CSE 31
Computer Organization

Bonus Lecture – Program Process

Program Process - Overview

- Interpretation vs Translation
- Translating C Programs
  - Compiler
  - Assembler
  - Linker
  - Loader
- An Example

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 improve performance

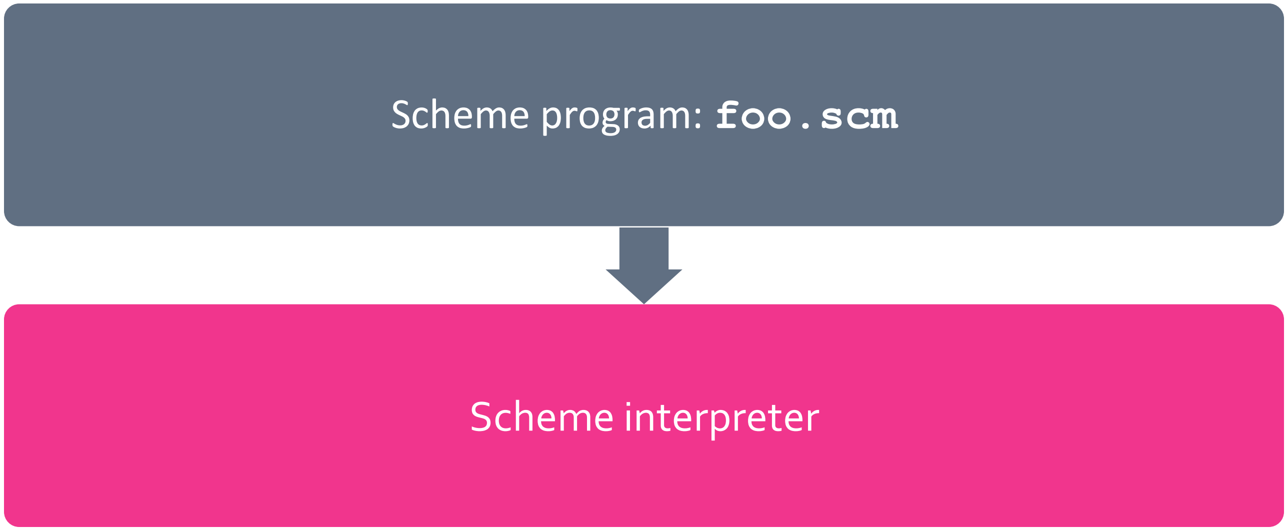
Matlab
Scheme Java C++ C Assembly Java bytecode machine language

Easy to program Difficult to program
Inefficient to interpret Efficient to interpret

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 Scheme program **foo.scm**

Interpretation



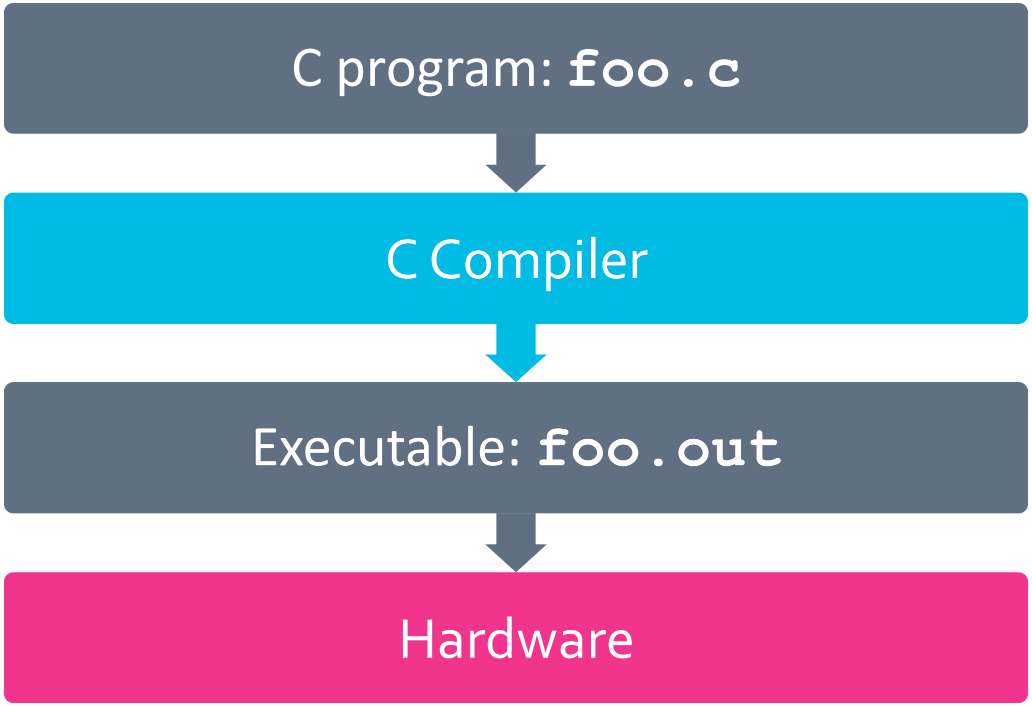
- Scheme Interpreter is just a program that reads a scheme program and performs the functions of that scheme program.

The process happens in run-time

Translation

- C Compiler is a translator from C to machine language.

- The processor is a hardware interpreter of machine language.



Interpretation

- Any good reason to interpret machine language in software?
  - MARS – useful for learning / debugging

- Apple Macintosh conversion
  - Switched from Motorola 680x0 instruction architecture to PowerPC.
    - » Similar issue with switching to x86 and M1.
  - 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
    - » Through emulation

Interpretation vs. Translation? (1/2)

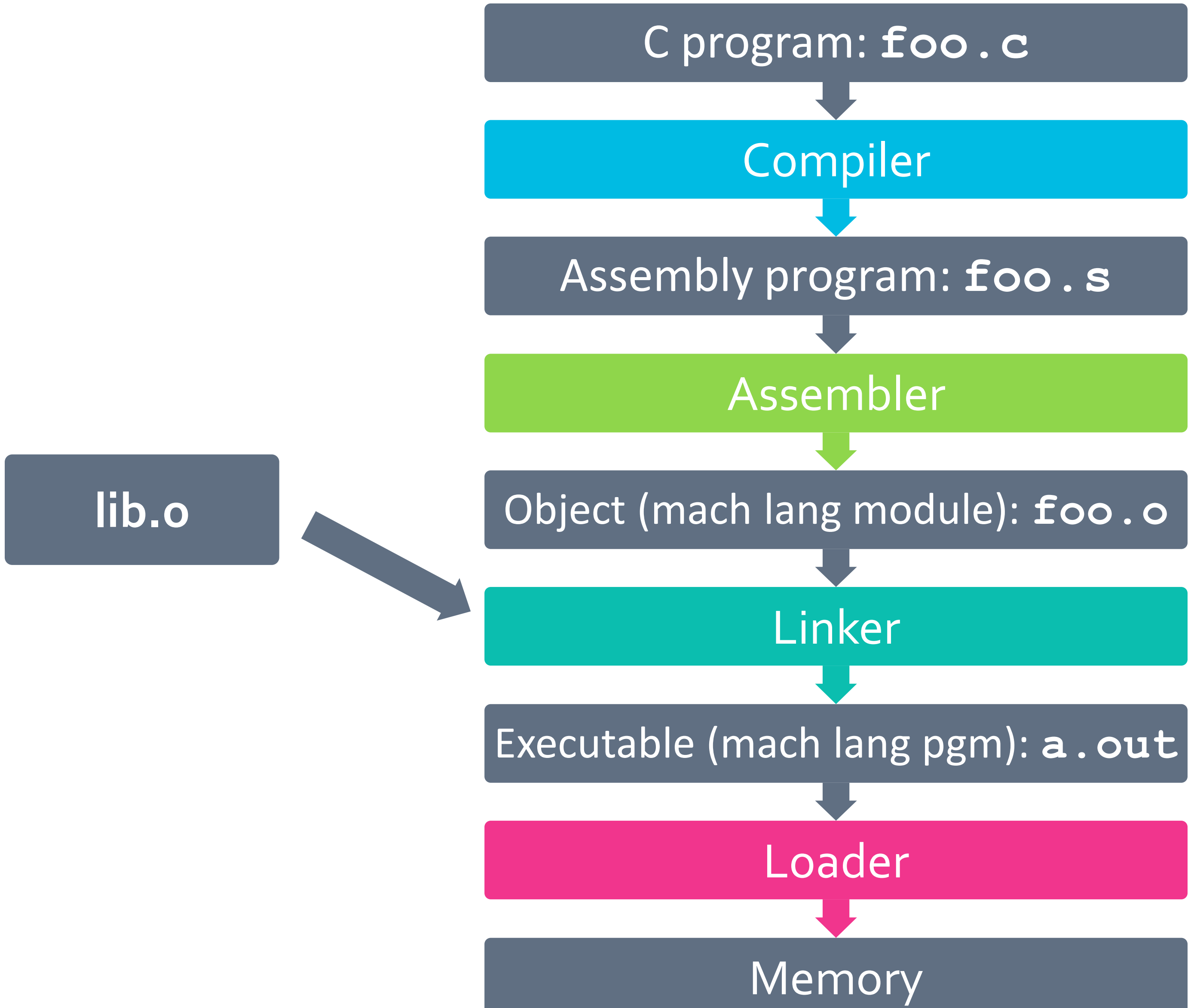
- Generally easier to write programs with interpreter
- Interpreter closer to high-level, so can give better error messages (e.g., MARS)
  - Translator reaction: add extra information to help debugging (line numbers, names)
- Interpreter slower (10x?), code smaller (2x?)
- Interpreter provides instruction 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/Apple keeps all their source code secret)
  - Alternative model, “open source”, creates value by publishing the source code and fostering a community of developers.

Steps to Starting a Program (translation)



Compiler

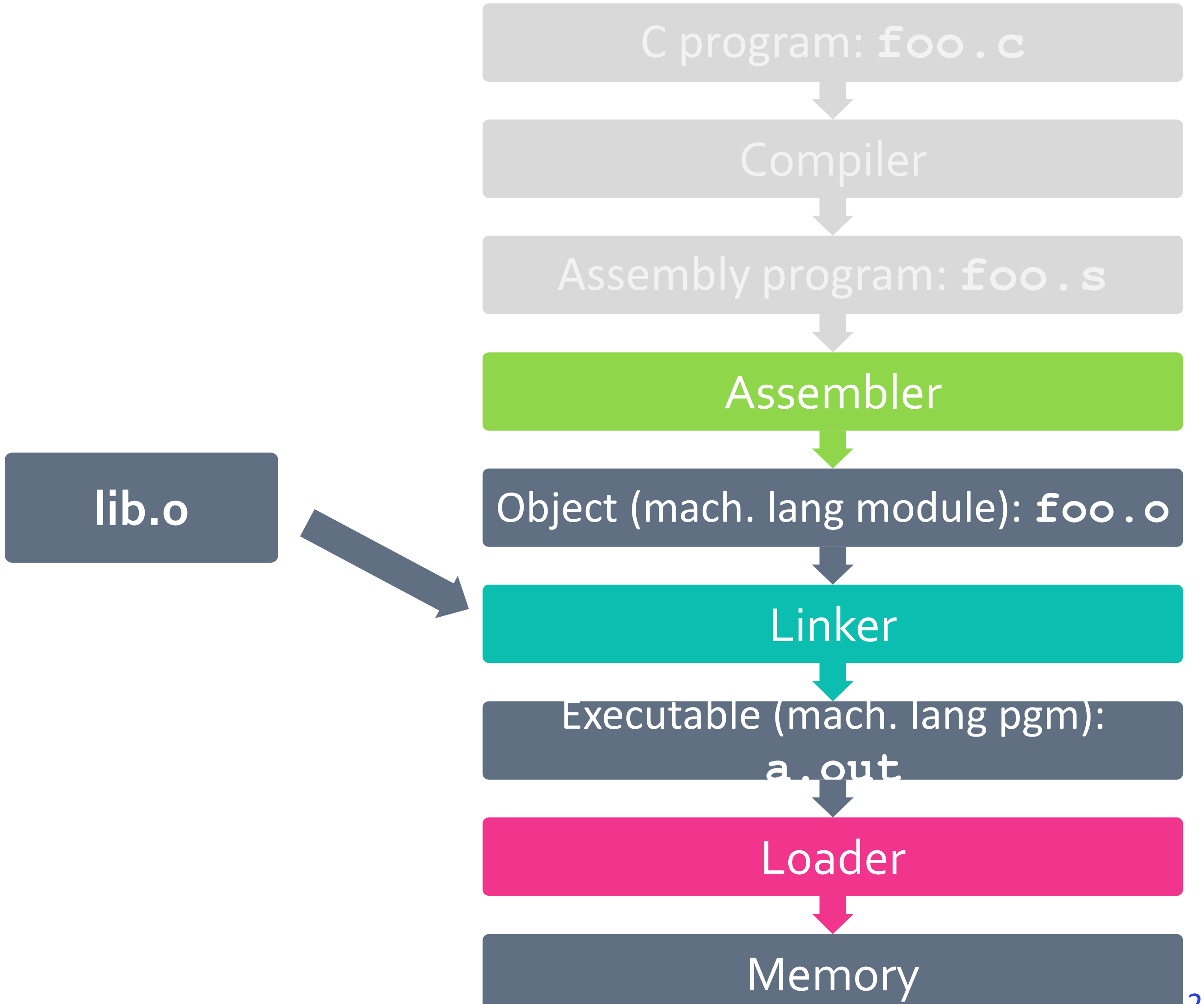
- Input: High-Level Language Code
  (e.g., C program such as `foo.c`)

- Output: Assembly Language Code
  (e.g., Assembly program `foo.s` for MIPS)

- Note: Output *may* contain pseudo-instructions
  - Pseudo-instructions: instructions that assembler understands but not in machine (previous lectures)
  - For example:

    `mov $s1,$s2` → **`or $s1,$s2,$zero`**

Where Are We Now?



Assembler

- Input: Assembly Language Code (e.g., `foo.s` for MIPS)
- Output: Object Code, information tables (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, program memory)
  - `.data`: Subsequent items put in user data segment (binary rep of data in source file, data memory)
  - `.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 uses convenient variations of machine language instructions

Pseudo:	Real:
`subu $sp,$sp,32`	`addiu $sp,$sp,-32`
`sd $a0, 32($sp)`	`sw $a0, 32($sp)` `sw $a1, 36($sp)`
`mul $t7,$t6,$t5`	`mult $t6,$t5` `mflo $t7`
`addu $t0,$t6,1`	`addiu $t0,$t6,1`
`ble $t0,100,loop`	`slti $at,$t0,101` `bne $at,$0,loop`
`la $a0, str`	`lui $at,left(str)` `ori $a0,$at,right(str)`

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
  - 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:

```
    or   $v0, $0,  $0
L1: slt  $t0, $0,  $a1
    beq  $t0, $0,  L2
    addi $a1, $a1, -1
    j    L1
L2: add  $t1, $a0, $a1
```

How does it know where in the future steps?

  - Solved by taking 2 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` and `jal`)?
  - Jumps require absolute address.
  - So, forward or not, still can't generate machine instruction without knowing the position of instructions in memory.

- What about references to data?
  - `la` gets broken up into `lui` and `ori`
  - These will require the full 32-bit address of the data.

- These can't be determined yet, so we create two tables…

Symbol Table

- Records list of "items" in this file that may be used by other files. (What you have)

- What are they?
  - Labels: function calling
  - Data: anything in the **.data** section; variables which may be accessed across files

Relocation Table

- List of "items" for which this file needs the address later.

- What are they?
  - Any label jumped to: **`j`** or **`jal`**
    - » Internal
    - » External (including `lib` files)
  - Any piece of data
    - » such as the **`la`** instruction

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

Quiz

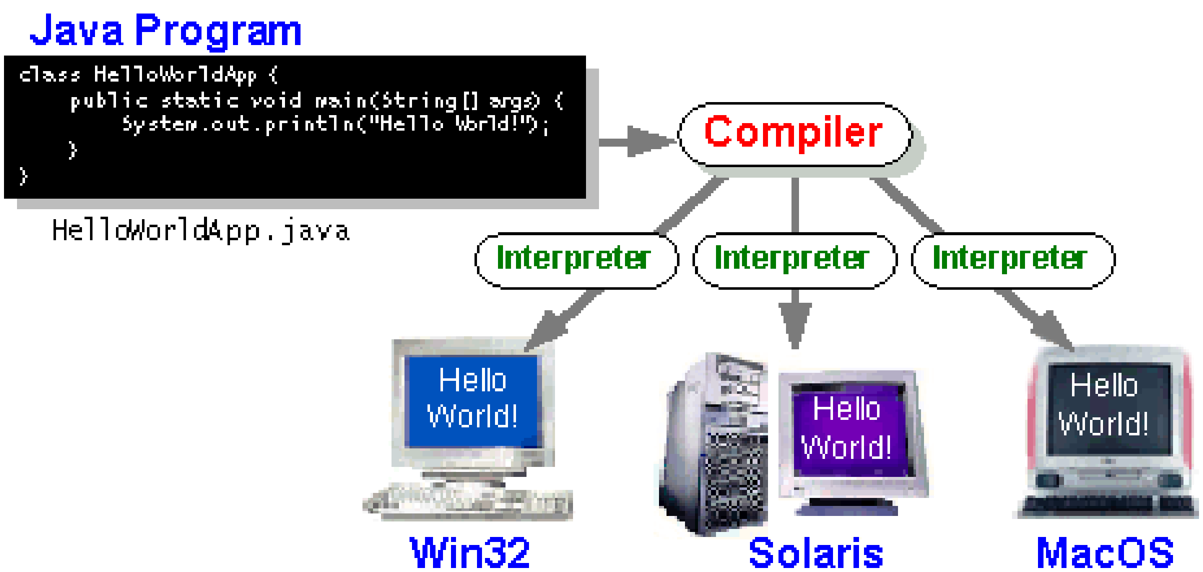
1) Assembler will ignore the instruction **`Loop:  nop`** (not an operation) because it does nothing.

2) Java designers used a translator AND interpreter (rather than just a translator) mainly because of (at least 1 of): ease of writing, better error messages, smaller object code.

```
    12
a)  FF
b)  FT
c)  TF
d)  TT
```

Quiz

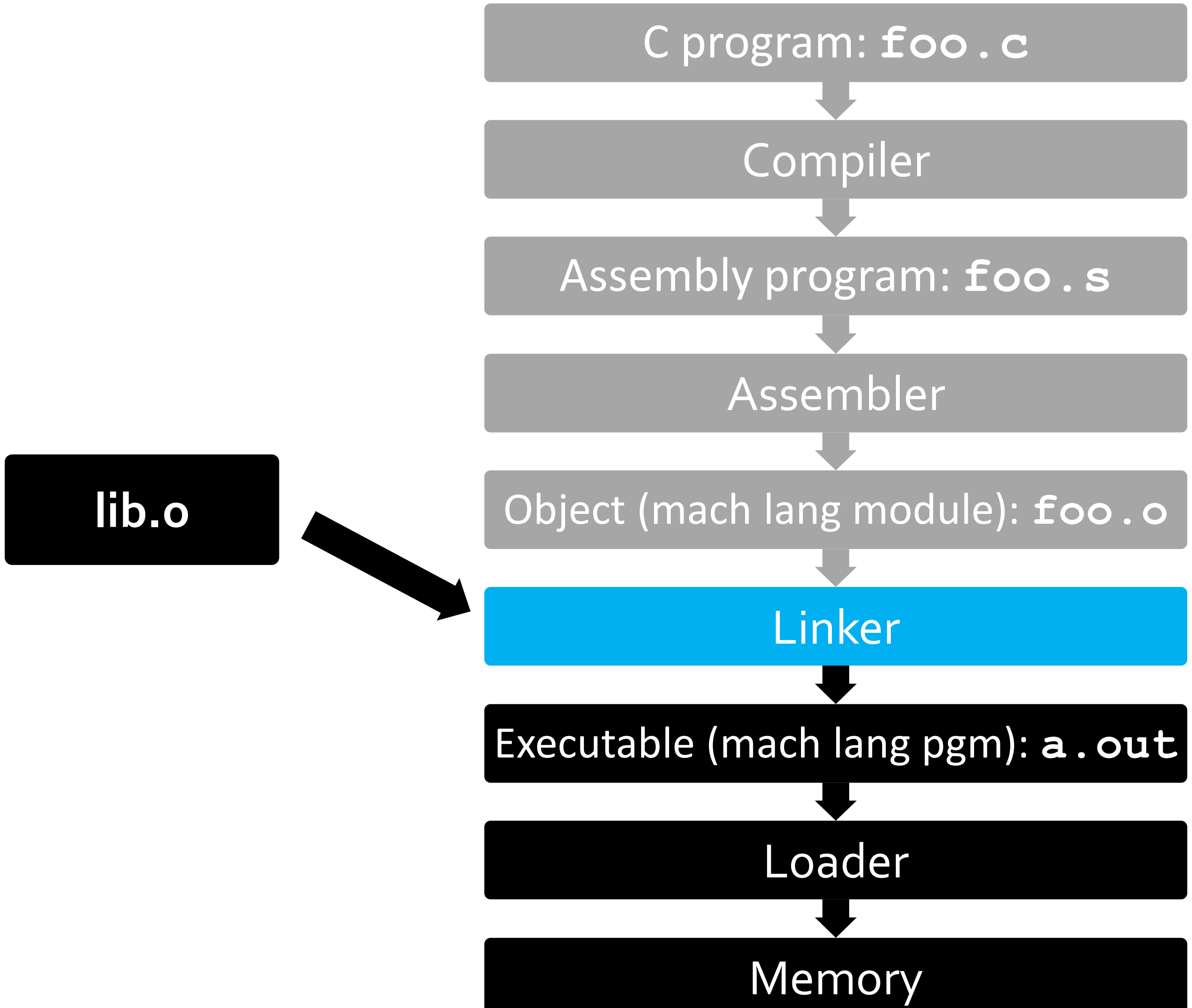
1) Assembler keeps track of all labels in symbol table…F!



2) Java designers used
both mainly because of code portability…F!

	12
a)	**FF**
b)	FT
c)	TF
d)	TT

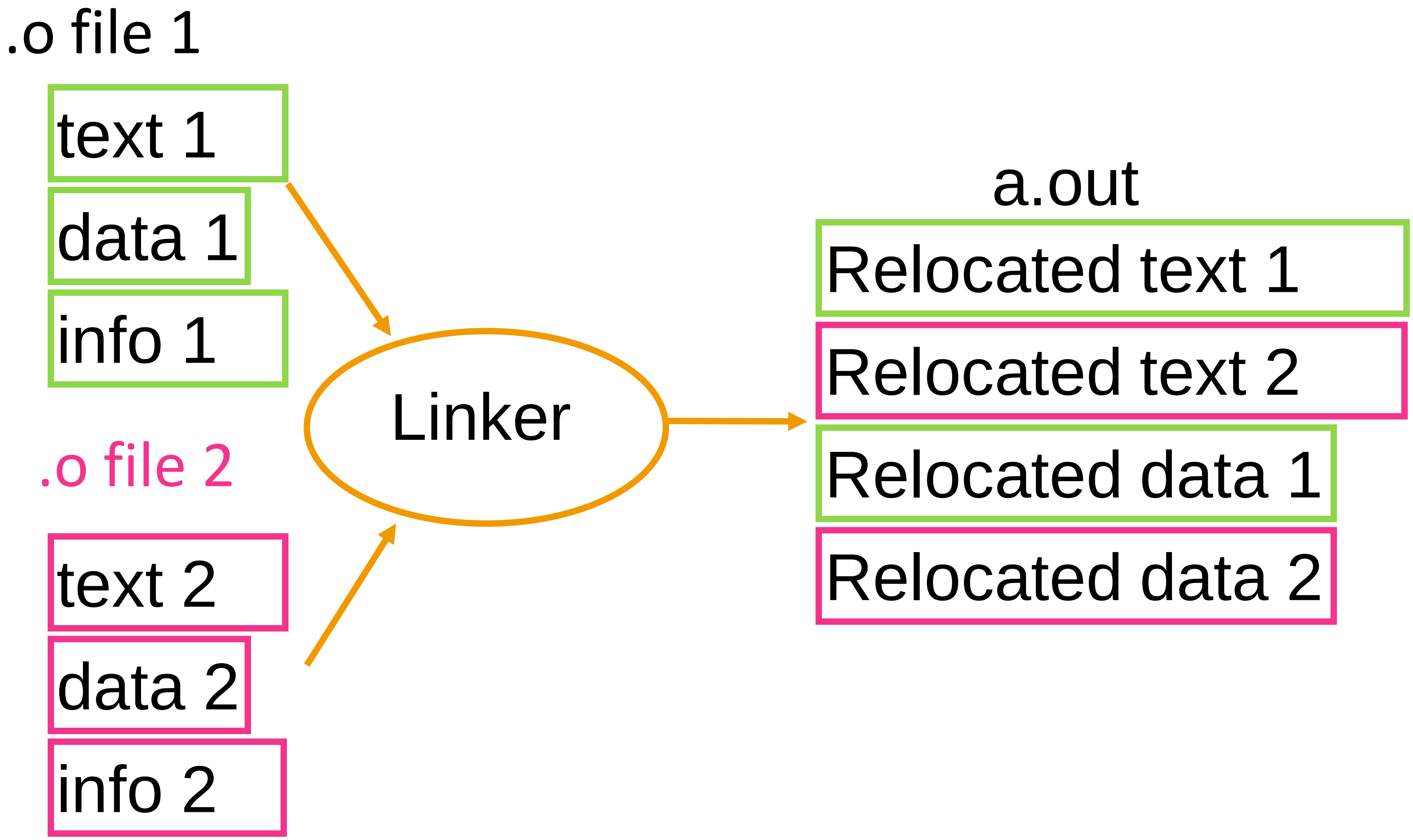
Steps to Starting a Program (translation)



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 whole program
    - » Windows 10 source is ~ 50 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`)
  - never relocate
- Absolute Address (`j`, `jal`)
  - always relocate
- External Reference (usually `jal`)
  - always relocate
- Data Reference (often `lui` and `ori`/load address)
  - always relocate

Absolute Addresses in MIPS

- Which instructions need relocation editing?
  - J-format: jump, jump and link (`jal`)

J/JAL	XXXXX

  - Loads and stores to variables in static area, relative to global pointer

LW/SW	$gp	$x	address

  - What about conditional branches?

beq/bne	$rs	$rt	address

    - » PC-relative addressing preserved even if code moves

Resolving References (1/2)

- Linker assumes first word of first text segment is at address **`0x00000000`**.

- 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)

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 <u>entire</u> library even if not all of it will be used.
  - Executable is self-contained.

- An alternative is dynamically linked libraries (DLL), common on Windows & UNIX 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

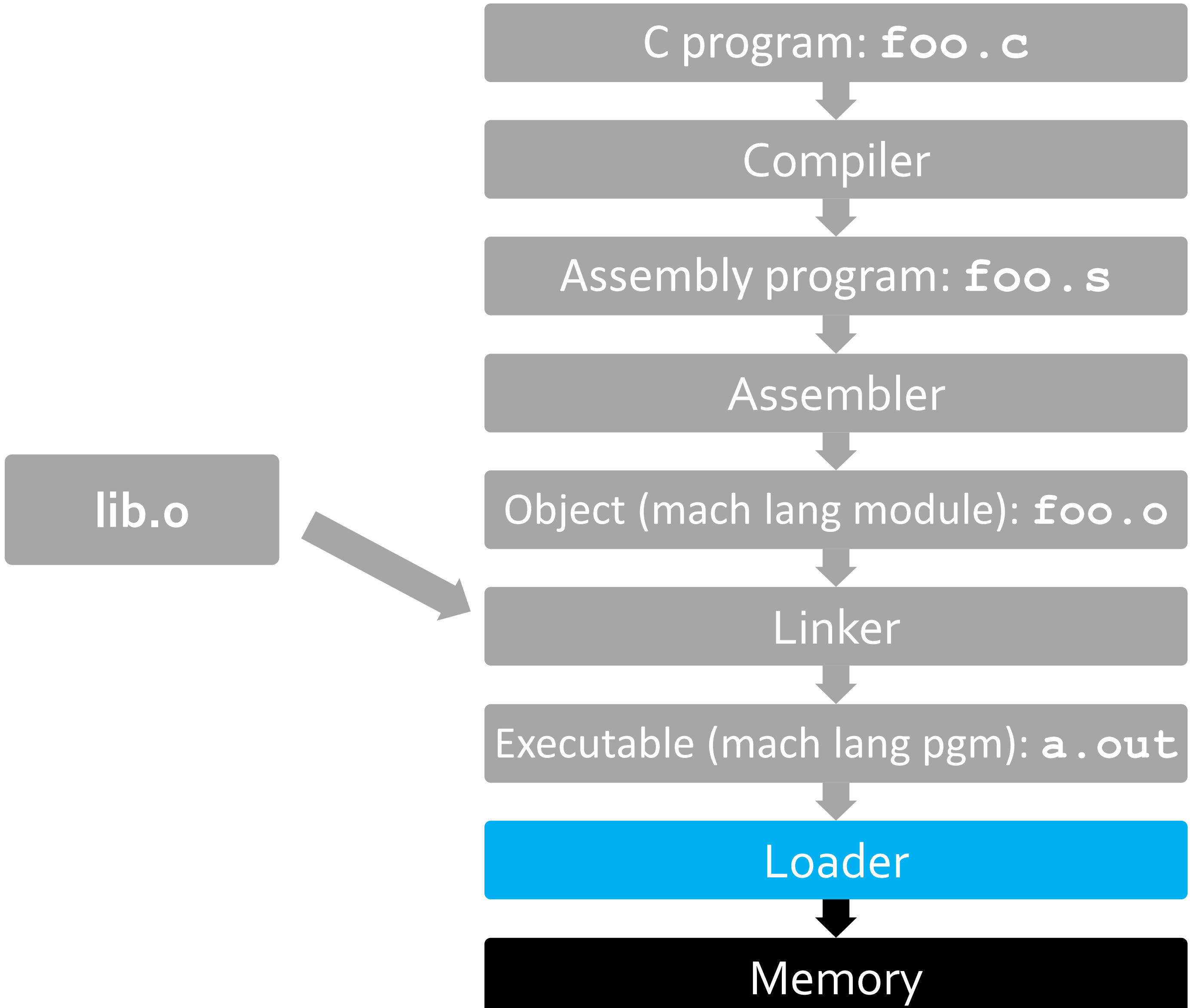
Overall, dynamic linking adds quite a bit of complexity to the compiler, linker, and operating system. However, it provides many benefits that often outweigh these.

en.wikipedia.org/wiki/Dynamic_linking

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”
  - There are other ways to achieve the same purpose

Steps to Starting a Program (translation)



Loader (1/3)

- Input: Executable Code
  (e.g., `a.out` for MIPS)

- 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 running.

- In reality, loader is the operating system (OS)
  - loading is one of the OS tasks

Loader (2/3)

- So what does a loader do?
  - Reads executable file's header to determine size of text and data segments
  - Creates new address space (static memory) 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

Loader (3/3)

- Copies arguments passed to the program onto the stack
- Initializes machine registers
  - Most registers cleared
  - 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

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);
}
```

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

Compilation: MAL

```
__.text
   .align  2
   .globl  main
main:
   subu $sp, $sp, 40
   sw $ra, 20($sp)
   sd $a0, 32($sp)
   sw $0, 24($sp)
   sw $0, 28($sp)
loop:
   lw $t6, 28($sp)
   mul $t7, $t6, $t6
   lw $t8, 24($sp)
   addu $t9, $t8, $t7
   sw $t9, 24($sp)
```

```
   addu $t0, $t6, 1
   sw $t0, 28($sp)
   ble $t0, 100, loop
   la $a0, str
   lw $a1, 24($sp)
   jal printf
   move $v0, $0
   lw $ra, 20($sp)
   addiu $sp, $sp, 40
   jr $ra
   .data
   .align  0
str:
   .asciiz "The sum
   of sq from 0 ..
   100 is %d\n"
```

Where are the 7 pseudo-instructions?

Compilation: MAL

```
    .text
    .align  2
    .globl  main
main:
    subu $sp, $sp, 40
    sw $ra, 20($sp)
    sd $a0, 32($sp)
    sw $0, 24($sp)
    sw $0, 28($sp)
loop:
    lw $t6, 28($sp)
    mul $t7, $t6, $t6
    lw $t8, 24($sp)
    addu $t9, $t8, $t7
    sw $t9, 24($sp)
    addu $t0, $t6, 1
    sw $t0, 28($sp)
    ble $t0, 100, loop
    la $a0, str
    lw $a1, 24($sp)
    jal printf
    move $v0, $0
    lw $ra, 20($sp)
    addiu $sp, $sp, 40
    jr $ra
    .data
    .align  0
str:
    .asciiz "The sum of sq from 0 .. 100 is %d\n"
```

7 pseudo-instructions underlined

Assembly step 1:

Remove pseudo instructions, assign addresses

```
00 addiu $29, $29, -40
04 sw    $31, 20($29)
08 sw    $4, 32($29)
0c sw    $5, 36($29)
10 sw    $0, 24($29)
14 sw    $0, 28($29)
18 lw    $14, 28($29)
1c multu $14, $14
20 mflo  $15
24 lw    $24, 24($29)
28 addu  $25, $24, $15
2c sw    $25, 24($29)
```

```
30 addiu $8, $14, 1
34 sw    $8,28($29)
38 slti  $1, $8, 101
3c bne   $1, $0, loop
40 lui   $4, l.str
44 ori   $4, $4, r.str
48 lw    $5, 24($29)
4c jal   printf
50 add   $2, $0, $0
54 lw    $31,20($29)
58 addiu $29,$29,40
5c jr    $31
```

Recall

- Symbol Table: Records 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” for which this file needs the address later.
  - What are they?
    - » Any label jumped to: **`j`** or **`jal`**
      - Internal
      - External (including `lib` files)
    - » Any piece of data
      - such as the **`la`** instruction

Assembly step 2

Create relocation table and symbol table

- Symbol Table

Label	address (in module)	type
`main:`	`0x00000000`	`global text`
`loop:`	`0x00000018`	`local text`
`str:`	`0x00000000`	`local data`

- Relocation Information

Address	Instr. type	Dependency
`0x00000040`	`lui`	`l.str`
`0x00000044`	`ori`	`r.str`
`0x0000004c`	`jal`	`printf`

Assembly step 3

Resolve local PC-relative labels

```
00 addiu $29 ,$29, -32
04 sw    $31, 20($29)
08 sw    $4, 32($29)
0c sw    $5, 36($29)
10 sw    $0, 24($29)
14 sw    $0, 28($29)
18 lw    $14, 28($29)
1c multu $14, $14
20 mflo  $15
24 lw    $24, 24($29)
28 addu  $25, $24, $15
2c sw    $25, 24($29)
```

```
30 addiu $8, $14, 1
34 sw    $8, 28($29)
38 slti  $1, $8, 101
3c bne   $1, $0, -10
40 lui   $4, l.str
44 ori   $4, $4, r.str
48 lw    $5,24($29)
4c jal   printf
50 add   $2, $0, $0
54 lw    $31, 20($29)
58 addiu $29, $29, 32
5c jr    $31
```

Assembly step 4

- 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.

Text segment in object file

0x000000	00100111101111011111111111101000
0x000004	10101111101111110000000000010100
0x000008	10101111101001000000000000100000
0x00000c	10101111101001010000000000100100
0x000010	10101111101000000000000000011000
0x000014	10101111101000000000000000011100
0x000018	10001111101011100000000000011100
0x00001c	10001111110111000000000000011000
0x000020	00000001110011100000000000011001
0x000024	00100101110010000000000000000001
0x000028	00101001000000010000000001100101
0x00002c	10101111101010000000000000011100
0x000030	00000000000000000111100000010010
0x000034	00000011000011111100100000100001
0x000038	00010100001000001111111111110111
0x00003c	10101111101110010000000000011000
0x000040	0011110000000100**0000000000000000**
0x000044	1000111110100101**0000000000000000**
0x000048	00001100000100000000000011101100
0x00004c	001001**00000000000000000000000000**
0x000050	10001111101111110000000000010100
0x000054	00100111101111010000000000100000
0x000058	00000011111000000000000000001000
0x00005c	00000000000000000001000000100001

Link 1: combine `prog.o`, `libc.o`

- Merge text/data segments
- Create absolute memory addresses
- Modify & merge symbol and relocation tables
- Symbol Table
  - Label / Address

Label	Address	
`main:`	`0x00000000`	
`loop:`	`0x00000018`	
`str:`	`0x10000430`	
`printf:`	`0x000003b0`	…

- Relocation Information
  - Address / Instr. Type / Dependency

Address	Instr. Type	Dependency	
`0x00000040`	`lui`	`l.str`	
`0x00000044`	`ori`	`r.str`	
`0x0000004c`	`jal`	`printf`	…

Link step 2:

Edit Addresses in relocation table

- (shown in TAL for clarity, but done in binary)

```
00 addiu  $29,$29,-32
04 sw     $31,20($29)
08 sw     $4, 32($29)
0c sw     $5, 36($29)
10 sw     $0, 24($29)
14 sw     $0, 28($29)
18 lw     $14, 28($29)
1c multu  $14, $14
20 mflo   $15
24 lw     $24, 24($29)
28 addu   $25,$24,$15
2c sw     $25, 24($29)
```

```
30 addiu  $8,$14, 1
34 sw     $8,28($29)
38 slti   $1,$8, 101
3c bne    $1,$0, -10
40 lui    $4, 0x1000
44 ori    $4,$4,0x0430
48 lw     $5,24($29)
4c jal    0x000003b0
50 add    $2, $0, $0
54 lw     $31,20($29)
58 addiu  $29,$29,32
5c jr     $31
```

Link step 3:

- Output executable of merged modules.
  - Single text (instruction) segment
  - Single data segment
  - Header detailing size of each segment

- NOTE:
  - The preceding example was a much simplified version of how ELF and other standard formats work, meant only to demonstrate the basic principles.

Things to Remember (1/2)

- 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.

Things to Remember (2/2)

- ***Stored Program*** concept is very powerful. It means that instructions sometimes act just like data.
  - Therefore, we can use programs to manipulate other programs!

- Compiler $\rightarrow$ Assembler $\rightarrow$ Linker ($\rightarrow$ Loader)

Quiz

Which of the following instructions may need to be edited during link phase?

```
Loop:   lui $at, 0xABCD            } #1
        ori $a0,$at, 0xFEDC        }
        bne $a0,$v0, Loop            #2
```

Quiz

Which of the following instructions may need to be edited during link phase?

```
Loop:   lui $at, 0xABCD          } #1
        ori $a0,$at, 0xFEDC      }
        bne $a0,$v0, Loop          #2
```

1: data reference; relocate   Yes
2: PC-relative branch; OK   No