Assemblers, Linkers, and Loaders

Hakim Weatherspoon
CS 3410
Computer Science
Cornell University

[Weatherspoon, Bala, Bracy, and Sirer]
Big Picture: Where are we going?

C compiler

RISC-V assembly

assembler

machine code

CPU

Circuits

Gates

Transistors

Silicon

int x = 10;
x = 2 * x + 15; x0 = 0
addi x5, x0, 10 x5 = x0 + 10
muli x5, x5, 2 x5 = x5<<1 #x5 = x5 * 2
addi x5, x5, 15 x5 = x15 + 15

op = r-type               x5    shamt=1     x5      func=sll
0000000010100000000001010010011
00000000001000101000001010000000
000000000111100101000001010010011
10                              r0                  r5 op = addi
15                               r5                 r5 op = addi

op = addi

32 32

RF
Big Picture: Where are we going?

C
compiler

RISC-V
assembly
assembler

machine
code

CPU

Circuits

Gates

Transistors

Silicon

High Level Languages

```
int x = 10;
x = 2 * x + 15;
addi x5, x0, 10
muli x5, x5, 2
addi x5, x5, 15
```

Instruction Set Architecture (ISA)
RISC-y Business Office Hours
Marathon and Pizza Party!
When most people say “compile” they mean the entire process: 
compile + assemble + link
Example: sum.c

• Compiler output is assembly files

• Assembler output is obj files

• Linker joins object files into one executable

• Loader brings it into memory and starts execution
Example: sum.c

```c
#include <stdio.h>

int n = 100;
int main (int argc, char* argv[ ]) {
    int i;
    int m = n;
    int sum = 0;

    for (i = 1; i <= m; i++) {
        sum += i;
    }
    printf("Sum 1 to %d is %d\n", n, sum);
}
```
Example: sum.c

• # Compile
  [ugclinux] riscv-unknown-elf-gcc –S sum.c

• # Assemble
  [ugclinux] riscv-unknown-elf-gcc –c sum.s

• # Link
  [ugclinux] riscv-unknown-elf-gcc –o sum sum.o

• # Load
  [ugclinux] qemu-riscv32 sum
  Sum 1 to 100 is 5050
  RISC-V program exits with status 0 (approx. 2007 instructions in 143000 nsec at 14.14034 MHz)
Compiler

**Input:** Code File (.c)
- Source code
- `#includes`, function declarations & definitions, global variables, etc.

**Output:** Assembly File (RISC-V)
- RISC-V assembly instructions (.s file)

```
for (i = 1; i <= m; i++) {
    sum += i;
}
```

```assembly
li   x2,1
lw   x3,fp,28
slt  x2,x3,x2
```
```assembly
.globl n
data
.type n, @object
n: .word 100
.rdata
$str0: .string "Sum 1 to %d is %d\n"
text
.globl main
type main, @function
main: addiu $sp,$sp,-48
sw $ra,44($sp)
sw $fp,40($sp)
move $fp,$sp
sw $a0,-36($fp)
sw $a1,-40($fp)
la $a5,n
lw $a5,0($a5)
sw $a5,-28($fp)
sw $0,-24($fp)
li $a5,1
sw $a5,-20($fp)
```

```
.globl n
.globl main
.globl $L2
.globl $L3
.data
.type n, @object
n: .word 100
.rdata
$str0: .string "Sum 1 to %d is %d\n"
text
.globl main
type main, @function
main: addiu $sp,$sp,-48
sw $ra,44($sp)
sw $fp,40($sp)
move $fp,$sp
sw $a0,-36($fp)
sw $a1,-40($fp)
la $a5,n
lw $a5,0($a5)
sw $a5,-28($fp)
sw $0,-24($fp)
li $a5,1
sw $a5,-20($fp)
```

```
$L2: lw $a4,-20($fp)
lw $a5,-28($fp)
blt $a5,$a4,$L3
lw $a4,-24($fp)
lw $a5,-20($fp)
addu $a5,$a4,$a5
sw $a5,-24($fp)
lw $a5,-20($fp)
addi $a5,$a5,1
sw $a5,-20($fp)
j $L2
la $4,$str0
lw $a1,-28($fp)
lw $a2,-24($fp)
jal printf
li $a0,0
mv $sp,$fp
lw $ra,44($sp)
lw $fp,40($sp)
addiu $sp,$sp,48
ir $ra
```

```
$L3: la $4,$str0
lw $a1,-28($fp)
lw $a2,-24($fp)
jal printf
li $a0,0
mv $sp,$fp
lw $ra,44($sp)
lw $fp,40($sp)
addiu $sp,$sp,48
ir $ra
```
sum.s  (abridged)

.globl  n
.data
.type  n, @object
n:  .word  100
.rdata
$str0$:  .string  "Sum 1 to %d is %d
.text
.globl  main
.type  main, @function
main:  addiu  $sp,$sp,-48
       sw  $ra,44($sp)
       sw  $fp,40($sp)
move  $fp,$sp
sw  $a0,-36($fp)$a0
sw  $a1,-40($fp)$a1
la  $a5,n
lw  $a5,0($a5)  n=100
sw  $a5,-28($fp)m=n=100
sw  $0,-24($fp) sum=0
li  $a5,1
sw  $a5,-20($fp)i=1

prologue
$L2$:  lw  $a4,-20($fp)  i=1
       lw  $a5,-28($fp)m=100
       blt  $a5,$a4,$L3  if(m < i)
             100 < 1
       lw  $a4,-24($fp)  0(sum)
       lw  $a5,-20($fp)  1(i)
       addu  $a5,$a4,$a5  1=(0+1)
       sw  $a5,-24($fp) sum=1
       lw  $a5,-20($fp)  a5=i=1
       addi  $a5,$a5,1  i=2=(1+1)
       sw  $a5,-20($fp)  i=2
       jl  $L2

$L3$:  call  printf
       j
       $a0,$4,$str0  str
       lw  $a1,$a1,-28($fp)m=100
       lw  $a2,$a2,-24($fp) sum
jal  printf
       li  $a0,0  main returns 0
       mv  $sp,$fp
       lw  $ra,44($sp)
       lw  $fp,40($sp)
addiu  $sp,$sp,48
       $ra

epilogue
When most people say “compile” they mean the entire process: compile + assemble + link

“It’s alive!”

Executing in Memory process
Assembler

**Input:** Assembly File (.s)
- assembly instructions, pseudo-instructions
- program data (strings, variables), layout directives

**Output:** Object File in binary machine code
RISC-V instructions in executable form
(.o file in Unix, .obj in Windows)

```
addi r5, r0, 10
mul r5, r5, 2
addi r5, r5, 15
```

```
000000001010000000000000001010010011
00000000010001010000101000001010000000
000000000111100101000001010010011
```
RISC-V Assembly Instructions

Arithmetic/Logical
- ADD, SUB, AND, OR, XOR, SLT, SLTU
- ADDI, ANDI, ORI, XORI, LUI, SLL, SRL, SLTI, SLTIU
- MUL, DIV

Memory Access
- LW, LH, LB, LHU, LBU,
- SW, SH, SB

Control flow
- BEQ, BNE, BLE, BLT, BGE
- JAL, JALR

Special
- LR, SC, SCALL, SBREAK
Pseudo-Instructions

Assembly shorthand, technically not machine instructions, but easily converted into 1+ instructions that are

<table>
<thead>
<tr>
<th>Pseudo-Insns</th>
<th>Actual Insns</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOP</td>
<td>ADDI x0, x0, 0</td>
<td># do nothing</td>
</tr>
<tr>
<td>MV reg, reg</td>
<td>ADD r2, r0, r1</td>
<td># copy between regs</td>
</tr>
<tr>
<td>LI reg, 0x45678</td>
<td>LUI reg, 0x4</td>
<td># load immediate</td>
</tr>
<tr>
<td></td>
<td>ORI reg, reg, 0x5678</td>
<td></td>
</tr>
<tr>
<td>LA reg, label</td>
<td></td>
<td># load address (32 bits)</td>
</tr>
<tr>
<td>B label</td>
<td>BEQ x0, x0, label</td>
<td># unconditional branch</td>
</tr>
</tbody>
</table>

+ a few more…
Program Layout

- Programs consist of segments used for different purposes
  - Text: holds instructions
  - Data: holds statically allocated program data such as variables, strings, etc.

<table>
<thead>
<tr>
<th>data</th>
<th>text</th>
</tr>
</thead>
<tbody>
<tr>
<td>“cornell cs”</td>
<td>add x1,x2,x3</td>
</tr>
<tr>
<td>13</td>
<td>ori x2, x4, 3</td>
</tr>
<tr>
<td>25</td>
<td>...</td>
</tr>
</tbody>
</table>
Assembling Programs

- Assembly files consist of a mix of
  - + instructions
  - + pseudo-instructions
  - + assembler (data/layout) directives
    (Assembler lays out binary values in memory based on directives)

- Assembled to an Object File
  - Header
  - Text Segment
  - Data Segment
  - Relocation Information
  - Symbol Table
  - Debugging Information

```
.text
.ent main
main: la $4, Larray
   li $5, 15
...
   li $4, 0
jal exit
.end main
.data
Larray:
   .long 51, 491, 3991
```
Assembling Programs

- Assembly using a (modified) Harvard architecture
- Need segments since data and program stored together in memory

![Diagram of CPU and memory segments](image)
Takeaway

• Assembly is a low-level task
  • Need to assemble assembly language into machine code binary. Requires
    - Assembly language instructions
    - pseudo-instructions
    - And Specify layout and data using assembler directives

• Today, we use a modified Harvard Architecture (Von Neumann architecture) that mixes data and instructions in memory
  … but kept in separate segments
  … and has separate caches
Symbols and References

Global labels: Externally visible “exported” symbols
- Can be referenced from other object files
- Exported functions, global variables
- Examples: pi, e, userid, printf, pick_prime, pick_random

Local labels: Internally visible only symbols
- Only used within this object file
- static functions, static variables, loop labels, ...
- Examples: randomval, is_prime

int pi = 3;
int e = 2;
static int randomval = 7;

extern int userid;
extern int printf(char *str, ...);

int square(int x) { ... }
static int is_prime(int x) { ... }
int pick_prime() { ... }
int get_n() {
    return userid;
}

(extern == defined in another file)
Handling forward references

Example:

```
bne x1, x2, L  Looking for L
sll x0, x0, 0
L:  addi x2, x3, 0x2  Found L
```

The assembler will change this to

```
bne x1, x2, +8
sll x0, x0, 0
addi x2, x3, 0x2
```

Final machine code

```
0x00208413 # bne    actually:   0000 0000 0010...
0x00001033 # sll    0000 0000 0000...
0x00018113 # addi    0000 0000 0000...
```
Object file

- Header
  - Size and position of pieces of file

- Text Segment
  - instructions

- Data Segment
  - static data (local/global vars, strings, constants)

- Debugging Information
  - line number → code address map, etc.

- Symbol Table
  - External (exported) references
  - Unresolved (imported) references
Object File Formats

Unix

- a.out
- COFF: Common Object File Format
- ELF: Executable and Linking Format

Windows

- PE: Portable Executable

All support both executable and object files
Objdump disassembly

> riscv-unknown-elf--objdump --disassemble math.o

Disassembly of section .text:

```
00000000  <get_n>:
  0:  27bdfff8  addi   sp,sp,-8
  4:  afbe0000  sw     fp,0(sp)
  8:  03a0f021  mv     fp,sp
  1c:  3c020000  lui    a0,0x0
  10:  8c420008  lw     a0,8(a0)
  14:  03c0e821  mv     sp,fp
  18:  8fbe0000  lw     fp,0(sp)
  1c:  27bd0008  addi   sp,sp,8
  20:  03e00008  jr      ra
```

elsewhere in another file: int usrid = 41;
int get_n() {
    return usrid;
}
## Objdump symbols

```plaintext
> riscv-unknown-elf--objdump --syms math.o

<table>
<thead>
<tr>
<th>SYMBOL TABLE:</th>
<th>segment</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000000 1 df <em>ABS</em></td>
<td></td>
<td>000000000 math.c</td>
</tr>
<tr>
<td>000000000 1 d .text</td>
<td></td>
<td>000000000 .text</td>
</tr>
<tr>
<td>000000000 1 d .data</td>
<td></td>
<td>000000000 .data</td>
</tr>
<tr>
<td>000000000 1 d .bss</td>
<td></td>
<td>000000000 .bss</td>
</tr>
<tr>
<td>000000008 1 O .data</td>
<td></td>
<td>000000004 randomval</td>
</tr>
<tr>
<td>000000060 1 F .text</td>
<td></td>
<td>000000028 is_prime</td>
</tr>
<tr>
<td>000000000 1 d .rodata</td>
<td></td>
<td>000000000 .rodata static local fn</td>
</tr>
<tr>
<td>000000000 1 d .comment</td>
<td></td>
<td>000000000 .comment @ addr 0x60</td>
</tr>
<tr>
<td>000000000 g O .data</td>
<td></td>
<td>000000004 pi size = 0x28 bytes</td>
</tr>
<tr>
<td>000000004 g O .data</td>
<td></td>
<td>000000004 e</td>
</tr>
<tr>
<td>000000000 g F .text</td>
<td></td>
<td>000000028 get_n</td>
</tr>
<tr>
<td>000000028 g F .text</td>
<td></td>
<td>000000038 square</td>
</tr>
<tr>
<td>000000088 g F .text</td>
<td></td>
<td>00000004c pick_prime</td>
</tr>
<tr>
<td>000000000 <em>UND</em></td>
<td></td>
<td>000000000 usrid</td>
</tr>
<tr>
<td>000000000 <em>UND</em></td>
<td></td>
<td>000000000 printf</td>
</tr>
</tbody>
</table>

external references (undefined)
```
Separate Compilation & Assembly

Compiler

Assembler

Linker

gcc -S

sum.c -> sum.s -> sum.o

math.c -> math.s -> math.o

gcc -c

sum.o

sum

gcc -o

load

exists on disk

Executing in Memory

program

process

small change?

→ recompile one module only

http://xkcd.com/303/
Linkers

Linker combines object files into an executable file
  • Resolve as-yet-unresolved symbols
  • Each has illusion of own address space
    → Relocate each object’s text and data segments
  • Record top-level entry point in executable file

End result: a program on disk, ready to execute
  E.g. ./sum Linux
       ./sum.exe Windows
       qemu-riscv32 sum Class RISC-V simulator
Static Libraries

*Static Library*: Collection of object files
(think: like a zip archive)

Q: Every program contains the entire library?!?
A: No, Linker picks only object files needed to resolve undefined references at link time

e.g. libc.a contains many objects:

• printf.o, fprintf.o, vprintf.o, sprintf.o, snprintf.o, …
• read.o, write.o, open.o, close.o, mkdir.o, readdir.o, …
• rand.o, exit.o, sleep.o, time.o, ….
Linker Example: Resolving an External Fn Call

```
main.o

.text
00 T main
00 D usrid
*UND* printf
*UND* pi
*UND* get_n

40, JAL, printf
54, JAL, get_n

math.o

.text

24 21032040
28 000000EF
2C 1b301402
30 00000B37
34 00028293

20 T get_n
00 D pi
*UND* printf
*UND* usrid

28, JAL, printf

★ JAL printf → JAL ???
Unresolved references to printf and get_n
```
# iClicker Question 1

Which symbols are undefined according to **both** main.o and math.o’s symbol table?

A) printf  
B) pi  
C) get_n  
D) usr  
E) printf & pi

---

<table>
<thead>
<tr>
<th>Symbol Table</th>
<th>lul</th>
<th>lul</th>
</tr>
</thead>
<tbody>
<tr>
<td>main.o</td>
<td></td>
<td></td>
</tr>
<tr>
<td>text</td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>00000000EF ⭐</td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>21035000</td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>1b80050C</td>
<td></td>
</tr>
<tr>
<td>4C</td>
<td>8C040000</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>21047002</td>
<td></td>
</tr>
<tr>
<td>54</td>
<td>00000000EF ⭐</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>T main</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>D usrid</td>
<td></td>
</tr>
<tr>
<td><em>UND</em></td>
<td>printf</td>
<td></td>
</tr>
<tr>
<td><em>UND</em></td>
<td>pi</td>
<td></td>
</tr>
<tr>
<td><em>UND</em></td>
<td>get_n</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>JAL printf</td>
<td></td>
</tr>
<tr>
<td>54</td>
<td>JAL get_n</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Symbol Table</th>
<th>lul</th>
<th>lul</th>
</tr>
</thead>
<tbody>
<tr>
<td>main.o</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.text</td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>00000000EF ⭐</td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>21035000</td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>1b80050C</td>
<td></td>
</tr>
<tr>
<td>4C</td>
<td>8C040000</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>21047002</td>
<td></td>
</tr>
<tr>
<td>54</td>
<td>00000000EF ⭐</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>T main</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>D usrid</td>
<td></td>
</tr>
<tr>
<td><em>UND</em></td>
<td>printf</td>
<td></td>
</tr>
<tr>
<td><em>UND</em></td>
<td>pi</td>
<td></td>
</tr>
<tr>
<td><em>UND</em></td>
<td>get_n</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>JAL printf</td>
<td></td>
</tr>
<tr>
<td>54</td>
<td>JAL get_n</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Symbol Table</th>
<th>lul</th>
<th>lul</th>
</tr>
</thead>
<tbody>
<tr>
<td>math.o</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.text</td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>00000000EF ⭐</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>21032040</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>000000EF ⭐</td>
<td></td>
</tr>
<tr>
<td>2C</td>
<td>1b301402</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>000000B37</td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>00028293</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>T get_n</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>D pi</td>
<td></td>
</tr>
<tr>
<td><em>UND</em></td>
<td>printf</td>
<td></td>
</tr>
<tr>
<td><em>UND</em></td>
<td>usrid</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>JAL printf</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Symbol Table</th>
<th>lul</th>
<th>lul</th>
</tr>
</thead>
<tbody>
<tr>
<td>printf.o</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.text</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3C</td>
<td>T printf</td>
<td></td>
</tr>
</tbody>
</table>

⭐ JAL printf → JAL ???  
Unresolved references to printf and get_n
Linker Example: Resolving an External Function Call

main.o
- .text
  - 40 000000EF
  - 44 21035000
  - 48 1b80050c
  - 4c 8c040000
  - 50 21047002
  - 54 000000EF

Relocation info Symbol table
- 00 T main
- 00 D userid
- *UND* printf
- *UND* pi
- *UND* get_n

40,JAL, printf
- 54,JAL, get_n

math.o
- .text
  - 24 21032040
  - 28 000000ef
  - 2c 1b301402
  - 30 000000b37
  - 34 00028293

- 20 T get_n
- 00 D pi
- *UND* printf
- *UND* userid

28,JAL, printf

printf.o
- ...
iClicker Question 2

Which which 2 symbols are currently assigned the same location?

A) main & printf
B) usrid & pi
C) get_n & printf
D) main & usrid
E) main & pi

<table>
<thead>
<tr>
<th>main.o</th>
<th>text</th>
<th>math.o</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>000000EF</td>
<td>24</td>
</tr>
<tr>
<td>44</td>
<td>21035000</td>
<td>28</td>
</tr>
<tr>
<td>48</td>
<td>1b80050C</td>
<td>2C</td>
</tr>
<tr>
<td>4C</td>
<td>8C040000</td>
<td>30</td>
</tr>
<tr>
<td>50</td>
<td>21047002</td>
<td>34</td>
</tr>
<tr>
<td>54</td>
<td>000000EF</td>
<td>20</td>
</tr>
<tr>
<td>00</td>
<td>D usrid</td>
<td>00</td>
</tr>
<tr>
<td><em>UND</em></td>
<td>printf</td>
<td><em>UND</em></td>
</tr>
<tr>
<td><em>UND</em></td>
<td>pi</td>
<td><em>UND</em></td>
</tr>
<tr>
<td>40</td>
<td>JAL, printf</td>
<td>28</td>
</tr>
<tr>
<td>54</td>
<td>JAL, get_n</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>printf.o</th>
</tr>
</thead>
<tbody>
<tr>
<td>3C</td>
</tr>
</tbody>
</table>

☆ JAL printf → JAL ???
Unresolved references to printf and get_n
Linker Example: Loading a Global Variable

main.o

.text

40  000000EF
44  21035000
48  1b80050C
4C  8C040000
50  21047002
54  000000EF

00 T main
00 D userid
*UND* printf
*UND* pi
*UND* get_n
40,JAL, printf
...
54,JAL, get_n

math.o

.text

24  21032040
28  000000EF
2C  1b301402
30  000000B37★
34  00028293★

20 T get_n
00 D pi
*UND* printf
*UND* userid
28,JAL, printf
30,LUI, userid
34,LA, userid

sum.exe

.text

21032040
40023CEF
1b301402
10000B37
00428293
400020EF
10201000
21040330
22500102

.data

.pi  00000003
.userid 0077616B

Entry: 0040 0100
.text: 0040 0000
.data: 1000 0000

★ LA = LUI/ADDI "userid" → ???
Unresolved references to userid
Need address of global variable

Notice: userid gets relocated due to collision with pi
iClicker Question

#include <stdio.h>
#include heaplib.h

#define HEAP_SIZE 16
static int ARR_SIZE = 4;

int main() {
    char heap[HEAP_SIZE];
    hl_init(heap, HEAP_SIZE * sizeof(char));
    char* ptr = (char *) hl_alloc(heap, ARR_SIZE * sizeof(char));
    ptr[0] = 'h';
    ptr[1] = 'i';
    ptr[2] = '\0';
    printf(%s\n, ptr); return 0;
}

Where does the assembler place the following symbols in the object file that it creates?
A. Text Segment
B. Data Segment
C. Exported reference in symbol table
D. Imported reference in symbol table
E. None of the above

Q1: HEAP_SIZE
Q2: ARR_SIZE
Q3: hl_init
Loaders

Loader reads executable from disk into memory
  • Initializes registers, stack, arguments to first function
  • Jumps to entry-point
Part of the Operating System (OS)
Shared Libraries

Q: Every program contains parts of same library?!
A: No, they can use shared libraries
  • Executables all point to single shared library on disk
  • final linking (and relocations) done by the loader

Optimizations:
  • Library compiled at fixed non-zero address
  • Jump table in each program instead of relocations
  • Can even patch jumps on-the-fly
Static and Dynamic Linking

Static linking
- Big executable files (all/most of needed libraries inside)
- Don’t benefit from updates to library
- No load-time linking

Dynamic linking
- Small executable files (just point to shared library)
- Library update benefits all programs that use it
- Load-time cost to do final linking
  - But dll code is probably already in memory
  - And can do the linking incrementally, on-demand
Takeaway

Compiler produces assembly files
  (contain RISC-V assembly, pseudo-instructions, directives, etc.)

Assembler produces object files
  (contain RISC-V machine code, missing symbols, some layout information, etc.)

Linker joins object files into one executable file
  (contains RISC-V machine code, no missing symbols, some layout information)

Loader puts program into memory, jumps to 1st insn, and starts executing a process
  (machine code)