CS 61C: Great Ideas in Computer Architecture

Running a Program

Instructor: Justin Hsia
Review of Last Lecture

- *Instruction formats* designed to be similar but still capable of handling all instructions

<table>
<thead>
<tr>
<th></th>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>R:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>immediate</td>
</tr>
<tr>
<td>J:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>target address</td>
</tr>
</tbody>
</table>

- Branches move relative to current address, Jumps go directly to a specific address

- Assembly/Disassembly: Use MIPS Green Sheet to convert
Question from Last Lecture

• Do we really need all 32 bits to address the instructions of our code?
  – Almost never
• Although very impractical and unlikely, want to be able to handle extreme cases such as:
Question: Which of the following statements is TRUE?

☐ $rt$ (target register) is misnamed because it never receives the result of an instruction

☐ All of the fields in an instruction are treated as unsigned numbers

☐ We can reach an instruction that is $2^{16} \times 4 = 2^{18}$ bytes away with a branch

☐ We can reach more instructions farther forward than we can backwards with a branch
Great Idea #1: Levels of Representation/Interpretation

Higher-Level Language Program (e.g. C)
  \( \text{Compiler} \)
  Assembly Language Program (e.g. MIPS)
  \( \text{Assembler} \)
  Machine Language Program (MIPS)
  \( \text{Machine Interpretation} \)
  Hardware Architecture Description (e.g. block diagrams)
  \( \text{Architecture Implementation} \)
  Logic Circuit Description (Circuit Schematic Diagrams)

\[ \begin{align*}
\text{temp} &= v[k]; \\
v[k] &= v[k+1]; \\
v[k+1] &= \text{temp}; \\
\text{lw} &= $t0, 0($2) \\
\text{lw} &= $t1, 4($2) \\
\text{sw} &= $t1, 0($2) \\
\text{sw} &= $t0, 4($2)
\end{align*} \]

We are here

0000 1001 1100 0110 1010 1111 0101 1000 1010 1111 0101 1000 0110 1111 0101 1000 0000 1001 0101 0000 1000 0000 1001 0101 1000 0110 1010 1111
Agenda

- Translation vs. Interpretation
- Compiler
- Administrivia
- Assembler
- Linker
- Loader
- Bonus: C.A.L.L. Example
Translation vs. Interpretation (1/3)

• 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

• In general, we interpret a high level language when efficiency is not critical and translate to a lower level language to up performance
Translation vs. Interpretation (2/3)

• Generally easier to write an 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 is slower (10x?), but code is smaller (2x?)
• Interpreter provides instruction set independence: can run on any machine
Translation vs. Interpretation (3/3)

• 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 (e.g. Microsoft keeps all their source code secret)
  – Alternative model, “open source”, creates value by publishing the source code and fostering a community of developers
C Translation

Steps to Starting a Program:

1) Compiler
2) Assembler
3) Linker
4) Loader
C Translation

• **Recall:** A key feature of C is that it allows you to compile files *separately*, later combining them into a single executable

• What can be accessed across files?
  – Functions
  – Static/global variables

• We will cover how the C.A.L.L. process allows us to do this
Compiler

• Input: High-level language (HLL) code (e.g. C, Java in files such as foo.c)

• Output: Assembly Language Code (e.g. foo.s for MIPS)

• Note that the output may contain pseudo-instructions
Compilers Are Non-Trivial

• There’s a whole course about them – CS164
  – We won’t go into further detail in this course

• Some examples of the task’s complexity:
  – Operator precedence: $2 + 3 \times 4$
  – Operator associativity: $a = b = c$
  – Determining locally whether a program is valid
    • if (a) { if (b) { ... /*long distance*/ ... } } //extra bracket
Compiler Optimization

• **gcc compiler options**
  – Level of optimization specified by the flag capital ‘O’ followed by a number (i.e. `–O1`)
  – The default is equivalent to `–O0` (no optimization) and goes up to `–O3` (heavy optimization)

• Trade-off is between compilation speed and output file size/performance

• For more details, see: [http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html](http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html)
Benefit of Compiler Optimization

• Example program shown here: BubbleSort.c

```c
#define ARRAY_SIZE 20000
int main() {
    int iarray[ARRAY_SIZE], x, y, holder;
    for(x = 0; x < ARRAY_SIZE; x++)
        for(y = 0; y < ARRAY_SIZE-1; y++)
            if(iarray[y] > iarray[y+1]) {
                holder = iarray[y+1];
                iarray[y+1] = iarray[y];
                iarray[y] = holder;
            }
}
```
Unoptimized MIPS Code

$L3:
lw $2,80016($sp)
slt $3,$2,20000
bne $3,$0,$L6
j $L4
$L6:
.set noreorder
nop
.set reorder
sw $0,80020($sp)
$L7:
lw $2,80020($sp)
slt $3,$2,19999
bne $3,$0,$L10
j $L5
$L10:
lw $2,80020($sp)
mov $3,$2
sll $2,$3,2
addu $3,$sp,16
sw $3,80024($sp)

$L11:
lw $2,80020($sp)
addu $2,$3,1
move $3,$2
sll $3,$4,2
addu $4,$sp,16
addu $3,$4,$3
lw $2,0($2)
lw $3,0($3)
slt $2,$3,$2
beq $2,$0,$L9
lw $3,80020($sp)
addu $2,$3,1
move $3,$2
sll $2,$3,2
addu $3,$sp,16
lw $3,80020($sp)
move $3,$2
sll $2,$3,2
addu $3,$sp,16
lw $3,80024($sp)
sw $3,0($2)

$L9:
lw $2,80020($sp)
addu $3,$2,1
sw $3,80020($sp)
j $L7
$L8:
lw $2,80016($sp)
addu $3,$2,1
sw $3,80016($sp)
j $L3
$L4:
li $12,65536
ori $12,$12,0x38b0
addu $13,$12,$sp
addu $sp,$sp,$12
j $31

$L5:
lw $2,80020($sp)
addu $3,$2,1
sw $3,80020($sp)
j $L3
$2:
li $12,65536
ori $12,$12,0x38b0
addu $13,$12,$sp
addu $sp,$sp,$12
j $31
-02 Optimized MIPS Code

li $13,65536
ori $13,$13,0x3890
addu $13,$13,$sp
sw $28,0($13)
move $4,$0
addu $8,$sp,16

$L6:
move $3,$0
addu $9,$4,1
.p2align 3

$L10:
slt $2,$4,$3
beq $2,$0,$L9
sw $3,0($5)
sw $4,0($6)

$L9:
move $3,$7
slt $2,$3,19999
bne $2,$0,$L10
move $4,$9
slt $2,$4,20000
bne $2,$0,$L6
li $12,65536
ori $12,$12,0x38a0
addu $13,$12,$sp
addu $sp,$sp,$12
j $31

6/28/2012 Summer 2012 -- Lecture #8
Administrivia

• End of the second week!
  – The majority of the programming portion of this class is done! (other than SIMD/OpenMP)
• HW 2 due Sunday
  – Remove extraneous output before you submit!
• Project 1 released tonight, due next Sunday
• Justin’s OH are normal this week
• Normal discussion on Wed next week (7/4) moved to Lab on Thurs (7/5)
Agenda

- Translation
- Compiler
- Administrivia
- Assembler
- Linker
- Loader
- Bonus
Assembler

• **Input:** Assembly language code (MAL) (e.g. `foo.s` for MIPS)

• **Output:** Object code (TAL), information tables (e.g. `foo.o` for MIPS)
  – Object file

• Reads and uses **directives**

• Replaces pseudo-instructions

• Produces machine language
Assembler Directives
(For more info, see p.B-5 and B-7 in P&H)

- 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-terminates it
  - `.word w1...wn`: Store the `n` 32-bit quantities in successive memory words
Pseudo-instruction Replacement

**Pseudo:**

- subu $sp,$sp,32
- sd $a0, 32($sp)
- mul $t7,$t6,$t5
- addu $t0,$t6,1
- la $a0, str

**Real:**

- addiu $sp,$sp,-32
- sw $a0, 32($sp)
- sw $a1, 36($sp)
- mul $t6,$t5
- mflo $t7
- addiu $t0,$t6,1
- slti $at,$t0,101
- bne $at,$0,loop
- lui $at,left(str)
- ori $a0,$at,right(str)
Producing Machine Language (1/3)

- **Simple Cases**
  - Arithmetic and logical instructions, shifts, etc.
  - All necessary info contained in the instruction

- **What about Branches?**
  - Branches require a *relative address*
  - Once pseudo-instructions are replaced by real ones, we know by how many instructions to branch

- **We’re good so far...**
• “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
```

– Solution: Make two passes over the program
  • First pass remembers position of labels
  • Second pass uses label positions to generate code
What about jumps (\texttt{j} and \texttt{jal})?

– Jumps require \textit{absolute address}
– Forward or not, can’t generate machine instruction without knowing the position of instructions in memory

What about references to data?

– \texttt{lui} gets broken up into \texttt{lui} and \texttt{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” that may be used by other files
  – *Each* file has its own symbol table

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

• List of “items” this file will need the address of later (currently undetermined)

• What are they?
  – Any label jumped to: j or jal
    • internal
    • external (including lib files)
  – Any piece of data
    • such as anything referenced by the la instruction

• Where do you suppose we get this info from?
Object File Format

- **object file header**: size and position of the other pieces of the object file
- **text segment**: the machine code
- **data segment**: data in the source file (binary)
- **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**
- **A standard format is ELF (except MS)**

http://www.skyfree.org/linux/references/ELF_Format.pdf
Agenda

- Translation
- Compiler
- Administrivia
- Assembler
- Linker
- Loader
- Bonus
Linker (1/3)

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

object file 1
- text 1
- data 1
- info 1

object file 2
- text 2
- data 2
- info 2

Linker

a.out
- Relocated text 1
- Relocated text 2
- Relocated data 1
- Relocated data 2
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 (\texttt{beq, bne})
  – Never relocate

• Absolute Address (\texttt{j, jal})
  – Always relocate

• External Reference (usually \texttt{jal})
  – Always relocate

• Data Reference (often \texttt{lui and ori})
  – Always relocate
Absolute Addresses in MIPS

• Which instructions need relocation editing?
  – J-format: jump, jump and link
    
    \[
    \begin{array}{c|c}
    \text{j/jal} & \text{xxxxxx} \\
    \end{array}
    \]
  
  – Loads and stores to variables in static area, relative to global pointer ($gp$)
    
    \[
    \begin{array}{c|c|c|c}
    \text{lw/sw} & \text{$gp$} & \text{$xx$} & \text{address} \\
    \end{array}
    \]
  
  – What about conditional branches?
    
    \[
    \begin{array}{c|c|c|c}
    \text{beq/bne} & \text{$rs$} & \text{$rt$} & \text{address} \\
    \end{array}
    \]

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

• Linker assumes the first word of the 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:
  
  1) Search for reference (data or label) in all “user” symbol tables
  2) If not found, search library files (e.g. printf)
  3) Once absolute address is determined, fill in the machine code appropriately

• Output of linker: executable file containing text and data (plus header)
Static vs. Dynamically Linked Libraries

• What we’ve described is the traditional way: **statically-linked libraries**
  – 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 the source files)
  – It includes the *entire* library even if not all of it will be used
  – Executable is self-contained (all you need to run)

• An alternative is **dynamically linked libraries** (DLL), common on Windows & UNIX platforms
Dynamically Linked Libraries (1/2)

• 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 upgrades every program that uses that library
    – Having the executable isn’t enough anymore
Dynamically Linked Libraries (2/2)

• 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

Get To Know Your Instructor
Agenda

- Translation
- Compiler
- Administrivia
- Assembler
- Linker
- Loader
- Bonus
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?

1) Reads executable file’s header to determine size of text and data segments

2) Creates new address space for program large enough to hold text and data segments, along with a stack segment

[more on this later in CS61C]

3) Copies instructions and data from executable file into the new address space
Loader ... what does it do?

4) Copies arguments passed to the program onto the stack

5) Initializes machine registers
   – Most registers cleared, but stack pointer assigned address of 1st free stack location

6) Jumps to start-up routine that copies program’s arguments from stack to registers and sets the PC
   – If main routine returns, start-up routine terminates program with the exit system call
Question: Which statement is TRUE about the following code?

```
la   $t0,Array
Loop: lw   $t1,0($t0)
      addi $t0,$t0,4
      bne  $a0,$t1,Loop
Exit: nop
```

- The `la` instruction will be edited during the link phase
- The `bne` instruction will be edited during the link phase
- Assembler will ignore the instruction `Exit: nop` because it does nothing
- This was written by a programmer because compilers don’t allow pseudo-instructions
Summary

• **Compiler** converts a single HLL file into a single assembly file \( .c \rightarrow .s \)

• **Assembler** removes pseudo-instructions, converts what it can to machine language, and creates a checklist for linker (relocation table) \( .s \rightarrow .o \)
  – Resolves addresses by making 2 passes (for internal forward references)

• **Linker** combines several object files and resolves absolute addresses \( .o \rightarrow .out \)
  – Enable separate compilation and use of libraries

• **Loader** loads executable into memory and begins execution

6/28/2012
You are responsible for the material contained on the following slides, though we may not have enough time to get to them in lecture. They have been prepared in a way that should be easily readable and the material will be touched upon in the following lecture.
Agenda

• Translation
• Compiler
• Administrivia
• Assembler
• Linker
• Loader

• Bonus: C.A.L.L. Example
C Program Source Code \textit{(prog.c)}

```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);
}
```

\textit{printf} \textit{lives in stdio.h}
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"
```

Identify the 7 pseudo-instructions!
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"
```
### Assembly

#### 1) Remove pseudoinstructions, assign addresses

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

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>0x0000000018</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. 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>
Assembly

3) Resolve local PC-relative labels

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, l.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

4) Generate object (.o) file:
   – Output binary representation for:
     • text segment (instructions)
     • data segment (data)
     • symbol and relocation tables
   – Using dummy “placeholders” for unresolved absolute and external references
     • Use all zeros where immediate or target address should be (see next slide)
## Text Segment in Object File

<table>
<thead>
<tr>
<th>Address</th>
<th>Binary String</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x000000</td>
<td>01001111101111011111111111111111000000</td>
</tr>
<tr>
<td>0x000004</td>
<td>10101111101111111111111111111111000000</td>
</tr>
<tr>
<td>0x000008</td>
<td>1010111110110010000000000000001000000</td>
</tr>
<tr>
<td>0x00000c</td>
<td>1010111110110010000000000000001001000</td>
</tr>
<tr>
<td>0x000010</td>
<td>1010111110110000000000000000001100000</td>
</tr>
<tr>
<td>0x000014</td>
<td>1010111110110000000000000000001110000</td>
</tr>
<tr>
<td>0x000018</td>
<td>100011111011011100000000000001110000</td>
</tr>
<tr>
<td>0x00001c</td>
<td>100011111011011100000000000001110000</td>
</tr>
<tr>
<td>0x000020</td>
<td>000000011100011100000000000001100100</td>
</tr>
<tr>
<td>0x000024</td>
<td>001001011101000000000000000001000000</td>
</tr>
<tr>
<td>0x000028</td>
<td>001001000100000000000000000001101000</td>
</tr>
<tr>
<td>0x00002c</td>
<td>101011111011010000000000000001110100</td>
</tr>
<tr>
<td>0x000030</td>
<td>000000000111001110000000000001001000</td>
</tr>
<tr>
<td>0x000034</td>
<td>000000110000111111100010000001000011</td>
</tr>
<tr>
<td>0x000038</td>
<td>0001010000100001111111111111111111</td>
</tr>
<tr>
<td>0x00003c</td>
<td>101011111011111011101000000001110000</td>
</tr>
<tr>
<td>0x000040</td>
<td>00111100001000000000000000000000000000</td>
</tr>
<tr>
<td>0x000044</td>
<td>100011111011011100000000000000000000</td>
</tr>
<tr>
<td>0x000048</td>
<td>00011000001000000000000000000011101000</td>
</tr>
<tr>
<td>0x00004c</td>
<td>00100100000000000000000000000000000000</td>
</tr>
<tr>
<td>0x000050</td>
<td>100011111011111011100000000001010000</td>
</tr>
<tr>
<td>0x000054</td>
<td>001001111011110100000000000001000000</td>
</tr>
<tr>
<td>0x000058</td>
<td>000000111110000000000000000000010000</td>
</tr>
<tr>
<td>0x00005c</td>
<td>00000000000000000000000000000011000011</td>
</tr>
</tbody>
</table>

The binary strings represent memory addresses and their corresponding values. Notably, the strings corresponding to `l.str`, `r.str`, and `printf` are highlighted.
1) Combine `prog.o` and `libc.o`
- Merge text/data segments
- Create absolute memory addresses
- Modify & merge symbol and relocation tables
- Symbol Table
  - Label               Address
    main:       0x00000000
    loop:       0x00000018
    str:        0x10000430
    printf:     0x00000cb0
- Relocation Information
  - Address   Instr. Type   Dependency
    0x00000040  lui          l.str
    0x00000044  ori          r.str
    0x0000004c  jal          printf
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

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