CS 110
Computer Architecture
Running a Program - CALL (Compiling, Assembling, Linking, and Loading)

Instructor:
Sören Schwertfeger

http://shtech.org/courses/ca/

School of Information Science and Technology SIST

ShanghaiTech University

Slides based on UC Berkley's CS61C

Summary of RISC-V Instruction Formats

31 30 25	24 21 20	19 15	14 12	11 8 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]				rd	opcode	U-type
imm[20|10:1|11]]		imm[19:12]		rd	opcode	J-type

Complete RV32I ISA

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI

0000000		shamt	rs1	001	rd	0010011	SLLI
0000000		shamt	rs1	101	rd	0010011	SRLI
0100000		shamt	rs1	101	rd	0010011	SRAI
0000000		rs2	rs1	000	rd	0110011	ADD
0100000		rs2	rs1	000	rd	0110011	SUB
0000000		rs2	rs1	001	rd	0110011	SLL
0000000		rs2	rs1	010	rd	0110011	SLT
0000000		rs2	rs1	011	rd	0110011	SLTU
0000000		rs2	rs1	100	rd	0110011	XOR
0000000		rs2	rs1	101	rd	0110011	SRL
0100000		rs2	rs1	101	rd	0110011	SRA
0000000		rs2	rs1	110	rd	0110011	OR
0000000		rs2	rs1	111	rd	0110011	AND
0000	pred	succ	00000	000	00000	0001111	FENCE
0000	0000	0000	00000	001	00000	0001111	FENCE.I
000000000000			00000	000	00000	1110011	ECALL
000000000001			00000	000	00000	1110011	EBREAK
csr			rs1	001	rd	1110011	CSRRW
csr			rs1	010	rd	1110011	CSRRS
csr			rs1	011	rd	1110011	CSRRC
csr			zimm	101	rd	1110011	CSRRWI
csr			zimm	110	rd	1110011	CSRRSI
csr			zimm	111	rd	1110011	CSRRCI

Not in CA lectures

Conclusion RISC-V

- Simplification works for RISC-V: Instructions are same size as data word (one word) so that they can use the same memory.
- Computer actually stores programs as a series of these 32-bit numbers.
- We have covered all RISC-V instructions and registers
  - R-type, I-type, S-type, B-type, U-type and J-type instructions
  - Practice assembling and disassembling

en.wikipedia.org/wiki/Big_endian

Big Endian vs. Little Endian

Big-endian and little-endian from Jonathan Swift's *Gulliver's Travels*

- **The order in which BYTES are stored in memory**
- **Bits always stored as usual. (E.g., 0xC2=0b 1100 0010)**

Consider the number 1025 as we normally write it:

BYTE3	BYTE2	BYTE1	BYTE0
00000000	00000000	00000100	00000001

Big Endian

ADDR3	ADDR2	ADDR1	ADDR0
BYTE0	BYTE1	BYTE2	BYTE3
00000001	00000100	00000000	00000000

Examples

Names in China (e.g., Schwertfeger, Sören)

Java Packages: (e.g., org.mypackage.HelloWorld)

Dates done correctly ISO 8601 YYYY-MM-DD (e.g., 2018-09-07)

Eating Pizza crust first

Unix file structure (e.g., /usr/local/bin/python)

"Network Byte Order": most network protocols

IBM z/Architecture; very old Macs

Little Endian

ADDR3	ADDR2	ADDR1	ADDR0
BYTE3	BYTE2	BYTE1	BYTE0
00000000	00000000	00000100	00000001

Examples

Names in the west (e.g., Sören Schwertfeger)

Internet names (e.g., sist.shanghaitech.edu.cn)

Dates written in England DD/MM/YYYY (e.g., 07/09/2018)

Eating Pizza skinny part first (the normal way)

CANopen

Intel x86; RISC-V

bi-endian: ARM (runs mostly little endian), MIPS, IA-64, PowerPC

RISC-V: Little Endian

(E.g., 1025 = 0x401 = 0b 0100 0000 0001)

ADDR3	ADDR2	ADDR1	ADDR0
BYTE3	BYTE2	BYTE1	BYTE0
00000000	00000000	00000100	00000001

Little Endian
Most significant byte in a word
(numbers are addresses)

...	...	...	...
12	13	14	15
8	9	10	11
4	5	6	7
0	1	2	3

- Hexadecimal number:
  0xFD34AB88 ($4,248,087,432_{ten}$) =>
  - Byte 0: 0x88 (136_{ten})
  - Byte 1: 0xAB (171_{ten})
  - Byte 2: 0x34 (52_{ten})
  - Byte 3: 0xFD (253_{ten})

Address:	64 address of word (e.g. int)			
Address:	64	65	66	67
Data:	0x88	0xAB	0x34	0xFD

- Little Endian: The "Endianess" is little:
  - It starts with the smallest (least significant) Byte
  - Swapped from how we write the number

Levels of Representation/Interpretation

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

Compiler

Assembly Language Program (e.g., MIPS)

Assembler

Machine Language Program (MIPS)

Machine Interpretation

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

Architecture Implementation

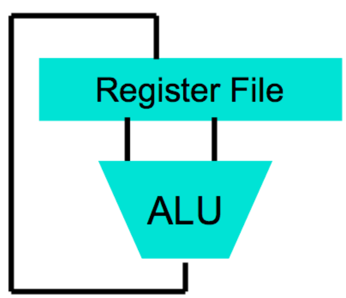
Logic Circuit Description (Circuit Schematic Diagrams)

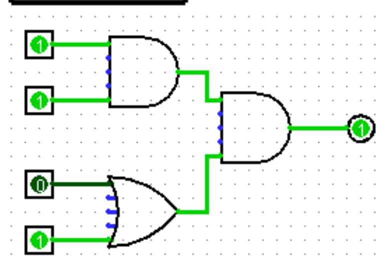
```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
lw   $t0, 0($2)
lw   $t1, 4($2)
sw   $t1, 0($2)
sw   $t0, 4($2)
```

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

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

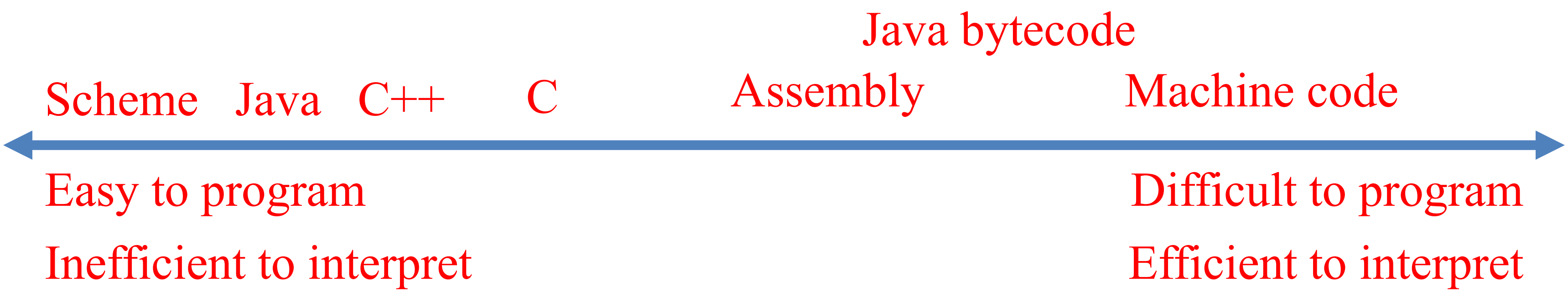




+ How to take a program and run it

Language Execution Continuum

- An Interpreter is a program that executes other programs.

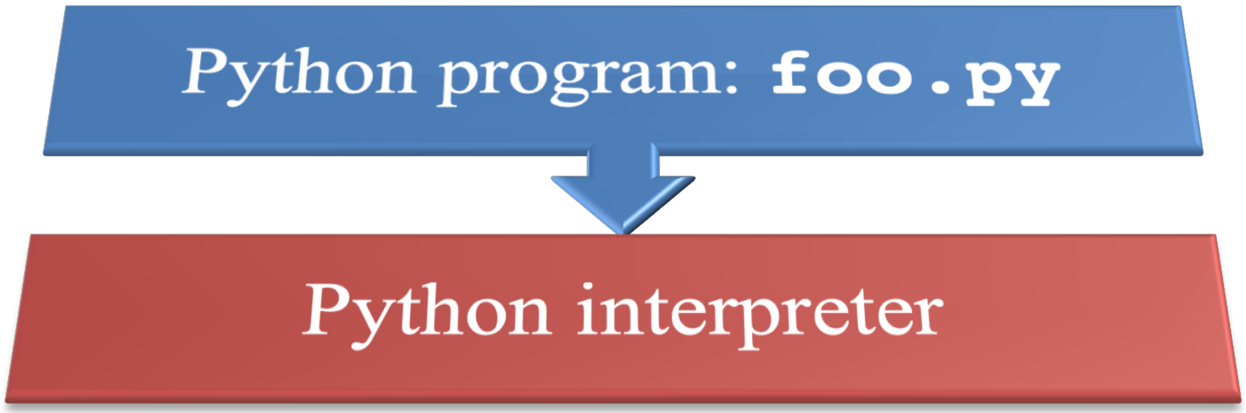


- Language translation gives us another option
- In general, we interpret a high-level language when efficiency is not critical and translate to a lower-level language to increase performance

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

Interpretation



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

Interpretation

- Any good reason to interpret machine language in software?
- VENUS RISC-V simulator: useful for learning / debugging
- Apple Macintosh conversion
  - Switched from Motorola 680x0 instruction architecture to PowerPC.
    - Similar issue with switch to 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)

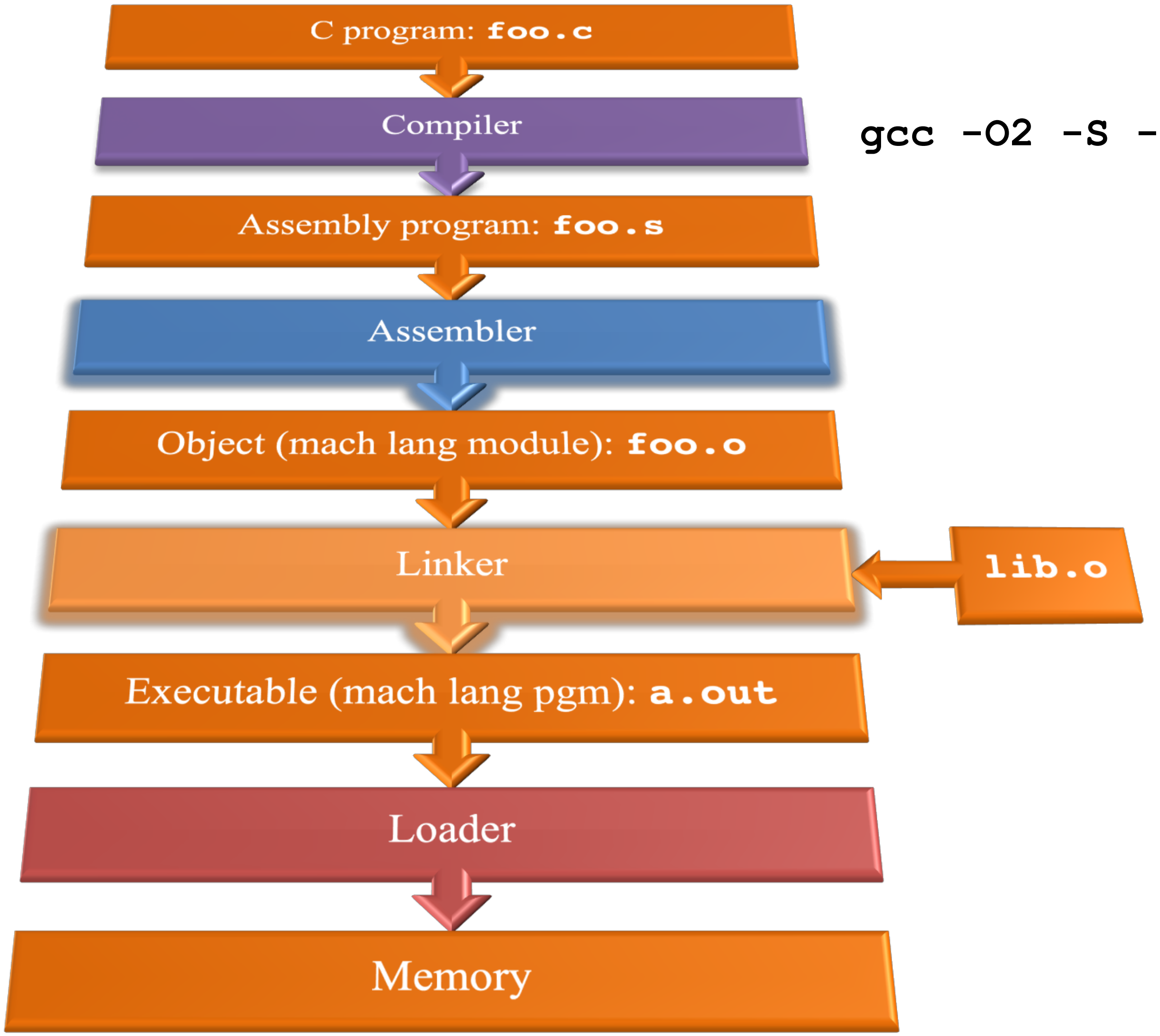
Interpretation vs. Translation? (1/2)

- Generally easier to write interpreter
- 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 (eg. Microsoft keeps all their source code secret)
  - Alternative model, “open source”, creates value by publishing the source code and fostering a community of developers.

Steps in compiling a C program

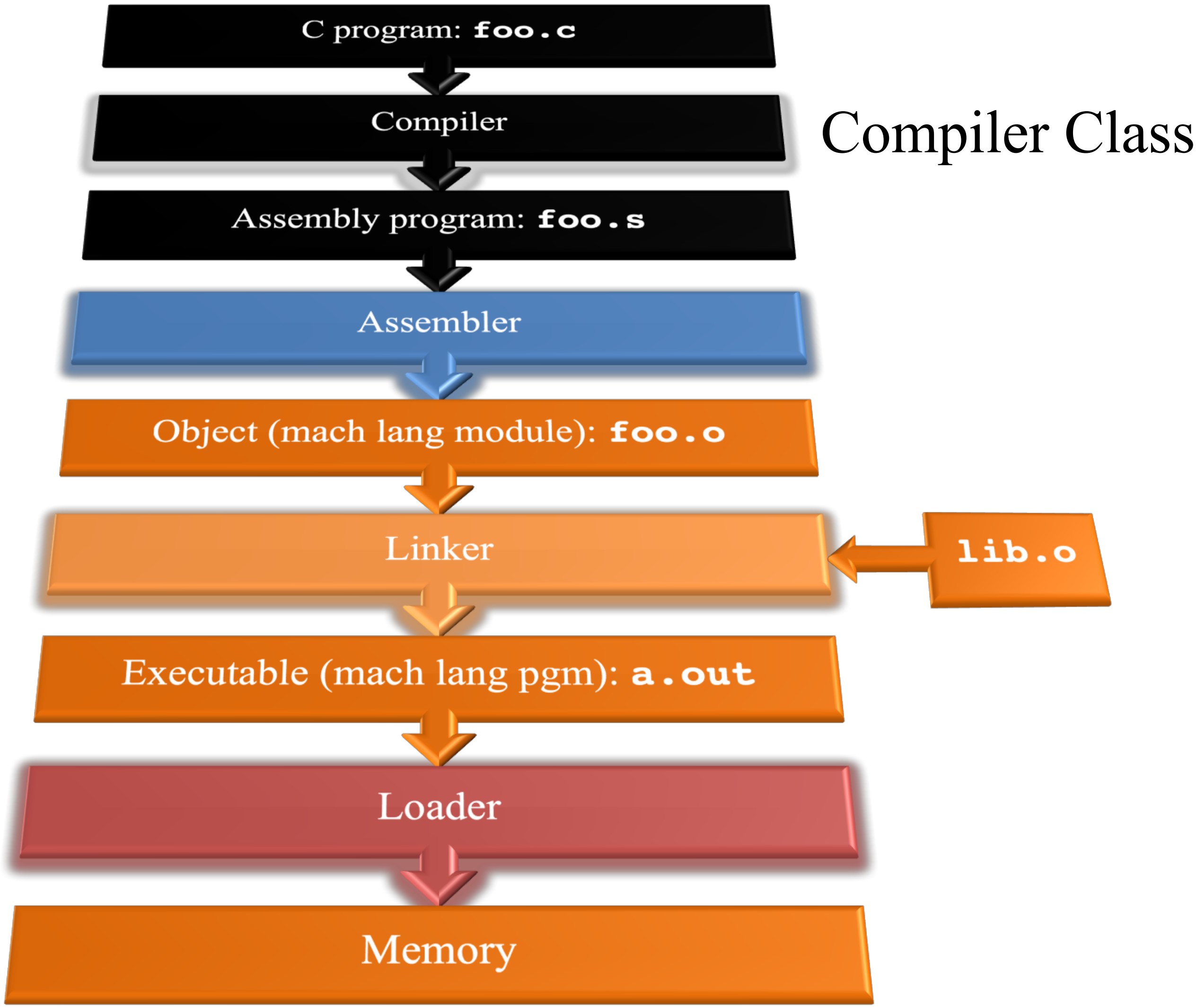


`gcc -O2 -S -c foo.c`

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:
  - **`move t1,t2`** ⇒ **`addi t1,t2,0`**

Where Are We Now?



Assembler

- Input: Assembly Language Code (MAL) (e.g., **`foo.s`** for MIPS)
- Output: Object Code, information tables (TAL) (e.g., **`foo.o`** for MIPS)
- Reads and Uses Directives
- Replace Pseudo-instructions
- Produce Machine Language
- Creates Object File

Assembler Directives

- Give directions to assembler, but do not produce machine instructions

  `.text`: Subsequent items put in user text segment (machine code)

  `.data`: Subsequent items put in user data segment (binary rep of data in source file)

  `.globl sym`: declares `sym` global and can be referenced from other files

  `.asciiz str`: Store the string **`str`** in memory and null-terminate it

  `.word w1…wn`: Store the *n* 32-bit quantities in successive memory words

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 immediates count halfwords

STATIC Addressing

PC-Relative Addressing

Producing Machine Language (1/3)

- Simple Case
  - Arithmetic, Logical, Shifts, and so on
  - All necessary info is within the instruction already
- What about Branches?
  - 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
- So these can be handled

Producing Machine Language (2/3)

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

3 words forward (6 halfwords) → beq

3 words **back** (6 halfwords) → j

```
      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                # Go to L1
L2:
```

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

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 *halfwords* 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

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

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

Peer Instruction

Which of the following is a correct assembly language sequence sequence for the pseudoinstruction: la t1, FOO?*

*Assume the address of FOO is 0xABCD0124

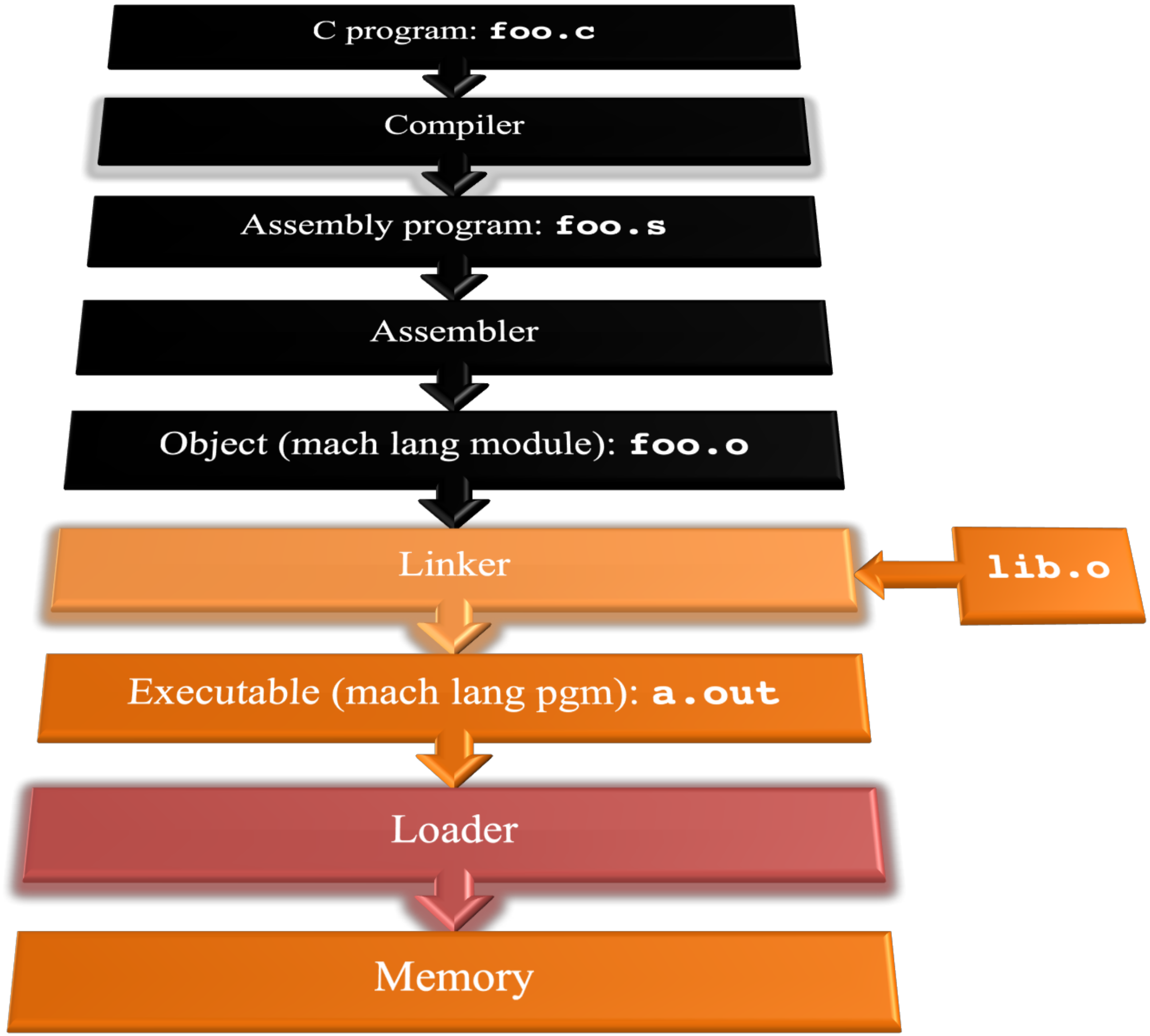
A:
```
ori   t1, 0xABCD0
addi  t1, 0x124
```

B:
```
ori   t1, 0x124
lui   t1, 0xABCD0
```

C:
```
lui   t1, 0xD0124
ori   t1, 0xABC
```

D:
```
lui   t1, 0xABCD0
addi  t1, 0x124
```

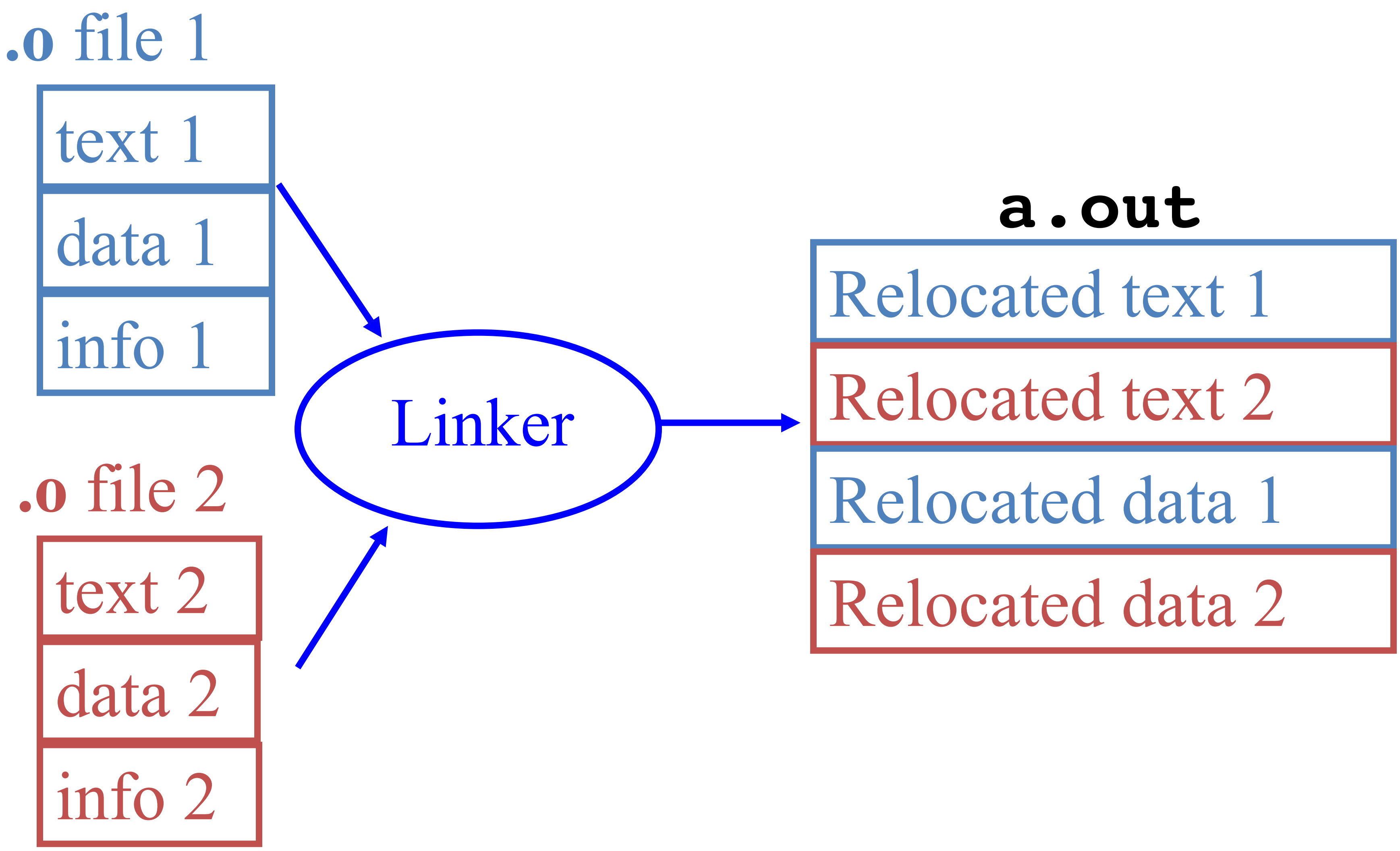
Where Are We Now?



Linker (1/3)

- Input: Object code files, information tables (e.g., `foo.o,libc.o` for MIPS)
- Output: Executable code (e.g., `a.out` for MIPS)
- 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
  - That is, fill in all absolute addresses

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

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

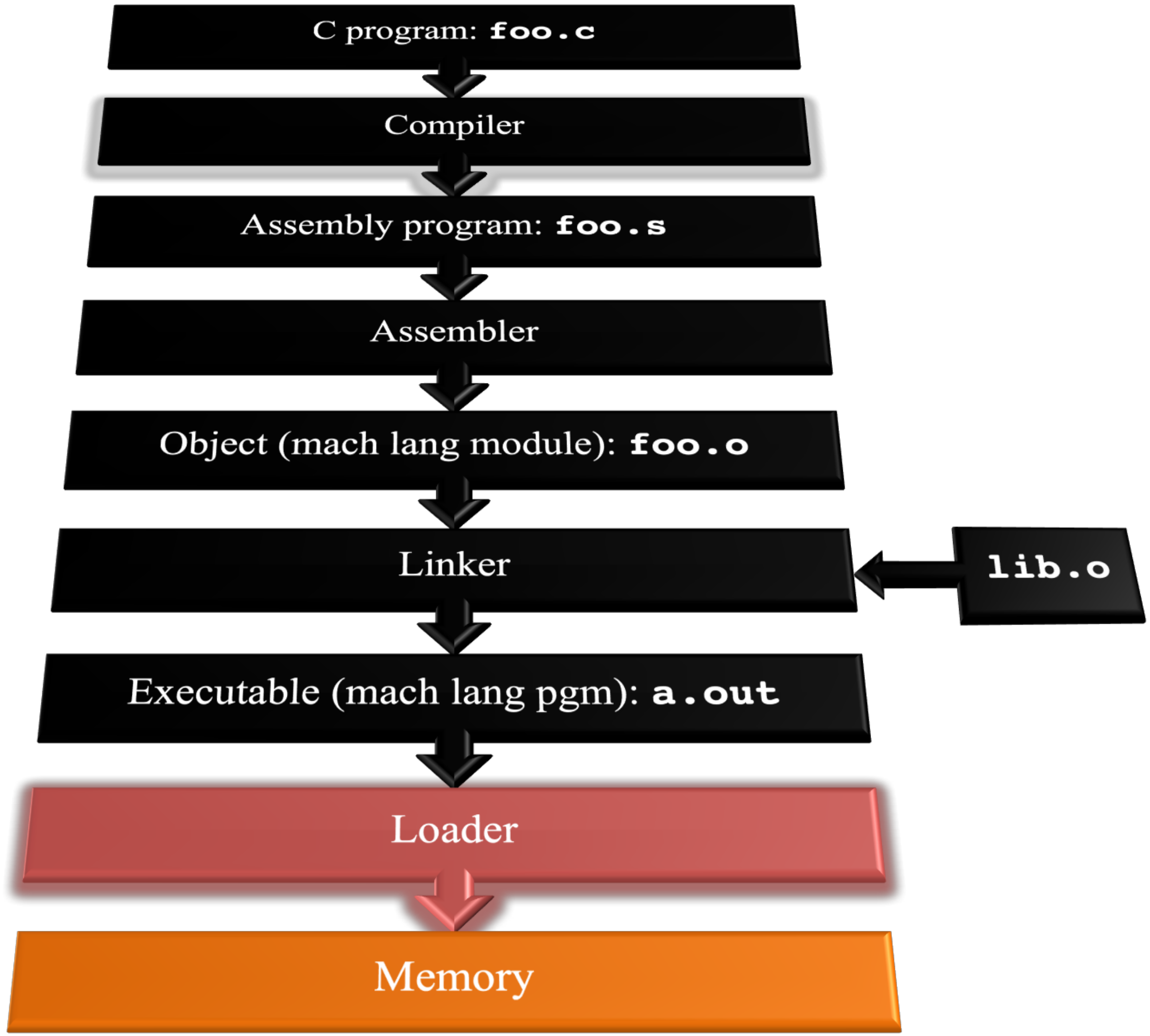
Resolving References (1/2)

- 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

Resolving References (2/2)

- To resolve references:
  - search for reference (data or label) in all "user" symbol tables
  - if not found, search library files (for example, for `printf`)
  - once absolute address is determined, fill in the machine code appropriately
- Output of linker: executable file containing text and data (plus header)

Where Are We Now?



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

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 and 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 the exit system call

Question

At what point in process are all the machine code bits generated for the following assembly instructions:

1) `add x6, x7, x8`
2) `jal x1, fprintf`

A: 1) & 2) After compilation

B: 1) After compilation, 2) After assembly

C: 1) After assembly, 2) After linking

D: 1) After assembly, 2) After loading

E: 1) After compilation, 2) After linking

Answer

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

1) `add x6, x7, x8`
2) `jal x1, fprintf`

(1) After assembly, (2) After linking

Example: C ⇒ Asm ⇒ Obj ⇒ Exe ⇒ Run

C Program Source Code: prog.c

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

"printf" lives in "libc"

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
```

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 string1
c:   00050513  addi   a0,a0,0         # addr placeholder string1
10:  000005b7  lui    a1,0x0          # addr placeholder string2
14:  00058593  addi   a1,a1,0         # addr placeholder string2
18:  00000097  auipc  ra,0x0          # addr placeholder printf
1c:  000080e7  jalr   ra              # addr placeholder printf
20:  00c12083  lw     ra,12(sp)
24:  01010113  addi   sp,sp,16
28:  00000513  addi   a0,a0,0
2c:  00008067  jalr   ra
```

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 # 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  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` does not 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`**

Static vs Dynamically linked libraries

- What we've described is the traditional way: statically-linked approach
  - The 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)
  - It includes the entire library even if not all of it will be used
  - Executable is self-contained
- An alternative is dynamically linked libraries (DLL), common on Windows (.dll) & UNIX (.so) platforms

Dynamically linked libraries

- 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

- The prevailing approach to dynamic linking uses machine code as the “lowest common denominator”
  - The linker does not use information about how the program or library was compiled (i.e., what compiler or language)
  - This can be described as “linking at the machine code level”
  - This isn’t the only way to do it ...

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.