Assemblers, Linkers, and Loaders

CS 3410
Computer System Organization & Programming

[K. Bala, A. Bracy, E. Sirer, and H. Weatherspoon]
Big Picture: Where are we going?

C
compiler

MIPS
assembly
assembler

machine
code
loader

CPU

Circuits

Gates

Transistors

Silicon

int x = 10;
x = x + 15;

addi r5, r0, 10
addi r5, r5, 15

r0 = 0
r5 = r0 + 10
r5 = r15 + 15

addi r0 r5 10

00100000000001010000000000001010
00100000101001010000000000001111

32 32

RF

A

B

Silicon
Big Picture: Where are we going?

```
int x = 10;
x = 2 * x + 15;
addi r5, r0, 10
muli r5, r5, 2
addi r5, r5, 15
```

High Level Languages

Instruction Set Architecture (ISA)
When most people say “compile” they mean the entire process: *compile* + *assemble* + *link*
sum.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);
}

Compiler

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

Output: Assembly File (MIPS)
• MIPS assembly instructions ( .s file )

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

```assembly
li  $2,1
lw  $3,28($fp)
slt $2,$3,$2
```
```assembly
data
.type n, @object
n: .word 100
.rdata
$str0$: .ascii "Sum 1 to %d is %d\n"
.text
.globl main
.type main, @function
main:  addiu $sp,$sp,-48
       sw $31,44($sp)
       sw $fp,40($sp)
       move $fp,$sp
       sw $4,48($fp)
       sw $5,52($fp)
       la $2,n
       lw $2,0($2)
       sw $2,28($fp)
       sw $0,32($fp)
       li $2,1
       sw $2,24($fp)

L2:    lw $2,24($fp)
       lw $3,28($fp)
       slt $2,$3,$2
       bne $2,$0,$L3
       lw $3,32($fp)
       lw $2,24($fp)
       addu $2,$3,$2
       sw $2,32($fp)
       lw $2,24($fp)
       addu $2,$3,$2
       sw $2,24($fp)
       lw $2,24($fp)
       addiu $2,$2,1
       sw $2,24($fp)
       addiu $2,$2,1
       sw $2,24($fp)
       lw $5,28($fp)
       lw $6,32($fp)
       jal printf
       move $sp,$fp
       lw $31,44($sp)
       lw $fp,40($sp)
       addiu $sp,$sp,48
       j $31
```
sum.s (abridged)

.globl n
.data
.type n, @object
n: .word 100
.rdata
$str0$: .ascii "Sum 1 to %d is %d\n"
.text
.globl main
.type main, @function
main:
  addiu $sp,$sp,-48
  sw $31,44($sp)
  sw $fp,40($sp)
  move $fp,$sp
  sw $4,48($fp)
  sw $5,52($fp)
  la $2,n
  lw $2,0($2)
  sw $2,28($fp)
  sw $0,32($fp)
  li $2,1
  sw $2,24($fp)
  $L2:
    lw $2,24($fp)  i=1
    lw $3,28($fp)  m=100
    slt $2,$3,$2  if(m < i)
    bne $2,$0,$L3  100 < 1
    lw $3,32($fp)  v1=0(sum)
    lw $2,24($fp)  v0=1(0+1)
    addu $2,$3,$2
    sw $2,32($fp)  sum=1
    lw $2,24($fp)  i=1
    addiu $2,$2,1  i=2 (1+1)
    sw $2,24($fp)
    b $L2
    $L3:
    la $2,n
    lw $2,0($2)
    sw $2,28($fp)
    sw $0,32($fp)
    li $2,1
    sw $2,24($fp)
    $a0$4,$str0  str
    call printf
    lw $5,28($fp)  m=100
    $a1$6,32($fp)sum
    jal printf
    move $sp,$fp
    lw $31,44($sp)
    lw $fp,40($sp)
    addiu $sp,$sp,48
    j $31

epilogue
Assembler

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

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

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

```
001000000000010100000000000001010
0000000000000101001001010000100000
00100001010010100000000000000111
```
MIPS Assembly Instructions

Arithmetic/Logical

• ADD, ADDU, SUB, SUBU, AND, OR, XOR, NOR, SLT, SLTU
• ADDI, ADDIU, ANDI, ORI, XORI, LUI, SLL, SRL, SLLV, SRLV, SRAV, SLTI, SLTIU
• MULT, DIV, MFLO, MTLO, MFHI, MTHI

Memory Access

• LW, LH, LB, LHU, LBU, LWL, LWR
• SW, SH, SB, SWL, SWR

Control flow

• BEQ, BNE, BLEZ, BLTZ, BGEZ, BGTZ
• J, JR, JAL, JALR, BEQL, BNEL, BLEZL, BGTZL

Special

• LL, SC, SYSCALL, BREAK, SYNC, COPROC
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>SLL r0, r0, 0</td>
<td># do nothing</td>
</tr>
<tr>
<td>MOVE 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>BLT reg, reg, label</td>
<td>SLT r1, rA, rB</td>
<td># branch less than</td>
</tr>
<tr>
<td></td>
<td>BNE r1, r0, label</td>
<td></td>
</tr>
</tbody>
</table>

+ a few more…
Symbols and References

math.c

```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;
}
```

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
Handling forward references

Example:

\[
\begin{align*}
\text{bne } & $1, $2, \text{ L} & \text{Looking for L} \\
\text{sll } & $0, $0, 0 \\
\text{L: addiu } & $2, $3, 0x2 & \text{Found L}
\end{align*}
\]

The assembler will change this to

\[
\begin{align*}
\text{bne } & $1, $2, +1 \\
\text{sll } & $0, $0, 0 \\
\text{addiu } & $2, $3, $0x2
\end{align*}
\]

Final machine code

\[
\begin{align*}
0x14220001 & \# \text{ bne} & \text{ actually: } & 000101... \\
0x00000000 & \# \text{ sll} & \text{ actually: } & 000000... \\
0x24620002 & \# \text{ addiu} & \text{ actually: } & 001001...
\end{align*}
\]
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

```
> objdump --disassemble math.o

Disassembly of section .text:

00000000 <get_n>:
  0: 27bdff8  addiu sp,sp,-8
  4: afbe0000  sw  s8,0(sp)
  8: 03a0f021  move s8,sp
 c: 3c020000  lui  v0,0x0
10: 8c420008  lw  v0,8(v0)
14: 03c0e821  move sp,s8
18: 8fbe0000  lw  s8,0(sp)
1c: 27bd0008  addiu sp,sp,8
20: 03e00008  jr   ra
24: 00000000  nop
```

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

```bash
> mipsel-linux-objdump --syms math.o
```

<table>
<thead>
<tr>
<th>SYMBOL TABLE:</th>
<th>segment</th>
<th>size</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000000</td>
<td>l</td>
<td><em>ABS</em></td>
<td>000000000 math.c</td>
</tr>
<tr>
<td>000000000</td>
<td>d</td>
<td>.text</td>
<td>000000000 .text</td>
</tr>
<tr>
<td>000000000</td>
<td>d</td>
<td>.data</td>
<td>000000000 .data</td>
</tr>
<tr>
<td>000000000</td>
<td>d</td>
<td>.bss</td>
<td>000000000 .bss</td>
</tr>
<tr>
<td>0000000000</td>
<td>O</td>
<td>.data</td>
<td>000000004 randomval</td>
</tr>
<tr>
<td>000000060</td>
<td>F</td>
<td>.text</td>
<td>000000028 is_prime</td>
</tr>
<tr>
<td>000000000</td>
<td>d</td>
<td>.rodata</td>
<td>000000000 .rodata</td>
</tr>
<tr>
<td>000000000</td>
<td>d</td>
<td>.comment</td>
<td>000000000 .comment</td>
</tr>
<tr>
<td>000000000</td>
<td>g O</td>
<td>.data</td>
<td>000000004 pi</td>
</tr>
<tr>
<td>000000004</td>
<td>g O</td>
<td>.data</td>
<td>000000004 e</td>
</tr>
<tr>
<td>000000000</td>
<td>g F</td>
<td>.text</td>
<td>000000028 get_n</td>
</tr>
<tr>
<td>000000028</td>
<td>g F</td>
<td>.text</td>
<td>000000038 square</td>
</tr>
<tr>
<td>000000088</td>
<td>g F</td>
<td>.text</td>
<td>00000004c pick_prime</td>
</tr>
<tr>
<td>000000000</td>
<td><em>UND</em></td>
<td></td>
<td>000000000 usrid</td>
</tr>
<tr>
<td>000000000</td>
<td><em>UND</em></td>
<td></td>
<td>000000000 printf</td>
</tr>
</tbody>
</table>

---

*external references (undefined)*
Separate Compilation & Assembly

Compiler  Assembler  Linker

sum.c → sum.s → sum.o
math.c → math.s → math.o

source  assembly files  obj files

Executable program
exists on disk

loader

Executing in Memory

small change? → recompile one module only

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

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
     ./sum.exe
   simulate sum  Class MIPS simulator

Linux
Windows
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

40 0C000000
44 21035000
48 1b80050C
4C 8C040000
50 21047002
54 0C000000

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 0C000000
2C 1b301402
30 3C040000
34 34040000

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

printf.o

.text

54,JAL, get_n

sum.exe

.text

0040 0000
21032040
0C40023C
1b301402
3C041000
34040004

0040 0100
0C40023C
21035000
1b80050C
8C048004
21047002
0C400020

0040 0200
10201000
21040330
22500102

.data

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

global variables go here (later)

★ JAL printf → JAL ???
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`
### 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

---

**main.o**

<table>
<thead>
<tr>
<th>40</th>
<th>0C000000</th>
</tr>
</thead>
<tbody>
<tr>
<td>44</td>
<td>21035000</td>
</tr>
<tr>
<td>48</td>
<td>1b80050C</td>
</tr>
<tr>
<td>4C</td>
<td>8C040000</td>
</tr>
<tr>
<td>50</td>
<td>21047002</td>
</tr>
<tr>
<td>54</td>
<td>0C000000</td>
</tr>
</tbody>
</table>

Symbol table:

<table>
<thead>
<tr>
<th>00</th>
<th>T</th>
<th>main</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>D</td>
<td>usrid</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>
</tbody>
</table>

Relocation info:

<table>
<thead>
<tr>
<th>40</th>
<th>JAL, printf</th>
</tr>
</thead>
<tbody>
<tr>
<td>54</td>
<td>JAL, get_n</td>
</tr>
</tbody>
</table>

**math.o**

<table>
<thead>
<tr>
<th>24</th>
<th>21032040</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>0C000000</td>
</tr>
<tr>
<td>2C</td>
<td>1b301402</td>
</tr>
<tr>
<td>30</td>
<td>3C040000</td>
</tr>
<tr>
<td>34</td>
<td>34040000</td>
</tr>
</tbody>
</table>

Symbol table:

<table>
<thead>
<tr>
<th>20</th>
<th>T</th>
<th>get_n</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>D</td>
<td>pi</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>
</tbody>
</table>

Relocation info:

| 28 | JAL, printf |

**printf.o**

| 3C | T | printf |

---

**Unresolved references to printf and get_n**

- JAL printf → JAL ???
- JAL ???
Linker Example: Loading a Global Variable

main.o

... 0C000000
44 21035000
48 1b80050C
4C 8C040000
50 21047002
54 0C000000
...

.relocate info
Symbol table

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

math.o

... 21032040
28 0C000000
2C 1b301402
30 3C040000
34 34040000
...

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

... 21032040
0C40023C
1b301402
3C041000
34040004
...

... 0C40023C
21035000
1b80050c
8C048004
21047002
0C400020
...

sum.exe

... 21032040
0C40023C
1b301402
3C041000
34040004
...

math

... 0C40023C
21035000
1b80050c
8C048004
21047002
0C400020
...

main

... 10201000
21040330
22500102
...

printf

... 10201000
21040330
22500102
...

.data

pi 00000003
usrid 0077616B

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

LA = LUI/ORI "usrid" → ???
Unresolved references to usrid
Need address of global variable

Notice: usrid 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

#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
C source files

Assembler

sum.c

sum.s

math.c

math.s

io.s

io.o

Assembly files

obj files

Compiler

math.s

math.o

sum.s

sum.o

sum.exe

Libc.obj

Libm.obj

Linker

Executable program

Executing in Memory process

exists on disk

loader
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 MIPS assembly, pseudo-instructions, directives, etc.)

**Assembler** produces object files
(contain MIPS machine code, missing symbols, some layout information, etc.)

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

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