Assemblers, Linkers, and Loaders

Hakim Weatherspoon
CS 3410
Computer Science
Cornell University

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

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

```
x0 = 0
x5 = x0 + 10
x5 = x5<<1 #x5 = x5 * 2
x5 = x15 + 15
```

```
00000000101000000000101010010011
00000000001000101000001010000000
00000000111100101000001010010011
```

```
10                r0
r5               op = addi
```

```
15               r5
```

```
x5  shamt=1     x5  func=sll
```

C

compiler

RISC-V

assembly

assembler

machine code

CPU

Circuits

Gates

Transistors

Silicon

A

B

32 32

RF

32
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;

addi x5, x0, 10
muli x5, x5, 2
addi x5, x5, 15

000000000101000000000001010010011
00000000001000101000001010000000
00000000111100101000001010010011

High Level Languages

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.

“’It’s alive!’”
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)

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

```asm
li  x2,1
lw  x3,fp,28
slt x2,x3,x2
```
From Writing to Running

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

“\textit{It’s alive!}”

Executing in Memory process

e Xecutable program
exists on disk
loader
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
muli r5, r5, 2
addi r5, r5, 15
```

```
0000000010100000000000010100010011
00000000010001010000101000000000
00000000111100100001000010100100
```
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>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.

```
add x1, x2, x3
ori x2, x4, 3
...```

```
“cornell cs”
13
25
```

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
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

```c
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;
}

(math.c)
(extern == defined in another file)```
Handling forward references

Example:

\[\text{bne } x1, x2, L \quad \text{Looking for } L\]
\[\text{sll } x0, x0, 0\]
\[L: \quad \text{addi } x2, x3, 0x2 \quad \text{Found } L\]

The assembler will change this to

\[\text{bne } x1, x2, +8\]
\[\text{sll } x0, x0, 0\]
\[\text{addi } x2, x3, 0x2\]

Final machine code

0x0000 0000 0000 0000 0010... # bne
0x0000 0000 0000 0000 0000... # sll
0x0000 0000 0000 0000 0000... # addi

Looking for L
Found L
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 \(\rightarrow\) 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
c:  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:

```c
int usrid = 41;
int get_n() {
    return usrid;
}
```
### Objdump symbols

```shell
> 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</td>
<td>l df</td>
<td><em>ABS</em> 00000000 math.c</td>
</tr>
<tr>
<td>000000000</td>
<td>l d</td>
<td>.text 00000000 .text</td>
</tr>
<tr>
<td>000000000</td>
<td>l d</td>
<td>.data 00000000 .data</td>
</tr>
<tr>
<td>000000000</td>
<td>l d</td>
<td>.bss 00000000 .bss</td>
</tr>
<tr>
<td>000000008</td>
<td>l O</td>
<td>.data 00000004 randomval</td>
</tr>
<tr>
<td>000000060</td>
<td>l F</td>
<td>.text 00000028 is_prime</td>
</tr>
<tr>
<td>000000000</td>
<td>l d</td>
<td>.rodata 00000000 .rodata</td>
</tr>
<tr>
<td>000000000</td>
<td>l d</td>
<td>.comment 00000000 .comment</td>
</tr>
<tr>
<td>000000000</td>
<td>g O</td>
<td>.data 00000004 pi</td>
</tr>
<tr>
<td>00000004</td>
<td>g O</td>
<td>.data 00000004 e</td>
</tr>
<tr>
<td>000000000</td>
<td>g F</td>
<td>.text 00000028 get_n</td>
</tr>
<tr>
<td>00000028</td>
<td>g F</td>
<td>.text 00000038 square</td>
</tr>
<tr>
<td>000000088</td>
<td>g F</td>
<td>.text 0000004c pick_prime</td>
</tr>
<tr>
<td>000000000</td>
<td><em>UND</em></td>
<td>00000000 usrid</td>
</tr>
<tr>
<td>000000000</td>
<td><em>UND</em></td>
<td>00000000 printf</td>
</tr>
</tbody>
</table>

**Function**
- static local fn
- @ addr 0x60
- size = 0x28 bytes
Separate Compilation & Assembly

Compiler

Assembler

Linker

executable program

exists on disk

loader

source

assembly files

obj files

small change?

→ recompile one module only

THE #1 PROGRAMMER EXCUSE FOR LEGITIMATELY SLACKING OFF:
“MY CODE’S COMPILING.”

HEY! GET BACK TO WORK!

COMPILING!

OH. CARRY ON.

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

Unresolved references to printf and get_n
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
Linker Example: Resolving an External Fn Call

main.o

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

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

math.o

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

20 T get_n
00 D pi
*UND* printf
*UND* usrid
28,JAL, printf

printf.o

... 3C T printf

sum.exe

0040 0000 21032040
40023CEF
1b301402
3C041000
34040004
...

0040 0100 40023CEF
21035000
1b80050C
8C040000
...

0040 0200 40023CEF
21035000
1b80050C
8C040000
...

10201000 1000 0000
21040330 22500102
...
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
Linker Example: Loading a Global Variable

main.o

... 000000EF
40 21035000
44 1b80050C
48 8c040000
50 21047002
54 000000EF
...

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

math.o

... 21032040
28 000000EF
2c 1b301402
30 00000BB37
34 00028293
...

20 T get_n
28 000000EF
*UND* printf
*UND* usrid
28,JAL, printf
30,LUI, usrid
34,LA, usrid

sum.exe

... 21032040
40023CEF
1b301402
10000BB37
00428293
...

main

math

printf

math.o

main.o

sum.exe

Relocation info

Unresolved references to userid
Need address of global variable

LA = LUI/ADDI "usrid" → ???

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

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

```c
#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;
}
```

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 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)