

# Lecture 3

CMPEN 331

A decorative blue graphic element consisting of a thin curved line starting from the left edge and a larger, solid blue shape that tapers from the curve towards the bottom right corner.

# Instructions: Language of the Computer

## **CHAPTER 2**

# Instruction Set

- The repertoire 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

# Introduction

- A *bit* is the most basic unit of information in a computer.
  - It is a state of “on” or “off” in a digital circuit.
  - Sometimes these states are “high” or “low” voltage instead of “on” or “off..”
- A *byte* is a group of eight bits.
  - A byte is the smallest possible *addressable* unit of computer storage.
  - The term, “addressable,” means that a particular byte can be retrieved according to its location in memory.

# The MIPS Instruction Set

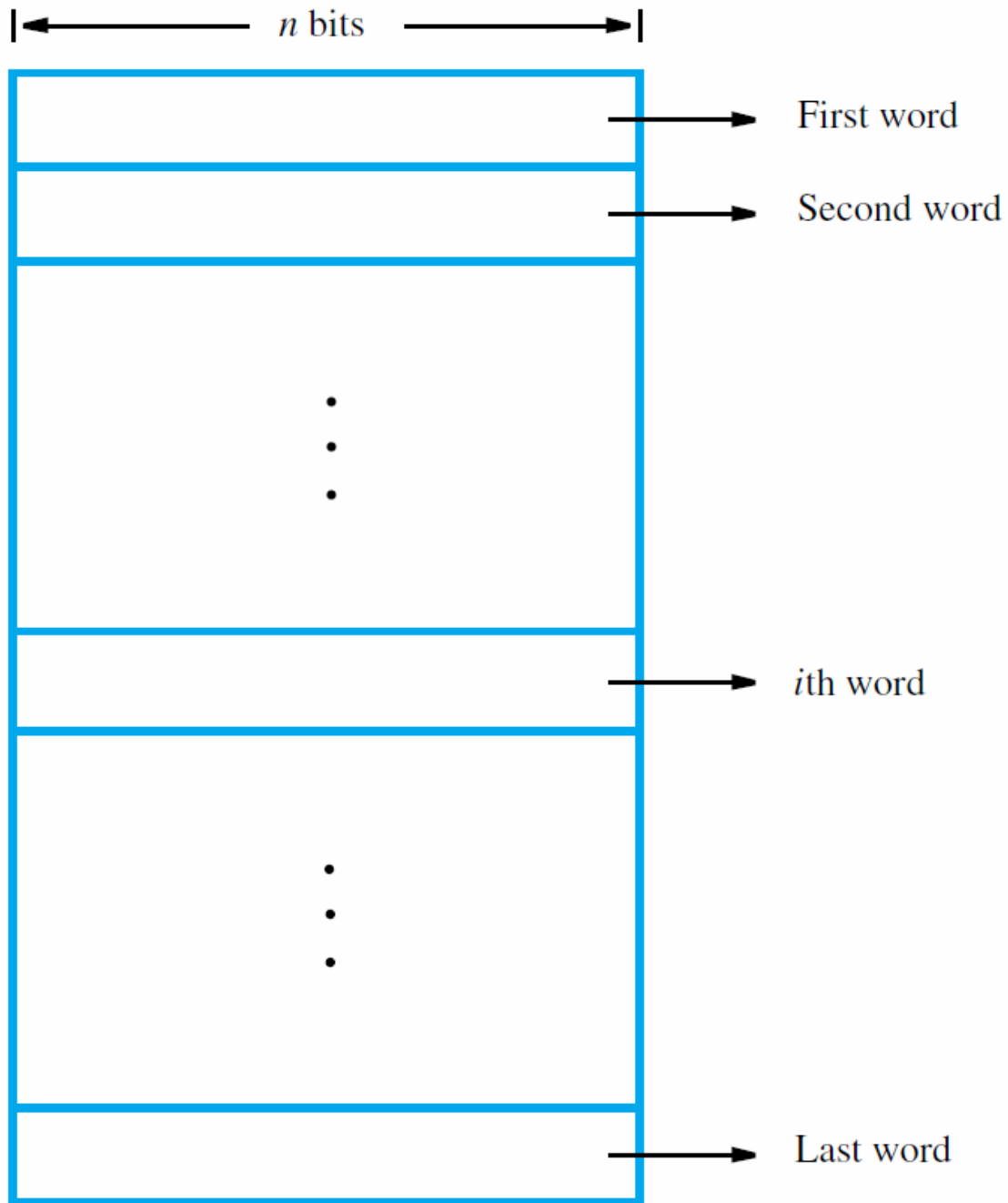
- Used as the example throughout the book
- Stanford MIPS commercialized by MIPS Technologies ([www.mips.com](http://www.mips.com))
- Large share of embedded core market
  - Applications in consumer electronics, network/storage equipment, cameras, printers, ...
- Typical of many modern ISAs

# Reminders

- Install *MARS* on your laptop
- Keep track of course updates on CANVAS
- Make sure your CSE account is operational
- Question/comments about the lab account/hardware/system, contact [helpdesk@cse.psu.edu](mailto:helpdesk@cse.psu.edu)
- For questions about the programming assignments, reach out to the Tas of the course.

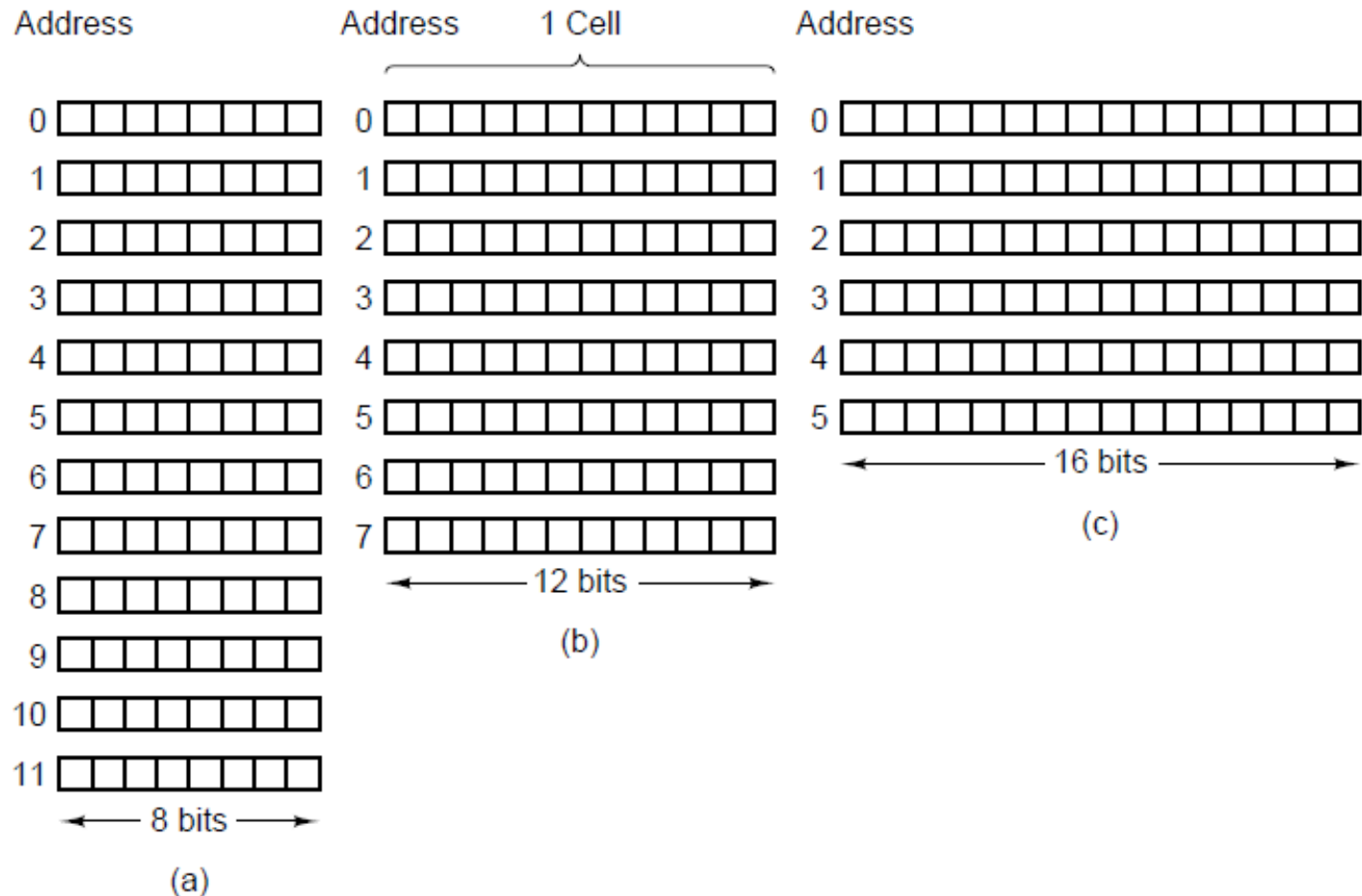
# Memory Organization

- Memory consists of many millions of **cells**
- Each cell holds a bit of information, 0 or 1
- Information is usually handled in larger units
- A **word** is a group of  $n$  bits
- **Word length** can be 16 to 64 bits but we will stick with 32 bits as a one word through out the semester
- **Memory** is a collection of consecutive words of the size specified by the word length





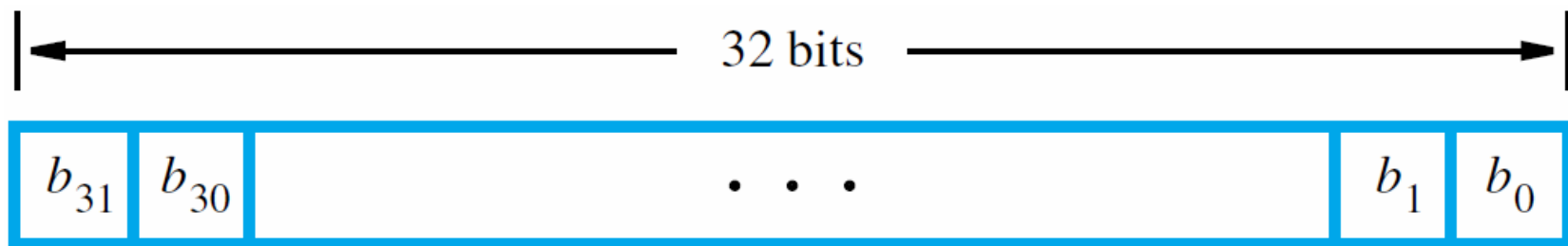
# Primary Memory (2)



Three ways of organizing a 96-bit memory

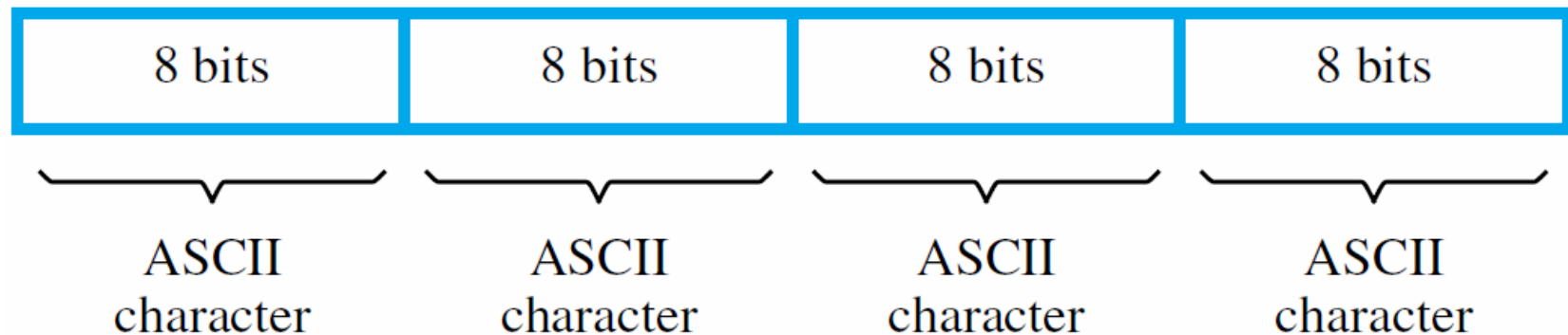
# Word and Byte Encoding

- A common word length is 32 bits
- Such a word can store a 32-bit signed integer or four 8-bit bytes (e.g., ASCII characters)
- For 32-bit integer encoding, bit  $b_{31}$  is sign bit
- Words in memory may store data or machine instructions for a program
- Each machine instruction may require one or more consecutive words for encoding



↑ Sign bit:  $b_{31} = 0$  for positive numbers  
 $b_{31} = 1$  for negative numbers

(a) A signed integer



(b) Four characters

# Addresses for Memory Locations

- To store or retrieve items of information, each memory location has a distinct **address**
- Numbers 0 to  $2^k - 1$  are used as addresses for successive locations in the memory
- The  $2^k$  locations constitute the **address space**
- Memory size set by  $k$  (number of address bits)
- Examples:  $k = 20 \rightarrow 2^{20}$  or 1M locations,  
 $k = 32 \rightarrow 2^{32}$  or 4G locations

# Byte Addressability

- Byte size is always 8 bits
- Impractical to assign an address to each bit
- Instead, provide a **byte-addressable** memory that assigns an address to each byte
- Byte locations have addresses 0, 1, 2, ...
- Assuming that the word length is **32** bits, word locations have addresses 0, 4, 8, ...

# Big- and Little-Endian Addressing

- Byte ordering, or *endianness*, is another major architectural consideration.
- Two ways to assign byte address across words
- **Big-endian** addressing assigns lower addresses to more significant (leftmost) bytes of word
- **Little-endian** addressing uses opposite order
- Addresses for 32-bit words are still 0, 4, 8, ...
- Bits in each byte labeled  $b_7 \dots b_0$ , left to right

# Big- and Little-Endian Addressing

- If we have a two-byte integer, the integer may be stored so that the least significant byte is followed by the most significant byte or vice versa.
  - In *little endian* machines, the least significant byte is followed by the most significant byte.
  - *Big endian* machines store the most significant byte first (at the lower address).

Word  
address

Byte address

0	0	1	2	3
4	4	5	6	7
$2^k - 4$	$2^k - 4$	$2^k - 3$	$2^k - 2$	$2^k - 1$

(a) Big-endian assignment

Byte address

0	3	2	1	0
4	7	6	5	4
$2^k - 4$	$2^k - 1$	$2^k - 2$	$2^k - 3$	$2^k - 4$

(b) Little-endian assignment



# Big- and Little-Endian Addressing

- As an example, suppose we have the hexadecimal number 0x12345678.
- The big endian and small endian arrangements of the bytes are shown below.

Address →	00	01	10	11
Big Endian	12	34	56	78
Little Endian	78	56	34	12

# Little Endian Order

- All data types larger than a byte store their individual bytes in reverse order. The least significant byte occurs at the first (lowest) memory address.

- Example:

**12345678h**

0000:	78
0001:	56
0002:	34
0003:	12

# Word Alignment

- # of bytes per word is normally a power of 2
- Word locations have **aligned** addresses if they begin at byte addresses that are multiples of the number of bytes in a word
- Examples of aligned addresses:
  - 2 bytes per word  $\rightarrow 0, 2, 4, \dots$
  - 8 bytes per word  $\rightarrow 0, 8, 16, \dots$

# Memory Models

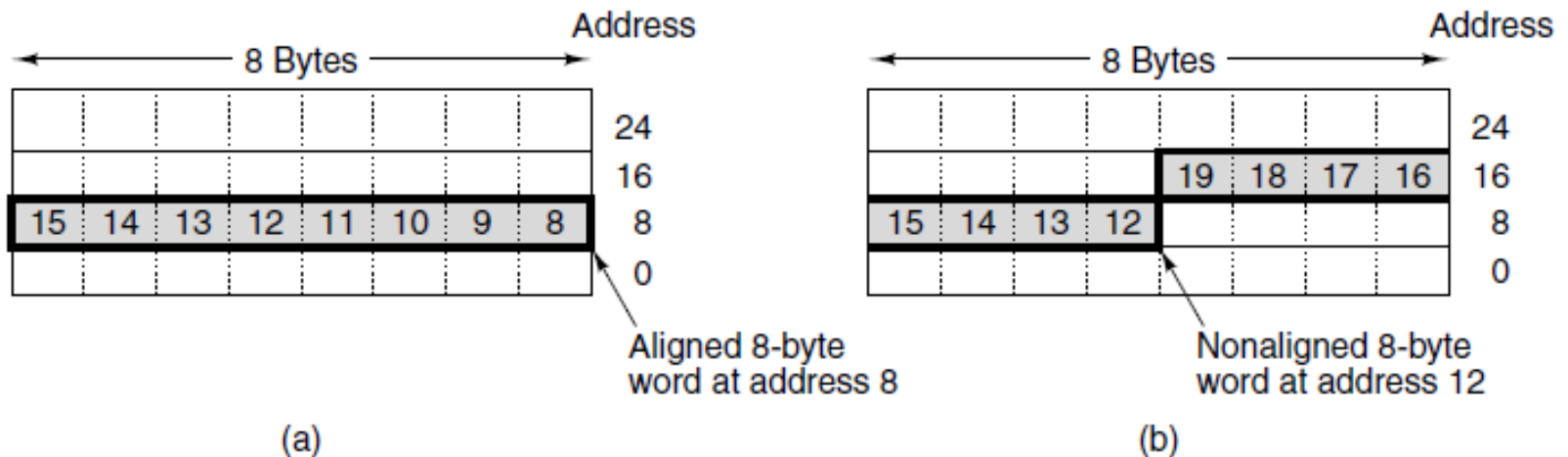


Figure 5-2. An 8-byte word in a memory.  
(a) Aligned. (b) Not aligned. Some machines require that words in memory be aligned.

# Memory Operations

- Memory contains data & program instructions
- Control circuits initiate transfer of data and instructions between memory and processor
- **Read operation:** memory retrieves contents at address location given by processor
- **Write operation:** memory overwrites contents at given location with given data

# Instructions and Sequencing

- Instructions for a computer must support:
  - data transfers to and from the memory
  - arithmetic and logic operations on data
  - program sequencing and control
  - input/output transfers

# Register Transfer Notation

- Register transfer notation is used to describe hardware-level data transfers and operations
- Arbitrary names for locations in memory
- Use [...] to denote contents of a location
- Use  $\leftarrow$  to denote transfer to a destination
- Example:  $R2 \leftarrow [LOC]$   
(transfer from LOC in memory to register R2)

# Register Transfer Notation

- RTN can be extended to also show arithmetic operations involving locations
- Example:  $R4 \leftarrow [R2] + [R3]$   
(add the contents of registers R2 and R3,  
place the sum in register R4)
- Right-hand expression always denotes a value, left-hand side always names a location



# Assembly-Language Notation

- RTN shows data transfers and arithmetic
- Another notation is needed to represent machine instructions & programs using them
- **Assembly language** is used for this purpose
- For the two preceding examples using RTN, the assembly-language instructions are:

lw            R2, LOC

add          R4, R2, R3

# RISC and CISC Instruction Sets

- Nature of instructions distinguishes computer
- Two fundamentally different approaches
- **Reduced Instruction Set Computers (RISC)** have one-word instructions and require arithmetic operands to be in registers
- **Complex Instruction Set Computers (CISC)** have multi-word instructions and allow operands directly from memory

# RISC Instruction Sets

- RISC is simpler
- RISC instructions each occupy a single word
- A **load/store architecture** is used, meaning:
  - only Load and Store instructions are used to access memory operands
  - operands for arithmetic/logic instructions must be in registers, or one of them may be given explicitly in instruction word

# Real World Architectures

- The MIPS family of CPUs has been one of the most successful in its class.
- In 1986 the first MIPS CPU was announced.
- It had a 32-bit word size and could address 4GB of memory.
- Over the years, MIPS processors have been used in general purpose computers as well as in games.
- The MIPS architecture now offers 32- and 64-bit versions.

# Real World Architectures

- MIPS was one of the first RISC microprocessors.
- MIPS was designed with performance in mind: It is a *load/store* architecture, meaning that only the **load and store** instructions can access memory.
- The registers in the MIPS architecture keeps bus traffic to a minimum.

# 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 favours 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
```

# Register Operands

- Arithmetic instructions use register operands
- MIPS has a  $32 \times 32$ -bit register file
  - Use 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*
  - 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 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

# Lecture 4

CMPEN 331

A decorative blue graphic consisting of a curved line that starts near the top left and sweeps down towards the bottom right, forming a large, stylized shape that resembles a quarter-circle or a wedge.

# Memory Operand Example 1

- 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 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
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, add immediate  
`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



# 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 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

# 2s-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

- $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$   
 $= -1 \times 2^{31} + 1 \times 2^{30} + \dots + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$   
 $= -2,147,483,648 + 2,147,483,644 = -4_{10}$

- Using 32 bits

- $-2,147,483,648$  to  $+2,147,483,647$

# 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

# Signed Negation

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

$$x + \bar{x} = 1111 \dots 111_2 = -1$$

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

- Example: negate +2
  - $+2 = 0000 \ 0000 \dots 0010_2$
  - $-2 = 1111 \ 1111 \dots 1101_2 + 1$   
 $= 1111 \ 1111 \dots 1110_2$

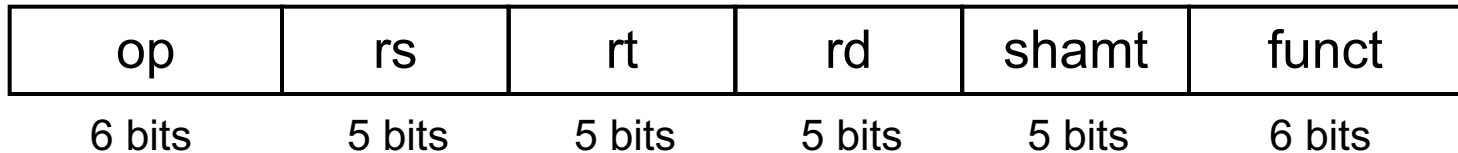
# Sign Extension

- Representing a number using more bits
  - Preserve the numeric value
- In MIPS instruction set
  - `addi`: extend immediate value
  - `lb`, `lh`: extend loaded byte/halfword
- Replicate the sign bit to the left
  - 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



- 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

$00000010001100100100000000100000_2 = 02324020_{16}$