CS 61c: Great Ideas in Computer Architecture

C.A.L.L.

Instructor: Alan Christopher

July 7, 2014

Review

- Three different instruction formats designed to be as similar as possible, while still handling all instructions:

R:	opcode	rs	rt	rd	shamt	funct
I:	opcode	rs	rt	immediate		
J:	opcode	jump address				

- Branches move relative to the PC, jumps go to a specific address
- Assembly/Disassembly: Use MIPS Green Sheet to convert

Question: Which of the following statements is **TRUE**

(blue) `$rt` is misnamed because it never receives the result of an instruction

(green) All of the fields in all instructions are treated as unsigned numbers

(purple) We can reach an instruction that is 2^{18} bytes away with a branch

(yellow) We can reach more instructions forward than we can backwards with a branch

Question: Which of the following statements is **TRUE**

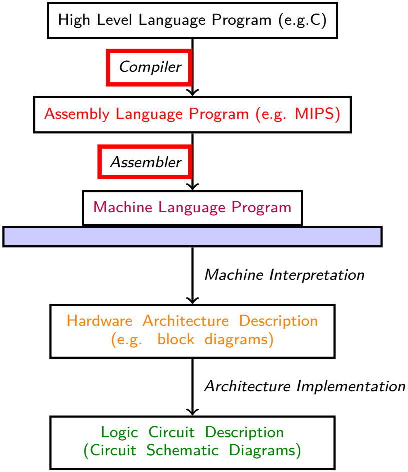
(blue) `$rt` is misnamed because it never receives the result of an instruction

(green) All of the fields in all instructions are treated as unsigned numbers

(purple) We can reach an instruction that is 2^{18} bytes away with a branch

(yellow) We can reach more instructions forward than we can backwards with a branch

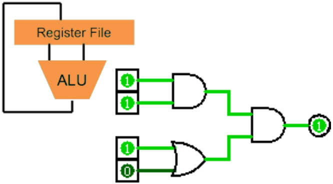
Great Idea #1: Levels of Representation/Interpretation




```
temp = v[k];
v[k] = v[k+1]
v[k+1] = temp;
```

```
lw $t0, 0($2)
lw $t1, 4($2)
sw $t1, 0($2)
sw $t0, 4($2)
```

```
1000 1100 0100 1000 0000 0000 0000 0000
1000 1100 0100 1001 0000 0000 0000 0100
1010 1100 0100 1001 0000 0000 0000 0000
1010 1100 0100 1000 0000 0000 0000 0100
```



Outline

Translation vs. Interpretation I

- How do we run a program written in a source language?

Translation vs. Interpretation I

- How do we run a program written in a source language?
  - Interpreter: Directly execute a program in the source language

Translation vs. Interpretation I

- How do we run a program written in a source language?
  - Interpreter: Directly execute 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* a high level language when performance is critical
- Can also use lower-level language to begin with

Translation vs. Interpretation II

- Generally easier to write an interpreter
- Interpreter closer to high-level, so can give better error messages more easily
- Interpereter is slower ($\approx$10x), but code is smaller ($\approx$2x)
- Interpreter provides instruction set independence: can run on any machine
  - Still need an interpreter for the machine, of course

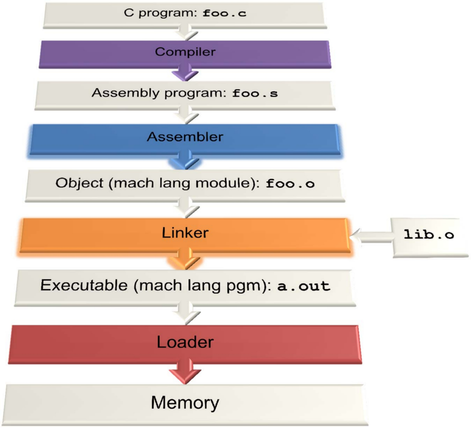
Translation vs. Interpretation III

- Translated/compiled code almost always more efficient/higher performance
  - Important for many applications, particularly OSs and real time systems
- Translation/compilation help to "hide" the source code form users
  - Can be used to protect intellectual property (e.g. many users run Microsoft OSs, but the source code is carefully controlled)
  - Alternative model, *free software* (sometimes called *open source*), publishes source code in order to foster a community of developers, among other things

C Translation

Steps to starting a program:

1. Compilation
2. Assembly
3. Linking
4. Loading



C Translation

- **Recall:** A key feature of C is that it allows you to compile files *separately*, later combining them into a single executable
  - Helps with code factoring
  - Reduces compilation times
- What can be accessed across files?
  - Functions
  - Static/global variables

Compiler

- **Input:** Higher level language (HLL) code
  - e.g. C or java files
  - e.g. `foo.c` or `foo.h`
- **Output:** Assembly Language code (e.g. `foo.s` for MIPS)

- Output may contain pseudo-instructions
  - We'll deal with those inside the assembler

Compilers are Non-Trivial

- There's a whole (fantastic) course about them – CS164
  - Project 1 is really just a taste of the topic
- Some examples of the task's complexity:
  - Operator precedence: 2 + 3 * 4
  - Operator associativity: a = b = c;
  - Static analysis of program validity:
    - ```
      if(a){if(b){.../*Lots of junk */...}}}//extra bracket
      ```
    - ```
      struct companion *cube;
      ... /* Lots of junk */ ...
      x = cube->cake; // companion cube has no cake
      ```

Compiler Optimization

- Almost all compilers are what's called an *optimizing compiler* – it tries to produce correct code that's fast too
- gcc provides different options for level of optimization
  - Level of optimization specified by the 'O#' flags (e.g. `-O1`)
  - The default is equivalent to `-O0` (almost no optimization) and goes up to `-O3` (throw every optimization in the book at the problem, whether it makes sense or not)
- Trade-off is between compilation speed and output file size/performance
  - Infrequently (*very* infrequently) optimizations will result in bugs that don't occur in non-optimized code
- For more details on gcc optimization options, see: `http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`

Benefits of Compiler Optimization

- Example program here: BubbleSort.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)
  move    $3,$2
  sll $2,$3,2
  addu $3,$sp,16
  addu $2,$3,$2
  lw $4,80020($sp)
  addu $3,$4,1
  move    $4,$3
  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
  addu $2,$3,$2
  lw $3,0($2)
  sw $3,80024($sp)
  lw $3,80020($sp)
  addu $2,$3,1
  move    $3,$2
  sll $2,$3,2
  addu $3,$sp,16
  addu $2,$3,$2
  lw $3,80020($sp)
  move    $4,$3
  sll $3,$4,2
  addu $4,$sp,16
  addu $3,$4,$3
  lw $4,0($3)
  sw $4,0($2)
  lw $2,80020($sp)
  move    $3,$2
  sll $2,$3,2
  addu $3,$sp,16
  addu $2,$3,$2
  lw $3,80024($sp)
  sw $3,0($2)
$L11:
$L9:
  lw $2,80020($sp)
  addu $3,$2,1
  sw $3,80020($sp)
  j $L7
$L8:
$L5:
  lw $2,80016($sp)
  addu $3,$2,1
  sw $3,80016($sp)
  j $L3
$L4:
$L2:
  li $12,65536
  ori $12,$12,0x38b0
  addu $13,$12,$sp
  addu $sp,$sp,$12
  j $31
```

-O2 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:
    sll $2,$3,2
    addu $6,$8,$2
    addu $7,$3,1
    sll $2,$7,2
    addu $5,$8,$2
    lw $3,0($6)
    lw $4,0($5)
    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
```

Outline

Administrivia

- We're into the 3rd week (25% done)
  - Pretty much done talking about programming for programming's sake
  - Midterm in only 2 weeks
- Project 1 due Sunday
  - Should have lexer, parser mostly finished
  - Don't forget to write tests
  - Expect to spend half or more of your time debugging (as with most any CS project)

Outline

The Assembler

- **Input:** Assembly language code (MAL)
- **Output:** Object code (TAL), information tables
  - Called an *object file* (e.g. `foo.o`)
- Reads and uses *directives*
- Translates pseudo-instructions
- Produces machine language

Assembler Directives[1]

- Give directions ot the 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` $w_1 \ldots w_n$: Store the n 32-bit quantities in successive memory words

[1] More info available in P&H appendices

Pseudo-instruction Replacement

Pseudo:

```
subu $sp, $sp, 32
sd $a0, 32($sp)

mul $t7, $t6, $t5

addu $t0, $t6,1
ble  $t0, 100, loop

la $a0, str
```

Real:

```
addiu $sp,$sp,-32
sw $a0, 32($sp)
sw $a1, 36($sp)
mult $t6,$t5
mflo $t7
addiu $t0,$t6,1
slti $at,$t0,101
bne $at,$0,loop
lui $at,%hi(str)
ori $a0,$at,%lo(str)
```

Producing Machine Languge I

- Simple cases
  - Arithmetic and logical instructions, shifts, etc.
  - All necessary info is contained in the instruction
- What about Branches?

Producing Machine Languge I

- Simple cases
  - Arithmetic and logical instructions, shifts, etc.
  - All necessary info is contained in the instruction
- What about Branches?
  - Branches require a *relative address*

Producing Machine Languge I

- Simple cases
  - Arithmetic and logical instructions, shifts, etc.
  - All necessary info is contained in the instruction
- What about Branches?
  - Branches require a *relative address*
  - Once pseudo-instructions replaced by real ones, we know by how many instructions to branch, so no problem

Producing Machine Language II

- "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
```

Producing Machine Language II

- "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

Producing Machine Language III

- What about jumps (`j` and `jal`)?
  - Jumps require *absolute address* of instructions
  - Forward or not, can't generate machine instruction without know 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

Producing Machine Language III

- What about jumps (`j` and `jal`)?
  - Jumps require *absolute address* of instructions
  - Forward or not, can't generate machine instruction without know 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" that may be used by other files
  - *Every* file has its own symbol table
- What are these "items"?

Symbol Table

- List of "items" that may be used by other files
  - *Every* file has its own symbol table
- What are these "items"?
  - **Labels**: for calling functions

Symbol Table

- List of "items" that may be used by other files
  - *Every* file has its own symbol table
- What are these "items"?
  - **Labels**: for calling functions
  - **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 these “items”?

Relocation Table

- List of "items" this file will need the address of later (currently undetermined)
- What are these "items"?
  - Any **label** jumped to:
    - internal (why?)
    - external (including library files)

Relocation Table

- List of "items" this file will need the address of later (currently undetermined)
- What are these "items"?
  - Any **label** jumped to:
    - internal (why?)
    - external (including library files)
  - Any piece of **data**
    - such as anything referenced by the **la** instruction

Object File Format

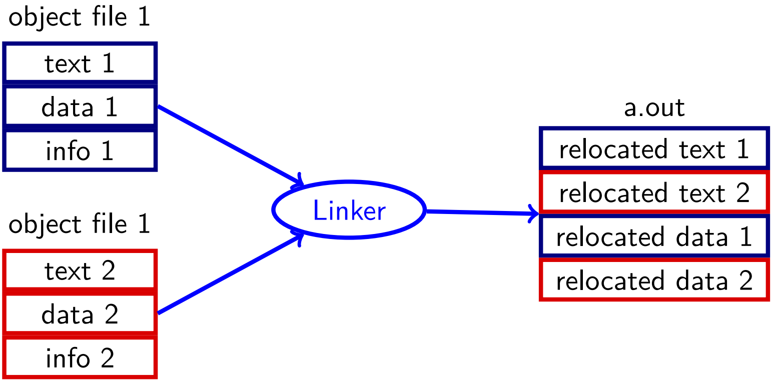
1. **object file header:** size and position of other pieces of the object file
2. **text segment**: the machine code
3. **data segment**: data in the source file (binary)
4. **relocation table**: identifies lines of code that need "handling"
5. **symbol table**: list of this file's labels and data that can be referenced
6. **debugging information**: information to make tools like `gdb` more effective

- A standard format is ELF `http://www.skyfree.org/linux/references/ELF_Format.pdf`

Linker I

- **Input:** Object files, information tables (e.g. `foo.o`)
- **Output:** Executable code (e.g. `a.out`)
- 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 II


object file 1
text 1
data 1
info 1
object file 1
text 2
data 2
info 2
Linker
a.out
relocated text 1
relocated text 2
relocated data 1
relocated data 2

Linker III

1. Take text segment from each `.o` file and put them together
2. Take data segment from each `.o` file and put them together, and concatenate this onto end of text segments
3. Resolve references
   - Go through relocation table; resolve each entry
   - I.e. fill in all absolute addresses

Four Types of Addresses

- PC-Relative (`beq`, `bne`)
  - Never relocate

Four Types of Addresses

- PC-Relative (`beq`, `bne`)
  - Never relocate
- Absolute (`j`, `jal`)
  - Always relocate

Four Types of Addresses

- PC-Relative (`beq`, `bne`)
  - Never relocate
- Absolute (`j`, `jal`)
  - Always relocate
- External Reference (usually `jal`)
  - Always relocate

Four Types of Addresses

- PC-Relative (`beq`, `bne`)
  - Never relocate
- Absolute (`j`, `jal`)
  - Always relocate
- External Reference (usually `jal`)
  - Always relocate
- Data Reference (often `lui` and `ori`)
  - Always relocate

Absolute Addresses in MIPS

- Which instructions need editing during relocation?
  - `j/jal`: Use (pseudo)absolute address, need to know position of code before filling in address

Absolute Addresses in MIPS

- Which instructions need editing during relocation?
  - `j/jal`: Use (pseudo)absolute address, need to know position of code before filling in address
  - `lui/ori`: If part of a `la` instruction, then need to know what address the label refers to is

Absolute Addresses in MIPS

- Which instructions need editing during relocation?
  - `j/jal`: Use (pseudo)absolute address, need to know position of code before filling in address
  - `lui/ori`: If part of a `la` instruction, then need to know what address the label refers to is
  - `beq/bne`: Do **NOT** need to modify – branches are PC-relative, and linking doesn't change the relative position of lines of code in a source file

Resolving References I

- Linker assumes the first word of the first text segment is at address 0x00000000
  - But how do we run multiple programs?
- 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 I

- Linker assumes the first word of the first text segment is at address 0x00000000
  - But how do we run multiple programs?
  - Virtual memory! (Covered later)
- 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 II

- 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 code*
  - All referenced code is part of the executable, so if a library updates, we don't get the fix (until we recompile)
  - It includes the *entire* library, even if only a small part of it is used
  - Executable is self-contained
- An alternative is *dynamic linking*, or *dynamically linked libraries* (DLL), common on both Windows and UNIX-like platforms

Dynamic Linking I

- 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 the link
- Upgrades
  - \+ Replacing one file upgrades every program that uses a library
  - \- Having the executable isn't enough anymore

Dynamic Linking II

- Overall, dynamic linking adds quite a bit of complexity to the compiler, linker, and the OS
- However, it provides many benefits that often outweigh the added complexity
- For more info, see `http://en.wikipedia.org/wiki/Dynamic_linking`

Technology Break

Loader Basics

- **Input:** Executable code (e.g. `a.out`)
- **Output:** <program is run>

- Executable files are stored on disk
- When program is run, loader's job is to load it into memory and start it running
- In practice, the loader is done by the OS

What the Loader Does

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
   - This is more of that virtual memory business
3. Copies instructions and data from executable file into the new address space

What the Loader Does

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   $1, 0($t0)
      addi $t0, $t0, 4
      bne  $a0, $t1, Loop
Exit: nop
```

(blue) the `la` instruction will be edited during the linking phase
(green) The `bne` instruction will be edited during the linking phase
(purple) The assembler will ignore the instruction `Exit:nop`, because it does nothing
(yellow) This was written by a human, since compilers don't generate pseudo-instructions

Question: Which statement is **TRUE** about the following code?

```
      la   $t0, Array
Loop: lw   $1, 0($t0)
      addi $t0, $t0, 4
      bne  $a0, $t1, Loop
Exit: nop
```

(blue) the `la` instruction will be edited during the linking phase
(green) The `bne` instruction will be edited during the linking phase
(purple) The assembler will ignore the instruction `Exit:nop`, because it does nothing
(yellow) This was written by a human, since compilers don't generate pseudo-instructions

CALL Example

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 += 1)
    sum = sum + i * i;
  /* Recall: printf declared in stdio.h */
  printf("sum of sq from 0-100 = %d\n", sum;)
}
```

Compilation: MAL

Identify the 7 pseudo-instructions!

```
.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"
```

Compilation: MAL

Identify the 7 pseudo-instructions!

```
.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 pseudo instructions, 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   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, loop
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

2. Create relocation table and symbol table

- Symbol table

Label	Address (in module)	Type
main:	0x00000000	global text
loop:	0x00000018	local text
str:	0x00000000	local data

- Relocation table

Address	Inst. Type	Dependency
0x00000040	lui	l.str
0x00000044	ori	r.str
0x0000004c	jal	printf

Assembly

3. Resolve local PC-relative labels

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

Assembly

4. Generate object file:

- Output binary representation for
  - text segment
  - data segment
  - symbol and relocation tables
- Using dummy "placeholders" for unresolved absolute and external references
  - Use all zeroes where immediate or target address should be

Text Segment in Object File

```
0x000000    00100111101111011111111111100000
0x000004    10101111101111110000000000010100
0x000008    10101111101001000000000000100000
0x00000c    10101111101001010000000000100100
0x000010    10101111101000000000000000011000
0x000014    10101111101000000000000000011100
0x000018    10001111101011100000000000011100
0x00001c    10001111101110000000000000011000
0x000020    00000001110011100000000000011001
0x000024    00100101110010000000000000000001
0x000028    00101001000000010000000001100101
0x00002c    10101111101010000000000000011100
0x000030    00000000000000001111000000010010
0x000034    00000011000011111100100000100001
0x000038    00010100001000001111111111110111
0x00003c    10101111101110010000000000011000
0x000040    00111100000001000000000000000000 ← l.str
0x000044    10001111101001010000000000000000 ← r.str
0x000048    00001100000100000000000011101100
0x00004c    00100100000000000000000000000000 ← printf
0x000050    10001111101111110000000000010100
0x000054    00100111101111010000000000100000
0x000058    00000011111000000000000000001000
0x00005c    00000000000000000001000000100001
```

Link

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 table

Address	Inst. Type	Dependency
0x00000040	`lui`	`l.str`
0x00000044	`ori`	`r.str`
0x0000004c	`jal`	`printf`

Link

2. Edit addresses in relocation table (shown in TAL for legibility, actually done in binary)

```
00   addiu $29, $29, -32       30   addiu $8,  $14, 1
04   sw    $31, 20($29)        34   sw    $8,  28($29)
08   sw    $4,  32($29)        38   slti  $1,  $8, 101
0c   sw    $5,  36($29)        3c   bne   $1,  $0, -10
10   sw    $0,  24($29)        40   lui   $4,  4096
14   sw    $0,  28($29)        44   ori   $4,  $4, 1072
18   lw    $14, 28($29)        48   lw    $5,  24($29)
1c   multu $14, $14            4c   jal   812
20   mflo  $15                 50   add   $2,  $0, $0
24   lw    $24, 24($29)        54   lw    $31, 20($29)
28   addu  $25, $24, $15       58   addiu $29, $29, 32
2c   sw    $25, 24($29)        5c   jr    $31
```

Link

3. **Output executable of merged modules**
    - Single text segment
    - Single data segment
    - Header detailing size of each segment

NOTE: This example was a very simplified version of how ELF and other standard formats work, intended only to demonstrate the basic principles of C.A.L.L.

Outline

Summary

- **Compiler** converts a single HLL file into a single assembly file
- **Assembler** removes pseudo-instructions, converts what it can into machine language, and creates a checklist for linker (relocation table)
  - Resolves addresses by making 2 passes (for forward references)
- **Linker** combines several object files and resolves absolute addresses
  - Enable separate compilation and use of libraries
- **Loader** loads executable into memory and begins execution