



中山大學
SUN YAT-SEN UNIVERSITY



计算机组成原理

第二章：指令：计算机的语言

中山大学计算机学院
陈刚

2022年秋季

回顾——2.4.2 数值数据的定点表示

要解决的问题

1、第一个问题：正数与负数的表示？ Positive and negative representations

所有数前面**设置符号位** Set symbol bit in front of all numbers

2、第二个问题：小数点的表示？ Representation of decimal point

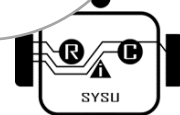
小数点的位置固定 Fixed decimal point

3、第三个问题：零的表示？ Representation of zero

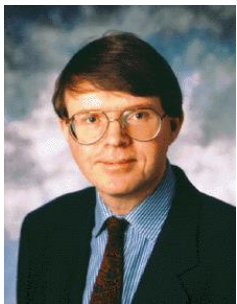
二进制补码表示 two' s complement representation

4、第四个问题：实数的表示？ Representation of real number

将实数分成两部分：尾数和指数(阶码)——浮点数的表示 Float point representation



回顾—2.4.3 数值数据的浮点表示—IEEE 754



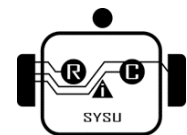
1985年制定了浮点数标准IEEE 754 (ch3.5.1)

符号s(Sign) 1bit	阶码e(整数)Exponent 8bits	尾数f(小数)Significand 23bits
-------------------	--------------------------	------------------------------

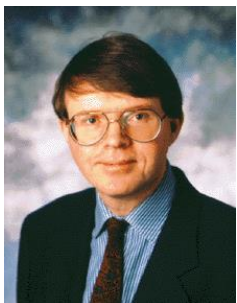
Prof. William Kahan

符号s:

□ 1 表示负数negative ; 0表示 正数positive



回顾—2.4.3 数值数据的浮点表示—IEEE 754



Prof. Kahan

1985年制定了浮点数标准IEEE 754 (ch3.5.1)

符号s(Sign) 1bit	阶码e(整数)Exponent 8bits	尾数f(小数)Significand 23bits
-------------------	--------------------------	------------------------------

➤ 尾数f

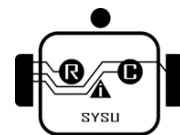
尾数 = $1 + \text{significand}$

$0 < \text{significand} < 1$

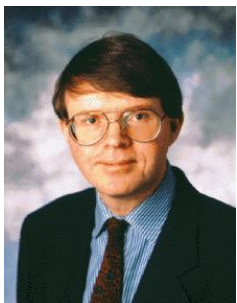
- 尾数为原码
- 规格化尾数最高位总是1，所以隐含表示，省1位
- 1 + 23 bits (single单精度), 1 + 52 bits (double双精度)

尾数精度 = 尾数的位数 + 1

(单精度SP: Single Precision 双精度DP: Double Precision)



回顾—2.4.3 数值数据的浮点表示—IEEE 754



Prof. Kahan

1985年制定了浮点数标准IEEE 754 (ch3.5.1)

符号s(Sign) 1bit	阶码e(整数)Exponent 8bits	尾数f(小数)Significand 23bits
-------------------	--------------------------	------------------------------

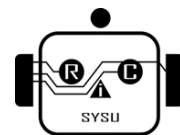
➤ 阶码/指数e: 移码

将每一个数值加上一个偏置常数, 当编码位数为 n 时, 通常bias取 $2^{n-1}-1$, but IEEE 754的阶码不同

- 偏置常数为: 127 (单精度SP); 1023 (双精度DP)
- 单精度规格化数阶码范围为0000 0001 (-126) ~ 11111110 (127)

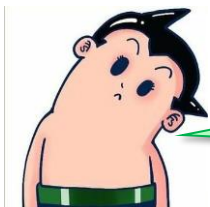
全0/全1编码用来表示特殊的值!

单精度SP: Single Precision 双精度DP: Double Precision



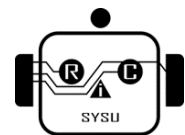
回顾—2.4.3 数值数据的浮点表示—IEEE 754

2.4.3 数值数据的浮点表示—IEEE 754



全0和全1编码用来表示什么特殊的值？

阶码(移码)	尾数	数据类型
1~254	任何值	规格化数（隐含小数点前为“1”）
0	0	?
0	非零的数	?
255	0	?
255	非零的数	?



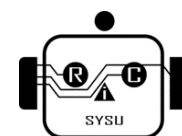
回顾—2.4.3 数值数据的浮点表示—IEEE 754

□ 如何表示0?

- 阶码/指数: 全0 (Exponent: all zeros)
- 尾数: 全0 (Significand: all zeros)
- 符号位? 正/负皆可 (What about sign? Both cases valid.)

+0: 0 00000000 000000000000000000000000

-0: 1 00000000 000000000000000000000000



回顾—2.4.3 数值数据的浮点表示—IEEE 754

□ 如何表示0?

■ 阶码/指数: 全0

■ 尾数: 全0

■ 符号位? 正/负皆可

+0: 0 00000000 00000000000000000000000000000000

-0: 1 00000000 00000000000000000000000000000000

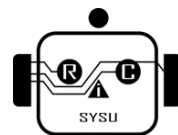
□ 如何表示 $+\infty/-\infty$?

■ 阶码/指数: 全1 ($11111111_2 = 255$) (Exponent : all ones)

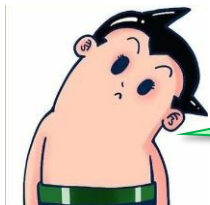
■ 尾数: 全0 (Significand: all zeros)

$+\infty$: 0 11111111 00000000000000000000000000000000

$-\infty$: 1 11111111 00000000000000000000000000000000



回顾—2.4.3 数值数据的浮点表示—IEEE 754



全0和全1编码用来表示什么特殊的值？

阶码(移码)	尾数	数据类型
1~254	任何值 (隐含小数点前为“1”)	规格化数
0	0	0
0	非零的数	非规格化数 Denormalized numbers
255	0	$+\infty/-\infty$
255	非零的数	非数 NaN(Not a Number)

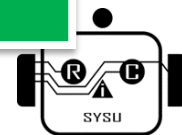
非数, 示例：

$\text{sqrt}(-4.0) = \text{NaN}$

$0/0 = \text{NaN}$

$+\infty + (-\infty) = \text{NaN}$

$+\infty - (+\infty) = \text{NaN}$



回顾—2.4.3 数值数据的浮点表示—IEEE 754

□ 如何表示规格化非0数 (Norms)

□ **Exponent**: nonzero

□ **Significand**: nonzero (大于等于1且小于2)

□ 如何表示非规格化数 (Denorms)

□ **Exponent**: all zeros

□ **Significand**: nonzero (大于0且小于1)

□ 如何表示非数 (NaN)

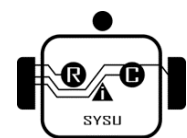
□ **Exponent** : 255

□ **Significand**: nonzero

□ **NaNs can help with debugging**

$$N = -1^s \times 1.\text{fraction} \times 2^{\text{exponent}-127}, \quad 1 \leq \text{exponent} \leq 254$$

$$N = -1^s \times 0.\text{fraction} \times 2^{-126}, \quad \text{exponent} = 0$$

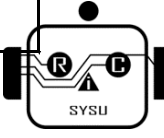


回顾—2.4.3 数值数据的浮点表示—IEEE 754

IEEE754 标准定义的浮点格式参数

***尾数精度 = 尾数位数 + 1**

参数	单精度	扩充单精度	双精度	扩充双精度
总位数	32	≥43	64	≥79
阶码位数	8	≥11	11	≥15
阶码编码	+127 移码	(未定义)	+1023 移码	(未定义)
最大阶码值	+127	+1023	+1023	+16383
最小阶码值	-126	-1022	-1022	-10382
可表示的阶码个数	254	(未定义)	2046	(未定义)
尾数位数*	23	≥31	52	≥63
可表示的尾数个数	2^{23}	(未定义)	2^{52}	(未定义)
可表示的数据总数	1.98×2^{31}	(未定义)	1.99×2^{63}	(未定义)
数值表示范围 (十进制)	$10^{-38}, 10^{+38}$	(未定义)	$10^{-308}, 10^{+308}$	(未定义)



回顾—关于IEEE 754的一些问题

单精度可表示值的范围是多少？

The largest number for single: $+1.11\dots1 \times 2^{127}$ 约 $+3.4 \times 10^{38}$

The least number for single: $+1.0\dots0 \times 2^{-126}$

How about double? $+1.11\dots1 \times 2^{1023}$ 约 $+1.8 \times 10^{308}$

强制类型转换会如何呢 有效位数可能丢失 ($2^{24}+1$)

`i = int (float) i`

How about double?

True!

`f = float (int) f`

How about double?

Not always true!

浮点加法满足结合律吗?

小数部分可能丢失

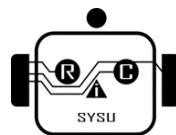
$$x = -1.5 \times 10^{38}, \quad y = 1.5 \times 10^{38}, \quad z = 1.0$$

$$(x + y) + z = (-1.5 \times 10^{38} + 1.5 \times 10^{38}) + 1.0 = 1.0$$

$$x + (y + z) = -1.5 \times 10^{38} + (1.5 \times 10^{38} + 1.0) = 0.0$$

结论: $(x + y) + z \neq x + (y + z)$

(ch3.9)



本讲内容

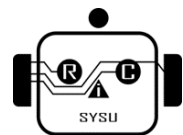
□ MIPS指令集

□ 基本指令和指令类型

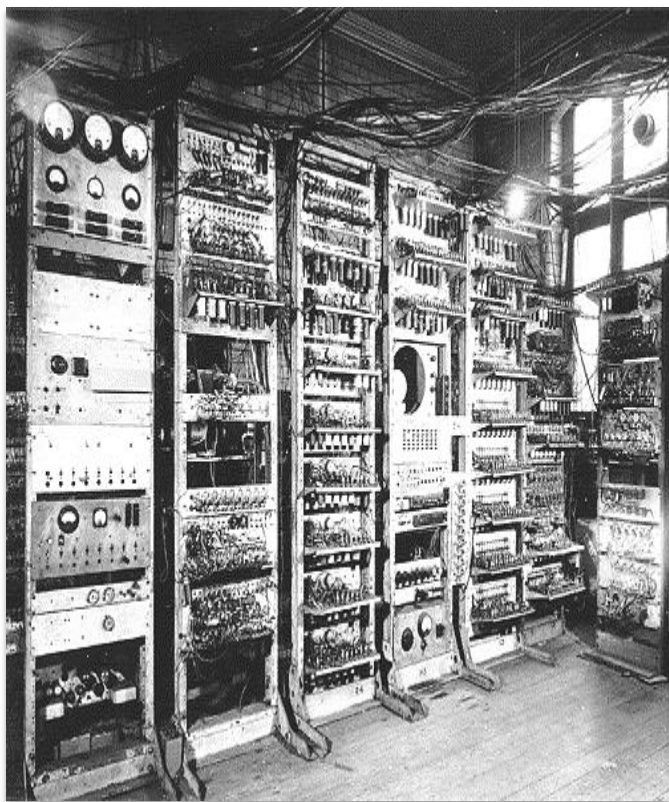
□ 程序的机器级表示

□ MIPS指令系统介绍

□ MIPS指令系统举例



2.4.8 基本指令和指令类型



1944年哈佛大学, **Mark I**
——世界最早的继电器通用计算机之一

- 存储器大小: 32字 (可扩充到8K字)
- 机器字长: 32位
- 提供6条指令:

jump

load accumulator

subtract

store accumulator

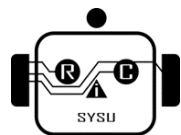
test for zero

stop

2.4.8 基本指令和指令类型

一个较完善的指令系统应包括

数据传送指令	Move、Store、Load、Exchange 等
算术运算指令	定点数、浮点数运算和十进制数运算
逻辑运算指令	And、Or、Not、Xor、Compare等
输入输出指令	IN, OUT
系统控制指令	启动IO设备指令、存取特殊寄存器指令等
程序控制指令	转移指令、循环控制指令(LOOP)、子程序调用与返回指令(CALL, RET)、程序中断指令及返回(INT, IRET)



2.4.8 基本指令和指令类型

数据传送指令



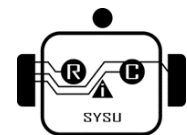
功能

- 实现寄存器与寄存器、寄存器与存储单元以及存储单元之间的数据传送



表征参数

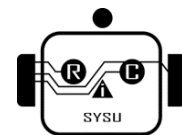
- 源操作数地址，可由指令给出
- 目的操作数地址，可由指令给出
- 传送数据的长度，可由指令**隐含**给出



2.4.8 基本指令和指令类型

IBM S/370数据传送指令举例

助记符	操作名称	传送字节数	传 送 说 明
L	取数	4	存储器传送到寄存器
LH	取半字	2	存储器传送到寄存器
LR	取数	4	寄存器传送到寄存器
LER	取数(短)	4	浮点寄存器传送到浮点寄存器
LE	取数(短)	4	存储器传送到浮点寄存器
LDR	取数(长)	8	浮点寄存器传送到浮点寄存器
LD	取数(长)	8	存储器传送到浮点寄存器
ST	存数	4	寄存器传送到存储器
STH	存半字	2	寄存器传送到存储器
STC	存字节	1	寄存器传送到存储器
STE	存数(短)	4	浮点寄存器传送到存储器
STD	存数(长)	8	浮点寄存器传送到存储器



2.4.8 基本指令和指令类型

算术逻辑运算指令



功能

- 提供二进制定点数的加、减、乘、除等基本运算指令



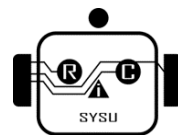
算术运算指令操作过程

- 将源操作数传送到算术逻辑部件ALU输入端
- 进行运算
- 将ALU的运算结果传送到目的操作数地址单元



逻辑运算指令

- 按位进行的运算，可以针对任何一种可寻址单元中的位进行操作
- 运算基础为布尔代数，包括与、或、非、异或等逻辑操作、非运算的位操作（如位测试、位清除等）以及移位操作



2.4.8 基本指令和指令类型

程序控制指令



功能

CPU执行程序时，通常每执行一条指令后，程序计数器PC会自动加 Δ ，要打破这种顺序，需要程序控制指令

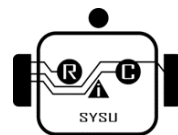
什么时候需要
变更顺序？

遇到循环、分支、子程序



程序控制指令类型

- 转移指令
- 子程序调用和返回指令



2.4.8 基本指令和指令类型

程序控制指令之转移指令



功能

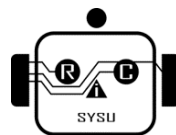
改变CPU执行指令顺序，即让CPU到新的地址去执行后续指令

分类

- 无条件转移：直接将控制**转移**到指令中所指定的目标，从那里开始执行。CPU在执行无条件转移指令时，只需将转移目标地址送入程序计数器PC就可完成
- 条件转移：**条件满足时转移**到指令中所指定的目标，**否则顺序**执行下一条指令

实现条件转移的方法：

条件码/条件寄存器/比较与转移指令等



2.4.8 基本指令和指令类型

程序控制指令之子程序调用和返回指令

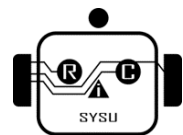
什么是子程序

子程序是将那些经常使用的、能够独立完成某一特定功能的程序段，独立出来为其他语句调用的整体

使用子程序的优势

- 降低程序设计的复杂度
- 提高程序的可重用性
- 节省存储空间
- 是模块化程序设计的基础，**子程序库**

子程序的使用称为**子程序调用**
调用子程序的程序称之为**调用程序或主程序**



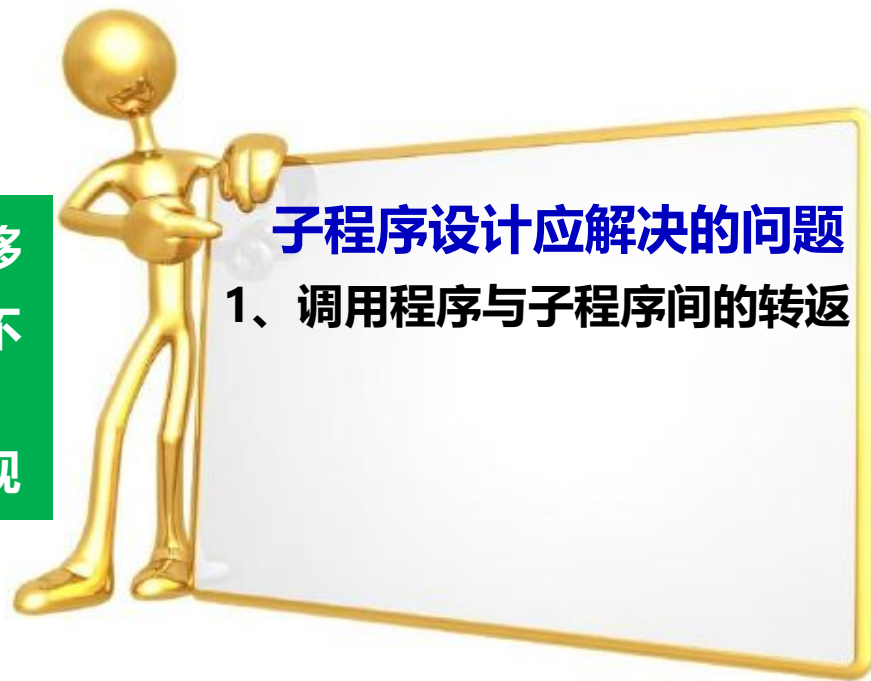
2.4.8 基本指令和指令类型

程序控制指令之子程序调用和返回指令

- 子程序调用和返回，实质上就是程序控制的转移
- 主程序是主动的、预知的；子程序是被动的，不能预知主程序什么时刻、在什么位置调用它
- 通过设置专门调用(CALL)和返回(RET)指令实现

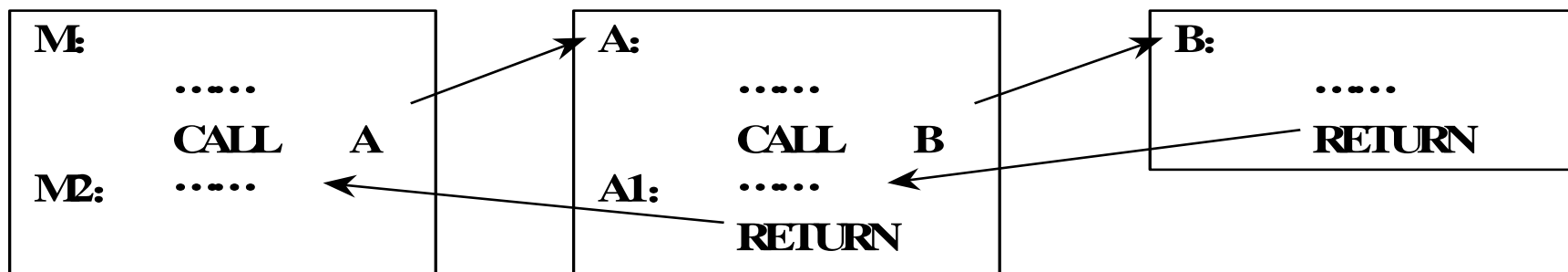
子程序设计应解决的问题

1、调用程序与子程序间的转返

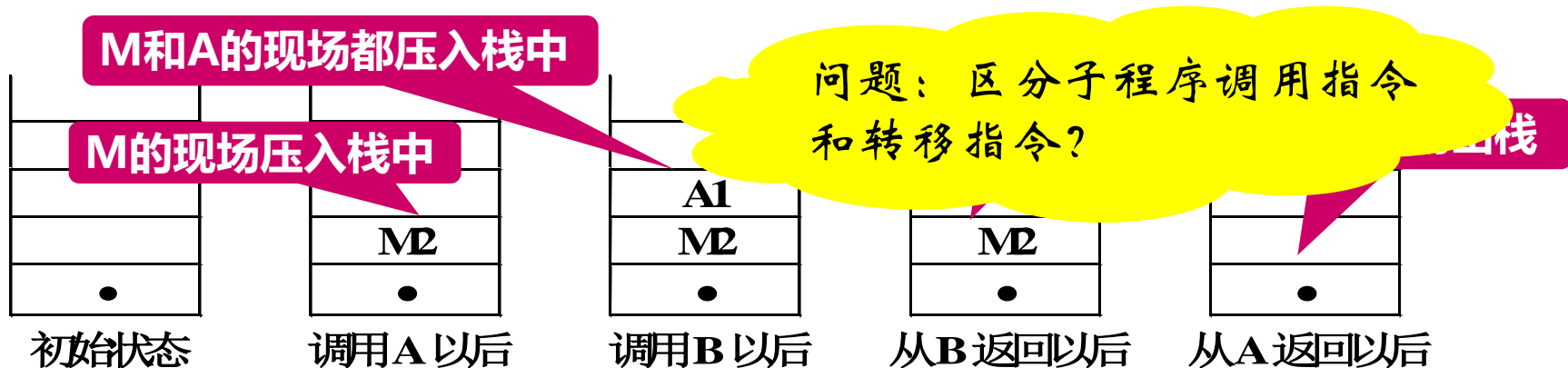


示例：调用程序和子程序的转返

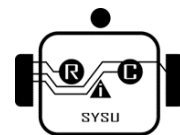
例：若有一个主程序M和两个子程序A、B，它们的调用关系是M调用A，A又调用B。



M、A和B的调用关系



使用堆栈实现M、A和B中的子程序嵌套

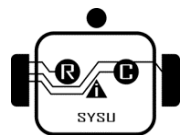


2.4.8 基本指令和指令类型

程序控制指令之子程序调用和返回指令

子程序调用和转移指令均可改变程序的执行顺序，它们有什么不同呢？

- 子程序要求返回，可嵌套和递归调用；转移指令不求返回，通常只在同一程序段中出现
- 子程序用于实现程序与程序之间转移；转移指令用于实现同一程序内转移



2.4.8 基本指令和指令类型

程序控制指令之子程序调用和返回指令

CALL X;

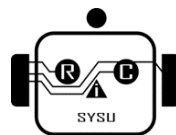
X为子程序的地址， Δ 为本指令的长度

■ 采用寄存器存放返回地址

□ $RA \leftarrow PC + \Delta$

□ $PC \leftarrow X$

不支持子程序
的再入！



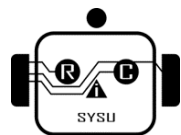
2.4.8 基本指令和指令类型

程序控制指令之子程序调用和返回指令



程序再入的形式：嵌套和递归

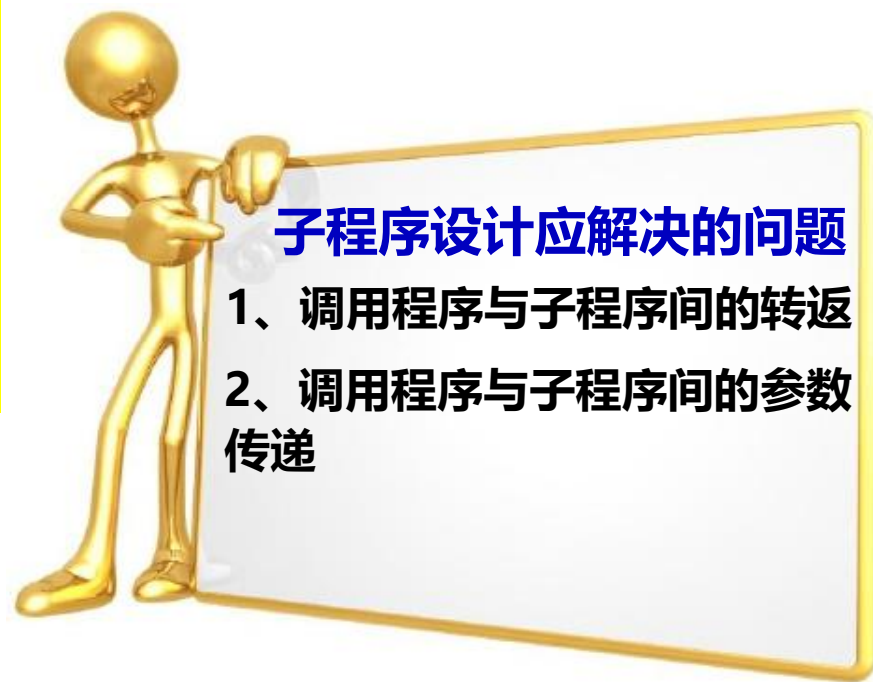
- 采用寄存器存放返回地址
 - ✓ 单寄存器不支持嵌套和递归；多寄存器支持嵌套，不支持递归
- 放在子程序的起始位置
 - ✓ 支持嵌套，不支持递归
- 用堆栈来保存返回地址
 - ✓ 由于堆栈具有后进先出的性质，因而用堆栈保存返回地址可实现子程序嵌套和递归调用



2.4.8 基本指令和指令类型

程序控制指令之子程序调用和返回指令

- 调用程序提供给子程序以便加工处理的信息称入口参数；经子程序加工处理后回送给调用程序的信息称出口参数
- 参数传递方法包括约定寄存器法、约定存储单元法、参数赋值法、堆栈法



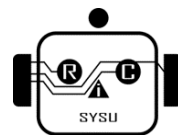
2.4.8 基本指令和指令类型

程序控制指令之子程序调用和返回指令

- 在子程序的开始处保护现场，返回前恢复现场
- 保护和恢复现场的原则：破坏什么现场就保护什么现场，保护什么现场就恢复什么现场
- 用PUSH指令构成保护程序段，POP指令构成恢复程序段，注意进出栈的顺序是相反的

子程序设计应解决的问题

- 1、调用程序与子程序间的转返
- 2、调用程序与子程序间的参数传递
- 3、调用程序和子程序公用寄存器的问题

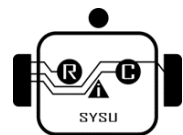


2.4.8 基本指令和指令类型

程序控制指令之子程序调用和返回指令

子程序的结构（编译后机器代码的视角）

- 保存子程序运行时将被破坏的寄存器的内容，即保存现场
- 依入口参数从指定位置取要加工处理的信息
- 加工处理
- 依出口参数向指定位置送经加工处理后的结果信息
- 将进入子程序时保存的寄存器的内容送回寄存器，即恢复现场
- 返回调用程序



2.4.8 基本指令和指令类型

输入输出指令

I/O设备独立编址

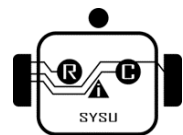
➤ **专用的I/O指令**：通常包含两个操作数（数据的地址，外部设备的地址）

如Intel公司X86的IN、OUT指令

I/O设备与内存统一编址

➤ **通用的数据传送指令**：称为存储器映像的I/O操作。要求外部设备的寄存器与主存单元统一编址

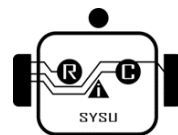
➤ **通过I/O处理机执行I/O操作**：对I/O系统的管理和控制都是由通道和IOP完成



2.4.8 基本指令和指令类型

系统控制指令

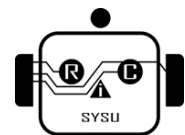
- 系统控制指令是指那些通常只能由操作系统执行，而不直接提供给一般用户使用的特权指令，处理机只有在处于特权状态时才能执行这些指令
- 系统控制指令主要用于实现对控制寄存器进行操作，检测或修改访问权限，改变系统的工作方式，访问进程控制块等。在多用户、多任务环境下使用较多
- 一般用户需要计算机系统的特权服务时，通过系统控制指令向操作系统发出请求，由操作系统处理，并将处理的结果返回给用户
 - 用户态与内核态的转换



小结

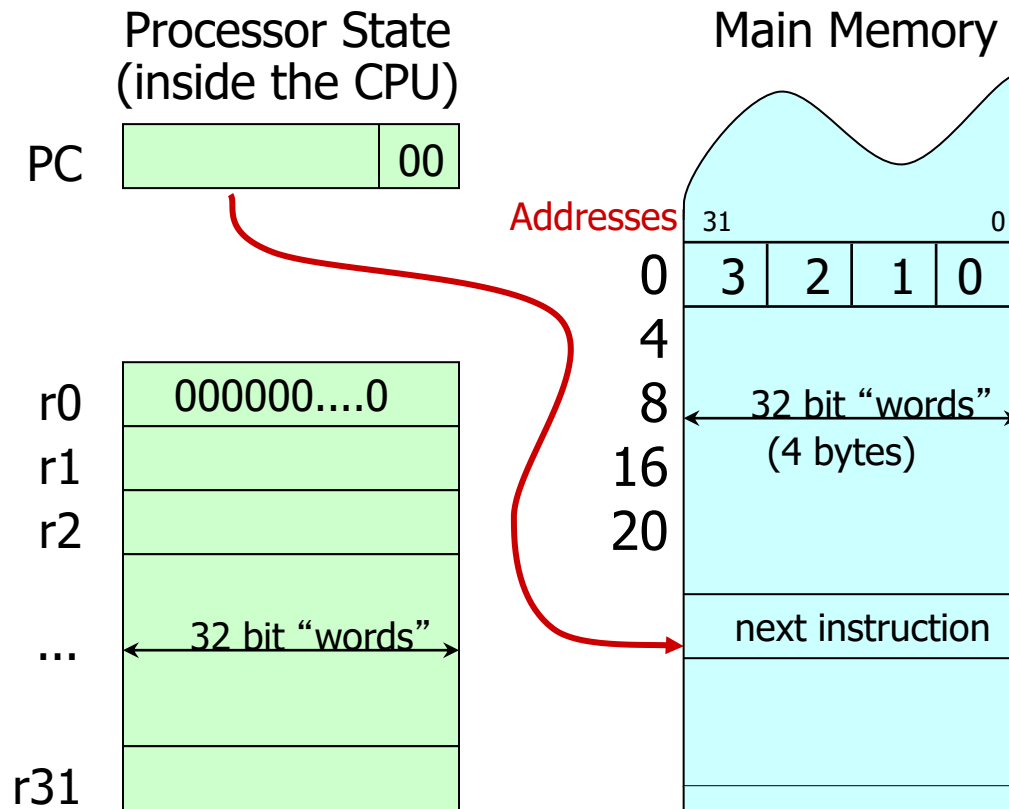
□基本指令和指令类型

- 数据传送指令
- 算术逻辑运算指令
- 程序控制指令
- 输入输出指令
- 系统控制指令



MIPS 编程模型

a representative simple RISC machine



General Registers:

A small scratchpad of frequently used or temporary variables

we'll use a clean and sufficient subset of the MIPS-32 core Instruction set.

Fetch/Execute loop:

- fetch $\text{Mem}[\text{PC}]$
- $\text{PC} = \text{PC} + 4^\dagger$
- execute fetched instruction (may change PC!)
- repeat!

*† MIPS uses byte memory addresses. However, each instruction is 32-bits wide, and *must* be aligned on a multiple of 4 (word) address. Each word contains four 8-bit bytes. Addresses of consecutive instructions (words) differ by 4.*

MIPS 寄存器堆 (Register File)

- ❑ Operands of arithmetic instructions must be from a limited number of special locations contained in the datapath's register file

- ❑ Thirty-two 32-bit registers

- ❑ Two read ports

- ❑ One write port

- ❑ Registers are

- Fast

- Smaller is faster & Make the common case fast

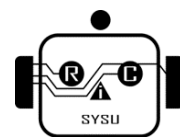
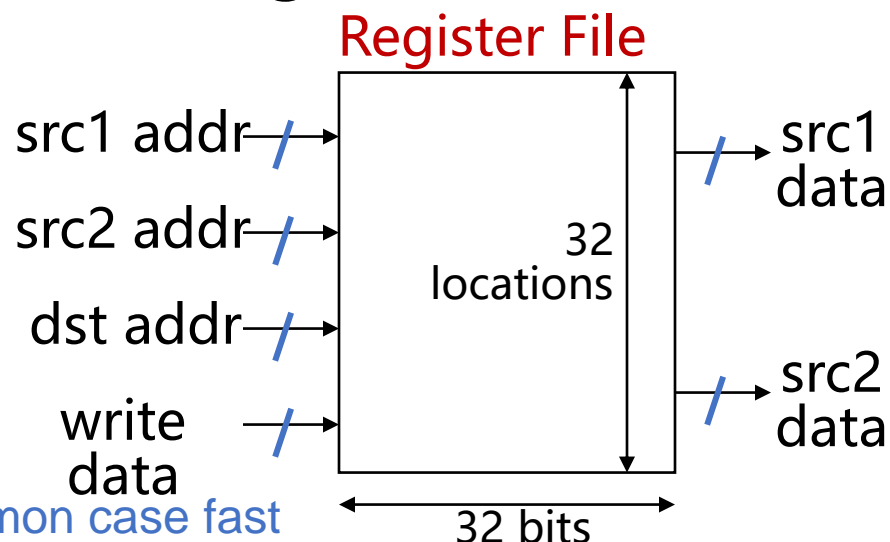
- Easy for a compiler to use

- e.g., $(A*B) - (C*D) - (E*F)$ can do multiplies in any order

- Improves code density

- Since register are named with fewer bits than a memory location

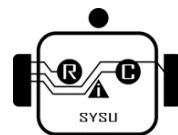
- ❑ Register addresses are indicated by using \$



MIPS体系结构中的通用寄存器

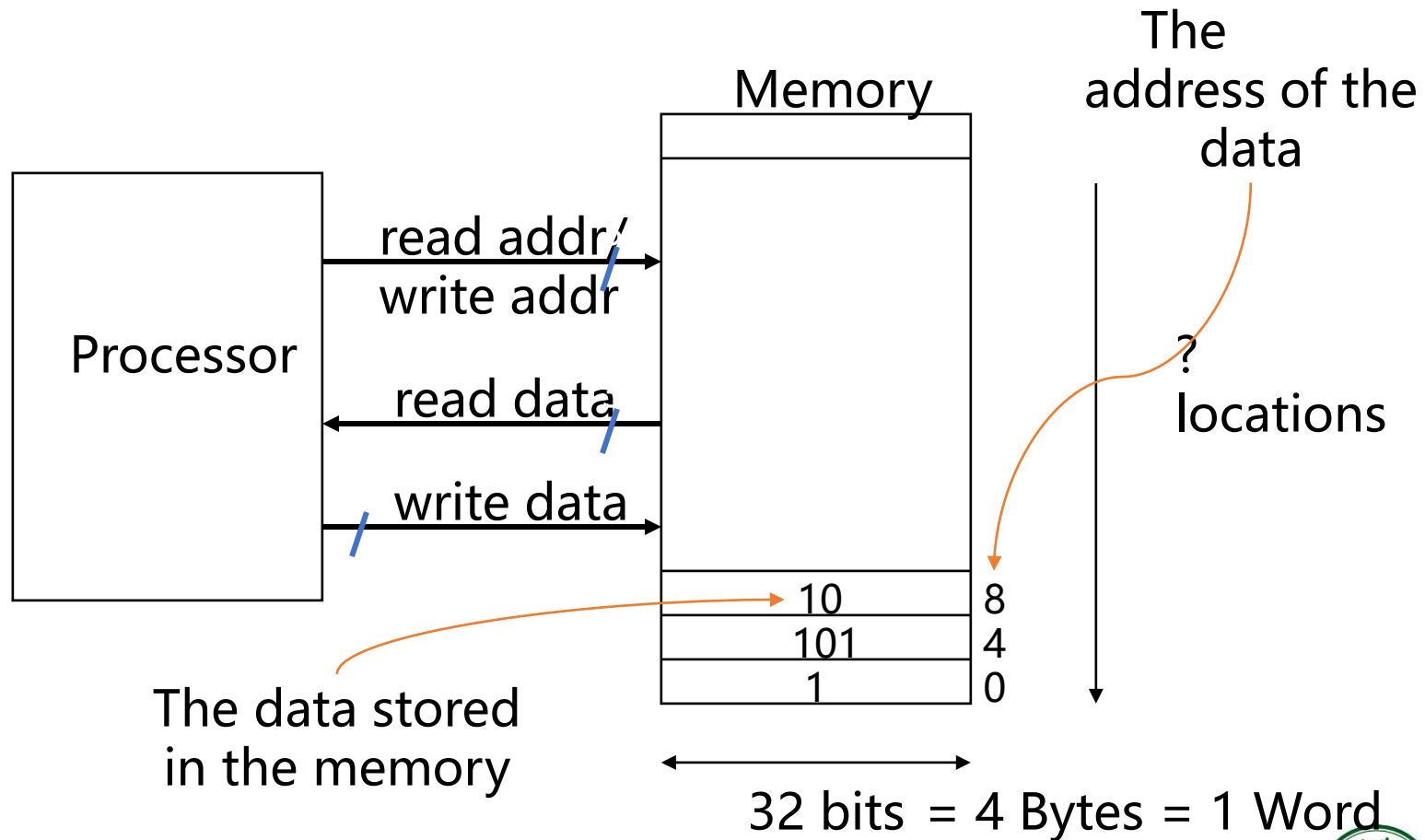
表 1-1 MIPS32 中通用寄存器的约定用法

寄存器名字	约定命名	用途
\$0	zero	总是为 0
\$1	at	留作汇编器生成一些合成指令
\$2、\$3	v0、v1	用来存放子程序返回值
\$4~\$7	a0~a3	调用子程序时，使用这 4 个寄存器传输前 4 个非浮点参数
\$8~\$15	t0~t7	临时寄存器，子程序使用时可以不用存储和恢复
\$16~\$23	s0~s7	子程序寄存器变量，改变这些寄存器值的子程序必须存储旧的值并在退出前恢复，对调用程序来说值不变
\$24、\$25	t8、t9	临时寄存器，子程序使用时可以不用存储和恢复
\$26、\$27	\$k0、\$k1	由异常处理程序使用
\$28 或 \$gp	gp	全局指针
\$29 或 \$sp	sp	堆栈指针
\$30 或 \$fp	s8/fp	子程序可以用来做堆栈帧指针
\$31	ra	存放子程序返回地址



处理器-存储器互联

- Memory is a large, single-dimensional array
- An address acts as the index into memory array



Some MIPS Memory Nits

- Memory locations are 32 bits wide

- BUT, they are addressable in different-sized chunks

- 8-bit chunks (bytes)

- 16-bit chunks (shorts)

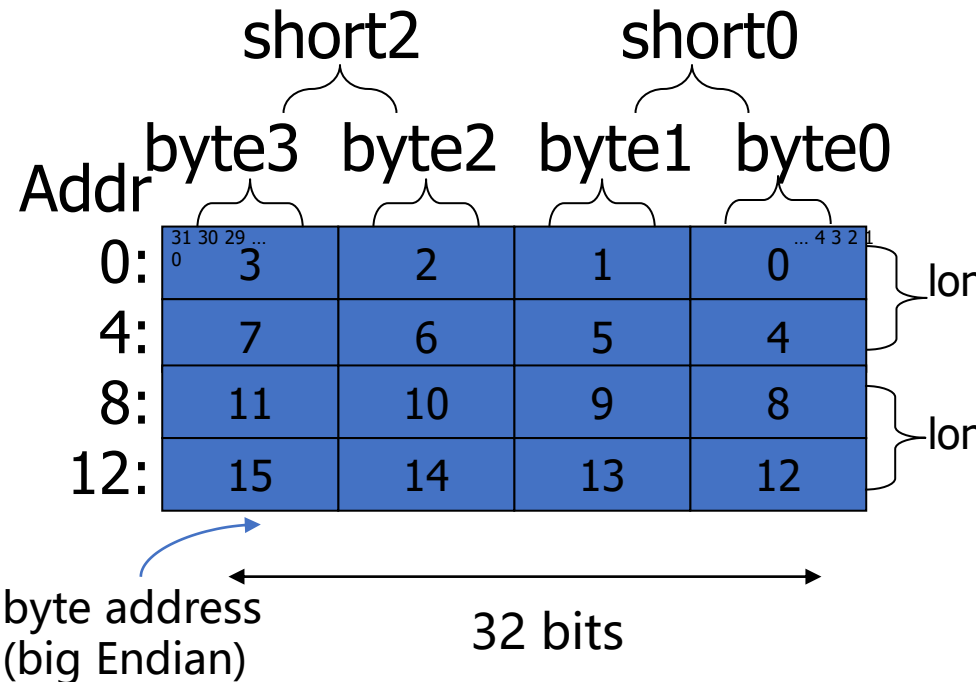
- 32-bit chunks (words)

- 64-bit chunks (longs/double)

- We also frequently need access to individual bits! (Instructions help w/ this)

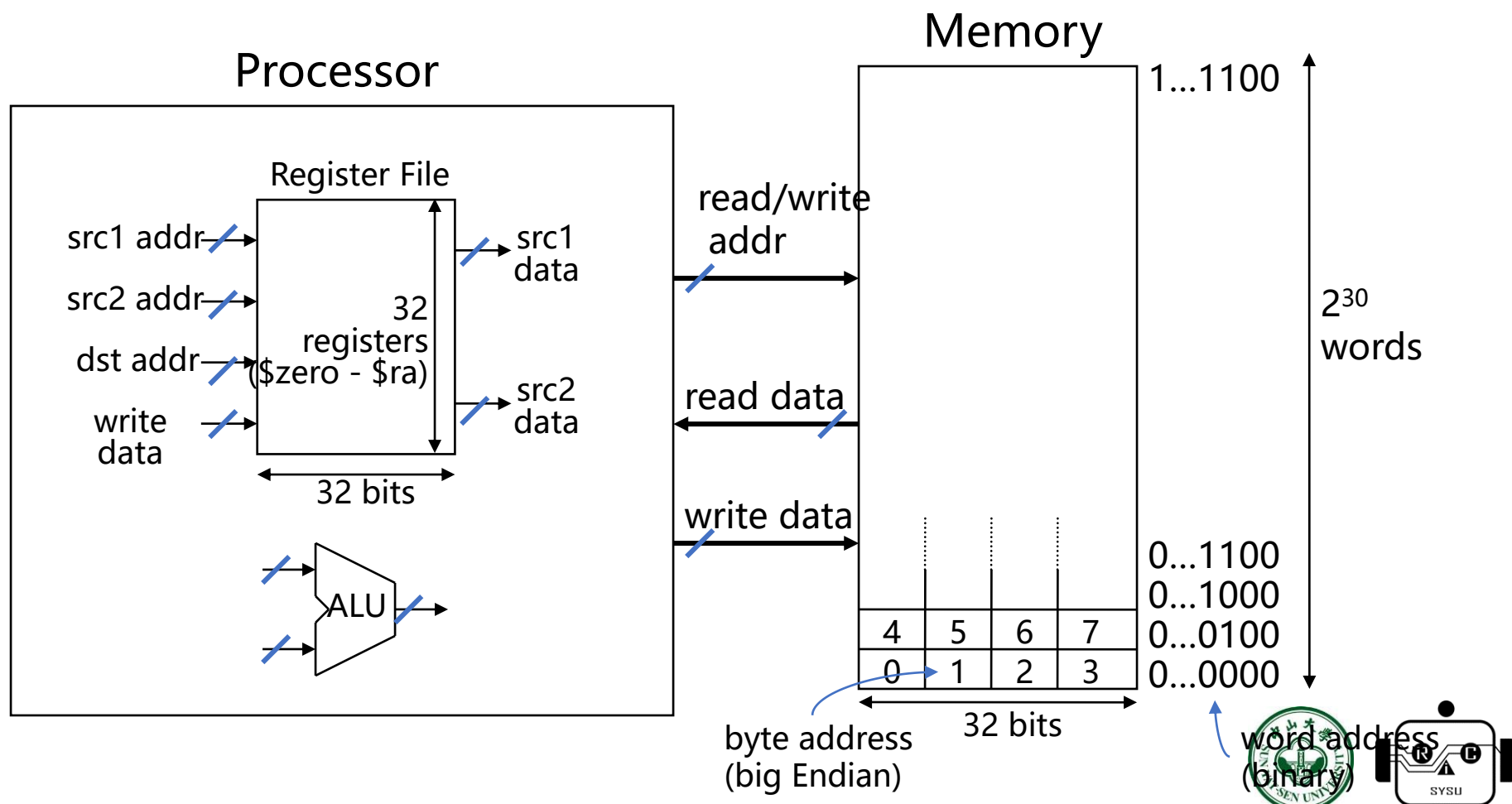
- Every BYTE has a unique address (MIPS is a byte-addressable machine)

- Every instruction is one word



MIPS结构组成

- ❑ Arithmetic instructions – to/from the register file
- ❑ Load/store instructions - to/from memory



MIPS Register Nits

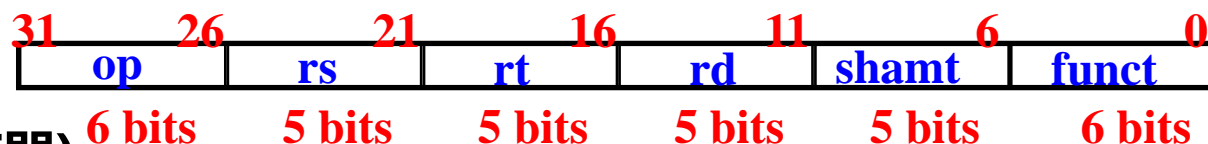
- ❑ There are 32 named registers [\$0, \$1, ..., \$31]
- ❑ The operands of a// ALU instructions are registers
 - ❑ This means to operate on a variables in memory you must:
 - ❑ Load the value/values from memory into a register
 - ❑ Perform the instruction
 - ❑ Store the result back into memory
- ❑ Going to and from memory can be expensive
 - ❑ (4x to 20x slower than operating on a register)
- ❑ Net effect: Keep variables in registers as much as possible!
- ❑ 2 registers have H/W specific “side-effects”
 - ❑ (ex: \$0 always contains the value ‘0’ ... more later)
- ❑ 4 registers dedicated to specific tasks by convention
- ❑ 26 are available for general use
- ❑ Further conventions delegate tasks to other registers



1. MIPS指令格式

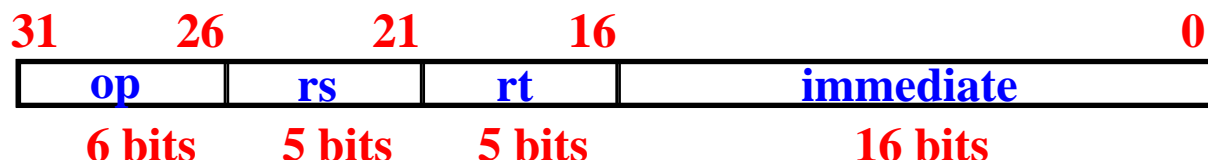
□ 所有指令都是32位宽，按字地址对齐

□ 三种指令格式



■ R-Type(用于寄存器)

- 两个操作数都是寄存器的运算指令。如: sub rd, rs, rt



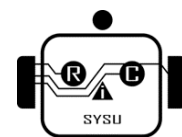
■ I-Type

- 运算指令：一个寄存器、一个立即数。如: ori rt, rs, imm16
- Load和Store指令。如: lw rt, rs, imm16
- 条件分支指令。如: beq rs, rt, imm16



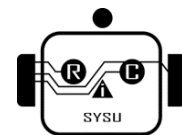
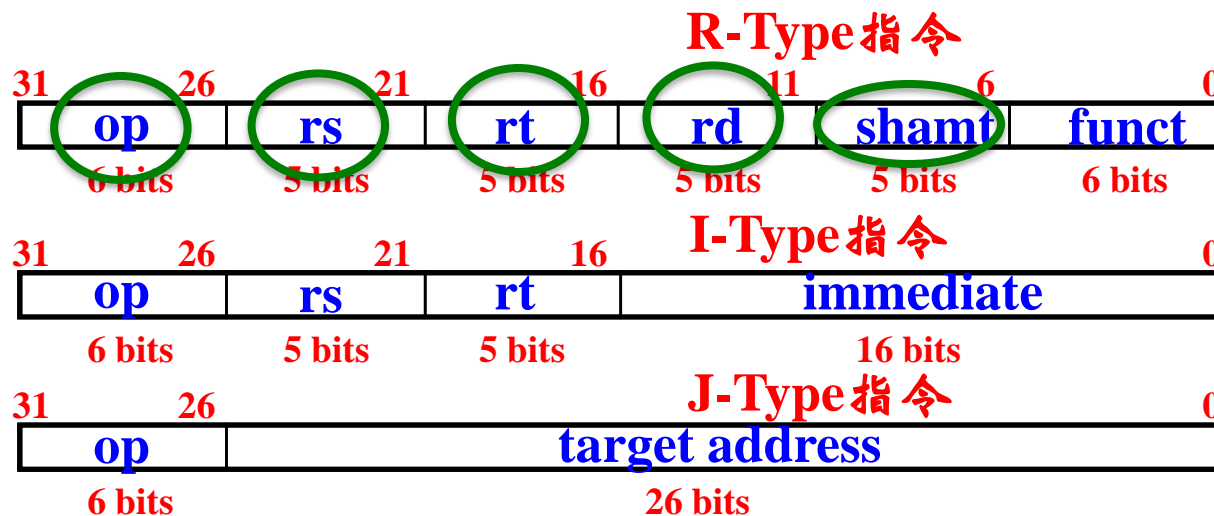
■ J-Type

- 无条件跳转指令。如: j target



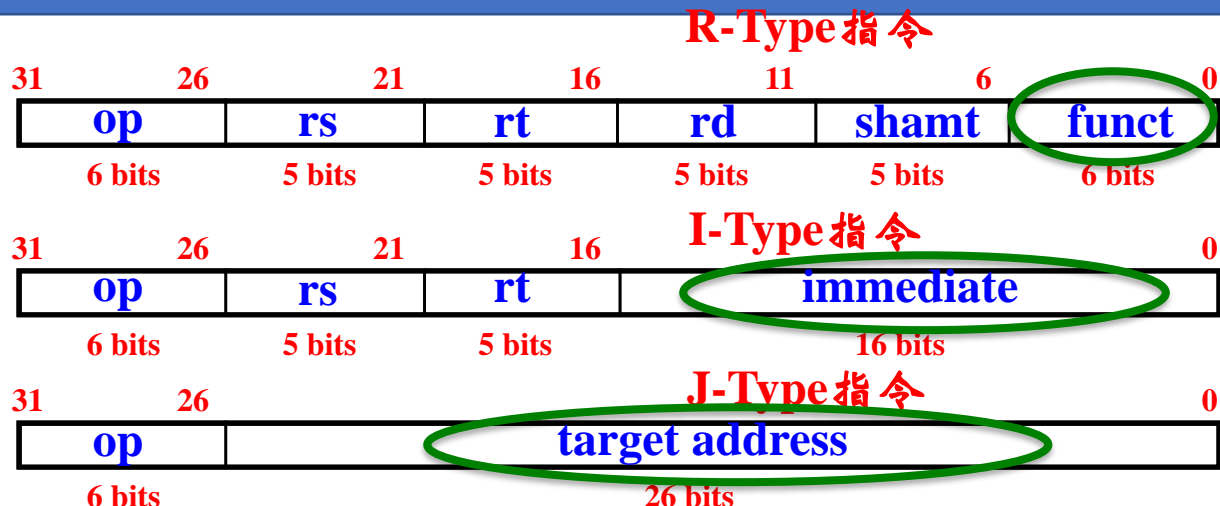
1. 指令格式——MIPS 指令字段含义

OP: 操作码
rs: 第一个源操作数寄存器
rt: 第二个源操作数寄存器
rd: 结果寄存器
shamt: 移位指令的位移量



1. 指令格式——MIPS 指令字段含义

OP: 操作码
rs: 第一个源操作数寄存器
rt: 第二个源操作数寄存器
rd: 结果寄存器
shamt: 移位指令的位移量



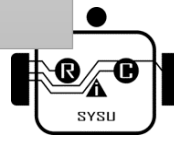
funct: R-Type指令的OP字段特定为“000000”，具体操作由funct字段给定。如：funct = “100000” 表示“加法”运算。

immediate: 立即数或load/store指令或分支指令的偏移地址

target address: 无条件转移地址的低26位。将PC高4位拼上26位直接地址，最后添2个“0” 就是32位目标地址。

为什么要添“0”？（相当于乘以4）

操作码的不同编码定义了不同的含义，若操作码相同时，再用不同编码的功能码来定义不同的含义！



2. 寻址方式

MIPS寻址方式

R-Format

寄存器寻址



Register

立即数寻址



I-Format

基址寻址



Register



Memory
Word

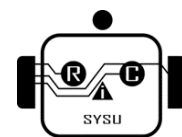
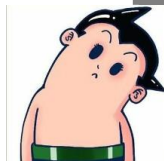
Byte Halfw

PC寻址



如何寻址?

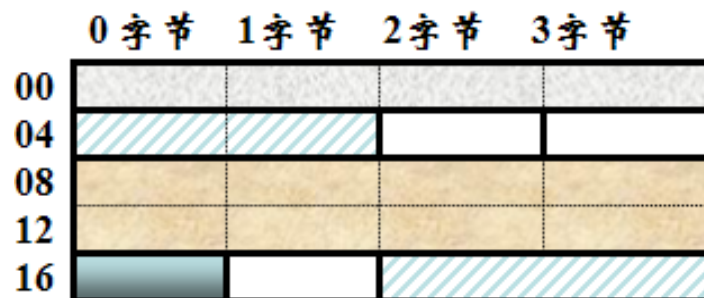
J-Format



2. 寻址方式

MIPS寻址方式

MIPS所有指令都是32位宽，按字地址对齐



边界对齐

如何寻址?

J-Format

伪直接寻址



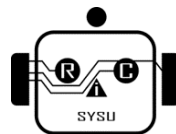
PC

高4位

低位"00"

Memory

Word



举例：汇编指令与机器指令的对应

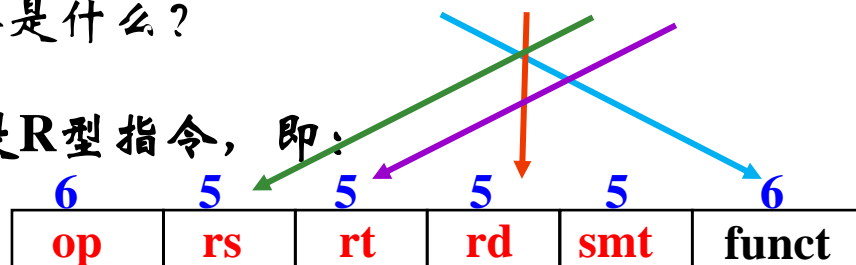
例1：若MIPS Assembly Instruction:

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

问题：如何知道是R型指令？

则对应MIPS的机器指令是什么？

从助记符表中查到add是R型指令，即：



根据汇编指令中的操作码助记符查表可知是何指令格式！

Decimal representation:

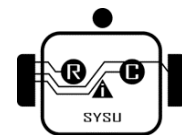
6	5	5	5	5	6
0	17	18	8	0	32

R-Type \$s1 \$s2 \$t0 No shift add

Binary representation:

6	5	5	5	5	6
000000	10001	10010	01000	00000	100000

这个过程称为“汇编”，所有汇编源程序都必须汇编成二进制机器代码



举例：汇编指令与机器指令的对应

例2：若从存储器取出的一条指令是00AF8020H，则对应的汇编指令是什么？

32位指令代码：0000 0000 1010 1111 1000 0000 0010 0000

指令的前6位为000000，根据指令解码表知，是一条R-Type指令，按照R-Type指令的格式：

31	6 bits	26	5 bits	21	5 bits	16	5 bits	11	5 bits	6	6 bits	0
op		rs		rt		rd		shamt		funct		
000000		00101		01111		10000		00000		100000		

得到：rs=00101, rt=01111, rd=10000, shamt=0, funct=100000

(1) 根据R-Type指令解码表，知是“add”操作(非移位操作)

(2) rs、rt、rd的十进制值分别为5、15、16，从MIPS寄存器功能表知：rs、rt、rd分别为：\$a1、\$t7、\$s0

故对应的汇编形式为：

add \$s0, \$a1, \$t7 功能：\$a1+\$t7 → \$s0

从机器指令→汇编指令：“反汇编”；可破解二进制代码(可执行程序)



3. 指令中的操作数

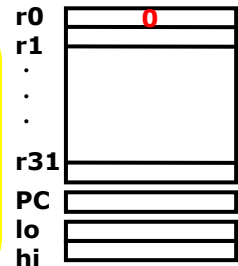
MIPS指令中的操作数

寄存器数据指定

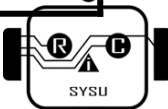
存储器数据指定

立即数/文本/位

- 32×32-bit GPRs ($r0 = 0$)
- 寄存器编号为5位
- 32×32-bit FP Regs ($f0 - f31$, paired DP)
- HI、LO、PC: 特殊寄存器



Name	number	Usage	Reserved on call?
zero	0	constant value =0(恒为0)	n.a.
at	1	reserved for assembler	n.a.
v0 – v1	2 – 3	values for results(过程调用返回值)	no
a0 – a3	4 – 7	Arguments(过程调用参数)	yes
t0 – t7	8 – 15	Temporaries(临时变量)	no
s0 – s7	16 – 23	Saved(保存)	yes
t8 – t9	24 – 25	more temporaries(其他临时变量)	no
k0 – k1	26 – 27	reserved for kernel(为OS保留)	n.a.
gp	28	global pointer(全局指针)	yes
sp	29	stack pointer (栈指针)	yes
fp	30	frame pointer (帧指针)	yes
ra	31	return address (过程调用返回地址)	yes



3. 指令中的操作数

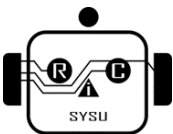
MIPS指令中的操作数

寄存器数据指定

存储器数据指定

立即数/文本/位

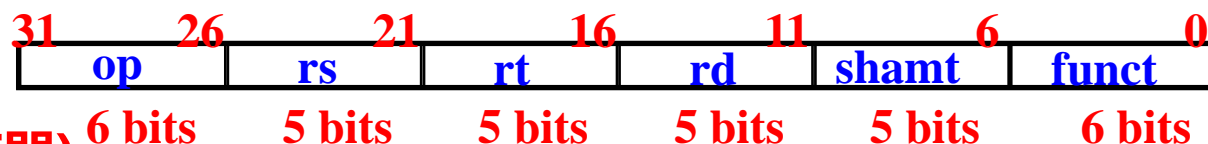
- 只能通过Load/Store指令访问存储器数据
- 字节编址，数据要求按边界对齐
- Big Endian（大端方式）
- 可访问空间： $2^{32}\text{bytes}=4\text{GB}$
- 访存地址通过一个32位寄存器内容加16位偏移量得到，16位偏移量为有符号整数



1. MIPS指令格式

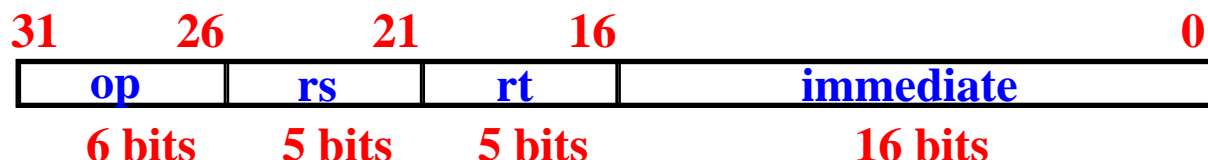
□ 所有指令都是32位宽，按字地址对齐

□ 三种指令格式



■ R-Type(用于寄存器)

- 两个操作数都是寄存器的运算指令。如: `sub rd, rs, rt`



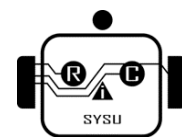
■ I-Type

- 运算指令：一个寄存器、一个立即数。如: `ori rt, rs, imm16`
- Load和Store指令。如: `lw rt, rs, imm16`
- 条件分支指令。如: `beq rs, rt, imm16`



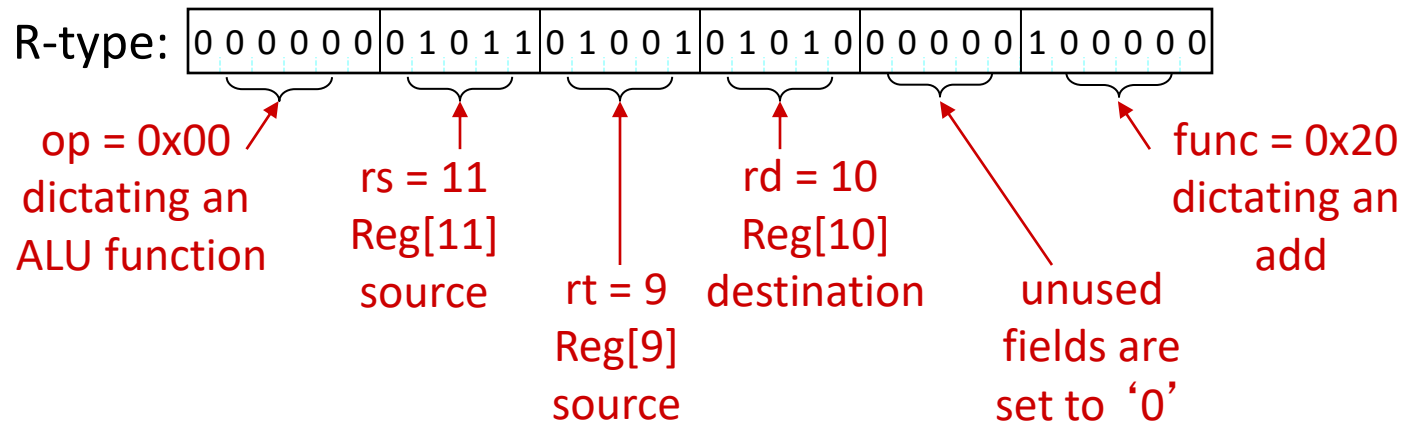
■ J-Type

- 无条件跳转指令。如: `j target`



MIPS ALU Operations

Sample coded operation: ADD instruction



References to register contents are prefixed by a "\$" to distinguish them from constants or memory addresses



What we prefer to write: `add $10, $11, $9` ("assembly language")

The convention with MIPS assembly language is to specify the destination operand first, followed by source operands.



`add rd, rs, rt:`

$\text{Reg}[\text{rd}] = \text{Reg}[\text{rs}] + \text{Reg}[\text{rt}]$

"Add the contents of rs to the contents of rt; store the result in rd"

Similar instructions for other ALU operations:

arithmetic: `add`, `sub`, `addu`, `subu`

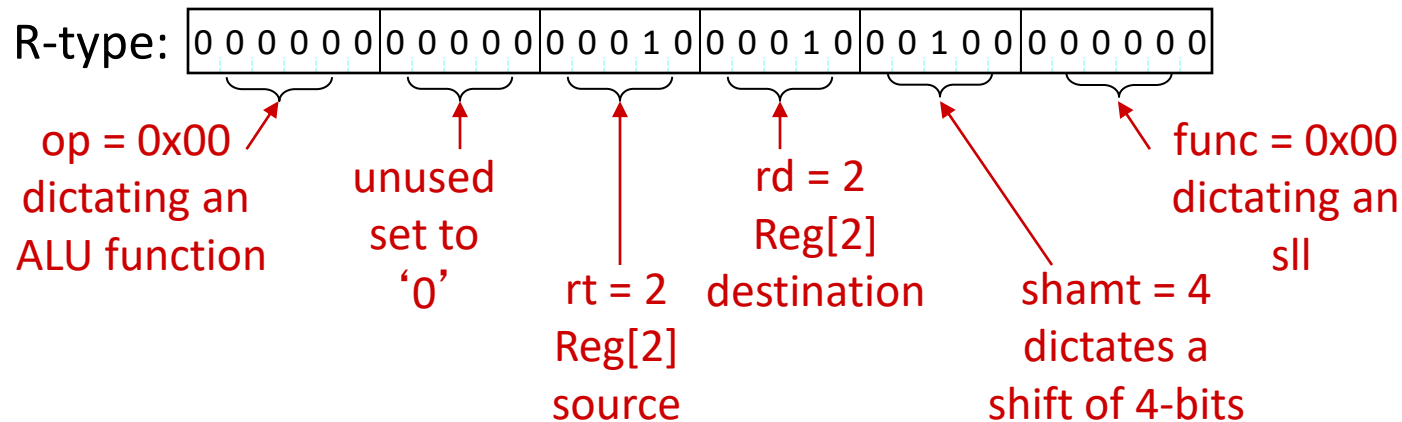
compare: `slt`, `sltu`

logical: `and`, `or`, `xor`, `nor`

shift: `sll`, `srl`, `sra`, `sllv`, `srav`, `srlv`

MIPS Shift Operations

Sample coded operation: SHIFT LOGICAL LEFT instruction



Assembly: `sll $2, $2, 4`

`sll rd, rt, shamt:`

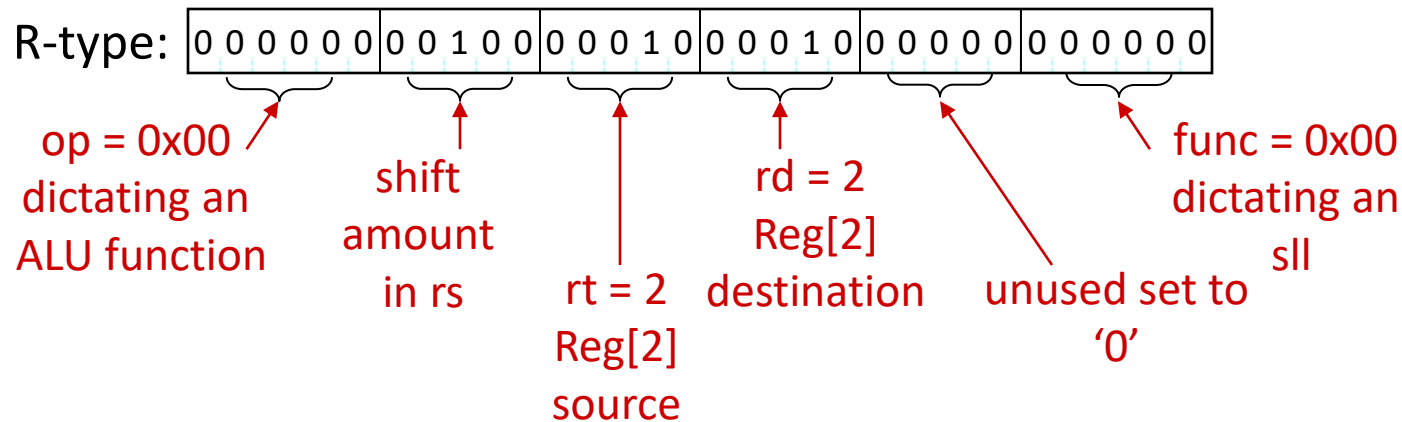
$\text{Reg}[rd] = \text{Reg}[rt] \ll \text{shamt}$

“Shift the contents of *rt* to the left by *shamt*; store the result in *rd*”

How are shifts useful?

MIPS Shift Operations

Sample coded operation: SLLV (SLL Variable)



This is peculiar syntax for MIPS, in this ALU instruction the *rt* operand precedes the *rs* operand. Usually, it's the other way around



Different flavor:
Shift amount is not in instruction, but in a register

Assembly: `sllv $2, $2, $4`

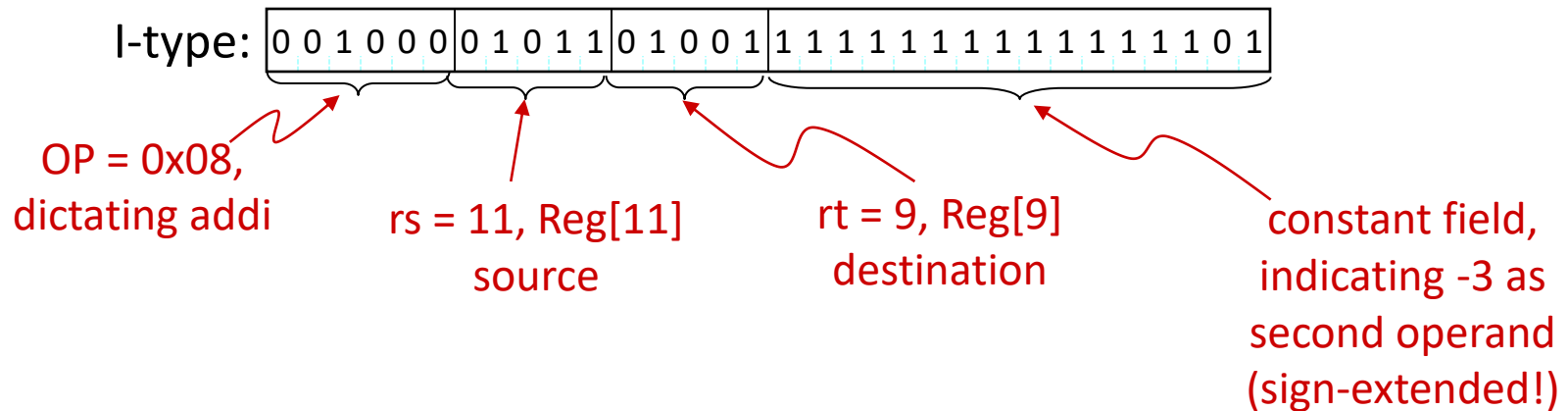
`sllv rd, rt, rs:`

$\text{Reg}[rd] = \text{Reg}[rt] \ll \text{Reg}[rs]$

“Shift the contents of *rt* left by the contents of *rs*; store the result in *rd*”

MIPS ALU Operations with Immediate

addi instruction: adds register contents, signed-constant:



Symbolic version: `addi $9, $11, -3`

`addi rt, rs, imm:`

$\text{Reg}[\text{rt}] = \text{Reg}[\text{rs}] + \text{sxt}(\text{imm})$

“Add the contents of rs to const;
store result in rt”

Similar instructions for other ALU operations:

arithmetic: `addi`, `addiu`
compare: `slti`, `sltiu`
logical: `andi`, `ori`, `xori`, `lui`

Immediate values are sign-extended for arithmetic and compare operations, but not for logical operations.



Why Built-in Constants? (Immediate)

- Where are constants/immediates useful?
 - SMALL constants used frequently (50% of operands)
 - In a C compiler (gcc) 52% of ALU operations use a constant
 - In a circuit simulator (spice) 69% involve constants
 - e. g. , $B = B + 1$; $C = W \& 0x00ff$; $A = B + 0$;

□ Examples:

```
addi $29, $29, 4
slti $8, $18, 10
andi $29, $29, 6
ori  $29, $29, 4
```

First MIPS Program (fragment)

□ Suppose you want to compute the expression:

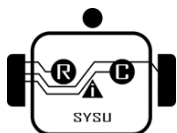
$$f = (g + h) - (i + j)$$

- where variables f , g , h , i , and j are assigned to registers \$16, \$17, \$18, \$19, and \$20 respectively
- what is the MIPS assembly code?

```
add $8,$17,$18      # (g + h)
add $9,$19,$20      # (i + j)
sub $16,$8,$9       # f = (g + h) - (i + j)
```

□ Questions to answer:

- How did these variables come to reside in registers?
- Answer: We need more instructions which allow data to be explicitly loaded from memory to registers, and stored from registers to memory



MIPS Register Usage Conventions

- Some MIPS registers assigned to specific uses
 - by convention, so programmers can combine code pieces
 - \$0 is hard-wired to the value 0

Name	Register number	Usage
\$zero	0	the constant value 0
\$at	1	assembler temporary
\$v0-\$v1	2-3	values for results and expression evaluation
\$a0-\$a3	4-7	arguments
\$t0-\$t7	8-15	temporaries
\$s0-\$s7	16-23	saved
\$t8-\$t9	24-25	more temporaries
\$gp	28	global pointer
\$sp	29	stack pointer
\$fp	30	frame pointer
\$ra	31	return address

□ More instructions

- signed vs. unsigned instructions
- larger constants
- accessing memory
- branches and jumps
- multiply, divide
- comparisons
- logical instructions

□ Reading

- Book Chapter 2.1–2.7

4. 指令类型

MIPS指令类型

算术逻辑运算指令

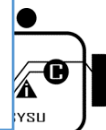
add, sub, multiply, divide, and, or, xor,....

数据传输指令

条件分支指令

无条件跳转指令

<i>Instruction</i>	<i>Example</i>	<i>Meaning</i>	<i>Comments</i>
add	add \$1,\$2,\$3	$\$1 = \$2 + \$3$	3 operands; excep. possible
subtract	sub \$1,\$2,\$3	$\$1 = \$2 - \$3$	3 operands; excep. possible
add immed.	addi \$1,\$2,100	$\$1 = \$2 + 100$	+ constant; excep. Possible
multiply	mult \$2,\$3	$Hi, Lo = \$2 \times \3	64-bit signed product
divide	div \$2,\$3	$Lo = \$2 \div \3 $Hi = \$2 \bmod \3	Lo = quotient, Hi = remainder



4. 指令类型

MIPS指令类型

算术逻辑运算指令

add, sub, multiply, divide, and, or, xor,....

数据传输指令

条件分支指令

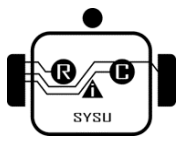
无条件跳转指令

<i>Instruction</i>	<i>Example</i>	<i>Meaning</i>	<i>Comments</i>
and	and \$1,\$2,\$3	$\$1 = \$2 \& \$3$	Logical AND
or immed.	ori \$1,\$2,20	$\$1 = \$2 \mid 20$	Bitwise-OR of constant
xor	xor \$1,\$2,\$3	$\$1 = \$2 \wedge \$3$	Logical XOR
nor	nor \$1,\$2,\$3	$\$1 = \sim(\$2 \mid \$3)$	Logical NOR



I-Format Instructions

- ❑ What about instructions with immediates?
 - ❑ 5-bit field only represents numbers up to the value 31: immediates may be much larger than this
 - ❑ Ideally, MIPS would have only one instruction format (for simplicity): unfortunately, we need to compromise
- ❑ Define new instruction format that is partially consistent with R-format:
 - ❑ First notice that, if instruction has immediate, then it uses at most 2 registers



I-Format Instructions

- Define “fields” of the following number of bits each: $6 + 5 + 5 + 16 = 32$ bits

6	5	5	16
---	---	---	----

- Again, each field has a name:

opcode	rs	rt	immediate
--------	----	----	-----------

- Key Concept:** Only one field is inconsistent with R-format. Most importantly, **opcode** is still in same location!

Working with Constants

- Immediate instructions allow constants to be specified within the instruction

- Examples

- add 2000 to register \$5

- `addi $5, $5, 2000`

- subtract 60 from register \$5

- `addi $5, $5, -60`

- ... no `subi` instruction!

- logically AND \$5 with 0x8723 and put the result in \$7

- `andi $7, $5, 0x8723`

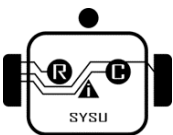
- put the number 1234 in \$10

- `addi $10, $0, 1234`

- But...

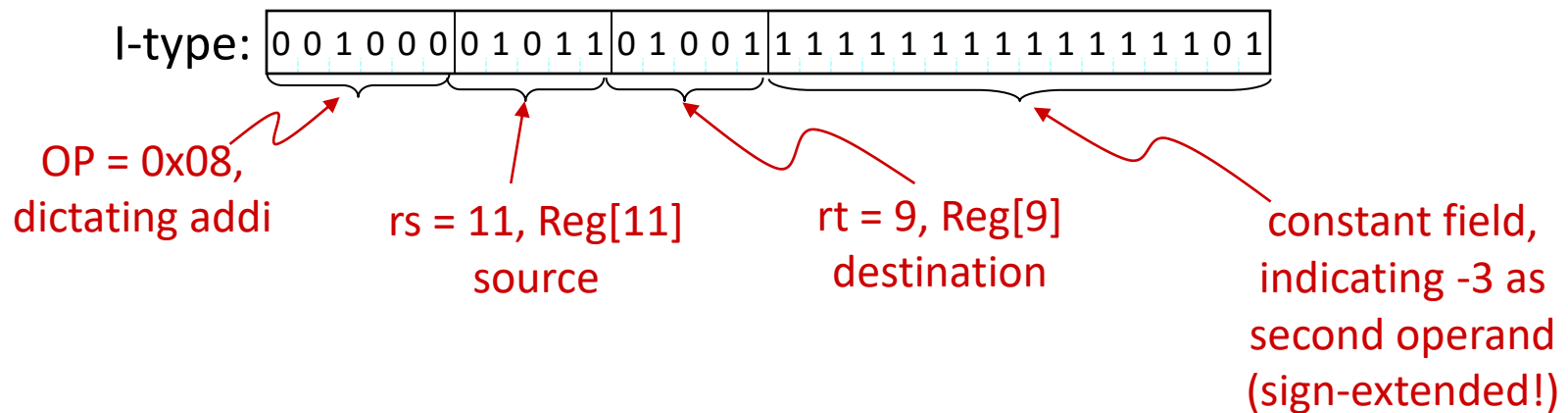
- these constants are limited to 16 bits only!

- Range is [-32768...32767] if signed, or [0...65535] if unsigned



Recap: ADDI

addi instruction: adds register contents, signed-constant:



Symbolic version: `addi $9, $11, -3`

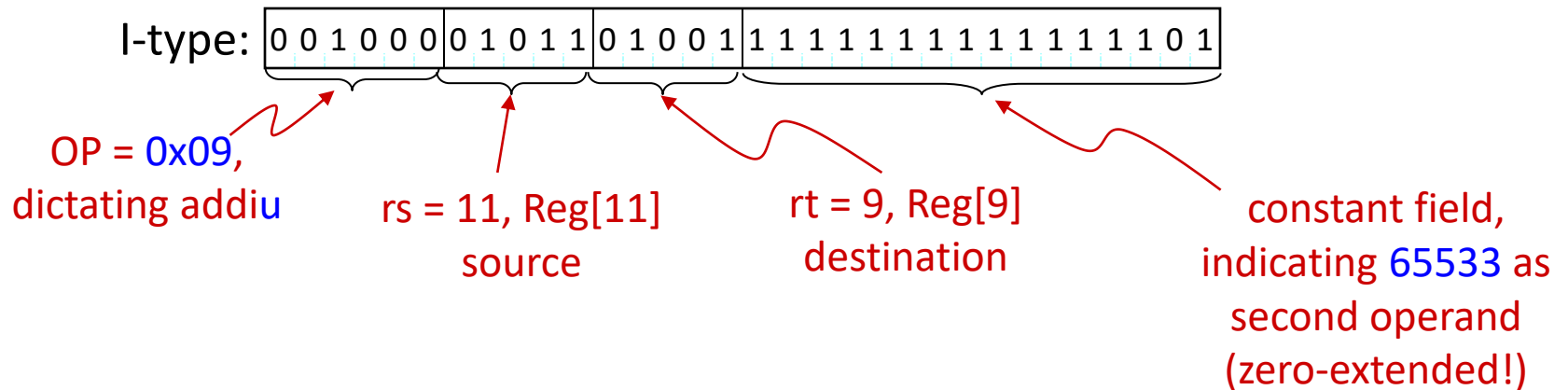
`addi rt, rs, imm:`

$\text{Reg}[\text{rt}] = \text{Reg}[\text{rs}] + \text{sxt}(\text{imm})$

“Add the contents of rs to const;
store result in rt”

ADDIU: Signed vs. Unsigned Constants

addiu instruction: adds register to unsigned-constant:



Symbolic version: addiu \$9, \$11, 65533

addiu rt, rs, imm:

$\text{Reg}[\text{rt}] = \text{Reg}[\text{rs}] + (\text{imm})$

“Add the contents of rs to const;
store result in rt”

Logical operations are always
“unsigned”, so always zero-
extended

How About Larger Constants?

❑ Problem: How do we work with bigger constants?

❑ Example: Put the 32-bit value 0x5678ABCD in \$5

❑ CLASS: How will you do it?

❑ One Solution:

❑ put the upper half (0x5678) into \$5

❑ then shift it left by 16 positions (0x5678 0000)

❑ now “add” the lower half to it (0x5678 0000 + 0xABCD)

```
addi $5, $0, 0x5678
```

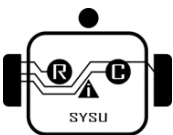
```
sll $5, $5, 16
```

```
addi $5, $5, 0xABCD
```

❑ One minor problem with this:

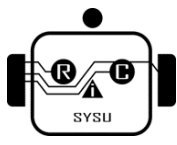
❑ **addi** can mess up by treating the constants are signed

❑ use **addiu** or ori instead



How About Larger Constants?

- ❑ Observation: This sequence is very common!
 - ❑ so, a special instruction was introduced to make it shorter
 - ❑ the first two (**addi + sll**) combo is performed by
lui
“load upper immediate”
 - ❑ puts the 16-bit immediate into the upper half of a register
- ❑ Example: Put the 32-bit value 0x5678ABCD in \$5
 - lui \$5, 0x5678**
 - ori \$5, \$5, 0xABCD**

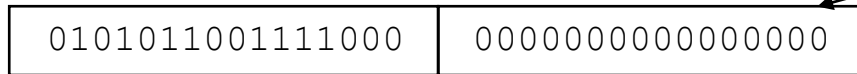


How About Larger Constants?

□ Look at this in more detail:

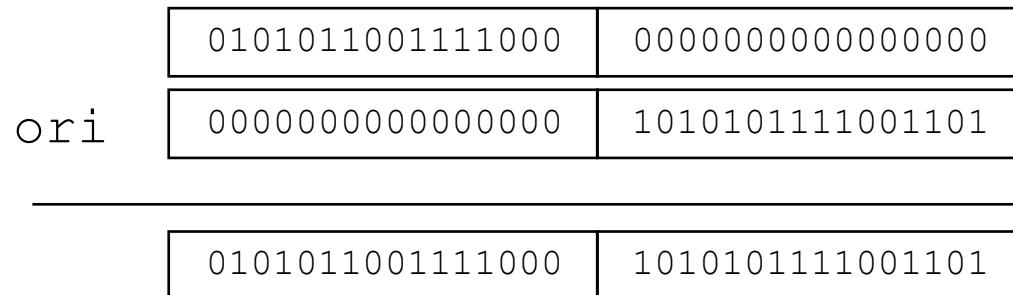
□ “load upper immediate”

lui \$5, 0x5678 // 0101 0110 0111 1000 in binary

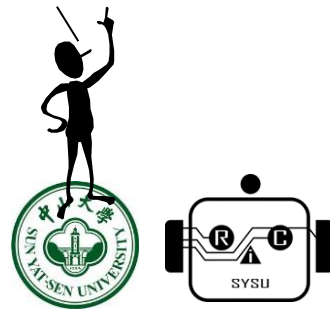


□ Then must get the lower order bits right

□ ori \$5, \$5, 0xABCD // 1010 1011 1100 1101



Reminder: In MIPS, logical Immediate instructions (ANDI, ORI, XORI) do not sign-extend their constant operand



Multiply and Divide

- Slightly more complicated than add/subtract
 - multiply: product is twice as long!
 - if A, B are 32-bit long, $A * B$ is how many bits?
 - divide: dividing integer A by B gives two results!
 - quotient and remainder
- Solution: two new special-purpose registers
 - “Hi” and “Lo”

Multiply

□ MULT instruction

- `mult rs, rt`

- Meaning: multiply contents of registers `$rs` and `$rt`, and store the (64-bit result) in the pair of special registers `{hi, lo}`

`hi:lo = $rs * $rt`

- upper 32 bits go into `hi`, lower 32 bits go into `lo`

□ To access result, use two new instructions

- `mfhi:` move from `hi`

`mfhi rd`

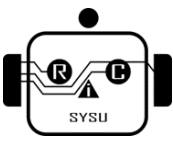
- move the 32-bit half result from `hi` to `$rd`

- `mflo:` move from `lo`

`mflo rd`

- move the 32-bit half result from `lo` to `$rd`

For example:
`mult $9,$8`
`mflo $9`
`mfhi $8`



Divide

- DIV instruction

- `div rs, rt`

- Meaning: divide contents of register `$rs` by `$rt`, and store the quotient in `lo`, and remainder in `hi`

- $lo = \$rs / \rt

- $hi = \$rs \% \rt

- To access result, use `mfhi` and `mflo`

- NOTE: There are also unsigned versions

- `multu`

- `divu`

Now we can do a real program: Factorial...

Synopsis (in C):

- Input in n, output in ans
- r1, r2 used for temporaries
- assume n is small

```
int n, ans, r1, r2;  
r1 = 1;  
r2 = n;  
while (r2 != 0) {  
    r1 = r1 * r2;  
    r2 = r2 - 1;  
}  
ans = r1;
```

MIPS code, in assembly language:

```
n:      .word    123  
ans:    .word    0
```

Now we can do a real program: Factorial...

Synopsis (in C):

- Input in n, output in ans
- r1, r2 used for temporaries
- assume n is small

```
int n, ans, r1, r2;  
r1 = 1;  
r2 = n;  
while (r2 != 0) {  
    r1 = r1 * r2;  
    r2 = r2 - 1;  
}  
ans = r1;
```

MIPS code, in assembly language:

```
n:      .word    123  
ans:    .word    0  
  
...  
addi $t0, $0, 1      # t0 = 1  
lw   $t1, n          # t1 = n  
loop: beq $t1, $0, done      # while (t1 != 0)  
      mult $t0, $t1      # hi:lo = t0 * t1  
      mflo $t0           # t0 = t0 * t1  
      addi $t1, $t1, -1   # t1 = t1 - 1  
      j     loop         # Always loop back  
done: sw   $t0, ans      # ans = r1
```


Comparison: `slt`, `slti`

- `slt` = set-if-less-than

- `slt rd, rs, rt`

- $\$rd = (\$rs < \$rt) \text{ // "1" if true and "0" if false}$

- `slti` = set-if-less-than-immediate

- `slti rt, rs, imm`

- $\$rt = (\$rs < \text{sign-ext}(\text{imm}))$

- also unsigned flavors

- `sltu`

- `sltiu`

Logical Instructions

- Boolean operations: bitwise on all 32 bits

- AND, OR, NOR, XOR

- and, andi

- or, ori

- nor // Note: There is no **nori**! Why?

- xor, xori

- Examples:

- and \$1, \$2, \$3

- \$1 = \$2 & \$3

- xori \$1, \$2, 0xFF12

- \$1 = \$2 ^ 0x0000FF12

- See all in textbook!

Logical Instructions

□ 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

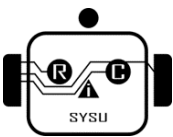
- Set some bits to 1, leave others unchanged

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

Binary Operations

- * 对寄存器\$t1的第7位置1, 第6位清0
- * 其他位保持不变
- * 将获得的值保存在\$t2



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

4. 指令类型

MIPS指令类型

算术逻辑运算指令

数据传输指令

条件分支指令

load, store

操作数长度的不同是由不同的操作码指定

无条件跳转指令

Instruction

Comment

Meaning

sw \$3, 500(\$4)

Store word

\$3 \rightarrow (\$4 + 500)

sh \$3, 502(\$2)

Store half

Low Half of \$3 \rightarrow (\$2 + 502)

sb \$2, 41(\$3)

Store byte

LQ of \$2 \rightarrow (\$3 + 41)

lw \$1, -30(\$2)

Load word

(\$2 - 30) \rightarrow \$1

lh \$1, 40(\$3)

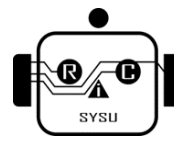
Load half

(\$3 + 40) \rightarrow LH of \$1

lb \$1, 40(\$3)

Load byte

(\$3 + 40) \rightarrow LQ of \$1



4. 指令类型

MIPS指令类型

算术逻辑运算指令

load, store

数据传输指令

条件分支指令

无条件跳转指令

Instruction

sw	\$3, 500(\$4)
sh	\$3, 502(\$2)
sb	\$2, 41(\$3)
lw	\$1, -30(\$2)
lh	\$1, 40(\$3)
lb	\$1, 40(\$3)

Comment

Store word
Store half
Store byte
Load word
Load half
Load byte

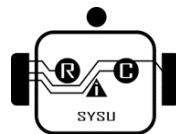
Meaning

\$3 \rightarrow (\$4 + 500)
Low Half of \$3 \rightarrow (\$2 + 502)
LQ of \$2 \rightarrow (\$3 + 41)
(\$2 - 30) \rightarrow \$1
(\$3 + 40) \rightarrow LH of \$1
(\$3 + 40) \rightarrow LQ of \$1



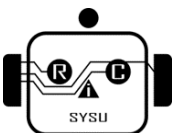
问题：为什么需要不同长度的操作数？

高级语言中的数据类型有char、short、int、long等，故需要存取不同长度的操作数；



Accessing Memory

- ❑ MIPS is a “load-store” architecture
 - ❑ all operands for ALU instructions are in registers or immediate
 - ❑ cannot directly add values residing in memory
 - ❑ must first bring values into registers from memory (called LOAD)
 - ❑ must store result of computation back into memory (called STORE)



MIPS Load Instruction

□ Load instruction is I-type



`lw rt, imm(rs)`

Meaning: $\text{Reg}[rt] = \text{Mem}[\text{Reg}[rs] + \text{sign-ext}(imm)]$

Abbreviation: `lw rt, imm` for `lw rt, imm($0)`

□ Does the following:

- takes the value stored in register $\$rs$
- adds to it the immediate value (signed)
- this is the address where memory is looked up
- value found at this address in memory is brought in and stored in register $\$rt$

MIPS Store Instruction

□ Store instruction is also I-type



`sw rt, imm(rs)`

Meaning: $\text{Mem}[\text{Reg}[\text{rs}] + \text{sign-ext}(\text{imm})] = \text{Reg}[\text{rt}]$

Abbreviation: `sw rt, imm` for `sw rt, imm($0)`

□ Does the following:

- takes the value stored in register `$rs`
- adds to it the immediate value (signed)
- this is the address where memory is accessed
- reads the value from register `$rt` and writes it into the memory at the address computed

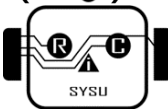
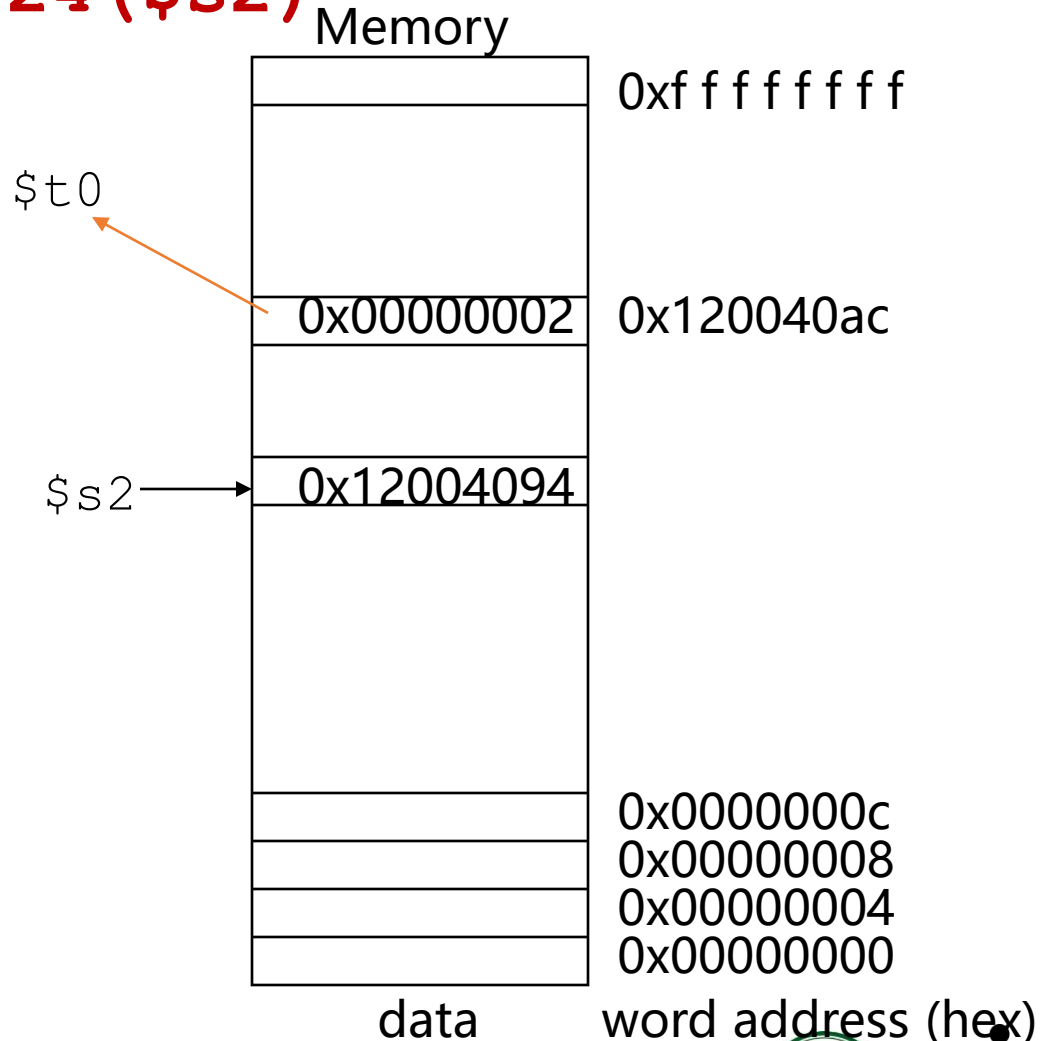
Memory Address Location

□ Example: `lw $t0, 24($s2)`

$$24_{10} + \$s2 =$$

$$\begin{array}{r} \dots 1001\ 0100 \\ + \dots 0001\ 1000 \\ \hline \dots 1010\ 1100 = \\ 0x120040ac \end{array}$$

Note that the **offset**
can be **positive** or
negative



MIPS Memory Addresses

- **lw** and **sw** read whole 32-bit words
 - so, addresses computed must be multiples of 4
 - $\text{Reg[rs]} + \text{sign-ext(imm)}$ must end in “00” in binary
 - otherwise: runtime exception
- There are also byte-sized flavors of these instructions
 - **lb** (load byte)
 - **sb** (store byte)
 - work the same way, but their addresses do not have to be multiples of 4

Storage Conventions

↙ Addr assigned at compile time

- Data and Variables are stored in memory
- Operations done on registers
- Registers hold Temporary results

1000:	n
1004:	r
1008:	x
100C:	y
1010:	

translates
to

or, more
humanely,
to

```
int x, y;  
y = x + 37;
```



Compilation approach:
LOAD, COMPUTE, STORE

```
lw $t0, 0x1008($0)  
addiu $t0, $t0, 37  
sw $t0, 0x100C($0)
```

```
x=【0x1008】  
y=【0x100C】  
lw $t0, x  
addiu $t0, $t0, 37  
sw $t0, y
```

↖ ↗
rs defaults to Reg[0] (0)

Byte Addresses

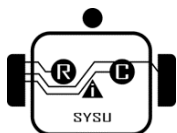
- Since bytes (8 bits) are so useful, most ISAs support addressing individual bytes in memory
- Therefore, the memory address of a word must be a multiple of 4 (alignment restriction)

□ **Big Endian:** leftmost byte is word address

IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA

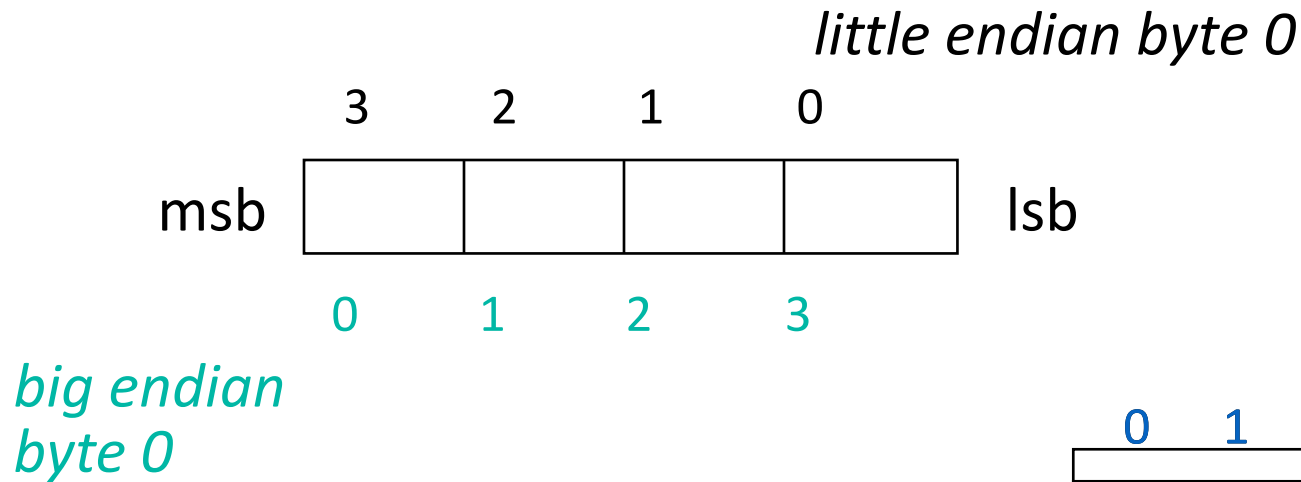
□ **Little Endian:** rightmost byte is word address

Intel 80x86, DEC Vax, DEC Alpha (Windows NT)

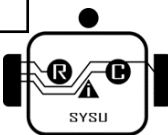
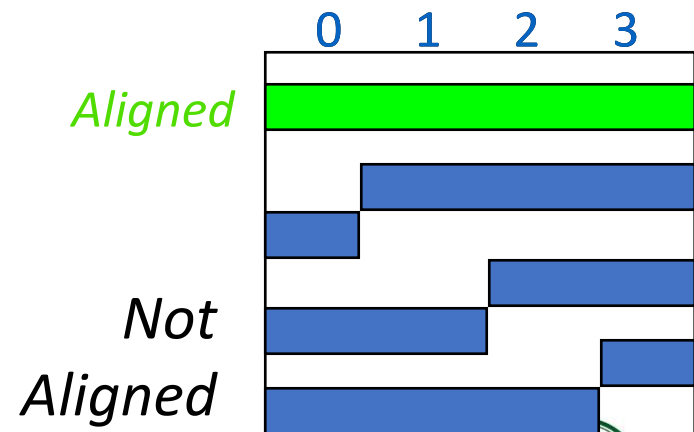


Addressing Objects: Endianness and Alignment

- **Big Endian (MIPS)**: leftmost byte is word address
- **Little Endian**: rightmost byte is word address



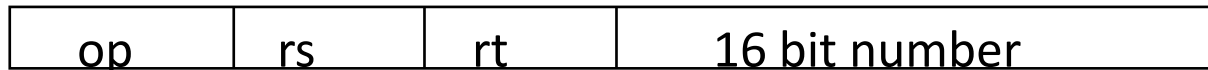
Alignment restriction: requires that objects fall on address that is multiple of their size



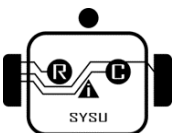
Loading and Storing Bytes

- ❑ MIPS provides special instructions to move bytes

```
lb    $t0, 1($s3)    #load byte from memory
sb    $t0, 6($s3)    #store byte to  memory
```



- ❑ What 8 bits get loaded and stored?
 - | load byte places the byte from memory in the rightmost 8 bits of the destination register
 - what happens to the other bits in the register?
 - | store byte takes the byte from the rightmost 8 bits of a register and writes it to the byte in memory
 - leaving the other bytes in the memory word unchanged



Example of Loading and Storing Bytes

- Given following code sequence and memory state what is the state of the memory after executing the code?

```
add    $s3, $zero, $zero
lb     $t0, 1($s3)
sb     $t0, 6($s3)
```

- What value is left in \$t0?

Memory	
0x 0 0 0 0 0 0 0 0	24
0x 0 0 0 0 0 0 0 0	20
0x 0 0 0 0 0 0 0 0	16
0x 1 0 0 0 0 0 1 0	12
0x 0 1 0 0 0 4 0 2	8
0x F F F F F F F F	4
0x 0 0 9 0 1 2 A 0	0
Data	

Word
Address (Decimal)

\$t0 = 0x00000090

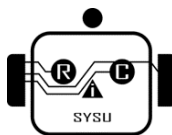
- What word is changed in Memory and to what?

mem(4) = 0xFFFF90FF

- What if the machine was **little Endian**?

\$t0 = 0x00000012

mem(4) = 0xFF12FF



Loading and Storing Half Words

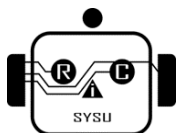
- ❑ MIPS also provides special instructions to move half words

lh \$t0, 1(\$s3) #load half word from memory

sh \$t0, 6(\$s3) #store half word to memory

op	rs	rt	16 bit number
----	----	----	---------------

- ❑ What 16 bits get loaded and stored?
 - | load half word places the half word from memory in the rightmost 16 bits of the destination register
 - what happens to the other bits in the register?
 - | store half word takes the half word from the rightmost 16 bits of the register and writes it to the half word in memory
 - leaving the other half word in the memory word unchanged



4. 指令类型

MIPS指令类型

算术逻辑运算指令

load, store

数据传输指令

条件分支指令

无条件跳转指令

Instruction

Comment

Meaning

sw \$3, 500(\$4)

Store word

\$3 \rightarrow (\$4 + 500)

sh \$3, 502(\$2)
502)

Store half

Low Half of \$3 \rightarrow (\$2 + 502)

sb \$2, 41(\$3)

Store byte

LQ of \$2 \rightarrow (\$3 + 41)

lw \$1, -30(\$2)

Load word

(\$2 - 30) \rightarrow \$1

lh \$1, 40(\$3)

Load half

(\$3 + 40) \rightarrow LH of \$1

lb \$1, 40(\$3)

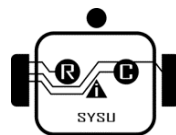
Load byte

(\$3 + 40) \rightarrow LQ of \$1



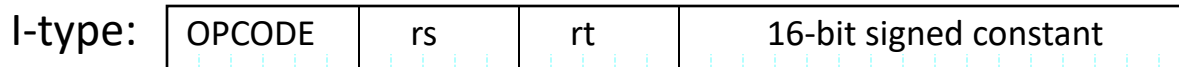
问题：为什么需要不同长度的操作数？

高级语言中的数据类型有char、short、int、long等，故需要存取不同长度的操作数；



MIPS Branch Instructions

MIPS *branch instructions* provide a way of conditionally changing the PC to some nearby location...



`beq rs, rt, label` # Branch if equal

`bne rs, rt, label` # Branch if not equal

```
if (REG[RS] == REG[RT])
{
    PC = PC + 4 + 4*offset;
}
```

```
if (REG[RS] != REG[RT])
{
    PC = PC + 4 + 4*offset;
}
```



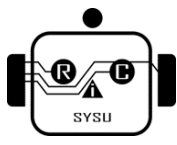
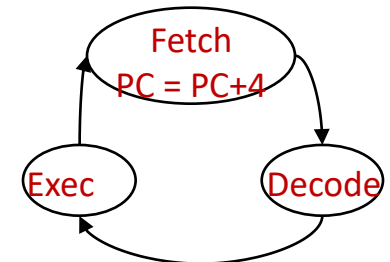
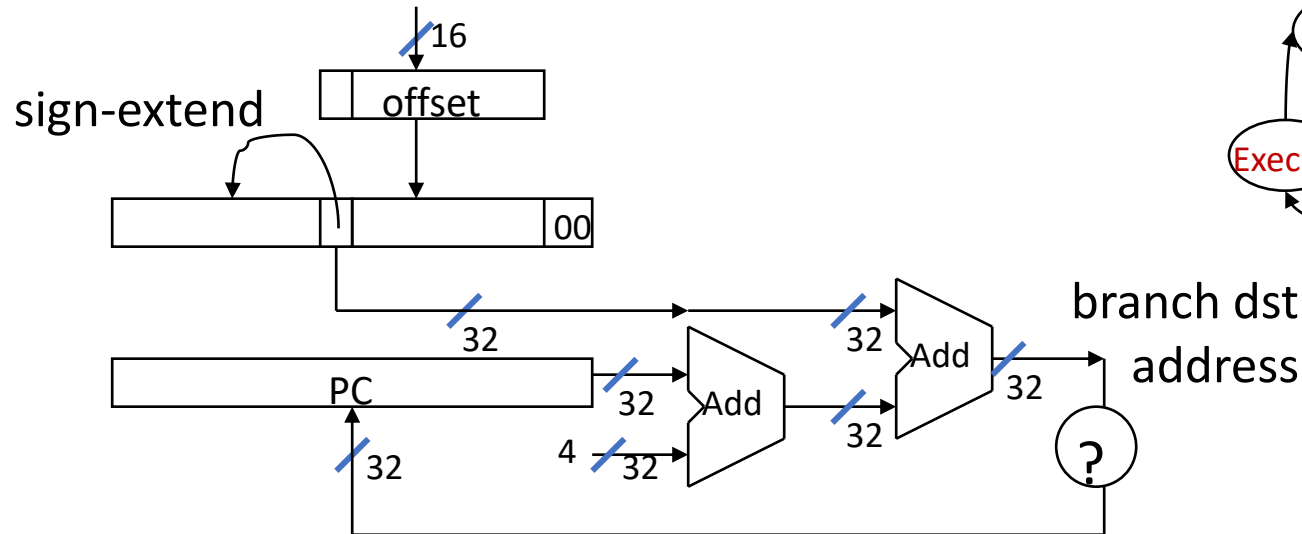
Notice on memory references offsets are multiplied by 4, so that branch targets are restricted to word boundaries.

NB: Branch targets are specified relative to the next instruction (which would be fetched by default). The assembler hides the calculation of these offset values from the user, by allowing them to specify a target address (usually a label) and it does the job of computing the offset's value. The size of the constant field (16-bits) limits the range of branches.

Disassembling Branch Destinations

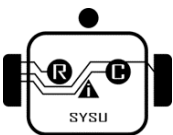
- ❑ The contents of the updated PC ($PC+4$) is added to the 16 bit branch offset which is converted into a 32 bit value by
 - ❑ concatenating two low-order zeros to make it a word address and then sign-extending those 18 bits
- ❑ The result is written into the PC if the branch condition is true – before the next Fetch cycle

from the low order 16 bits of the branch instruction



Offset Tradeoffs

- ❑ Why not just store the word offset in the low order 16 bits? Then the two low order zeros wouldn't have to be concatenated, it would be less confusing, ...
- ❑ That would limit the branch distance to -2^{15} to $+2^{15}-1$ instr's from the (instruction after the) branch
- ❑ And concatenating the two zero bits costs us very little in additional hardware and has no impact on the clock cycle time



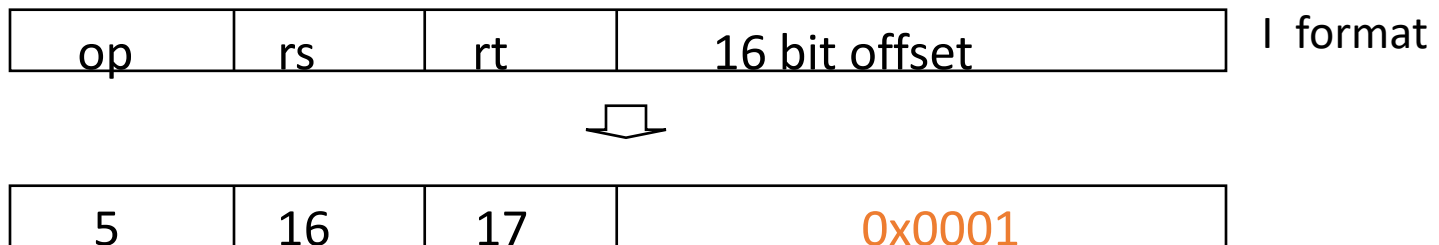
Assembling Branches Example

□ Assembly code

```
bne $s0, $s1, Lb11  
add $s3, $s0, $s1
```

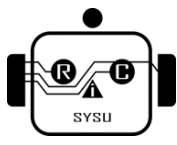
```
Lb11:    ...
```

□ Machine Format of bne:



□ Remember

- After the `bne` instruction is fetched, the PC is updated so that it is addressing the `add` instruction ($PC = PC + 4$).
- The offset (plus 2 low-order zeros) is sign-extended and added to the (updated) PC



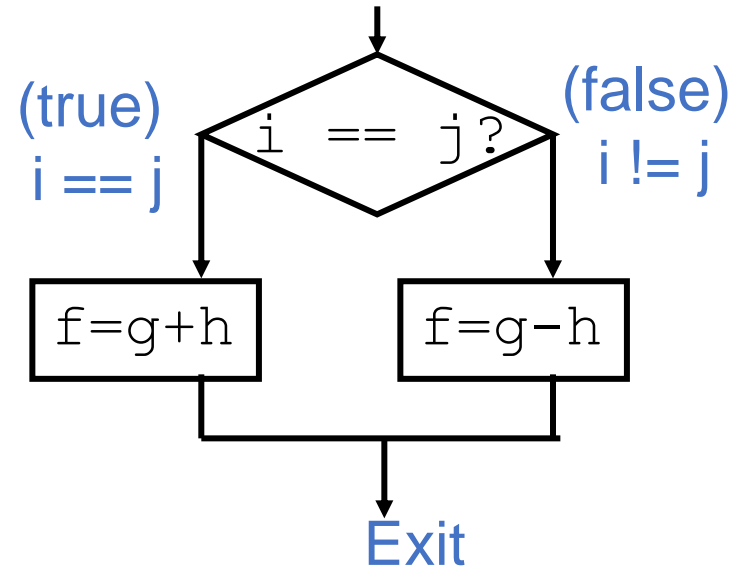
Compiling C `if` into MIPS (1/2)

□ Compile by hand

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

□ Use this mapping:

```
f:  $s0  
g:  $s1  
h:  $s2  
i:  $s3  
j:  $s4
```



Compiling C `if` into MIPS (2/2)

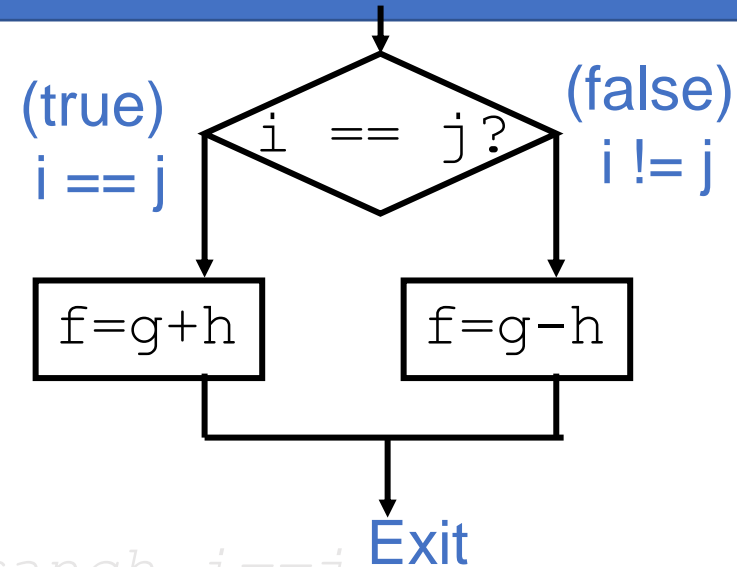
- **Compile by hand**

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

- ▣ Final compiled MIPS code:

```
        beq $s3,$s4,True    # branch i==j  
        sub $s0,$s1,$s2    # f=g-h (false)  
        j    Fin           # goto Fin  
True:   add $s0,$s1,$s2    # f=g+h (true)  
Fin:
```

Note: Compiler automatically creates labels to handle decisions (branches).



MIPS Jumps

- The range of MIPS branch instructions is limited to approximately $\pm 32\text{K}$ instructions ($\pm 128\text{K}$ bytes) from the branch instruction.
- To branch farther: an unconditional jump instruction is used.
- Instructions:
 - `j label` # jump to label ($\text{PC} = \text{PC}[31-28] \parallel \text{CONST}[25:0]*4$)
 - `jal label` # jump to label and **store PC+4 in \$31**
 - `jr $t0` # jump to address specified by register's contents
 - `jalr $t0, $ra` # jump to address specified by first register's contents and **store PC+4 in second register**

- Formats:
 - J-type: used for j
 - J-type: used for jal
 - R-type, used for jr
 - R-type, used for jalr

OP = 2	26-bit constant				
--------	-----------------	--	--	--	--

OP = 3	26-bit constant				
--------	-----------------	--	--	--	--

OP = 0	r_s	0	0	0	func = 8
--------	-------	---	---	---	----------

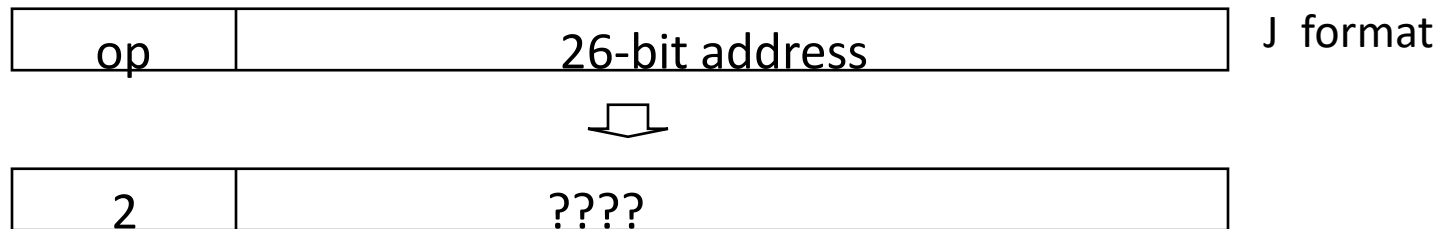
OP = 0	r_s	0	r_d	0	func = 9
--------	-------	---	-------	---	----------

Assembling Jumps

□ Instruction:

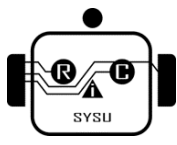
j Lbl #go to Lbl

□ Machine Format:



□ How is the jump destination address specified?

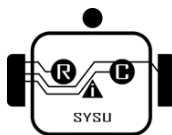
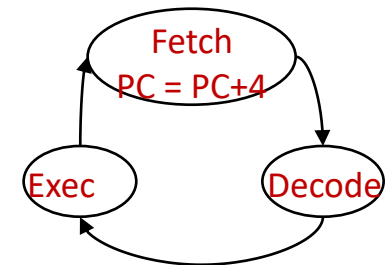
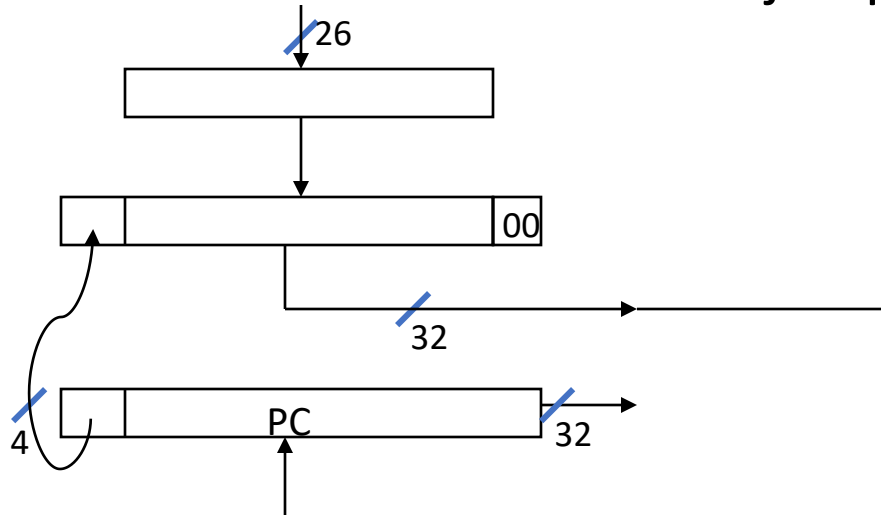
- | As an absolute address formed by
 - concatenating 00 as the 2 low-order bits to make it a word address
 - concatenating the upper 4 bits of the current PC (now PC+4)



Disassembling Jump Destinations

- The low order 26 bits of the jump instr converted into a 32 bit jump destination address by
 - concatenating two low-order zeros to create an 28 bit (word) address and then concatenating the upper 4 bits of the current PC (now PC+4) to create a 32 bit (word) address that is put into the PC prior to the next Fetch cycle

from the low order 26 bits of the jump instruction



Compiling Loop Statements

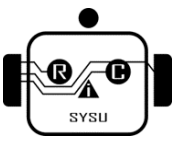
* C code:

```
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:  ...
```

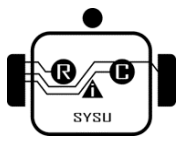



Assembling Branches and Jumps

- Assemble the MIPS machine code (in decimal is fine) for the following code sequence. Assume that the addr of the beq instr is $0x00400020_{\text{hex}}$

```
beq    $s0, $s1, Else
add    $s3, $s0, $s1
j      Exit
Else:  sub    $s3, $s0, $s1
Exit:  ...
```

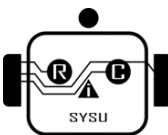
0x00400020	4	16	17	2	
0x00400024	0	16	17	19	0 0x20
0x00400028	2	0000	0100	0	... 0 0011 00 ₂
					jmp dst = (0x0) 0x040003 00 ₂ (00 ₂)
					= 0x00400030
0x0040002c	0	16	17	19	0 0x22
0x00400030	...				



Instruction Support for Functions (2/6)

```
C    ... sum(a,b); ... /* a,b:$s0,$s1 */  
    }  
    int sum(int x, int y) {  
        return x+y;  
    }
```

	address	
M	1000	add \$a0,\$s0,\$zero # x = a
I	1004	add \$a1,\$s1,\$zero # y = b
	1008	addi \$ra,\$zero,1016 # \$ra=1016
P	1012	j sum #jump to sum
S	1016	...
	2000	sum: add \$v0,\$a0,\$a1
	2004	jr \$ra # <i>new instruction</i>



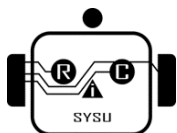
Instruction Support for Functions (3/6)

C ... sum(a,b) ; ... /* a,b:\$s0,\$s1 */
}
int sum(int x, int y) {
 return x+y;
}

M • Question: Why use **jr** here? Why not simply use **j**?

• Answer: **sum** might be called by many places, so we can't return to a fixed place. The calling proc to **sum** must be able to say "return here" somehow.

2000 sum: add \$v0,\$a0,\$a1
2004 jr \$ra # new instruction



Instruction Support for Functions (4/6)

- Single instruction to jump and save return address:
jump and link (jal)

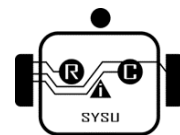
- Before:

```
1008 addi $ra,$zero,1016 #$ra=1016  
1012 j sum #goto sum
```

- After:

```
1008 jal sum # $ra=1012,goto sum
```

- Why have a jal? Make the common case fast: function calls are very common. (Also, you don't have to know where the code is loaded into memory with jal.)

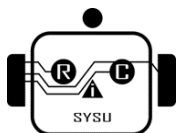


Instruction Support for Functions (5/6)

- Syntax for `jal` (jump and link) is same as for `j` (jump):

`jal label`

- `jal` should really be called `laj` for “link and jump” :
 - Step 1 (link): Save address of *next* instruction into `$ra` (Why next instruction? Why not current one?)
 - Step 2 (jump): Jump to the given label



Instruction Support for Functions (6/6)

- Syntax for `jr` (jump register):

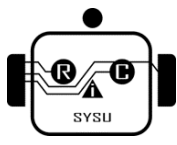
`jr register`

- Instead of providing a label to jump to, the `jr` instruction provides a register which contains an address to jump to.

- Very useful for function calls:

- `jal` stores return address in register (`$ra`)

- `jr $ra` jumps back to that address



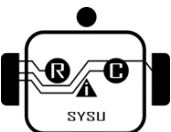
Branching Far Away

- What if the branch destination is further away than can be captured in 16 bits?
- **The assembler comes to the rescue – it inserts an unconditional jump to the branch target and inverts the condition**

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

becomes

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



联系方式

□ Acknowledgements:

□ This slides contains materials from following lectures:

- Computer Architecture (ETH, NUDT, USTC)

□ Research Area:

- 计算机视觉与机器人应用计算加速,
- 人工智能和深度学习芯片及智能计算机

□ Contact:

- 中山大学计算机学院
- 管理学院D101 (图书馆右侧)
- 机器人与智能计算实验室
- cheng83@mail.sysu.edu.cn

