Assemblers, Linkers, and Loaders

CS 3410
Computer System Organization & Programming

These slides are the product of many rounds of teaching CS 3410 by Professors Weatherspoon, Bala, Bracy, and Sirer.
Big Picture: Where are we going?

\[
\text{int } x = 10; \\
x = x + 15;
\]

\[
\text{addi } r5, r0, 10 \\
\text{addi } r5, r5, 15
\]

\[
\begin{array}{cccc}
\text{addi} & r0 & r5 & 10 \\
00100000000001010 & 10000000000000001010 \\
00100001010010101 & 10000000000000001111 \\
\end{array}
\]
Big Picture: Where are we going?

C
    compiler

MIPS
    assembly
    assembler

machine
    code
    loader

CPU

Circuits

Gates

Transistors

Silicon

int x = 10;
x = 2 * x + 15;

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

0010000000000101000000000000010100
000000000000010100100001000001000000
00100000101001010000000000001111

High Level Languages

Instruction Set Architecture (ISA)
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

#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] mipsel-linux-gcc –S sum.c

- # Assemble
  - [ugclinux] mipsel-linux-gcc –c sum.s

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

- # Load
  - [ugclinux] simulate sum
  - Sum 1 to 100 is 5050
  - MIPS 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 (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
```
sum.s  (abridged)  $L2$:  

.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)  
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
sum.s  (abridged)  $L2:  

.globl n  
.data  
.type n, @object  
n: .word 100  
rdata  
$str0: .ascii "Sum 1 to %d is %d
"  
.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) $a0  
  sw $5,52($fp) $a1  
  la $2,n $v0  
  lw $2,0($2) $v0=100  
  lw $2,28($fp) m=100  
  sw $0,32($fp) sum=0  
  li $2,1  
  sw $2,24($fp) i=1  
$L3:  
  call printf  
  jal printf  
  move $sp,$fp  
  lw $31,44($sp)  
  lw $fp,40($sp)  
  addiu $sp,$sp,48  
  j $31  

prologue  

$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 v0=1(0+1)  
  sw $2,32($fp) sum=1  
  lw $2,24($fp) i=1  
  addiu $2,$2,1 i=2 (1+1)  
  sw $2,24($fp) i=2  

epilogue  

$str0: .ascii "Sum 1 to %d is %d
"  
$va0  
$va1  
$va2  

10
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
```

```
00100000000001010000000000001010
0000000000000101001001010000010000000000001111
```

11
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, COPROCE
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>LA reg, label</td>
<td></td>
<td># load address (32 bits)</td>
</tr>
<tr>
<td>B</td>
<td></td>
<td># unconditional branch</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…
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 r1,r2,r3
ori r2, r4, 3
...
```

```
data
“cornell cs"
13
25
```

```
text
add r1,r2,r3
ori r2, r4, 3
...
```
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, Registers, ALU, Control, Data Memory, Program 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
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

Symbols and References

math.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;
}

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

Example:

```
  bne $1, $2, L    Looking for L
  sll $0, $0, 0
  L: addiu $2, $3, 0x2  Found L
```

The assembler will change this to

```
  bne $1, $2, +1
  sll $0, $0, 0
  addiu $2, $3, $0x2
```

Final machine code

```
0X14220001 # bne    actually: 000101...
0x00000000 # sll     000000...
0x24620002 # addiu    001001...
```
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

> mipsel-linux-objdump --disassemble math.o

Disassembly of section .text:

00000000  <get_n>:
    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

prologue  prologue  unresolved
symbol

unresolved
symbol

(see symbol

(see symbol

body  body  table next slide)

body  body  table next slide)

epilogue  epilogue  epilogue

unresolved
symbol

unresolved
symbol

elsewhere in another file:

```c
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>
</tr>
</thead>
<tbody>
<tr>
<td><strong>000000000 1 d</strong></td>
<td>df <em>ABS</em></td>
<td>00000000</td>
</tr>
<tr>
<td><strong>000000000 1 d</strong></td>
<td>.text</td>
<td>00000000</td>
</tr>
<tr>
<td><strong>000000000 1 d</strong></td>
<td>.data</td>
<td>00000000</td>
</tr>
<tr>
<td><strong>000000000 1 d</strong></td>
<td>.bss</td>
<td>00000000</td>
</tr>
<tr>
<td><strong>000000008 1 o</strong></td>
<td>.data</td>
<td>00000004</td>
</tr>
<tr>
<td><strong>000000060 1 f</strong></td>
<td>.text</td>
<td>00000028</td>
</tr>
<tr>
<td><strong>000000000 1 d</strong></td>
<td>.rodata</td>
<td>00000000</td>
</tr>
<tr>
<td><strong>000000000 1 d</strong></td>
<td>.comment</td>
<td>00000000</td>
</tr>
<tr>
<td><strong>000000000 1 g</strong></td>
<td>.data</td>
<td>00000004</td>
</tr>
<tr>
<td><strong>000000004 1 g</strong></td>
<td>.data</td>
<td>00000004</td>
</tr>
<tr>
<td><strong>000000000 1 g</strong></td>
<td>.text</td>
<td>00000028</td>
</tr>
<tr>
<td><strong>00000028 1 g</strong></td>
<td>.text</td>
<td>00000038</td>
</tr>
<tr>
<td><strong>00000088 1 g</strong></td>
<td>.text</td>
<td>0000004c</td>
</tr>
<tr>
<td><strong>000000000 <em>UND</em></strong></td>
<td></td>
<td>00000000</td>
</tr>
<tr>
<td><strong>000000000 <em>UND</em></strong></td>
<td></td>
<td>00000000</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 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
       simulate sum 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, strftime.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

★ 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
Linker Example: Resolving an External Fn Call

main.o

<table>
<thead>
<tr>
<th>.text</th>
</tr>
</thead>
<tbody>
<tr>
<td>40 0C000000</td>
</tr>
<tr>
<td>48 1b80050C</td>
</tr>
<tr>
<td>50 21047002</td>
</tr>
</tbody>
</table>

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 0C000000
2C 1b301402
30 3C040000
34 34040000
20 T get_n
00 D pi
*UND* printf
*UND* usrid
28,JAL, printf

printf.o

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

... JAL printf
... JAL ???

Unresolved references to printf and get_n

sum.exe

<table>
<thead>
<tr>
<th>.text</th>
</tr>
</thead>
<tbody>
<tr>
<td>21032040</td>
</tr>
<tr>
<td>0C40023C</td>
</tr>
<tr>
<td>1b301402</td>
</tr>
<tr>
<td>3C041000</td>
</tr>
<tr>
<td>34040004</td>
</tr>
</tbody>
</table>

main

<table>
<thead>
<tr>
<th>.text</th>
</tr>
</thead>
<tbody>
<tr>
<td>21035000</td>
</tr>
<tr>
<td>1b80050C</td>
</tr>
<tr>
<td>8C048004</td>
</tr>
<tr>
<td>21047002</td>
</tr>
<tr>
<td>0C000000</td>
</tr>
</tbody>
</table>

... JAL get_n
... JAL printf

printf

<table>
<thead>
<tr>
<th>.text</th>
</tr>
</thead>
<tbody>
<tr>
<td>0C40023C</td>
</tr>
<tr>
<td>1b80050C</td>
</tr>
<tr>
<td>8C048004</td>
</tr>
<tr>
<td>21047002</td>
</tr>
<tr>
<td>0C400020</td>
</tr>
</tbody>
</table>

main

<table>
<thead>
<tr>
<th>.data</th>
</tr>
</thead>
<tbody>
<tr>
<td>10201000</td>
</tr>
<tr>
<td>21040330</td>
</tr>
<tr>
<td>22500102</td>
</tr>
</tbody>
</table>

... global variables...go here (later)

Entry: 0040 0100
text: 0040 0000
data: 1000 0000
iClicker Question 2

Which of the following symbols are currently assigned the same location?

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

Unresolved references to printf and get_n
Linker Example: Loading a Global Variable

main.o

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

Symbol table

- T main
- D userid
- *UND* printf
- *UND* pi
- *UND* get_n

.text

- 40 JAL, printf
- 54 JAL, get_n

Math.o

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

Relocation info

- 24 T get_n
- 00 D printf
- *UND* printf
- *UND* usrid
- 28 JAL printf
- 30 LUI usrid
- 34 LA usrid

sum.exe

```
0040 0000
```

```
21032040
0C400023C
1b301402
3C041000
34040004
```

```
0040 0100
```

```
math
```

```
main
```

```
printf
```

```
40,JAL, printf
```

```
54,JAL, get_n
```

```
28,JAL, printf
```

```
30,LUI, usrid
```

```
34,LA, usrid
```

```
LA = LUI/ORI "userid" -> ???
Unresolved references to userid
Need address of global variable
```

```
Notice: userid gets relocated due to collision with pi
```

```
Entry: 0040 0100
```

```
text: 0040 0000
```

```
data: 1000 0000
```

```
pi 00000003
```

```
userid 0077616B
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```
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;
}
Compiler

C source files

math.c

Assembler

assembly files

math.s

Linker

obj files

sum.s

io.s

Executable program

C source files

Sum.c

Math.c

Libc.o

Math.o

Libm.o

obj files

Executing in Memory process

sum.exe

Executable program

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)