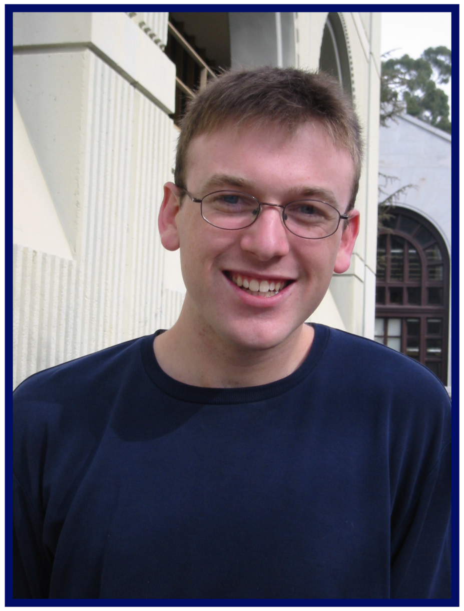
inst.eecs.berkeley.edu/~cs61c

CS61CL : Machine Structures

Lecture #7 – Compilation, Assembly, and Linking

2009-07-12



Jeremy Huddleston

Review: Two’s Complement

- Change every 0 to 1 and 1 to 0 (invert or complement), then add 1 to the result

- Example: -3 to +3 to -3
  x : 1111 1111 1111 1111 1111 1111 1111 1101$_{two}$
  x’: 0000 0000 0000 0000 0000 0000 0000 0010$_{two}$
  +1: 0000 0000 0000 0000 0000 0000 0000 0011$_{two}$
  ()’: 1111 1111 1111 1111 1111 1111 1111 1100$_{two}$
  +1: 1111 1111 1111 1111 1111 1111 1111 1101$_{two}$

Review: IEEE 754 Single Precision FP

- Reserve exponents, significands:

Exponent	Significand	Object
0	0	+/- 0
0	nonzero	+/- Denorm
1-254	anything	+/- Norm
255	0	+/- ∞
255	nonzero	NaN

Review: Rounding

- When we perform math on floating point numbers, we have to worry about rounding to fit the result in the significand field.
- The FP hardware carries two extra bits of precision, and then round to get the proper value
- Rounding also occurs when converting:
  - double to a single precision value
  - floating point number to an integer
  - integer > ___ to floating point

Review: IEEE FP Rounding Modes

Examples in decimal (but, of course, IEEE754 in binary)

- Round towards + ∞
  - ALWAYS round "up": 2.001 → 3, -2.001 → -2
- Round towards - ∞
  - ALWAYS round "down": 1.999 → 1, -1.999 → -2
- Truncate
  - Just drop the last bits (round towards 0)
- Unbiased (default mode). Midway? Round to even
  - Normal rounding, almost: 2.4 → 2, 2.6 → 3, 2.5 → 2, 3.5 → 4
  - Round like you learned in grade school (nearest int)
  - Except if the value is right on the borderline, in which case we round to the nearest EVEN number
  - Insures fairness on calculation
  - This way, half the time we round up on tie, the other half time we round down. Tends to balance out inaccuracies

Assignment Administrivia

- hw3, proj1 will be graded soon.
- hw4 – Expertiza & peer review
- proj2 Friday

Feedback Administrivia

- **Feedback Responses (Thank You)**
  - **Discussion Sections**
    - Wednesday 12-1, _______ (Paul)
    - Thursday 10-11, 277 Cory (Jeremy)
  - **More OH time?**
    - Currently 5.5hrs/wk
    - Email if you need another block. I'm flexible.
  - **Labs rushed...**
    - Should have about 2 staff in lab at all times now
  - **More animated examples, blackboard**
  - **"Extra for Experts"**
    - If it's not labeled as such, it's "fair game"

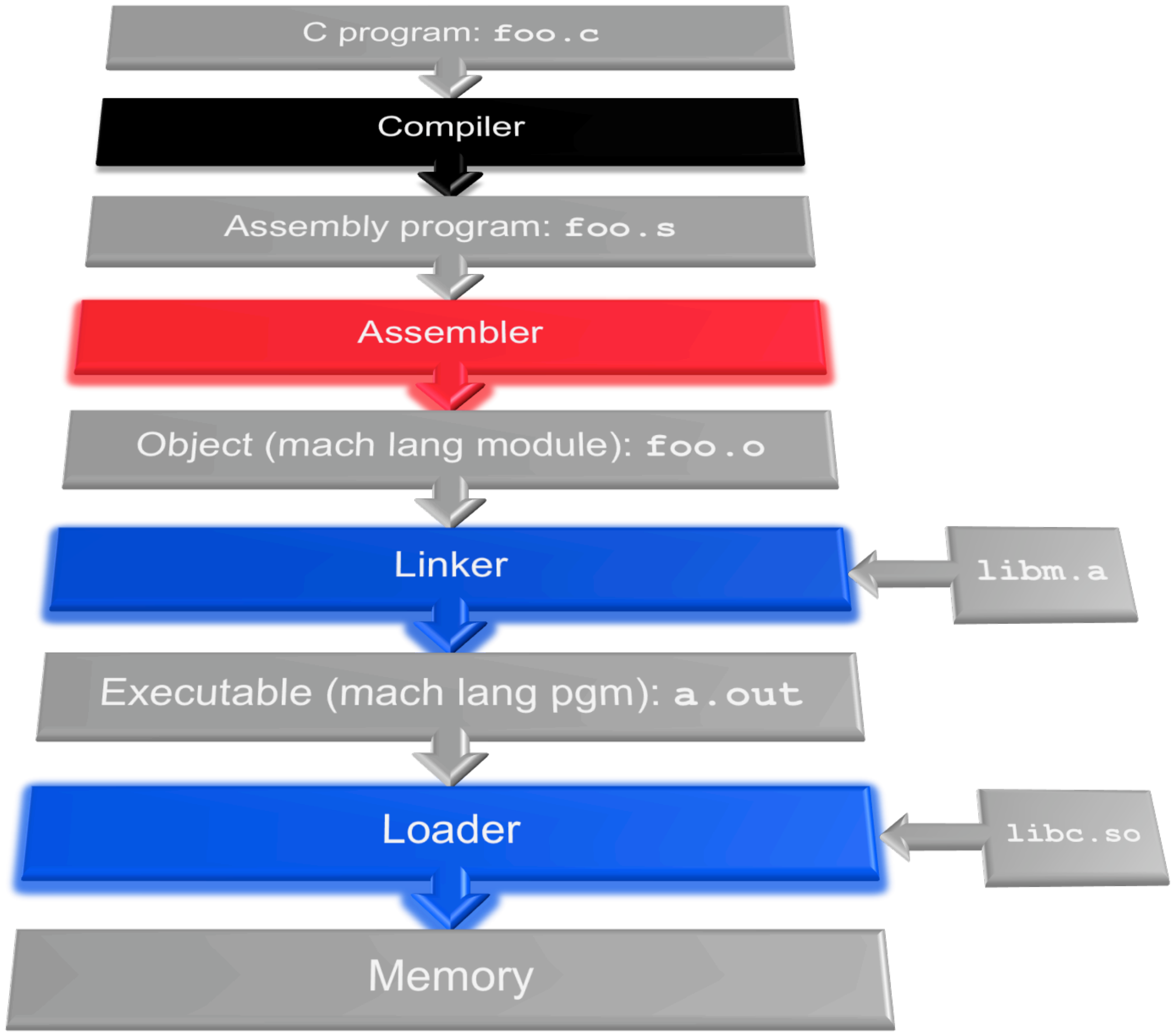
Midterm Administrivia

- Next Monday (in class)
  - No Lab (catch up!)
- "FAA" rules for what you can bring
  - Nothing electronic
  - 2 books and a few pages of notes
    - "a few pages" means no more than ~35 sheets
    - Bringing in P&H, K&R, all the slides, and some handwritten notes is OK
  - Don't kill trees printing out every lab, brainstorm answer, etc!
  - If we say it's too much, it's too much. If in doubt, ask ahead of time.

Overview

- Translating C Programs
  - Compiler
  - Assembler
  - Linker
  - Loader
- Interpretation vs Translation

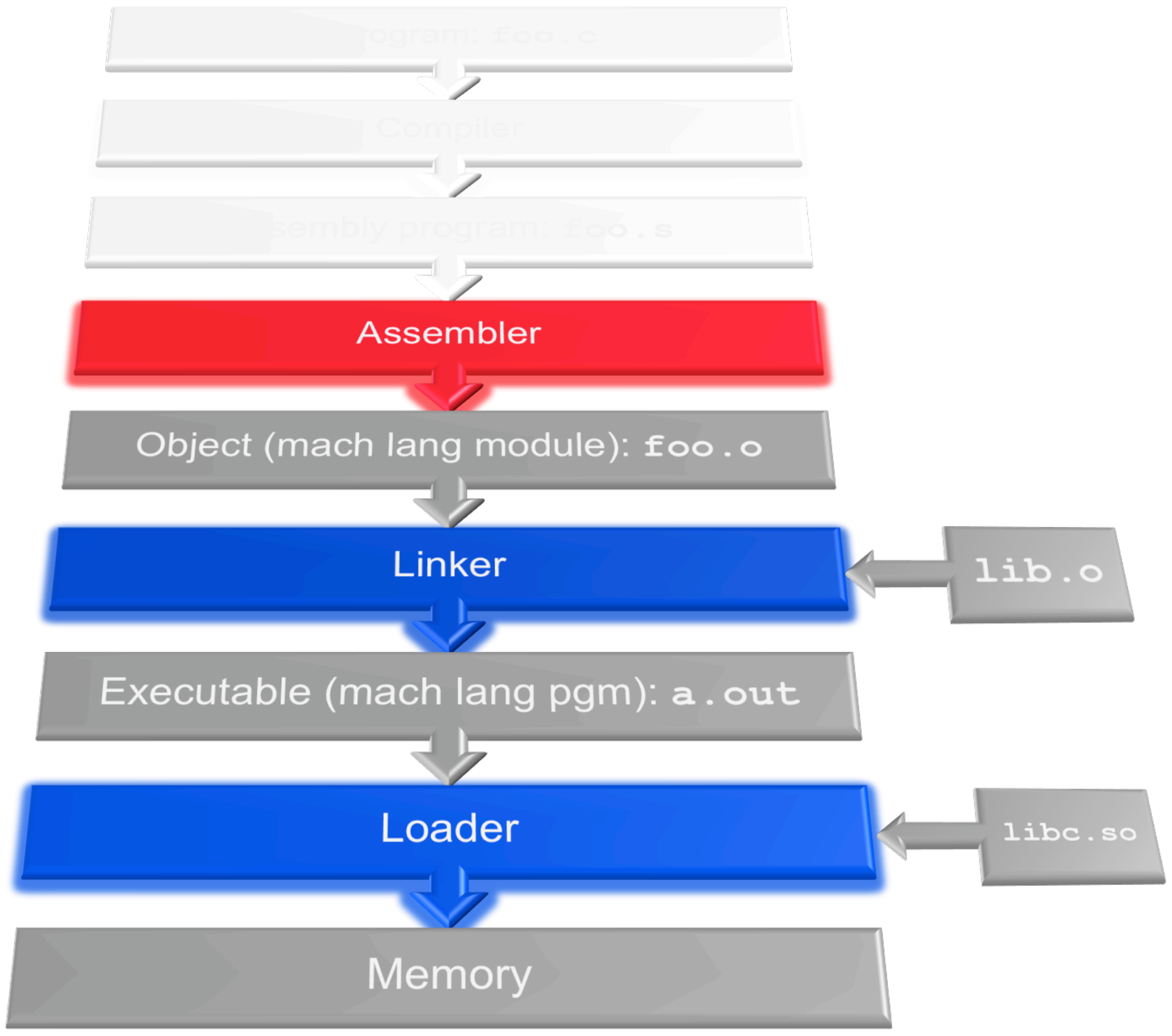
Steps to Starting a Program (translation)



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

Where Are We Now?



Assembler

- Input: Assembly Language Code (e.g., `foo.s` for MIPS)
- Output: Object Code, information tables (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

True Assembly Language (1/6)

- Pseudoinstruction: A MIPS instruction that doesn't turn directly into a machine language instruction, but into other MIPS instructions
- What happens with pseudo-instructions?
    - They're broken up by the assembler into several "real" MIPS instructions.
- Some examples follow

True Assembly Language (2/6)

- **Register Move**

```
move   reg2,reg1
```

  Expands to:

```
add    reg2,$zero,reg1
```

  or perhaps:

```
or     reg2,$zero,reg1
```

- **Load Immediate**

```
li     reg,value
```

  If value fits in 16 bits:

```
addi   reg,$zero,value
```

  else:

```
lui    reg,upper 16 bits of value
ori    reg,$zero,lower 16 bits
```

True Assembly Language (3/6)

- **Problem:**
  - When breaking up a pseudo-instruction, the assembler may need to use an extra register
  - If it uses any regular register, it'll overwrite whatever the program has put into it.
- **Solution:**
  - Reserve a register (`$1`, called `$at` for "assembler temporary") that assembler will use to break up pseudo-instructions.
  - Since the assembler may use this at any time, it's not safe to code with it.

True Assembly Language (4/6)

- So how does `lui` help us?
  - Example:

```
addi   $t0, $t0, 0xABABCDCD
```

    becomes:

```
lui    $at,      0xABAB
ori    $at, $at, 0xCDCD
add    $t0, $t0, $at
```

  - Now each I-format instruction has only a 16-bit immediate.

- The assembler does this for us!
  - If number too big, then just automatically replace `addi` with `lui`, `ori`, `add`

True Assembly Language (3/3)

- MAL (MIPS Assembly Language): the set of instructions that a programmer may use to code in MIPS; this includes pseudoinstructions
- TAL (True Assembly Language): set of instructions that can actually get translated into a single machine language instruction (32-bit binary string)
- A program must be converted from MAL into TAL before translation into 1s & 0s.

Pseudoinstruction Replacement

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

Pseudo:	Real:
`subu $sp,$sp,32`	`addiu $sp,$sp,-32`

`sd $a0, 32($sp)`	`sw $a0, 32($sp)`
	`sw $a1, 36($sp)`

`mul $t7,$t6,$t5`	`mul $t6,$t5`
	`mflo $t7`

`addu $t0,$t6,1`	`addiu $t0,$t6,1`

`ble $t0,100,loop`	`slti $at,$t0,101`
	`bne $at,$0,loop`

`la $a0, str`	`lui $at,upper(str)`
	`ori $a0,$at,lower(str)`

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

Producing Machine Language (3/3)

- 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 symbols exported by this file that may be used by other files and symbols local to this file that will need relocation.
- In C, function names and variable names
- What are they in MAL?
  - Labels in `.data` and `.text` section; variables/functions which may be accessed across files

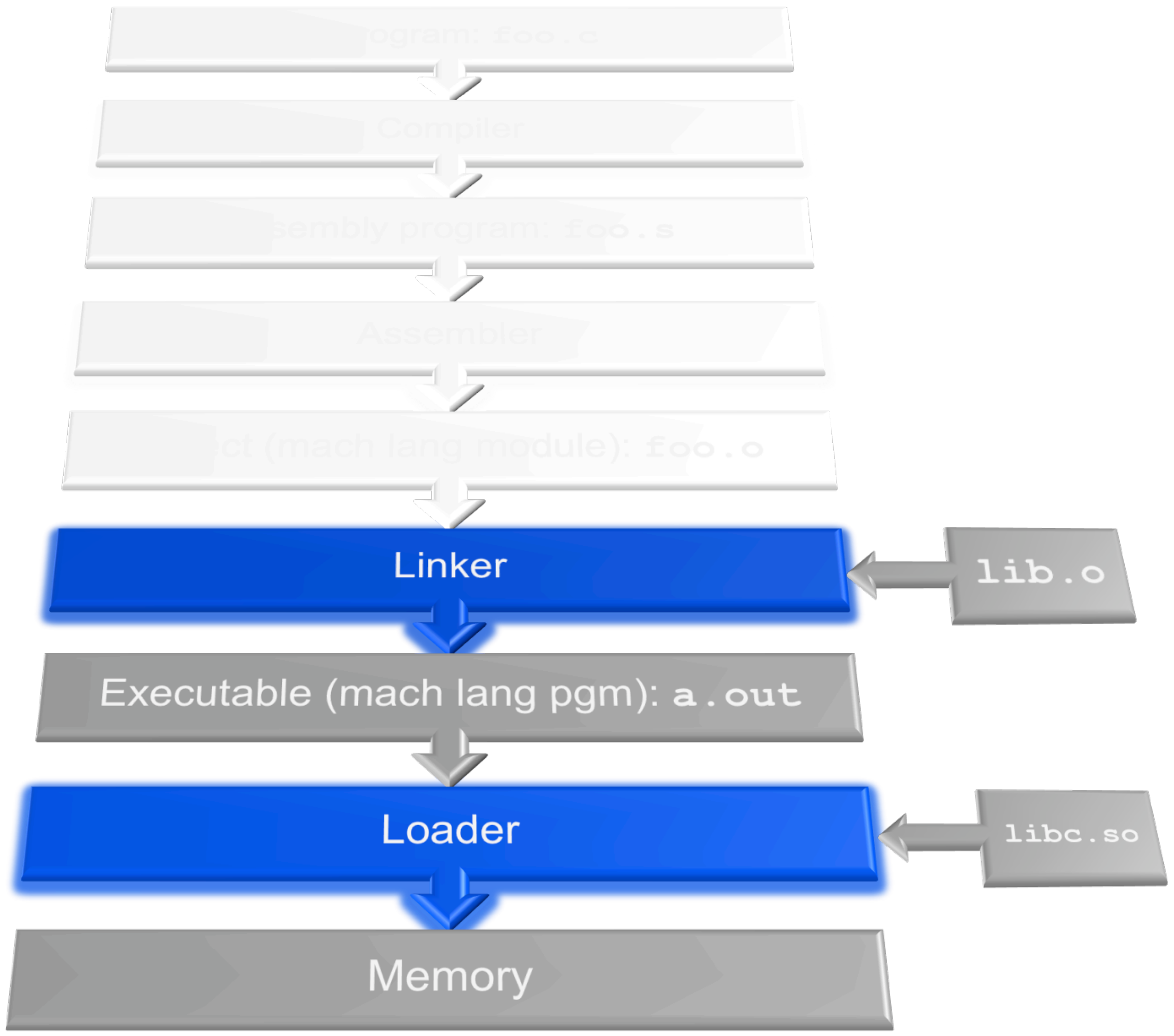
Relocation Table

- List of instructions that will need to be edited based on where a symbol is located at runtime.
- What are they?
  - Any label jumped to: `j` or `jal`
    - internal
    - external (including lib files)
  - Any piece of data
    - 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 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
- A common format is ELF
  `http://www.skyfree.org/linux/references/ELF_Format.pdf`

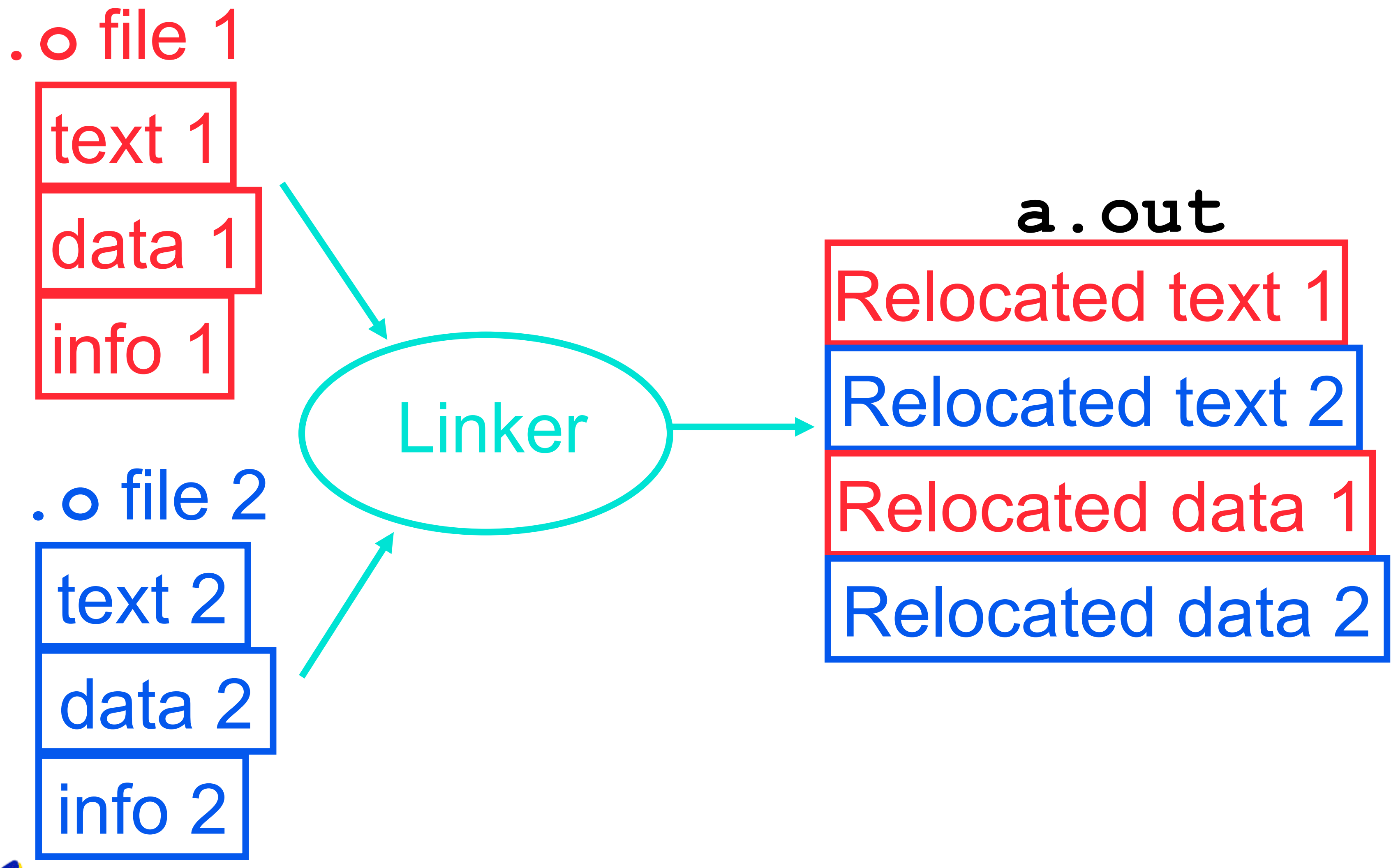
Where Are We Now?



Linker (1/3)

- Input: Object Code files, information tables (e.g., `foo.o`,`libm.a` 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
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 we'll discuss

- 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`)
  - always relocate

Absolute Addresses in MIPS

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

`j/jal`	`xxxxx`

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

`lw/sw`	`$gp`	`$x`	`address`

  - **What about conditional branches?**

`beq/bne`	`$rs`	`$rt`	`address`

  - **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)

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 libraries (DLL, SO, DYLIB), common on Windows & UNIX platforms

Dynamically linked libraries

en.wikipedia.org/wiki/Dynamic_linking

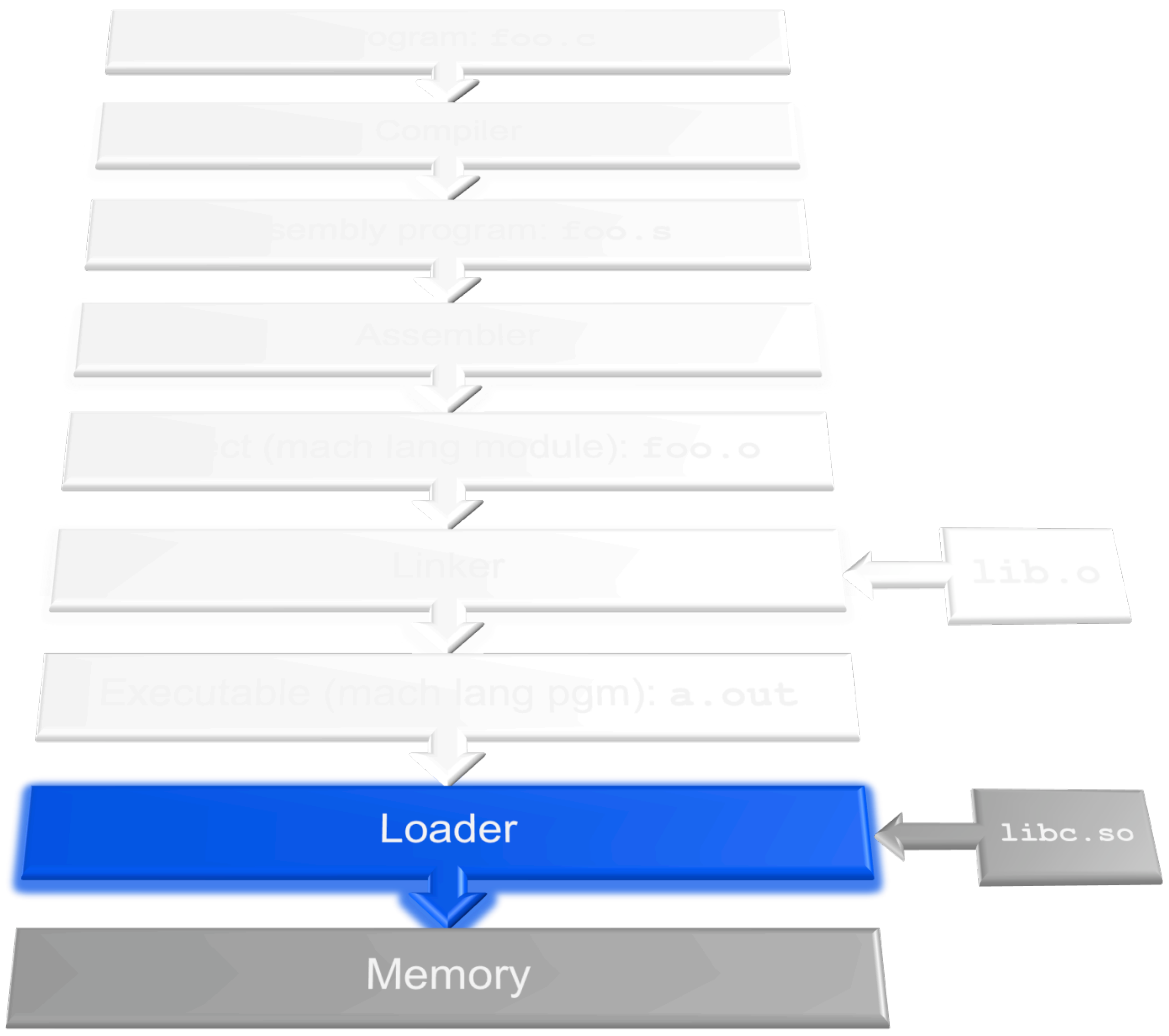
- Space/time issues
  - + Storing a program requires less disk space
  - + Sending a program requires less time
  - + Executing two programs requires less memory (if they share a library)
  - – At runtime, there's time overhead to do link
- Upgrades
  - + Replacing one file (`libXYZ.so`) upgrades every 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)
- Executables produced by the linker still contain symbol tables and relocation tables to be resolved later.

Where Are We Now?



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

Loader (2/3)

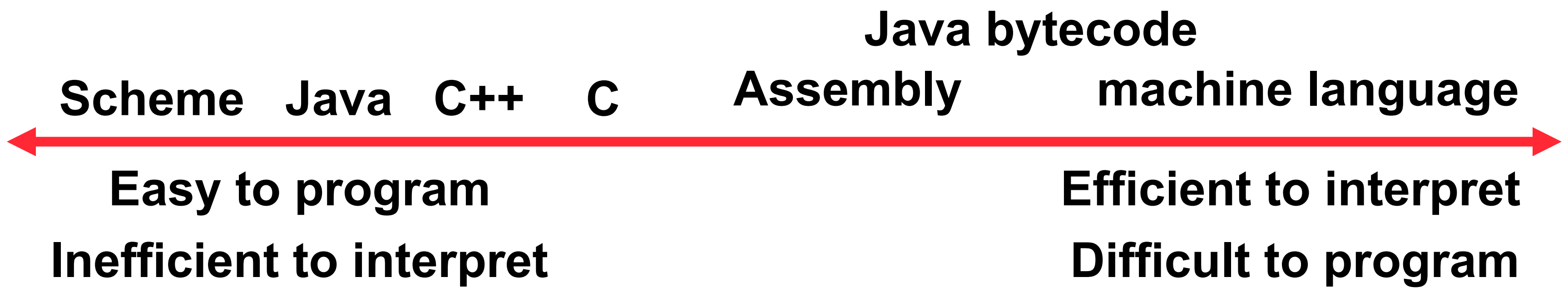
- So what does a loader do?
- Reads executable file's header to determine size of text and data segments
- Creates new address space for program large enough to hold text and data segments, along with a stack segment
- Copies instructions and data from executable file into the new address space

Loader (3/3)

- Copies arguments passed to the program onto the stack
- Initializes machine registers
  - Most registers cleared, but stack pointer assigned address of 1st free stack location
- 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

Language Execution Continuum

- An Interpreter is a program that executes other programs.

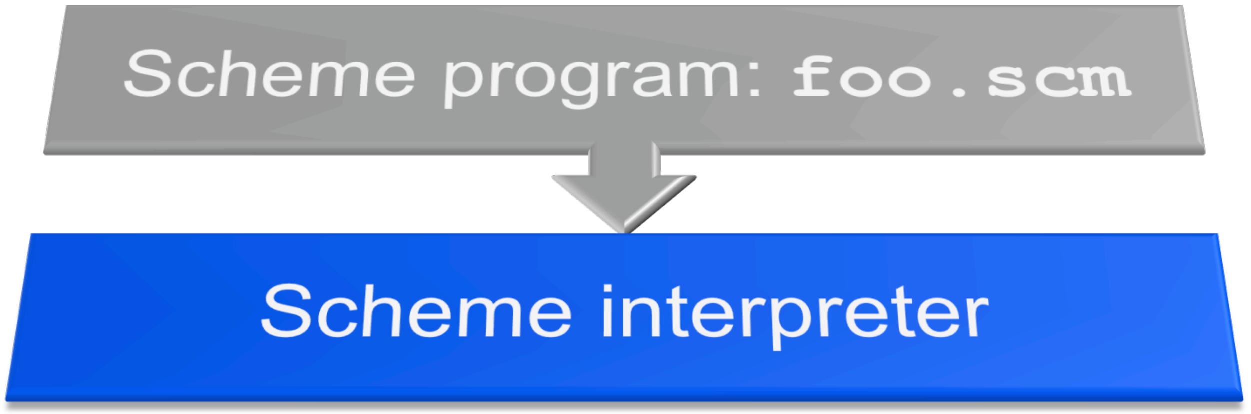


- Language translation gives us another option.
- In general, we interpret a high level language when efficiency is not critical and translate to a lower level language to up performance

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 Scheme program `foo.scm`

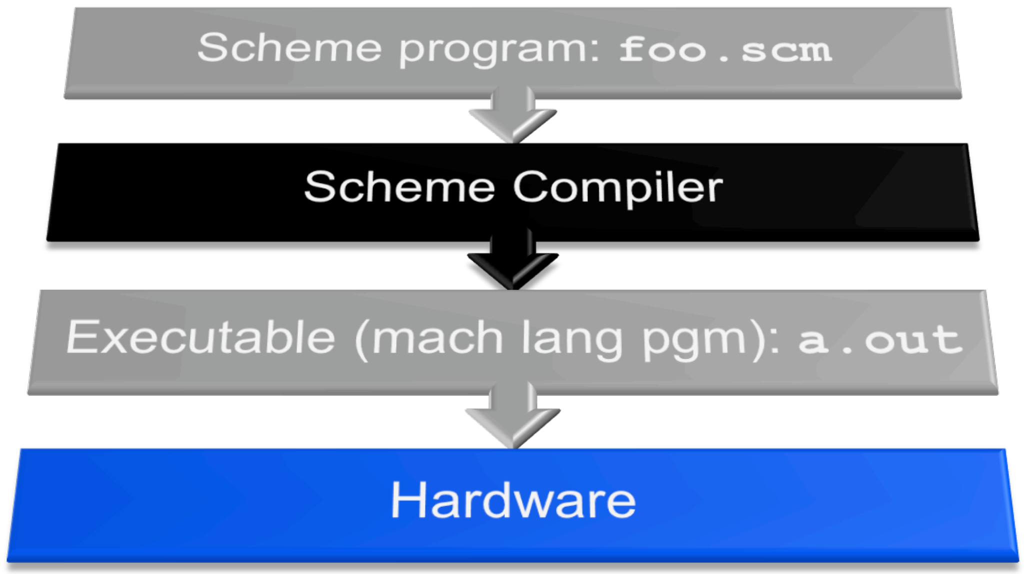
Interpretation



- Scheme Interpreter is just a program that reads a scheme program and performs the functions of that scheme program.

Translation

- Scheme Compiler is a translator from Scheme to machine language.
- The processor is a hardware interpeter of machine language.



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)

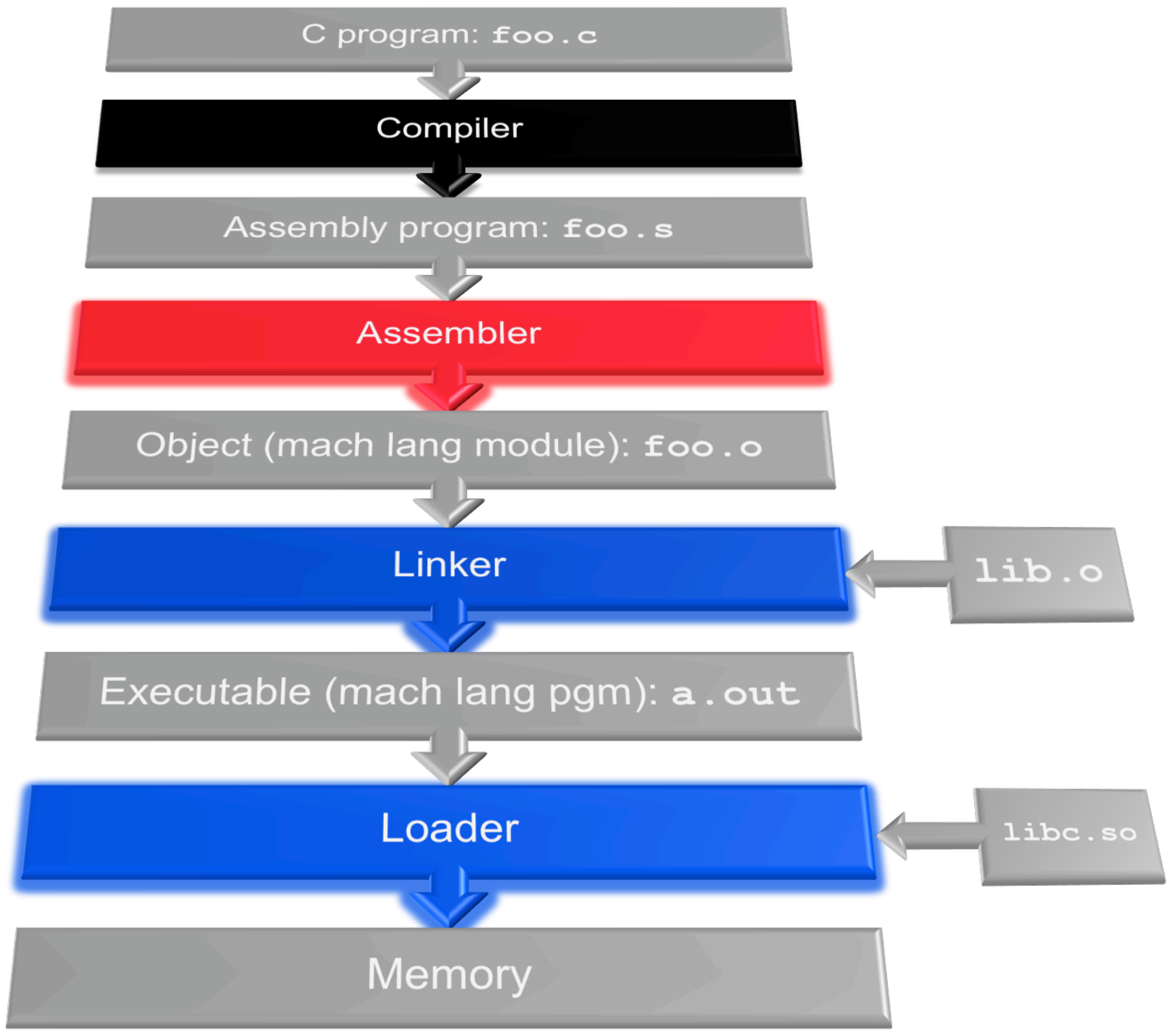
Interpretation vs. Translation? (1/2)

- Generally easier to write interpreter
- Interpreter closer to high-level, so can give better error messages (MARS)
  - Translator reaction: add extra information to help debugging (line numbers, names)
- Interpreter slower (10x?)
- 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 and graphics.
- 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. (eg. Darwin [core of OSX], Linux, BSD, Solaris…)

Things to Remember (1/3)



Things to Remember (2/3)

- Compiler converts a single HLL file into a single assembly language file.
- Assembler removes pseudoinstructions, 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 3/3

- 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)

Bonus slides

- These are extra slides that used to be included in lecture notes, but have been moved to this, the “bonus” area to serve as a supplement.
- The slides will appear in the order they would have in the normal presentation

Bonus

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
  - 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

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 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
```

Big Endian vs. Little Endian

Big-endian and little-endian derive from Jonathan Swift's *Gulliver's Travels* in which the Big Endians were a political faction that broke their eggs at the large end ("the primitive way") and rebelled against the Lilliputian King who required his subjects (the Little Endians) to break their eggs at the small end.

(E.g., 0xC2=0b 1100 0010)

Consider the number 1025 as we normally write it:

BYTE2	BYTE1	BYTE0
00000000	00000100	00000001

Big Endian

-

ADDR3	ADDR2	ADDR1	ADDR0
BYTE0	BYTE1	BYTE2	
00000001	00000100	00000000	

-

ADDR0	ADDR1	ADDR2	ADDR3
	BYTE2	BYTE1	BYTE0
	00000000	00000100	00000001

Little Endian

-

ADDR3	ADDR2	ADDR1	ADDR0
	BYTE2	BYTE1	BYTE0
	00000000	00000100	00000001

-

ADDR0	ADDR1	ADDR2	ADDR3
BYTE0	BYTE1	BYTE2	
00000001	00000100	00000000	

`www.webopedia.com/TERM/b/big_endian.html`
`searchnetworking.techtarget.com/sDefinition/0,,sid7_gci211659,00.html`
`www.noveltheory.com/TechPapers/endian.asp`
`en.wikipedia.org/wiki/Big_endian`

Example: C ⇒ Asm ⇒ Obj ⇒ Exe ⇒ Ru

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,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

```
_.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

```
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 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
    - ext segment (instructions),
    - data segment (data),
    - symbol and relocation tables.
  - Using dummy “placeholders” for unresolved absolute and external references.

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      00000000000000000111100000010010
0x000034      00000011000011111100100000100001
0x000038      00010100001000001111111111110111
0x00003c      10101111101110010000000000011000
0x000040      00111100000001000000000000000000
0x000044      10001111101001010000000000000000
0x000048      00001100000100000000000011101100
0x00004c      00100100000000000000000000000000
0x000050      10001111101111110000000000010100
0x000054      00100111101111010000000000100000
0x000058      00000011111000000000000000001000
0x00005c      00000000000000000001000000100001
```

Link step 1: combine `prog.o`, `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:`	`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, 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 preceeding example was a much simplified version of how ELF and other standard formats work, meant only to demonstrate the basic principles.