Assemblers, Linkers, and Loaders

Anne Bracy
CS 3410
Computer Science
Cornell University

The slides are the product of many rounds of teaching CS 3410 by Professors Weatherspoon, Bala, Bracy, McKee, and Sirer.

See: P&H Appendix A1-2, A.3-4 and 2.12
When most people say “compile” they mean the entire process: compile + assemble + link
```
#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);
}

csug03> mipsel-linux-gcc -S sum.c
export PATH=${PATH}:/courses/cs3410/mipsel-linux/bin:/courses/cs3410/mips-sim/bin
or
setenv PATH ${PATH}:/courses/cs3410/mipsel-linux/bin:/courses/cs3410/mips-sim/bin
```
From Writing to Running: Command Line

# Compile
csug01> mipsel-linux-gcc -S sum.c

# Assemble
csug01> mipsel-linux-gcc -c sum.s

# Link
csug01> mipsel-linux-gcc -o sum sum.o ${LINKFLAGS} 
# -nostartfiles -nodefaultlibs
# -static -mno-xgot -mno-embedded-pic
- mno-abicalls -G 0 -DMIPS -Wall

# Load
csug01> simulate sum
Sum 1 to 100 is 5050
MIPS program exits with status 0 (approx. 2007 instructions in 143000 nsec at 14.14034 MHz)
.data
.globl n
.align 2
.n: .word 100
.rdata
.align 2
$str0: .asciiz
   "Sum 1 to %d is %d\n"
.text
.align 2
.globl main
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)
addiu $2,$2,1
sw $2,24($fp)
b $L2

$L3: la $4,$str0
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
.data
.globl n
.align 2
.n: .word 100
.rdata
.align 2
$str0$: .asciiz
   "Sum 1 to %d is %d\n"
.text
.align 2
.globl main
main: 
   addiu $sp,$sp,-48
   sw $31,44($sp)
   sw $fp,40($sp)
   move $fp,$sp
   sw $a0$4,48($fp)
   sw $a1$5,52($fp)
   la $v0$2,n
   lw $2,0($2) $v0=100
   sw $2,28($fp) m=100
   sw $0,32($fp) sum=0
   li $2,1
   sw $2,24($fp) i=1
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
000000000000010100101010000100000
0010000001010010100000000000001111
```
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

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

math.c

```c
int pi = 3;
int e = 2;
static int randomval = 7;
    (external == defined in another file)
extern int usrid;
extern int printf(char *str, ...);

int square(int x) { ... }
static int is_prime(int x) { ... }
int pick_prime() { ... }
int pick_random() {
    return usrid;
}
```
Handling forward references

Example:

\[
\text{bne } $1, $2, L
\]
\[
\text{sll } $0, $0, 0
\]
\[
\text{L: addiu } $2, $3, 0x2
\]

Looking for L

Found L

The assembler will change this to

\[
\text{bne } $1, $2, +1
\]
\[
\text{sll } $0, $0, 0
\]
\[
\text{addiu } $2, $3, $0x2
\]

Final machine code

<table>
<thead>
<tr>
<th>Machine Code (Hex)</th>
<th>Machine Code (Binary)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0X14220001 # bne</td>
<td>000101000011000100000000000001</td>
</tr>
<tr>
<td>0x00000000 # sll</td>
<td>00000000000000000000000000000000000000000000</td>
</tr>
<tr>
<td>0x24620002 # addiu</td>
<td>001001000110001000000000000000010</td>
</tr>
</tbody>
</table>
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**

csug01> mipsel-linux-objdump --disassemble math.o
math.o:    file format elf32-tradlittlemips
Disassembly of section .text:

00000000 <pick_random>:
  0: 27bdfff8       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

static int usr = 41;

int pick_random() { return usr; }
Objdump symbols

csug01 ~> mipsel-linux-objdump --syms math.o

math.o: file format elf32-tradlittlemips

```
SYMBOL TABLE:

<table>
<thead>
<tr>
<th>Address</th>
<th>Type</th>
<th>Name</th>
<th>Address</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>l</td>
<td>df <em>ABS</em></td>
<td>00000000</td>
<td>math.c</td>
</tr>
<tr>
<td>00000000</td>
<td>l</td>
<td>d .text</td>
<td>00000000</td>
<td>.text</td>
</tr>
<tr>
<td>00000000</td>
<td>l</td>
<td>d .data</td>
<td>00000000</td>
<td>.data</td>
</tr>
<tr>
<td>00000000</td>
<td>l</td>
<td>d .bss</td>
<td>00000000</td>
<td>.bss</td>
</tr>
<tr>
<td>00000000</td>
<td>l</td>
<td>d .mdebug.abi32</td>
<td>00000000</td>
<td>.mdebug.abi32</td>
</tr>
<tr>
<td>00000008</td>
<td>l</td>
<td>O .data</td>
<td>00000004</td>
<td>randomval</td>
</tr>
<tr>
<td>00000060</td>
<td>l</td>
<td>F .text</td>
<td>00000028</td>
<td>is_prime</td>
</tr>
<tr>
<td>00000000</td>
<td>l</td>
<td>d .rodata</td>
<td>00000000</td>
<td>.rodata</td>
</tr>
<tr>
<td>00000000</td>
<td>l</td>
<td>d .comment</td>
<td>00000000</td>
<td>.comment</td>
</tr>
<tr>
<td>00000000</td>
<td>g</td>
<td>O .data</td>
<td>00000004</td>
<td>pi</td>
</tr>
<tr>
<td>00000004</td>
<td>g</td>
<td>O .data</td>
<td>00000004</td>
<td>e</td>
</tr>
<tr>
<td>00000000</td>
<td>g</td>
<td>F .text</td>
<td>00000028</td>
<td>pick_random</td>
</tr>
<tr>
<td>00000028</td>
<td>g</td>
<td>F .text</td>
<td>00000038</td>
<td>square</td>
</tr>
<tr>
<td>00000088</td>
<td>g</td>
<td>F .text</td>
<td>0000004c</td>
<td>pick_prime</td>
</tr>
<tr>
<td>00000000</td>
<td><em>UND</em></td>
<td></td>
<td>00000000</td>
<td>usr</td>
</tr>
<tr>
<td>00000000</td>
<td><em>UND</em></td>
<td></td>
<td>00000000</td>
<td>printf</td>
</tr>
</tbody>
</table>
```

external references (undefined)
Separate Compilation & Assembly

Compiler

source files

sum.c

math.c

Assembler

assembly files

sum.s

math.s

Linker

obj files

sum.o

math.o

executable program

sum

exists on disk

loader

Executing in Memory process

small change?

→ recompile one module only

THE #1 PROGRAMMER EXCUSE FOR LEGITIMATELY SLACKING OFF:
"MY CODE'S COMPILED."

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
      ./sum.exe
      simulate sum

  Linux
  Windows
  Class MIPS 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

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

... 00 T main
00 D usr
*UND* printf
*UND* pi
*UND* square
40,JAL, printf
...
54,JAL, square

... 24 21032040
28 0C000000
2C 1b301402
30 3C040000
34 34040000

... 20 T square
00 D pi
*UND* printf
*UND* usr

28,JAL, printf

... 3C T printf

math.o

Relocation info

.Symbol table

JAL printf

Unresolved references to printf and square

printf.o

.sum.exe

.text

21032040
0C40023C
1b301402
3C041000
34040004

.JAL printf

.math

... 0C40023C
21035000
1b80050C
8C048004
21047002
0C400020

... 10201000
21040330
22500102

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

global variables

go here (later)
Which symbols are undefined according to both main.o and math.o’s symbol table?

A) printf
B) pi
C) square
D) usr
E) printf & pi
Which which 2 symbols are currently assigned the same location?

A) main & printf
B) usr & pi
C) square & printf
D) main & usr
E) main & pi

<table>
<thead>
<tr>
<th>main.o</th>
<th>math.o</th>
</tr>
</thead>
<tbody>
<tr>
<td>.text</td>
<td>.text</td>
</tr>
<tr>
<td>40</td>
<td>24</td>
</tr>
<tr>
<td>0C000000</td>
<td>21032040</td>
</tr>
<tr>
<td>44</td>
<td>28</td>
</tr>
<tr>
<td>21035000</td>
<td>0C000000</td>
</tr>
<tr>
<td>48</td>
<td>2C</td>
</tr>
<tr>
<td>1b80050C</td>
<td>1b301402</td>
</tr>
<tr>
<td>4C</td>
<td>30</td>
</tr>
<tr>
<td>8C040000</td>
<td>3C040000</td>
</tr>
<tr>
<td>50</td>
<td>34</td>
</tr>
<tr>
<td>21047002</td>
<td>34040000</td>
</tr>
<tr>
<td>54</td>
<td></td>
</tr>
<tr>
<td>0C000000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>20</td>
</tr>
<tr>
<td>T</td>
<td>square</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>D</td>
<td>pi</td>
</tr>
<tr>
<td><em>UND</em></td>
<td><em>UND</em></td>
</tr>
<tr>
<td>printf</td>
<td>printf</td>
</tr>
<tr>
<td><em>UND</em></td>
<td><em>UND</em></td>
</tr>
<tr>
<td>pi</td>
<td>usr</td>
</tr>
<tr>
<td><em>UND</em></td>
<td></td>
</tr>
<tr>
<td>square</td>
<td></td>
</tr>
<tr>
<td>40,JAL, printf</td>
<td>28,JAL, printf</td>
</tr>
<tr>
<td>4C,LW/gp, pi</td>
<td>30,LUI, usr</td>
</tr>
<tr>
<td>54,JAL, square</td>
<td>34,LA, usr</td>
</tr>
</tbody>
</table>
## Linker Example: Loading a Global Variable

**main.o**

| 40  | 0C000000 |
| 44  | 21035000 |
| 48  | 1b80050c |
| 4C  | 8C040000 | ① |
| 50  | 21047002 |
| 54  | 0C000000 |

**math.o**

| 24  | 21032040 |
| 28  | 0C000000 |
| 2C  | 1b301402 |
| 30  | 3C040000 |
| 34  | 34040000 |

| 20  | T  | square |
| 00  | D  | pi     |
| *UND* |  | printf |
| *UND* |  | pi |
| *UND* |  | square |
| 40  | JAL, | printf |
| 4C  | LW/gp, | pi |
| 54  | JAL, | square |

**sum.exe**

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

**Relocation info**

`28, JAL, printf`  
`30, LUI, usr`  
`34, LA, usr`

- **LW $4 “pi” → LW $4 ???**  
  Unresolved reference to pi

- **LW $4, -32764($gp) (A-32768)**
### Linker Ex: Resolving Addr of Global Variable

#### main.o

<table>
<thead>
<tr>
<th>Entry</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>0C000000</td>
</tr>
<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>

#### .text

- **T** main
- **D** usr
- *UND* printf
- *UND* pi
- *UND* square

### math.o

<table>
<thead>
<tr>
<th>Entry</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>21032040</td>
</tr>
<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>

#### .text

- **T** square
- **D** pi
- *UND* printf
- *UND* usr

#### Relocation info

- 28, JAL, printf
- 30, LUI, usr

#### sum.exe

<table>
<thead>
<tr>
<th>Entry</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>21032040</td>
<td>0C40023C</td>
</tr>
<tr>
<td>1b301402</td>
<td>3C041000</td>
</tr>
<tr>
<td>34040004</td>
<td>0C400020</td>
</tr>
</tbody>
</table>

#### LA num:

- LUI 1000
- ORI 0004

- **LA** = LUI/ORI "usr" → ???

Unresolved reference to usr

Notice: usr gets relocated due to collision with pi

- **text**: 0040 0000
- **data**: 1000 0000
#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;
}
Compiler

sum.c
math.c
C source files

Assembler

sum.s
math.s
io.s
assembly files

Linker

sum.o
math.o
io.o
libc.o
libm.o
obj files

Executable program

sum.exe

exists on disk

loader

Executing in Memory process
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)