CS 110
Computer Architecture
Lecture 8:
Running a Program - CALL (Compiling, Assembling, Linking, and Loading)

Instructors:

Sören Schwertfeger & Chundong Wang

https://robotics.shanghaitech.edu.cn/courses/ca/20s/

School of Information Science and Technology SIST

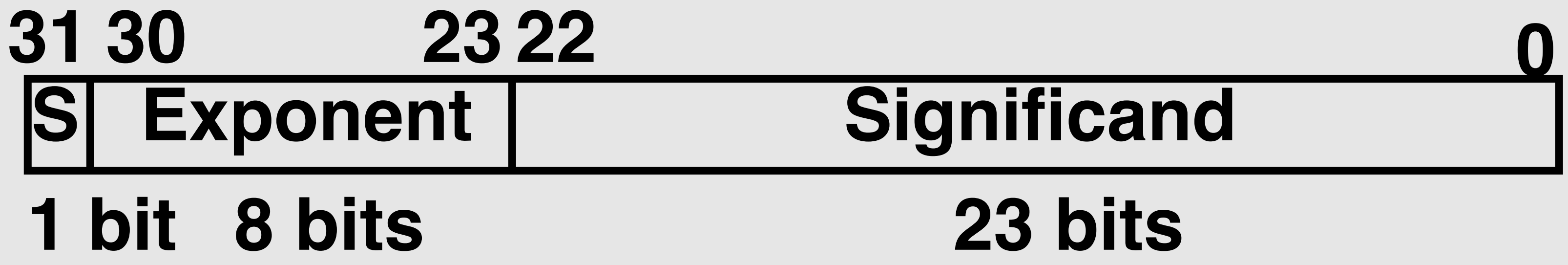
ShanghaiTech University

Slides based on UC Berkley's CS61C

IEEE 754 Floating Point Standard

- Called Biased Notation, where bias is number subtracted to get final number
  - IEEE 754 uses bias of 127 for single prec.
  - Subtract 127 from Exponent field to get actual value for exponent

- **Summary (single precision):**



- $(-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent}-127)}$

  - Double precision identical, except with exponent bias of 1023 (half, quad similar)

Representation for ± ∞

- In FP, divide by 0 should produce ± ∞, not overflow.
- Why?
  - OK to do further computations with ∞
    E.g., X/0 > Y may be a valid comparison

- IEEE 754 represents ± ∞
  - Most positive exponent reserved for ∞
  - Significands all zeroes

Representation for 0

- Represent 0?
  - exponent all zeroes
  - significand all zeroes
  - What about sign? Both cases valid

```
+0: 0 00000000 00000000000000000000000
-0: 1 00000000 00000000000000000000000
```

Special Numbers

- What have we defined so far? (Single Precision)

Exponent	Significand	Object
0	0	0
0	nonzero	**???**
1-254	anything	+/- fl. pt. #
255	0	+/- ∞
255	nonzero	**???**

Clever idea:

– Use exp=0,255 & Sig!=0

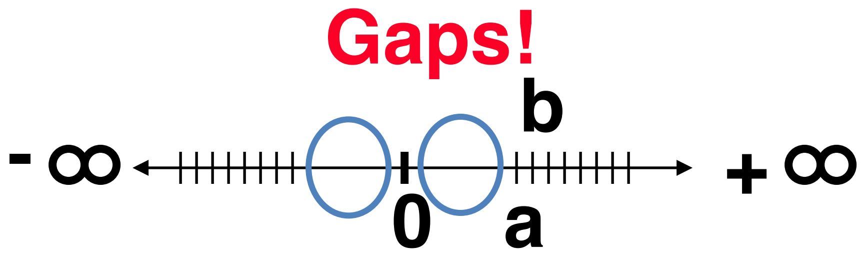
Representation for Not a Number

- What do I get if I calculate
  `sqrt(-4.0)` or `0/0`?
  - If ∞ not an error, these shouldn't be either
  - Called Not a Number (NaN)
  - Exponent = 255, Significand nonzero
- Why is this useful?
  - Hope NaNs help with debugging?
  - They contaminate: op(NaN, X) = NaN
  - Can use the significand to identify which!

Representation for Denorms (1/2)

- Problem: There's a gap among representable FP numbers around 0
  - Smallest representable pos num:
    - a = 1.000… * 2^{-126} = 2^{-126}
  - Second smallest representable pos num:
    - b = 1.000…1 * 2^{-126}
      = (1 + 0.000…1) * 2^{-126}
      = (1 + 2^{-23}) * 2^{-126}
      = 2^{-126} + 2^{-149}
  - a - 0 = 2^{-126}
  - b - a = 2^{-149}

Normalization and implicit 1 is to blame!

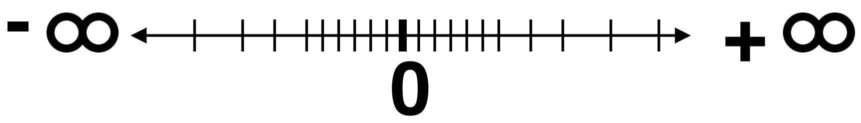


Representation for Denorms (2/2)

- **Solution:**
  - **We still haven't used Exponent = 0, Significand nonzero**
  - **DEnormalized number: no (implied) leading 1, implicit exponent = -126.**
  - **Smallest representable pos num:**

    $a = 2^{-149}$
  - **Second smallest representable pos num:**

    $b = 2^{-148}$



Special Numbers

Exponent	Significand	Object
0	0	0
0	nonzero	Denorm
1-254	anything	+/- fl. pt. #
255	0	+/- ∞
255	nonzero	NAN

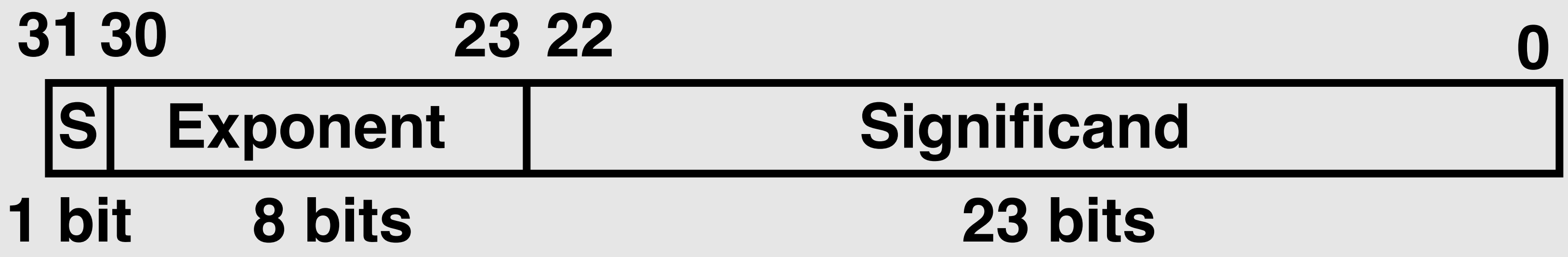
Summary

Exponent tells Significand how much (2^i) to count by (…, 1/4, 1/2, 1, 2, …)

- Floating Point lets us:
  - Represent numbers containing both integer and fractional parts; makes efficient use of available bits.
  - Store approximate values for very large and very small #s.
- IEEE 754 Floating-Point Standard is most widely accepted attempt to standardize interpretation of such numbers (Every desktop or server computer sold since ~1997 follows these conventions)

Can store NaN, ± ∞

- **Summary (single precision):**



- $(-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent}-127)}$

- Double precision identical, except with exponent bias of 1023 (half, quad similar)

Levels of Representation/Interpretation

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

Compiler

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

Assembler

Machine Language Program (e.g. RISC-V)

Machine Interpretation

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

Architecture Implementation

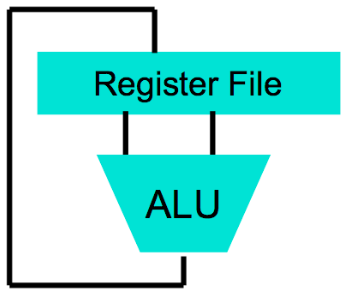
Logic Circuit Description (Circuit Schematic Diagrams)

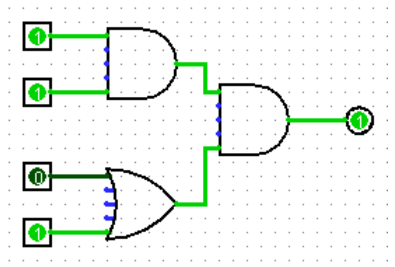
```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
lw    t0, 0(s2)
lw    t1, 4(s2)
sw    t1, 0(s2)
sw    t0, 4(s2)
```

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
```

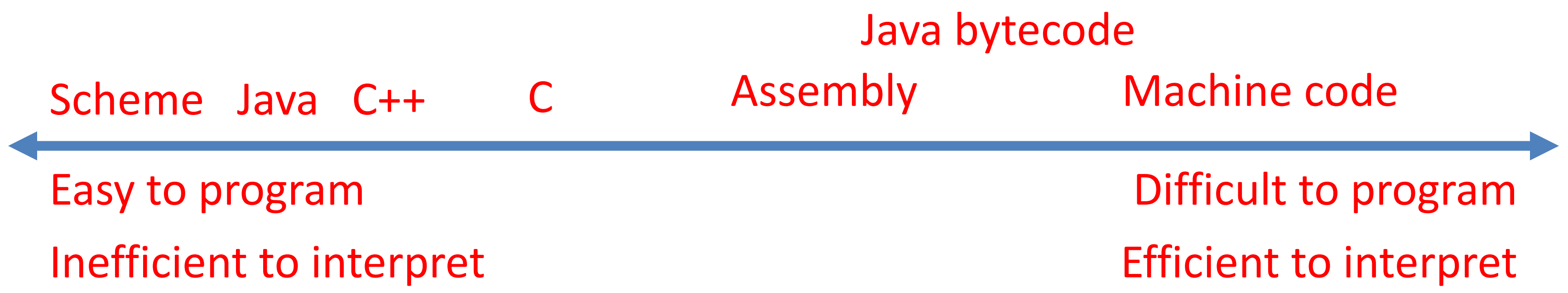




+ How to take a program and run it

Language Execution Continuum

- An Interpreter is a program that executes other programs.

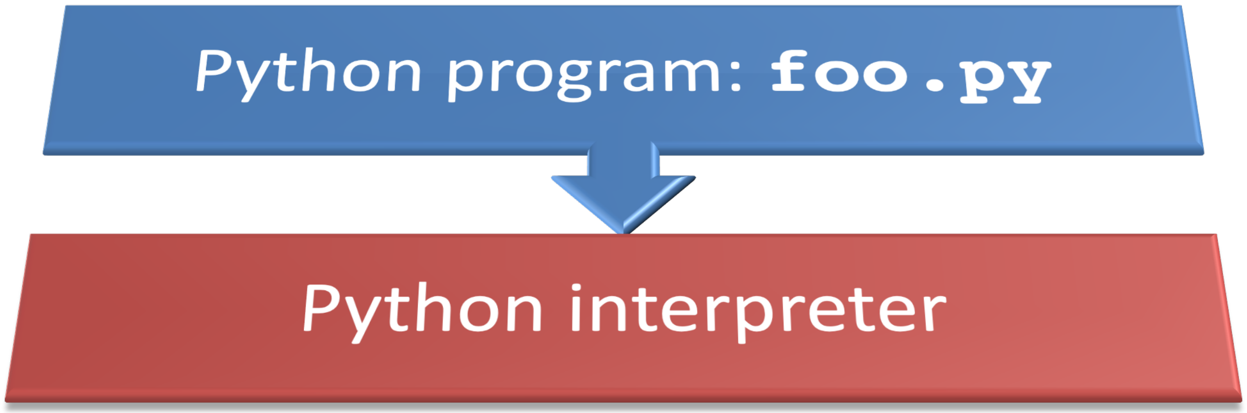


- Language translation gives us another option
- In general, we interpret a high-level language when efficiency is not critical and translate to a lower-level language to increase performance

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
    - Similar issue with switch to ARM
  - 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)

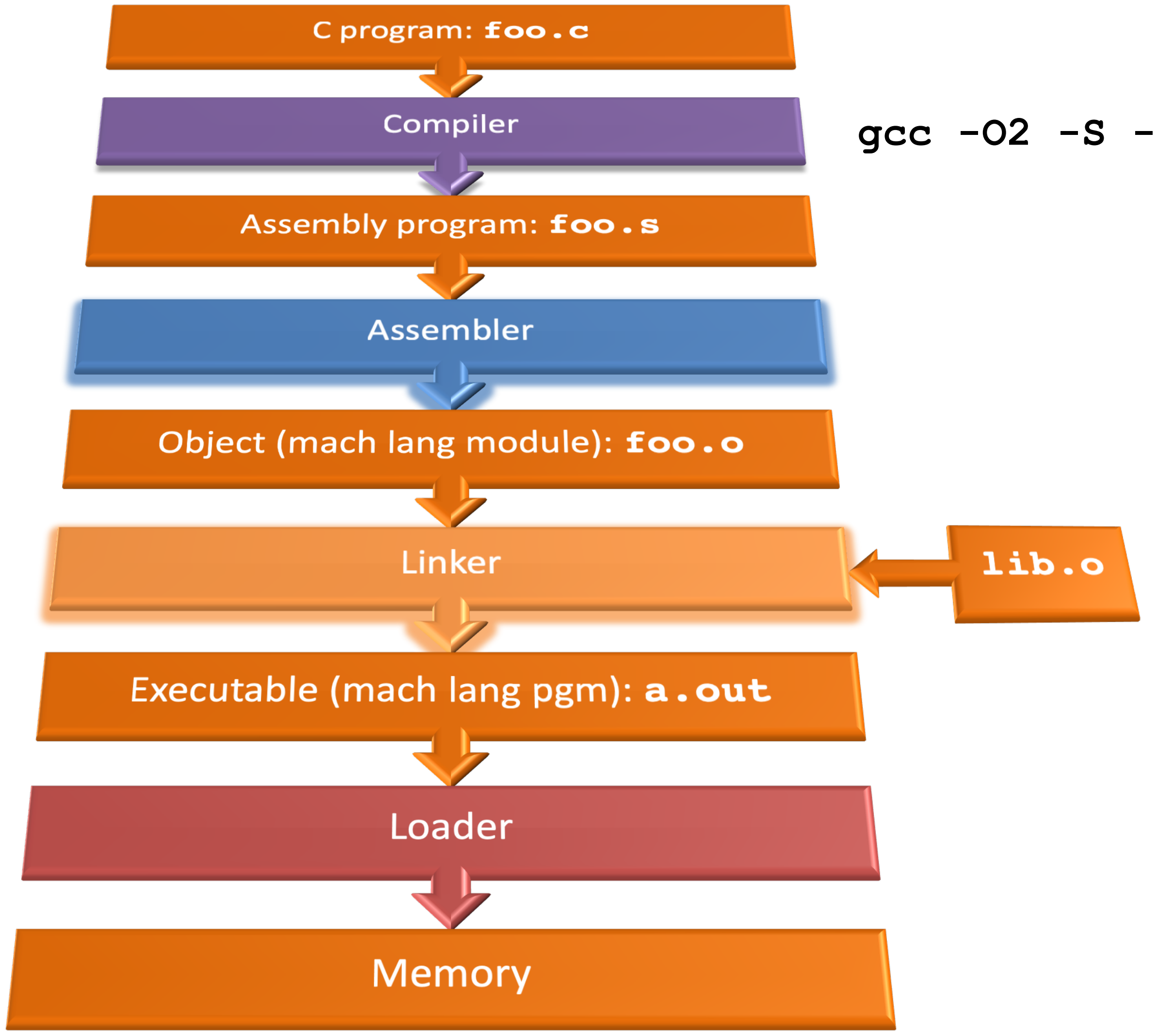
Interpretation vs. Translation? (1/2)

- 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
```

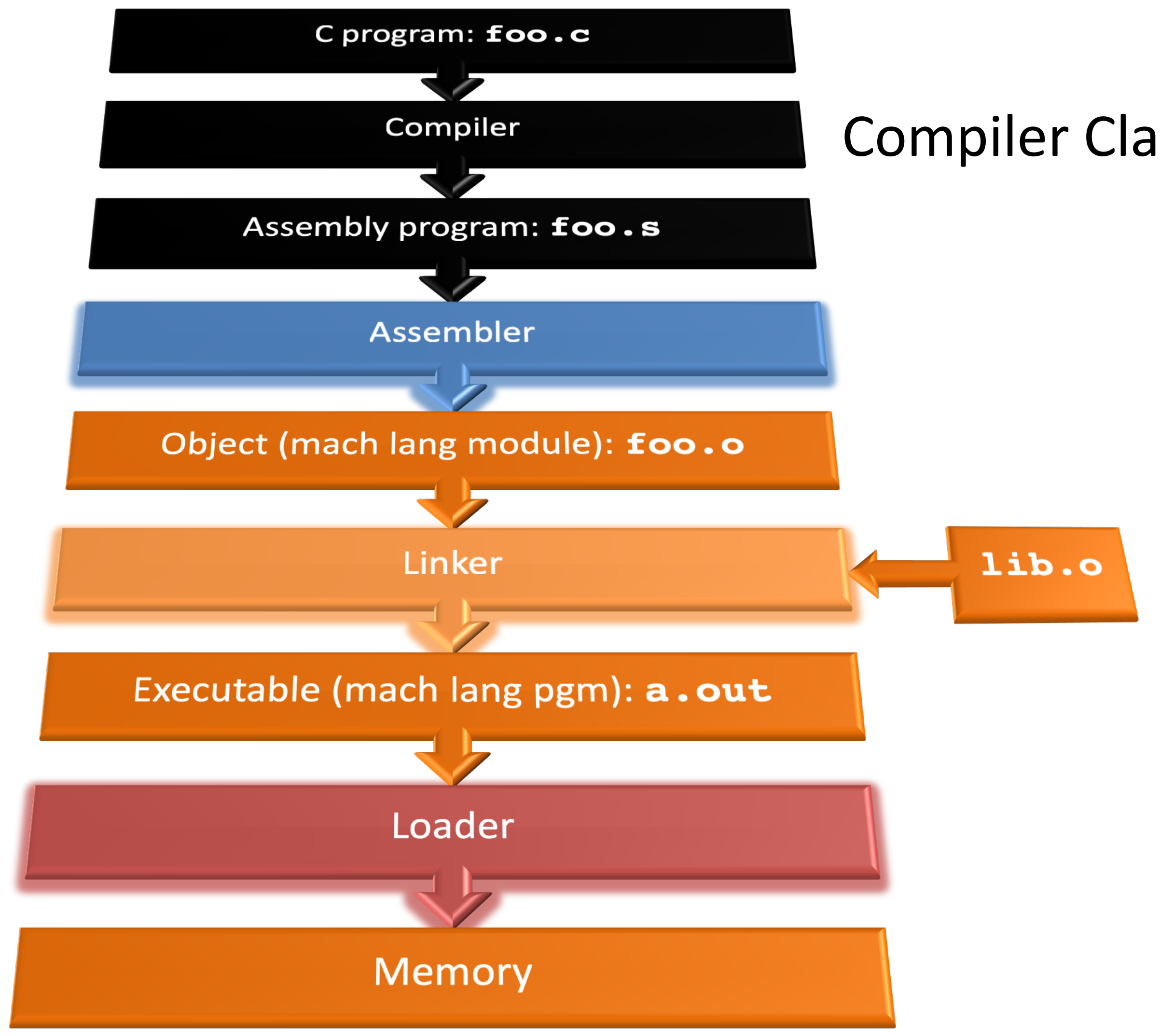
Compiler

- 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
- <u>Pseudo-instructions</u>: instructions that assembler understands but not in machine
  For example:
  - **`move t1,t2`** ⇒ **`addi t1,t2,0`**

Steps In The Compiler

- Lexer:
  - Turns the input into "tokens", recognizes problems with the tokens
- Parser:
  - Turns the tokens into an "Abstract Syntax Tree", recognizes problems in the program structure
- Semantic Analysis and Optimization:
  - Checks for semantic errors, may reorganize the code to make it better
- Code generation:
  - Output the assembly code

Where Are We Now?



Compiler Class

Assembler

- Input: Assembly Language Code
- (e.g., **`foo.s`** for RISC-V)
- Output: Object Code, information tables (e.g., **`foo.o`** for RISC-V)
- 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

Pseudo	Real
`nop`	`addi x0, x0, 0`
`not rd, rs`	`xori rd, rs, -1`
`beqz rs, offset`	`beq rs, x0, offset`
`bgt rs, rt, offset`	`blt rt, rs, offset`
`j offset`	`jal x0, offset`
`ret`	`jalr x0, x1, offset`

`call offset` (if too big for just a `jal`)	`auipc x6, offset[31:12]`
	`jalr x1, x6, offset[11:0]`

`tail offset` (if too far for a `j`)	`auipc x6, offset[31:12]`
	`jalr x0, x6, offset[11:0]`

So what is "tail" about...

- Often times your code has a convention like this:

```
{   ...
    lots of code
    return foo(y);
}
```

  - It can be a recursive call to **foo()** if this is within **foo()**, or call to a different function...
- So for efficiency...
  - Evaluate the arguments for **foo()** and place them in **a0-a7**...
  - Restore **ra**, all callee saved registers, and **sp**
  - Then call **foo()** with **`j`** or **`tail`**
- Then when **foo()** returns, it can return ***directly*** to where it needs to return to
  - Rather than returning to wherever **foo()** was called and returning from there
  - ***Tail Call Optimization***

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

16b "RISC-V C" Instruction Set

- Last lecture: the RISC-V includes an optional "C" (Compact) 16b ISA
  - https://content.riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf
  - Understanding why it was designed this way is useful, but not used in class. Might inspire exam questions…
- At this point in CALL, assembler can pattern match and turn 32b instructions into 16b instructions
  - So the presence of the 16b instructions ***doesn't need to be known to anybody but the assembler and the RISC-V processor itself***!
  - EG, pattern of:
    sw s0 4(sp) converts to **c.swsp s0 4**
    beq x0 s2 20 converts to **c.beqz s2 20**

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
3 words forward
(6 halfwords)  ----------→  beq  t1,  zero,  L2    # NO! t2 <= 0; Go to L2
                            addi t2,  t2, -1       # YES! t2 > 0; t2--
3 words back
(6 halfwords)  ----------→  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 jumps (**`j, jal`**) and branches (**`beq, bne`**)?
  - Jumps within a file are PC relative (and we can easily compute):
    - Just count the number of instruction *halfwords* between target and jump to determine the offset: *position-independent code (PIC)*
  - Jumps to *other* files we can't
- What about references to static data?
  - **`la`** gets broken up into **`lui`** and **`addi`**
  - 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

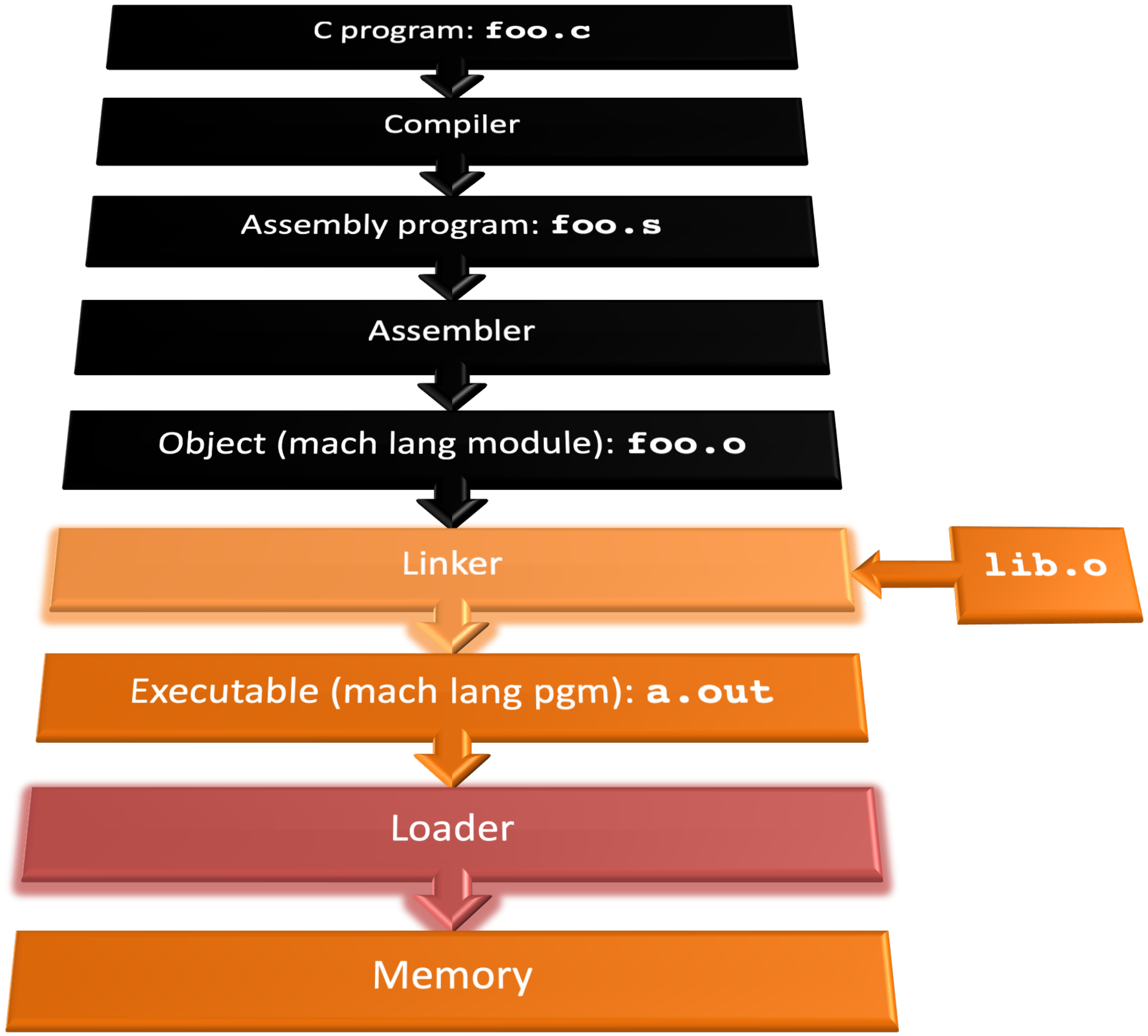
Relocation Table

- List of "items" whose address this file needs
  What are they?
  - Any external label jumped to: **`jal, jalr`**
    - 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

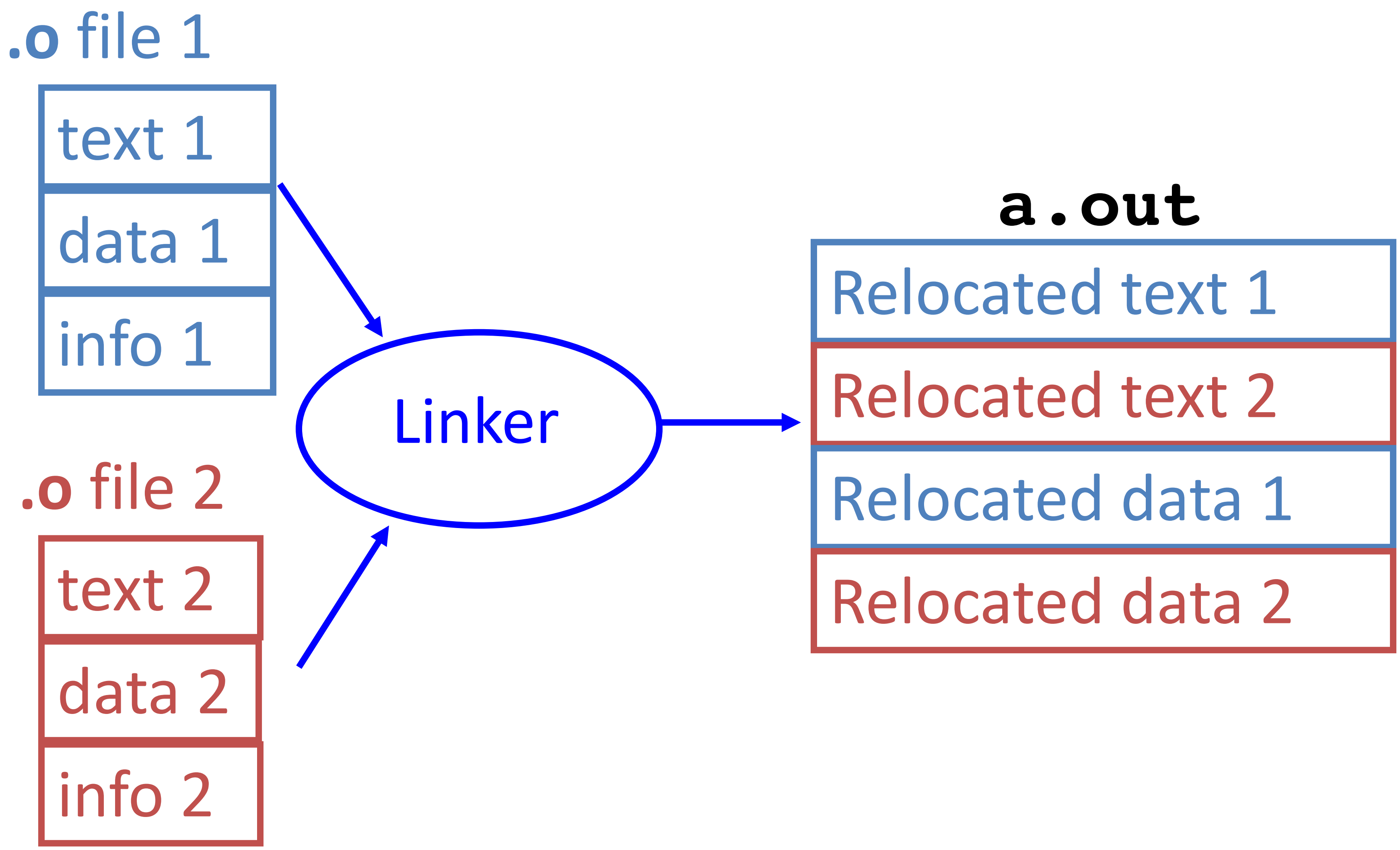
Where Are We Now?



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
  - That is, fill in all absolute addresses

Four Types of Addresses

- PC-Relative Addressing (**`beq`**, **`bne`**, **`jal`**)
  - Never need to relocate (PIC: position independent code)
- External Function Reference (usually **`jal`**)
  - Always relocate
- Static Data Reference (often **`auipc/addi`**)
  - Always relocate
  - RISC-V often uses **auipc** rather than **lui** so that a big block of stuff can be further relocated as long as it is fixed relative to the pc

Absolute Addresses in RISC-V

- Which instructions need relocation editing?
  - J-format: jump and link: ONLY for external jumps

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

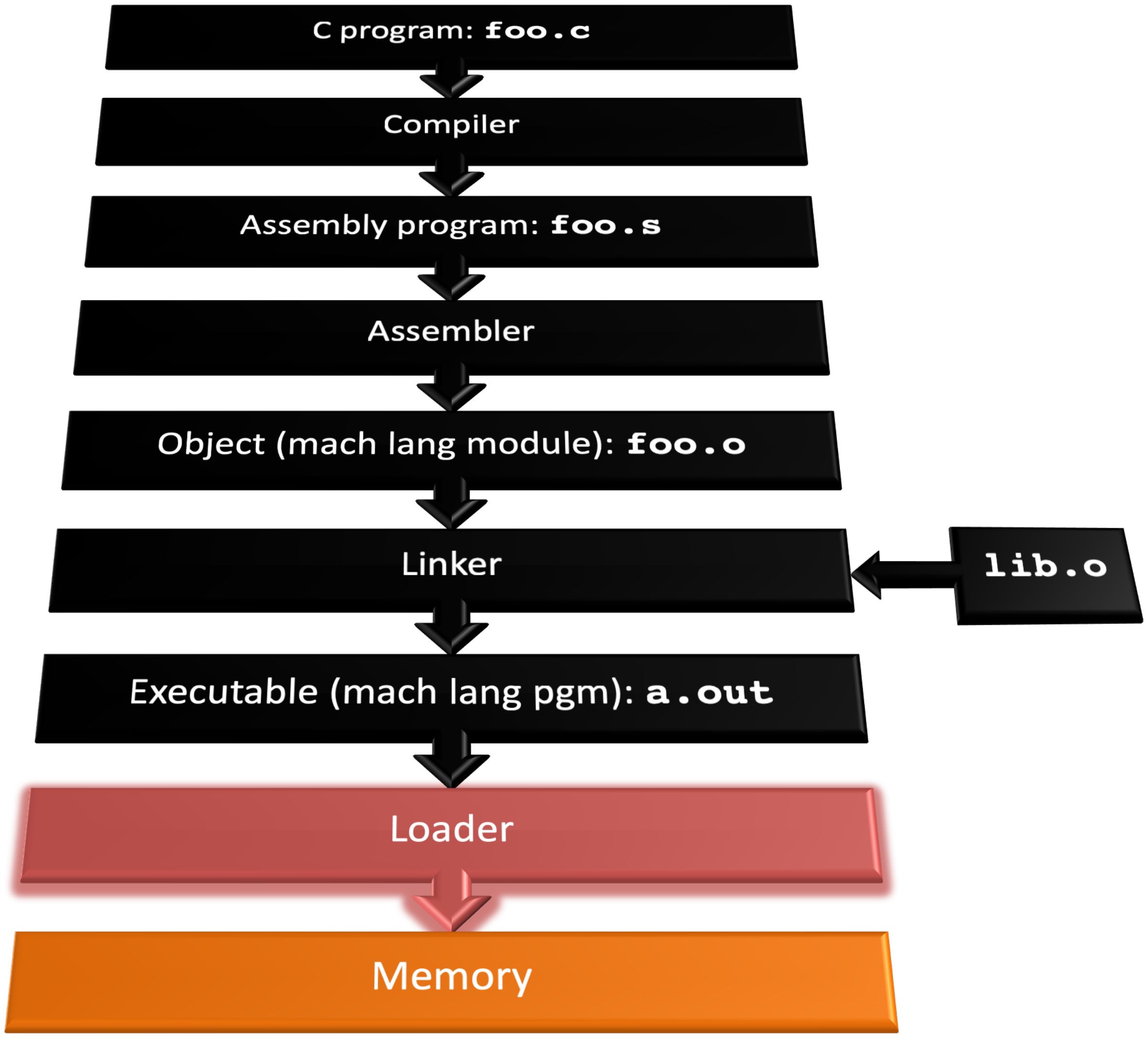
Resolving References (1/2)

- Linker assumes first word of first text segment is at address **`0x04000000`** 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 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

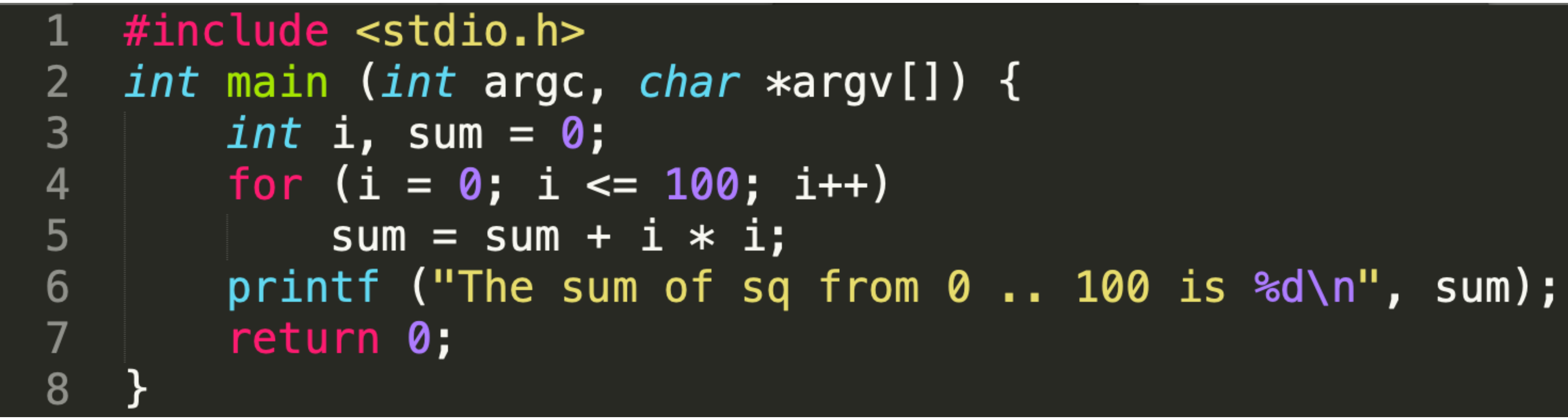
At what point in process are all the machine code bits determined for the following assembly instructions:

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

C: (1) After assembly, (2) After linking

Example: C ⇒ Asm ⇒ Obj ⇒ Exe ⇒ Run

C Program Source Code: `prog.c`



```
#include <stdio.h>
int main (int argc, char *argv[]) {
    int i, sum = 0;
    for (i = 0; i <= 100; i++)
        sum = sum + i * i;
    printf ("The sum of sq from 0 .. 100 is %d\n", sum);
    return 0;
}
```

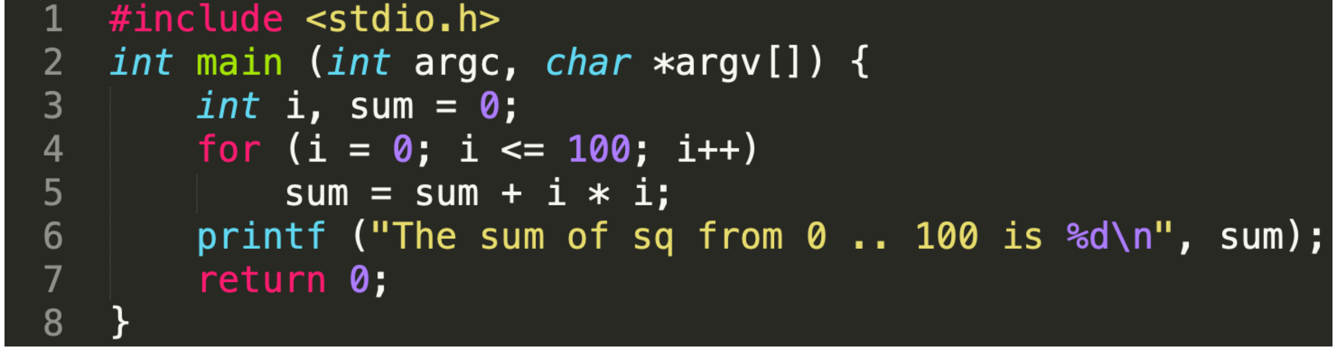
"`printf`" lives in "`libc`"

Compile to RISC-V Assembly: prog.s

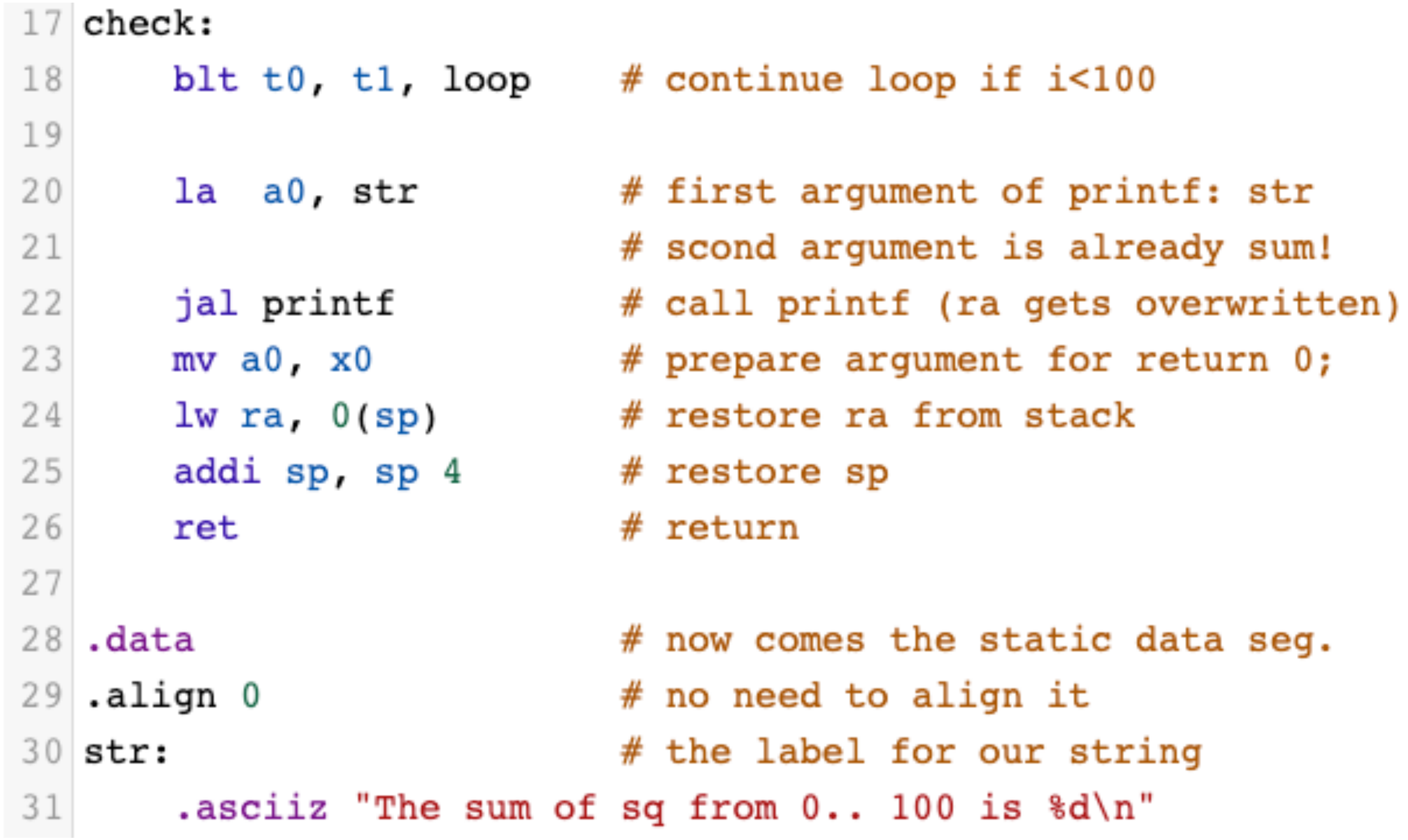
```c
#include <stdio.h>
int main (int argc, char *argv[]) {
    int i, sum = 0;
    for (i = 0; i <= 100; i++)
        sum = sum + i * i;
    printf ("The sum of sq from 0 .. 100 is %d\n", sum);
    return 0;
}
```

```
# Register Allocation: i = t0, sum = a1
.text                       # the text segment
.align 2                    # aligned to 2 byte (RV32C!)
.globl main                 # we have a global symbol "main"

main:
    addi sp, sp, -4         # reserve stack for ra
    sw ra, 0(sp)            # save ra on stack
    mv t0, x0               # initialize i with 0
    mv a1, x0               # initialize sum with 0
    li t1, 100              # set condition variable to 100
    j check                 # jump to check: for loop
loop:                           # checks first!
    mul t2, t0, t0          # loop code: t2 = i * i
    add a1, a1, t2          #            sum = sum + t2
    addi t0, t0, 1          # i++
```



```c
#include <stdio.h>
int main (int argc, char *argv[]) {
    int i, sum = 0;
    for (i = 0; i <= 100; i++)
        sum = sum + i * i;
    printf ("The sum of sq from 0 .. 100 is %d\n", sum);
    return 0;
}
```



```asm
check:
    blt t0, t1, loop     # continue loop if i<100

    la  a0, str          # first argument of printf: str
                         # scond argument is already sum!
    jal printf           # call printf (ra gets overwritten)
    mv a0, x0            # prepare argument for return 0;
    lw ra, 0(sp)         # restore ra from stack
    addi sp, sp 4        # restore sp
    ret                  # return

.data                    # now comes the static data seg.
.align 0                 # no need to align it
str:                     # the label for our string
    .asciiz "The sum of sq from 0.. 100 is %d\n"
```

Example: C ⇒ Asm ⇒ Obj ⇒ Exe ⇒ Run

Pseudo Instructions?

```
# i = t0, sum = a1
.text
.align 2
.globl main

main:
    addi sp, sp, -4
    sw ra, 0(sp)
    mv t0, x0
    mv a1, x0
    li t1, 100
    j check
loop:
    mul t2, t0, t0
    add a1, a1, t2
    addi t0, t0, 1
check:
    blt t0, t1, loop

    la  a0, str

    jal printf
    mv a0, x0
    lw ra, 0(sp)
    addi sp, sp 4
    ret

.data
.align 0
str:
    .asciiz "The sum of sq from 0.. 100 is %d\n"
```

7 Pseudo Instructions

```
# i = t0, sum = a1
.text
.align 2
.globl main

main:
     addi sp, sp, -4
     sw ra, 0(sp)
     mv t0, x0
     mv a1, x0
     li t1, 100
     j check
loop:
     mul t2, t0, t0
     add a1, a1, t2
     addi t0, t0, 1
check:
     blt t0, t1, loop

     la  a0, str

     jal printf
     mv a0, x0
     lw ra, 0(sp)
     addi sp, sp 4
     ret

.data
.align 0
str:
     .asciiz "The sum of sq from 0.. 100 is %d\n"
```

Assembly Step 1:
Remove Pseudo Instructions, assign jumps

Assigned jumps

Unknown addresses

Basic Code	Original Code	Label
addi x2 x2 -4	addi sp, sp, -4	main:
sw x1 0(x2)	sw ra, 0(sp)	
addi x5 x0 0	mv t0, x0	
addi x11 x0 0	mv a1, x0	
addi x6 x0 100	li t1, 100	
jal x0 16	j check	
mul x7 x5 x5	mul t2, t0, t0	loop:
add x11 x11 x7	add a1, a1, t2	
addi x5 x5 1	addi t0, t0, 1	
blt x5 x6 **-12**	blt t0, t1, loop	check:
auipc x10 l.str	la a0, str	
addi x10 x10 r.str	la a0, str	
jal x1 printf	jal printf	
addi x10 x0 0	mv a0, x0	
lw x1 0(x2)	lw ra, 0(sp)	
addi x2 x2 4	addi sp, sp 4	
jalr x0 x1 0	ret	

Assembly Step 1: Instructions and Labels have addresses!

PC	Basic Code	Original Code	Label
0x00	addi x2 x2 -4	addi sp, sp, -4	main:
0x04	sw x1 0(x2)	sw ra, 0(sp)	
0x08	addi x5 x0 0	mv t0, x0	
0x0c	addi x11 x0 0	mv a1, x0	
0x10	addi x6 x0 100	li t1, 100	
0x14	jal x0 16	j check	
0x18	mul x7 x5 x5	mul t2, t0, t0	loop:
0x1c	add x11 x11 x7	add a1, a1, t2	
0x20	addi x5 x5 1	addi t0, t0, 1	
0x24	blt x5 x6 -12	blt t0, t1, loop	check:
0x28	auipc x10 l.str	la a0, str	
0x2c	addi x10 x10 r.str	la a0, str	
0x30	jal x1 printf	jal printf	
0x34	addi x10 x0 0	mv a0, x0	
0x38	lw x1 0(x2)	lw ra, 0(sp)	
0x3c	addi x2 x2 4	addi sp, sp 4	
0x40	jalr x0 x1 0	ret	

Assembly Step 2: Create relocation table and symbol table

- Symbol Table

Label	address (in module)	Type
main:	0x00000000	global text
loop:	0x00000018	local text
check:	0x00000024	local text
str:	0x00000000	local data

- Relocation Table

Address	Instr. type	Dependency
0x0000000028	auipc	l.str
0x000000002c	addi	r.str
0x0000000030	jal	printf

Assembly Step 3:

- Generate object (.o) file:
  - Output binary representation for
    - text segment (instructions)
    - data segment (data)
    - symbol and relocation tables
  - Using dummy “placeholders” for unresolved absolute and external references

Example: C ⇒ Asm ⇒ Obj ⇒ Exe ⇒ Run
Text segment of Assembled prog.s: prog.o

PC	Machine Code	Basic Code	Original Code	Label
0x00	0xFFC10113	addi x2 x2 -4	addi sp, sp, -4	main:
0x04	0x00112023	sw x1 0(x2)	sw ra, 0(sp)	
0x08	0x00000293	addi x5 x0 0	mv t0, x0	
0x0c	0x00000593	addi x11 x0 0	mv a1, x0	
0x10	0x06400313	addi x6 x0 100	li t1, 100	
0x14	0x0100006F	jal x0 16	j check	
0x18	0x025283B3	mul x7 x5 x5	mul t2, t0, t0	loop:
0x1c	0x007585B3	add x11 x11 x7	add a1, a1, t2	
0x20	0x00128293	addi x5 x5 1	addi t0, t0, 1	
0x24	0xFE62CAE3	blt x5 x6 -12	blt t0, t1, loop	check:
0x28	0x**00000**517	auipc x10 **0**	la a0, str	
0x2c	0x**000**50513	addi x10 x10 **0**	la a0, str	
0x30	0x**00000**0EF	jal x1 **0**	jal printf	
0x34	0x00000513	addi x10 x0 0	mv a0, x0	
0x38	0x00012083	lw x1 0(x2)	lw ra, 0(sp)	
0x3c	0x00410113	addi x2 x2 4	addi sp, sp 4	
0x40	0x00008067	jalr x0 x1 0	ret	

Example: C ⇒ Asm ⇒ Obj ⇒ Exe ⇒ Run

Move text segment to text location

PC	Machine Code	Basic Code	Original Code	Label
00400000	0xFFC10113	addi x2 x2 -4	addi sp, sp, -4	main:
00400004	0x00112023	sw x1 0(x2)	sw ra, 0(sp)	
00400008	0x00000293	addi x5 x0 0	mv t0, x0	
0040000c	0x00000593	addi x11 x0 0	mv a1, x0	
00400010	0x06400313	addi x6 x0 100	li t1, 100	
00400014	0x0100006F	jal x0 16	j check	
00400018	0x025283B3	mul x7 x5 x5	mul t2, t0, t0	loop:
0040001c	0x007585B3	add x11 x11 x7	add a1, a1, t2	
00400020	0x00128293	addi x5 x5 1	addi t0, t0, 1	
00400024	0xFE62CAE3	blt x5 x6 -12	blt t0, t1, loop	check:
00400028	0x**00000**517	auipc x10 **0**	la a0, str	
0040002c	0x**000**50513	addi x10 x10 **0**	la a0, str	
00400030	0x**00000**0EF	jal x1 **0**	jal printf	
00400034	0x00000513	addi x10 x0 0	mv a0, x0	
00400038	0x00012083	lw x1 0(x2)	lw ra, 0(sp)	
0040003c	0x00410113	addi x2 x2 4	addi sp, sp 4	
00400040	0x00008067	jalr x0 x1 0	ret	

Example: C ⇒ Asm ⇒ Obj ⇒ <u>Exe</u> ⇒ Run
Linking: PC relative static data str!

- Static Data str
  - Above text segment, so assume: 0x00401B08
  - `la a0 str   =>`
    ```
    auipc x10 ?????
    addi  x10 ???
    ```
  - PC relative addr with auipc!
    - Can move entire program around!
  - auipc at address: 0x00400028

  => (str) 0x00401B08 = (PC auipc) 0x00400028 + offset =>
  offset = 0x1AE0
  - represent 0x1AE0 as auipc/ addi pair:
    - addi immediate: 0xAE0
    - addi with Two's Complement => -1312 =>
      need to add 1 to auipc immediate
    - auipc immediate: 0x00002

Example: C ⇒ Asm ⇒ Obj ⇒ Exe ⇒ Run
Linking: PC relative to printf!

- Libc was linked to executable
  - Assume printf at: 0x0040C4F
  - `jal printf   =>`
    `jal x1 ?????`
  - PC relative addr!
    - Can move entire program around!
  - jal at address: 0x00400030

=> (printf) 0x00400C4F = (PC jal) 0x00400030 + offset =>
offset = 0xC1F

Example: C ⇒ Asm ⇒ Obj ⇒ Exe ⇒ Run

Text segment of Linked prog.o: a.out

PC	Machine Code	Basic Code	Original Code	Label
00400000	0xFFC10113	addi x2 x2 -4	addi sp, sp, -4	main:
00400004	0x00112023	sw x1 0(x2)	sw ra, 0(sp)	
00400008	0x00000293	addi x5 x0 0	mv t0, x0	
0040000c	0x00000593	addi x11 x0 0	mv a1, x0	
00400010	0x06400313	addi x6 x0 100	li t1, 100	
00400014	0x0100006F	jal x0 16	j check	
00400018	0x025283B3	mul x7 x5 x5	mul t2, t0, t0	loop:
0040001c	0x007585B3	add x11 x11 x7	add a1, a1, t2	
00400020	0x00128293	addi x5 x5 1	addi t0, t0, 1	
00400024	0xFE62CAE3	blt x5 x6 -12	blt t0, t1, loop	check:
00400028	0x**00002**517	auipc x10 **2**	la a0, str	
0040002c	0x**AE0**50513	addi x10 x10 **-1312**	la a0, str	
00400030	0x**00C1F**0EF	jal x1 **0xC1F**	jal printf	
00400034	0x00000513	addi x10 x0 0	mv a0, x0	
00400038	0x00012083	lw x1 0(x2)	lw ra, 0(sp)	
0040003c	0x00410113	addi x2 x2 4	addi sp, sp 4	
00400040	0x00008067	jalr x0 x1 0	ret	

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

`en.wikipedia.org/wiki/Dynamic_linking`

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
  - – Thus "containers": We hate dependencies, so we are just going to ship around all the libraries and everything else as part of the 'application'

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

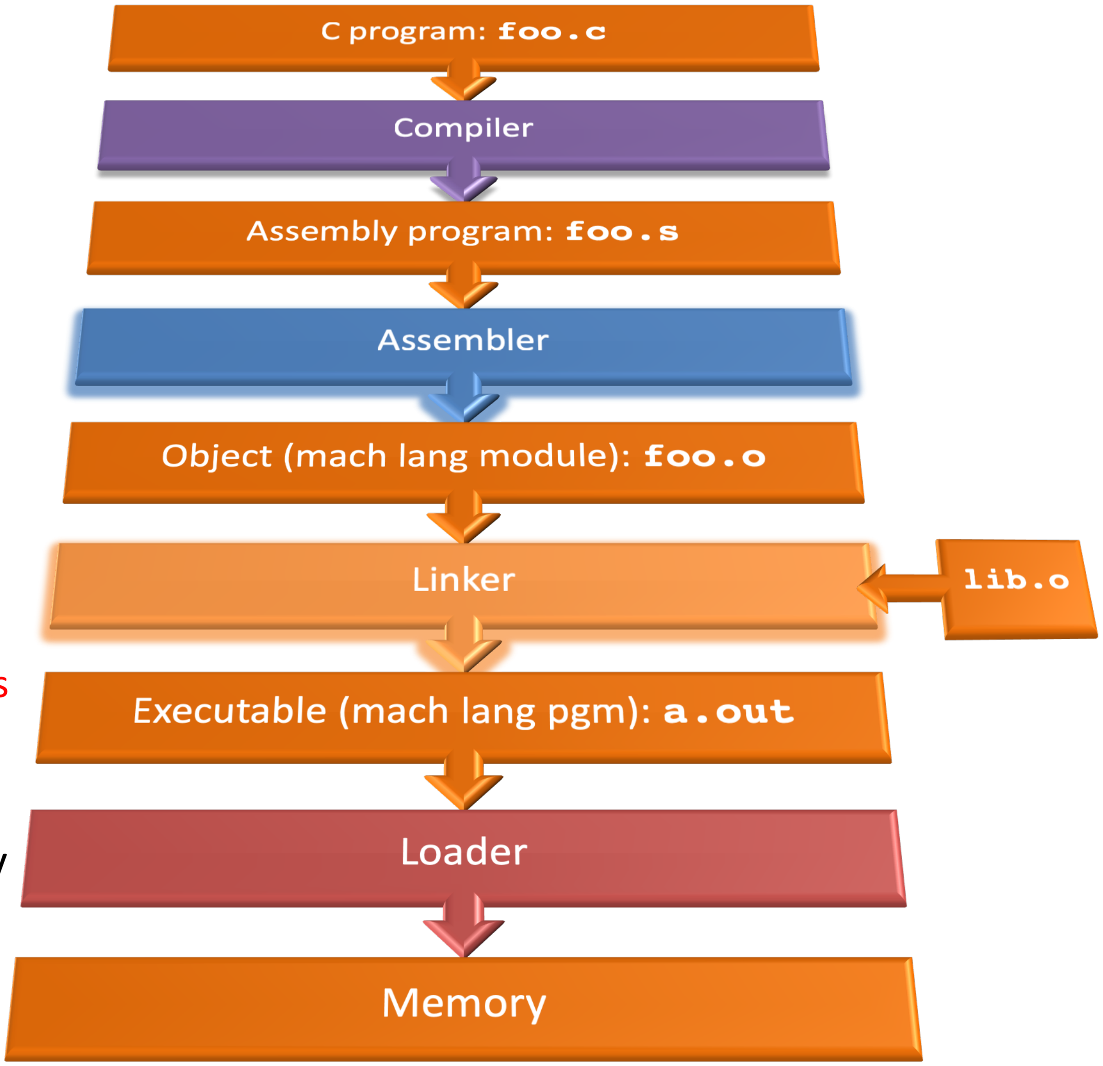
- The prevailing approach to dynamic linking uses machine code as the “lowest common denominator”
  - The linker does not use information about how the program or library was compiled (i.e., what compiler or language)
  - This can be described as “linking at the machine code level”
  - This isn’t the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

Address Space Layout Randomization

- With C memory errors, attackers traditionally often were able to jump to interesting functions of libraries (“Return oriented programming”)
  - E.g.: overwrite the ra saved on the stack to jump to another function!
- Randomized layout for libraries during linking => cannot predict address of function without linker info =>
- Attackers cannot easily jump to existing code
- Attackers need this, because with Virtual Memory, we can mark heap & stack as unexecutable!

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.


C program: foo.c
Compiler
Assembly program: foo.s
Assembler
Object (mach lang module): foo.o
Linker
lib.o
Executable (mach lang pgm): a.out
Loader
Memory