Chapter 2

Instructions: Language of the Computer

Instruction Set

- The complete set 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

The MIPS Instruction Set

- Used as the example throughout the book
- Stanford MIPS commercialized by MIPS Technologies (www.mips.com)
  - Founded by John Hennessy
- Large share of embedded core market
  - Applications in consumer electronics, network/storage equipment, cameras, printers, …

Arithmetic Operations

- Add and subtract, three operands
  - Two sources and one destination

```
add a, b, c  # a gets b + c
```

- All arithmetic operations have the same form --- ***Regularity!***

Design Principle

- *Design Principle 1:* Simplicity favors regularity
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost

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
```

Operand

Types of Operands

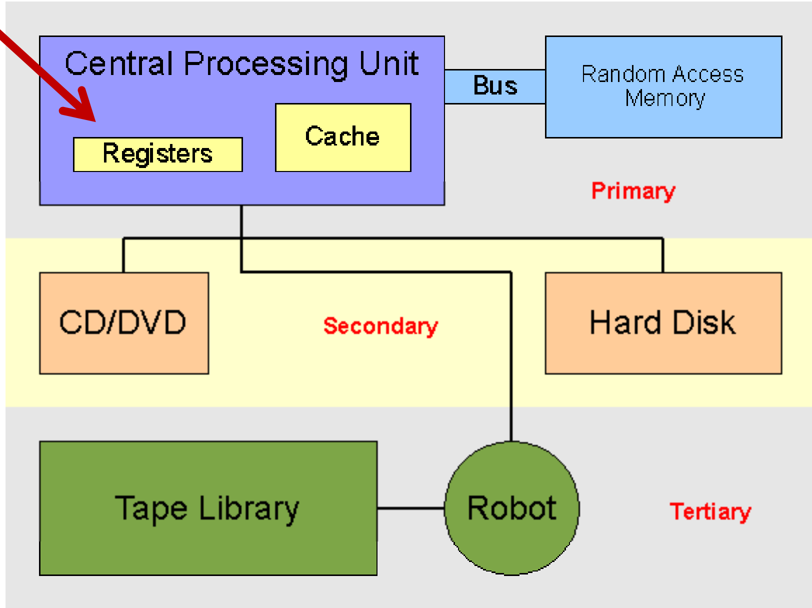
- Register operand
- Memory operand
- Immediate operand

Register vs. Memory ?

Register

- Register: a small amount of storage available on the CPU
  - can be accessed more quickly than storage available elsewhere.
  - Not considered as part of the normal memory range for the machine.

Data is loaded from cache or RAM into registers, manipulated in some way, and then stored back into memory.



Register Operands

- Arithmetic instructions use register operands
- MIPS has 32 × 32-bit registers
  - Use for frequently accessed data
  - Numbered 0 to 31
  - 32-bit data called a "word"
- Naming
  - $t0, $t1, … for temporary registers needed
  - $s0, $s1, …for registers that correspond to variables in C or Java program

Design Principle

- *Design Principle 2:* Smaller is faster

Architecture	Integer registers	Double FP registers
x86	8	8
x86-64	16	16
IBM/360	16	4
Z/Architecture	16	16
Itanium	128	128
UltraSPARC	32	32
POWER	32	32
Alpha	32	32
6502	3	0
PIC microcontroller	1	0
AVR microcontroller	32	0
ARM	16	16

More registers may increase the clock cycle time.

Register Operand Example

- C code:

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

$s0 $s1 $s2 $s3 $s4

- Compiled MIPS code:

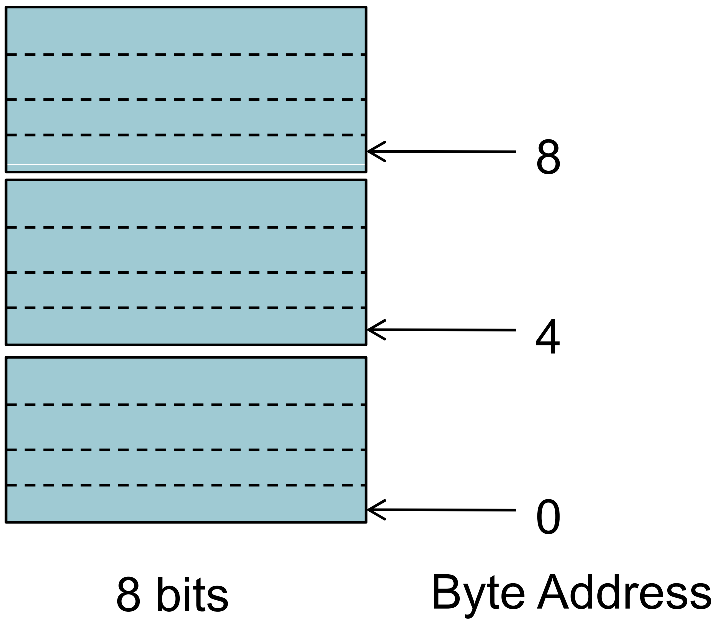
```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```

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
  - Word address must be a multiple of 4

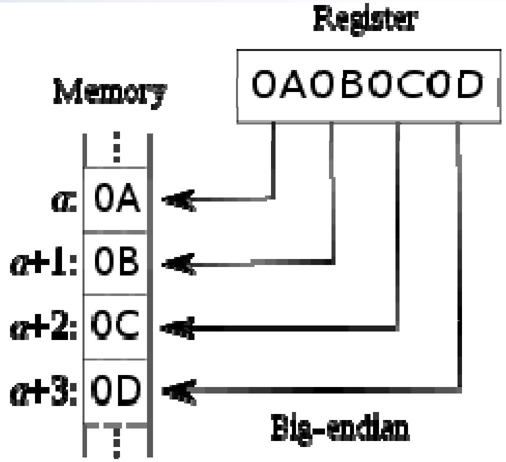


Byte vs. Word ?

Big Endian vs. Little Endian

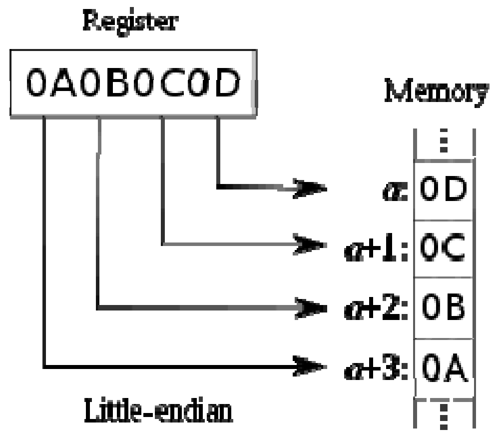
- Big Endian: use the address of the leftmost byte as the word address
- *Increasing addresses* →

...$0A_h 0B_h 0C_h 0D_h$...



- Little Endian: use the address of the rightmost byte as the word address
- *Increasing addresses* →

...$0D_h 0C_h 0B_h 0A_h$...



- MIPS is Big-endian

Memory Operand Example 1

- C code: $s2

```
g = h + A[8];
```

$s1

  - 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 2

- 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 from A[8]
add $t0, $s2, $t0
sw  $t0, 48($s3)     # store word in A[12]
```

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

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

- *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
```

Signed/unsigned Numbers

Unsigned Binary Integers

- Given an n-bit number

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- Range: 0 to $+2^n - 1$
- Example
  - $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$
    $= 0 + \ldots + 1\times 2^3 + 0\times 2^2 + 1\times 2^1 + 1\times 2^0$
    $= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$
- Using 32 bits
  - 0 to +4,294,967,295

How to represent signed numbers?

2s-Complement Signed Integers

- Given an n-bit number

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- Range: -2^{n-1} to $+2^{n-1} - 1$
- Example
  - $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
    $= -1 \times 2^{31} + 1 \times 2^{30} + \ldots + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$
    $= -2{,}147{,}483{,}648 + 2{,}147{,}483{,}644 = -4_{10}$
- Using 32 bits
  - –2,147,483,648 to +2,147,483,647

2s-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 2s-complement representation
- Some specific numbers
  - 0: 0000 0000 … 0000
  - –1: 1111 1111 … 1111
  - Most-negative: 1000 0000 … 0000
  - Most-positive: 0111 1111 … 1111

???

Negation Shortcut

- Complement and add 1
  - Complement means 1 → 0, 0 → 1

$$x + \overline{x} = 1111...111_2 = -1$$
$$-x = \overline{x} + 1$$

- Example: negate +2
  - +2 = 0000 0000 … 0010_2
  - –2 = 1111 1111 … 1101_2 + 1
    = 1111 1111 … 1110_2

How to calculate -2?

Sign Extension Shortcut

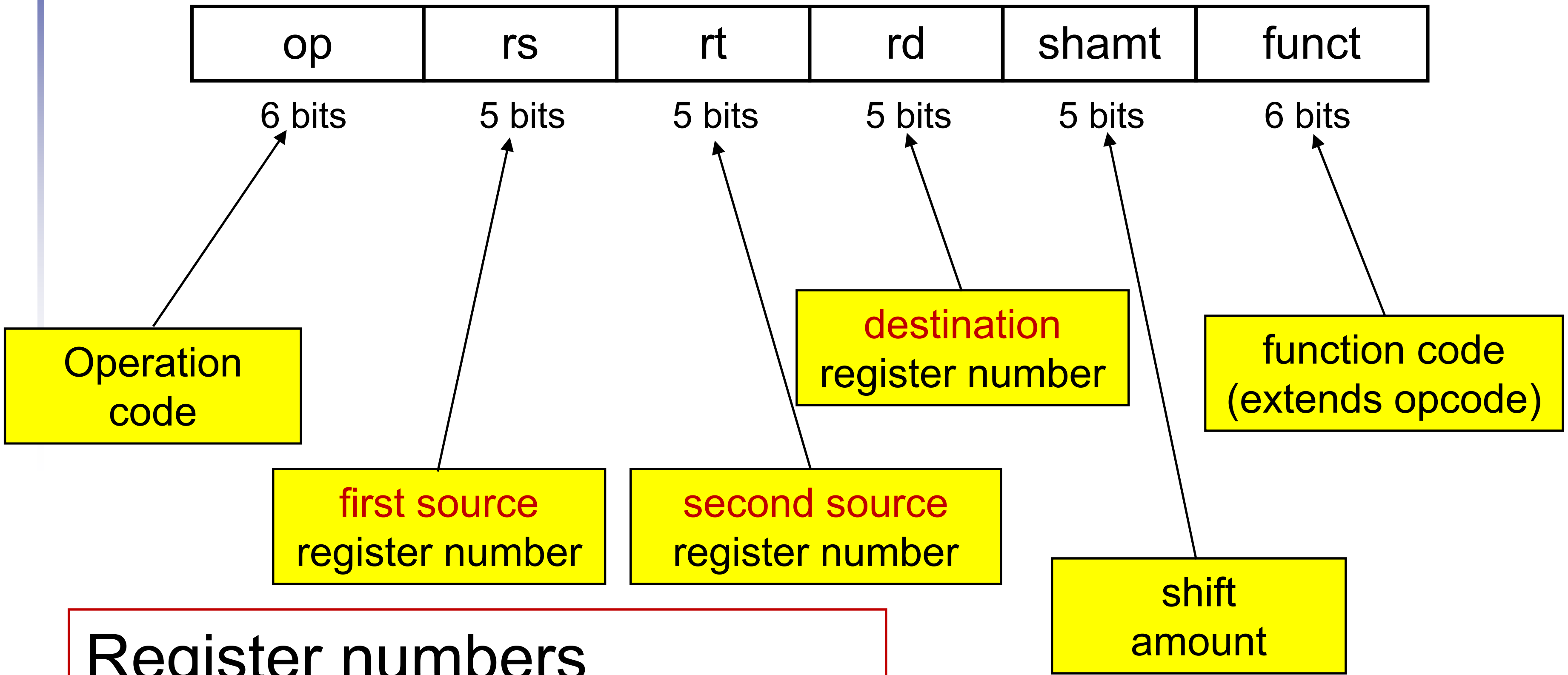
- Representing a number using more bits
  - Preserve the numeric value
- Replicate the sign bit to the left
  - c.f. unsigned values: extend with 0s
- Examples: 8-bit to 16-bit
  - +2: 0000 0010 => 0000 0000 0000 0010
  - –2: 1111 1110 => 1111 1111 1111 1110

Representing Instructions

Representing Instructions

- Instructions are encoded in binary
  - Called machine code
- MIPS instructions
  - Encoded as 32-bit instruction words
  - Small number of formats
    - R-format : for registers
    - I-format: for immediate
- Regularity!

MIPS R-format Instructions



Register numbers

- $t0 – $t7 are reg's 8 – 15
- $t8 – $t9 are reg's 24 – 25
- $s0 – $s7 are reg's 16 – 23

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
```

0	\$s1	\$s2	\$t0	0	add
0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

$00000010001100100100000000100000_2 = 02324020_{16}$

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

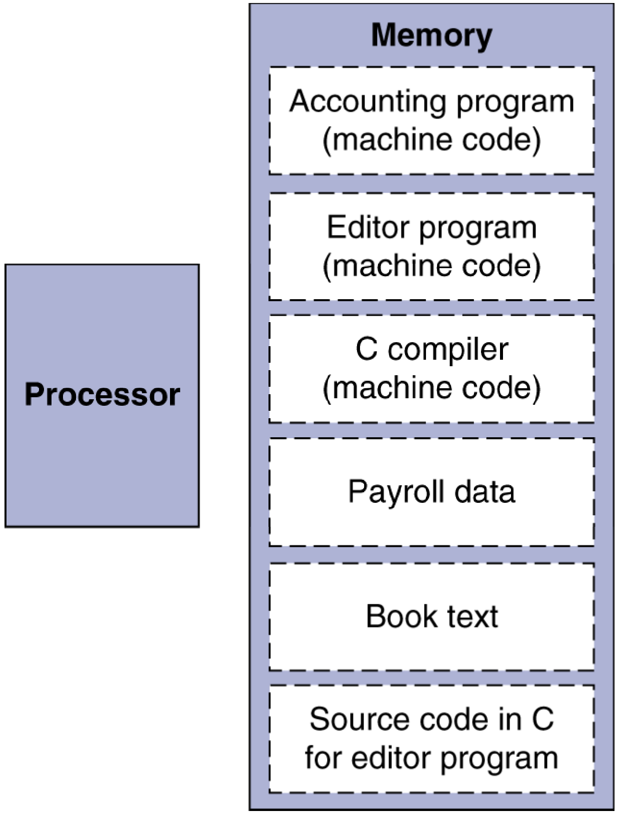
- Immediate arithmetic and load/store instructions
  - rt: destination or source register number
  - Constant: -2^{15} to $+2^{15}-1$
  - Address: offset added to base address in rs

Design Principle

- *Design Principle 4:* Good design demands good compromises
  - Different formats complicate decoding, but allow 32-bit instructions uniformly
    - R-format vs. I-format
  - Keep formats as similar as possible

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