inst.eecs.berkeley.edu/~cs61c

Running a Program
(Compiling, Assembling, Linking, Loading)

UCB CS61C : GREAT IDEAS IN
COMPUTER ARCHITECTURE
An **Interpreter** is a program that executes other programs.

- Language **translation** gives us another option.
- In general, we **interpret** a high-level language when efficiency is not critical and **translate** to a lower-level language to up performance.
Interpretation vs Translation

- How do we run a program written in a source language?
  - **Interpreter**: Directly executes a program in the source language
  - **Translator**: Converts a program from the source language to an equivalent program in another language

- For example, consider a Python program `foo.py`
Python interpreter is just a program that reads a python program and performs the functions of that python program.
Interpretation

- Any good reason to interpret machine language in software?
- MARS—useful for learning / debugging
- Apple Macintosh conversion
  - Switched from Motorola 680x0 instruction architecture to PowerPC.
    - Similar issue with switch to x86.
  - Could require all programs to be re-translated from high level language
  - Instead, let executables contain old and/or new machine code, interpret old code in software if necessary (emulation)
Interpretation vs. Translation? (1/2)

- Generally easier to write interpreter
- Interpreter closer to high-level, so can give better error messages (e.g., MARS, stk)
  - Translator reaction: add extra information to help debugging (line numbers, names)
- Interpreter slower (10x?), code smaller (2x?)
- Interpreter provides instruction set independence: run on any machine
Interpretation vs. Translation? (2/2)

- Translated/compiled code almost always more efficient and therefore higher performance:
  - Important for many applications, particularly operating systems.

- Translation/compilation helps “hide” the program “source” from the users:
  - One model for creating value in the marketplace (eg. Microsoft keeps all their source code secret)
  - Alternative model, “open source”, creates value by publishing the source code and fostering a community of developers.
Steps in compiling a C program

1. C program: `foo.c`
2. Compiler
3. Assembly program: `foo.s`
4. Assembler
5. Object (mach lang module): `foo.o`
6. Linker
7. Executable (mach lang pgm): `a.out`
8. Loader
9. Memory

Note: `lib.o` is an external library file.
Compiler

- **Input:** High-Level Language Code (e.g., *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 ⇒ add $s1,$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

CS164
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.. 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:  |  Real:
--- | ---
\text{subu} $sp,$sp,32  |  \text{addiu} $sp,$sp,\text{-32}
\text{sd} $a0, 32($sp)  |  \text{sw} $a0, 32($sp)
\text{sd} $a1, 36($sp)  |  \text{sw} $a1, 36($sp)
\text{mul} $t7,$t6,$t5  |  \text{mul} $t6,$t5
\text{addu} $t0,$t6,1  |  \text{addiu} $t0,$t6,1
\text{ble} $t0,100,\text{loop}  |  \text{slti} $at,$t0,101
\text{la} $a0, str  |  \text{lui} $at,\text{left}(str)$

ori $a0,$at,right(str)
Paper and pencil example (unsigned):

Multiplicand  1000  8
Multiplier     x1001  9

1000
0000
0000
0000
+1000
---
01001000

m bits x n bits = m + n bit product
In MIPS, we multiply registers, so:
- 32-bit value x 32-bit value = 64-bit value

Syntax of Multiplication (signed):
- `mult register1, register2`
- Multiplies 32-bit values in those registers & puts 64-bit product in special result regs:
  - puts product upper half in `hi`, lower half in `lo`
- `hi` and `lo` are 2 registers separate from the 32 general purpose registers
- Use `mfhi register` & `mflo register` to move from `hi`, `lo` to another register
Example:

- in C:  \( a = b \times c; \)
- in MIPS:
  - let \( b \) be \( $s2 \); let \( c \) be \( $s3 \); and let \( a \) be \( $s0 \) and \( $s1 \) (since it may be up to 64 bits)

  ```
  mult $s2,$s3 \# b*c
  mfhi $s0 \# upper half of
  \# product into $s0
  mflo $s1 \# lower half of
  \# product into $s1
  ```

Note: Often, we only care about the lower half of the product.

Pseudo-inst. `mul` expands to `mult/mflo.`
Integer Division (1/2)

- Paper and pencil example (unsigned):

  \[
  \begin{array}{c|c}
  \text{Divisor} & 1001 \\
  \hline
  \text{Quotient} & 1000 | 1001010 \\
  \hline
  \text{Dividend} & -1000 \\
  \hline
  \text{Remainder} & 10 \\
  \hline
  \text{10} \\
  \hline
  \text{101} \\
  \hline
  \text{1010} \\
  \hline
  \text{-1000} \\
  \hline
  \text{10} \\
  \hline
  \text{Remainder} \\
  \text{(or Modulo result)}
  \end{array}
  \]

- Dividend = Quotient x Divisor + Remainder
Syntax of Division (signed):
- `div` register1, register2
- Divides 32-bit register 1 by 32-bit register 2:
- Puts remainder of division in `hi`, quotient in `lo`

Implements C division (`/`) and modulo (`%`)

Example in C: 
```
a = c / d;
b = c % d;
```

In MIPS: 
```
a←$s0; b←$s1; c←$s2; d←$s3
div $s2,$s3    # lo=c/d, hi=c%d
mflo $s0       # get quotient
mfhi $s1      # get remainder
Administrivia

- HW2 due Sunday Feb 15
- Must register your Project 1 team (you and a partner) by this Friday @ 23:59:59 (Tomorrow!)
  - Not registering in time = lost EPA
  - Pinned Piazza post for finding a partner
  - Project 1 Part 1 will release Sunday
- HW3 also out Sunday (but is ungraded midterm prep)
In the news: RISC-I ceremony

- At 3:30pm TODAY, Plaque unveiled in Soda Hall
- IEEE MILESTONE IN ELECTRICAL ENGINEERING AND COMPUTING First RISC (Reduced Instruction-Set Computing) Microprocessor 1980-1982 Berkeley students designed and built the first VLSI reduced instruction-set computer in 1981. The simplified instructions of RISC-I reduced the hardware for instruction decode and control, which enabled a flat 32-bit address space, a large set of registers, and pipelined execution. A good match to C programs and the Unix operating system, RISC-I influenced instruction sets widely used today, including those for game consoles, smartphones and tablets.
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.
“Forward Reference” problem

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

```assembly
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 static 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 of later.
- What are they?
  - Any label jumped to: j or jal
    - internal
    - external (including lib files)
  - Any piece of data in static section
    - such as the 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 static data in the source file
- **relocation information**: identifies lines of code that need to be fixed up later
- **symbol table**: list of this file’s labels and static data that can be referenced
- **debugging information**

A standard format is ELF (except MS)

http://www.skyfree.org/linux/references/ELF_Format.pdf
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
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
    - Windows NT source was > 40 M lines of code!
  - 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

Relocated text 1
Relocated text 2
Relocated data 1
Relocated data 2

a.out
Linker (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

- **PC-Relative Addressing** (*beq, bne*)
  - *never relocate*

- **Absolute Function Address** (*j, jal*)
  - *always relocate*

- **External Function Reference** (usually *jal*)
  - *always relocate*

- **Static Data Reference** (often *lui and ori*)
  - *always relocate*
Absolute Addresses in MIPS

- Which instructions need relocation editing?
  - J-format: jump, jump and link
  - Loads and stores to variables in static area, relative to global pointer
  - What about conditional branches?
    - PC-relative addressing preserved even if code moves

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Address Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>j/jal</td>
<td></td>
<td>xxxxx</td>
</tr>
<tr>
<td>lw/sw</td>
<td>$gp</td>
<td>$x</td>
</tr>
<tr>
<td>beq/bne</td>
<td>$rs</td>
<td>$rt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>address</td>
</tr>
</tbody>
</table>
Resolving References (1/2)

- Linker assumes first word of first text segment is at address 0x04000000.
  - (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?

1. C program: `foo.c`
2. Compiler
3. Assembly program: `foo.s`
4. Assembler
5. Object (mach lang module): `foo.o`
6. Linker
7. Executable (mach lang pgm): `a.out`
8. Library: `lib.o`
9. Loader
10. Memory
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
Clicker/Peer Instruction

At what point in process are all the machine code bits determined for the following assembly instructions:

1) addu $6, $7, $8
2) jal fprintf

A: 1) & 2) After compilation
B: 1) After compilation, 2) After assembly
C: 1) After assembly, 2) After linking
D: 1) After assembly, 2) After loading
E: 1) After compilation, 2) After linking
#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

```
.text
.align 2
.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
.text
.align 2
.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

<table>
<thead>
<tr>
<th>Line</th>
<th>Instruction</th>
<th>Address</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, loop</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 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</td>
<td>global text</td>
</tr>
<tr>
<td>loop:</td>
<td>0x000000018</td>
<td>local text</td>
</tr>
<tr>
<td>str:</td>
<td>0x0000000000</td>
<td>local 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>0x00000040</td>
<td>lui</td>
<td></td>
<td>l.str</td>
</tr>
<tr>
<td>0x00000044</td>
<td>ori</td>
<td></td>
<td>r.str</td>
</tr>
<tr>
<td>0x0000004c</td>
<td>jal</td>
<td></td>
<td>printf</td>
</tr>
</tbody>
</table>
## Assembly step 3

### Resolve local PC-relative labels

<table>
<thead>
<tr>
<th>Line</th>
<th>Instruction</th>
<th>Address</th>
<th>Value 1</th>
<th>Value 2</th>
<th>Value 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>addiu $29, $29, -32</td>
<td>0x00</td>
<td>$29</td>
<td>$29</td>
<td>-32</td>
</tr>
<tr>
<td>04</td>
<td>sw $31, 20($29)</td>
<td>0x04</td>
<td>$31</td>
<td>20</td>
<td>$29</td>
</tr>
<tr>
<td>08</td>
<td>sw $4, 32($29)</td>
<td>0x08</td>
<td>$4</td>
<td>32</td>
<td>$29</td>
</tr>
<tr>
<td>0c</td>
<td>sw $5, 36($29)</td>
<td>0x0c</td>
<td>$5</td>
<td>36</td>
<td>$29</td>
</tr>
<tr>
<td>10</td>
<td>sw $0, 24($29)</td>
<td>0x10</td>
<td>$0</td>
<td>24</td>
<td>$29</td>
</tr>
<tr>
<td>14</td>
<td>sw $0, 28($29)</td>
<td>0x14</td>
<td>$0</td>
<td>28</td>
<td>$29</td>
</tr>
<tr>
<td>18</td>
<td>lw $14, 28($29)</td>
<td>0x18</td>
<td>$14</td>
<td>28</td>
<td>$29</td>
</tr>
<tr>
<td>1c</td>
<td>multu $14, $14</td>
<td>0x1c</td>
<td>$14</td>
<td>$14</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>mflo $15</td>
<td>0x20</td>
<td>$15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>lw $24, 24($29)</td>
<td>0x24</td>
<td>$24</td>
<td>24</td>
<td>$29</td>
</tr>
<tr>
<td>28</td>
<td>addu $25, $24, $15</td>
<td>0x28</td>
<td>$25</td>
<td>$24</td>
<td>$15</td>
</tr>
<tr>
<td>2c</td>
<td>sw $25, 24($29)</td>
<td>0x2c</td>
<td>$25</td>
<td>24</td>
<td>$29</td>
</tr>
<tr>
<td>30</td>
<td>addiu $8, $14, 1</td>
<td>0x30</td>
<td>$8</td>
<td>$14</td>
<td>1</td>
</tr>
<tr>
<td>34</td>
<td>sw $8, 28($29)</td>
<td>0x34</td>
<td>$8</td>
<td>28</td>
<td>$29</td>
</tr>
<tr>
<td>38</td>
<td>slti $1, $8, 101</td>
<td>0x38</td>
<td>$1</td>
<td>$8</td>
<td>101</td>
</tr>
<tr>
<td>3c</td>
<td>bne $1, $0, -10</td>
<td>0x3c</td>
<td>$1</td>
<td>$0</td>
<td>-10</td>
</tr>
<tr>
<td>40</td>
<td>lui $4, l.str</td>
<td>0x40</td>
<td>$4</td>
<td></td>
<td>l.str</td>
</tr>
<tr>
<td>44</td>
<td>ori $4, $4, r.str</td>
<td>0x44</td>
<td>$4</td>
<td>$4</td>
<td>r.str</td>
</tr>
<tr>
<td>48</td>
<td>lw $5, 24($29)</td>
<td>0x48</td>
<td>$5</td>
<td>24</td>
<td>$29</td>
</tr>
<tr>
<td>4c</td>
<td>jal printf</td>
<td>0x4c</td>
<td></td>
<td></td>
<td>printf</td>
</tr>
<tr>
<td>50</td>
<td>add $2, $0, $0</td>
<td>0x50</td>
<td>$2</td>
<td>$0</td>
<td>$0</td>
</tr>
<tr>
<td>54</td>
<td>lw $31, 20($29)</td>
<td>0x54</td>
<td>$31</td>
<td>20</td>
<td>$29</td>
</tr>
<tr>
<td>58</td>
<td>addiu $29, $29, 32</td>
<td>0x58</td>
<td>$29</td>
<td>$29</td>
<td>32</td>
</tr>
<tr>
<td>5c</td>
<td>jr $31</td>
<td>0x5c</td>
<td></td>
<td></td>
<td>$31</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.
<table>
<thead>
<tr>
<th>Address</th>
<th>Binary Code</th>
<th>Hex Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x000000</td>
<td>00100111101111011111111111111110000000</td>
<td>0x000000</td>
</tr>
<tr>
<td>0x000004</td>
<td>1010111110111111100000000000010100000</td>
<td>0x000004</td>
</tr>
<tr>
<td>0x000008</td>
<td>1010111110100100000000000010000000000</td>
<td>0x000008</td>
</tr>
<tr>
<td>0x00000c</td>
<td>101011111011111000000000000001111000000</td>
<td>0x00000c</td>
</tr>
<tr>
<td>0x000010</td>
<td>100011111011111000000000000001111000000</td>
<td>0x000010</td>
</tr>
<tr>
<td>0x000014</td>
<td>1000111110100100000000000010000000000</td>
<td>0x000014</td>
</tr>
<tr>
<td>0x000018</td>
<td>000000111001110000000000001100111111111</td>
<td>0x000018</td>
</tr>
<tr>
<td>0x00001c</td>
<td>001001011100100000000000000000010000000</td>
<td>0x00001c</td>
</tr>
<tr>
<td>0x000020</td>
<td>001010010000000100000000110010100000000</td>
<td>0x000020</td>
</tr>
<tr>
<td>0x000024</td>
<td>101011111010100000000000000111100000000</td>
<td>0x000024</td>
</tr>
<tr>
<td>0x000028</td>
<td>0000000000000000011110000001001000000000</td>
<td>0x000028</td>
</tr>
<tr>
<td>0x00002c</td>
<td>0000001100001111110010000010000100000000</td>
<td>0x00002c</td>
</tr>
<tr>
<td>0x000030</td>
<td>0010011110111101000000000010000000000000</td>
<td>0x000030</td>
</tr>
<tr>
<td>0x000034</td>
<td>00000011111000000000000000001000000000000</td>
<td>0x000034</td>
</tr>
<tr>
<td>0x000038</td>
<td>00000011111000000000000000001000000000000</td>
<td>0x000038</td>
</tr>
<tr>
<td>0x00003c</td>
<td>10101111101111100000000000010100000000000</td>
<td>0x00003c</td>
</tr>
<tr>
<td>0x000040</td>
<td>00000011111000000000000000001000000000000</td>
<td>0x000040</td>
</tr>
<tr>
<td>0x000044</td>
<td>00000011111000000000000000001000000000000</td>
<td>0x000044</td>
</tr>
<tr>
<td>0x000048</td>
<td>00000011111000000000000000001000000000000</td>
<td>0x000048</td>
</tr>
<tr>
<td>0x00004c</td>
<td>00000011111000000000000000001000000000000</td>
<td>0x00004c</td>
</tr>
<tr>
<td>0x000050</td>
<td>00000011111000000000000000001000000000000</td>
<td>0x000050</td>
</tr>
<tr>
<td>0x000054</td>
<td>00000011111000000000000000001000000000000</td>
<td>0x000054</td>
</tr>
<tr>
<td>0x000058</td>
<td>00000011111000000000000000001000000000000</td>
<td>0x000058</td>
</tr>
<tr>
<td>0x00005c</td>
<td>00000011111000000000000000001000000000000</td>
<td>0x00005c</td>
</tr>
</tbody>
</table>
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)

```
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  multu $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, -10
40  lui   $4, 4096
44  ori   $4,$4,1072
48  lw    $5,24($29)
4c  jal   812
50  add   $2, $0, $0
54  lw    $31,20($29)
58  addiu $29,$29,32
5c  jr    $31
```
Link step 3:

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

**NOTE:**
- The preceding example was a much simplified version of how ELF and other standard formats work, meant only to demonstrate the basic principles.
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...
In Conclusion…

- Compiler converts a single HLL file into a single assembly lang. 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.