Translation and Startup

Many compilers produce object modules directly.

- C program
  - Compiler
  - Assembly language program
  - Assembler
    - Object: Machine language module
    - Object: Library routine (machine language)
  - Linker
    - Executable: Machine language program
  - Loader
    - Memory

Static linking
Steps to Starting a Program (translation)

C program: foo.c

Compiler

Assembly program: foo.s

Assembler

Object (mach lang module): foo.o

Linker

Executable (mach lang pgm): a.out

Loader

Memory
Compiler

- **Input:** High-Level Language Code (e.g., C, Java such as `foo.c`)
- **Output:** Assembly Language Code (e.g., `foo.s` for MIPS)
- **Note:** Output *may* contain pseudoinstructions
- **Pseudoinstructions:** instructions that assembler understands but not in machine

For example:

```
move $s1,$s2 ⇒ or $s1,$s2,$zero
```
Where Are We Now?

- C program: foo.c
- Compiler
- Assembly program: foo.s
- Assembler
- Object (mach lang module): foo.o
- Linker
- Executable (mach lang pgm): a.out
- Loader
- Memory
- lib.o
Assembler

- **Input:** Assembly Language Code (MAL) (e.g., `foo.s` for MIPS)
- **Output:** Object Code, information tables (TAL) (e.g., `foo.o` for MIPS)
- Reads and Uses **Directives**
- Replace Pseudoinstructions
- Produce Machine Language
- Creates **Object File**
Assembler Directives (p. A-51 to A-53)

- Give directions to assembler, but do not produce machine instructions
  - `.text:` Subsequent items put in user text segment (machine code)
  - `.data:` Subsequent items put in user data segment (binary rep of data in source file)
  - `.globl sym:` declares sym global and can be referenced from other files
  - `.asciiz str:` Store the string str in memory and null-terminate it
  - `.word w1...wn:` Store the n 32-bit quantities in successive memory words
Pseudoinstruction Replacement

- Asm. treats convenient variations of machine language instructions as if real instructions.

**Pseudo:**

<table>
<thead>
<tr>
<th>Pseudo</th>
<th>Real</th>
</tr>
</thead>
<tbody>
<tr>
<td>subu $sp,$sp,32</td>
<td>addiu $sp,$sp,-32</td>
</tr>
<tr>
<td>sd $a0, 32($sp)</td>
<td>sw $a0, 32($sp)</td>
</tr>
<tr>
<td>mul $t7,$t6,$t5</td>
<td>mul $t6,$t5</td>
</tr>
<tr>
<td>addu $t0,$t6,1</td>
<td>addiu $t0,$t6,1</td>
</tr>
<tr>
<td>bne $t0,100,loop</td>
<td>slti $at,$t0,101</td>
</tr>
<tr>
<td>la $a0, str</td>
<td>lui $at,left(str)</td>
</tr>
</tbody>
</table>
Producing Machine Language (1/3)

• Simple Case
  • Arithmetic, Logical, Shifts, and so on.
  • All necessary info is within the instruction already.

• What about Branches?
  • PC-Relative
  • So once pseudo-instructions are replaced by real ones, we know by how many instructions to branch.

• So these can be handled.
Producing Machine Language (2/3)

• “Forward Reference” problem

• Branch instructions can refer to labels that are “forward” in the program:

  or $v0, $0, $0

  L1: slt $t0, $0, $a1
  beq $t0, $0, L2
  addi $a1, $a1, -1
  j L1

  L2: add $t1, $a0, $a1

• Solved by taking 2 passes over the program.
  • First pass remembers position of labels
  • Second pass uses label positions to generate code
• What about jumps (j and jal)?
  • Jumps require **absolute address**.
  • So, forward or not, still can’t generate machine instruction without knowing the position of instructions in memory.

• What about references to data?
  • *la* gets broken up into *lui* and *ori*
  • These will require the full 32-bit address of the data.

• These can’t be determined yet, so we create two tables…
Symbol Table

• List of “items” in this file that may be used by other files.

• What are they?
  • Labels: function calling
  • Data: anything in the `.data` section; variables which may be accessed across files
Relocation Table

- List of “items” this file needs the address later.
- What are they?
  - Any label jumped to: \texttt{j} or \texttt{jal}
    - internal
    - external (including lib files)
  - Any piece of data
    - such as the \texttt{la} instruction
Object File Format

- **object file header**: size and position of the other pieces of the object file
- **text segment**: the machine code
- **data segment**: binary representation of the data in the source file
- **relocation information**: identifies lines of code that need to be “handled”
- **symbol table**: list of this file’s labels and data that can be referenced
- **debugging information**
Where Are We Now?

C program: foo.c

Compiler

Assembly program: foo.s

Assembler

Object (mach lang module): foo.o

Linker

Executable (mach lang pgm): a.out

Loader

Memory

lib.o
Linker (1/3)

- **Input**: Object Code files, information tables (e.g., foo.o, libc.o for MIPS)
- **Output**: Executable Code (e.g., a.out for MIPS)
- Combines several object (.o) files into a single executable (“linking”)
- Enable Separate Compilation of files
  - Changes to one file do not require recompilation of whole program
  - Old name “Link Editor” from editing the “links” in jump and link instructions
Linker (2/3)

.o file 1
- text 1
- data 1
- info 1

.o file 2
- text 2
- data 2
- info 2

Linker

a.out
- Relocated text 1
- Relocated text 2
- Relocated data 1
- Relocated data 2
Linke (3/3)

- **Step 1:** Take text segment from each `.o` file and put them together.
- **Step 2:** Take data segment from each `.o` file, put them together, and concatenate this onto end of text segments.
- **Step 3:** Resolve References
  - Go through Relocation Table; handle each entry
  - That is, fill in all absolute addresses
Four Types of Addresses we’ll discuss

- PC-Relative Addressing ($\text{beq, bne}$)
  - never relocate
- Absolute Address ($j, \text{jal}$)
  - always relocate
- External Reference (usually $\text{jal}$)
  - always relocate
- Data Reference (often $\text{lui and ori}$)
  - always relocate
**Absolute Addresses in MIPS**

- Which instructions need relocation editing?
  - J-format: jump, jump and link

<table>
<thead>
<tr>
<th>j/jal</th>
<th>xxxxx</th>
</tr>
</thead>
</table>

- Loads and stores to variables in static area, relative to global pointer

<table>
<thead>
<tr>
<th>lw/sw</th>
<th>$gp</th>
<th>$x</th>
<th>address</th>
</tr>
</thead>
</table>

- What about conditional branches?

<table>
<thead>
<tr>
<th>beq/bne</th>
<th>$rs</th>
<th>$rt</th>
<th>address</th>
</tr>
</thead>
</table>

- PC-relative addressing preserved even if code moves
Resolving References (1/2)

• Linker *assumes* first word of first text segment is at address 0x00000000.
  • (More later when we study “virtual memory”)

• Linker knows:
  • length of each text and data segment
  • ordering of text and data segments

• Linker calculates:
  • absolute address of each label to be jumped to (internal or external) and each piece of data being referenced
Resolving References (2/2)

- To resolve references:
  - search for reference (data or label) in all “user” symbol tables
  - if not found, search library files (for example, for printf)
  - once absolute address is determined, fill in the machine code appropriately

- Output of linker: executable file containing text and data (plus header)
Where Are We Now?

C program: foo.c

Compiler

Assembly program: foo.s

Assembler

Object (mach lang module): foo.o

Linker

Executable (mach lang pgm): a.out

Loader

Memory

lib.o

Dr Dan Garcia
Loader Basics

• Input: Executable Code (e.g., a.out for MIPS)
• Output: (program is run)
• Executable files are stored on disk.
• When one is run, loader’s job is to load it into memory and start it running.
• In reality, loader is the operating system (OS)
  • loading is one of the OS tasks
Loader ... what does it do?

- Reads executable file’s header to determine size of text and data segments
- Creates new address space for program large enough to hold text and data segments, along with a stack segment
- Copies instructions and data from executable file into the new address space
- Copies arguments passed to the program onto the stack
- Initializes machine registers
  - Most registers cleared, but stack pointer assigned address of 1st free stack location
- Jumps to start-up routine that copies program’s arguments from stack to registers & sets the PC
  - If main routine returns, start-up routine terminates program with the exit system call
Conclusion

- Compiler converts a single HLL file into a single assembly language file.
- Assembler removes pseudo instructions, converts what it can to machine language, and creates a checklist for the linker (relocation table). A `.s` file becomes a `.o` file.
  - Does 2 passes to resolve addresses, handling internal forward references
- Linker combines several `.o` files and resolves absolute addresses.
  - Enables separate compilation, libraries that need not be compiled, and resolves remaining addresses
- Loader loads executable into memory and begins execution.

- Stored Program concept is very powerful. It means that instructions sometimes act just like data. Therefore we can use programs to manipulate other programs!
  - Compiler ⇒ Assembler ⇒ Linker ⇒ Loader

Dr Dan Garcia
Peer Instruction

Which of the following instr. may need to be edited during link phase?

Loop: lui $at, 0xABCD
     ori $a0,$at, 0xFEDC} # 1
     bne $a0,$v0, Loop # 2

Dr Dan Garcia
Which of the following instructions may need to be edited during link phase?

Loop:  
lui $at, 0xABCD
ori $a0,$at, 0xFEDC
bne $a0,$v0, Loop

Data reference; relocate  

Dr Dan Garcia
Static vs Dynamically linked libraries

• What we’ve described is the traditional way: **statically-linked** approach
  • The library is now part of the executable, so if the library updates, we don’t get the fix (have to recompile if we have source)
  • It includes the **entire** library even if not all of it will be used.
  • Executable is self-contained.

• An alternative is **dynamically linked libraries (DLL)**, common on Windows & UNIX platforms
Dynamically linked libraries

- **Space/time issues**
  - + Storing a program requires less disk space
  - + Sending a program requires less time
  - + Executing two programs requires less memory (if they share a library)
  - – At runtime, there’s time overhead to do link

- **Upgrades**
  - + Replacing one file (`libXYZ.so`) upgrades every program that uses library “XYZ”
  - – Having the executable isn’t enough anymore

Overall, dynamic linking adds quite a bit of complexity to the compiler, linker, and operating system. However, it provides many benefits that often outweigh these.
Dynamically linked libraries

- The prevailing approach to dynamic linking uses machine code as the “lowest common denominator”
  - The linker does not use information about how the program or library was compiled (i.e., what compiler or language)
  - This can be described as “linking at the machine code level”
  - This isn’t the only way to do it...
C Program Source Code: prog.c

#include <stdio.h>
int main (int argc, char *argv[]) {
    int i, sum = 0;
    for (i = 0; i <= 100; i++)
        sum = sum + i * i;
    printf ("The sum of sq from 0 .. 100 is \%d\n", sum);
}

"printf" lives in "libc"
Compilation: MAL

```assembly
.globl main
main:
    subu $sp, $sp, 32
    sw $ra, 20($sp)
    sd $a0, 32($sp)
    sw $0, 24($sp)
    sw $0, 28($sp)
loop:
    lw $t6, 28($sp)
    mul $t7, $t6,$t6
    lw $t8, 24($sp)
    addu $t9,$t8,$t7
    sw $t9, 24($sp)
    addu $t0, $t6, 1
    sw $t0, 28($sp)
    ble $t0,100, loop
    la $a0, str
    lw $a1, 24($sp)
    jal printf
    move $v0, $0
    lw $ra, 20($sp)
    addiu $sp,$sp,32
    jr $ra
.data
    .align 0
    str:
    .asciiz "The sum of sq from 0 .. 100 is %d\n"
```

Where are 7 pseudo-instructions?
Compilation: MAL

```assembly
.globl main
main:
    subu $sp, $sp, 32
    sw $ra, 20($sp)
    sd $a0, 32($sp)
    sw $0, 24($sp)
    sw $0, 28($sp)
loop:
    lw $t6, 28($sp)
    mul $t7, $t6, $t6
    lw $t8, 24($sp)
    addu $t9, $t8, $t7
    sw $t9, 24($sp)
    addu $t0, $t6, 1
    sw $t0, 28($sp)
    ble $t0, 100, loop
    la $a0, str
    lw $a1, 24($sp)
    jal printf
    move $v0, $0
    lw $ra, 20($sp)
    addiu $sp, $sp, 32
    jr $ra
.data
    .align 0
    str:
.asciiz "The sum of sq from 0 .. 100 is \%d\n"
```

7 pseudo-instructions underlined
Assembly step 1:

Remove pseudoinSTRUCTIONS, assign addresses

00 addiu $29, $29, -32
04 sw $31, 20($29)
08 sw $4, 32($29)
0c sw $5, 36($29)
10 sw $0, 24($29)
14 sw $0, 28($29)
18 lw $14, 28($29)
1c mul tu $14, $14
20 mflo $15
24 lw $24, 24($29)
28 addu $25, $24, $15
2c sw $25, 24($29)
30 addiu $8, $14, 1
34 sw $8, 28($29)
38 slti $1, $8, 101
3c bne $1, $0, loop
40 lui $4, 1.str
44 ori $4, $4, r.str
48 lw $5, 24($29)
4c jal printf
50 add $2, $0, $0
54 lw $31, 20($29)
58 addiu $29, $29, 32
5c jr $31
Assembly step 2

Create relocation table and symbol table

• Symbol Table

<table>
<thead>
<tr>
<th>Label</th>
<th>address (in module)</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
<td>0x0000000000 global</td>
<td>text</td>
</tr>
<tr>
<td>loop</td>
<td>0x0000000018 local</td>
<td>text</td>
</tr>
<tr>
<td>str</td>
<td>0x0000000000 local</td>
<td>data</td>
</tr>
</tbody>
</table>

• Relocation Information

<table>
<thead>
<tr>
<th>Address</th>
<th>Instr.</th>
<th>type</th>
<th>Dependency</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x000000040</td>
<td>lui</td>
<td></td>
<td>l.str</td>
</tr>
<tr>
<td>0x000000044</td>
<td>ori</td>
<td></td>
<td>r.str</td>
</tr>
<tr>
<td>0x00000004c</td>
<td>jal</td>
<td></td>
<td>printf</td>
</tr>
</tbody>
</table>
Assembly step 3

Resolve local PC-relative labels

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>addiu $29, $29, -32</td>
<td></td>
</tr>
<tr>
<td>04</td>
<td>sw $31, 20($29)</td>
<td></td>
</tr>
<tr>
<td>08</td>
<td>sw $4, 32($29)</td>
<td></td>
</tr>
<tr>
<td>0c</td>
<td>sw $5, 36($29)</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>sw $0, 24($29)</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>sw $0, 28($29)</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>lw $14, 28($29)</td>
<td></td>
</tr>
<tr>
<td>1c</td>
<td>multu $14, $14</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>mflo $15</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>lw $24, 24($29)</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>addu $25, $24, $15</td>
<td></td>
</tr>
<tr>
<td>2c</td>
<td>sw $25, 24($29)</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>addiu $8, $14, 1</td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>sw $8, 28($29)</td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>slti $1, $8, 101</td>
<td></td>
</tr>
<tr>
<td>3c</td>
<td>bne $1, $0, -10</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>lui $4, l.str</td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>ori $4, $4, r.str</td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>lw $5, 24($29)</td>
<td></td>
</tr>
<tr>
<td>4c</td>
<td>jal printf</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>add $2, $0, $0</td>
<td></td>
</tr>
<tr>
<td>54</td>
<td>lw $31, 20($29)</td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>addiu $29, $29, 32</td>
<td></td>
</tr>
<tr>
<td>5c</td>
<td>jr $31</td>
<td></td>
</tr>
</tbody>
</table>
**Assembly step 4**

- Generate object (.o) file:
  - Output binary representation for
    - ext segment (instructions),
    - data segment (data),
    - symbol and relocation tables.
  - Using dummy “placeholders” for unresolved absolute and external references.
Link step 1: combine prog.o, libc.o

- Merge text/data segments
- Create absolute memory addresses
- Modify & merge symbol and relocation tables

Symbol Table

<table>
<thead>
<tr>
<th>Label</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>main:</td>
<td>0x00000000</td>
</tr>
<tr>
<td>loop:</td>
<td>0x00000018</td>
</tr>
<tr>
<td>str:</td>
<td>0x10000430</td>
</tr>
<tr>
<td>printf:</td>
<td>0x000003b0</td>
</tr>
</tbody>
</table>

Relocation Information

<table>
<thead>
<tr>
<th>Address</th>
<th>Instr. Type</th>
<th>Dependency</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000040</td>
<td>lui</td>
<td>l.str</td>
</tr>
<tr>
<td>0x00000044</td>
<td>ori</td>
<td>r.str</td>
</tr>
<tr>
<td>0x0000004c</td>
<td>jal</td>
<td>printf</td>
</tr>
</tbody>
</table>
Link step 2:

• Edit Addresses in relocation table
  • (shown in TAL for clarity, but done in binary )

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>addiu</td>
<td>$29, $29, -32</td>
<td></td>
</tr>
<tr>
<td>04</td>
<td>sw</td>
<td>$31, 20($29)</td>
<td></td>
</tr>
<tr>
<td>08</td>
<td>sw</td>
<td>$4, 32($29)</td>
<td></td>
</tr>
<tr>
<td>0C</td>
<td>sw</td>
<td>$5, 36($29)</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>sw</td>
<td>$0, 24($29)</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>sw</td>
<td>$0, 28($29)</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>lw</td>
<td>$14, 28($29)</td>
<td></td>
</tr>
<tr>
<td>1C</td>
<td>multu</td>
<td>$14, $14</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>mflo</td>
<td>$15</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>lw</td>
<td>$24, 24($29)</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>addu</td>
<td>$25, $24, $15</td>
<td></td>
</tr>
<tr>
<td>2C</td>
<td>sw</td>
<td>$25, 24($29)</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>addiu</td>
<td>$8, $14, 1</td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>sw</td>
<td>$8, 28($29)</td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>slti</td>
<td>$1, $8, 101</td>
<td></td>
</tr>
<tr>
<td>3C</td>
<td>bne</td>
<td>$1, $0, -10</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>lui</td>
<td>$4, 4096</td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>ori</td>
<td>$4, $4, 1072</td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>lw</td>
<td>$5, 24($29)</td>
<td></td>
</tr>
<tr>
<td>4C</td>
<td>jal</td>
<td>812</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>add</td>
<td>$2, $0, $0</td>
<td></td>
</tr>
<tr>
<td>54</td>
<td>lw</td>
<td>$31, 20($29)</td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>addiu</td>
<td>$29, $29, 32</td>
<td></td>
</tr>
<tr>
<td>5C</td>
<td>jr</td>
<td>$31</td>
<td></td>
</tr>
</tbody>
</table>
Link step 3:

- Output executable of merged modules.
  - Single text (instruction) segment
  - Single data segment
  - Header detailing size of each segment