

**MK** ROGAN KAUFMANN  
COMPUTER ORGANIZATION AND DESIGN  
The Hardware/Software Interface  
5th Edition

## Chapter 2

### Instructions: Language of the Computer

**MK** ROGAN KAUFMANN  
COMPUTER ORGANIZATION AND DESIGN  
The Hardware/Software Interface  
5th Edition

## Instructions: Language of the Computer

- Introduction
- Operations of the Computer Hardware
- Translating and Starting a Program
- Operands of the Computer Hardware
- Signed and Unsigned Numbers
- Representing Instructions In the Computer
- Logical Operations
- Instructions for Making Decisions
- Communicating with People
- MIPS Addressing for 32-Bit immediates and Addresses
- Parallelism and Instructions: Synchronization
- A C Sort Example to Put It All Together
- Concluding Remarks

Chapter 2 — Instructions: Language of the Computer — 2

**MK** ROGAN KAUFMANN  
COMPUTER ORGANIZATION AND DESIGN  
The Hardware/Software Interface  
5th Edition

## Instruction Set

- The collection of instructions of a computer
- Different computers have different instruction sets
  - But with many aspects in common
- Early computers had very simple instruction sets
  - Simplified implementation
- Many modern computers also have simple instruction sets

Chapter 2 — Instructions: Language of the Computer — 3

**MK** ROGAN KAUFMANN  
COMPUTER ORGANIZATION AND DESIGN  
The Hardware/Software Interface  
5th Edition

## The MIPS Instruction Set

- Used as an example throughout the course
- Stanford MIPS commercialized by MIPS Technologies ([www.mips.com](http://www.mips.com))
- Large share of embedded core market
  - Applications in consumer electronics, network/storage equipment, cameras, printers, ...
- Typical of many modern ISAs
  - See MIPS Reference Data tear-out card, and Appendices B and E

Chapter 2 — Instructions: Language of the Computer — 4

**MK** ROGAN KAUFMANN  
COMPUTER ORGANIZATION AND DESIGN  
The Hardware/Software Interface  
5th Edition

## Arithmetic Operations

- Add and subtract, three operands
  - Two sources and one destination
  - add a,b,c # a gets b + c
- All arithmetic operations have this form
- **Design Principle 1: Simplicity favors regularity**
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost

Chapter 2 — Instructions: Language of the Computer — 5

**MK** ROGAN KAUFMANN  
COMPUTER ORGANIZATION AND DESIGN  
The Hardware/Software Interface  
5th Edition

## MIPS assembly language

| Category           | Instruction | Example              | Comment                             |
|--------------------|-------------|----------------------|-------------------------------------|
| Arithmetic         | add         | add \$s1,\$s2,\$s3   | \$s1 = \$s2 + \$s3                  |
|                    | addui       | addui \$s1,\$s2,\$s3 | \$s1 = \$s2 + \$s3                  |
|                    | addi        | addi \$s1,\$s2,\$s3  | \$s1 = \$s2 + \$s3                  |
|                    | and         | and \$s1,\$s2,\$s3   | \$s1 = \$s2 & \$s3                  |
|                    | andui       | andui \$s1,\$s2,\$s3 | \$s1 = (\$s2 & \$s3)                |
|                    | ori         | ori \$s1,\$s2,\$s3   | \$s1 = (\$s2   \$s3)                |
|                    | or          | or \$s1,\$s2,\$s3    | \$s1 = \$s2   \$s3                  |
|                    | nor         | nor \$s1,\$s2,\$s3   | \$s1 = (~\$s2 & \$s3)               |
|                    | andi        | andi \$s1,\$s2,\$s3  | \$s1 = (\$s2 & \$s3)                |
|                    | andiu       | andiu \$s1,\$s2,\$s3 | \$s1 = (~\$s2 & \$s3)               |
| Data transfer      | lwr         | lwr \$s1,\$s2,\$s3   | Word from register to register      |
|                    | lwl         | lwl \$s1,\$s2,\$s3   | Word from register to register      |
|                    | lbu         | lbu \$s1,\$s2,\$s3   | Byte from register to register      |
|                    | lbu         | lbu \$s1,\$s2,\$s3   | Byte from register to register      |
|                    | lbu         | lbu \$s1,\$s2,\$s3   | Byte from register to register      |
|                    | lbu         | lbu \$s1,\$s2,\$s3   | Byte from register to register      |
|                    | lbu         | lbu \$s1,\$s2,\$s3   | Byte from register to register      |
|                    | lbu         | lbu \$s1,\$s2,\$s3   | Byte from register to register      |
|                    | lbu         | lbu \$s1,\$s2,\$s3   | Byte from register to register      |
|                    | lbu         | lbu \$s1,\$s2,\$s3   | Byte from register to register      |
| Logical            | and         | and \$s1,\$s2,\$s3   | Three reg. operands; bit-by-bit AND |
|                    | andi        | andi \$s1,\$s2,\$s3  | Three reg. operands; bit-by-bit AND |
|                    | andiu       | andiu \$s1,\$s2,\$s3 | Three reg. operands; bit-by-bit AND |
|                    | ori         | ori \$s1,\$s2,\$s3   | Three reg. operands; bit-by-bit OR  |
|                    | ori         | ori \$s1,\$s2,\$s3   | Three reg. operands; bit-by-bit OR  |
|                    | ori         | ori \$s1,\$s2,\$s3   | Three reg. operands; bit-by-bit OR  |
|                    | ori         | ori \$s1,\$s2,\$s3   | Three reg. operands; bit-by-bit OR  |
|                    | ori         | ori \$s1,\$s2,\$s3   | Three reg. operands; bit-by-bit OR  |
|                    | ori         | ori \$s1,\$s2,\$s3   | Three reg. operands; bit-by-bit OR  |
|                    | ori         | ori \$s1,\$s2,\$s3   | Three reg. operands; bit-by-bit OR  |
| Conditional branch | beq         | beq \$s1,\$s2,\$s3   | Branch if equal                     |
|                    | bne         | bne \$s1,\$s2,\$s3   | Branch if not equal                 |
|                    | ble         | ble \$s1,\$s2,\$s3   | Branch if less than or equal        |
|                    | bgt         | bgt \$s1,\$s2,\$s3   | Branch if greater than              |
|                    | ble         | ble \$s1,\$s2,\$s3   | Branch if less than or equal        |
|                    | bgt         | bgt \$s1,\$s2,\$s3   | Branch if greater than              |
|                    | ble         | ble \$s1,\$s2,\$s3   | Branch if less than or equal        |
|                    | bgt         | bgt \$s1,\$s2,\$s3   | Branch if greater than              |
|                    | ble         | ble \$s1,\$s2,\$s3   | Branch if less than or equal        |
|                    | bgt         | bgt \$s1,\$s2,\$s3   | Branch if greater than              |
| Unconditional jump | jalr        | jalr \$s1,\$s2,\$s3  | Jump to register                    |
|                    | jalr        | jalr \$s1,\$s2,\$s3  | Jump to register                    |
|                    | jalr        | jalr \$s1,\$s2,\$s3  | Jump to register                    |
|                    | jalr        | jalr \$s1,\$s2,\$s3  | Jump to register                    |
|                    | jalr        | jalr \$s1,\$s2,\$s3  | Jump to register                    |
|                    | jalr        | jalr \$s1,\$s2,\$s3  | Jump to register                    |
|                    | jalr        | jalr \$s1,\$s2,\$s3  | Jump to register                    |
|                    | jalr        | jalr \$s1,\$s2,\$s3  | Jump to register                    |
|                    | jalr        | jalr \$s1,\$s2,\$s3  | Jump to register                    |
|                    | jalr        | jalr \$s1,\$s2,\$s3  | Jump to register                    |

Chapter 2 — Instructions: Language of the Computer — 6

## Arithmetic Example

- C code:
 

```
f = (g + h) - (i + j);
```
- Compiled MIPS code:
 

```
add t0, g, h    # temp t0 = g + h
add t1, i, j    # temp t1 = i + j
sub f, t0, t1   # f = t0 - t1
```

## Register Operands

- Arithmetic instructions use register operands
- MIPS has a 32 by 32-bit register file
  - Used for frequently accessed data
  - Numbered 0 to 31
  - 32-bit data called a “word”
- Assembler names
  - \$t0, \$t1, ..., \$t9 for temporary values
  - \$s0, \$s1, ..., \$s7 for saved variables
- Design Principle 2: Smaller is faster*
  - c.f. main memory: millions of locations

## Register Operand Example

- C code:
 

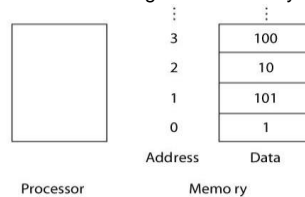
```
f = (g + h) - (i + j);
```

  - f, ..., j in \$s0, ..., \$s4
- Compiled MIPS code:
 

```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```

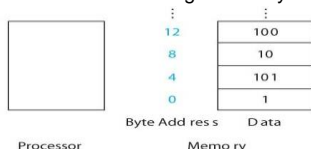
## Memory Operands

- Main memory used for composite data
  - Arrays, structures, dynamic data
- To apply arithmetic operations
  - Load values from memory into registers
  - Store result from register to memory



## Memory Operands

- Memory is byte addressed
  - Each address identifies an 8-bit byte
- Words are aligned in memory
  - Address must be a multiple of 4
- MIPS is Big Endian
  - Most-significant byte at least address of a word
  - c.f. Little Endian: least-significant byte at least address



## Memory Operands

- Data is transferred between memory and register using data transfer instructions: lw and sw

| Category      | Instruction | Example            | Meaning                 | Comments           |
|---------------|-------------|--------------------|-------------------------|--------------------|
| Data transfer | load word   | lw \$s1, 100(\$s2) | \$s1 ← memory[\$s2+100] | Memory to Register |
|               | store word  | sw \$s1, 100(\$s2) | memory[\$s2+100] ← \$s1 | Register to memory |

- \$s1 is receiving register
- \$s2 is base address of memory, 100 is called the offset, so (\$s2+100) is the address of memory location

## Memory Operand Example

- C code:
  - g = h + A[8];
    - g in \$s1, h in \$s2, base address of A in \$s3
- Compiled MIPS code:
  - Index 8 requires offset of 32
    - 4 bytes per word

```
lw $t0, 32($s3) # load word
add $s1, $s2, $t0
```

offset

base register



## Memory Operand Example

- C code:
  - A[12] = h + A[8];
    - h in \$s2, base address of A in \$s3
- Compiled MIPS code:
  - Index 8 requires offset of 32

```
lw $t0, 32($s3) # load word
add $t0, $s2, $t0
sw $t0, 48($s3) # store word
```



## Registers vs. Memory

- Registers are faster to access than memory
- Operating on memory data requires loads and stores
  - More instructions to be executed
- Compiler must use registers for variables as much as possible
  - Only spill to memory for less frequently used variables
  - Register optimization is important!



## Immediate Operands

- Constant data specified in an instruction
  - addi \$s3, \$s3, 4
- No subtract immediate instruction
  - Just use a negative constant
    - addi \$s2, \$s1, -1
- *Design Principle 3: Make the common case fast*
  - Small constants are common
  - Immediate operand avoids a load instruction

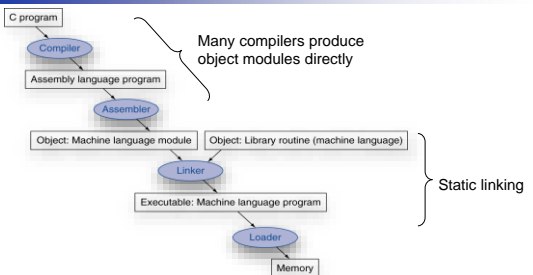


## The Constant Zero

- MIPS register 0 (\$zero) is the constant 0
  - Cannot be overwritten
- Useful for common operations
  - E.g., move between registers
    - add \$t2, \$s1, \$zero



## Translation and Startup



**UNIX:** C source files are named x.c, assembly files are x.s, object files are named x.o, statically linked library routines are x.a, dynamically linked library routines are x.so, and executable files by default are called a.out  
**MS-DOS** uses the .C, .ASM, .OBJ, .LIB, .DLL, and .EXE to the same effect



## Translation

- Assembler (or compiler) translates program into machine instructions
- Linker produces an executable image
- Loader loads from image file on disk into memory

§2.12 Translating and Starting a Program



## SPIM Simulator

- SPIM is a software simulator that runs assembly language programs
- SPIM is just MIPS spelled backwards
- SPIM can read and immediately execute assembly language files
- Two versions for different machines
  - Unix: xspim(used in lab), spim
  - PC/Mac: QtSpim
- Resources and Download
  - <http://spimsimulator.sourceforge.net>

§2.12 Translating and Starting a Program



## System Calls in SPIM

- SPIM provides a small set of system-like services through the system call (syscall) instruction.
- Format for system calls
  - Place value of input argument in \$a0
  - Place value of system-call-code in \$v0
  - Syscall

§2.12 Translating and Starting a Program



## System Calls

Example: print a string

```
.data
str: # -----
.asciiz "answer is:"

.text
addi $v0,$zero,4
la $a0, str
syscall
```

| Service         | System Call Code | Arguments  | Result                    |
|-----------------|------------------|--|---------------------------|
| print_int       | 1                | \$a0 = integer   |                           |
| print_float     | 2                | \$f12 = float  |                           |
| print_double    | 3                | \$f12 = double   |                           |
| print_string    | 4                | \$a0 = string  |                           |
| read_int        | 5                |  | integer (in \$v0)         |
| read_float      | 6                |  | float (in \$f0)           |
| read_double     | 7                |  | double (in \$f0)          |
| read_string     | 8                | \$a0 = buffer, \$a1 = length                           |                           |
| srk             | 9                | \$a0 = amount  | address (in \$v0)         |
| exit            | 10               |  |                           |
| print_character | 11               | \$a0 = character                                       |                           |
| read_character  | 12               |  | character (in \$v0)       |
| open            | 13               | \$a0 = filename,<br>\$a1 = flags, \$a2 = mode          | file descriptor (in \$v0) |
| read            | 14               | \$a0 = file descriptor,<br>\$a1 = buffer, \$a2 = count | bytes read (in \$v0)      |
| write           | 15               | \$a0 = file descriptor,<br>\$a1 = buffer, \$a2 = count | bytes written (in \$v0)   |
| close           | 16               | \$a0 = file descriptor                                 | 0 (in \$v0)               |
| exit2           | 17               | \$a0 = value   |                           |

§2.12 Translating and Starting a Program



## Assembler Pseudoinstructions

- Most assembler instructions represent machine instructions one-to-one
- Pseudoinstructions: figments of the assembler's imagination
  - move \$t0, \$t1 → add \$t0, \$zero, \$t1
  - blt \$t0, \$t1, L → slt \$at, \$t0, \$t1  
bne \$at, \$zero, L
  - \$at (Register 1): assembler temporary

§2.12 Translating and Starting a Program



## Assembler Pseudoinstructions

- Pseudoinstructions give MIPS a richer set of assembly language instructions than those implemented by the hardware
- Register \$at (assembler temporary) reserved for use by the assembler
- For productivity, use pseudoinstructions to write assembly programs
- For performance, use real MIPS instructions

§2.12 Translating and Starting a Program



## Reading

- Read Appendix A.9 for SPIM
- List of Pseudoinstructions can be found on page 235 of book

## Producing an Object Module

- Assembler (or compiler) translates program into machine instructions
- Provides information for building a complete program from pieces
  - Header: contains size and position of pieces of object module
  - Text segment: translated machine instructions
  - Static data segment: data allocated for the life of the program
  - Relocation info: for instructions and data words that depend on absolute location of loaded program
  - Symbol table: global definitions and external refs
  - Debug info: for associating with source code

## Linking Object Modules

- Produces an executable file
  - Merges segments
  - Resolves labels (determines their addresses)
  - Patches location-dependent and external refs
- Could leave location dependencies for fixing by a relocating loader
  - But with virtual memory, no need to do this
  - Program can be loaded into absolute location in virtual memory space

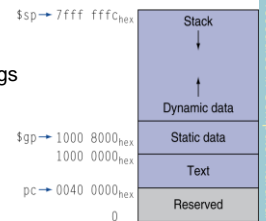
| Object file header     |           |                    |            |
|------------------------|-----------|--------------------|------------|
|                        | Name      | Procedure A        |            |
|                        | Text size | 100 <sub>hex</sub> |            |
|                        | Data size | 20 <sub>hex</sub>  |            |
| Text segment           | Address   | Instruction        |            |
|                        | 0         | lw \$a0, 0(\$gp)   |            |
|                        | 4         | jal 0              |            |
|                        | ...       | ...                |            |
|                        | 0         | (X)                |            |
|                        | ...       | ...                |            |
| Data segment           |           |                    |            |
|                        | 0         | (Y)                |            |
|                        | ...       | ...                |            |
| Relocation information | Address   | Instruction type   | Dependency |
|                        | 0         | lw                 | X          |
|                        | 4         | jal                | B          |
| Symbol table           | Label     | Address            |            |
|                        | X         | -                  |            |
|                        | B         | -                  |            |
| Object file header     |           |                    |            |
|                        | Name      | Procedure B        |            |
|                        | Text size | 200 <sub>hex</sub> |            |
|                        | Data size | 30 <sub>hex</sub>  |            |
| Text segment           | Address   | Instruction        |            |
|                        | 0         | sw \$a1, 0(\$gp)   |            |
|                        | 4         | jal 0              |            |
|                        | ...       | ...                |            |
|                        | 0         | (Y)                |            |
|                        | ...       | ...                |            |
| Relocation information | Address   | Instruction type   | Dependency |
|                        | 0         | sw                 | Y          |
|                        | 4         | jal                | A          |
| Symbol table           | Label     | Address            |            |
|                        | Y         | -                  |            |
|                        | A         | -                  |            |

## Linking Object Modules

| Executable file header |                          |                                     |
|------------------------|--------------------------|-------------------------------------|
|                        | Text size                | 300 <sub>hex</sub>                  |
|                        | Data size                | 50 <sub>hex</sub>                   |
| Text segment           | Address                  | Instruction                         |
|                        | 0040 0000 <sub>hex</sub> | lw \$a0, 8000 <sub>hex</sub> (\$gp) |
|                        | 0040 0004 <sub>hex</sub> | jal 40 0100 <sub>hex</sub>          |
|                        | ...                      | ...                                 |
|                        | 0040 0100 <sub>hex</sub> | sw \$a1, 8020 <sub>hex</sub> (\$gp) |
|                        | 0040 0104 <sub>hex</sub> | jal 40 0000 <sub>hex</sub>          |
|                        | ...                      | ...                                 |
| Data segment           | Address                  |                                     |
|                        | 1000 0000 <sub>hex</sub> | (X)                                 |
|                        | ...                      | ...                                 |
|                        | 1000 0020 <sub>hex</sub> | (Y)                                 |
|                        | ...                      | ...                                 |

## Memory Layout

- Text: program code
- Static data: global variables
  - e.g., static variables in C, constant arrays and strings
  - \$gp initialized to address allowing ±offsets into this segment
- Dynamic data: heap
  - e.g., malloc in C, new in Java
- Stack: automatic storage



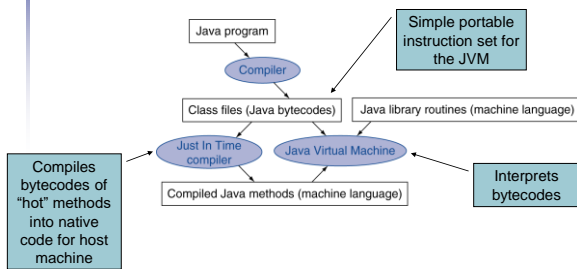
## Loading a Program

- Load from file on disk into memory
  1. Read header to determine segment sizes
  2. Create address space for text and data
  3. Copy text and initialized data into memory
  4. Set up arguments on stack
  5. Initialize registers (including \$sp, \$fp, \$gp)
  6. Jump to startup routine
    - Copies arguments to \$a0, ... and calls main
    - When main returns, do exit syscall(10)

## Dynamic Linking

- Only link/load library procedure when it is called
  - Requires procedure code to be relocatable
  - Avoids image enlarge caused by static linking of all (transitively) referenced libraries
  - Automatically picks up new library versions

## Starting Java Applications



## An Example MIPS Program

```
# Program: (descriptive name)          Programmer: NAME
# Due Date:                            Course: CSE 2021
# Functional Description: Find the sum of the integers from 1 to N where
# N is a value input from the keyboard.
#####
# Register Usage: $t0 is used to accumulate the sum
#                 $v0 the loop counter, counts down to zero
#####
# Algorithmic Description in Pseudocode:
# main:  v0 << value read from the keyboard (syscall 5)
#        if (v0 <= 0) stop
#        t0 = 0;           # t0 is used to accumulate the sum
#        While (v0 > 0) { t0 = t0 + v0; v0 = v0 - 1 }
#        Output to monitor syscall(1) << t0;   goto main
#####
.data
prompt: .asciiz      "\n\n Please Input a value for N = "
result: .asciiz      " The sum of the integers from 1 to N is "
bye:    .asciiz      "\n **** Have a good day **** "
main:   .globl
main
```

## An Example MIPS Program

```
.text
main:
li      $v0, 4           # system call code for print_str
la      $a0, prompt     # load address of prompt into a0
syscall # print the prompt message
li      $v0, 5           # system call code for read int
syscall # reads a value of N into v0
blez   $v0, done       # if (v0 <= 0) go to done
li      $t0, 0          # clear $t0 to zero
loop:
add     $t0, $t0, $v0   # sum of integers in register $t0
addi   $v0, $v0, -1    # summing in reverse order
bnez   $v0, loop       # branch to loop if $v0 is != zero
li      $v0, 4           # system call code for print_str
la      $a0, result     # load address of message into $a0
syscall # print the string
li      $v0, 1           # system call code for print_int
move    $a0, $t0        # a0 = $t0
syscall # prints the value in register $a0
b       main
done:
li      $v0, 4           # system call code for print_str
la      $a0, bye        # load address of msg. into $a0
syscall # print the string
li      $v0, 10         # terminate program
syscall # return control to system
```

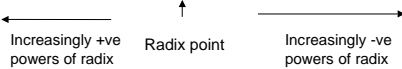
## Four Important Number Systems

| System  | Why?   | Remarks                            |
|---------|--|------------------------------------|
| Decimal | Base 10: (10 fingers)                              | Most used system                   |
| Binary  | Base 2: On/Off systems                             | 2-4 times more digits than decimal |
| Octal   | Base 8: Shorthand notation for working with binary | 3 times less digits than binary    |
| Hex     | Base 16  | 4 times less digits than binary    |

## Positional Number Systems

- Have a radix  $r$  (base) associated with them.
- In the decimal system,  $r = 10$ :
  - Ten symbols: 0, 1, 2, ..., 8, and 9
  - More than 9 move to next position, so each position is power of 10
  - Nothing special about base 10 (used because we have 10 fingers)
- What does  $642.391_{10}$  mean?

$$6 \times 10^2 + 4 \times 10^1 + 2 \times 10^0 + 3 \times 10^{-1} + 9 \times 10^{-2} + 1 \times 10^{-3}$$



## Positional Number Systems

- What does  $642.391_{10}$  mean?

|                          |   |                |               |                    |                     |                      |
|--------------------------|---|----------------|---------------|--------------------|---------------------|----------------------|
|                          | Radix point                                     |                |               |                    |                     |                      |
|                          |   |                |               | ↓                  |                     |                      |
| Base 10<br>( $r$ )       | $10^2$<br>(100)                                 | $10^1$<br>(10) | $10^0$<br>(1) | $10^{-1}$<br>(0.1) | $10^{-2}$<br>(0.01) | $10^{-3}$<br>(0.001) |
| Coefficient<br>( $a_i$ ) | 6   | 4              | 2             | 3                  | 9                   | 1                    |
| Product: $a_i r^i$       | 600   | 40             | 2             | 0.3                | 0.09                | 0.001                |
| Value                    | $= 600 + 40 + 2 + 0.3 + 0.09 + 0.001 = 642.391$ |                |               |                    |                     |                      |

- Multiply each digit by appropriate power of 10 and add them together

- In general: 
$$\sum_{i=-m}^n a_i \times r^i$$



## Positional Number Systems

| Number system | Radix | Symbols                           |
|---------------|-------|-----------------------------------|
| Binary        | 2     | {0,1}                             |
| Octal         | 8     | {0,1,2,3,4,5,6,7}                 |
| Decimal       | 10    | {0,1,2,3,4,5,6,7,8,9}             |
| Hexadecimal   | 16    | {0,1,2,3,4,5,6,7,8,9,a,b,c,d,e,f} |



## Binary Number System

| Decimal | Binary | Decimal | Binary |
|---------|--------|---------|--------|
| 0       | 0000   | 8       | 1000   |
| 1       | 0001   | 9       | 1001   |
| 2       | 0010   | 10      | 1010   |
| 3       | 0011   | 11      | 1011   |
| 4       | 0100   | 12      | 1100   |
| 5       | 0101   | 13      | 1101   |
| 6       | 0110   | 14      | 1110   |
| 7       | 0111   | 15      | 1111   |



## Octal Number System

| Decimal | Octal | Decimal | Octal |
|---------|-------|---------|-------|
| 0       | 0     | 8       | 10    |
| 1       | 1     | 9       | 11    |
| 2       | 2     | 10      | 12    |
| 3       | 3     | 11      | 13    |
| 4       | 4     | 12      | 14    |
| 5       | 5     | 13      | 15    |
| 6       | 6     | 14      | 16    |
| 7       | 7     | 15      | 17    |



## Hexadecimal Number System

| Decimal | Hex | Decimal | Hex |
|---------|-----|---------|-----|
| 0       | 0   | 8       | 8   |
| 1       | 1   | 9       | 9   |
| 2       | 2   | 10      | A   |
| 3       | 3   | 11      | B   |
| 4       | 4   | 12      | C   |
| 5       | 5   | 13      | D   |
| 6       | 6   | 14      | E   |
| 7       | 7   | 15      | F   |



## Four Number Systems

| Decimal | Binary | Octal | Hex | Decimal | Binary | Octal | Hex |
|---------|--------|-------|-----|---------|--------|-------|-----|
| 0       | 0000   | 0     | 0   | 8       | 1000   | 10    | 8   |
| 1       | 0001   | 1     | 1   | 9       | 1001   | 11    | 9   |
| 2       | 0010   | 2     | 2   | 10      | 1010   | 12    | A   |
| 3       | 0011   | 3     | 3   | 11      | 1011   | 13    | B   |
| 4       | 0100   | 4     | 4   | 12      | 1100   | 14    | C   |
| 5       | 0101   | 5     | 5   | 13      | 1101   | 15    | D   |
| 6       | 0110   | 6     | 6   | 14      | 1110   | 16    | E   |
| 7       | 0111   | 7     | 7   | 15      | 1111   | 17    | F   |

## Conversion: Binary to Decimal

Binary  $\longrightarrow$  Decimal

$1101.011_2 \longrightarrow (??)_{10}$

| $r^i$  | $2^3(8)$ | $2^2(4)$ | $2^1(2)$ | $2^0(1)$ | $2^{-1}(0.5)$ | $2^{-2}(0.25)$ | $2^{-3}(0.125)$ |
|--|----------|----------|----------|----------|---------------|----------------|-----------------|
| $a_j$  | 1        | 1        | 0        | 1        | 0             | 1              | 1               |
| $a_j r^j$  | 8        | 4        | 0        | 1        | 0             | 0.25           | 0.125           |
| $(1101.011)_2 = 8 + 4 + 1 + 0.25 + 0.125 = 13.375$ |          |          |          |          |               |                |                 |

$$1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3} = 13.375_{10}$$

Binary point

## Conversion: Decimal to Binary

- A decimal number can be converted to binary by repeated division by 2 if it is an integer

| number | $\div 2$ | Remainder |                             |
|--------|----------|-----------|-----------------------------|
| 155    | 77       | 1         | Least Significant Bit (LSB) |
| 77     | 38       | 1         |                             |
| 38     | 19       | 0         |                             |
| 19     | 9        | 1         |                             |
| 9      | 4        | 1         |                             |
| 4      | 2        | 0         |                             |
| 2      | 1        | 0         |                             |
| 1      | 0        | 1         |                             |

Arrange remainders in reverse order

$155_{10} = 10011011_2$

## Conversion: Decimal to Binary

- If the number includes a radix point, it is necessary to separate the number into an integer part and a fraction part, each part must be converted separately.

Decimal  $\longrightarrow$  Binary

$(27.375)_{10} \longrightarrow (??)_2$

| number | $\div 2$ | Remainder |                             |
|--------|----------|-----------|-----------------------------|
| 27     | 13       | 1         | Least Significant Bit (LSB) |
| 13     | 6        | 1         |                             |
| 6      | 3        | 0         |                             |
| 3      | 1        | 1         |                             |
| 1      | 0        | 1         |                             |

Arrange remainders in reverse order: 11011

$\Rightarrow 27.375_{10} = 11011.011_2$

| number | $\times 2$ | Integer |                             |
|--------|------------|---------|-----------------------------|
| 0.375  | 0.75       | 0       | Least Significant Bit (LSB) |
| 0.75   | 1.50       | 1       |                             |
| 0.50   | 1.0        | 1       |                             |

Arrange in order: 011

Arrange remainders in reverse order: 11011

$$\Rightarrow 27.375_{10} = 11011.011_2$$

## Conversion: Octal to Binary

Octal  $\longrightarrow$  Binary

$345.5602_8 \longrightarrow (???)_2$

3 4 5 . 5 6 0 2

011 100 101 . 101 110 000 010

Discard leading zero(s)

Discard trailing zero(s)

$$345.5602_8 = 11100101.1011100001_2$$

## Conversion: Binary to Octal

Binary  $\longrightarrow$  Octal

$11001110.0101101_2 \longrightarrow (??)_8$

Add leading zero(s)

Add trailing zero(s)

011 001110 . 010110100

3 1 6 . 2 6 4

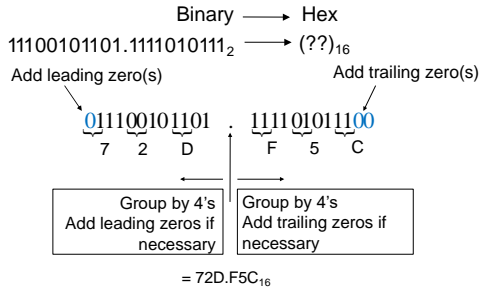
Group by 3's  
Add leading zeros if necessary

Group by 3's  
Add trailing zeros if necessary

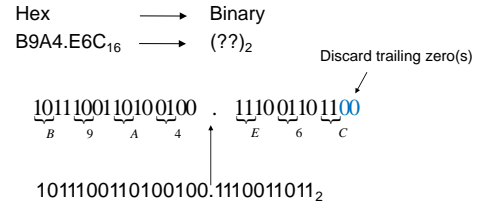
$$11001110.0101101_2 = 316.264_8$$



## Conversion: Binary to Hex



## Conversion: Hex to Binary



## Conversion: Hex to Decimal

Hex  $\longrightarrow$  Decimal  
 $B63.4C_{16} \longrightarrow (??)_{10}$

| $16^2$  | $16^1$ | $16^0$ | $16^{-1}$ | $16^{-2}$ |
|---|--------|--------|-----------|-----------|
| B (=11)   | 6      | 3      | 4         | C (=12)   |
| $= 2816 + 96 + 3 + 0.25 + 0.046875 = 2915.296875$ |        |        |           |           |

$$11 \times 16^2 + 6 \times 16^1 + 3 \times 16^0 + 4 \times 16^{-1} + 12 \times 16^{-2} = (2915.296875)_{10}$$



## Conversion: Activity 1

- Convert the hexadecimal number A59.FCE to binary

$1010 \ 0101 \ 1001 \cdot 1111 \ 1100 \ 1110$

- Convert the decimal number 166.34 into binary

$$\begin{array}{r} 83 \\ 2 \overline{)166} \leftarrow 2 \quad 41 \\ \underline{166} \\ 0 \end{array} \quad \begin{array}{r} 41 \\ 2 \overline{)83} \leftarrow 2 \quad 20 \\ \underline{82} \\ 1 \end{array} \quad \begin{array}{r} 20 \\ 2 \overline{)41} \leftarrow 2 \quad 20 \\ \underline{40} \\ 1 \end{array} \quad \begin{array}{r} 10 \\ 2 \overline{)20} \leftarrow 2 \quad 10 \\ \underline{20} \\ 0 \end{array} \quad \begin{array}{r} 5 \\ 2 \overline{)10} \leftarrow 2 \quad 5 \\ \underline{10} \\ 0 \end{array} \quad \begin{array}{r} 2 \\ 2 \overline{)5} \leftarrow 2 \quad 2 \\ \underline{4} \\ 1 \end{array} \quad \begin{array}{r} 1 \\ 2 \overline{)2} \leftarrow 2 \quad 1 \\ \underline{2} \\ 0 \end{array} \quad \begin{array}{r} 0 \\ 2 \overline{)1} \leftarrow 2 \quad 0 \\ \underline{0} \\ 1 \end{array}$$

$$.34 \times 2 = 0.68 \rightarrow .68 \times 2 = 1.36 \rightarrow .36 \times 2 = 0.72 \rightarrow .72 \times 2 = 1.44 \dots$$

$$(A59.FCE)_{16} = (10100110.0101\dots)_2$$



## Binary Numbers

- How many distinct numbers can be represented by  $n$  bits?

| No. of bits | Distinct nos.                              |
|-------------|--|
| 1           | 2 {0,1}                                    |
| 2           | 4 {00, 01, 10, 11}                         |
| 3           | 8 {000, 001, 010, 011, 100, 101, 110, 111} |
| ...         | ...  |
| $n$         | $2^n$                                      |

- Number of permutations double with every extra bit
- $2^n$  unique numbers can be represented by  $n$  bits



## Number System and Computers

- Some tips
  - Binary numbers often grouped in fours for easy reading
  - 1 byte=8-bit, 1 word = 4-byte (32 bits)
  - Decimal is assumed in computer programs (e.g. Verilog, C) by default
  - To represent other number bases, use

| System      | Representation       | Example for 20 |
|-------------|----------------------|----------------|
| Hexadecimal | 0x...                | 0x14           |
| Binary      | 0b...                | 0b10100        |
| Octal       | 0o... (zero and 'O') | 0o24           |



## Number System and Computers

- Addresses often written in Hex
  - Most compact representation
  - Easy to understand given their hardware structure
  - For a range 0x000 – 0xFFFF, we can immediately see that 12 bits are needed, 4K locations
  - Tip: 10 bits = 1K



## Negative Number Representation

- Three kinds of representations are common:
  - Signed Magnitude (SM)
  - One's Complement
  - Two's Complement



## Signed Magnitude Representation

[0,1] {.....}

Sign bit (left most)      (n-1) magnitude bits

- 0 indicates +ve
- 1 indicates -ve

8 bit representation for +13 is      0 0001101

8 bit representation for -13 is      1 0001101



## 1's Complement Notation

Let  $N$  be an  $n$ -bit number and  $\tilde{N}(1)$  be the 1's Complement of the number. Then,

$$\tilde{N}(1) = 2^n - 1 - |N|$$

- The idea is to leave positive numbers as is, but to represent negative numbers by the 1's Complement of their magnitude.
- Example: Let  $n = 4$ . What is the 1's Complement representation for +6 and -6?
  - +6 is represented as 0110 (as usual in binary)
  - 6 is represented by 1's complement of its magnitude (6)



## 1's Complement Notation

- 1's C representation can be computed in 2 ways:
  - Method 1:** 1's C representation of -6 is:
 
$$2^4 - 1 - |M| = (16 - 1 - 6)_{10} = (9)_{10} = (1001)_2$$
  - Method 2:** For -6, the magnitude = 6 =  $(0110)_2$ 
    - The 1's C representation is obtained by complementing the bits of the magnitude:  $(1001)_2$



## 2's Complement Notation

Let  $N$  be an  $n$  bit number and  $\tilde{N}(2)$  be the 2's Complement of the number. Then,

$$\tilde{N}(2) = 2^n - |N|$$

- Again, the idea is to leave positive numbers as is, but to represent negative numbers by the 2's C of their magnitude.
- Example: Let  $n = 5$ . What is 2's C representation for +11 and -13?
  - +11 is represented as 01011 (as usual in binary)
  - 13 is represented by 2's complement of its magnitude (13)



## 2's Complement Notation

- 2's C representation can be computed in 2 ways:

**Method 1:** 2's C representation of -13 is  
 $2^5 - |N| = (32 - 13)_{10} = (19)_{10} = (10011)_2$

**Method 2:** For -13, the magnitude is  
 $13 = (01101)_2$

- The 2's C representation is obtained by adding 1 to the 1's C of the magnitude
- $2^5 - |N| = (2^5 - 1 - |N|) + 1 = 1's\ C + 1$

$01101 \xrightarrow{1's\ C} 10010 \xrightarrow{add\ 1} 10011$



## Comparing All Signed Notations

| 4-bit No. | SM | 1's C | 2's C |
|-----------|----|-------|-------|
| 0000      | +0 | +0    | 0     |
| 0001      | 1  | 1     | 1     |
| 0010      | 2  | 2     | 2     |
| 0011      | 3  | 3     | 3     |
| 0100      | 4  | 4     | 4     |
| 0101      | 5  | 5     | 5     |
| 0110      | 6  | 6     | 6     |
| 0111      | 7  | 7     | 7     |
| 1000      | -0 | -7    | -8    |
| 1001      | -1 | -6    | -7    |
| 1010      | -2 | -5    | -6    |
| 1011      | -3 | -4    | -5    |
| 1100      | -4 | -3    | -4    |
| 1101      | -5 | -2    | -3    |
| 1110      | -6 | -1    | -2    |
| 1111      | -7 | -0    | -1    |

- In all 3 representations, a -ve number has a 1 in MSB location
- To handle -ve numbers using  $n$  bits,
  - $= 2^{n-1}$  symbols can be used for positive numbers
  - $= 2^{n-1}$  symbols can be used for negative numbers
- In 2's C notation, only 1 combination used for 0



## Unsigned Binary Integers

- Given an  $n$ -bit number

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \dots + x_12^1 + x_02^0$$

- Range: 0 to  $+2^n - 1$
- Example
  - 0000 0000 0000 0000 0000 0000 1011<sub>2</sub>  
 $= 0 + \dots + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$   
 $= 0 + \dots + 8 + 0 + 2 + 1 = 11_{10}$
- Using 32 bits
  - 0 to +4,294,967,295



## 2's-Complement Signed Integers

- Given an  $n$ -bit number

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \dots + x_12^1 + x_02^0$$

- Range:  $-2^{n-1}$  to  $+2^{n-1} - 1$
- Example
  - 1111 1111 1111 1111 1111 1111 1100<sub>2</sub>  
 $= -1 \times 2^{31} + 1 \times 2^{30} + \dots + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$   
 $= -2,147,483,648 + 2,147,483,644 = -4_{10}$
- Using 32 bits
  - 2,147,483,648 to +2,147,483,647



## 2's-Complement Signed Integers

- Bit 31 is sign bit
  - 1 for negative numbers
  - 0 for non-negative numbers
- Non-negative numbers have the same unsigned and 2's-complement representation
- Some specific numbers
  - 0: 0000 0000 ... 0000
  - 1: 1111 1111 ... 1111
  - Most-negative: 1000 0000 ... 0000
  - Most-positive: 0111 1111 ... 1111



## Signed Negation

- Complement and add 1
  - Complement means  $1 \rightarrow 0, 0 \rightarrow 1$

$$\overline{x} + x = 1111 \dots 1111_2 = -1$$

$$\overline{x} + 1 = -x$$

- Example: negate +2
  - +2 = 0000 0000 ... 0010<sub>2</sub>
  - 2 = 1111 1111 ... 1101<sub>2</sub> + 1  
 $= 1111 1111 \dots 1110_2$



## Sign Extension

- Representing a number using more bits
  - preserve the numeric value
- In MIPS instruction set
  - addi: extend immediate value
  - lb, lh: extend loaded byte/halfword
  - beq, bne: extend the displacement
- Replicate the sign bit to the left
  - c.f. unsigned values: extend with 0s
- Examples: 8-bit to 16-bit
  - +2: 0000 0010 => 0000 0000 0000 0010
  - 2: 1111 1110 => 1111 1111 1111 1110



## Representing Instructions

- Instructions are encoded in binary
  - Called machine code
- MIPS instructions
  - Encoded as 32-bit instruction words
  - Small number of formats encoding operation code (opcode), register numbers, ...
  - Regularity!
- Register numbers
  - \$t0 – \$t7 are reg's 8 – 15
  - \$t8 – \$t9 are reg's 24 – 25
  - \$s0 – \$s7 are reg's 16 – 23



## MIPS R-format Instructions

|        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|
| op     | rs     | rt     | rd     | shamt  | funct  |
| 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |

- Instruction fields
  - op: operation code (opcode)
  - rs: first source register number
  - rt: second source register number
  - rd: destination register number
  - shamt: shift amount (00000 for now)
  - funct: function code (extends opcode)



## R-format Example

|        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|
| op     | rs     | rt     | rd     | shamt  | funct  |
| 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |

add \$t0, \$s1, \$s2

|         |      |      |      |   |     |
|---------|------|------|------|---|-----|
| special | \$s1 | \$s2 | \$t0 | 0 | add |
|---------|------|------|------|---|-----|

|   |    |    |   |   |    |
|---|----|----|---|---|----|
| 0 | 17 | 18 | 8 | 0 | 32 |
|---|----|----|---|---|----|

|        |       |       |       |       |        |
|--------|-------|-------|-------|-------|--------|
| 000000 | 10001 | 10010 | 01000 | 00000 | 100000 |
|--------|-------|-------|-------|-------|--------|

00000010001100100100000000100000<sub>2</sub> = 02324020<sub>16</sub>



## Hexadecimal

- Base 16
  - Compact representation of bit strings
  - 4 bits per hex digit

|   |      |   |      |   |      |   |      |
|---|------|---|------|---|------|---|------|
| 0 | 0000 | 4 | 0100 | 8 | 1000 | c | 1100 |
| 1 | 0001 | 5 | 0101 | 9 | 1001 | d | 1101 |
| 2 | 0010 | 6 | 0110 | a | 1010 | e | 1110 |
| 3 | 0011 | 7 | 0111 | b | 1011 | f | 1111 |

- Example: eca8 6420
  - 1110 1100 1010 1000 0110 0100 0010 0000



## MIPS I-format Instructions

|        |        |        |                     |
|--------|--------|--------|---------------------|
| op     | rs     | rt     | constant or address |
| 6 bits | 5 bits | 5 bits | 16 bits             |

- Immediate arithmetic and load/store instructions
  - rt: destination or source register number
  - Constant:  $-2^{15}$  to  $+2^{15} - 1$
  - Address: offset added to base address in rs
- Design Principle 4:** Good design demands good compromises
  - Different formats complicate decoding, but allow 32-bit instructions uniformly
  - Keep formats as similar as possible



## MIPS I-format Example

|        |        |        |                     |
|--------|--------|--------|---------------------|
| op     | rs     | rt     | constant or address |
| 6 bits | 5 bits | 5 bits | 16 bits             |

lw \$t0, 32(\$s3) # Temporary reg \$t0 gets A[8]

|        |        |        |         |
|--------|--------|--------|---------|
| lw     | \$s3   | \$t0   | address |
| 6 bits | 5 bits | 5 bits | 16 bits |

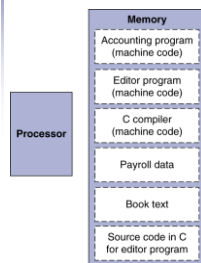
|        |        |        |         |
|--------|--------|--------|---------|
| 35     | 19     | 8      | 32      |
| 6 bits | 5 bits | 5 bits | 16 bits |

|        |        |        |                 |
|--------|--------|--------|-----------------|
| 100011 | 10011  | 01000  | 000000000100000 |
| 6 bits | 5 bits | 5 bits | 16 bits         |



## Stored Program Computers

### The BIG Picture



- Instructions represented in binary, just like data
- Instructions and data stored in memory
- Programs can operate on programs
  - e.g., compilers, linkers, ...
- Binary compatibility allows compiled programs to work on different computers
  - Standardized ISAs



## Logical Operations

- Instructions for bitwise manipulation

| Operation   | C  | Java | MIPS      |
|-------------|----|------|-----------|
| Shift left  | << | <<   | sll       |
| Shift right | >> | >>   | srl       |
| Bitwise AND | &  | &    | and, andi |
| Bitwise OR  |    |      | or, ori   |
| Bitwise NOT | ~  | ~    | nor       |

- Useful for extracting and inserting groups of bits in a word



## Shift Operations

|        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|
| op     | rs     | rt     | rd     | shamt  | funct  |
| 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |

- shamt: how many positions to shift
- Shift left logical
  - Shift left and fill with 0 bits
  - sll by  $i$  bits multiplies by  $2^i$
- Shift right logical
  - Shift right and fill with 0 bits
  - srl by  $i$  bits divides by  $2^i$  (unsigned only)



## AND Operations

- Useful to mask bits in a word
    - Select some bits, clear others to 0
- and \$t0, \$t1, \$t2

|      |   |
|------|---|
| \$t2 | 0000 0000 0000 0000 0000 1101 1100 0000 |
| \$t1 | 0000 0000 0000 0000 0011 1100 0000 0000 |
| \$t0 | 0000 0000 0000 0000 0000 1100 0000 0000 |



## OR Operations

- Useful to include bits in a word
    - Set some bits to 1, leave others unchanged
- or \$t0, \$t1, \$t2

|      |   |
|------|---|
| \$t2 | 0000 0000 0000 0000 0000 1101 1100 0000 |
| \$t1 | 0000 0000 0000 0000 0011 1100 0000 0000 |
| \$t0 | 0000 0000 0000 0000 0011 1101 1100 0000 |



## NOT Operations

- Useful to invert bits in a word
  - Change 0 to 1, and 1 to 0
- MIPS has NOR 3-operand instruction
  - $a \text{ NOR } b == \text{NOT} (a \text{ OR } b)$

`nor $t0, $t1, $zero` ← Register 0: always read as zero

\$t1: 0000 0000 0000 0000 0011 1100 0000 0000

\$t0: 1111 1111 1111 1111 1100 0011 1111 1111

## Conditional Operations

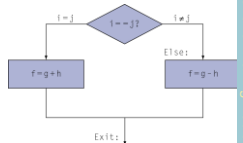
- Branch to a labeled instruction if a condition is true
  - Otherwise, continue sequentially
- `beq rs, rt, L1`
  - if  $(rs == rt)$  branch to instruction labeled L1;
- `bne rs, rt, L1`
  - if  $(rs != rt)$  branch to instruction labeled L1;
- `j L1`
  - unconditional jump to instruction labeled L1

## Compiling If Statements

- C code:
 

```
if (i==j) f = g+h;
else f = g-h;
```

  - f, g, h in \$s0, \$s1, \$s2



- Compiled MIPS code:
 

```
bne $s3, $s4, Else
add $s0, $s1, $s2
j Exit
Else: sub $s0, $s1, $s2
Exit: ...
```

Assembler calculates addresses

## Compiling Loop Statements

- C code:
 

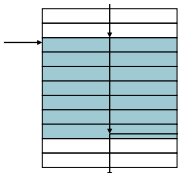
```
while (save[i] == k) i += 1;
```

  - i in \$s3, k in \$s5, address of save in \$s6
- Compiled MIPS code:

```
Loop: sll $t1, $s3, 2
add $t1, $t1, $s6
lw $t0, 0($t1)
bne $t0, $s5, Exit
addi $s3, $s3, 1
j Loop
Exit: ...
```

## Basic Blocks

- A basic block is a sequence of instructions with
  - No embedded branches (except at end)
  - No branch targets (except at beginning)



- A compiler identifies basic blocks for optimization
- An advanced processor can accelerate execution of basic blocks

## More Conditional Operations

- Set result to 1 if a condition is true
  - Otherwise, set to 0
- `slt rd, rs, rt`
  - if  $(rs < rt)$   $rd = 1$ ; else  $rd = 0$ ;
- `slti rt, rs, constant`
  - if  $(rs < \text{constant})$   $rt = 1$ ; else  $rt = 0$ ;
- Use in combination with `beq`, `bne`

```
slt $t0, $s1, $s2 # if ($s1 < $s2)
bne $t0, $zero, L # branch to L
```

## Branch Instruction Design

- Why not `b1t`, `bge`, etc?
- Hardware for `<`, `≥`, ... slower than `=`, `≠`
  - Combining with branch involves more work per instruction, requiring a slower clock
  - All instructions penalized!
- `beq` and `bne` are the common case
- This is a good design compromise



Chapter 2 — Instructions: Language of the Computer — 85

## Signed vs. Unsigned

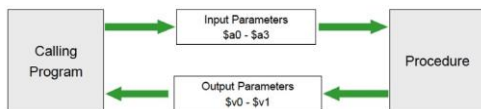
- Signed comparison: `slt`, `slti`
- Unsigned comparison: `sltu`, `sltui`
- Example
  - `$s0 = 1111 1111 1111 1111 1111 1111 1111 1111`
  - `$s1 = 0000 0000 0000 0000 0000 0000 0000 0001`
  - `slt $t0, $s0, $s1 # signed`
    - $-1 < +1 \Rightarrow \$t0 = 1$  (Set)
  - `sltu $t0, $s0, $s1 # unsigned`
    - $+4,294,967,295 < +1 \Rightarrow \$t0 = 0$  (Reset)



Chapter 2 — Instructions: Language of the Computer — 86

## Procedure Calling

- Procedure (function) performs a specific task and returns results to caller.



Chapter 2 — Instructions: Language of the Computer — 87

## Procedure Calling

- Calling program
  - Place parameters in registers `$a0 - $a3`
  - Transfer control to procedure
- Called procedure
  - Acquire storage for procedure, save values of required register(s) on stack `$sp`
  - Perform procedure's operations, restore the values of registers that it used
  - Place result in register for caller `$v0 - $v1`
  - Return to place of call by returning to instruction whose address is saved in `$ra`



Chapter 2 — Instructions: Language of the Computer — 88

## Register Usage

- `$a0 - $a3`: arguments (reg's 4 - 7)
- `$v0, $v1`: result values (reg's 2 and 3)
- `$t0 - $t9`: temporaries
  - Can be overwritten by callee
- `$s0 - $s7`: saved
  - Must be saved/restored by callee
- `$gp`: global pointer for static data (reg 28)
- `$sp`: stack pointer for dynamic data (reg 29)
- `$fp`: frame pointer (reg 30)
- `$ra`: return address (reg 31)



Chapter 2 — Instructions: Language of the Computer — 89

## Procedure Call Instructions

- Procedure call: jump and link
  - `jal ProcedureLabel`
    - Address of following instruction put in `$ra`
    - Jumps to target address
- Procedure return: jump register
  - `jr $ra`
    - Copies `$ra` to program counter
    - Can also be used for computed jumps
      - e.g., for case/switch statements



Chapter 2 — Instructions: Language of the Computer — 90

## Leaf Procedure Example

- C code:
 

```
int leaf_example (int g, h, i, j)
{ int f;
  f = (g + h) - (i + j);
  return f;
}
```

  - Arguments g, ..., j in \$a0, ..., \$a3
  - f in \$s0 (hence, need to save \$s0 on stack)
  - Result in \$v0



## Leaf Procedure Example (2)

- MIPS code:

| leaf_example:          |                    |
|------------------------|--------------------|
| addi \$sp, \$sp, -4    | Save \$s0 on stack |
| sw \$s0, 0(\$sp)       |                    |
| add \$t0, \$a0, \$a1   | Procedure body     |
| add \$t1, \$a2, \$a3   |                    |
| sub \$s0, \$t0, \$t1   |                    |
| add \$v0, \$s0, \$zero | Result             |
| lw \$s0, 0(\$sp)       | Restore \$s0       |
| addi \$sp, \$sp, 4     |                    |
| jr \$ra                | Return             |



## Leaf Procedure Example (3)

- MIPS code for calling function:

```
main:
...
jal leaf_example
...
```



## Non-Leaf Procedures

- Procedures that call other procedures
- For nested call, caller needs to save on the stack:
  - Its return address
  - Any arguments and temporaries needed after the call
- Restore from the stack after the call



## Non-Leaf Procedure Example

- C code:
 

```
int fact (int n)
{
  if (n < 1) return 1;
  else return n * fact(n - 1);
}
```

  - Argument n in \$a0
  - Result in \$v0



## Non-Leaf Procedure Example

- MIPS code:

| fact:                   |                            |
|-------------------------|----------------------------|
| addi \$sp, \$sp, -8     | # adjust stack for 2 items |
| sw \$ra, 4(\$sp)        | # save return address      |
| sw \$a0, 0(\$sp)        | # save argument            |
| slti \$t0, \$a0, 1      | # test for n < 1           |
| beq \$t0, \$zero, L1    |                            |
| addi \$v0, \$zero, 1    | # if so, result is 1       |
| addi \$sp, \$sp, 8      | # pop 2 items from stack   |
| jr \$ra                 | # and return               |
| L1: addi \$a0, \$a0, -1 | # else decrement n         |
| jal fact                | # recursive call           |
| lw \$a0, 0(\$sp)        | # restore original n       |
| lw \$ra, 4(\$sp)        | # and return address       |
| addi \$sp, \$sp, 8      | # pop 2 items from stack   |
| mul \$v0, \$a0, \$v0    | # multiply to get result   |
| jr \$ra                 | # and return               |





### Non-Leaf Procedure Example

**Main call (\$a0)=4, (\$ra)=return addr in main**

```
fact:
addi $sp, $sp, -8
sw $ra, 4($sp)
sw $a0, 0($sp)
slti $t0, $a0, 1
beq $t0, $zero, L1
addi $v0, $zero, 1
addi $sp, $sp, 8
jr $ra
L1: addi $a0, $a0, -1
jal fact
lw $a0, 0($sp)
lw $ra, 4($sp)
addi $sp, $sp, 8
mul $v0, $a0, $v0
jr $ra
```

**fact1 call (\$a0)<sub>1</sub>=3 (\$ra)<sub>1</sub>=return addr in fact1**

```
fact:
addi $sp, $sp, -8
sw $ra, 4($sp)
sw $a0, 0($sp)
slti $t0, $a0, 1
beq $t0, $zero, L1
addi $v0, $zero, 1
addi $sp, $sp, 8
jr $ra
L1: addi $a0, $a0, -1
jal fact
lw $a0, 0($sp)
lw $ra, 4($sp)
addi $sp, $sp, 8
mul $v0, $a0, $v0
jr $ra
```

Chapter 2 — Instructions: Language of the Computer — 97

### Non-Leaf Procedure Example

**fact2 call (\$a0)<sub>2</sub>=2 (\$ra)<sub>2</sub>=return addr in fact2**

```
fact:
addi $sp, $sp, -8
sw $ra, 4($sp)
sw $a0, 0($sp)
slti $t0, $a0, 1
beq $t0, $zero, L1
addi $v0, $zero, 1
addi $sp, $sp, 8
jr $ra
L1: addi $a0, $a0, -1
jal fact
lw $a0, 0($sp)
lw $ra, 4($sp)
addi $sp, $sp, 8
mul $v0, $a0, $v0
jr $ra
```

**fact3 call (\$a0)<sub>3</sub>=1 (\$ra)<sub>3</sub>=return addr in fact3**

```
fact:
addi $sp, $sp, -8
sw $ra, 4($sp)
sw $a0, 0($sp)
slti $t0, $a0, 1
beq $t0, $zero, L1
addi $v0, $zero, 1
addi $sp, $sp, 8
jr $ra
L1: addi $a0, $a0, -1
jal fact
lw $a0, 0($sp)
lw $ra, 4($sp)
addi $sp, $sp, 8
mul $v0, $a0, $v0
jr $ra
```

Chapter 2 — Instructions: Language of the Computer — 98

### Non-Leaf Procedure Example

**fact4 call (\$a0)<sub>4</sub>=0 (\$ra)<sub>4</sub>=return addr in fact4**

```
fact:
addi $sp, $sp, -8
sw $ra, 4($sp)
sw $a0, 0($sp)
slti $t0, $a0, 1
beq $t0, $zero, L1
addi $v0, $zero, 1
addi $sp, $sp, 8
jr $ra
L1: addi $a0, $a0, -1
jal fact
lw $a0, 0($sp)
lw $ra, 4($sp)
addi $sp, $sp, 8
mul $v0, $a0, $v0
jr $ra
```

**fact3 call (\$a0)<sub>3</sub>=1 (\$ra)<sub>3</sub>=return addr in fact3**

```
fact:
addi $sp, $sp, -8
sw $ra, 4($sp)
sw $a0, 0($sp)
slti $t0, $a0, 1
beq $t0, $zero, L1
addi $v0, $zero, 1
addi $sp, $sp, 8
jr $ra
L1: addi $a0, $a0, -1
jal fact
lw $a0, 0($sp)
lw $ra, 4($sp)
addi $sp, $sp, 8
mul $v0, $a0, $v0
jr $ra
```

Chapter 2 — Instructions: Language of the Computer — 99

### Non-Leaf Procedure Example

**fact2 call (\$a0)<sub>2</sub>=2 (\$ra)<sub>2</sub>=return addr in fact2**

```
fact:
addi $sp, $sp, -8
sw $ra, 4($sp)
sw $a0, 0($sp)
slti $t0, $a0, 1
beq $t0, $zero, L1
addi $v0, $zero, 1
addi $sp, $sp, 8
jr $ra
L1: addi $a0, $a0, -1
jal fact
lw $a0, 0($sp)
lw $ra, 4($sp)
addi $sp, $sp, 8
mul $v0, $a0, $v0
jr $ra
```

**fact1 call (\$a0)<sub>1</sub>=3 (\$ra)<sub>1</sub>=return addr in fact1**

```
fact:
addi $sp, $sp, -8
sw $ra, 4($sp)
sw $a0, 0($sp)
slti $t0, $a0, 1
beq $t0, $zero, L1
addi $v0, $zero, 1
addi $sp, $sp, 8
jr $ra
L1: addi $a0, $a0, -1
jal fact
lw $a0, 0($sp)
lw $ra, 4($sp)
addi $sp, $sp, 8
mul $v0, $a0, $v0
jr $ra
```

Chapter 2 — Instructions: Language of the Computer — 100

### Non-Leaf Procedure Example

**Main call (\$a0)=4, (\$ra)=return addr in main**

```
fact:
addi $sp, $sp, -8
sw $ra, 4($sp)
sw $a0, 0($sp)
slti $t0, $a0, 1
beq $t0, $zero, L1
addi $v0, $zero, 1
addi $sp, $sp, 8
jr $ra
L1: addi $a0, $a0, -1
jal fact
lw $a0, 0($sp)
lw $ra, 4($sp)
addi $sp, $sp, 8
mul $v0, $a0, $v0
jr $ra
```

**fact1 call (\$a0)<sub>1</sub>=3 (\$ra)<sub>1</sub>=return addr in fact1**

```
fact:
addi $sp, $sp, -8
sw $ra, 4($sp)
sw $a0, 0($sp)
slti $t0, $a0, 1
beq $t0, $zero, L1
addi $v0, $zero, 1
addi $sp, $sp, 8
jr $ra
L1: addi $a0, $a0, -1
jal fact
lw $a0, 0($sp)
lw $ra, 4($sp)
addi $sp, $sp, 8
mul $v0, $a0, $v0
jr $ra
```

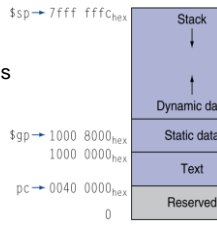
Chapter 2 — Instructions: Language of the Computer — 101

### Local Data on the Stack

Chapter 2 — Instructions: Language of the Computer — 102

## Memory Layout

- Text: program code
- Static data: global variables
  - e.g., static variables in C, constant arrays and strings
  - \$gp initialized to address allowing  $\pm$ offsets into this segment
- Dynamic data: heap
  - E.g., malloc in C, new in Java
- Stack: automatic storage



## Register Summary

- The following registers are preserved on call
  - \$s0 - \$s7, \$gp, \$sp, \$fp, and \$ra

| Register Number | Mnemonic Name | Conventional Use                                   | Register Number | Mnemonic Name | Conventional Use         |
|-----------------|---------------|--|-----------------|---------------|--------------------------|
| \$0             | zero          | Permanently 0                                      | \$24, \$25      | \$t8, \$t9    | Temporary                |
| \$1             | \$at          | Assembler Temporary (reserved)                     | \$26, \$27      | \$t0, \$t1    | Kernel (reserved for OS) |
| \$2, \$3        | \$v0, \$v1    | Value returned by a subroutine                     | \$28            | \$gp          | Global Pointer           |
| \$4-\$7         | \$a0-\$a3     | Arguments to a subroutine                          | \$29            | \$sp          | Stack Pointer            |
| \$8-\$15        | \$t0-\$t7     | Temporary (not preserved across a function call)   | \$30            | \$fp          | Frame Pointer            |
| \$16-\$23       | \$s0-\$s7     | Saved registers (preserved across a function call) | \$31            | \$ra          | Return Address           |

## Character Data

- Byte-encoded character sets
  - ASCII: (7-bit) 128 characters
    - 95 graphic, 33 control
  - Latin-1: (8-bit) 256 characters
    - ASCII, +96 more graphic characters
- Unicode: 32-bit character set
  - Used in Java, C++ wide characters, ...
  - Most of the world's alphabets, plus symbols
  - UTF-8, UTF-16: variable-length encodings

## ASCII Representation of Characters

| Dec | Hex | Name                  | Char | Ctrl-char | Dec | Hex | Char  | Dec | Hex | Char | Dec | Hex | Char |
|-----|-----|-----------------------|------|-----------|-----|-----|-------|-----|-----|------|-----|-----|------|
| 0   | 0   | Null                  | NUL  | CTRL-@    | 32  | 20  | Space | 64  | 40  | @    | 96  | 60  |      |
| 1   | 1   | Start of heading      | SOH  | CTRL-A    | 33  | 21  | !     | 65  | 41  | A    | 97  | 61  | a    |
| 2   | 2   | Start of text         | STX  | CTRL-B    | 34  | 22  | "     | 66  | 42  | B    | 98  | 62  | b    |
| 3   | 3   | End of text           | ETX  | CTRL-C    | 35  | 23  | #     | 67  | 43  | C    | 99  | 63  | c    |
| 4   | 4   | End of transmit       | EOF  | CTRL-D    | 36  | 24  | \$    | 68  | 44  | D    | 100 | 64  | d    |
| 5   | 5   | Enquiry               | ENQ  | CTRL-E    | 37  | 25  | %     | 69  | 45  | E    | 101 | 65  | e    |
| 6   | 6   | Acknowledge           | ACK  | CTRL-F    | 38  | 26  | &     | 70  | 46  | F    | 102 | 66  | f    |
| 7   | 7   | Bell                  | BEL  | CTRL-G    | 39  | 27  | '     | 71  | 47  | G    | 103 | 67  | g    |
| 8   | 8   | Backspace             | BS   | CTRL-H    | 40  | 28  | (     | 72  | 48  | H    | 104 | 68  | h    |
| 9   | 9   | Horizontal tab        | HT   | CTRL-I    | 41  | 29  | )     | 73  | 49  | I    | 105 | 69  | i    |
| 10  | 0A  | Line feed             | LF   | CTRL-J    | 42  | 2A  | *     | 74  | 4A  | J    | 106 | 6A  | j    |
| 11  | 0B  | Vertical tab          | VT   | CTRL-K    | 43  | 2B  | +     | 75  | 4B  | K    | 107 | 6B  | k    |
| 12  | 0C  | Form feed             | FF   | CTRL-L    | 44  | 2C  | ,     | 76  | 4C  | L    | 108 | 6C  | l    |
| 13  | 0D  | Carriage feed         | CR   | CTRL-M    | 45  | 2D  | -     | 77  | 4D  | M    | 109 | 6D  | m    |
| 14  | 0E  | Shift out             | SO   | CTRL-N    | 46  | 2E  | .     | 78  | 4E  | N    | 110 | 6E  | n    |
| 15  | 0F  | Shift in              | SI   | CTRL-O    | 47  | 2F  | /     | 79  | 4F  | O    | 111 | 6F  | o    |
| 16  | 10  | Data line escape      | DLE  | CTRL-P    | 48  | 30  | 0     | 80  | 50  | P    | 112 | 70  | p    |
| 17  | 11  | Device control 1      | DC1  | CTRL-Q    | 49  | 31  | 1     | 81  | 51  | Q    | 113 | 71  | q    |
| 18  | 12  | Device control 2      | DC2  | CTRL-R    | 50  | 32  | 2     | 82  | 52  | R    | 114 | 72  | r    |
| 19  | 13  | Device control 3      | DC3  | CTRL-S    | 51  | 33  | 3     | 83  | 53  | S    | 115 | 73  | s    |
| 20  | 14  | Device control 4      | DC4  | CTRL-T    | 52  | 34  | 4     | 84  | 54  | T    | 116 | 74  | t    |
| 21  | 15  | Neg acknowledge       | NAK  | CTRL-U    | 53  | 35  | 5     | 85  | 55  | U    | 117 | 75  | u    |
| 22  | 16  | Synchronous idle      | SYN  | CTRL-V    | 54  | 36  | 6     | 86  | 56  | V    | 118 | 76  | v    |
| 23  | 17  | End of transmit block | ETB  | CTRL-W    | 55  | 37  | 7     | 87  | 57  | W    | 119 | 77  | w    |
| 24  | 18  | Cancel                | CAN  | CTRL-X    | 56  | 38  | 8     | 88  | 58  | X    | 120 | 78  | x    |
| 25  | 19  | End of medium         | EM   | CTRL-Y    | 57  | 39  | 9     | 89  | 59  | Y    | 121 | 79  | y    |
| 26  | 1A  | Substitute            | SUB  | CTRL-Z    | 58  | 3A  | :     | 90  | 5A  | Z    | 122 | 7A  | z    |
| 27  | 1B  | Escape                | ESC  | CTRL-[    | 59  | 3B  | ;     | 91  | 5B  | [    | 123 | 7B  | {    |
| 28  | 1C  | File separator        | FS   | CTRL-\    | 60  | 3C  | <     | 92  | 5C  | \    | 124 | 7C  |      |
| 29  | 1D  | Group separator       | GS   | CTRL-]    | 61  | 3D  | =     | 93  | 5D  | ]    | 125 | 7D  | }    |
| 30  | 1E  | Record separator      | RS   | CTRL-^    | 62  | 3E  | >     | 94  | 5E  | ^    | 126 | 7E  | ~    |
| 31  | 1F  | Unit separator        | US   | CTRL-`    | 63  | 3F  | ?     | 95  | 5F  | `    | 127 | 7F  | DEL  |

## ASCII Characters

- American Standard Code for Information Interchange (ASCII).
- Most computers use 8-bit to represent each character. (Java uses Unicode, which is 16-bit).
- Signs are combination of characters.
- How to load a byte?
  - lb, lbu, sb for byte (ASCII)
  - lh, lhu, sh for half-word instruction (Unicode)

## Byte/Halfword Operations

- Could use bitwise operations
- MIPS byte/halfword load/store
  - String processing is a common case
- lb rt, offset(rs)      lh rt, offset(rs)
- sb rt, offset(rs)      sh rt, offset(rs)
- lhbu rt, offset(rs)    lhu rt, offset(rs)
- sbu rt, offset(rs)    shu rt, offset(rs)
- Zero extend to 32 bits in rt
- Store just rightmost byte/halfword

## String Copy Example

- C code:
  - Null-terminated string

```
void strcpy (char x[], char y[])
{ int i;
  i = 0;
  while ((x[i]=y[i])!='\0')
    i += 1;
}
```

  - Addresses of x, y in \$a0, \$a1
  - i in \$s0



## String Copy Example

- MIPS code:

```
strcpy:
  addi $sp, $sp, -4 # adjust stack for 1 item
  sw $s0, 0($sp) # save $s0
  add $s0, $zero, $zero # i = 0
L1: add $t1, $s0, $a1 # addr of y[i] in $t1
    lbu $t2, 0($t1) # $t2 = y[i]
    add $t3, $s0, $a0 # addr of x[i] in $t3
    sb $t2, 0($t3) # x[i] = y[i]
    beq $t2, $zero, L2 # exit loop if y[i] == 0
    addi $s0, $s0, 1 # i = i + 1
    j L1 # next iteration of loop
L2: lw $s0, 0($sp) # restore saved $s0
    addi $sp, $sp, 4 # pop 1 item from stack
    jr $ra # and return
```



## 32-bit Constants

- Most constants are small
  - 16-bit immediate is sufficient
- For the occasional 32-bit constant
 

```
lui rt, constant
```

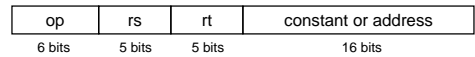
  - Copies 16-bit constant to left 16 bits of rt
  - Clears right 16 bits of rt to 0

```
lui $s0, 61      0000 0000 0011 1101 0000 0000 0000 0000
ori $s0, $s0, 2304 0000 0000 0011 1101 0000 1001 0000 0000
```



## Branch Addressing

- Branch instructions specify
  - Opcode, two registers, target address
- Most branch targets are near branch
  - Forward or backward

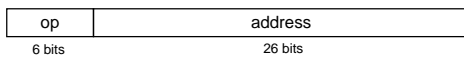


- PC-relative addressing
  - Target address = PC + offset × 4
  - PC already incremented by 4 by this time



## Jump Addressing

- Jump (j and jal) targets could be anywhere in text segment
  - Encode full address in instruction



- PseudoDirect jump addressing
  - Target address =  $PC_{31..28} : (\text{address} \times 4)$



## Target Addressing Example

- Loop code from earlier example
  - Assume Loop at location 80000

|                         |       |    |    |    |   |   |       |
|-------------------------|-------|----|----|----|---|---|-------|
| Loop: sll \$t1, \$s3, 2 | 80000 | 0  | 0  | 19 | 9 | 2 | 0     |
| add \$t1, \$t1, \$s6    | 80004 | 0  | 9  | 22 | 9 | 0 | 32    |
| lw \$t0, 0(\$t1)        | 80008 | 35 | 9  | 8  |   |   | 0     |
| bne \$t0, \$s5, Exit    | 80012 | 5  | 8  | 21 |   |   | 2     |
| addi \$s3, \$s3, 1      | 80016 | 8  | 19 | 19 |   |   | 1     |
| j Loop                  | 80020 | 2  |    |    |   |   | 20000 |
| Exit: ...               | 80024 |    |    |    |   |   |       |



## Branching Far Away

- If branch target is too far to encode with 16-bit offset, assembler rewrites the code

- Example

```
beq $s0,$s1, L1
```

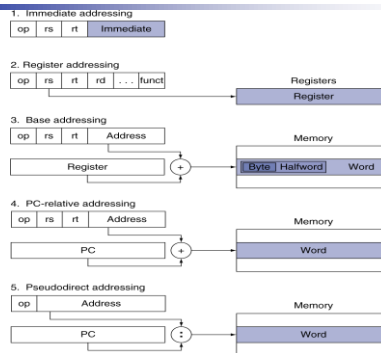
written as

```
bne $s0,$s1, L2
j L1
```

```
L2: ...
```



## Addressing Mode Summary



## Synchronization (Parallelism)

- Two processors sharing an area of memory
  - P1 writes, then P2 reads
  - Data race if P1 and P2 don't synchronize
    - Result depends on order of accesses
- Hardware support required
  - Atomic read/write memory operation
  - No other access to the location allowed between the read and write
- Could be a single instruction
  - E.g., atomic swap of register ↔ memory
  - Or an atomic pair of instructions

## Synchronization in MIPS

- Load linked: `ll rt, offset(rs)`
- Store conditional: `sc rt, offset(rs)`
  - Succeeds if location not changed since the `ll`
    - Returns 1 in `rt`
  - Fails if location is changed
    - Returns 0 in `rt`
- Example: atomic swap (to test/set lock variable)
 

```
try: add $t0,$zero,$s4 ;copy exchange value
ll $t1,0($s1) ;load linked
sc $t0,0($s1) ;store conditional
beq $t0,$zero,try ;branch store fails
add $s4,$zero,$t1 ;put load value in $s4
```

## C Sort Example

- Illustrates use of assembly instructions for a C bubble sort function
- Swap procedure (leaf)
 

```
void swap(int v[], int k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

  - `v` in `$a0`, `k` in `$a1`, `temp` in `$t0`

## The Procedure Swap

```
swap: sll $t1, $a1, 2 # $t1 = k * 4
add $t1, $a0, $t1 # $t1 = v+(k*4)
# (address of v[k])
lw $t0, 0($t1) # $t0 (temp) = v[k]
lw $t2, 4($t1) # $t2 = v[k+1]
sw $t2, 0($t1) # v[k] = $t2 (v[k+1])
sw $t0, 4($t1) # v[k+1] = $t0 (temp)
jr $ra # return to calling routine
```

## Example

```
.data
STR: .asciiz "a1b2c3d4e5f6g7h8i9" # STR[0,1,...,17] = a,1,b,...,0 (8 bits)
MAX: .word 0x44556677;           # MAX = 0x44556677 (32 bits)
SIZE: .byte 33,22,11;           # SIZE[0,1,2] = 33,22,11 (8 bits)
count: .word 0,1,2;             # count[0,1,2] = 0,1,2 (32 bits)
#-----
.text
main:
la    $t0, STR                # $t0 = address of STR[0]
lb    $t1, 0($t0)              # $t1 = 97 (ascii code for 'a' in decimal)
addi  $t2, $t1, -4            # $t2 = 93
lb    $t3, 3($t0)              # $t3 = 50 (ascii code for '2' in decimal)
lb    $t4, 23($t0)             # $t4 = 68 = 44 hex (word alignment)
lb    $t5, 24($t0)             # $t5 = 33
lb    $t6, 32($t0)             # $t6 = 1 (word=0001)
lb    $t7, 33($t0)             # $t7 = 0 (word=0001)
lh    $t8, 26($t0)             # $t8 = 11 = b hex
lw    $t9, 36($t0)             # $t9 = 2
#-----
```

jr

\$ra

# return

Chapter 2 — Instructions: Language of the Computer — 121

## Concluding Remarks

- Design principles
  1. Simplicity favors regularity
  2. Smaller is faster
  3. Make the common case fast
  4. Good design demands good compromises
- Layers of software/hardware
  - Compiler, assembler, hardware
- MIPS: typical of RISC ISAs
  - c.f. x86

Chapter 2 — Instructions: Language of the Computer — 122

## Acknowledgement

The slides are adopted from Computer Organization and Design, 5th Edition by David A. Patterson and John L. Hennessy 2014, published by MK (Elsevier)



Chapter 2 — Instructions: Language of the Computer — 123