inst.eecs.berkeley.edu/~cs61c

UCB CS61C : Machine Structures

Lecture 13 – Running a Program
(Compiling, Assembling, Linking, Loading)

Instructor
Paul Pearce

2010-07-13

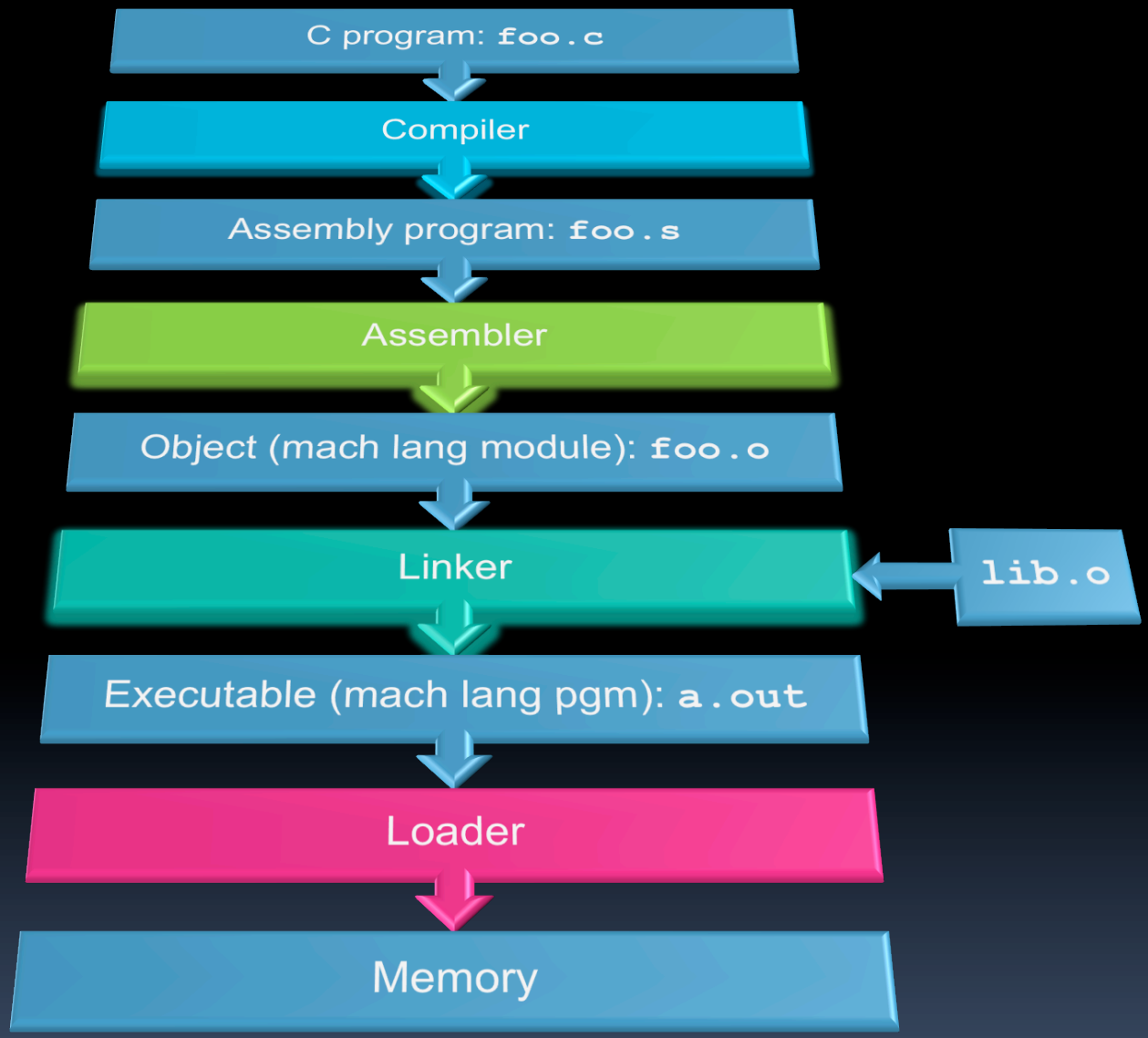
SUPPORT FOR WINDOWS XP SP2 ENDS TODAY. UPGRADE TO SP3 NOW OR YOUR LAPTOP WILL EXPLODE!

Ok. It won't actually explode. But MS will no longer be releasing updates for SP2 as of today. This means you are left extremely vulnerable! So please, for the sake of the internet, upgrade to SP3 if you still use Windows XP. (Or get a Mac)

http://tinyurl.com/38dfts2

Pearce, Summer 2010 © UCB

Steps to Starting a Program (translation)



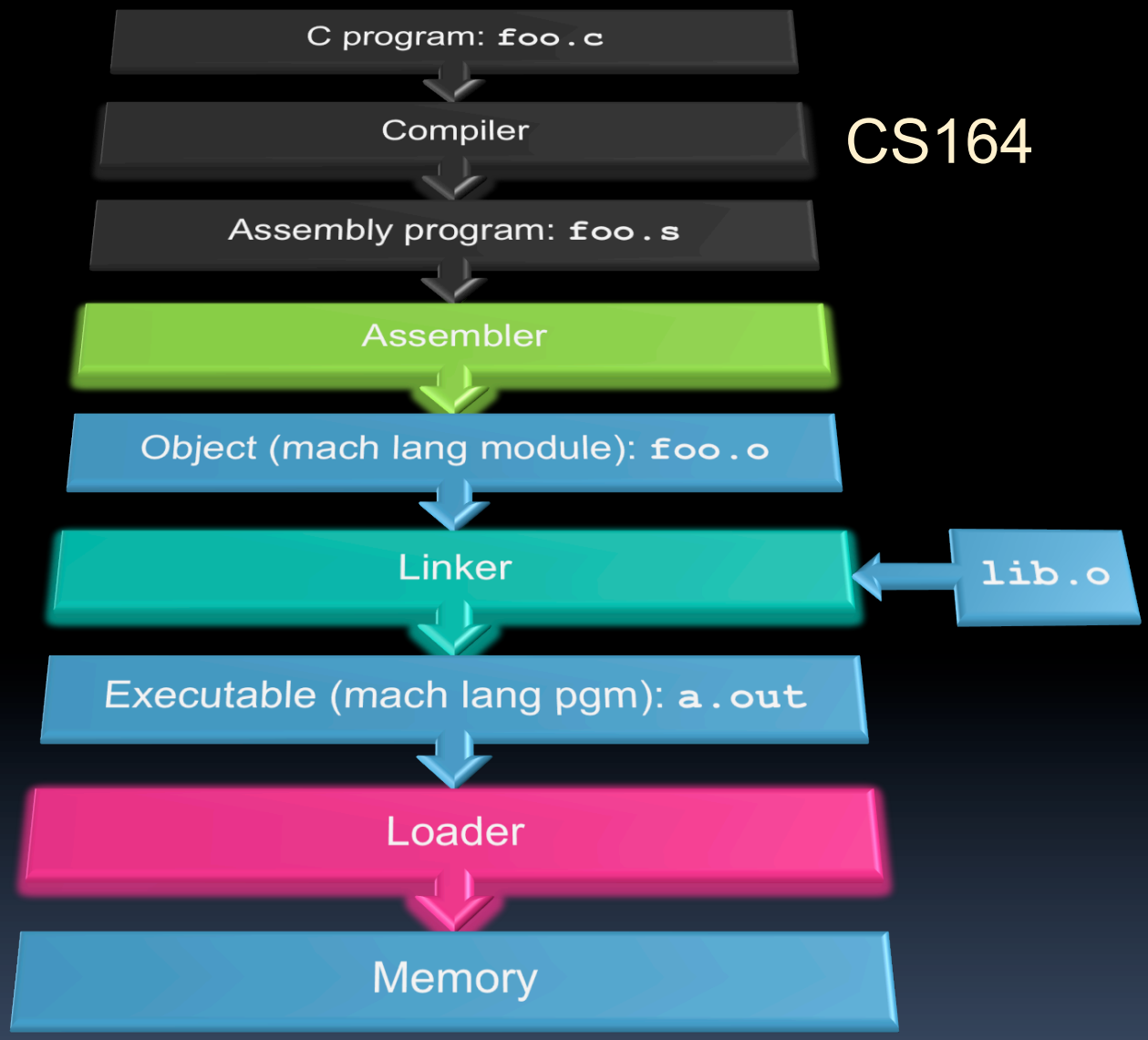
Pearce, Summer 2010 © UCB

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
- Pseudoinstructions: instructions that assembler understands but are not machine language instructions (last lecture)
  For example:
  - `mov $s1,$s2` ⇒ `or $s1,$s2,$zero`

Pearce, Summer 2010 © UCB

Where Are We Now?



Pearce, Summer 2010 © UCB

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

Pearce, Summer 2010 © UCB

Assembler Directives (p. B-5)

- 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

Pearce, Summer 2010 © UCB

Pseudoinstruction Replacement

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

Pseudo (MAL):	Real (TAL):
`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,left(str)` `ori $a0,$at,right(str)`

Pearce, Summer 2010 © UCB

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.
  - Extra for experts: What about long branches?

Pearce, Summer 2010 © UCB

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

Pearce, Summer 2010 © UCB

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…

Pearce, Summer 2010 © UCB

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

Pearce, Summer 2010 © UCB

Relocation Table

- List of "items" this file needs the address later.
- What are they?
  - Any label jumped to: `j` or `jal`
    - internal
    - external (including lib files)
  - Any piece of data that references an address
    - such as the `la` instruction

Pearce, Summer 2010 © UCB

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 standard format is ELF (except MS, Apple)

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

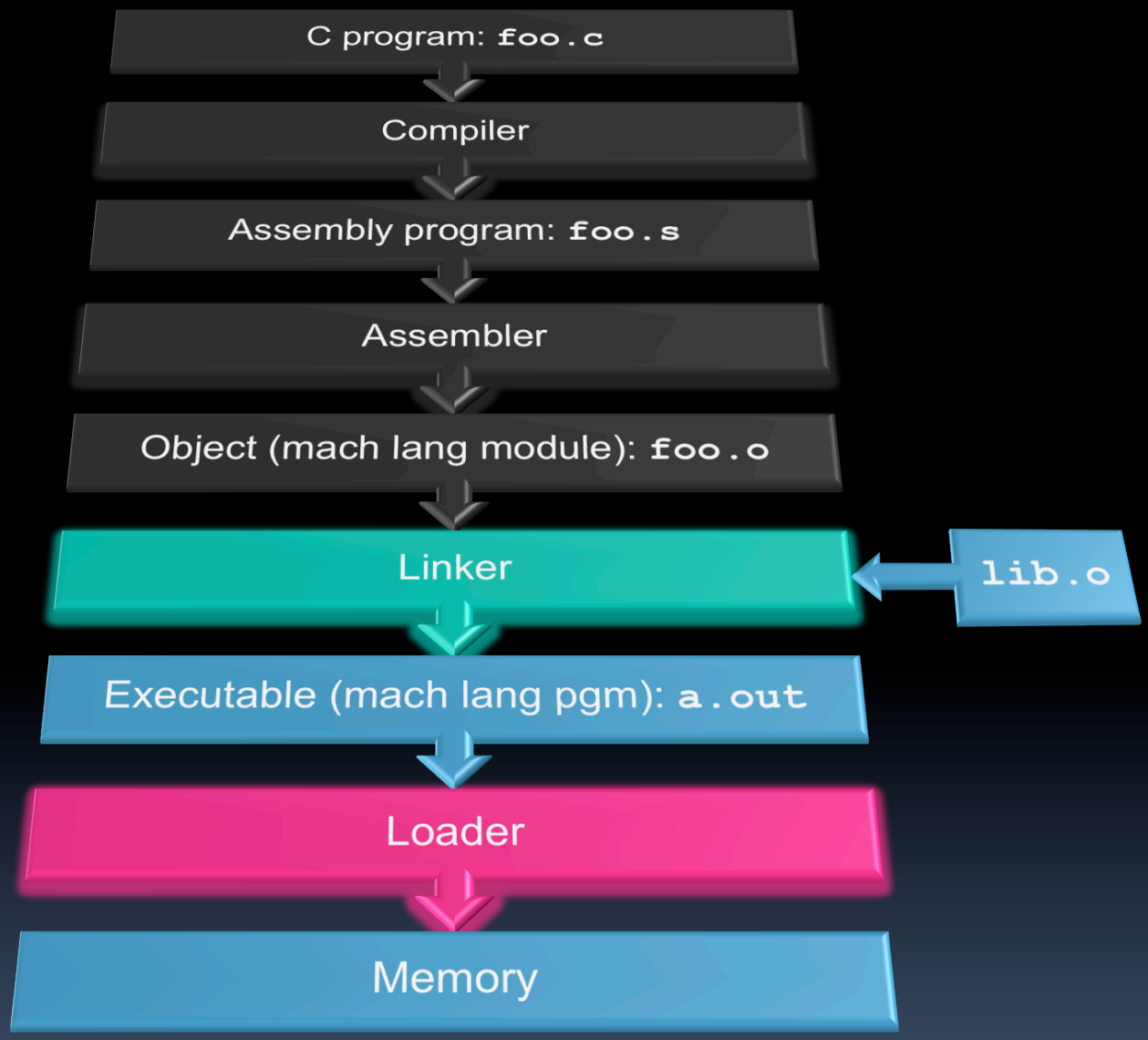
Pearce, Summer 2010 © UCB

Administrivia…Midterm on Friday!

- Review Thursday in lecture
- Exam Friday 9:30am – 12:30pm in 100 Lewis
- Covers labs, hw, proj, lec, book through **<u>today</u>**
- Bring…
  - NO cells, calculators, pagers, PDAs, laptops, books
  - 2 writing implements (we'll provide write-in exam booklets) – pencils only, please!
  - 1 page of notes, front and back
- A word of advice:
  - GET A GOOD NIGHT SLEEP BOTH WEDNESDAY AND THURSDAY NIGHT. This is as important as knowing the material!
- Mid-semester survey as part of the Lab.
  - Help give us feedback on how to make the next 4 weeks better!

Pearce, Summer 2010 © UCB

Where Are We Now?



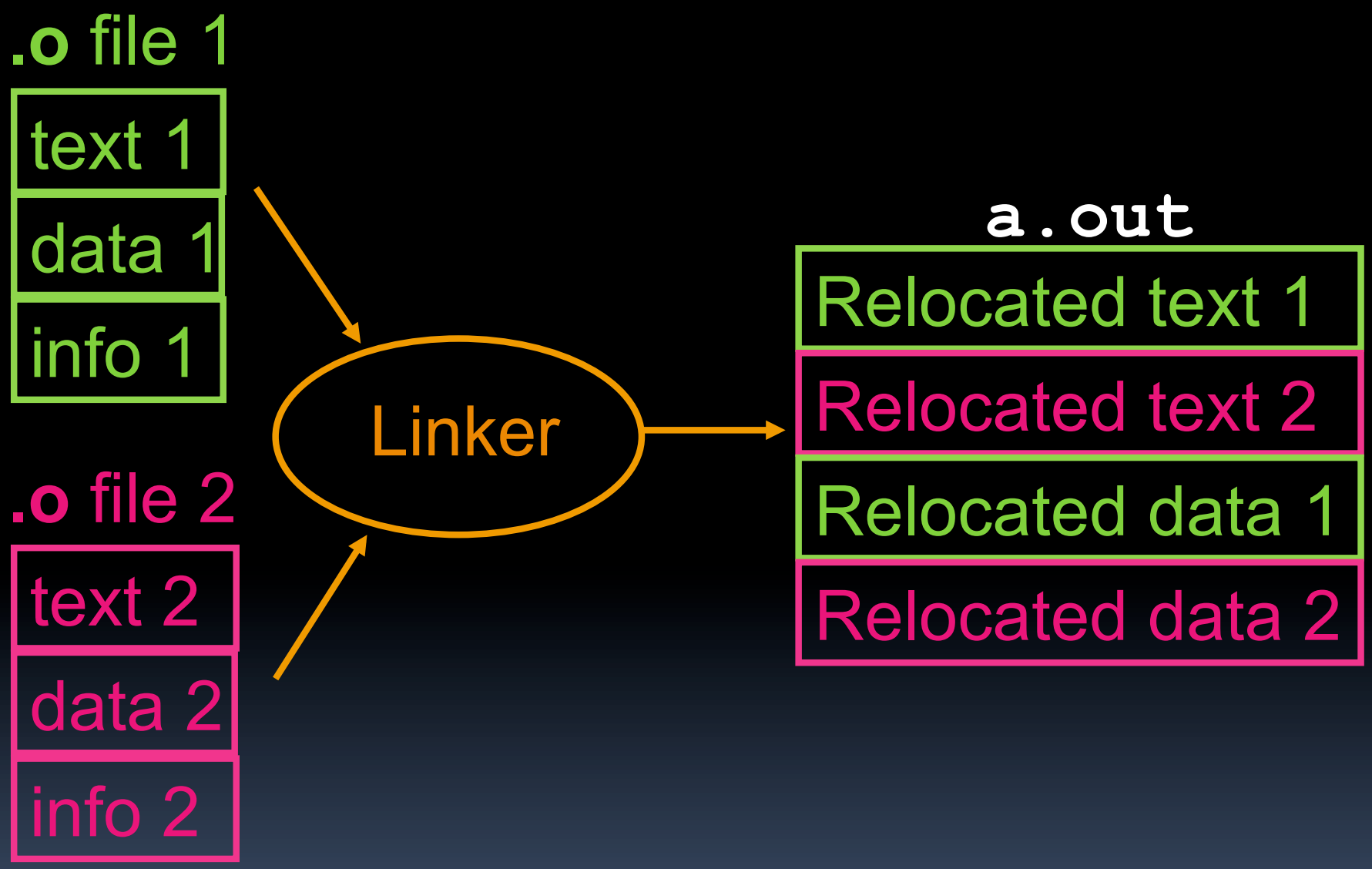
Pearce, Summer 2010 © UCB

Linker (1/3)

- 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 Vista source was > 50 M lines of code!

Pearce, Summer 2010 © UCB

Linker (2/3)



Pearce, Summer 2010 © UCB

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

Pearce, Summer 2010 © UCB

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

Pearce, Summer 2010 © UCB

Absolute Addresses in MIPS

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

j/jal	??????

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

lw/sw	$gp	$x	???????

  - What about conditional branches?

beq/bne	$rs	$rt	address

  - PC-relative addressing preserved even if code moves
  - Extra for experts: What about long branches?

Pearce, Summer 2010 © UCB

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

Pearce, Summer 2010 © UCB

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)

Pearce, Summer 2010 © UCB

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

Pearce, Summer 2010 © UCB

en.wikipedia.org/wiki/Dynamic_linking

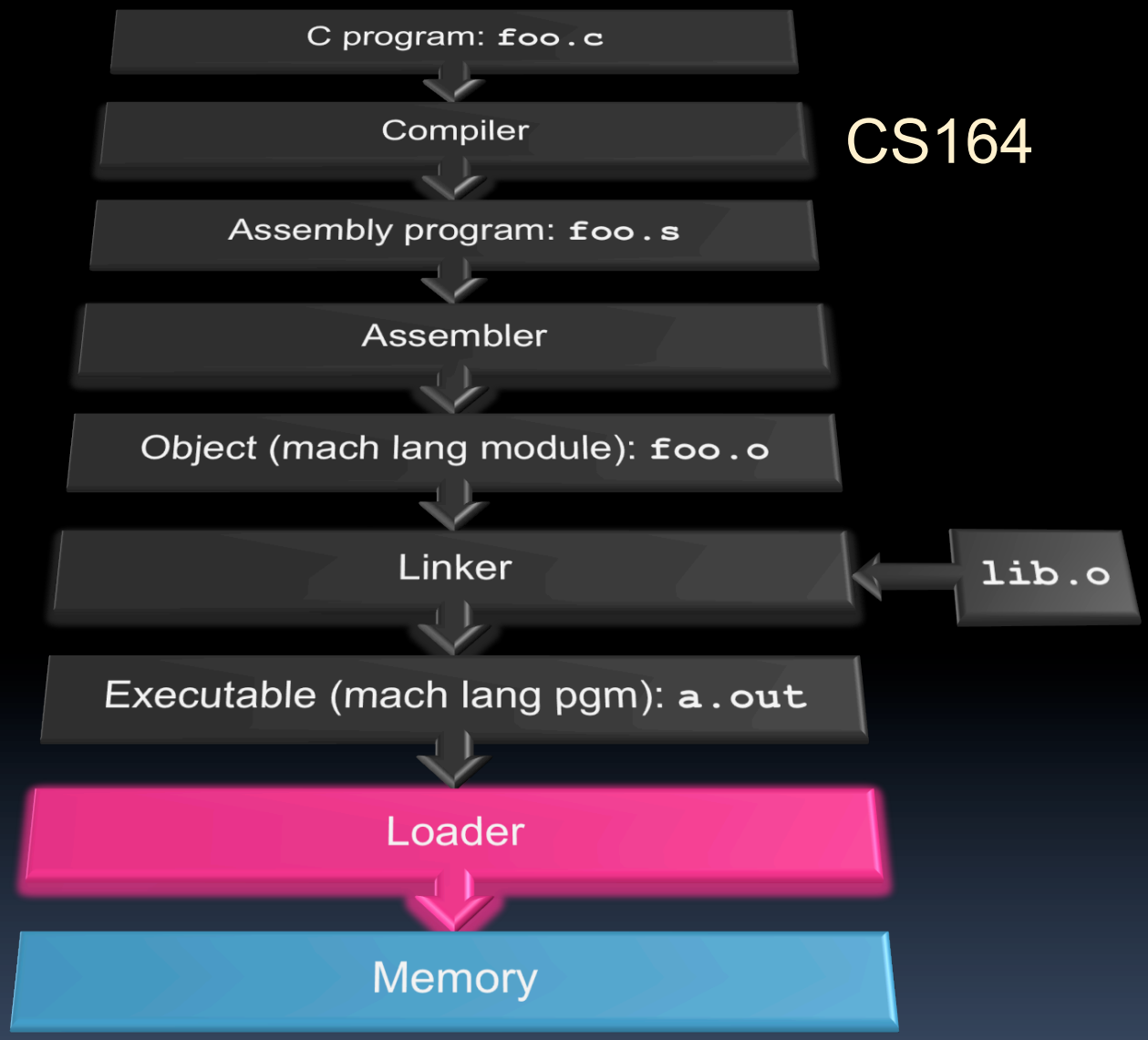
Dynamically linked libraries

- 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.

Pearce, Summer 2010 © UCB

Where Are We Now?



Pearce, Summer 2010 © UCB

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 the operating system (OS)
  - loading is one of the OS tasks

Pearce, Summer 2010 © UCB

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

Pearce, Summer 2010 © UCB

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 & sets the PC to main.
  - If main routine returns, start-up routine terminates program with the exit system call

Pearce, Summer 2010 © UCB

Peer Instruction

1) Assembler will ignore the instruction `Loop:nop` because it does nothing.

2) Java designers used a translator AND interpreter (rather than just a translator) mainly because of (at least 1 of): ease of writing, better error msgs, smaller object code.

	12
a)	FF
b)	FT
c)	TF
d)	TT

Pearce, Summer 2010 © UCB

Peer Instruction

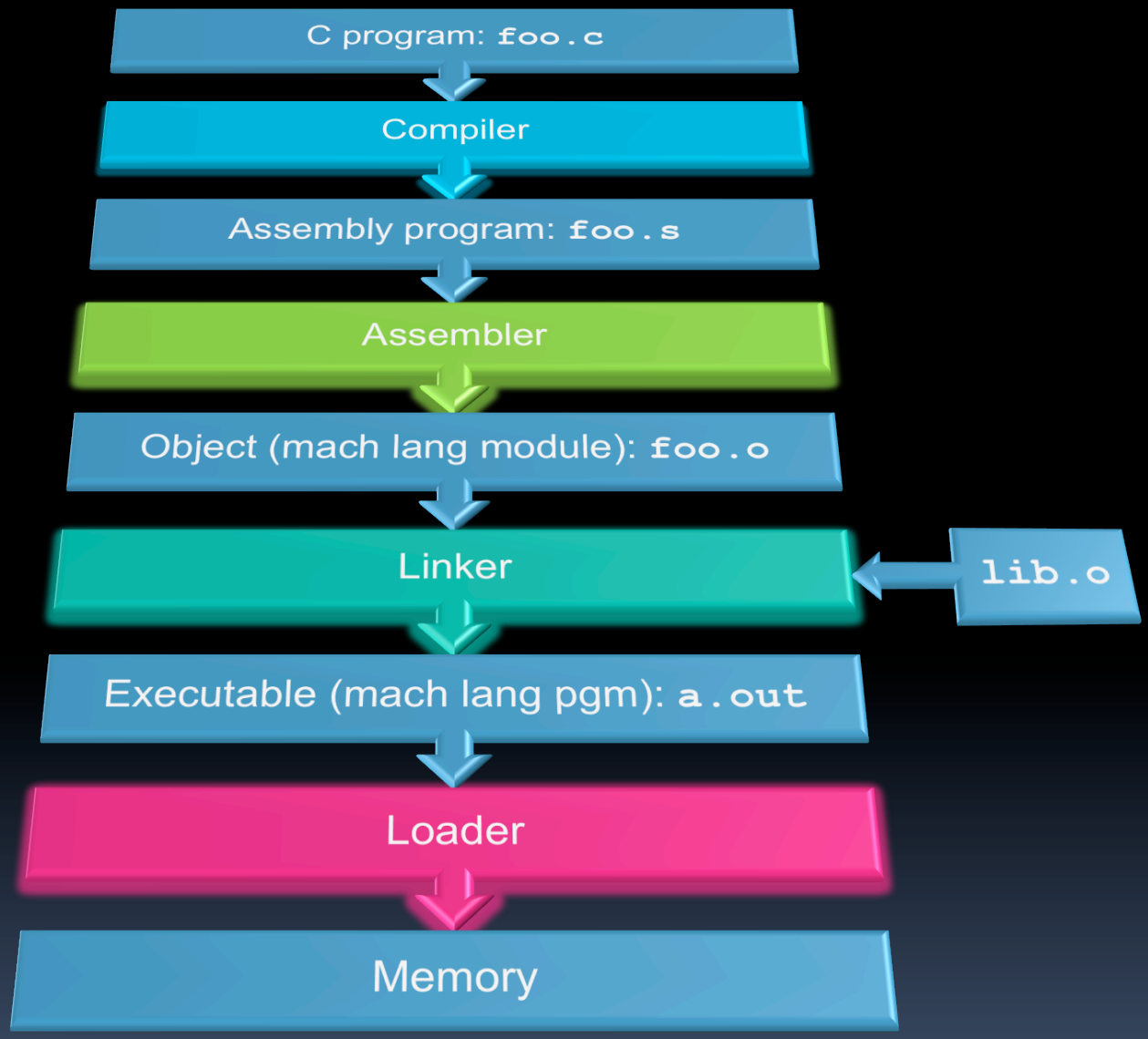
The following instr. may need to be edited during link phase. T or F.

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

	12
a)	FF
b)	FT
c)	TF
d)	TT

Pearce, Summer 2010 © UCB

Things to Remember (1/3)



Pearce, Summer 2010 © UCB

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.

Pearce, Summer 2010 © UCB

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)

Pearce, Summer 2010 © UCB

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

Pearce, Summer 2010 © UCB

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

Pearce, Summer 2010 © UCB

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

Pearce, Summer 2010 © UCB

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.

Pearce, Summer 2010 © UCB

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

Pearce, Summer 2010 © UCB

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
```


Pearce, Summer 2010 © UCB

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.

- **The order in which BYTES are stored in memory**
- **Bits always stored as usual. (E.g., 0xC2=0b 1100 0010)**

Consider the number 1025 as we normally write it:

BYTE3	BYTE2	BYTE1	BYTE0
00000000	00000000	00000100	00000001

Big Endian

- ADDR3 ADDR2 ADDR1 ADDR0
  BYTE0 BYTE1 BYTE2 BYTE3
  00000001 00000100 00000000 00000000
- ADDR0 ADDR1 ADDR2 ADDR3
  BYTE3 BYTE2 BYTE1 BYTE0
  00000000 00000000 00000100 00000001

Little Endian

- ADDR3 ADDR2 ADDR1 ADDR0
  BYTE3 BYTE2 BYTE1 BYTE0
  00000000 00000000 00000100 00000001
- ADDR0 ADDR1 ADDR2 ADDR3
  BYTE0 BYTE1 BYTE2 BYTE3
  00000001 00000100 00000000 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 ⇒ 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`"

Pearce, Summer 2010 © UCB

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`

Pearce, Summer 2010 © UCB

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
```


Pedram, Spring 2010 © UCB

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.

Pearce, Summer 2010 © UCB

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    00101001000000001000000000001100101
0x00002c    10101111101010000000000000011100
0x000030    00000000000000000111100000010010
0x000034    00000011000011111110010000100001
0x000038    00010100001000000111111111110111
0x00003c    10101111101110010000000000011000
0x000040    00111100000000100000000000000000
0x000044    10001111101001010000000000000000
0x000048    00001100000100000000000011101100
0x00004c    001001000000000000000000000000000
0x000050    10001111101111110000000000010100
0x000054    00100111101111101000000000100000
0x000058    00000011111000000000000000001000
0x00005c    00000000000000000001000000100001
```


Pearce, Summer 2010 © UCB

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 …

Pearce, Summer 2010 © UCB

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.

Pearce, Summer 2010 © UCB