CSE261: Computer Architecture

3. Instruction Set Architecture (2)

Seongil Wi

Department of Computer Science and Engineering

Recap: Instruction Set

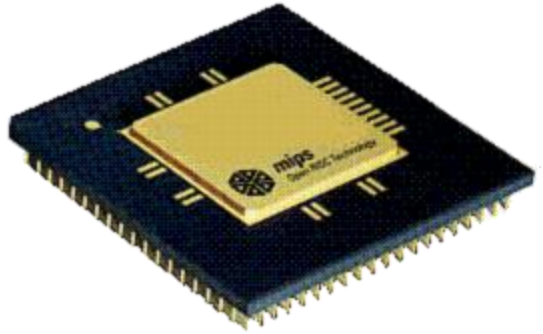
- The commands understood by a given architecture

ARM's instructions

```
pop {r0}
mov r0, r1
add r0, r0, r1
add r0, #16
```

Intel's Instructions

```
pop eax
mov eax, ebx
add eax, ebx
add eax, 0x10
```



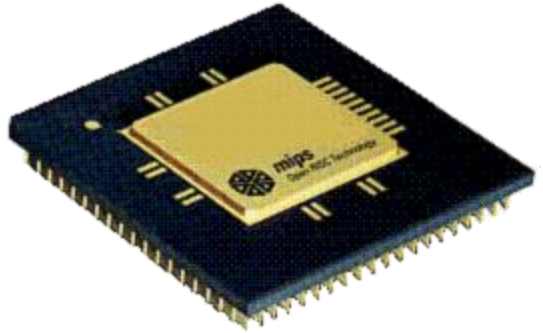
MIPS's Instructions

```
slt $t0, $s0, $s1
add $s2, $s0, $s1
sub $t2, $s1, $zero
lw  $t0, 8($s3)
```

Recap: Instruction Set

- The commands understood by a given architecture

Today's topic



MIPS's Instructions

```
slt $t0, $s0, $s1
add $s2, $s0, $s1
sub $t2, $s1, $zero
lw  $t0, 8($s3)
```

Background: 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: `0xECA8 6420`
  - 1110 1100 1010 1000 0110 0100 0010 0000

Introduction to MIPS ISA

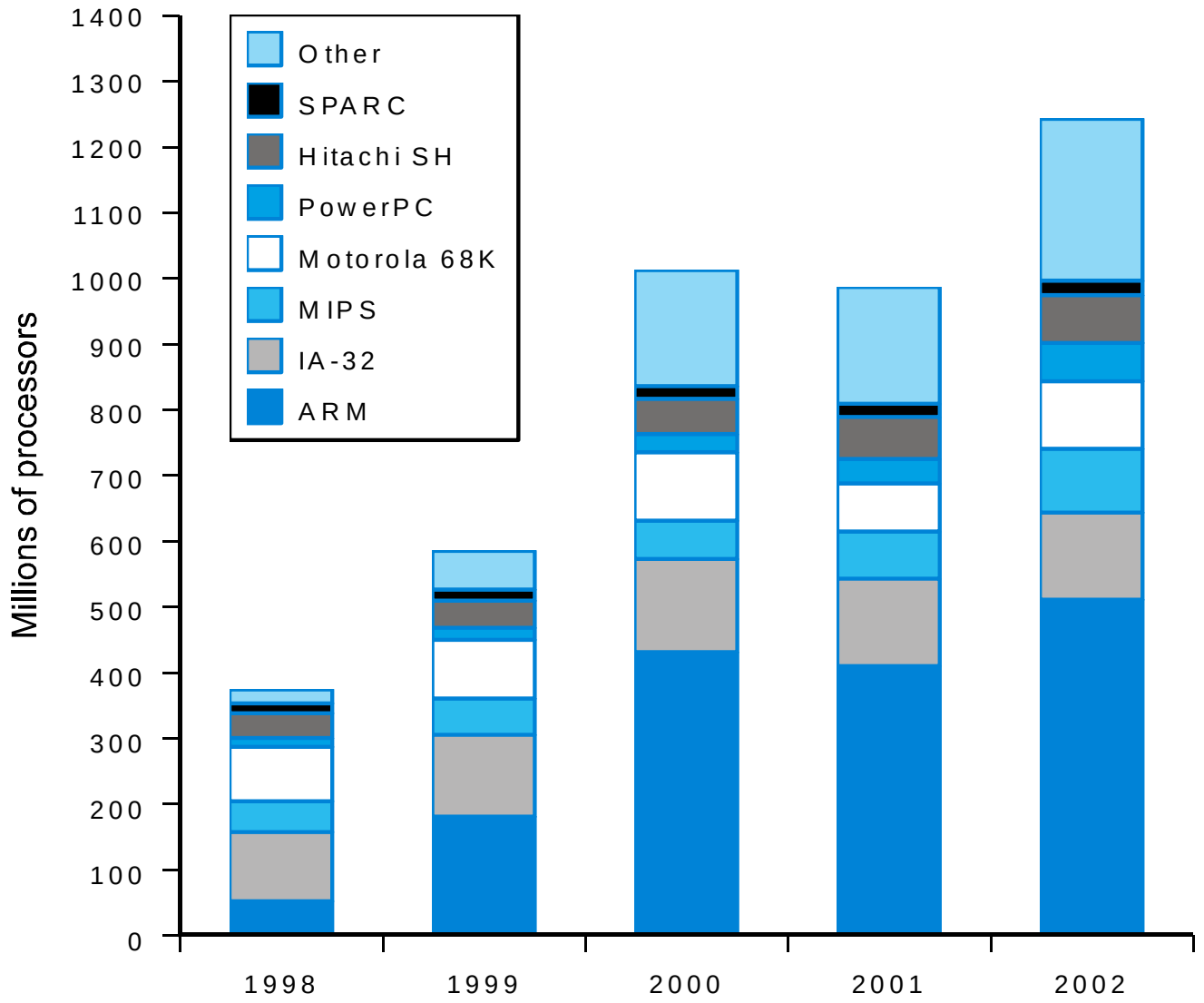
The MIPS Instruction Set

- **M**icroprocessor without **I**nterlocked **P**ipeline **S**tages (MIPS)

- Stanford MIPS commercialized by MIPS Technologies (www.mips.com)
- Typical of many modern embedded ISAs
  - Applications in consumer electronics, network/storage equipment, cameras, printers, …
  - Almost 100 million MIPS processors manufactured in 2002

- Reduced Instruction Set Computer (RISC)

The MIPS Instruction Set



MIPSA ISA Key Idea

- Goals of Instruction Set Design for MIPS
  - Maximize performance
  - Minimize cost
  - Reduce design time (of compiler and hardware)

By the Simplicity of Hardware!

MIPS Register Model

- **32** x 32 bits general purpose registers (available to programmers)
- **32** x 32 bits floating point registers
  - Paired as **16** x 64 bits for double precision
- HI (32 bits) and LO (32 bits): for multiply and divide
- PC

MIPS General Purpose Registers

#	Name	Usage
0	$zero	The constant value 0
1	$at	Assembler temporary
2	$v0	Values for results and expression evaluation
3	$v1	
4	$a0	Arguments
5	$a1	
6	$a2	
7	$a3	
8	$t0	Temporaries (Caller-save registers)
9	$t1	
10	$t2	
11	$t3	
12	$t4	
13	$t5	
14	$t6	
15	$t7	

#	Name	Usage
16	$s0	Saved temporaries (Callee-save registers)
17	$s1	
18	$s2	
19	$s3	
20	$s4	
21	$s5	
22	$s6	
23	$s7	
24	$t8	More temporaries (Caller-save registers)
25	$t9	
26	$k0	Reserved for OS kernel
27	$k1	
28	$gp	Global pointer
29	$sp	Stack pointer
30	$fp	Frame pointer
31	$ra	Return address

MIPS General Purpose Registers

#	Name	Usage
0	$zero	The constant value 0
1	$at	Assembler temporary
2	$v0	Values for results and expression evaluation
3	$v1	
4	$a0	Arguments
5	$a1	
6	$a2	
7	$a3	
8	$t0	Temporaries (Caller-save registers)
9	$t1	
14	$t6	
15	$t7	

#	Name	Usage
16	$s0	Saved temporaries (Callee-save registers)
17	$s1	
18	$s2	
19	$s3	
20	$s4	
21	$s5	
22	$s6	
23	$s7	
24	$t8	More temporaries (Caller-save registers)
25	$t9	
26	$k0	Reserved for OS kernel
27	$k1	
28	$gp	Global pointer
29	$sp	Stack pointer
30	$fp	Frame pointer
31	$ra	Return address

Used for function calls

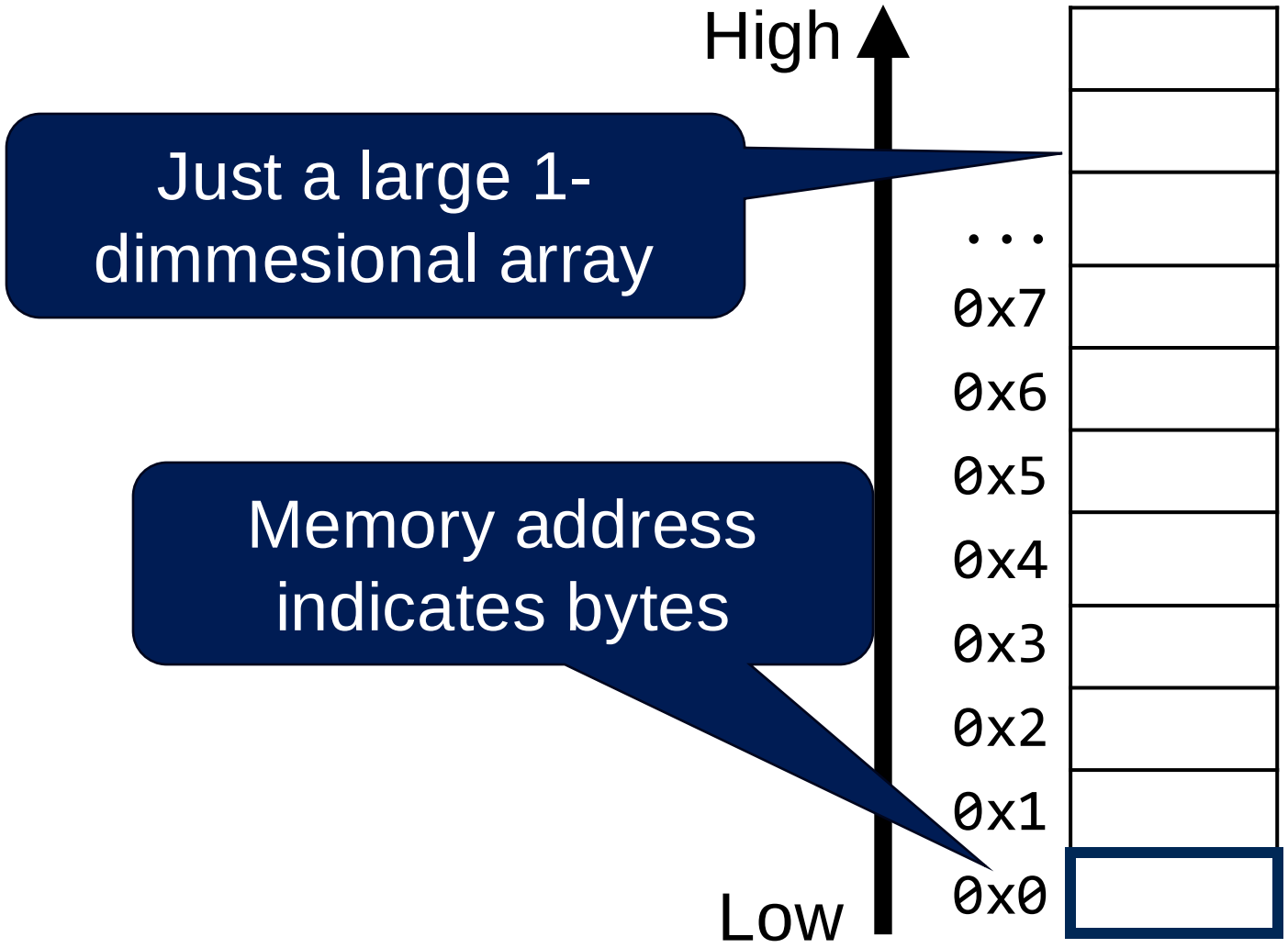
MIPS General Purpose Registers

Registers primarily used as variables in programs

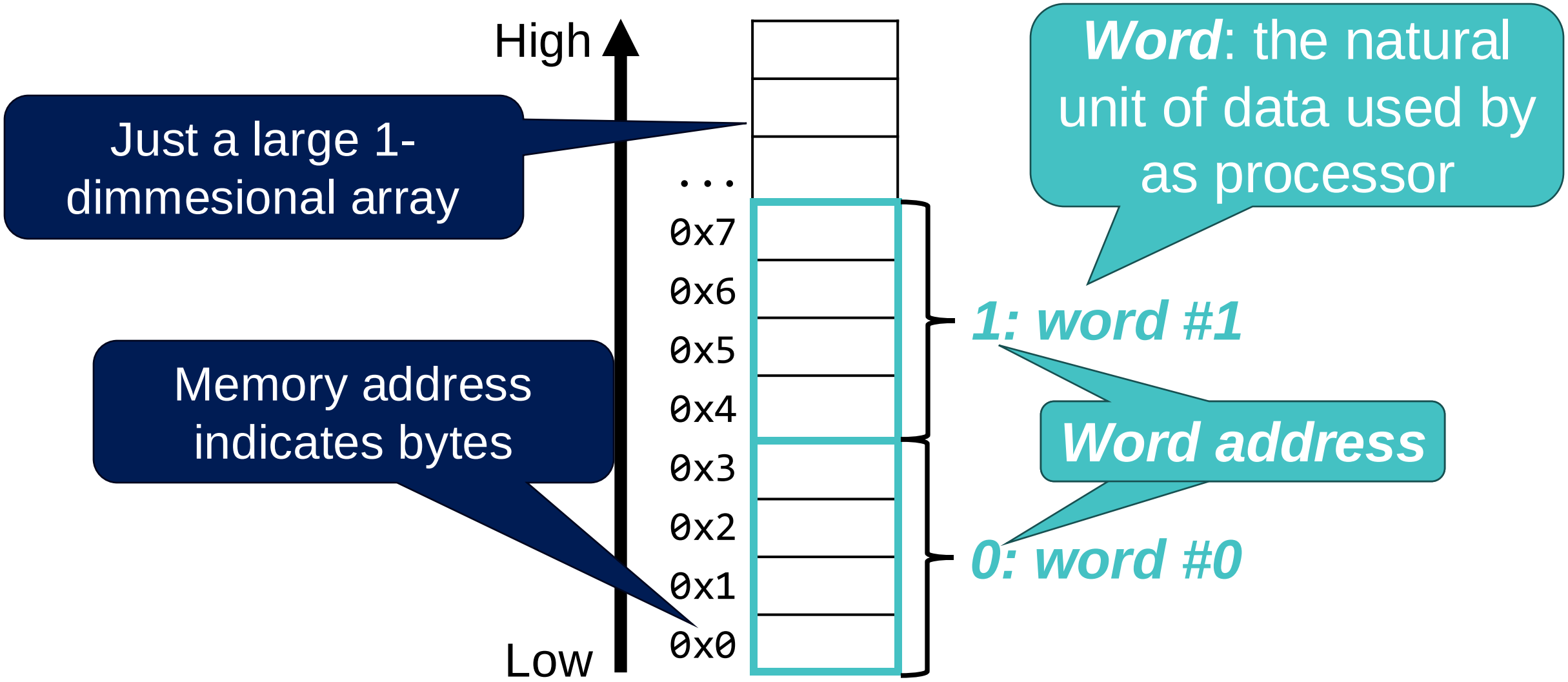
#	Name	Usage
0	$zero	The constant value 0
	$a1	
	$a2	
7	$a3	
8	$t0	Temporaries (Caller-save registers)
9	$t1	
10	$t2	
11	$t3	
12	$t4	
13	$t5	
14	$t6	
15	$t7	

#	Name	Usage
16	$s0	Saved temporaries (Callee-save registers)
17	$s1	
18	$s2	
19	$s3	
20	$s4	
21	$s5	
22	$s6	
23	$s7	
24	$t8	More temporaries (Caller-save registers)
25	$t9	
26	$k0	Reserved for OS kernel
27	$k1	
28	$gp	Global pointer
29	$sp	Stack pointer
30	$fp	Frame pointer
31	$ra	Return address

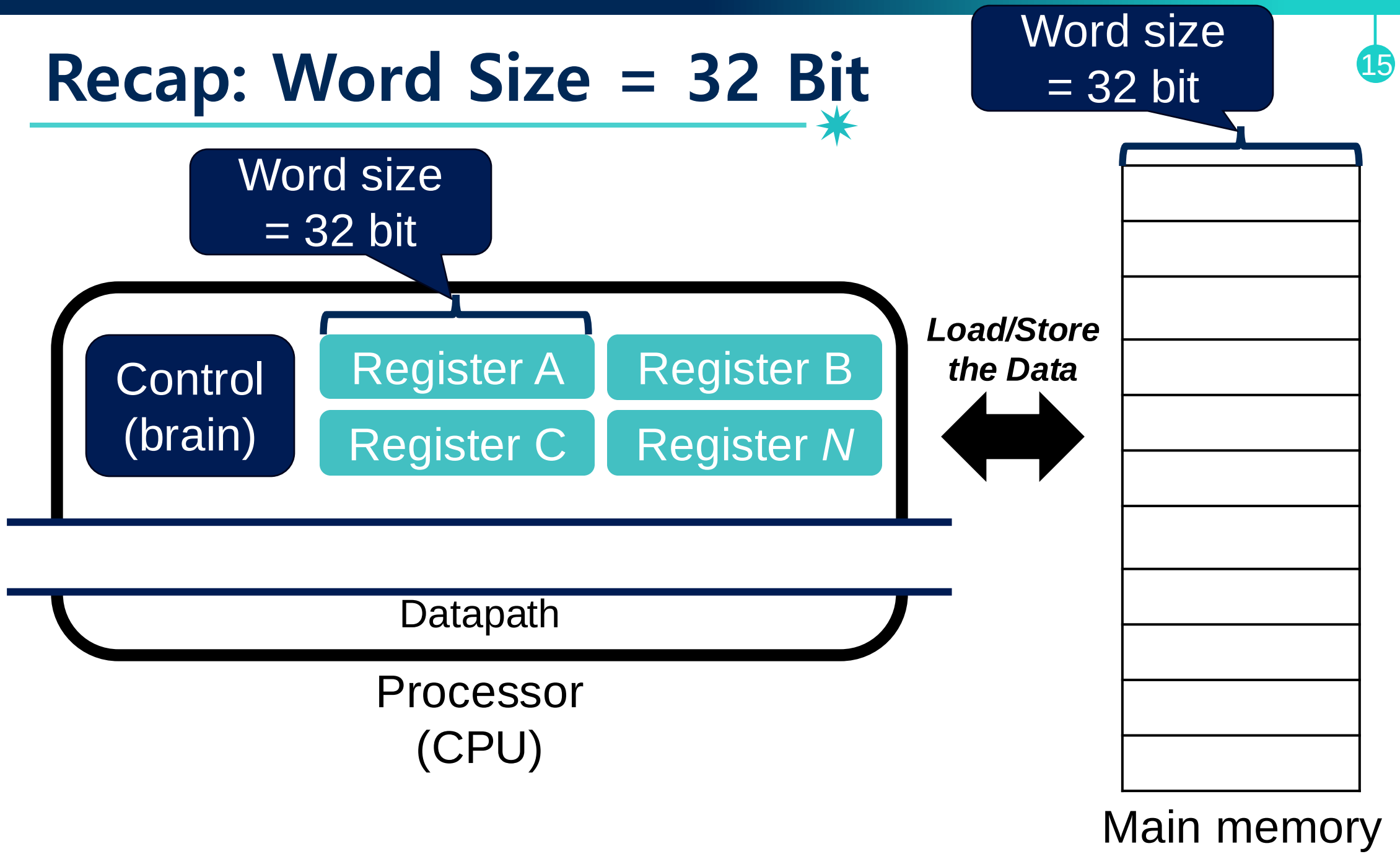
Memory Structure in MIPS



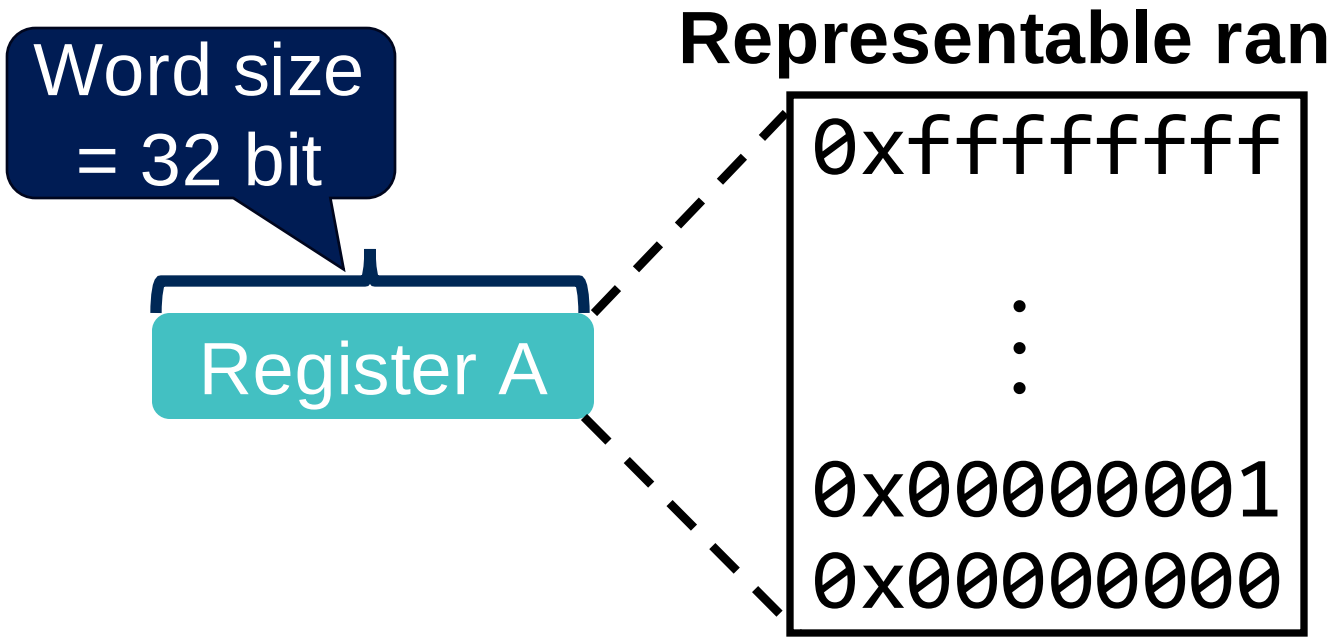
Memory Structure in MIPS



Recap: Word Size = 32 Bit



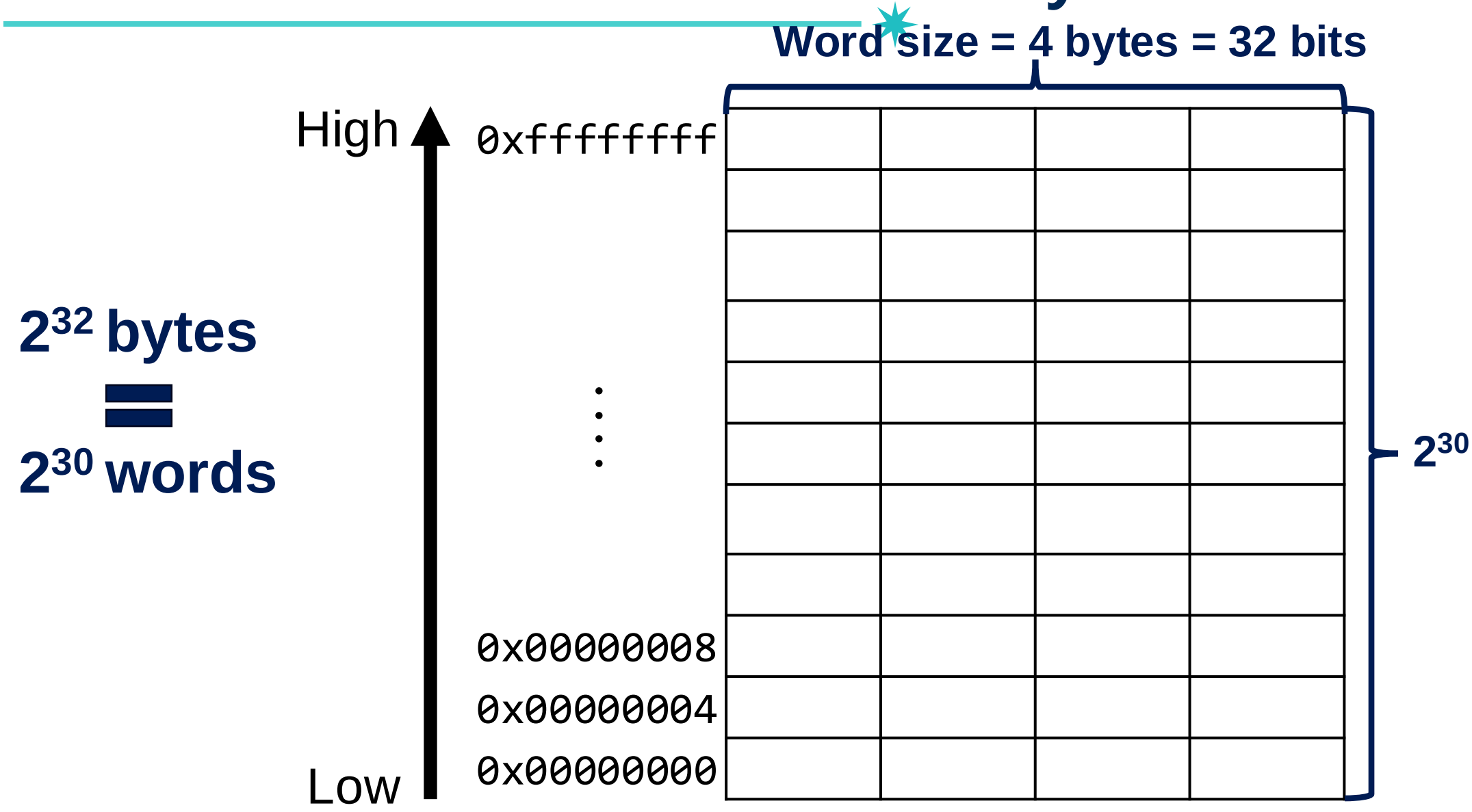
Q. What is the range of memory addresses a 32-bit machine (e.g., MIPS) can access?



Q. What is the maximum memory capacity a 32-bit machine can handle?

2^{32} bytes = 4GB

A More Intuitive View of Memory



Endianness

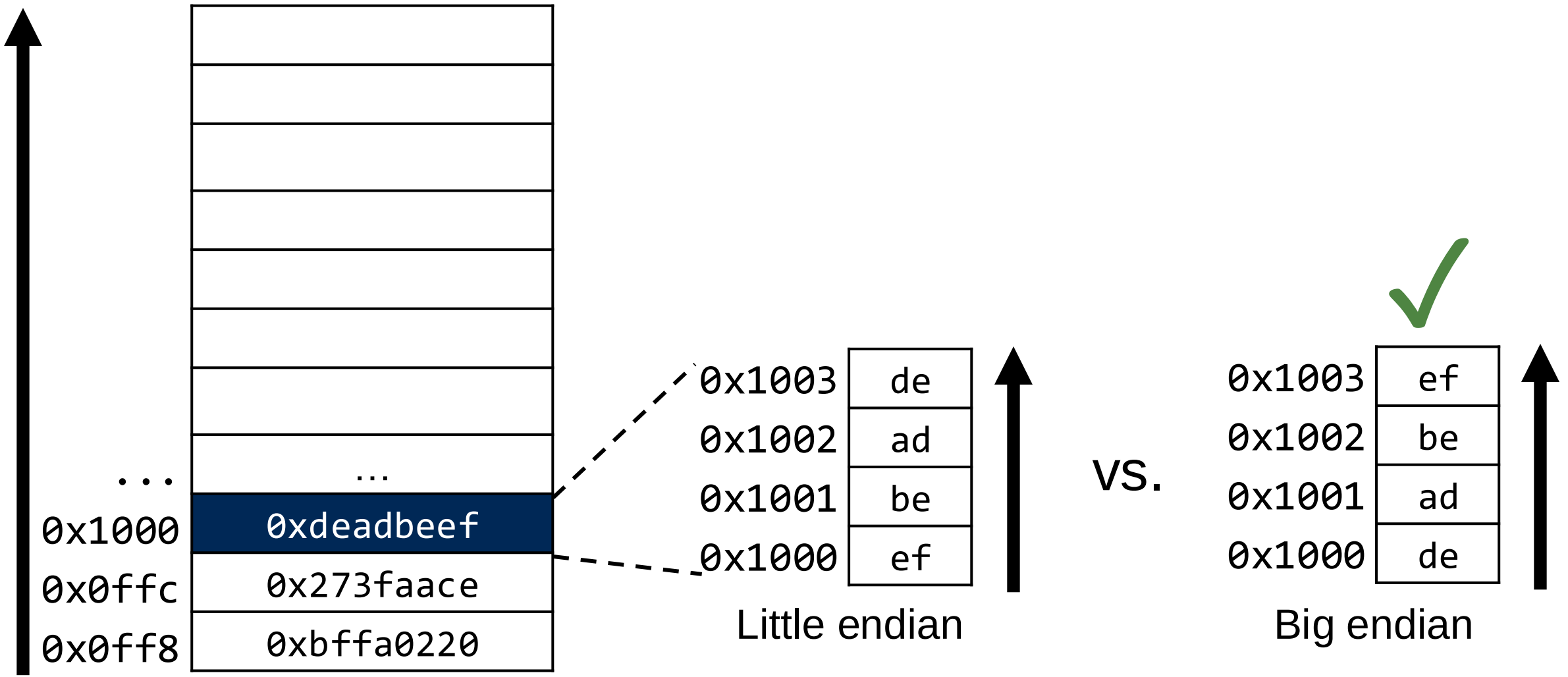
- The order in which a sequence of bytes are stored in memory

- Big Endian = The MSB goes to the lowest address
- Little Endian = The LSB goes to the lowest address

MIPS uses Big Endian

Big endian

```
sw $s0, 0($s1) ; $s0 = 0xdeadbeef, $s1 = 0x1000
```



MIPS Design Principles

MIPS Design Principles

1. Simplicity favors regularity
2. Smaller is faster
3. Make common case fast
4. Good design demands a compromise

Principle #1: Simplicity Favors Regularity

- Most of arithmetic/logic instructions have **three operands**
  - Order is fixed (destination first)

```
add   a, b, c   // a = b + c
sub   a, b, c   // a = b - c
```

One destination

Two sources

- *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 (Assembly):**

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

Principle #2: Smaller is Faster

- MIPS provides **only 32 registers** available to programmers

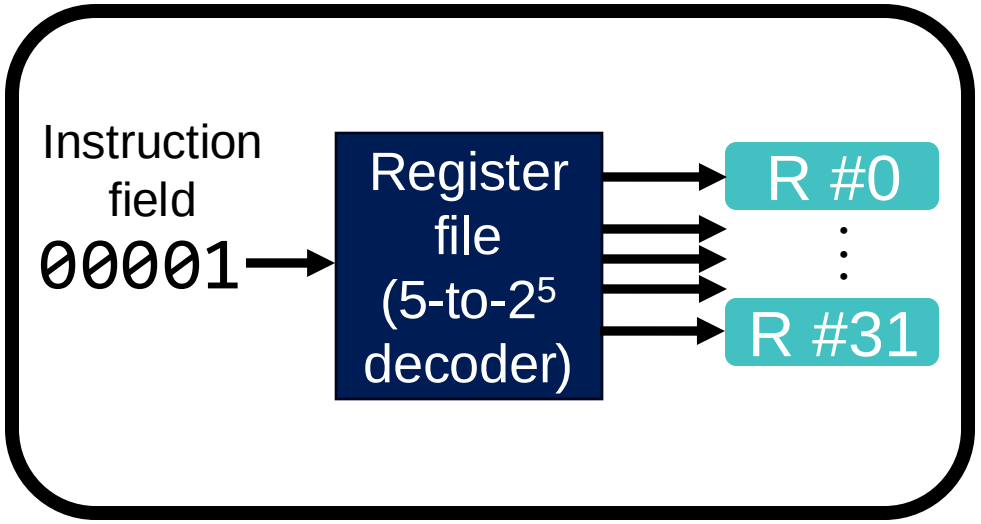
- Most of the operands of MIPS arithmetic/logic instructions are restricted to “registers” (*register addressing mode*)
  - E.g., `int a = b + c` → `add $s0,$s1,$s2`
  - Compiler associates the variables with the registers

Design Principle 2: Smaller is Faster

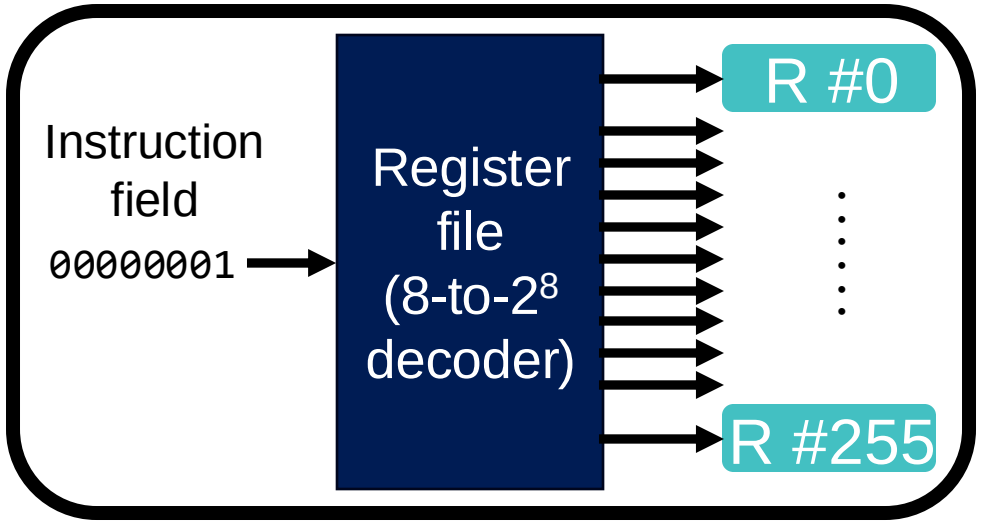
Why Only 32 Registers? (Why Smaller is Faster?)

Why not include a lot of registers inside a processor?

→ It may increase the *clock cycle period* because it takes more time for electronic signals to traverse **farther**



Processor #1



Processor #2

Register Operand Example

- **C code:**

```
f = (g + h) - (i + j);
  // Assume g,h,i,j,f are assigned to s0,s1,s2,s3,s4
```

- **Compiled MIPS code (Assembly):**

```
add t0, s0, s1    # temp t0 = g + h
add t1, s2, s3    # temp t1 = i + j
sub s4, t0, t1    # f = t0 - t1
```

Principle #3: Make Common Case Fast

- **Observation**: constants are used quite frequently as operands

```
i ++
```

```
a = b + 3
```

- **Solution**: make *constants part* of arithmetic instructions
  - E.g., `addi $s3, $s3, 4`

(Loading a constant from memory into a register can slow down the speed)

- *Design Principle 3:* Make the common case fast
  - Small constants are common
  - Immediate operand avoids a load instruction

Principle #3: Make Common Case Fast

- MIPS register 0 (`$zero`) is the constant 0
  - Cannot be overwritten (i.e., read-only)

- Useful for **common operations**
  - E.g., move between registers:

```
add $t2, $s1, $zero
```

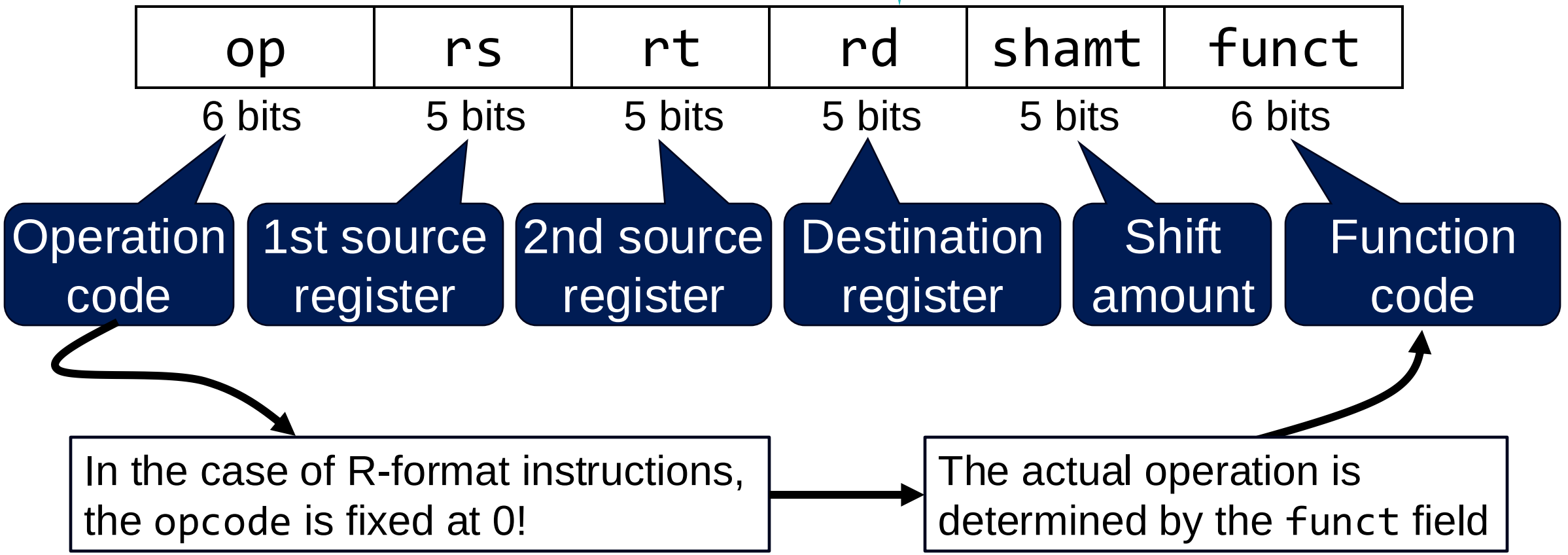
- *Design Principle 3:* Make the common case fast
  - Small constants are common
  - Immediate operand avoids a load instruction

MIPS Instruction Formats

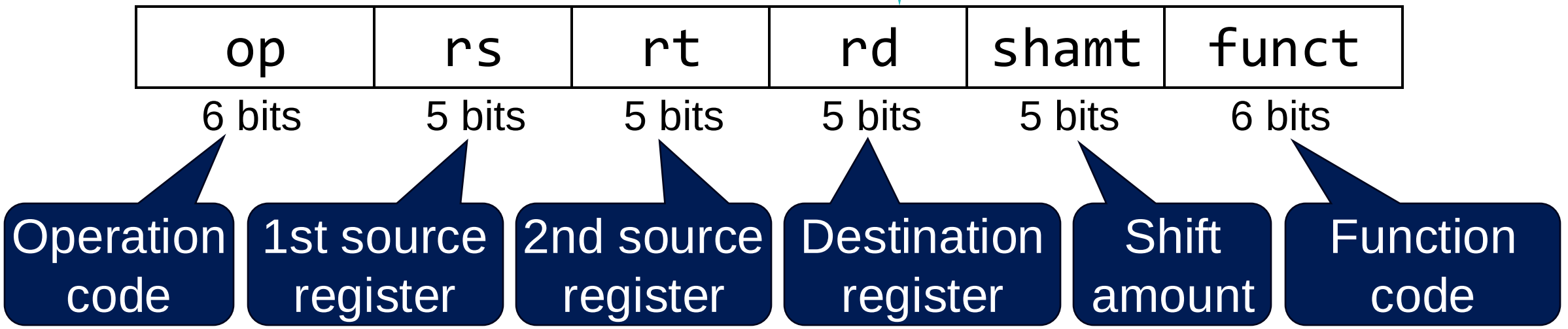
- Before talking about principle #4, let's look at the MIPS instruction formats

- There are three types of instruction formats
  - R-format instructions: `add, sub, and, or, …`
  - I-format instructions: `addi, andi, beq, …`
  - J-format instructions: `j` and `jal`

R-Format Instructions



R-Format Instructions

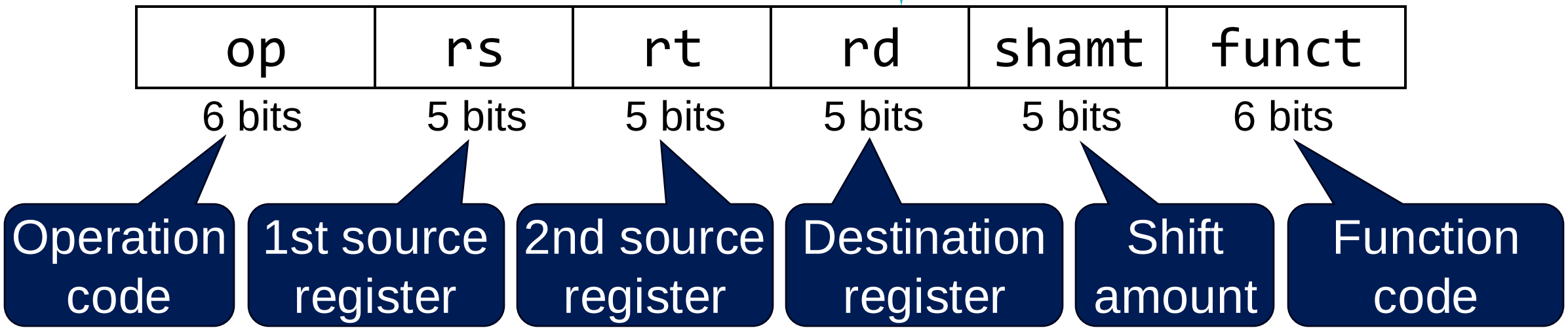


Example:

```
add $t0, $s1, $s2
```

Decimal:	0	17	18	8	0	32

R-Format Instructions



Example:

```
add $t0, $s1, $s2
```

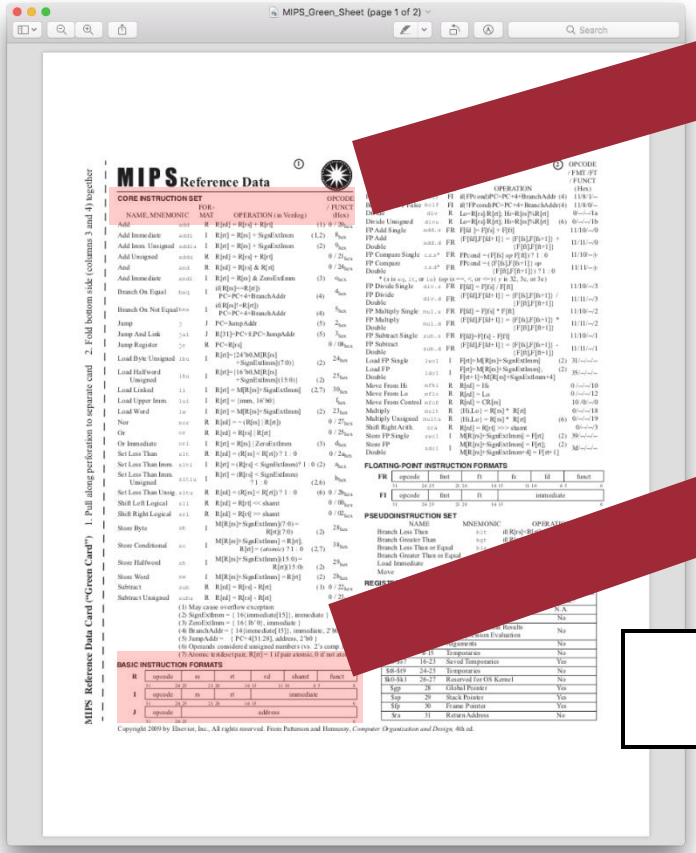
Decimal:	0	17	18	8	0	32
Binary:	000000	10001	10010	01000	00000	100000

$0000\ 0010\ 0011\ 0010\ 0100\ 0000\ 0010\ 0000_2 = 02324020_{16}$

Tip: Converting to a Machine Instruction


```
add  $t0, $s1, $s2
```

1. Refer to the ISA manual (e.g., MIPS green sheet)



2. Look up the table for add instruction

CORE INSTRUCTION SET

NAME, MNEMONIC		FOR-MAT	OPERATION (in Verilog)	OPCODE / FUNCT (Hex)
Add	add	R	R[rd] = R[rs] + R[rt]	(1) 0 / 20_{hex}

3. Encode the instruction by following the format

BASIC INSTRUCTION FORMATS

R	opcode	rs	rt	rd	shamt	funct
	31 26	25 21	20 16	15 11	10 6	0
I	opcode	rs	rt	immediate		
	31 26	25 21	20 16	0		
J	opcode	address				
	31 26	25	0			

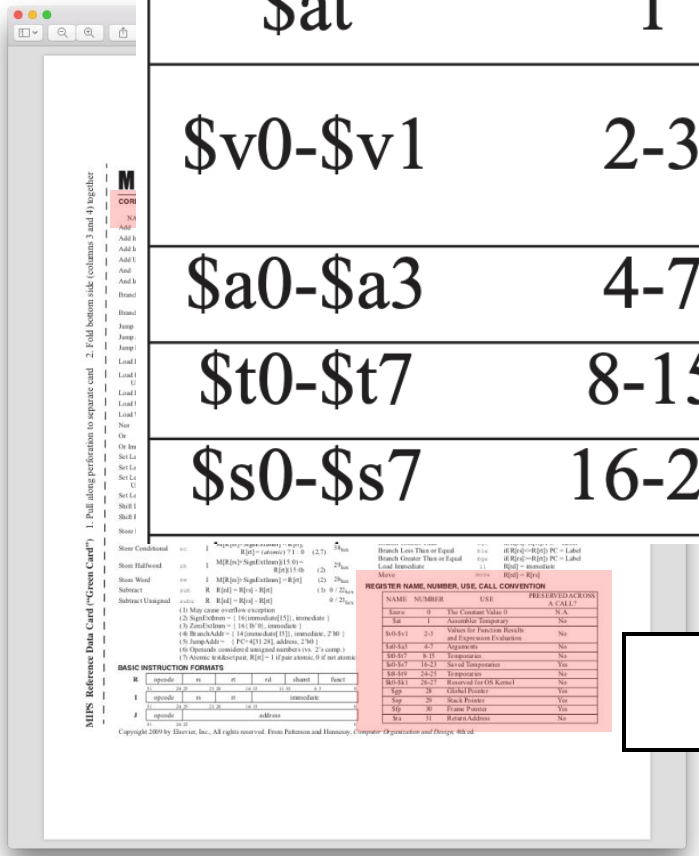
0	$s1	$s2	$t0	0	32

Tip: Converting to a Machine Instruction

add

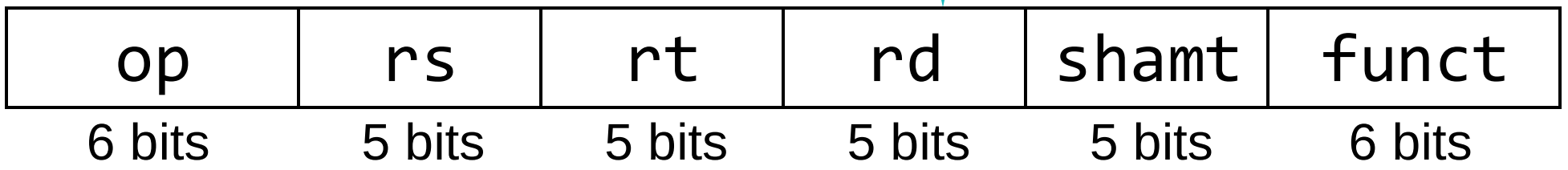
1. Re
(e.g.,

NAME	NUMBER	USE	PRESERVED ACROSS A CALL?
$zero	0	The Constant Value 0	N.A.
$at	1	Assembler Temporary	No
$v0-$v1	2-3	Values for Function Results and Expression Evaluation	No
$a0-$a3	4-7	Arguments	No
$t0-$t7	8-15	Temporaries	No
$s0-$s7	16-23	Saved Temporaries	Yes



0	~~$s1~~	~~$s2~~	~~$t0~~	0	32
	17	18	8		

Problems with R-Format Instructions



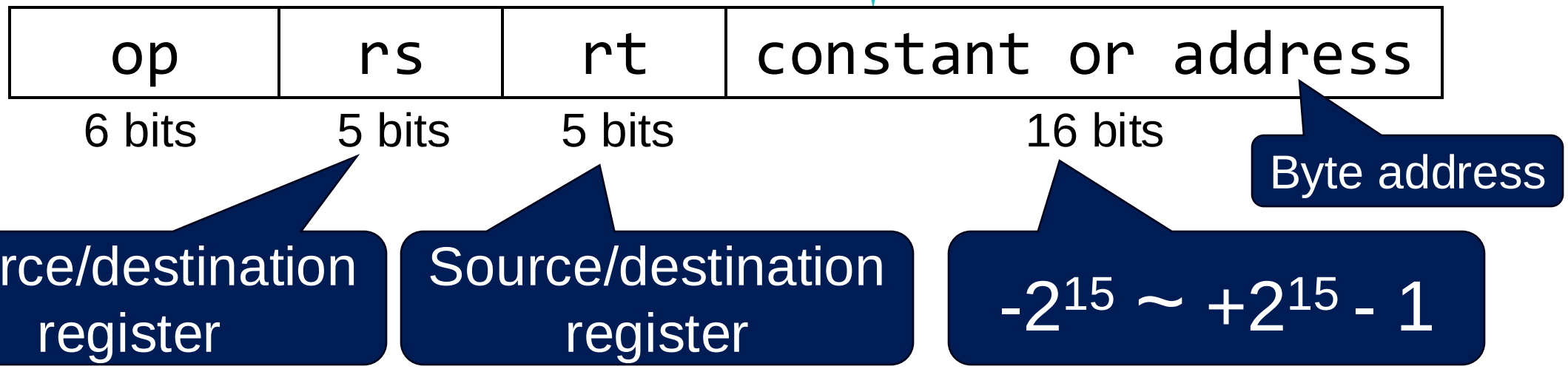
Problem: the 5-bit field is too small for a constant value

Example:

- `addi $s3, $s3, 1000`
- `lw $s0, 1000($s1)`

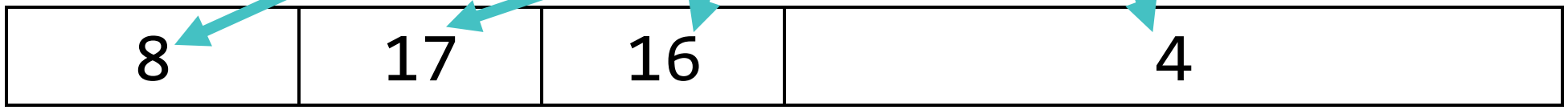
Solution: I-format instructions for immediate value

I-Format Instructions for Immediate Value

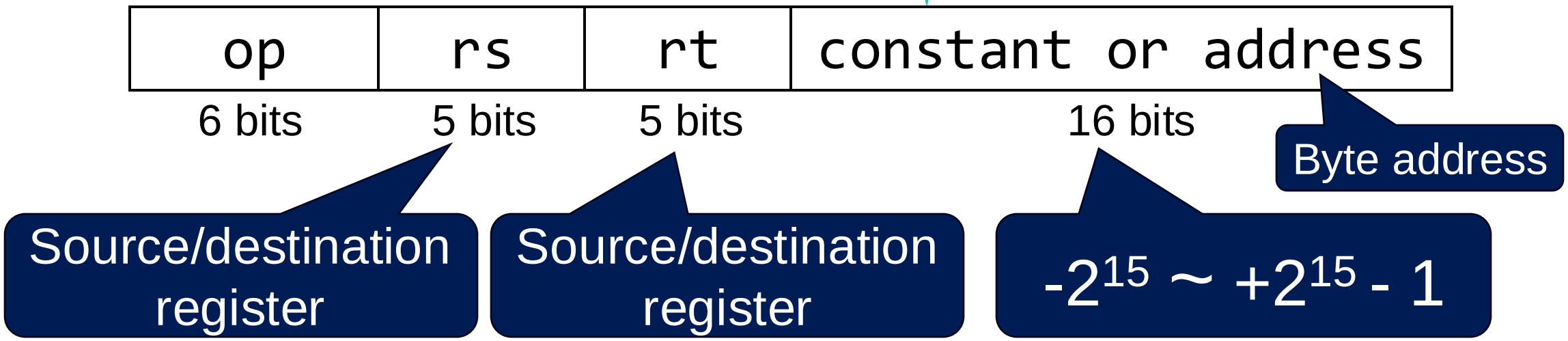


Example:

```
addi $s0, $s1, 4
```



Principle #4: Good Design demands Good Compromises



- *Design Principle 4:* Good design demands good compromises
  - Different formats complicate decoding, but keep 32-bit instructions same length (principle 1)
  - Keep formats as similar as possible

Summary: MIPS Design Principles

Principle #1: Simplicity Favors Regularity

22

- Most of arithmetic/logic instructions have **three operands**
  - Order is fixed (destination first)

```
add  a, b, c   // a = b + c
sub  a, b, c   // a = b - c
```

One destination | Two sources

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

Principle #2: Smaller is Faster

24

- MIPS provides **only 32 registers** available to programmers
- Most of the operands of MIPS arithmetic/logic instructions are restricted to "registers" (*register addressing mode*)
  - E.g., `int a = b + c` → `add $s0,$s1,$s2`
  - Compiler associates the variables with the registers

Design Principle 2: Smaller is Faster

Principle #3: Make Common Case Fast

27

- **Observation**: constants are used quite frequently as operands

```
i ++
```

```
a = b + 3
```

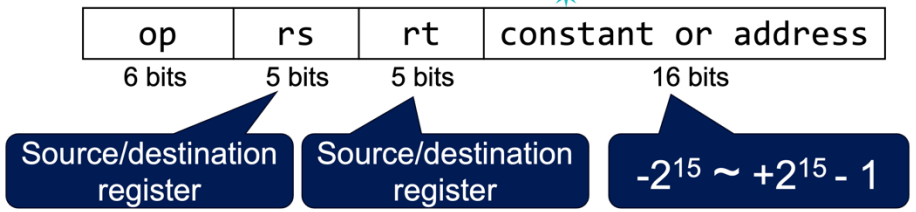
- **Solution**: make *constants part* of arithmetic instructions
  - E.g., `addi $s3, $s3, 4`

(Loading a constant from memory into a register can slow down the speed)

- *Design Principle 3:* Make the common case fast
  - Small constants are common
  - Immediate operand avoids a load instruction

Principle #4: Good Design demands Good Compromises

31



- *Design Principle 4:* Good design demands good compromises
  - Different formats complicate decoding, but keep 32-bit instructions same length (principle 1)
  - Keep formats as similar as possible

Thank You