CS61C – Machine Structures

Lecture 19 - Running a Program II

aka Compiling, Assembling, Linking, Loading

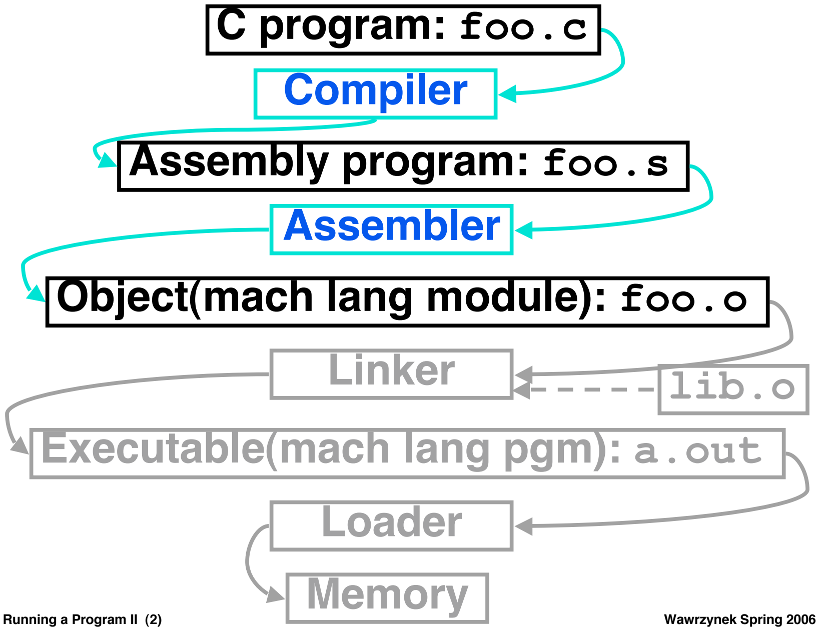
3/3/2006

John Wawrzynek

(www.cs.berkeley.edu/~johnw)

www-inst.eecs.berkeley.edu/~cs61c/

Review

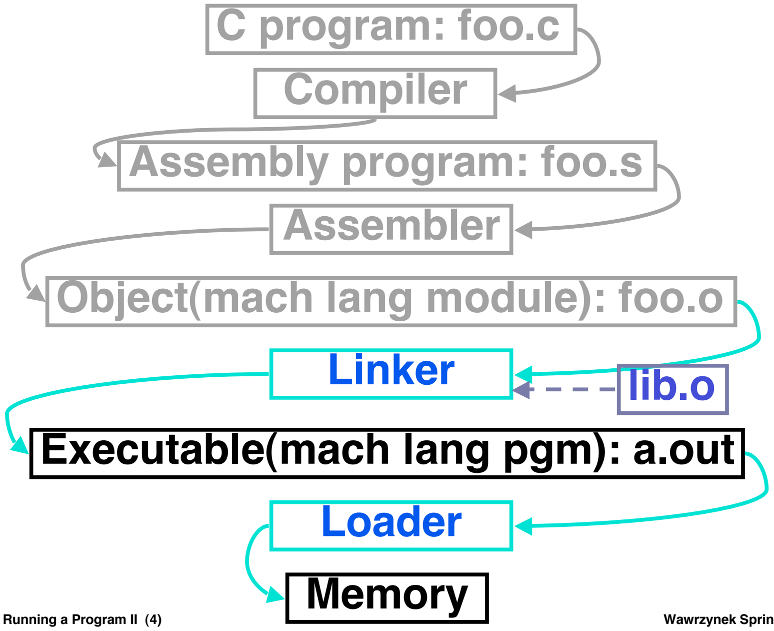


Object File Format (review)

- **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
- **debugging information**
- A standard format is ELF (except MS)

  http://www.skyfree.org/linux/references/ELF_Format.pdf

Where Are We Now?



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

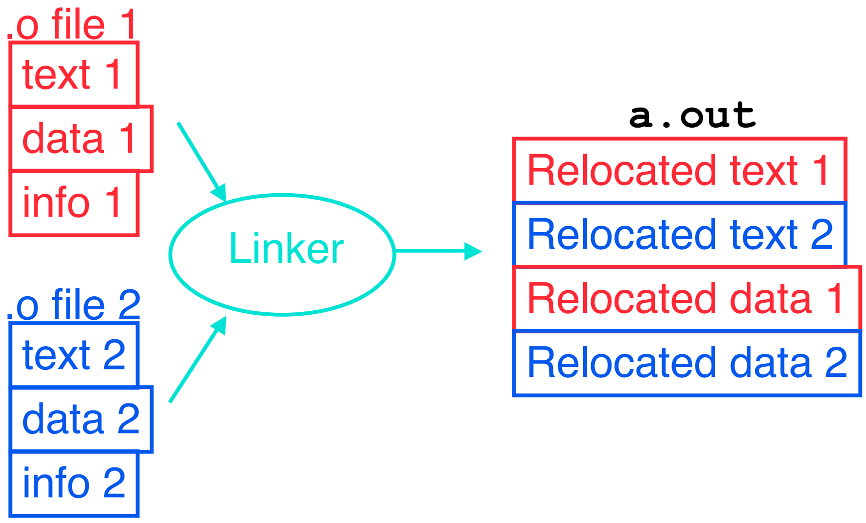
Loader

Memory

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 NT source is >40 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 and handle each entry
  - That is, fill in all absolute addresses

Four Types of Addresses we'll discuss

- 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`): always relocate

Absolute Addresses in MIPS

- Which instructions need relocation editing?
- J-format: jump, jump and link

`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.

  (More on this 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)

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 & UNIX platforms

Dynamically linked libraries

This does add quite a bit of complexity to the compiler, linker, and operating system. However, provides many benefits:

- **Space/time savings**
  - **Storing a program requires less disk space**
  - **Sending a program requires less time**
  - **Executing two programs requires less memory (if they share a library)**
- **Upgrades**
  - **By replacing one file (libXYZ.so), you upgrade every program that uses library "XYZ”**

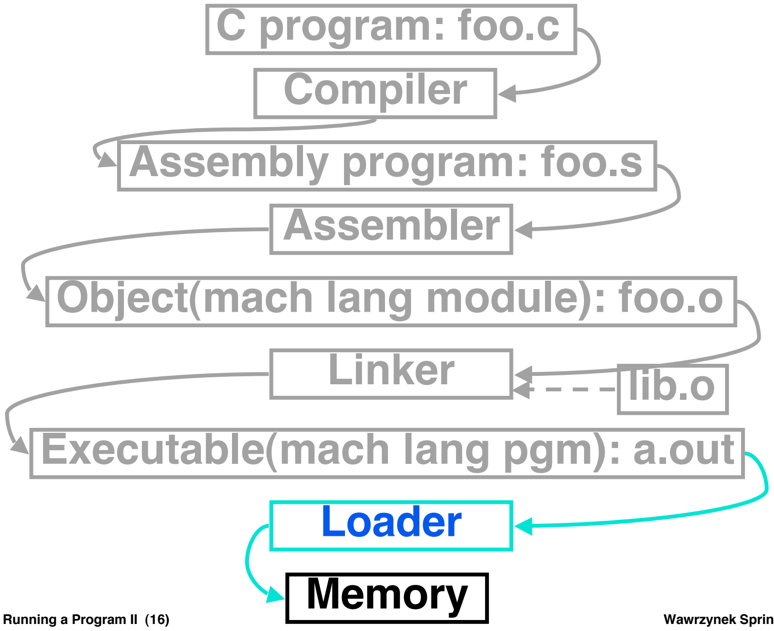
Dynamically linked libraries

- **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...**

Administrivia…

- Exam Regrade requests must be in writing.
  - Attach a written cover-sheet with your exam, explaining your concern.
  - Turn-in in class, no later than Monday.
- Remember to work on project 3: MIPS instruction interpreter.
- Impending Grade Freeze!
  - HW 1-6, Project 1&2 grades must be settled before Spring break.
  - Use glookup to verify your grades.

Where Are We Now?



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

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 it 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 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)

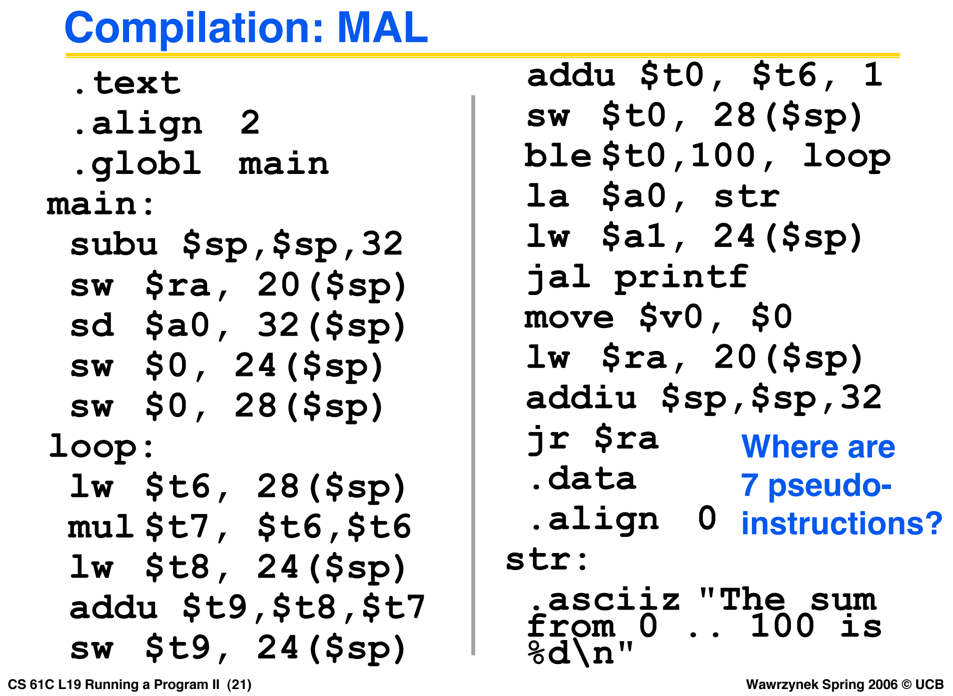
- Copies arguments passed to the program onto the stack
- Initializes machine registers
  - Most registers cleared, but stack pointer assigned address of 1st free stack location
- Jumps to start-up routine that copies program’s arguments from stack to registers and sets the PC
  - If main routine returns, 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 from 0 .. 100 is %d\n",
     sum);

}
```

“printf” lives in “libc”

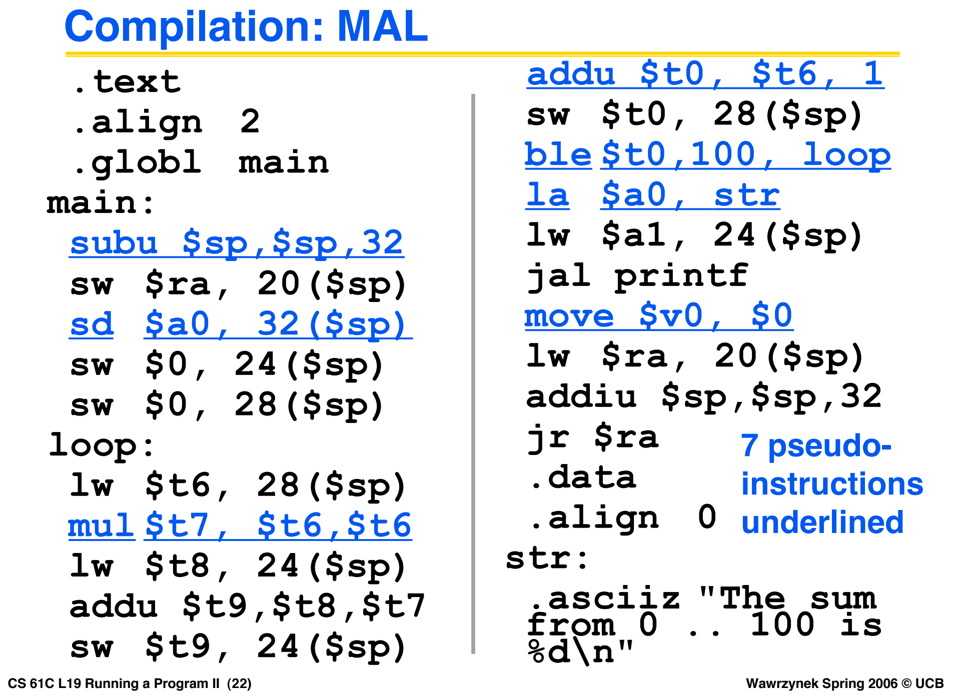


Compilation: MAL

```
 .text
 .align  2
 .globl  main
main:
 subu $sp,$sp,32
 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,32
 jr $ra
 .data
 .align  0
str:
 .asciiz "The sum from 0 .. 100 is %d\n"
```

Where are 7 pseudo-instructions?



Compilation: MAL

```
 .text
 .align  2
 .globl  main
main:
 subu $sp,$sp,32
 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,32
 jr $ra
 .data
 .align  0
str:
 .asciiz "The sum from 0 .. 100 is %d\n"
```

7 pseudo-instructions underlined

Assembly step 1:

- Remove pseudoinstructions, assign addresses

```
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, 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,$29,32
5c jr     $31
```

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
    - ext segment (instructions),
    - data segment (data),
    - symbol and relocation tables.
  - Using dummy “placeholders” for unresolved absolute and external references.

Text segment in object file

```
0x000000  00100111101111011111111111100000
0x000004  10101111101111110000000000010100
0x000008  10101111101001000000000000100000
0x00000c  10101111101001010000000000100100
0x000010  10101111101000000000000000011000
0x000014  10101111101000000000000000011100
0x000018  10001111101011100000000000011100
0x00001c  10001111101110000000000000011000
0x000020  00000001110011100000000000011001
0x000024  00100101110010000000000000000001
0x000028  00101001000000010000000001100101
0x00002c  10101111101010000000000000011100
0x000030  00000000000000000111100000010010
0x000034  00000011000011111100100000100001
0x000038  00010100001000001111111111110111
0x00003c  10101111101110010000000000011000
0x000040  00111100000001000000000000000000
0x000044  10001111101001010000000000000000
0x000048  00001100000100000000000011101100
0x00004c  00100100000000000000000000000000
0x000050  10001111101111110000000000010100
0x000054  00100111101111010000000000100000
0x000058  00000011111000000000000000001000
0x00005c  00000000000000000001000000100001
```

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

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

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

- Relocation Information

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

Link step 2:

- Edit Addresses in relocation table (*show in TAL for clarity, but done in binary.*)

```
00 addiu $29,$29,-32 | 30 addiu $8,$14, 1
04 sw    $31,20($29) | 34 sw    $8,28($29)
08 sw    $4, 32($29) | 38 slti  $1,$8, 101
0c sw    $5, 36($29) | 3c bne   $1,$0, -10
10 sw     $0, 24($29)| 40 lui   $4, 4096
14 sw     $0, 28($29)| 44 ori   $4,$4,1072
18 lw    $14, 28($29)| 48 lw     $5,24($29)
1c multu $14, $14    | 4c jal    812
20 mflo   $15        | 50 add    $2, $0, $0
24 lw    $24, 24($29)| 54 lw    $31,20($29)
28 addu $25,$24,$15  | 58 addiu $29,$29,32
2c sw    $25, 24($29)| 5c jr     $31
```

Link step 3:

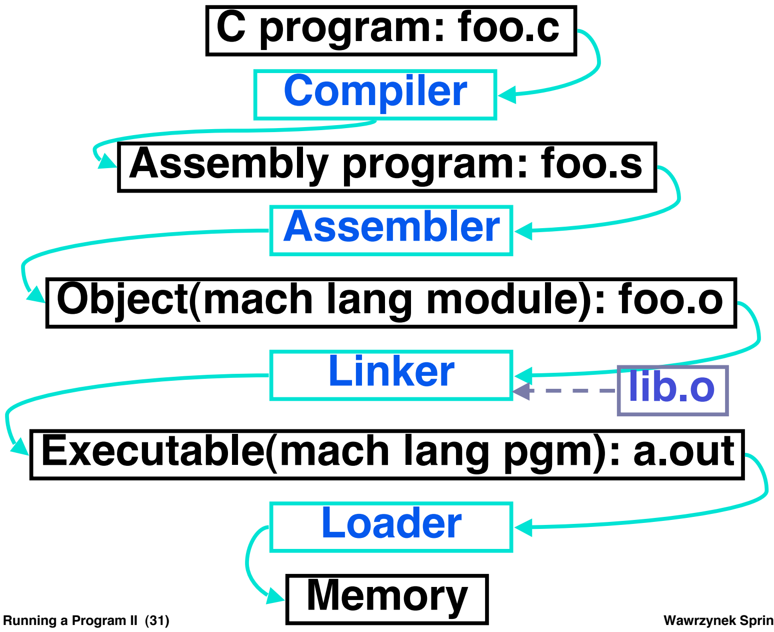
° Output executable of merged modules.

- Single text (instruction) segment
- Single data segment
- Header detailing size of each segment

° NOTE:

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

Things to Remember (1/3)



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

Things to Remember (2/3)

- Compiler converts a single HLL file into a single assembly language file.
- Assembler removes pseudoinstructions, converts what it can to machine language, and creates a checklist for the linker (relocation table). This changes each .s file into 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 3/3

- 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 ⇒ Assembler ⇒ Linker (⇒ Loader)