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?

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
001000000000010100000000000001010
00100000101001010000000000001111

A
B
32
RF
32

Silicon

machine
code
loader
CPU
Circuits
Gates
Transistors
Silicon

add
r0
r5
10

001000000000010100000000000001010
00100000101001010000000000001111

A
B
32
RF
32

Silicon

machine
code
loader
CPU
Circuits
Gates
Transistors
Silicon

add
r0
r5
10

001000000000010100000000000001010
00100000101001010000000000001111

A
B
32
RF
32

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`
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
#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
```
**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)
       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,$2
       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
```
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)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Binary Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi r5, r0, 10</td>
<td>00100000000001010000000000001010</td>
</tr>
<tr>
<td>muli r5, r5, 2</td>
<td>00000000000001010101010000001111</td>
</tr>
<tr>
<td>addi r5, r5, 15</td>
<td>00100000101001010000000000001111</td>
</tr>
</tbody>
</table>
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>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>
<tr>
<td>+ a few more...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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
...
```

<table>
<thead>
<tr>
<th>data</th>
<th>“cornell cs”</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>25</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>text</th>
<th>add r1,r2,r3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ori r2, r4, 3</td>
</tr>
<tr>
<td></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:
- CPU
  - Registers
  - ALU
  - Control
  - Data, address, control
  - Program Memory
  - Data Memory

Excerpt:
00100000001
00100000010
00010000100
...
Program Memory

0110001000
10110000011
0010010101
...
Data 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

math.c

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

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

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

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

Example:

\[
\text{bne } \$1, \$2, \text{ L } \quad \text{Looking for L}
\]
\[
\text{sll } \$0, \$0, 0
\]

\[
\text{L: addiu } \$2, \$3, 0x2 \quad \text{Found L}
\]

The assembler will change this to

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

Final machine code

\[
0x14220001 \ # \text{ bne } \quad \text{actually: } \ 000101\ldots
\]
\[
0x00000000 \ # \text{ sll } \quad \text{ actually: } \ 000000\ldots
\]
\[
0x24620002 \ # \text{ addiu } \quad \text{ actually: } \ 001001\ldots
\]
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:**

```assembly
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
```

*elsewhere in another file:*

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

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

<table>
<thead>
<tr>
<th>SYMBOL TABLE:</th>
<th>segment</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000000 l df <em>ABS</em></td>
<td>0000000000</td>
<td>math.c</td>
</tr>
<tr>
<td>0000000000 l d .text</td>
<td>0000000000</td>
<td>.text</td>
</tr>
<tr>
<td>0000000000 l d .data</td>
<td>0000000000</td>
<td>.data</td>
</tr>
<tr>
<td>0000000000 l d .bss</td>
<td>0000000000</td>
<td>.bss</td>
</tr>
<tr>
<td>0000000008 l O .data</td>
<td>0000000004</td>
<td>randomval</td>
</tr>
<tr>
<td>0000000600 l F .text</td>
<td>000000028</td>
<td>is_prime</td>
</tr>
<tr>
<td>0000000000 l d .rodata</td>
<td>0000000000</td>
<td>.rodata</td>
</tr>
<tr>
<td>0000000000 l d .comment</td>
<td>0000000000</td>
<td>.comment</td>
</tr>
<tr>
<td>0000000000 g O .data</td>
<td>0000000004</td>
<td>pi</td>
</tr>
<tr>
<td>0000000004 g O .data</td>
<td>0000000004</td>
<td>e</td>
</tr>
<tr>
<td>0000000000 g F .text</td>
<td>000000028</td>
<td>get_n</td>
</tr>
<tr>
<td>0000000280 g F .text</td>
<td>000000038</td>
<td>square</td>
</tr>
<tr>
<td>000000088 g F .text</td>
<td>00000004c</td>
<td>pick_prime</td>
</tr>
<tr>
<td>00000000 <em>UND</em></td>
<td>0000000000</td>
<td>usrid</td>
</tr>
<tr>
<td>00000000 <em>UND</em></td>
<td>0000000000</td>
<td>printf</td>
</tr>
</tbody>
</table>
Separate Compilation & Assembly

Compiler  Assembler  Linker

sum.c  sum.s  sum.o
source files  assembly files  obj files

math.c  math.s  math.o

executable program
exists on disk
loader

Executing in Memory process
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?!?
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

math.o

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

... *

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

printf.o

.text
... *

3C T printf

math.o

.text
0040 0000
21032040
0C40023C
1b301402
3C041000
34040004

... *

0C40023C
21035000
1b80050C
8C048004
21047002
0C400020

printf.o

.text
0040 0000
0040 0000
0040 0000

... *

0040 0100
10201000
21040330
22500102

... *

10201000
21040330
22500102

global variables

Entry: 0040 0100
text: 0040 0000
data: 1000 0000
The diagram illustrates the process of compiling, assembling, and linking C source files to create an executable program. The C source files, `sum.c` and `math.c`, are compiled into assembly files, `sum.s` and `math.s`, respectively. These assembly files are then assembled into object files, `sum.o` and `math.o`. The object files, along with other object files (`io.o`, `libc.o`, and `libm.o`), are linked together to form the executable program, `sum.exe`. This executable program exists on disk and is loaded into memory for execution.
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?!
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 1\textsuperscript{st} insn, and starts executing a process
  (machine code)