


inst.eecs.berkeley.edu/~cs61c  
**UCB CS61C : Machine Structures**

**Lecture 16 – Running a Program  
 (Compiling, Assembling, Linking, Loading)**

**Sr Lecturer SOE  
 Dan Garcia**

**FACULTY “RE-IMAGINE” UGRAD EDUCATION**

Highlights: Big Ideas courses, more team teaching, Academic Honor code, report avg and median grades to share context, meaning.




is.berkeley.edu/about-college/strategic-plan-UGR-ED

### Administrivia...

- Midterm Exam - You get to bring
  - Your study sheet
  - Your green sheet
  - Pens & Pencils
- What you don't need to bring
  - Calculator, cell phone, pagers
- Conflicts? DSP accomodations? Email Head TA

Cal CS61C: Running a Program I ... Compiling, Assembling, Linking, and Loading II © UC Berkeley, Spring 2014

### Interpretation

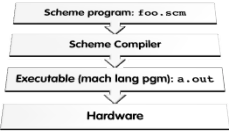


- Scheme Interpreter is just a program that reads a scheme program and performs the functions of that scheme program.

Cal CS61C: Running a Program I ... Compiling, Assembling, Linking, and Loading II © UC Berkeley, Spring 2014

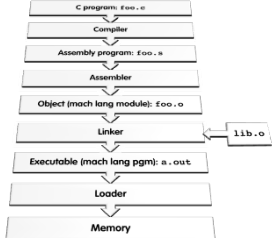
### Translation

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



Cal CS61C: Running a Program I ... Compiling, Assembling, Linking, and Loading II © UC Berkeley, Spring 2014

### Steps to Starting a Program (translation)



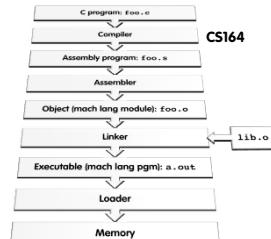
Cal CS61C: Running a Program I ... Compiling, Assembling, Linking, and Loading II © UC Berkeley, Spring 2014

### 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 not in machine
  - `move $s1,$s2` ⇒ or `$s1,$s2,$zero`

Cal CS61C: Running a Program I ... Compiling, Assembling, Linking, and Loading II © UC Berkeley, Spring 2014

### Where Are We Now?



Cal CS61C: Running a Program I ... Compiling, Assembling, Linking, and Loading II © UC Berkeley, Spring 2014

### Assembler

- Input: Assembly Language Code (MAL) (e.g., `foo.s` for MIPS)
- Output: Object Code, information tables (TAL) (e.g., `foo.o` for MIPS)
- Reads and Uses Directives
- Replace Pseudoinstructions
- Produce Machine Language
- Creates Object File

Cal CS61C: Running a Program I ... Compiling, Assembling, Linking, and Loading II © UC Berkeley, Spring 2014

### 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
  - .ascii str:** Store the string `str` in memory and null-terminate it
  - .word w1...wn:** Store the `n` 32-bit quantities in successive memory words

Cal CS61C: Running a Program I ... Compiling, Assembling, Linking, and Loading II © UC Berkeley, Spring 2014

## Pseudoinstruction Replacement

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

Pseudo:	Real:
<code>subu \$sp,\$sp,32</code>	<code>addiu \$sp,\$sp,-32</code>
<code>sd \$a0, 32(\$sp)</code>	<code>sw \$a0, 32(\$sp)</code>
	<code>sw \$a1, 36(\$sp)</code>
<code>mul \$t7,\$t6,\$t5</code>	<code>mul \$t6,\$t5</code>
	<code>mflo \$t7</code>
<code>addu \$t0,\$t6,1</code>	<code>addiu \$t0,\$t6,1</code>
<code>ble \$t0,100,loop</code>	<code>slti \$at,\$t0,101</code>
	<code>bne \$at,\$0,loop</code>
<code>la \$a0, str</code>	<code>lui \$at,left(str)</code>
	<code>ori \$a0,\$at,right(str)</code>



## 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 "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



## 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
    - such as the `la` instruction



## Object File Format

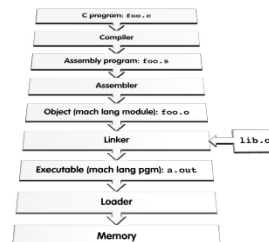
- object file header**: size and position of the other pieces of the object file
- text segment**: the machine code
- data segment**: binary representation of the data in the source file
- relocation information**: identifies lines of code that need to be "handled"
- symbol table**: list of this file's labels and data that can be referenced
- debugging information**

- A standard format is ELF (except MS)

[https://www.skyfree.org/linux/references/ELF\\_Format.pdf](https://www.skyfree.org/linux/references/ELF_Format.pdf)



## Where Are We Now?

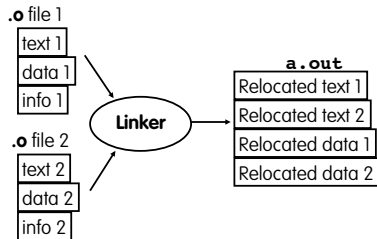


## Linker (1/3)

- Input: Object Code files, information tables (e.g., `foo.o`, `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 NT source was > 40 M lines of code!
  - Old name "Link Editor" from editing the "links" in jump and link instructions



### Linker (2/3)



### 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

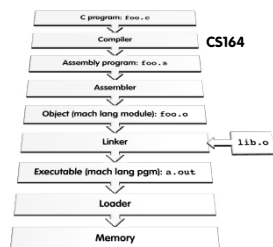


### Resolving References (2/2)

- 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)



### Where Are We Now?



### 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?

- 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
- 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
  - if main routine returns, start-up routine terminates program with the exit system call



### Conclusion

- 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)

```

    graph TD
      C[C program: foo.c] -->|Compiler| A[Assembly program: foo.s]
      A -->|Assembler| O[Object (mach lang module): foo.o]
      O -->|Linker| E[Executable (mach lang pgm): a.out]
      L[lib.o] --> E
      E -->|Loader| Mem[Memory]
  
```

- Compiler converts a single HLL file into a single assembly lang. 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.

Cal CMSC 154: Building a Program I ... Compiling, Assembling, Linking, and Loading 208 Oerlis, Spring 2014 © UCSD

### 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

Cal CMSC 154: Building a Program I ... Compiling, Assembling, Linking, and Loading 208 Oerlis, Spring 2014 © UCSD

### Language Execution Continuum

Easy to program / Inefficient to interpret ← Scheme Java C++ C Assembly Java bytecode machine language → Difficult to program / Efficient to interpret

- 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

Cal CMSC 154: Building a Program I ... Compiling, Assembling, Linking, and Loading 208 Oerlis, Spring 2014 © UCSD

### 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`

Cal CMSC 154: Building a Program I ... Compiling, Assembling, Linking, and Loading 208 Oerlis, Spring 2014 © UCSD

### Interpretation

- Any good reason to interpret machine language in software?
  - SPIM – 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)

Cal CMSC 154: Building a Program I ... Compiling, Assembling, Linking, and Loading 208 Oerlis, Spring 2014 © UCSD

### Interpretation vs. Translation? (1/2)

- Generally easier to write 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 slower (10x?), code smaller (2x?)
- Interpreter provides instruction set independence: run on any machine

Cal CMSC 154: Building a Program I ... Compiling, Assembling, Linking, and Loading 208 Oerlis, Spring 2014 © UCSD

### Interpretation vs. Translation? (2/2)

- 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 (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.

Cal CMSC 154: Building a Program I ... Compiling, Assembling, Linking, and Loading 208 Oerlis, Spring 2014 © UCSD

### 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

Cal CMSC 154: Building a Program I ... Compiling, Assembling, Linking, and Loading 208 Oerlis, Spring 2014 © UCSD

[en.wikipedia.org/wiki/Dynamic\\_linking](http://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.

Cal CMSC 154: Building a Program I ... Compiling, Assembling, Linking, and Loading 208 Oerlis, Spring 2014 © UCSD

## 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"
  - This isn't the only way to do it...



## 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,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:
        .ascii "The sum
        of sq from 0 ..
        100 is %d\n"
```



## 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:
        .ascii "The sum
        of sq from 0 ..
        100 is %d\n"
```



## 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, 1.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
2c sw $31, $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, 1.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 0010011101110111111111111100000
0x000004 10101111011111000000000010100
0x000008 101011110100100000000000100000
0x00000c 101011110100100000000000001000
0x000010 1010111101000000000000001000
0x000014 101011110100000000000000001100
0x000018 100011110101100000000000001100
0x00001c 100011110110000000000000001100
0x000020 000000001000110000000000101001
0x000024 00100101100100000000000000001
0x000028 0010100100000000000000001100101
0x00002c 0010111101010000000000001100
0x000030 0000000000000000001110000010010
0x000034 00000011000011111001000010001
0x000038 00010000010000011111111110111
0x00003c 1010111101100100000000001000
0x000040 00111000000010000000000000000
0x000044 1000111101001010000000000000000
0x000048 00011000010000000000001101100
0x00004c 00100100000000000000000000000
0x000050 1000111101111100000000000010100
0x000054 00100111011010000000000010000
0x000058 000000111100000000000000001000
0x00005c 00000000000000000010000010001
```



### 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: 0x100000430
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      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 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.



### 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 Lillipian 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 Little Endian
• ADDR3 ADDR2 ADDR1 ADDR0 • ADDR3 ADDR2 ADDR1 ADDR0
  BYTE0 BYTE1 BYTE2 BYTE3  BYTE3 BYTE2 BYTE1 BYTE0
  00000001 00000100 00000000 00000000  00000000 00000000 00000100 00000001
• ADDR0 ADDR1 ADDR2 ADDR3 • ADDR0 ADDR1 ADDR2 ADDR3
  BYTE3 BYTE2 BYTE1 BYTE0  BYTE3 BYTE2 BYTE1 BYTE0
  00000000 00000000 00000100 00000001  00000001 00000100 00000000 00000000
```

www.webopedia.com/TERM/b/big\_endian.html  
searchnetworking.techtarget.com/definition/0,,1467\_gci211659,,00.html  
www.novelltheory.com/TechPapers/endian.asp  
en.wikipedia.org/wiki/Big\_endian

