func_name	fa_start

Relocation Table for File A

Label Name	State



## 2-Pass Assembler Example

file\_a.s

```
0 func_name:
    addi sp, sp, -4
1    beq a0, x0, loop
    # <8 instructions here>
10   jal ra, malloc
11 loop: bneq t0, x0, done
    # <3 instructions here>
15   j loop
16 done: ret
```

### First Pass

#### Symbol Table for File A

Label Name	Relative Addressing
func_name	fa_start

#### Relocation Table for File A

Label Name	State
loop	?

## 2-Pass Assembler Example

file\_a.s

```
0 func_name:
    addi sp, sp, -4
1    beq a0, x0, loop
    # <8 instructions here>
10   jal ra, malloc
11 loop: bneq t0, x0, done
    # <3 instructions here>
15   j loop
16 done: ret
```

### First Pass

#### Symbol Table for File A

Label Name	Relative Addressing
func_name	fa_start

#### Relocation Table for File A

Label Name	State
loop	?

## 2-Pass Assembler Example

file\_a.s

```
0 func_name:
    addi sp, sp, -4
1    beq a0, x0, loop
    # <8 instructions here>
10   jal ra, malloc
11 loop: bneq t0, x0, done
    # <3 instructions here>
15   j loop
16 done: ret
```

### First Pass

#### Symbol Table for File A

Label Name	Relative Addressing
func_name	fa_start

#### Relocation Table for File A

Label Name	State
loop	?
<b>malloc</b>	<b>?</b>

## 2-Pass Assembler Example

file\_a.s

```
0 func_name:
    addi sp, sp, -4
1    beq a0, x0, loop
    # <8 instructions here>
10   jal ra, malloc
11 loop: bneq t0, x0, done
    # <3 instructions here>
15   j loop
16 done: ret
```

### First Pass

#### Symbol Table for File A

Label Name	Relative Addressing
func_name	fa_start
loop	fa_start + (4 * 11)

#### Relocation Table for File A

Label Name	State
loop	?
malloc	?

## 2-Pass Assembler Example

file\_a.s

```
0 func_name:
    addi sp, sp, -4
1    beq a0, x0, loop
    # <8 instructions here>
10   jal ra, malloc
11 loop: bneq t0, x0, done
    # <3 instructions here>
15   j loop
16 done: ret
```

### First Pass

#### Symbol Table for File A

Label Name	Relative Addressing
func_name	fa_start
loop	fa_start + (4 * 11)

#### Relocation Table for File A

Label Name	State
loop	?
malloc	?
<b>done</b>	<b>?</b>

## 2-Pass Assembler Example

file\_a.s

```
0 func_name:
    addi sp, sp, -4
1    beq a0, x0, loop
    # <8 instructions here>

10    jal ra, malloc
11 loop: bneq t0, x0, done
    # <3 instructions here>

15    j loop
16 done: ret
```

### First Pass

#### Symbol Table for File A

Label Name	Relative Addressing
func_name	fa_start
loop	fa_start + (4 * 11)

#### Relocation Table for File A

Label Name	State
loop	?
malloc	?
done	?

## 2-Pass Assembler Example

file\_a.s

```
0 func_name:
    addi sp, sp, -4
1    beq a0, x0, loop
    # <8 instructions here>

10    jal ra, malloc
11 loop: bneq t0, x0, done
    # <3 instructions here>

15    j loop
16 done: ret
```

### First Pass

#### Symbol Table for File A

Label Name	Relative Addressing
func_name	fa_start
loop	fa_start + (4 * 11)

#### Relocation Table for File A

Label Name	State
loop	?
malloc	?
done	?

## 2-Pass Assembler Example

file\_a.s

```
0 func_name:
    addi sp, sp, -4
1    beq a0, x0, loop
    # <8 instructions here>
10   jal ra, malloc
11 loop: bneq t0, x0, done
    # <3 instructions here>
15   j loop fa_start + 44
16 done: ret
```

### First Pass

#### Symbol Table for File A

Label Name	Relative Addressing
func_name	fa_start
loop	fa_start + (4 * 11)

#### Relocation Table for File A

Label Name	State
loop	?
malloc	?
done	?



## 2-Pass Assembler Example

file\_a.s

```
0 func_name:
    addi sp, sp, -4
1    beq a0, x0, loop
    # <8 instructions here>
10   jal ra, malloc
11 loop: bneq t0, x0, done
    # <3 instructions here>
15   j loop
16 done: ret
```

### First Pass

#### Symbol Table for File A

Label Name	Relative Addressing
func_name	fa_start
loop	fa_start + 44
done	fa_start + (4 * 16)

#### Relocation Table for File A

Label Name	State
loop	?
malloc	?
done	?

# 2-Pass Assembler Example

```
file_a.s
0 func_name:
    addi sp, sp, -4
1    beq a0, x0, loop
    # <8 instructions here>
10   jal ra, malloc
11 loop: bneq t0, x0, done
    # <3 instructions here>
15   j loop
16 done: ret
```

## Second Pass

Symbol Table for File A

Label Name	Relative Addressing
func_name	fa_start
loop	fa_start + 44
done	fa_start + (4 * 16)

Relocation Table for File A

Label Name	State
loop	?
malloc	?
done	?

## 2-Pass Assembler Example

file\_a.s

```
0 func_name:
    addi sp, sp, -4
1    beq a0, x0, loop
    # <8 instructions here>

10    jal ra, malloc

11 loop: bneq t0, x0, done
    # <3 instructions here>

15    j loop

16 done: ret
```

### Second Pass

#### Symbol Table for File A

Label Name	Relative Addressing
func_name	fa_start
loop	fa_start + 44
done	fa_start + (4 * 16)

#### Relocation Table for File A

Label Name	State
loop	<b>fa_start + 44</b>
malloc	?
done	?

## 2-Pass Assembler Example

file\_a.s

```
0 func_name:
    addi sp, sp, -4
1    beq a0, x0, loop
    # <8 instructions here>
10   jal ra, malloc (?)
11 loop: bneq t0, x0, done
    # <3 instructions here>
15   j loop
16 done: ret
```

### Second Pass

#### Symbol Table for File A

Label Name	Relative Addressing
func_name	fa_start
loop	fa_start + 44
done	fa_start + (4 * 16)

#### Relocation Table for File A

Label Name	State
loop	fa_start + 44
malloc	?
done	?

## 2-Pass Assembler Example

file\_a.s

```
0 func_name:
    addi sp, sp, -4
1    beq a0, x0, loop
    # <8 instructions here>

10    jal ra, malloc

11 loop: bneq t0, x0, done
    # <3 instructions here>

15    j loop

16 done: ret
```

### Second Pass

#### Symbol Table for File A

Label Name	Relative Addressing
func_name	fa_start
loop	fa_start + 44
done	fa_start + (4 * 16)

#### Relocation Table for File A

Label Name	State
loop	fa_start + 44
malloc	?
done	<b>fa_start + 64</b>

## 2-Pass Assembler Example

file\_a.s

```
0 func_name:
    addi sp, sp, -4
1    beq a0, x0, loop
    # <8 instructions here>

10    jal ra, malloc

11 loop: bneq t0, x0, done
    # <3 instructions here>

15    j loop

16 done: ret
```

### Second Pass

#### Symbol Table for File A

Label Name	Relative Addressing
func_name	fa_start
loop	fa_start + 44
done	fa_start + (4 * 16)

#### Relocation Table for File A

Label Name	State
loop	fa_start + 44
malloc	?
done	fa_start + 64

# 2-Pass Assembler Example

Q: what is fa\_start?

A: The "base address" we're using in this example; the assembler and system determine where we're offsetting from.

Q: what does it mean that malloc's state in the relocation table is still "?"

A: malloc wasn't defined in File A; so, it must be defined in a separate user file OR in a separate library. These "external references" will be resolved in the linking stage.

## Second Pass

### Symbol Table for File A

Label Name	Relative Addressing
func_name	fa_start
loop	fa_start + 44
done	fa_start + (4 * 16)

### Relocation Table for File A

Label Name	State
loop	fa_start + 44
malloc	?
done	fa_start + 64

# Pseudo-Instruction Replacement

- Assembler converts to real instructions

Pseudo-Instruction	"Real" Instruction
<code>mv t0, t1</code>	<code>addi t0,t1,0</code>
<code>neg t0, t1</code>	<code>sub t0, zero, t1</code>
<code>li t0, imm</code>	<code>addi t0, zero, imm</code>
<code>not t0, t1</code>	<code>xori t0, t1, -1</code>
<code>beqz t0, loop</code>	<code>beq t0, zero, loop</code>
<code>la t0, str</code>	<code>lui t0, str[31:12] + addi t0, t0, str[11:0]</code> <code>auipc t0, str[31:12] + addi t0, t0, str[11:0]</code>



# Producing Machine Language (1/3)

- Simple Case: Arithmetic, Logical, Shifts, and so on
  - All necessary info is within the instruction already

# Producing Machine Language (2/3)

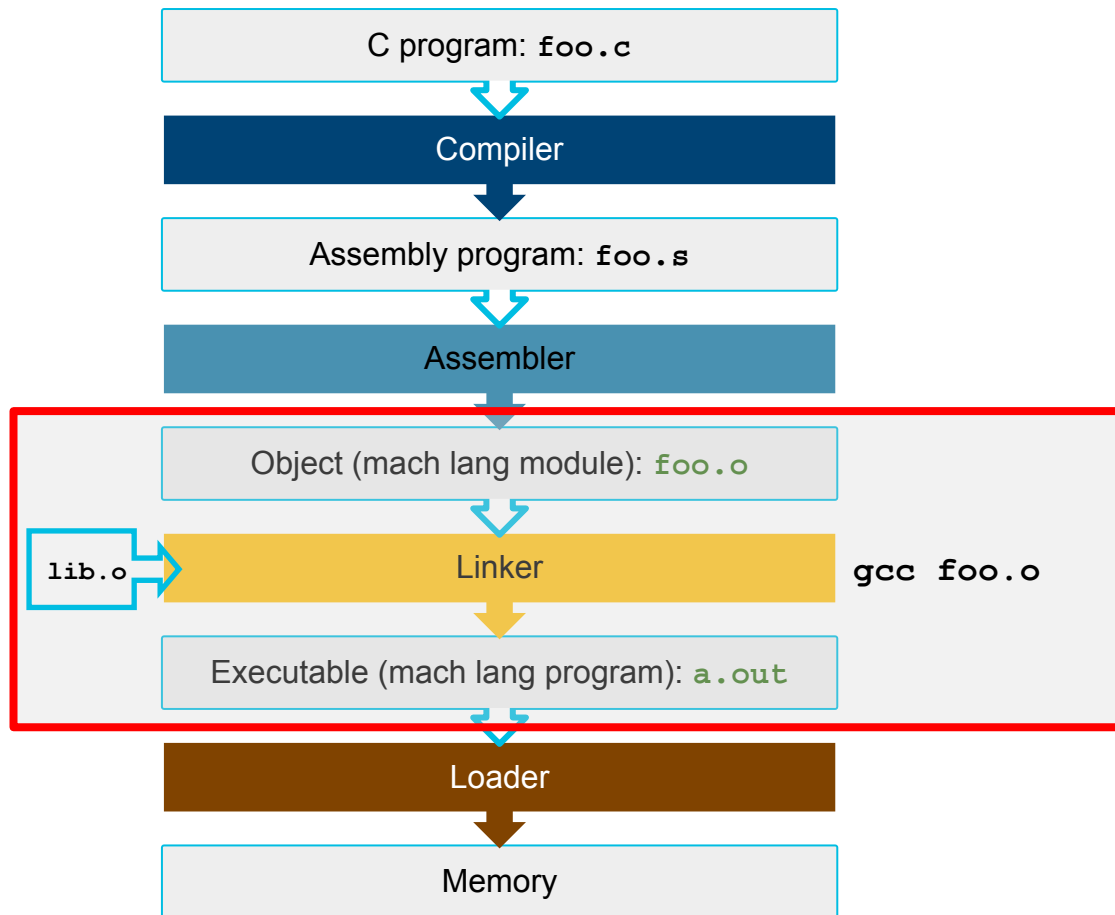
- What about PC-relative jumps (jal) and branches (beq, bne)?
  - Count the number of instruction half-words between target and jump to determine the offset
  - Note: Need to replace pseudoinstructions first to know how many instructions to branch/jump
- What about references to static data?
  - la sometimes can be broken up into lui and addi (use auipc/addi for PC-relative)
  - These require the full 32-bit address of the data
  - These can't be determined yet, so we create two tables
  - Symbol and relocation table, for the linker to resolve!

# Producing Machine Language (3/3)

- “Forward Reference” problem
  - Instructions using labels can refer to labels that are “forward” in the program
  - Solved by taking two passes over the program:
    - First pass remembers position of labels
    - Second pass uses label positions to generate code

L (1): Linker

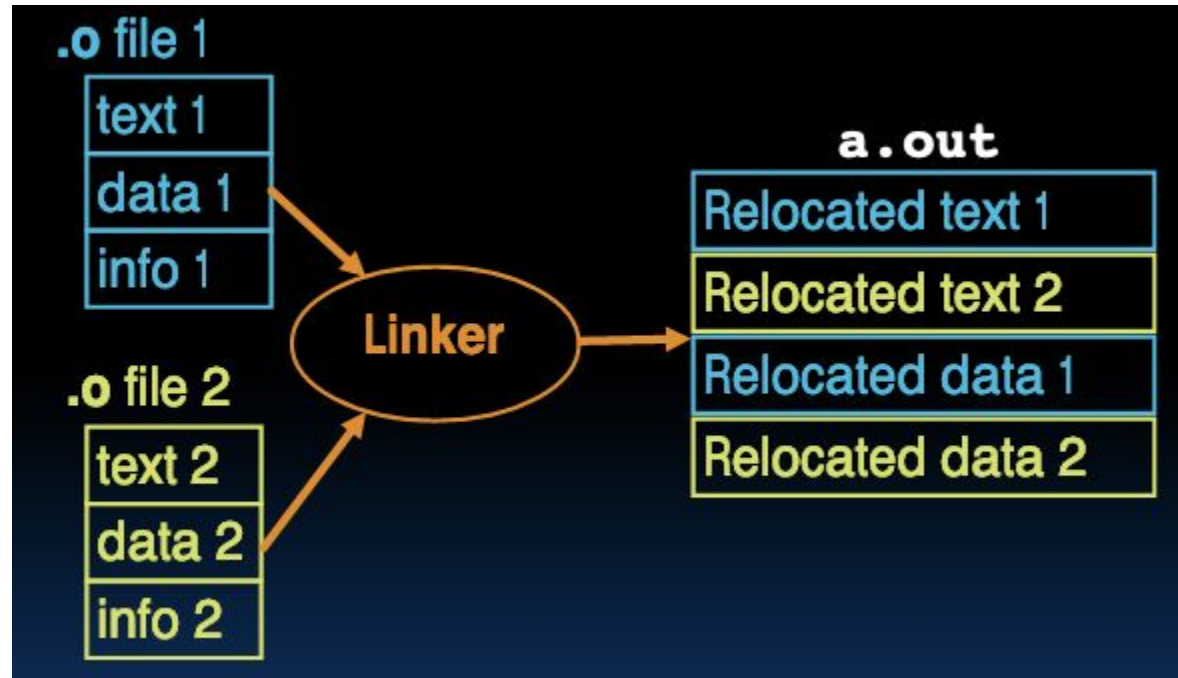
# Linker/Linking



# Linker/Linking

- Object files (\*.o)  $\Rightarrow$  executable (\*.out)
- Categorises code chunks (segments) from each object file and groups them together in order
  - Order is determined by the linker
  - Standard segments: text, data
- Resolves **all** references
  - References from user files  $\rightarrow$  symbol table
  - External references  $\rightarrow$  static/dynamic linking library files
  - Goes through all relocation entries
- Result: all absolute addresses filled in

# Linker Visualization



# Linker Execution

- 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



# \*.out (executable) files

- Contains:
  - Header
    - Size of text and data segments
    - Why do we not need the location of the text/data segments?
      - Everything is combined together at this point
  - Text + data segments

# 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; jal ext\_label)
  - Always relocate
- Static Data Reference (often lui/addi)
  - Always relocate

# Resolving References (1/2)

- Linker assumes first word of first text segment is at address 0x10000 for RV32 (0x40000000 for RV64)
- 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

Q: How can the linker assume the code will be loaded to the same address every time?

A: Later this semester: virtual memory!

## 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 malloc)
- Once absolute address is determined, fill in the machine code appropriately

# 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, and thus, portable
- Alternative: dynamically-linked libraries (DLL), common on Windows & UNIX platforms

# 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

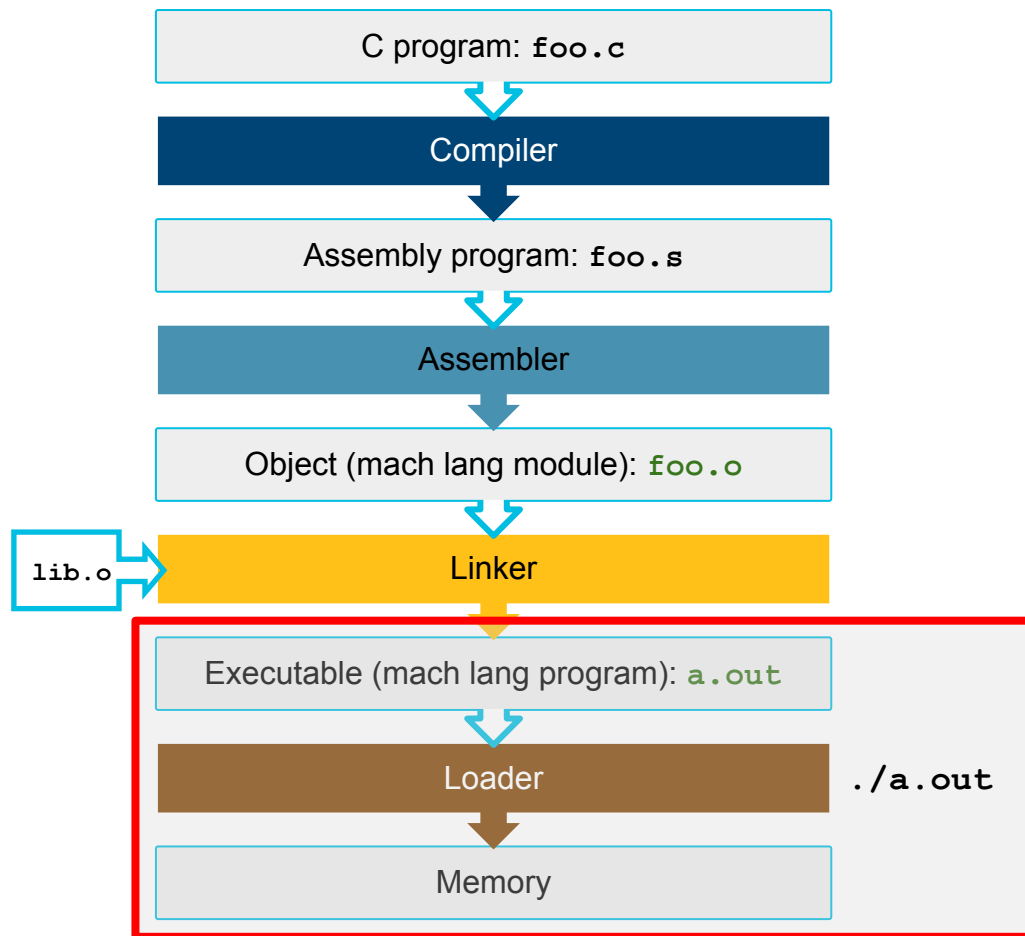
## Dynamically Linked Libraries (2/2)

- 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)

L (2): Loader



# Loader/Loading



# Loader/Loading

- executable (\*.out)  $\Rightarrow$  memory
- Sets up the necessary space for loading text + data segments into memory space
  - And copies text+data into this space
- Initializes stack with program arguments
  - Stack pointer set to highest position under arguments
  - Stack register set to stack pointer address
- Clear most other registers (set to zero)
- Jumps to start-up routine that copies program's arguments from stack to registers & sets the PC (program counter)
  - **Program counter:** a special purpose register that holds the memory address of the currently executing line of code
    - NOTE: **not** one of the general purpose 32 registers we've talked about!
  - If program returns, start-up routine also terminates program with exit system call
- System-dependent behavior—part of the operating system (CS 162!)

# Summary

- Today we briefly reviewed interpretation vs compilation and keyed in on why the language we've used so far might not be the most precise
- We covered CALL:
  - Compilation: what it actually means (and discovered it's just the first step in a very long process)
  - Assembly: how human-readable parts of assembly code get swapped out for more machine-readable features
  - Linking: how different files and libraries make their way into the final product
  - Loading: how a system processes our code to run