

# EECE 2322: Fundamentals of Digital Design and Computer Organization

## Lecture 10\_1: MIPS Misc

Xiaolin Xu  
Department of ECE  
Northeastern University

---

# Exception Causes

---

Exception	Cause
Hardware Interrupt	0x00000000
System Call	0x00000020
Breakpoint / Divide by 0	0x00000024
Undefined Instruction	0x00000028
Arithmetic Overflow	0x00000030

---

# Exception Flow

---

- Processor saves cause and exception PC in `Cause` and `EPC`
- Processor jumps to exception handler (`0x80000180`)
- Exception handler:
  - Saves registers on stack
  - Reads `Cause` register

```
mfc0 $t0, Cause
```
  - Handles exception
  - Restores registers
  - Returns to program

```
mfc0 $k0, EPC  
jr $k0
```

---

# Signed & Unsigned Instructions

---

- Addition and subtraction
- Multiplication and division
- Set less than

---

# Addition & Subtraction

---

- **Signed:** `add`, `addi`, `sub`
  - Same operation as unsigned versions
  - But processor takes exception on overflow
- **Unsigned:** `addu`, `addiu`, `subu`
  - Doesn't take exception on overflow

**Note:** `addiu` sign-extends the immediate

---

# Multiplication & Division

---

- **Signed:** `mult, div`
- **Unsigned:** `multu, divu`

---

# Set Less Than

---

- **Signed:** `slt, slti`
- **Unsigned:** `sltu, sltiu`

**Note:** `sltiu` sign-extends the immediate before comparing it to the register

---

# Loads

---

- **Signed:**
  - Sign-extends to create 32-bit value to load into register
  - Load halfword: `lh`
  - Load byte: `lb`
- **Unsigned:**
  - Zero-extends to create 32-bit value
  - Load halfword unsigned: `lhu`
  - Load byte: `lbu`



---

# Floating-Point Instructions

---

- Floating-point coprocessor (Coprocessor 1)
- 32 32-bit floating-point registers (\$f0-\$f31)
- Double-precision values held in two floating point registers
  - e.g., \$f0 and \$f1, \$f2 and \$f3, etc.
  - Double-precision floating point registers: \$f0, \$f2, \$f4, etc.

---

# Floating-Point Instructions

---

Name	Register Number	Usage
\$fv0 - \$fv1	0, 2	return values
\$ft0 - \$ft3	4, 6, 8, 10	temporary variables
\$fa0 - \$fa1	12, 14	Function arguments
\$ft4 - \$ft8	16, 18	temporary variables
\$fs0 - \$fs5	20, 22, 24, 26, 28, 30	saved variables

# F-Type Instruction Format

- Opcode = 17 ( $010001_2$ )
- Single-precision:
  - cop = 16 ( $010000_2$ )
  - add.s, sub.s, div.s, neg.s, abs.s, etc.
- Double-precision:
  - cop = 17 ( $010001_2$ )
  - add.d, sub.d, div.d, neg.d, abs.d, etc.
- 3 register operands:
  - fs, ft: source operands
  - fd: destination operands

## F-Type



---

# Floating-Point Branches

---

- Set/clear condition flag: `fpcond`
  - Equality: `c.seq.s`, `c.seq.d`
  - Less than: `c.lt.s`, `c.lt.d`
  - Less than or equal: `c.le.s`, `c.le.d`
- Conditional branch
  - `bclf`: branches if `fpcond` is FALSE
  - `bclt`: branches if `fpcond` is TRUE
- Loads and stores
  - `lwc1: lwc1 $ft1, 42($s1)`
  - `swc1: swc1 $fs2, 17($sp)`

# EECE 2322: Fundamentals of Digital Design and Computer Organization

## Lecture 10\_1: MIPS Wrap-up and FSM

Xiaolin Xu

Department of ECE  
Northeastern University

---

# Today will be a Review Lecture

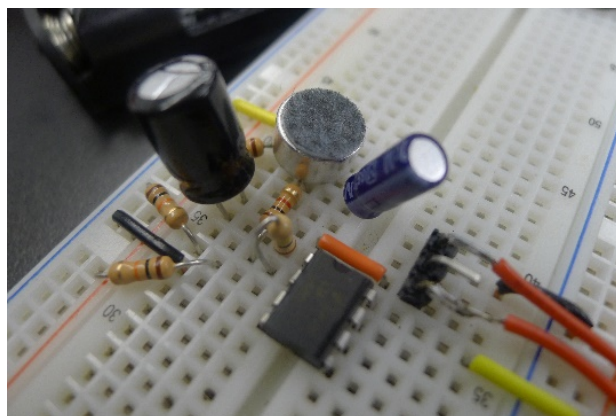
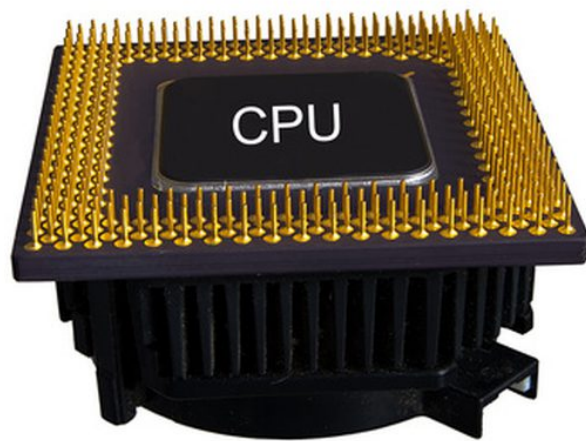
---

# From Digital Circuit to Architecture

- ❖ Jumping up a few levels of abstraction
- ❖ **Architecture:** programmer's view of computer
  - ❖ Defined by instructions & operand locations
- ❖ **Microarchitecture:** how to implement an architecture in hardware (we will learn this later)

Application Software	programs
Operating Systems	device drivers
Architecture	instructions registers
Micro-architecture	datapaths controllers
Logic	adders memories
Digital Circuits	AND gates NOT gates
Analog Circuits	amplifiers filters
Devices	transistors diodes
Physics	electrons

# Who Controls the HWs and How to?



Application/software
Operating Systems (OS)
Architecture
Micro-architecture
Logic
Digital Circuits
Analog Circuits
Devices
Physics

Programs

Drivers

Instructions

Datapath/controller

Address/memory

Gates (AND, OR)

Amplifier/filter

Transistor

Electron



---

# Architecture and Micro-architecture

---

- ❖ Architecture: programmer's view of computer
  - ❖ Defined by instructions & operand locations
- ❖ Microarchitecture: how to implement an architecture in hardware
- ❖ RISC: Reduced Instruction Set Computer
- ❖ CISC: Complex Instruction Set Computer, e.g., x86

---

# First Step to Learn any Computer Architecture

---

- ❖ Language!
- ❖ Computer's language are called *instructions*
- ❖ **Instruction set**
  - ❖ The language of computer hardware
  - ❖ Used by all programs on a computer

---

# Machine Language

---

- ❖ The main part of Instruction Set Architecture (ISA)
- ❖ Computer hardware only understands '1' and '0'
- ❖ So, all the instruction set are encoded as binary numbers — the machine language

---

# Machine Language

---

- ❖ The main part of Instruction Set Architecture (ISA)
- ❖ Computer hardware only understands '1' and '0'
- ❖ So, all the instruction set are encoded as binary numbers — the machine language

## Machine Code

op	rs	rt	rd	shamt	funct	
000000	10001	10010	10000	00000	100000	(0x02328020)
000000	01011	01101	01000	00000	100010	(0x016D4022)
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

---

# Assembly Language

---

- ❖ **Instructions:** commands in a computer's language
  - ❖ **Assembly language:** human-readable format of instructions
  - ❖ **Machine language:** computer-readable format (1's and 0's)
- ❖ **MIPS architecture:**
  - ❖ Developed by John Hennessy and his colleagues at Stanford and in the 1980's.
  - ❖ Used in many commercial systems, including Silicon Graphics, Nintendo, and Cisco

Once you've learned one architecture, it's easy to learn others

---

# MIPS

---

- ❖ **Our focus**

- ❖ **Microprocessor without Interlocked Pipeline Stages**

- ❖ Developed by John Hennessy and his colleagues at Stanford in 1980's

- ❖ Used in many commercial systems

- ❖ Similar as RISC-V

- ❖ **Underlying design principles:**

- ❖ Simplicity favors regularity

- ❖ Make the common case fast

- ❖ Smaller is faster

- ❖ Good design demands good compromises

---

# Architecture Design Principles

---

- ❖ **Underlying design principles, as articulated by Hennessy and Patterson:**
  1. **Simplicity favors regularity**
  2. **Make the common case fast**
  3. **Smaller is faster**
  4. **Good design demands good compromises**

---

# Design Principle 1

---

## **Simplicity favors regularity**

- ❖ Consistent instruction format
- ❖ Same number of operands (two sources and one destination)
- ❖ easier to encode and handle in hardware



---

# Multiple Instructions

---

- More complex code is handled by multiple MIPS instructions.

## C Code

```
a = b + c - d;
```

## MIPS assembly code

```
add t, b, c    # t = b + c  
sub a, t, d    # a = t - d
```

---

# Design Principle 2

---

## **Make the common case fast**

- ❖ MIPS includes only simple, commonly used instructions
- ❖ Hardware to decode and execute instructions can be simple, small, and fast
- ❖ More complex instructions (that are less common) performed using multiple simple instructions

---

# Design Principle 2

---

## Make the common case fast

- ❖ MIPS is a *reduced instruction set computer (RISC)*, with a small number of simple instructions
- ❖ Other architectures, such as Intel's x86, are *complex instruction set computers (CISC)*
- ❖ Tradeoffs
  - ❖
  - ❖
  - ❖

---

# Design Principle 2

---

## Make the common case fast

- ❖ MIPS is a *reduced instruction set computer (RISC)*, with a small number of simple instructions
- ❖ Other architectures, such as Intel's x86, are *complex instruction set computers (CISC)*
- ❖ **Tradeoffs**
  - ❖ CISC has implement highly complex instructions — high cost (overhead)
  - ❖ RISC achieves the same with MANY small instructions — low cost
  - ❖ What more? Faster VS. Slower! — Rarely used instruction consumes HW!

---

# Operands

---

- ❖ Operand location: physical location in computer
  - ❖ Registers
  - ❖ Memory
  - ❖ Constants (also called immediates)
- ❖ Why a computer needs physical location?
  - ❖ Computer only calculates using 1s and 0s, not a, b, c
  - ❖ It needs to know where to find the variables

---

# Operands: Registers

---

- ❖ MIPS has 32 32-bit registers
- ❖ Registers are faster than memory
- ❖ MIPS called “32-bit architecture”
  - ❖ Because it operates on 32-bit data

---

# Register

---

- ❖ \$ before name
- ❖ Some registers used for specific purposes:
  - ❖ \$0 always holds the constant value 0
  - ❖ the saved registers, \$s0-\$s7, used to hold variables
  - ❖ the temporary registers, \$t0 - \$t9, used to hold intermediate values during a larger computation

---

# Design Principle 3

---

## Smaller is Faster

- ❖ MIPS includes only a small number of registers



---

# Operands: Registers

---

- Registers:
  - \$ before name
  - Example: \$0, “register zero”, “dollar zero”
- Registers used for specific purposes:
  - \$0 always holds the constant value 0.
  - the *saved registers*, \$s0-\$s7, used to hold variables
  - the *temporary registers*, \$t0 - \$t9, used to hold intermediate values during a larger computation
  - Discuss others later

# MIPS Register Set

Name	Register Number	Usage
<b>\$0</b>	0	the constant value 0
<b>\$at</b>	1	assembler temporary
<b>\$v0-\$v1</b>	2-3	Function return values
<b>\$a0-\$a3</b>	4-7	Function arguments
<b>\$t0-\$t7</b>	8-15	temporaries
<b>\$s0-\$s7</b>	16-23	saved variables
<b>\$t8-\$t9</b>	24-25	more temporaries
<b>\$k0-\$k1</b>	26-27	OS temporaries
<b>\$gp</b>	28	global pointer
<b>\$sp</b>	29	stack pointer
<b>\$fp</b>	30	frame pointer
<b>\$ra</b>	31	Function return address

---

# Instructions with Registers

---

## ❖ Revisit add instruction

### C Code

```
a = b + c
```

### MIPS assembly code

```
# $s0 = a, $s1 = b, $s2 = c  
add __, ____, $s2
```

---

# Instructions with Registers

---

## ❖ Revisit add instruction

### C Code

```
a = b + c
```

### MIPS assembly code

```
# $s0 = a, $s1 = b, $s2 = c  
add $s0, $s1, $s2
```

---

# From High-level Code to Assembly Language

---

## MIPS assembly code

```
# assuming that a-to-c are stored in $s0-  
to-$s2, f-to-j are stored in $s3-to-$s7
```

```
# $s0 = a, $s1 = b, $s2 = c, $s3 = f, $s4 =  
g, $s5 = h # $s6 = i, $s7 = j
```

How to convert the C code to assembly code?

## C Code

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

```
_____ #a=b-c  
_____ #t0=g+h  
_____ #t1=i+j  
_____ #f=(g+h)-(i+j)
```

---

# From High-level Code to Assembly Language

---

## MIPS assembly code

```
# assuming that a-to-c are stored in $s0-  
to-$s2, f-to-j are stored in $s3-to-$s7
```

```
# $s0 = a, $s1 = b, $s2 = c, $s3 = f, $s4 =  
g, $s5 = h # $s6 = i, $s7 = j
```

How to convert the C code to assembly code?

## C Code

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

sub \$s0, \$s1, \$s2 #a=b-c	_____	#a=b-c
add \$t0, \$s4, \$s5 # \$t0=g+h	_____	# \$t0=g+h
add \$t1, \$s6, \$s7 # \$t1=i+j	_____	# \$t1=i+j
sub \$s3, \$t0, \$t1 #f=(g+h)-(i+j)	_____	#f=(g+h)-(i+j)

# MIPS Instructions: Arithmetic

Instruction	Example	Meaning	Comments
<b>add</b>	<code>add \$1, \$2, \$3</code>	$\$1 = \$2 + \$3$	
<b>subtract</b>	<code>sub \$1, \$2, \$3</code>	$\$1 = \$2 - \$3$	
<b>add immediate</b>	<code>addi \$1, \$2, 100</code>	$\$1 = \$2 + 100$	"Immediate" means a constant number
<b>add unsigned</b>	<code>addu \$1, \$2, \$3</code>	$\$1 = \$2 + \$3$	Values are treated as unsigned integers, not two's complement integers
<b>subtract unsigned</b>	<code>subu \$1, \$2, \$3</code>	$\$1 = \$2 - \$3$	Values are treated as unsigned integers, not two's complement integers

# MIPS Instructions: Arithmetic

<b>add immediate unsigned</b>	<code>addiu \$1, \$2, 100</code>	$\$1 = \$2 + 100$	Values are treated as unsigned integers, not two's complement integers
<b>Multiply (without overflow)</b>	<code>mul \$1, \$2, \$3</code>	$\$1 = \$2 * \$3$	Result is only 32 bits!
<b>Multiply</b>	<code>mult \$2, \$3</code>	$\$hi, \$lo = \$2 * \$3$	Upper 32 bits stored in special register <code>hi</code> Lower 32 bits stored in special register <code>lo</code>
<b>Divide</b>	<code>div \$2, \$3</code>	$\$hi, \$lo = \$2 / \$3$	Remainder stored in special register <code>hi</code> Quotient stored in special register <code>lo</code>



# MIPS Instructions: Logical

Instruction	Example	Meaning	Comments
<b>and</b>	<code>and \$1, \$2, \$3</code>	$\$1 = \$2 \& \$3$	Bitwise AND
<b>or</b>	<code>or \$1, \$2, \$3</code>	$\$1 = \$2   \$3$	Bitwise OR
<b>and immediate</b>	<code>andi \$1, \$2, 100</code>	$\$1 = \$2 \& 100$	Bitwise AND with immediate value
<b>or immediate</b>	<code>or \$1, \$2, 100</code>	$\$1 = \$2   100$	Bitwise OR with immediate value
<b>shift left logical</b>	<code>sll \$1, \$2, 10</code>	$\$1 = \$2 \ll 10$	Shift left by constant number of bits
<b>shift right logical</b>	<code>srl \$1, \$2, 10</code>	$\$1 = \$2 \gg 10$	Shift right by constant number of bits

# MIPS Instructions: Data Transfer

Instruction	Example	Meaning	Comments
<b>load word</b>	<code>lw \$1, 100(\$2)</code>	$\$1 = \text{Memory}[\$2 + 100]$	Copy from memory to register
<b>store word</b>	<code>sw \$1, 100(\$2)</code>	$\text{Memory}[\$2 + 100] = \$1$	Copy from register to memory
<b>load upper immediate</b>	<code>lui \$1, 100</code>	$\$1 = 100 \times 2^{16}$	Load constant into upper 16 bits. Lower 16 bits are set to zero.
<b>load address</b>	<code>la \$1, label</code>	$\$1 = \text{Address of label}$	<i>Pseudo-instruction</i> (provided by assembler, not processor!) Loads computed address of label (not its contents) into register
<b>load immediate</b>	<code>li \$1, 100</code>	$\$1 = 100$	<i>Pseudo-instruction</i> (provided by assembler, not processor!) Loads immediate value into register

# MIPS Instructions: Data Transfer

<b>move from hi</b>	<code>mfhi \$2</code>	<b>\$2=hi</b>	Copy from special register <code>hi</code> to general register
<b>move from lo</b>	<code>mflo \$2</code>	<b>\$2=lo</b>	Copy from special register <code>lo</code> to general register
<b>move</b>	<code>move \$1, \$2</code>	<b>\$1=\$2</b>	<i>Pseudo-instruction</i> (provided by assembler, not processor!) Copy from register to register.

---

# MIPS Instructions: Others

---

- ❖ See reference: [https://www.dsi.unive.it/~gasparetto/materials/MIPS\\_Instruction\\_Set.pdf](https://www.dsi.unive.it/~gasparetto/materials/MIPS_Instruction_Set.pdf)
- ❖ Un/Conditional Branch: enabling if-else, etc.
- ❖ System call: print/read/exit, etc.

---

# Operands: Memory

---

- ❖ Too much data to fit in only 32 registers
- ❖ Store more data in memory
  - ❖ However, Memory is large, but slow
- ❖ Commonly used variables kept in registers

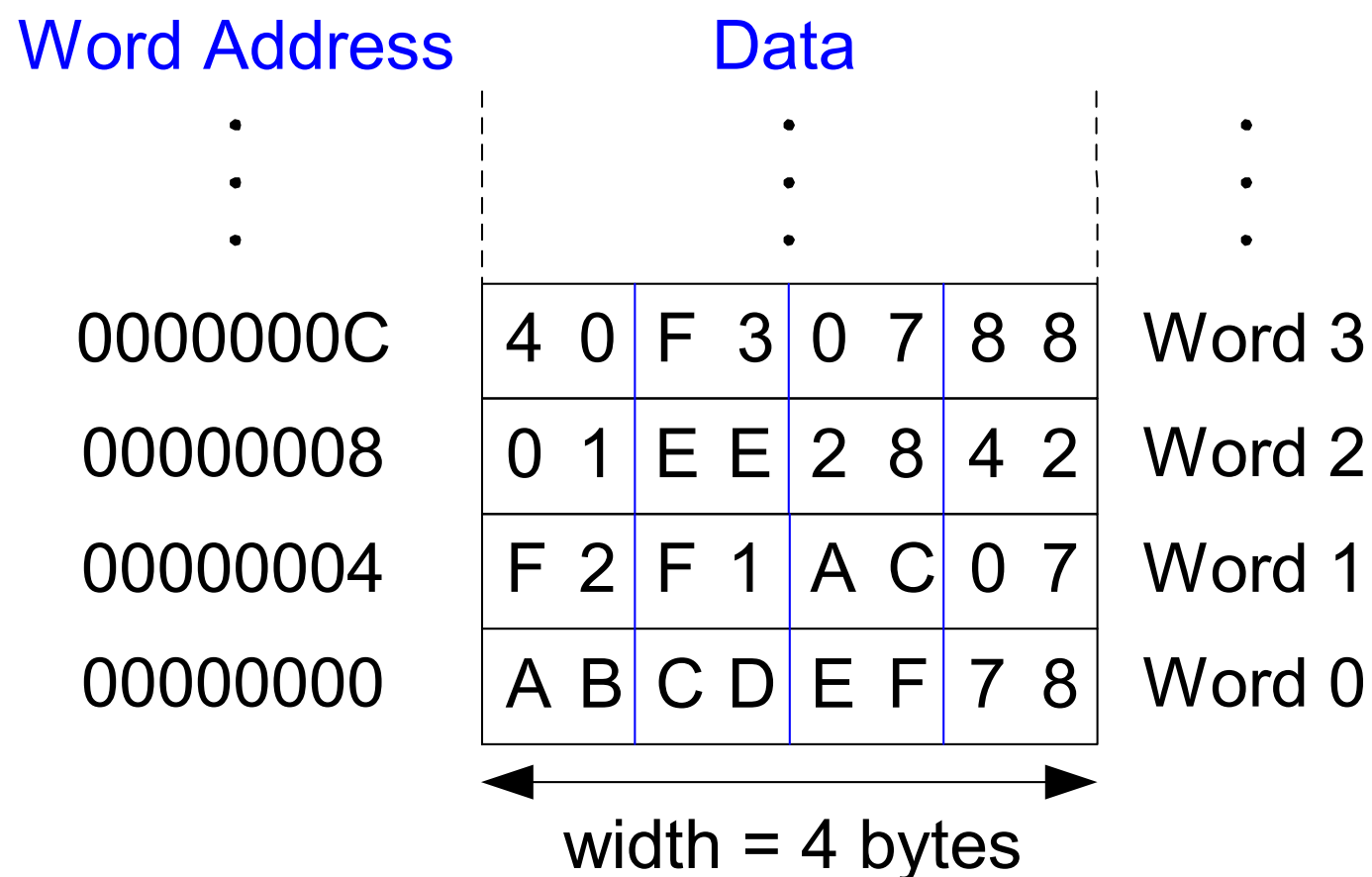
# Word-Addressable Memory

- Each 32-bit data word has a unique address

Word Address	Data	
⋮	⋮	⋮
00000003	4 0 F 3 0 7 8 8	Word 3
00000002	0 1 E E 2 8 4 2	Word 2
00000001	F 2 F 1 A C 0 7	Word 1
00000000	A B C D E F 7 8	Word 0

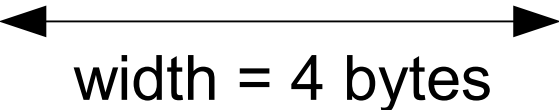
# Byte-Addressable Memory

- Each data byte has unique address
- Load/store words or single bytes: load byte (lb) and store byte (sb)
- 32-bit word = 4 bytes, so word address increments by 4



# Word / Byte-Addressable Memory

Word Address	Data		Word Address	Data	
⋮	⋮	⋮	⋮	⋮	⋮
00000003	4 0 F 3 0 7 8 8	Word 3	0000000C	4 0 F 3 0 7 8 8	Word 3
00000002	0 1 E E 2 8 4 2	Word 2	00000008	0 1 E E 2 8 4 2	Word 2
00000001	F 2 F 1 A C 0 7	Word 1	00000004	F 2 F 1 A C 0 7	Word 1
00000000	A B C D E F 7 8	Word 0	00000000	A B C D E F 7 8	Word 0





---

# Reading Byte-Addressable Memory

---

- The address of a memory word must now be multiplied by 4. For example,
  - the address of memory word 2 is  $2 \times 4 = 8$
  - the address of memory word 10 is  $10 \times 4 = 40$  (0x28)
- **MIPS is byte-addressed, not word-addressed**

# Reading Byte-Addressable Memory

- **Example:** Load a word of data at memory address 4 into `$s3`.
- `$s3` holds the value       ? after load

## MIPS assembly code

```
lw $s3, 4($0) # read word at address 4 into $s3
```

Word Address	Data								
⋮	⋮								⋮
0000000C	4	0	F	3	0	7	8	8	Word 3
00000008	0	1	E	E	2	8	4	2	Word 2
00000004	F	2	F	1	A	C	0	7	Word 1
00000000	A	B	C	D	E	F	7	8	Word 0

← width = 4 bytes →

# Reading Byte-Addressable Memory

- **Example:** Load a word of data at memory address 4 into `$s3`.
- `$s3` holds the value `0xF2F1AC07` after load

## MIPS assembly code

```
lw $s3, 4($0) # read word at address 4 into $s3
```

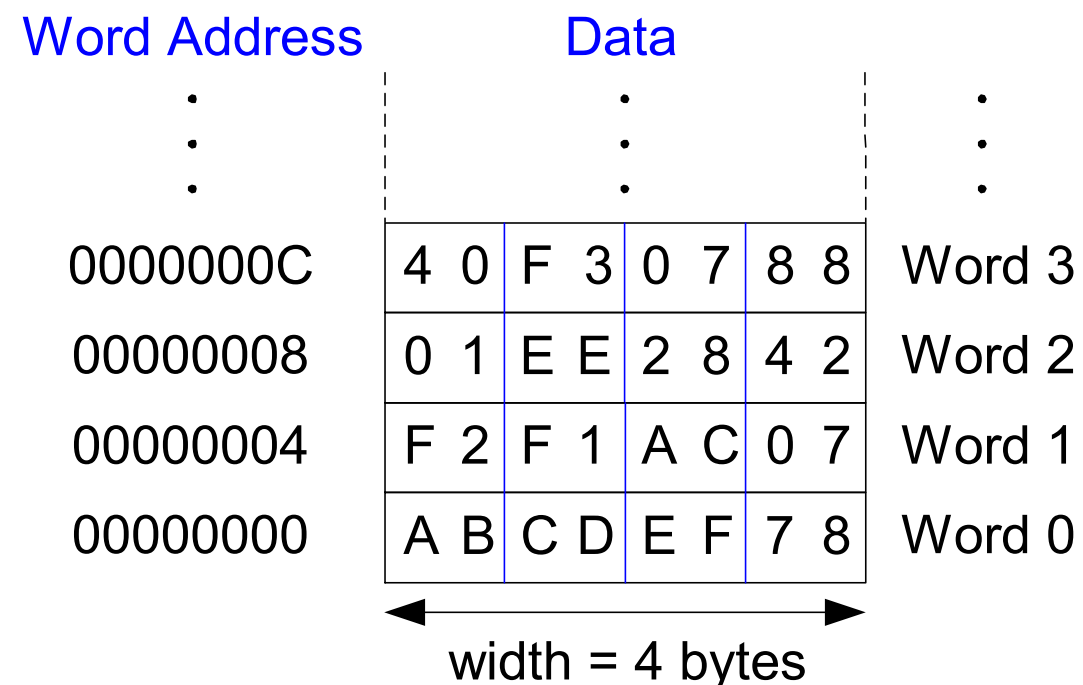
Word Address	Data								
⋮	⋮								⋮
0000000C	4	0	F	3	0	7	8	8	Word 3
00000008	0	1	E	E	2	8	4	2	Word 2
00000004	F	2	F	1	A	C	0	7	Word 1
00000000	A	B	C	D	E	F	7	8	Word 0
<div>← width = 4 bytes →</div>									

# Writing Byte-Addressable Memory

- **Example:** stores the value held in  $\$t7$  into memory address 0x2C (44)

## MIPS assembly code

```
sw $t7, 44($0)    # write $t7 into address 44
```



# Writing Byte-Addressable Memory

- **Example:** stores the value held in  $\$t7$  into memory address 0x2C (44)

## MIPS assembly code

```
sw $t7, 44($0)    # write $t7 into address 44
```

*where is address 44?*

Word Address	Data								
⋮	⋮								
0000000C	4	0	F	3	0	7	8	8	Word 3
00000008	0	1	E	E	2	8	4	2	Word 2
00000004	F	2	F	1	A	C	0	7	Word 1
00000000	A	B	C	D	E	F	7	8	Word 0

width = 4 bytes

# Practice: Reading / Writing Byte-addressable Memory

Word Address	Data	
⋮	⋮	⋮
0000000C	4 0 F 3 0 7 8 8	Word 3
00000008	0 1 E E 2 8 4 2	Word 2
00000004	F 2 F 1 A C 0 7	Word 1
00000000	A B C D E F 7 8	Word 0

← width = 4 bytes →

## MIPS assembly code

```
lw $s0, 0($0)
lw $s1, 8($0)
lw $s2, 0xC($0)
sw $s3, 4($0)
sw $s4, 0x20($0)
sw $s5, 400($0)
```

## MIPS assembly code

```
# read data word 0 (0xABCDEF78) into $s0
# read data word 2 (0x01EE2842) into $s1
# read data word 3 (0x40F30788) into $s2
# write $s3 to data word 1
# write $s4 to data word 8
# write $s5 to data word 100
```

---

# Problematic / Illegal Usage

---

❖ Why?

**MIPS assembly code**

```
lw $s0, 7($0)
```

---

# Problematic / Illegal Usage

---

## ❖ Why?

### **MIPS assembly code**

```
lw $s0, 7($0)
```

In the MIPS architecture, word addresses for lw and sw must be word aligned. That is, the address must be divisible by 4



---

# Big-Endian & Little-Endian Memory

---

- How to number bytes within a word?
- **Little-endian:** byte numbers start at the little (least significant) end
- **Big-endian:** byte numbers start at the big (most significant) end
- Difference:
  - Byte address start from where?

---

# Big-Endian & Little-Endian Memory

---

- It doesn't really matter which addressing type used – except when the two systems need to share data!
- *Computer itself not get confused at all!*

# Big-Endian & Little-Endian Example

- Suppose `$t0` initially contains `0x23456789`
- After following code runs on big-endian system, what value is `$s0`?
- In a little-endian system?

```
sw $t0, 0($0)
```

```
lb $s0, 1($0)
```

- Big-endian: `0x00000045`
- Little-endian: `0x00000067`

## Big-Endian

Byte Address	0	1	2	3
Data Value	23	45	67	89
	MSB			LSB

Word  
Address  
0

## Little-Endian

Byte Address	3	2	1	0
Data Value	23	45	67	89
	MSB			LSB

---

# Design Principle 4

---

## Good design demands good compromises

- ❖ Multiple instruction formats allow flexibility
  - add, sub: use 3 register operands
  - lw, sw: use 2 register operands and a constant
- ❖ Number of instruction formats kept small
  - to adhere to design principles 1 and 3 (simplicity favors regularity and smaller is faster).

---

# Machine Language

---

- ❖ Binary representation of instructions
- ❖ Computers only understand 1's and 0's
- ❖ 32-bit instructions
  - ❖ Simplicity favors regularity: 32-bit data & instructions
- ❖ 3 instruction formats:
  - ❖ **R-Type:** register operands
  - ❖ **I-Type:** immediate operand
  - ❖ **J-Type:** for jumping (discuss later)

---

# R-Type

---

- *Register-type*
- 3 register operands:
  - rs, rt: source registers
  - rd: destination register
- Other fields:
  - op: the *operation code* or *opcode* (0 for R-type instructions)
  - funct: the *function*
    - with opcode, tells computer what operation to perform
  - shamt: the *shift amount* for shift instructions, otherwise it's 0

## R-Type



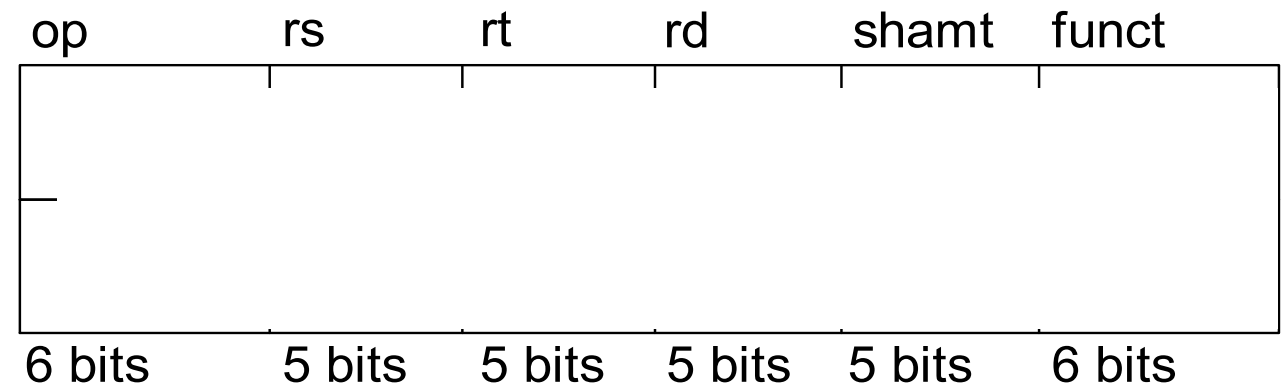
# R-Type Examples

## Assembly Code

`add $s0, $s1, $s2`

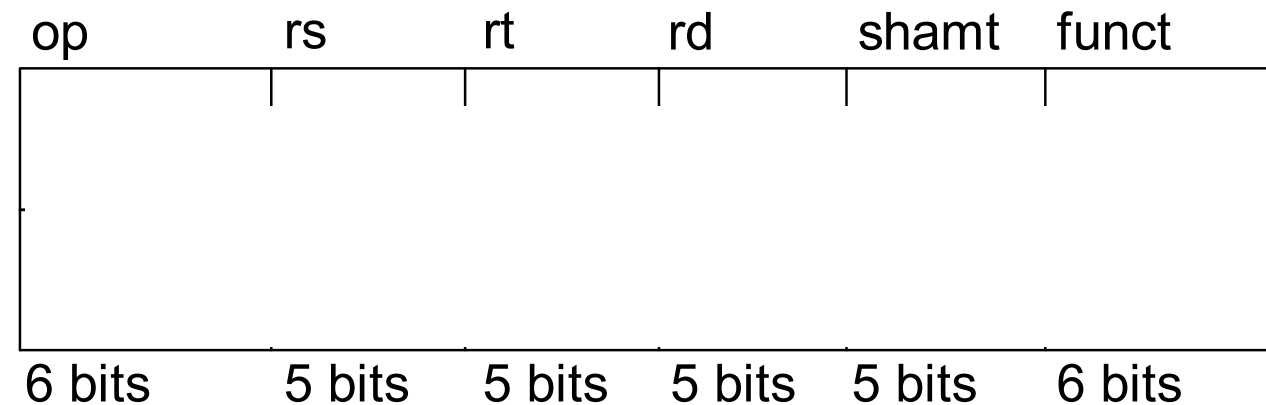
`sub $t0, $t3, $t5`

## Field Values



**add** has op code of 0 and funct code of 32

## Machine Code



**Note** the order of registers in the assembly code:

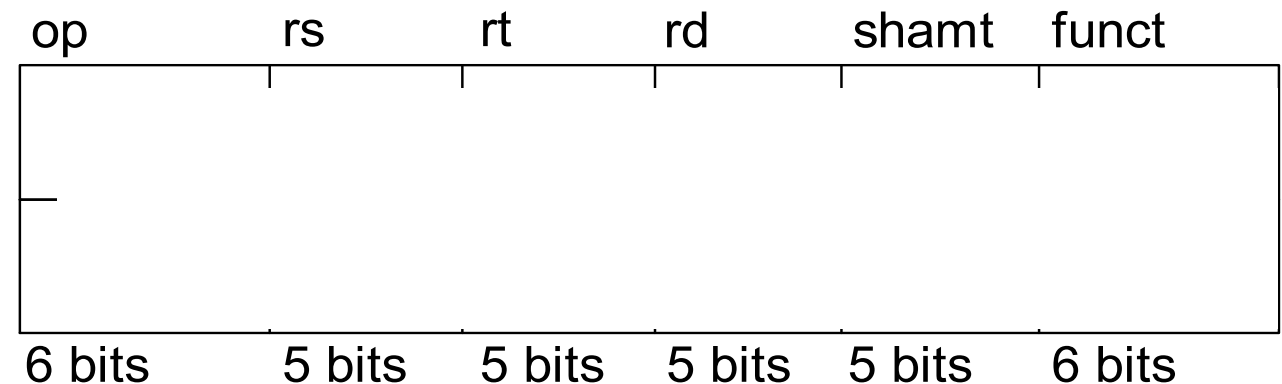
`add rd, rs, rt`

# R-Type Examples

## Assembly Code

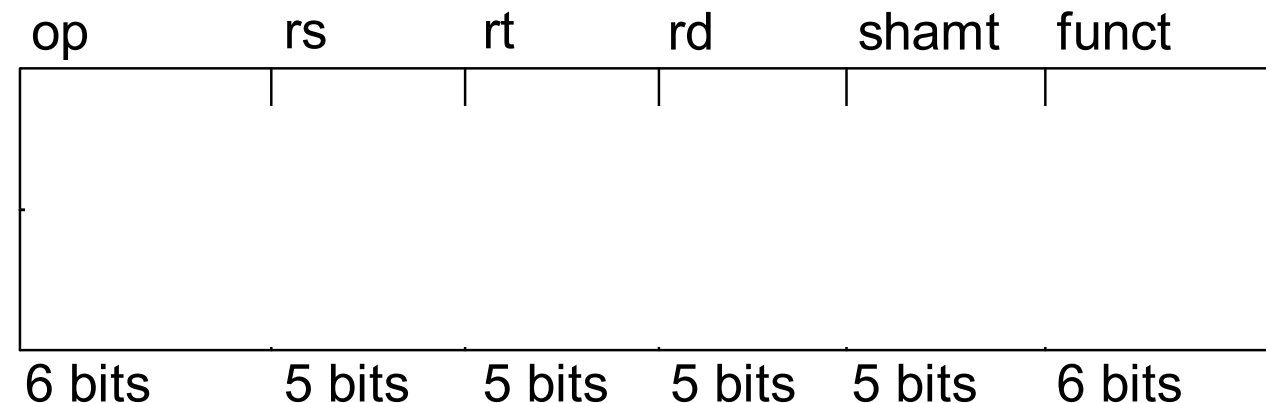
```
add $s0, $s1, $s2
sub $t0, $t3, $t5
```

## Field Values



**add** has op code of 0 and funct code of 32

## Machine Code



**Note** the order of registers in the assembly code:

```
add rd, rs, rt
```



# R-Type Examples

## Assembly Code

## Field Values

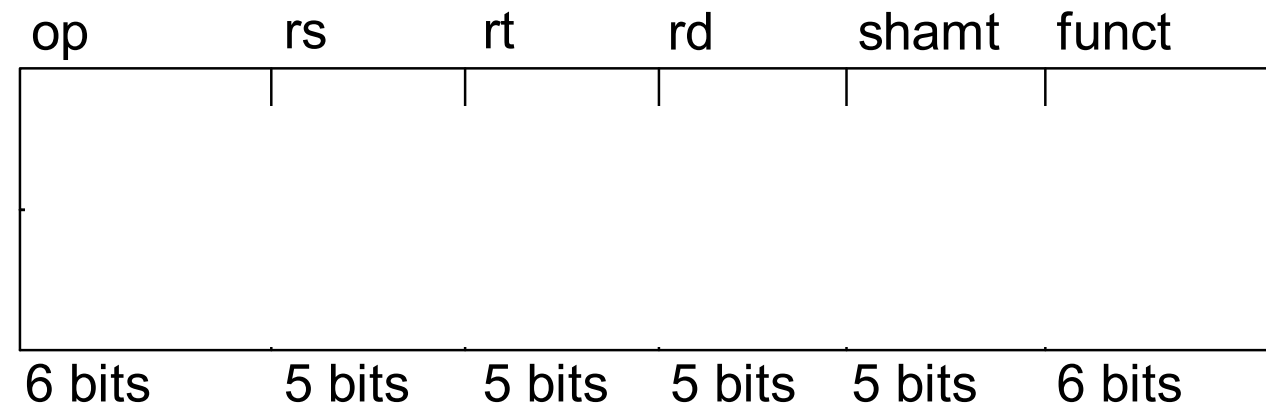
add \$s0, \$s1, \$s2

sub \$t0, \$t3, \$t5

op	rs	rt	rd	shamt	funct
0	17	18	16	0	32
0	11	13	8	0	34
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

**add** has op code of 0 and funct code of 32

## Machine Code



**Note** the order of registers in the assembly code:

add rd, rs, rt

# R-Type Examples

## Assembly Code

## Field Values

add \$s0, \$s1, \$s2

sub \$t0, \$t3, \$t5

op	rs	rt	rd	shamt	funct
0	17	18	16	0	32
0	11	13	8	0	34
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

**add** has op code of 0 and funct code of 32

## Machine Code

op	rs	rt	rd	shamt	funct	
000000	10001	10010	10000	00000	100000	(0x02328020)
000000	01011	01101	01000	00000	100010	(0x016D4022)
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

**Note** the order of registers in the assembly code:

add rd, rs, rt

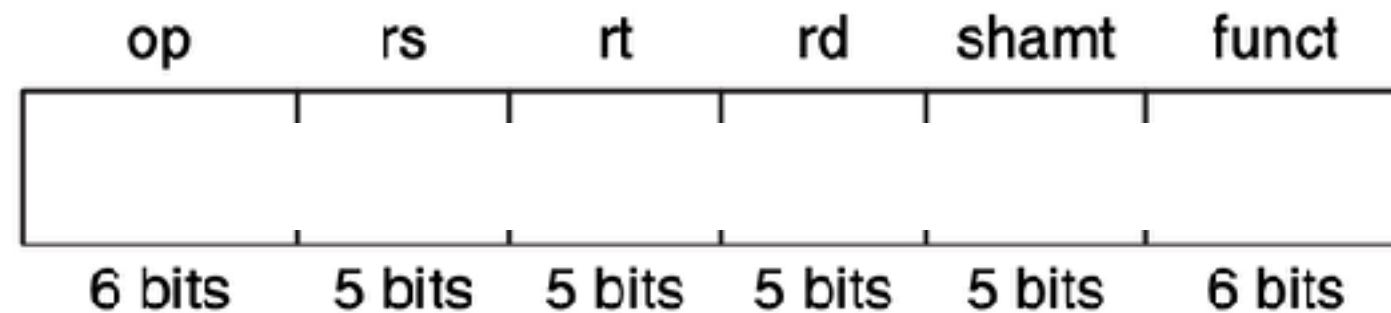
# Practice of R-type

add \$t4, \$s4, \$s5

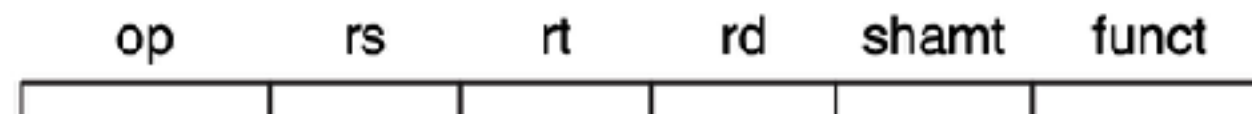
**add** has op code of 0 and  
funct code of 32

Name	Number
\$0	0
\$at	1
\$v0-\$v1	2-3
\$a0-\$a3	4-7
\$t0-\$t7	8-15
\$s0-\$s7	16-23
\$t8-\$t9	24-25
\$k0-\$k1	26-27
\$gp	28
\$sp	29
\$fp	30
\$ra	31

## Field Values



## Machine Code



# Practice of R-type

add \$t4, \$s4, \$s5

**add** has op code of 0 and  
funct code of 32

Name	Number
\$0	0
\$at	1
\$v0-\$v1	2-3
\$a0-\$a3	4-7
\$t0-\$t7	8-15
\$s0-\$s7	16-23
\$t8-\$t9	24-25
\$k0-\$k1	26-27
\$gp	28
\$sp	29
\$fp	30
\$ra	31

## Field Values

op	rs	rt	rd	shamt	funct
0	20	21	8	0	32
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

## Machine Code

op	rs	rt	rd	shamt	funct

# Practice of R-type

add \$t4, \$s4, \$s5

**add** has op code of 0 and  
funct code of 32

Name	Number
\$0	0
\$at	1
\$v0-\$v1	2-3
\$a0-\$a3	4-7
\$t0-\$t7	8-15
\$s0-\$s7	16-23
\$t8-\$t9	24-25
\$k0-\$k1	26-27
\$gp	28
\$sp	29
\$fp	30
\$ra	31

## Field Values

op	rs	rt	rd	shamt	funct
0	20	21	8	0	32
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

## Machine Code

op	rs	rt	rd	shamt	funct		
000000	10100	10101	01000	00000	100000	(0x02954020)	
0	2	9	5	4	0	2	0