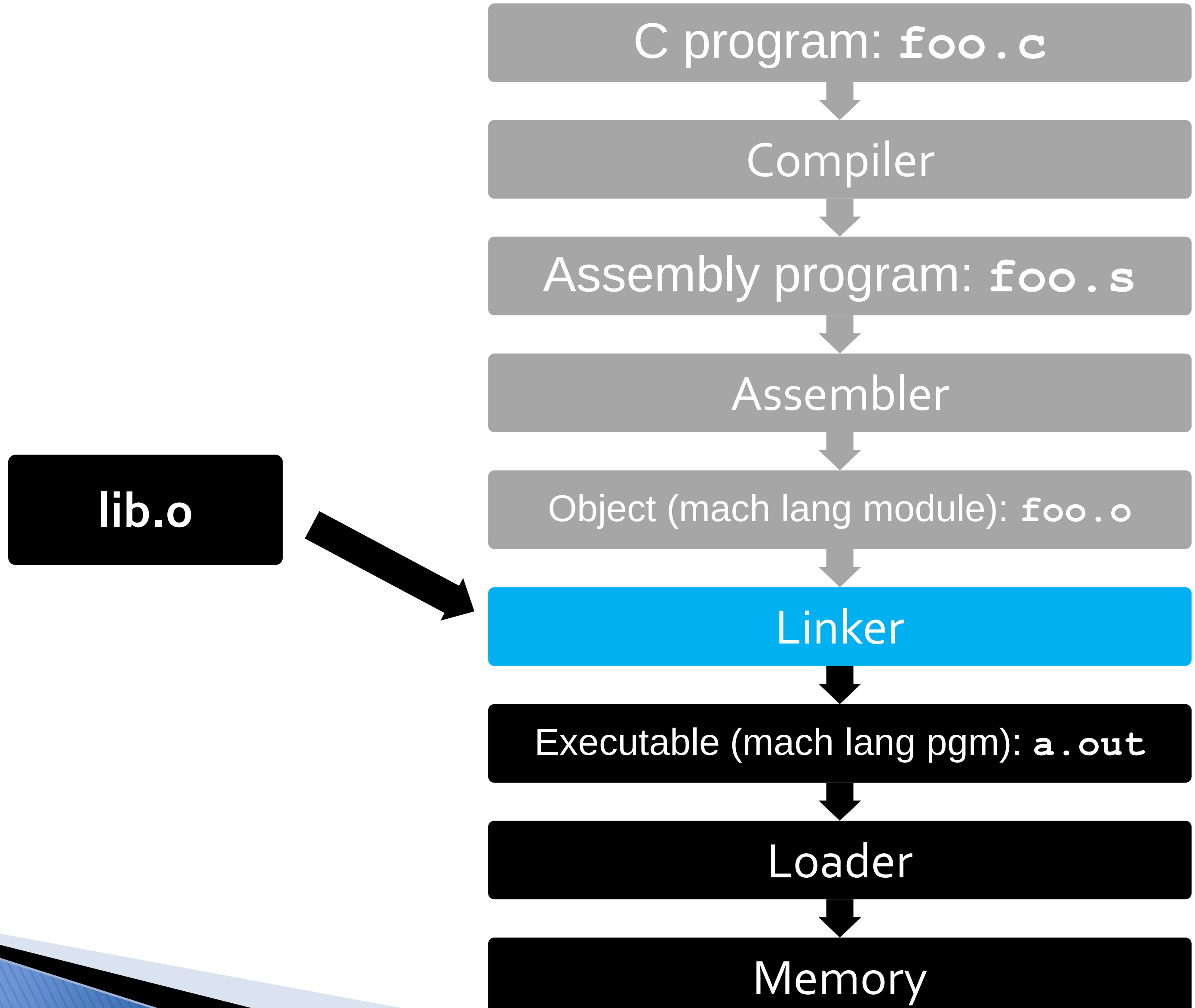
# CSE 31
# Computer Organization

## Lecture 16 – Program Process (2)

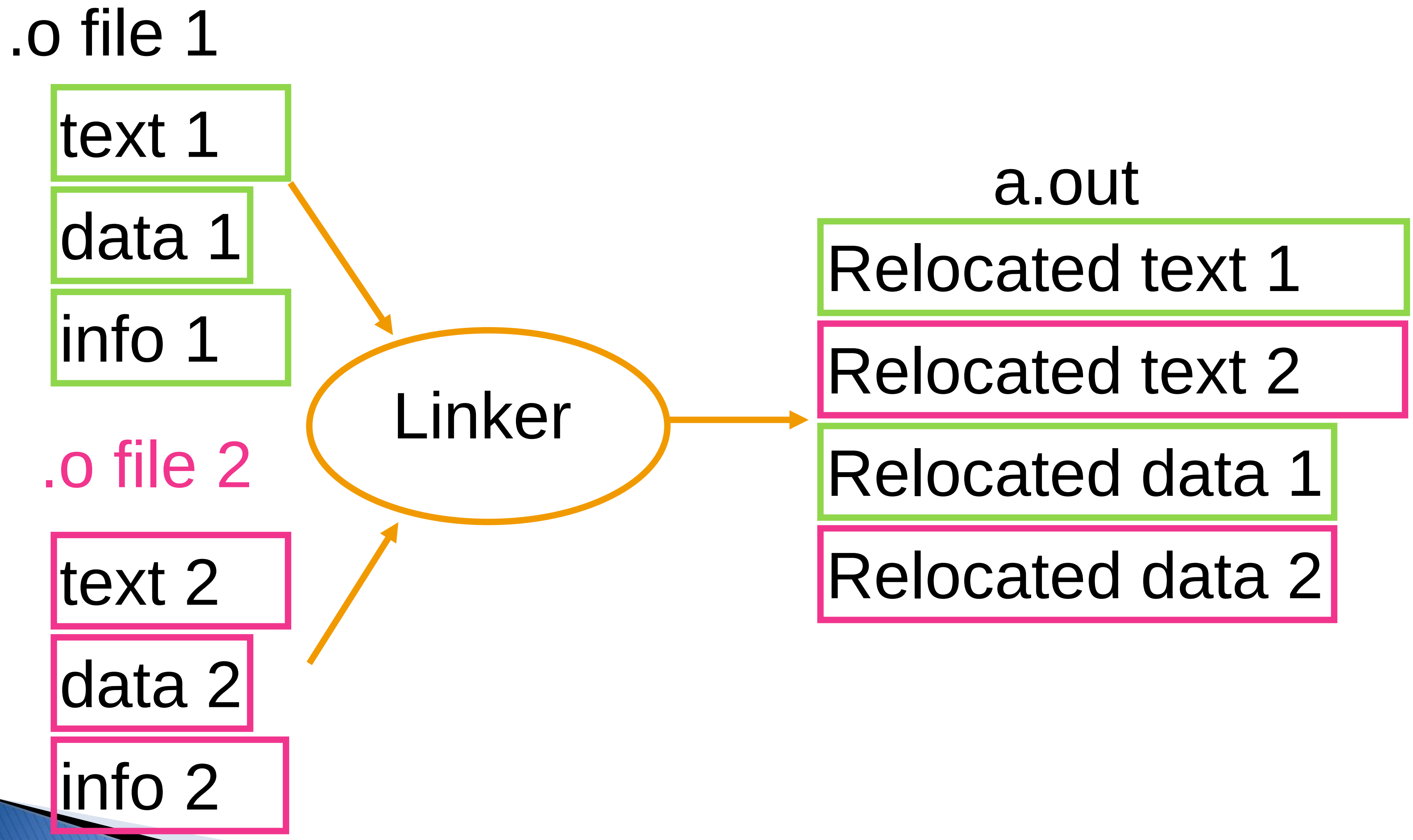
# Announcement

- Lab #7 this week
  - Due in one week
- HW #5 (from zyBooks) out this Friday
  - Due Monday (11/5)
- Project #1
  - Due Monday (10/22)
    - Don’t start late, you won’t have time!
- Reading assignment
  - Chapter 1.6, 6.1 – 6.7 of zyBooks (Reading Assignment #5)
    - Make sure to do the Participation Activities
    - Due Monday (10/29)

# Steps to Starting a Program (translation)



# Linker


.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

# Four Types of Addresses

- PC-Relative Addressing (`beq`, `bne`)
  - never relocate
- Absolute Address (`j`, `jal`)
  - always relocate
- External Reference (usually `jal`)
  - always relocate
- Data Reference (often `lui` and `ori`/`load address`)
  - always relocate

# Resolving References (1/2)

- Linker assumes first word of first text segment is at address `0x00000000`.

- 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)

# 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

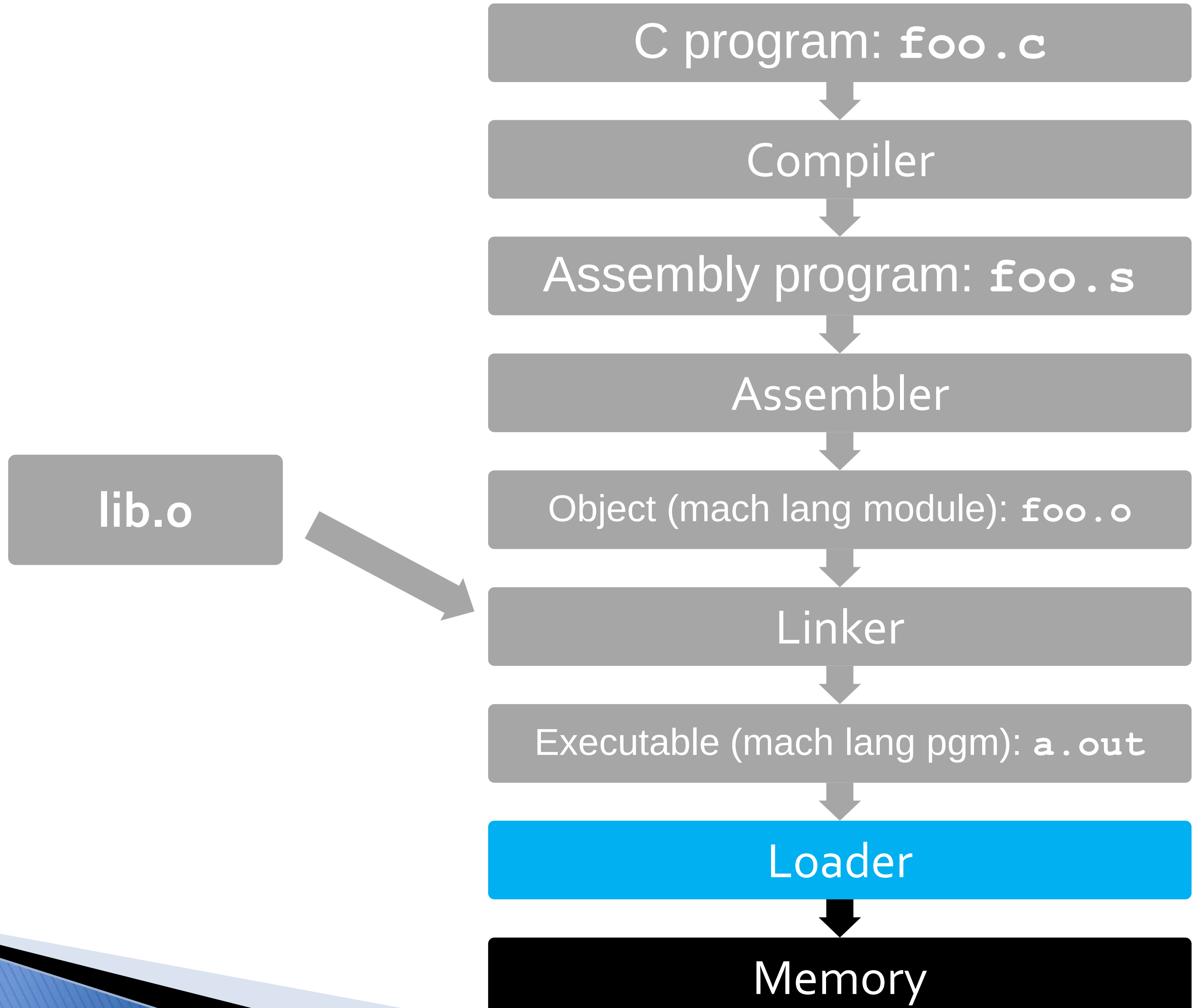
Overall, dynamic linking adds quite a bit of complexity to the compiler, linker, and operating system. However, it provides many benefits that often outweigh these.

en.wikipedia.org/wiki/Dynamic_linking

# 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"
  - There are other ways to achieve the same purpose

# Steps to Starting a Program (translation)



# Loader (1/3)

- 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 running.
- In reality, loader is the operating system (OS)
  - loading is one of the OS tasks

# Loader (2/3)

- So what does a loader do?
  - Reads executable file's header to determine size of text and data segments
  - Creates new address space (static memory) 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

# Loader (3/3)

- Copies arguments passed to the program onto the stack
- Initializes machine registers
  - Most registers cleared
  - 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

# Example: C ⇒ Asm ⇒ Obj ⇒ Exe ⇒ Run

*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

```
__.text
   .align  2
   .globl  main
main:
   subu $sp,$sp,40
   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,40
   jr $ra
   .data
   .align  0
str:
   .asciiz "The sum
   of sq from 0 ..
   100 is %d\n"
```

Where are 7 pseudo-instructions?

# Compilation: MAL

```
    .text
    .align  2
    .globl  main
main:
    subu $sp,$sp,40
    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,40
    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,-40
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,$29,40
5c jr       $31
```

# Assembly step 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 Information

| Address | Instr. type | Dependency |
|---|---|---|
| 0x00000040 | lui | l.str |
| 0x00000044 | ori | r.str |
| 0x0000004c | jal | printf |

# Assembly step 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 step 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.

# Text segment in object file

| | |
|---|---|
| 0x000000 | 00100111101111011111111111101000 |
| 0x000004 | 10101111101111110000000000010100 |
| 0x000008 | 10101111101001000000000000100000 |
| 0x00000c | 10101111101001010000000000100100 |
| 0x000010 | 10101111101000000000000000011000 |
| 0x000014 | 10101111101000000000000000011100 |
| 0x000018 | 10001111101011100000000000011100 |
| 0x00001c | 10001111101011110000000000011000 |
| 0x000020 | 00000001110011100000000000011001 |
| 0x000024 | 00100101110010000000000000000001 |
| 0x000028 | 00101001000000010000000001100101 |
| 0x00002c | 10101111101010000000000000011100 |
| 0x000030 | 00000000000000000111100000010010 |
| 0x000034 | 00000011000011111100100000100001 |
| 0x000038 | 00010100001000001111111111110111 |
| 0x00003c | 10101111101110010000000000011000 |
| 0x000040 | 00111100000001000000000000000000 |
| 0x000044 | 10001111101001010000000000000000 |
| 0x000048 | 00001100000100000000000011101100 |
| 0x00004c | 00100100000000000000000000000000 |
| 0x000050 | 10001111101111110000000000010100 |
| 0x000054 | 00100111101111010000000000100000 |
| 0x000058 | 00000011111000000000000000001000 |
| 0x00005c | 00000000000000000001000000100001 |

# Link 1: combine `prog.o`, `libc.o`

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

| Label | Address | |
|---|---|---|
| `main:` | `0x00000000` | |
| `loop:` | `0x00000018` | |
| `str:` | `0x10000430` | |
| `printf:` | `0x000003b0` | … |

- Relocation Information

| Address | Instr. Type | Dependency | |
|---|---|---|---|
| `0x00000040` | `lui` | `l.str` | |
| `0x00000044` | `ori` | `r.str` | |
| `0x0000004c` | `jal` | `printf` | … |

# 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, 0x1000
44 ori    $4,$4,0x0430
48 lw     $5,24($29)
4c jal    0x000003b0
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.

# Things to Remember (1/2)

- 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.

# Things to Remember (2/2)

- ***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)

# Quiz

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

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

# Quiz

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
```

# 1: data reference; relocate Yes

# 2: PC-relative branch; OK No

# Integer Multiplication (1/3)

- Paper and pencil example (unsigned):

```
   Multiplicand        1000      8
     Multiplier       x1001      9
                       1000
                      0000
                     0000
                   +1000
                   01001000
```

- m bits x n bits = m + n bit product

# Integer Multiplication (2/3)

- In MIPS, we multiply registers, so:
  - 32-bit value x 32-bit value = 64-bit value
- Syntax of Multiplication (signed):
  - `mult` register1, register2 No destination register!
  - 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 and mflo register to move from hi, lo to another register

# Integer Multiplication (3/3)

- Example:
  - in C: `a = b * 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.

# Integer Division (1/2)

- Paper and pencil example (unsigned):

```
                  1001      Quotient
                ________
Divisor 1000 | 1001010      Dividend
              -1000
                  10
                  101
                  1010
                 -1000
                    10  Remainder
                     (or Modulo result)
```

- Dividend = Quotient x Divisor + Remainder

# Integer Division (2/2)

- Syntax of Division (signed):
  - `div` register1, register2
  - Divides 32-bit register1 by 32-bit register2
  - 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
```