

计算机组成与实现

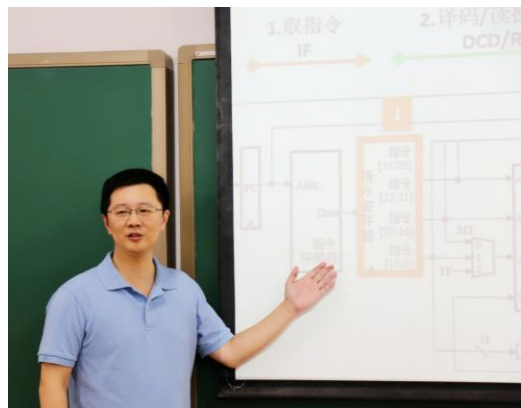
计算机指令

高小鹏

北京航空航天大学计算机学院

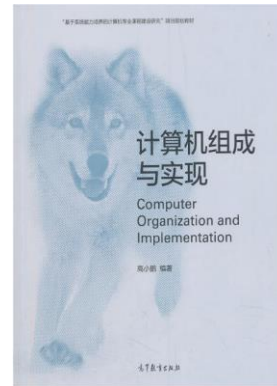
个人基本情况

- 职务：计算机学院副院长
- 主讲：计算机组成、计算机组成课程设计
- 联系方式
 - ◆ 手机：13911392138
 - ◆ QQ：2250422348
 - ◆ WX：tacs2007



概述

- 授课总时长：16学时
- 教学内容
 - ◆ MIPS指令系统：指令格式、指令集、汇编
 - ◆ 单周期CPU：数据通路、控制器、**单周期工程化开发方法**
 - ◆ 多周期CPU：数据通路、控制器、**多周期工程化开发方法**
- 课程教材
 - ◆ 计算机组成与实现，高等教育出版社
- 课程特点
 - ◆ 强调方法学，重视逻辑推演



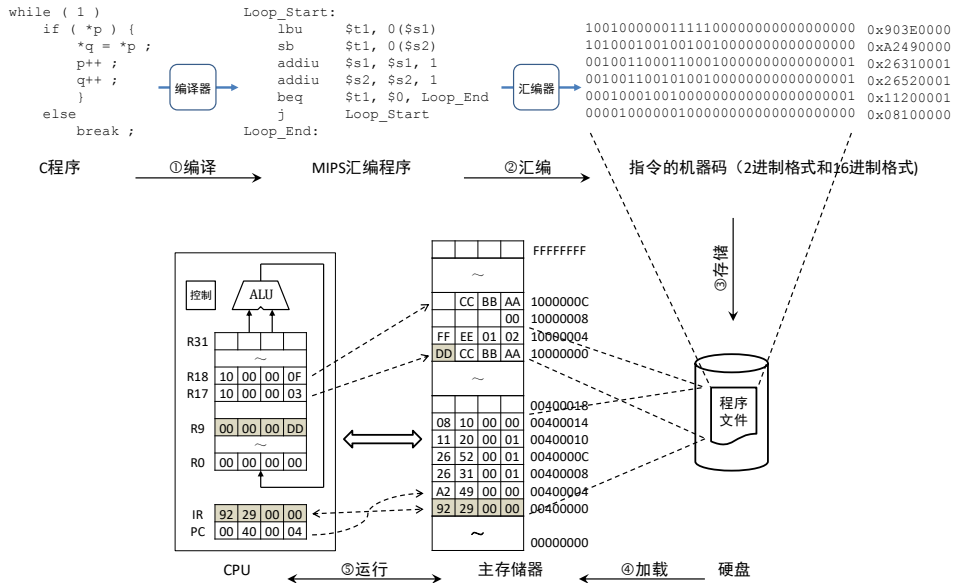
3

目录

- **程序执行的基本原理**
- 指令格式及其操作数
- 指令集与汇编程序
- 指令编码
- 汇编与反汇编实战

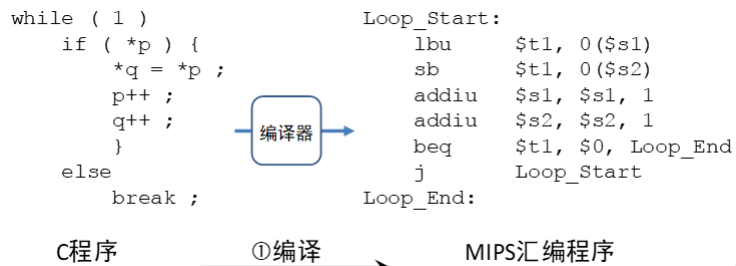
从C代码到可执行文件运行^{1/6}

- 大体涉及5个环节：编译、汇编、存储、加载、运行



从C代码到可执行文件运行^{2/6}

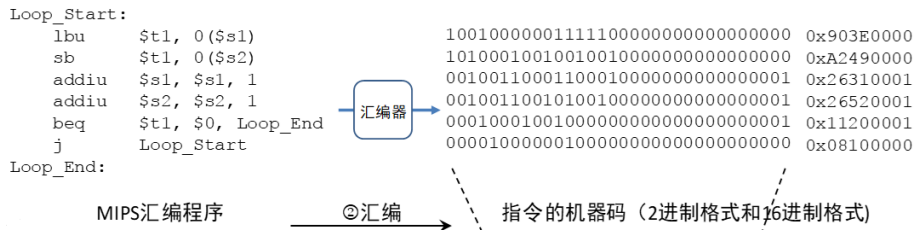
- CPU不能直接执行C源代码，必须利用编译器将C源代码转换为对应的汇编程序
- ◆ C源代码：其描述方式是适合人书写和阅读的
 - ◆ 汇编代码：用更加接近CPU可以理解和执行的语言编写的程序



编译相关内容属于
编译器技术范畴

从C代码到可执行文件运行^{3/6}

- CPU也不能直接执行汇编代码，必须利用汇编器将汇编代码转换为对应的二进制机器码
 - ◆ 汇编程序：每行对应一条机器指令
 - ◆ 机器指令：由一组二进制01串组成，是CPU可以理解与执行的



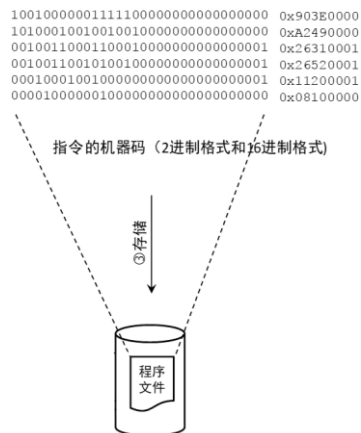
汇编相关内容属于
编译器技术范畴

7

计算机组成与实现

从C代码到可执行文件运行^{4/6}

- C程序被编译为一组CPU指令后，就以文件方式被存储在硬盘中
 - ◆ 可执行文件是由CPU指令及相关数据组成的
 - ◆ 由于CPU指令及相关数据以二进制方式存储，因此可执行文件是不适合人阅读的



文件相关内容属于
操作系统技术范畴

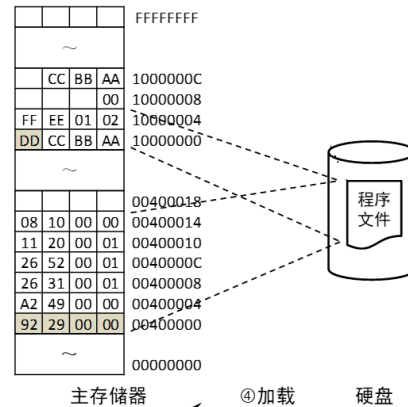
8

计算机组成与实现

从C代码到可执行文件运行^{5/6}

- 可执行文件要被加载到主存后才能被执行。指令部分和数据部分会被分别加载到主存中的不同区域
 - 代码段：这部分主存区域存储的是CPU指令
 - 数据段：这部分主存区域存储的是数据

将可执行文件加载到内存属于操作系统技术范畴
代码段、数据段等内容也与编译器技术相关

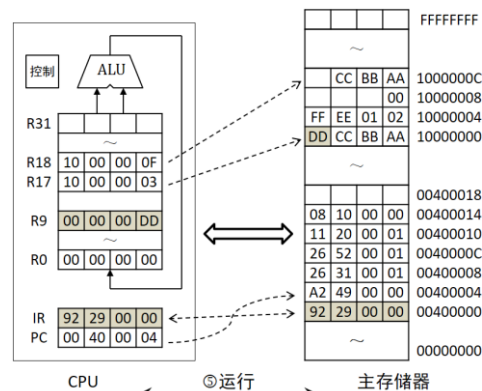


9

计算机组成与实现

从C代码到可执行文件运行^{6/6}

- CPU执行程序的基本过程就是不断的读取、分析和执行指令
 - 读取指令：CPU从主存的代码段中读取一条指令到内部
 - 分析指令：CPU分析指令的功能
 - 执行指令：CPU控制内部的功能部件执行相应的操作
- CPU有读、写主存的指令
 - 负责在CPU与主存间传输数据



10

计算机组成与实现

机器语言^{1/2}

- 指令：CPU理解的“单词”
- 指令集：CPU理解的全部“单词”集合
- Q1：为什么有时不同的计算机使用相同的指令集？
 - ◆ 例如：iPhone与iPad使用相同的指令集（都是ARM）
- Q2：为什么有时不同的计算机使用不同的指令集？
 - ◆ 例如：iPhone与Macbook使用不同的指令集（前者是ARM，后者是X86）

ISA~Instruction Set Architecture

11

计算机组成与实现

机器语言^{2/2}

- 如果只有一种ISA
 - ◆ 可以很好的利用公共软件，如编译器、操作系统等
- 如果有多种ISA
 - ◆ 针对不同的应用可以选择更适用的ISA
 - ◆ 不同的指令集有不同的设计平衡性考虑
 - 功能、性能、存储器、功耗、复杂度。。。。
 - ◆ 会激发竞争和创新

12

计算机组成与实现

为什么要学习汇编？

- 在更深层次理解计算机行为
 - ◆ 学习如何写更紧凑和有效的代码
 - ◆ 某些情况下，手工编码的优化水平比编译器高
- 对于资源紧张的应用，可能只适合手工汇编
 - ◆ 例如：分布式传感器应用
 - 为了降低功耗和芯片大小，甚至没有OS和编译器

13

计算机组成与实现

RISC

- *Complex Instruction Set Computing* (CISC)
 - ◆ 指令集设计早期阶段倾向于：应用有什么操作模式，就增加对应的指令。这导致了CISC
- *Reduced Instruction Set Computing* (RISC)
 - ◆ 另一种对立的设计哲学
 - ◆ 自然界存在2-8定律。程序也类似，为什么？
- RISC的指导思想
 - ◆ 1) 加速大概率事件
 - ◆ 2) 简单意味着更容易设计、电路频率更高
 - ◆ 3) 简单功能由硬件实现；复杂功能（由大量小功能组成）交给软件处理
 - 隐含的物理背景：复杂的功能也是小概率的

14

计算机组成与实现

RISC设计原则

- 指导思想：CPU越简单，性能越高
- 设计目标：减少指令数量，去除复杂指令（等效于降低复杂度）
- RISC的基本策略
 - ◆ 指令定长：所有指令都占用32位（1个字）
降低了从存储器中读取指令的复杂度
 - ◆ 简化指令寻址模式：以基地址+偏移为主
降低了从主存中读取操作数的复杂度
 - ◆ ISA的指令不仅数量少，而且简单
降低了指令执行的复杂度
 - ◆ 只有load与store两类指令能够访存
例如，不允许寄存器+存储器或存储器+存储器
 - ◆ 把复杂留给编译
编译器将高层语言复杂语句转换为若干简单的汇编指令

15

计算机组成与实现

主流的ISA

- Intel 80x86
 - ◆ PC、服务器、笔记本
- ARM（Advanced RISC Machine）
 - ◆ 手机、平板
 - ◆ 出货量最大的RISC：是x86的20倍
- PowerPC
 - ◆ IBM/Motorola/Apple联盟的产物
 - ◆ 航空电子设备：飞控、机载雷达等
 - ◆ 网络设备：交换机、路由器
 - ◆ 引擎控制器

16

计算机组成与实现

为什么选择MIPS

- 真实：工业界实际使用的CPU
 - ◆ 是设计师在实践中多次迭代、反复权衡的产物
 - ◆ 学习标准就是在学习设计师的思考方法与过程
- 简明：类别有限、结构简单、层次清晰，易于实现
 - ◆ MIPS是RISC的典型代表
- 生态：软件开发环境丰富，易于学习和实践
 - ◆ 多种模拟器、C编译等

17

计算机组成与实现



北京航空航天大学计算机学院
School of Computer Science and Engineering, Beihang University

目录

- 程序执行的基本原理
- 指令格式及其操作数
 - ◆ 指令基本格式
- 指令集与汇编程序
- 指令编码
- 汇编与反汇编实战

MIPS指令^{1/2}

- 指令的**一般性**语法格式：1个操作符，3个操作数

op dst, src1, src2

- ◆ op: 指令的基本功能
- ◆ dst: 保存结果的寄存器 (“destination”)
- ◆ src1: 第1个操作数 (“source 1”)
- ◆ src2: 第2个操作数 (“source 2”)

- 🗨️ 固定的格式：①有助于人的记忆和书写；②有助于使得硬件简单

- ◆ 硬件越简单，延迟就越小，时钟频率就越高

19

计算机组成与实现

MIPS指令^{2/2}

- 每条指令只有1个操作
- 每行写一条指令
- 很多指令与C运算高度相关
 - ◆ 如：=, +, -, *, /, &, |
- 一行C代码会对应多条指令

20

计算机组成与实现

目录

- 程序执行的基本原理
- 指令格式及其操作数
 - ◆ 第1类操作数：寄存器
- 指令集与汇编程序
- 指令编码
- 汇编与反汇编实战

计算机硬件的操作数

- C程序：变量的数量仅仅受限于内存容量
 - ◆ 程序员通常认为内存“无限大”，故声明变量时一般不考虑变量的数量
- ISA：有一组数量有限且固定的操作数，称之为寄存器
 - ◆ 寄存器被内置在CPU内部
 - ◆ 寄存器的优势：速度极快（工作速度小于1ns）
 - ◆ 寄存器的劣势：数量少

寄存器^{1/2}

□ MIPS寄存器数量：32

- ◆ 每个寄存器的宽度都是32位
- ◆ 寄存器没有类型（即无正负）
 - 根据指令的功能来解读寄存器值的正负
 - 即32位编码是按无符号还是符号解读，取决于指令

预留思考题

MIPS寄存器为什么不是16个，也不是64个？！

□ 寄存器数量的是设计均衡的体现

- ◆ 均衡的要素：性能与可用性
- ◆ 数量少：结构简单，速度快，能够存储在CPU内的数据少
- ◆ 数量多：结构复杂，速度慢，能够存储在CPU内的数据多

23

计算机组成与实现

寄存器^{2/2}

□ 寄存器编号：0~31

□ 寄存器表示：\$x（x为0~31），即\$0~\$31

□ 寄存器名字

- ◆ 程序员变量寄存器
 - \$s0-\$s7↔\$16-\$23
- ◆ 临时变量寄存器
 - \$t0-\$t7↔\$8-\$15
 - \$t8-\$t9↔\$20-\$21

编号	名称	用途
0	\$zero	常量0
1	\$at	汇编器保留
2-3	\$v0~\$v1	返回值
4-7	\$a0~\$a4	参数
8-15	\$t0-\$t7	临时变量
16-23	\$s0-\$s7	程序变量
24-25	\$t8-\$t9	临时变量
26-27	\$k0-\$k1	操作系统临时变量
28	\$gp	全局指针
29	\$sp	栈指针
30	\$fp	帧框架指针
31	\$ra	返回地址

本课程要学习的寄存器

注意

使用寄存器名字会让代码可读性更好

计算机组成与实现

MIPS指令示例^{1/2}

- 假设：变量a, b和c分别存储在\$s1, \$s2和\$s3

◆ $a \leftrightarrow \$s1, b \leftrightarrow \$s2, c \leftrightarrow \$s3$

- 整数加法指令

◆ C: $a = b + c$

◆ MIPS: `add $s1, $s2, $s3`

- 整数减法指令

◆ C: $a = b - c$

◆ MIPS: `sub $s1, $s2, $s3`

25

计算机组成与实现



MIPS指令示例^{2/2}

- 假设： $x \leftrightarrow \$s0, a \leftrightarrow \$s1, b \leftrightarrow \$s2, c \leftrightarrow \$s3, d \leftrightarrow \$s4$

- C语句: $x = (a + b) - (c + d) ;$

- MIPS汇编程序片段

执行 序 ↓	1	<code>add \$t1, \$s3, \$s4</code>	# t1 = c+d
	2	<code>add \$t2, \$s1, \$s2</code>	# t2 = a+b
	3	<code>sub \$s0, \$t2, \$t1</code>	# a = (a+b) - (c+d)

	
MIPS汇编程序	注释

- ◆ \$t1, \$t2: 临时变量寄存器



- ◆ 注释: 提高可读性; 帮助追踪寄存器/变量的分配与使用

- #: 是注释语句的开始

26

计算机组成与实现

0号寄存器

- 由于0在程序中的频度极高，为此MIPS设置了0号寄存器
 - ◆ 表示方法：\$0或\$zero
 - ◆ 值恒为0：读出的值恒为0；写入的值被丢弃
 - 指令的dst为\$0：指令使用本身无错，但执行时没有实际意义
- 示例

假设：a \longleftrightarrow \$s1, b \longleftrightarrow \$s2, c \longleftrightarrow \$s3, d \longleftrightarrow \$s4

```
1  add $s3, $0, $0      # c=0
2  add $s1, $s2, $0     # a=b
```

27

计算机组成与实现



北京航空航天大学计算机学院
School of Computer Science and Engineering, Beihang University

目录

- 程序执行的基本原理
- 指令格式及其操作数
 - ◆ 第2类操作数：立即数
- 指令集与汇编程序
- 指令编码
- 汇编与反汇编实战

立即数

- 指令中出现的**常量数值**被称为立即数
- 语法格式

op dst, src, *imm*

- 立即数替代了第2个操作数

- 示例

假设: $a \leftrightarrow \$s1$, $b \leftrightarrow \$s2$, $c \leftrightarrow \$s3$, $d \leftrightarrow \$s4$

```
1  addi $s1, $s2, 5      # a=b+5
2  addi $s3, $s3, 1      # c++
```

问题
为什么MIPS不设置subi指令?
*subi: 减立即数

immediate~立即数

29

计算机组成与实现



北京航空航天大学计算机学院
School of Computer Science and Engineering, Beihang University

目录

- 程序执行的基本原理
- 指令格式及其操作数**
 - 第3类操作数：主存单元**
- 指令集与汇编程序
- 指令编码
- 汇编与反汇编实战

主存单元

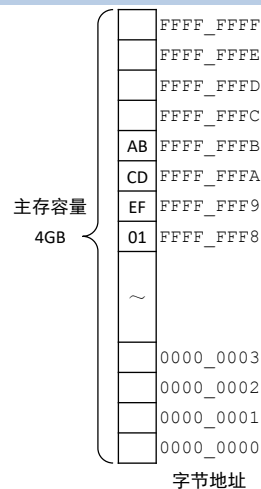
- 仅使用寄存器的挑战：有限的寄存器无法满足无限的变量需求
 - ◆ 1) 变量的个数无限：理论上，程序员可以定义任意多变量
 - ◆ 2) 变量的容量巨大：如数组这样的大型数据结构
- 解决问题途径：主存
 - ◆ 绝大多数变量（包括数据）存储在主存中
 - ◆ 需要使用时再将其加载至寄存器中
- CPU访问主存，需要解决问题：
 - ◆ 1) CPU如何看待主存的？这就是存储视图
 - ◆ 2) CPU如何定位某个主存单元？这就是寻址方式
 - ◆ 3) CPU如何访问访问主存单元？

31

计算机组成与实现

主存的抽象模型^{1/3}

- 主存：可以被抽象为一个数组
 - ◆ 概念1：主存单元的颗粒度
 - 字节：是存储器最常用的存储单位
 - ◆ 概念2：地址
 - 主存单元的编号就是地址，等同于数组下标
- 对于4GB主存
 - ◆ 4GB主存共有4G个字节，即有4G个存储单元
 - ◆ 4G个存储单元对应的地址位数为32位
 - $2^{32}=4G$
 - ◆ 地址范围：0000_0000h至FFFF_FFFFh



bit~比特, byte~字节, word~字

计算机组成与实现

主存的抽象模型^{2/3}

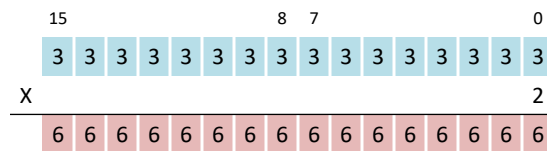
概念3：CPU字长

- ◆ CPU字长一般是指CPU一条指令可以计算的数据的宽度
- ◆ 例如MIPS的字长是32位，即单条MIPS指令可以计算的数据为32位

注意：CPU字长只是限制了单条指令的计算能力，但是完全可以通过组合多条指令及必要的存储单元来计算更大位数的数据

示例：大数乘法

- ◆ 提示：用一个数组的每个单元来存储每一位数字；然后组织循环，利用最基本的数学运算知识独立计算每位结果及其进位



用16个单元（16字节）的数组计算乘法
**循环16次，每次仅计算 $3 \times 2 = 6$

计算机组成与实现

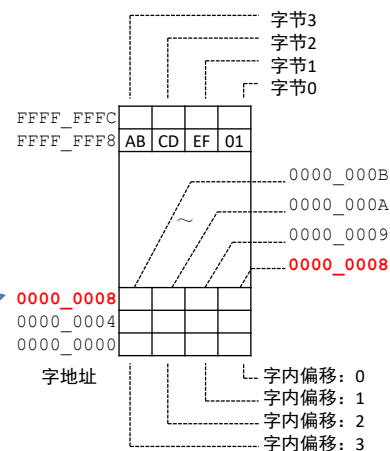
主存的抽象模型^{3/3}

为了方便从CPU的角度观察主存，经常视主存为2维数组模型

- ◆ 数组的每一行的单元数等于CPU字长
- ◆ 示例：4GB主存包含1G个字
 - $4GB \div (4B/W) = 1GW$

概念4：字地址

- ◆ 字地址是指某字第0列单元的字节地址
- ◆ 示例：字2的第0列单元的地址为 **0000_0008**，故字2的字地址为 **0000_0008**



计算机组成与实现

地址的表示方式

□ 方式1：绝对地址

- ◆ 主存单元的地址直接用具体数值表示
- ◆ 示例：直接给出黄色单元的地址 $P = \text{FFFF_FFF9}$

□ 方式2：相对地址

- ◆ 主存单元的地址采用“**基地址**+**偏移**”的表示方式
- ◆ 示例1：以 $B1$ 为基地址，以 1 为偏移，即 $P = B1 + 1$
- ◆ 示例2：以 $B2$ 为基地址，以 -2 为偏移，即 $P = B2 + (-2)$

	FFFF_FFFF
	FFFF_FFFE
	FFFF_FFFD
	FFFF_FFFC
B2 →	AB FFFF_FFFB
	CD FFFF_FFFA
P →	EF FFFF_FFF9
B1 →	01 FFFF_FFF8
	~
	0000_0003
	0000_0002
	0000_0001
	0000_0000
字节地址	

base address~基地址， offset~偏移

计算机组成与实现

“基地址+偏移”的优点

□ 一致：与数据结构的访问方式高度一致

- ◆ 示例1：数组单元 $A[5]$ 的地址 = **数组首地址** + 4×5
- ◆ 示例2：以 $B.z$ 为例，可以推算出 z 与 B 的起始地址的距离
 - 为了便于分配存储器，编译器会把 x 、 y 、 z 连续存储

```
int A[100] ;

struct {
    int    x ;
    short y ;
    char  z ;
} B ;
```

□ 统一：可以用某个固定base与不同的offset计算得到任意地址

□ 灵活：不同的{base,offset}组合可以对应同一个地址，为软件编程带来很大的灵活性

- ◆ 示例：可以从数组起始向末尾遍历（base为数组第0单元地址，offset为正），也可以从数组末尾向起始遍历（base为数组最后单元地址，offset为负）

计算机组成与实现

数据传输指令^{1/2}

□ 语法格式

op reg, off(base)

- ◆ reg: 写入或读出的寄存器
- ◆ base: 存储基地址的寄存器
 - 由于存储的是地址信息，因此base的值被作为无符号数
- ◆ off: 以字节为单位的偏移量
 - off是立即数，可正可负

□ 读写的存储单元的实际地址 = base + off

offset~偏移

37

计算机组成与实现

数据传输指令^{2/2}

□ 加载字: lw (Load Word)

- ◆ 读取地址为base+off的主存单元，然后写入reg

□ 存储字: sw (Store Word)

- ◆ 读取reg，然后写入地址为base+off的存储单元

□ 示例

```
int A[100] ;
A[10] = A[3] + a ;
```

假设: $A[] \leftrightarrow \$s3$, $a \leftrightarrow \$s0$

1	lw	\$t0, 12(\$s3)	# \$t0=A[3]
2	add	\$t0, \$s2, \$t0	# \$t0=A[3]+a
3	sw	\$t0, 40(\$s3)	# A[10]=A[3]+a

注意

lw/sw读写对象是字，因此偏移量必须是4的倍数

38

计算机组成与实现

主存单元使用的限制

❑ MIPS不支持主存单元参与运算

- ◆ 例如以下用法均是错误用法

```
add $t0, $s1, 0($s2)
```

```
sub $t0, 0($s2), 12
```

❑ MIPS支持的运算

- ◆ 寄存器—寄存器：寄存器与寄存器运算，结果写入寄存器
- ◆ 寄存器—立即数：寄存器与立即数运算，结果写入寄存器

注意

在MIPS中，主存单元仅能与寄存器进行数据交换。
之所以这样设计，是为了便于设计流水线CPU。

39

计算机组成与实现

寄存器 vs. 主存

❑ 变量比寄存器多怎么办？

- ◆ 把最常用的变量保存在寄存器中
- ◆ 其他不常用的变量保存在主存中

❑ 为什么不把变量都放在存储器中？

- ◆ 寄存器比存储器快100~500倍

40

计算机组成与实现

目录

- 程序执行的基本原理
- 指令格式及其操作数
- 指令集与汇编程序
 - ◆ 汇编程序基本结构
- 指令编码
- 汇编与反汇编实战

代码段与数据段

- 程序由指令和数据构成，分别存储在主存中的不同区域
 - ◆ 所有程序都有指令；绝大多数程序都会包含数据
 - ◆ 分离存储是为了防止两者之间相互干扰
- 代码段：存放程序中**指令序列**的一块内存区域
- 数据段：存储程序中**全局变量**的一块内存区域

	代码段	数据段
空间大小	由程序中的指令部分决定 程序运行前就已经确定	由程序中的全局变量部分决定 程序运行前就已经确定
读写权限	只读*(Read Only)	可读可写(Read/Write)
汇编关键字	.text	.data

*某些CPU允许代码段为可写，即允许程序自修改（self-modifying code）

汇编程序的基本结构

- 关键字 “.text” 和 “.data” 用来区分程序的数据部分和代码部分

<pre> 1 .data 2 str : .asciiz "1234+4321" 3 4 5 .text 6 lui \$s7, 0x1001 7 ori \$s7, \$s7, 0x0 </pre>	<div style="display: flex; align-items: flex-start;"> <div style="margin-right: 10px;"> <p>数据段</p> <p>定义一个全局变量</p> <p>正文段</p> </div> </div>
---	---

- “ .text ” 与 “ .data ” 本质上是代表基地址的标号
 - CPU的内存布局是确定的，即代码段、数据段在主存中的基地址是确定的
 - 根据数据段和代码段的基地址，汇编器能推算.data后面的各全局变量以及.text后面的各指令对应的相对偏移，也自然可以计算相应的绝对地址
- 示例：ori指令的偏移量是4

计算机组成与实现



北京航空航天大学计算机学院
School of Computer Science and Engineering, Beihang University

目录

- 程序执行的基本原理
- 指令格式及其操作数
- 指令集与汇编程序
 - 全局变量声明
- 指令编码
- 汇编与反汇编实战

全局变量声明的基本要素

- 全局变量位于 “.data” 关键字之后
- 要素1) 变量名
 - ◆ 可以包含字母、数字、下划线等；但不能有空格、+、-等符号
- 要素2) 关键字
 - ◆ 用于定义变量的类型
- 要素3) 值
 - ◆ 用于定义变量的值
- 示例：str是一个字符串变量，其值为 “1234+4321”

```
.data
str : .ascii "1234+4321"
```

计算机组成与实现

常见变量类型^{1/2}

- 与C语言不同，汇编语言关注的是变量的存储类型，即一个变量需要占用多少字节
- 汇编语言的变量类型都是无符号的

关键字	基本用途
.byte	声明8位变量
.half	声明16位变量
.word	声明32位变量
.ascii	声明字符串。字符串结尾无'\0'
.asciiz	声明字符串。字符串结尾有'\0'
.space	为一个变量预留指定的字节数

计算机组成与实现

常见变量类型^{2/2}

□ 注意汇编代码与C代码的区别

- ◆ 例如half，汇编代码中只知道占据2个字节；C代码无论是short还是unsigned short，都与之对应
- ◆ 例如数组array，汇编中只定义了100个单元，但C语言还能明确其是char而不是unsigned char类型

汇编代码	C代码
1 .data	
2 str : .asciiz "1234+4321"	char str[]="1234+432" ;
3 half : .space 2	short half ;
4 i : .word 0xAABBCCDD	int i=0xAABBCCDD ;
5 array : .space 100	char array[100] ;
6	
7 .text	void main(void)
8

计算机组成与实现

目录

- 程序执行的基本原理
- 指令格式及其操作数
- 指令集与汇编程序
 - ◆ 读/写存储器
- 指令编码
- 汇编与反汇编实战

lb: 读字节^{1/3}

- lb格式: lb rt, offset(base)
- lb功能: 从以编号为base的寄存器为基地址、offset为偏移的地址单元处, 读取一个字节, 然后将其写入编号为rt的寄存器
- 以lb指令功能为例, 可以看出自然语言描述方式的缺陷
 - ◆ 1) 不精确, 易引起二义性
 - 例如, 读出的是1个字节, 要写入的寄存器是4个字节, 两者是何种对应关系?
 - ◆ 2) 不简练
 - 如果希望表达的无二义, 则自然语言表达方式往往会非常繁琐
- RTL是一种形式化方式描述指令功能的语言
 - ◆ RTL: Register Transfer Language
 - ◆ 它用于描述从寄存器到寄存器之间的行为

注意

所有读/写存储器指令的offset都是以字节为单位的

计算机组成与实现

lb: 读字节^{2/3}

- lb格式: lb rt, offset(base)
- RTL描述的lb功能

$$R[rt] \leftarrow M[R[base] + offset]$$

- ◆ R[]: 对应MIPS的32个寄存器
 - 这些寄存器有时也被称为通用寄存器 (General Purpose Register)
 - 示例: R[base]代表标号为base的寄存器
- ◆ M[]: 对应主存
 - 示例: M[5]代表地址为5的主存单元

注意

lb的功能是读取字节, 但上述描述并未明确表达这一点
这就必须进一步明确描述lb的具体操作细节

计算机组成与实现

lb: 读字节^{3/3}

□ RTL描述lb的更多功能细节

```
Addr ← R[base] + sign_ext(offset)
memword ← M[Addr]
byte ← Addr1..0
R[rt] ← sign_ext(memword7+8*byte..8*byte)
```

- ◆ sign_ext(): 代表符号扩展
- ◆ 第1步: 计算主存单元的地址
- ◆ 第2步: 从主存中读取的一个字
- ◆ 第3步: 计算字内正确的字节偏移
- ◆ 第4步: 将字节写入寄存器

注意
第2步之所以读取的是字而不是字节，是使得CPU与主存间始终以32位宽度交换数据，从而简化数据传输方式

□ 相对自然语言描述方式，RTL描述具有准确、无二义的特点

计算机组成与实现

sb: 写字节

□ sb的功能与lb恰好相反，用于将寄存器的最低字节写入主存单元

□ sb格式: sb rt, offset(base)

□ sb的RTL描述

```
M[R[base]+offset] ← R[rt]
```

□ sb的RTL细节描述

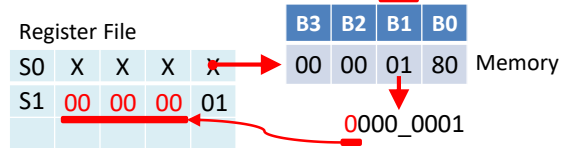
```
Addr ← R[base] + sign_ext(offset)
byte ← Addr1..0
M[Addr]7+8*byte..8*byte ← R[rt]7..0
```

计算机组成与实现

lb/sb的用法^{1/3}

□ 示例：假设*(\$s0) = 0x00000180

lb \$s1, 1(\$s0) # \$s1=0x00000001

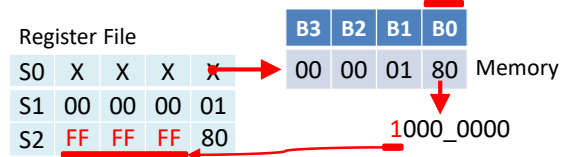


lb/sb的用法^{2/3}

□ 示例：假设*(\$s0) = 0x00000180

lb \$s1, 1(\$s0) # \$s1=0x00000001

lb \$s2, 0(\$s0) # \$s2=0xFFFFF80



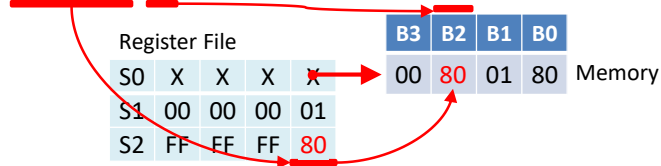
lb/sb的用法^{3/3}

- 示例：假设 $*(\$s0) = 0x00000180$

lb \$s1, 1(\$s0) # \$s1=0x00000001

lb \$s2, 0(\$s0) # \$s2=0xFFFFFFFF80

sb \$s2, 2(\$s0) # $*(\$s0) = 0x00800180$



计算机组成与实现

加载和存储指令汇总

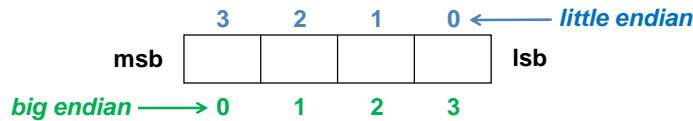
- 字操作：偏移必须是4的倍数
 - ◆ lw、sw
- 半字操作：偏移必须是2的倍数
 - ◆ lh、lhu
 - ◆ sh
- 字节操作
 - ◆ lb、lbu
 - ◆ sb

注意

lbu/lhu：没有符号扩展

大/小印第安

- 大印第安：最高有效字节在字内的最低地址
- 小印第安：最高有效字节在字内的最高地址



- MIPS：同时支持2种类型
 - ◆ 本课程用小印第安

endianness~字节顺序

57

计算机组成与实现

目录

- 程序执行的基本原理
- 指令格式及其操作数
- 指令集与汇编程序
 - ◆ 算术运算
- 指令编码
- 汇编与反汇编实战

算术运算指令概述

- 计算机最重要的功能就是完成加、减、乘及除等算术运算
- 几乎所有的高级程序设计都支持这些算术运算
- MIPS为此设置相应的算术运算指令
- 除从运算功能角度分类，还可从操作数角度分类
 - ◆ 寄存器—寄存器型（R-R）：参与运算的2个操作数都是寄存器，结果写入寄存器
 - ◆ 寄存器—立即数型（R-I）：参与运算的2个操作数一个是寄存器，另一个是立即数，结果写入寄存器

计算机组成与实现

加法和减法

- 加法有4条指令：add, addu, addi, addiu
- 减法有2条指令：sub, subu

指令	功能	格式	描述	示例
add	加法 (检测溢出)	add rd, rs, rt	$R[rd] \leftarrow R[rs] + R[rt]$	add \$s1, \$s2, \$s3
addu	加法 (不检测溢出)	add rd, rs, rt	$R[rd] \leftarrow R[rs] + R[rt]$	addu \$s1, \$s2, \$s3
addi	立即数加 (检测溢出)	addi rt, rs, imm16	$R[rt] \leftarrow R[rs] + \text{sign_ext}(\text{imm16})$	addi \$s1, \$s2, -3
addiu	立即数加 (不检测溢出)	addiu rt, rs, imm16	$R[rt] \leftarrow R[rs] + \text{sign_ext}(\text{imm16})$	addiu \$s1, \$s2, 3
sub	减法 (检测溢出)	sub rd, rs, rt	$R[rd] \leftarrow R[rs] - R[rt]$	sub \$s1, \$s2, \$s3
subu	减法 (不检测溢出)	sub rd, rs, rt	$R[rd] \leftarrow R[rs] - R[rt]$	subu \$s1, \$s2, \$s3

问题
为什么没有subi和subiu？

计算机组成与实现

算术溢出^{1/2}


- 复习：当计算结果的位数超出计算机硬件实际能保存的位数，即为**溢出**
 - ◆ 换言之，即没有足够的位数保存运算结果
- MIPS会检测溢出（并且当溢出发生时**产生错误**）
 - ◆ 有unsigned关键字的算术类指令忽略溢出

检测溢出	不检测溢出
add dst,src1,src2	addu dst,src1,src2
addi dst,src1,src2	addiu dst,src1,src2
sub dst,src1,src2	subu dst,src1,src2

61

算术溢出^{1/2}

- 示例


 复习：这是最小的负数！

```

# $s0=0x80000000, $s1=0x1
add    $t0,$s0,$s0 # 溢出（出错）
addu   $t1,$s0,$s0 # $t1=0
addi   $t2,$s0,-1  # 溢出（出错）
addiu  $t3,$s0,-1  # $t3=0x7FFFFFFF
sub     $t4,$s0,$s1 # 溢出（出错）
subu   $t5,$s0,$s1 # $t5=0x7FFFFFFF
  
```

62

计算机组成与实现

乘除法指令

- 乘除法指令计算结果：不是直接写入32个通用寄存器，而是保存在2个特殊寄存器HI与LO
- 用2条专用指令读写HI/LO
 - “move from HI” (mfhi dst)
 - “move from LO” (mflo dst)
- **Multiplication** (mult)
 - ◆ mult src1,src2
 - ◆ src1*src2: LO保存结果的低32位, HI保存结果的高32位
- **Division** (div)
 - ◆ div src1,src2
 - ◆ src1/src2: LO保存商, HI保存余数

quotient~商; remainder~余数

63

计算机组成与实现

乘除法指令

- 示例：用div求模

```
# $s2 = $s0 mod $s1
```

```
mod:
```

```
div $s0,$s1    # LO = $s0/$s1
```

```
mfhi $s2       # HI = $s0 mod $s1
```

64

计算机组成与实现

目录

- 程序执行的基本原理
- 指令格式及其操作数
- 指令集与汇编程序
 - ◆ 逻辑运算（位运算）
- 指令编码
- 汇编与反汇编实战

位运算指令

- 假设： $a \rightarrow \$s1, b \rightarrow \$s2, c \rightarrow \$s3$

指令	C	MIPS
与	$a = b \ \& \ c;$	<code>and \$s1,\$s2,\$s3</code>
与立即数	$a = b \ \& \ 0x1;$	<code>andi \$s1,\$s2,0x1</code>
或	$a = b \ \ c;$	<code>or \$s1,\$s2,\$s3</code>
或立即数	$a = b \ \ 0x5;$	<code>ori \$s1,\$s2,0x5</code>
或非	$a = \sim(b \ \ c);$	<code>nor \$s1,\$s2,\$s3</code>
异或	$a = b \ \wedge \ c;$	<code>xor \$s1,\$s2,\$s3</code>
异或立即数	$a = b \ \wedge \ 0xF;$	<code>xori \$s1,\$s2,0xF</code>

目录

- 程序执行的基本原理
- 指令格式及其操作数
- 指令集与汇编程序
 - ◆ 分支指令
- 指令编码
- 汇编与反汇编实战

基本的决策机制

- C：有if-else, for, while, do-while等语句块
 - ◆ 决策机制：根据条件转移并执行相应的语句块
- MIPS：通过标号机制来实现转移
 - ◆ MIPS没有语句块的概念，只有地址的概念
 - ◆ 每条指令都对应一个word地址
 - ◆ 为了提高可读性，汇编程序使用标号来标记其后的指令的地址
 - 标号是由字符串+':'组成。例如：ForBegin:
 - ◆ 汇编语言再通过跳转机制跳转到标号处，从而实现转移

注意

C也有类似的机制，例如goto，但被认为是不好的编程风格

分支指令

- **Branch If Equal (beq): 相等时转移**
 - ◆ `beq reg1, reg2, label`
 - ◆ 如果`reg1`的值=`reg2`的值, 则转移至`label`处执行
- **Branch If Not Equal (bne): 不等时转移**
 - ◆ `bne reg1, reg2, label`
 - ◆ 如果`reg1`的值 \neq `reg2`的值, 则转移至`label`处执行
- **Jump (j): 无条件转移**
 - ◆ `j label`
 - ◆ 无条件转移至`label`处执行
 - ◆ 对应C的goto机制

69

计算机组成与实现

用beq构造if-else

- 与C不同之处: b类指令是条件为**TRUE**则**转移**, **FALSE**则**顺序**!

C代码:

```
if (i==j) {
    a = b /* then */
} else {
    a = -b /* else */
}
```

执行逻辑

- 如果TRUE, 执行**THEN**语句块
- 如果FALSE, 执行**ELSE**语句块

MIPS (beq):

```
# i→$s0, j→$s1
# a→$s2, b→$s3

beq $s0, $s1, ???
??? ← 可以去除该标号
sub $s2, $0, $s3
j    end
then:
add $s2, $s3, $0
end:
```

70

计算机组成与实现

用bne构造if-else

- 与C不同之处：beq/bne构造if-else是**条件为TRUE则转移**！

C代码：

```
if(i==j) {
    a = b /* then */
} else {
    a = -b /* else */
}
```

MIPS (bne):

```
# i→$s0, j→$s1
# a→$s2, b→$s3

bne $s0,$s1,???,
???,
add $s2, $s3, $0
j    end
else:
sub $s2, $0, $s3
end:
```

执行逻辑

- 如果TRUE, 执行THEN语句块
- 如果FALSE, 执行ELSE语句块

71

计算机组成与实现

switch-case

- 大多数高级程序设计语言具有switch-case语句结构
- 在MIPS汇编程序中，可以用一组b类指令来构造switch-case

```
1 switch ()
2     case 条件1
3         语句块1
4     case 条件2
5         语句块2
6     ...
7     case 条件N
8         语句块N
9
10
11
12
13
14
15
16
17
```

```
b类, $条件1, Case1
b类, $条件2, Case2
...
b类, $条件N, CaseN
j SwitchEnd

Case1:
    语句块1
    j SwitchEnd
Case2:
    语句块2
    j SwitchEnd
...
CaseN:
    语句块N
    j SwitchEnd

SwitchEnd :
```

注意1
每个case后面都需要一条j指令

注意2
该MIPS汇编框架没有考虑default情况

计算机组成与实现

更多分支指令

- 与0比较的分支指令（简称**bxxz**类指令）
 - ◆ blez（小于等于0转移）
 - ◆ bgtz（大于0转移）
 - ◆ bltz（小于0转移）
 - ◆ bgez（大于等于0转移）
- **bxxz**类指令格式均为：**bxxz** rs, label
 - ◆ 由于**bxxz**指令固定与\$0比较，因此指令中无需再描述\$0了

计算机组成与实现

循环

- C语言有3种循环：for, while, do...while
 - ◆ 3种语句是等价的，即任意一种循环都可以改写为其他两种循环
- MIPS只需要一种决策机制即可
 - ◆ 核心：根据条件转移

循环实战：从C到MIPS^{1/5}

□ 实例：字符串赋值

□ C代码

```
/* 字符串复制 */
char *p, *q;

while ((*q++ = *p++) != '\0') ;
```

□ 代码结构有什么特征

- ◆ 单一的while循环
- ◆ 退出循环是一个相等测试

75

计算机组成与实现

循环实战：从C到MIPS^{2/5}

□ 实例：字符串赋值

□ C代码

```
/* 字符串复制 */
char *p, *q;

while ((*q++ = *p++) != '\0') ;
```

□ 代码结构有什么特征？

- ◆ 单一的while循环
- ◆ 退出循环是一个相等测试

Q: 2种写法的区别在哪里？

```
while ( *p )
    *q++ = *p++ ;
```

循环实战：从C到MIPS^{3/5}

□ STEP1: 构造循环的框架

字符串复制

p→\$s0, q→\$s1 (p和q都是指针)

```

Loop:                                # $t0 = *p
                                       # *q = $t0
                                       # p = p + 1
                                       # q = q + 1
                                       # if *p==0, go to Exit
                                       # go to Loop
        j Loop
Exit:

```

77

计算机组成与实现

循环实战：从C到MIPS^{4/5}

□ STEP2: 构造循环主体

字符串复制

p→\$s0, q→\$s1 (p和q都是指针)

```

Loop: lb    $t0,0($s0)    # $t0 = *p
        sb    $t0,0($s1)    # *q = $t0
        addi $s0,$s0,1      # p++
        addi $s1,$s1,1      # q++
        beq  $t0,$0,Exit    # if $t0==0, go to Exit
        j    Loop          # go to Loop
Exit:

```

Q

1个字符需要6条指令。能否优化？

78

循环实战：从C到MIPS^{5/5}

- 优化代码（减少了1条指令）

字符串复制

$p \rightarrow \$s0$, $q \rightarrow \$s1$ (p 和 q 都是指针)

```
Loop: lb    $t0, 0($s0)    # $t0 = *p
      sb    $t0, 0($s1)    # *q = $t0
      addi  $s0, $s0, 1    # p = p + 1
      addi  $s1, $s1, 1    # q = q + 1
      bne   $t0, $0, Loop  # if *p!=0, go to Loop
```

79

计算机组成与实现

不等式

- 不等关系：<, <=, >, >=
 - ◆ MIPS：用增加1条额外的指令来支持所有的比较
- **Set on Less Than (slt)**
 - ◆ `slt dst, src1, src2`
 - ◆ 如果 $src1 < src2$, `dst` 写入1, 否则写入0
- 与 `bne`, `beq`, `$0` 组合即可实现所有的比较

80

计算机组成与实现

不等式

□ 实现<

C代码	MIPS汇编
<pre>if (a < b) { ... /* then */ }</pre> <p>(假设: $a \rightarrow \\$s0, b \rightarrow \\$s1$)</p>	<pre>slt \$t0, \$s0, \$s1 # \$t0=1 if a<b # \$t0=0 if a>=b bne \$t0, \$0, then # go to then # if \$t0≠0</pre>

81

计算机组成与实现

不等式

□ 实现>=

C代码	MIPS汇编
<pre>if (a >= b) { ... /* then */ }</pre> <p>(假设: $a \rightarrow \\$s0, b \rightarrow \\$s1$)</p>	<pre>slt \$t0, \$s0, \$s1 # \$t0=1 if a<b # \$t0=0 if a>=b beq \$t0, \$0, then # go to then # if \$t0=0</pre>

□ Q: 请自行完成

- ◆ 交换src1与src2
- ◆ 分别用beq与bne

82

计算机组成与实现

不等式

- `slt`的3种变形
 - ◆ `sltu dst,src1,src2`: 无符号数比较
 - ◆ `slti dst,src,imm`: 与常量比较
 - ◆ `sltiu dst,src,imm`: 与无符号常量比较

□ 示例:

```
addi $s0,$0,-1 # $s0=0xFFFFFFFF
slti $t0,$s0,1 # $t0=1
sltiu $t1,$s0,1 # $t1=0
```

不等式

- 用`slt`实现`<`, `>`, `<=`, `>=`
 - ◆ `a<b`: 直接运用`slt`指令即可
 - ◆ `a>b`: 由于`a>b`与`b<a`完全是相同的, 因此可以改为用`slt`判断`b<a`
 - ◆ `a≤b`: 由于`a≤b`与 $\overline{b < a}$ 是等价的, 因此将`slt`判断`b<a`的结果取反即可
 - “`slt 结果, b, a`”执行结果为0, 则原条件为真
 - ◆ `a≥b`: 方法同上

原条件	等价条件	指令用法	结果寄存器的0/1值含义
<code>a < b</code>		<code>slt 结果寄存器, a, b</code>	0: 原条件为假 1: 原条件为真
<code>a > b</code>	<code>b < a</code>	<code>slt 结果寄存器, b, a</code>	0: 原条件为假 1: 原条件为真
<code>a <= b</code>	$\overline{b < a}$	<code>slt 结果寄存器, b, a</code>	0: 原条件为真 1: 原条件为假
<code>a >= b</code>	$\overline{a < b}$	<code>slt 结果寄存器, a, b</code>	0: 原条件为真 1: 原条件为假

[AD]MIPS的Signed与Unsigned

- 术语Signed与Unsigned有3种不同含义
 - ◆ 符号位扩展
 - lb: 扩展
 - lbu: 无扩展
 - ◆ 溢出
 - add, addi, sub, mult, div: 检测
 - addu, addiu, subu, multu, divu: 不检测
 - ◆ 符号数
 - slt, slti: 符号数
 - sltu, sltiu: 无符号数

85

计算机组成与实现

小测试

- 根据汇编程序，请选择正确的C语句填入C代码的空白处

```
do {i--;} while(_____);
```

```

Loop:           # i→$s0, j→$s1
addi $s0,$s0,-1 # i = i - 1
slti $t0,$s1,2  # $t0 = (j < 2)
beq  $t0,$0 ,Loop # goto Loop if $t0==0
slt  $t0,$s1,$s0 # $t0 = (j < i)
bne  $t0,$0 ,Loop # goto Loop if $t0!=0
  
```

1 j ≥ 2 || j < i

2 j ≥ 2 && j < i

3 j < 2 || j ≥ i

4 j < 2 && j ≥ i

计算机组成与实现

目录

- 程序执行的基本原理
- 指令格式及其操作数
- 指令集与汇编程序
 - ◆ 伪指令
- 指令编码
- 汇编与反汇编实战

伪指令

- 很多C语句对应到MIPS指令时很不直观
 - ◆ 例如：C的赋值语句a=b，会用某条运算指令实现，如addi
- MIPS定义了一组 伪指令，从而使得程序更可读更易编写
 - ◆ 伪指令不是真正的指令
 - ◆ 伪指令只是增加了可读性
 - ◆ 伪指令要被转换为实际指令
- 示例

```

move dst,src    转换为
addi dst,src,0
  
```

常用伪指令

- ▣ **Move**
 - ◆ `move dst, src`
 - ◆ 把src赋值给dst
- ▣ **Load Address (la)**
 - ◆ `la dst, label`
 - ◆ 加载特定标号对应的地址至dst
- ▣ **Load Immediate (li)**
 - ◆ `li dst, imm`
 - ◆ 加载一个32位立即数至dst

89

计算机组成与实现

汇编寄存器

- ▣ **问题**
 - ◆ 汇编器把一条伪指令转换为真实指令时，可能需要多条真实指令
 - ◆ 这组真实指令之间就必须通过某个寄存器来传递信息
 - ◆ 如果任意使用某个寄存器，则存在这个寄存器被汇编器误写的可能
- ▣ **解决方案**
 - ◆ 保留\$1 (\$at) 作为汇编器专用寄存器
 - ◆ 由于汇编器会使用这个寄存器，因此从代码安全角度，其他代码不应再使用这个寄存器

编号	名称	用途
0	\$zero	常量0
1	\$at	汇编器保留
2-3		
4-7		
8-15	\$t0-\$t7	临时变量
16-23	\$s0-\$s7	程序变量
24-25	\$t8-\$t9	临时变量
28		
29		
30		
31		

90

计算机组成与实现

MAL vs. TAL

- True Assembly Language (TAL)
 - ◆ 真实指令，是计算机能够理解和执行的
- MIPS Assembly Language (MAL)
 - ◆ 提供给汇编程序员使用的指令（包含伪指令）
 - ◆ 每条MAL指令对应1条或多条TAL指令
 - 主要是针对伪指令
- $TAL \subset MAL$

91

计算机组成与实现



北京航空航天大学计算机学院
School of Computer Science and Engineering, Beihang University

目录

- 程序执行的基本原理
- 指令格式及其操作数
- 指令集与汇编程序
 - ◆ 移位运算
- 指令编码
- 汇编与反汇编实战

移位指令^{1/3}

- C语言有移位操作，MIPS也定义了多条移位指令
- 移位指令可以从3个维度来分析
 - ◆ 方向：左移还是向右移
 - ◆ 性质：逻辑移位还是算术移位
 - 对于向左移位来说，低位永远是补0
 - 只有向右移位，才存在高位是补0还是符号位的选择问题。如果补0，那就是逻辑移位，如果是补符号位，则为算术移位。
 - ◆ 移位量：对于32位寄存器，移动位数的合理最大取值为31，即0x1F
 - 如何在指令中表示这个移位量呢？
 - 方式1：由一个5位的立即数来表示移位量
 - 方式2：用某寄存器的值来表示移位量。如果用寄存器来表示移位量，则只有该寄存器的最低5位被CPU识别为移位量，而高27位无论取何值均无意义。

问题

根据上述3个维度，应该定义几条指令？

计算机组成与实现

移位指令^{2/3}

- MIPS共6条移位指令
 - ◆ 如果使用立即数：只有0~31有效
 - ◆ 如果使用寄存器：寄存器的低5位有效（按5位无符号数对待）

指令	功能	示例
sll	逻辑左移	sll \$t0, \$s0, 16
srl	逻辑右移	srl \$t0, \$s0, 16
sra	算术右移	sra \$t0, \$s0, 16
sllv	逻辑可变左移	sllv \$t0, \$s0, \$s1
srlv	逻辑可变右移	srlv \$t0, \$s0, \$s1
srav	算术可变右移	srav \$t0, \$s0, \$s1

移位指令^{3/3}

□ 示例

```
addi $t0,$0,-256 # $t0=0xFFFFFFFF00
sll  $s0,$t0,3   # $s0=0xFFFFFFFF800
srl  $s1,$t0,8   # $s1=0x00FFFFFFF
sra  $s2,$t0,8   # $s2=0xFFFFFFFFF
```

```
addi $t1,$0,-22  # $t1=0xFFFFFEEA
                        # low 5: 0b01010
sllv $s3,$t0,$t1 # $s3=0xFFFC0000
```

95

计算机组成与实现



北京航空航天大学计算机学院
School of Computer Science and Engineering, Beihang University

目录

- 程序执行的基本原理
- 指令格式及其操作数
- 指令集与汇编程序
 - ◆ 寄存器加载立即数高位
- 指令编码
- 汇编与反汇编实战

如何计算32位立即数？

- 32位立即数的应用场景，如：
 - ◆ `addi/slti/andi/ori`：等需要计算2个32位数
 - ◆ `lw/sw`：在使用前，需要先设置基地址寄存器的值（32位）
- 解决方案：不改变指令格式，而是增加一条新指令
- **Load Upper Immediate** (`lui`)
 - ◆ `lui reg,imm`
 - ◆ `reg`的高16位写入`imm`，低16位写入0
 - RTL: $R[reg] \leftarrow imm \parallel 0^{16}$

97

计算机组成与实现

lui示例

- 需求：`addi $t0,$t0,0xABABCD`



◆ 这是一条伪指令！

- 会被assembler转换为3条指令

```
lui $at,0xABAB      # 高16位
ori $at,$at,0xCD CD # 低16位
add $t0,$t0,$at     # 赋值
```

T
手工编写汇编时，
尽量不适\$at；只
应由assembler使
用\$at

- 通过增加`lui`，MIPS可以用16位立即数来处理任意大小的数据

98

计算机组成与实现

目录

- 程序执行的基本原理
- 指令格式及其操作数
- 指令集与汇编程序
 - ◆ 函数
- 指令编码
- 汇编与反汇编实战

实现函数的6个步骤

- 1、调用者把参数放置在某个地方以便函数能访问
- 2、调用者转移控制给被调用的函数
- 3、函数获取局部变量对应的空间
- 4、函数执行具体功能
- 5、函数把返回值放置在某个地方，然后恢复使用的资源
- 6、返回控制给调用者

函数相关寄存器

- 由于寄存器比主存快，因此尽可能的用寄存器
- `$a0-$a3`：4个传递参数的寄存器
- `$v0-$v1`：2个传递返回值的寄存器
- `$ra`：返回地址寄存器，保存着调用者的地址

编号	名称	用途
0	\$zero	常量0
1	\$at	汇编器保留
2-3	<code>\$v0-\$v1</code>	返回值
4-7	<code>\$a0-\$a3</code>	参数
8-15	\$t0-\$t7	临时变量
16-23	\$s0-\$s7	程序变量
24-25	\$t8-\$t9	临时变量
28		
29		
30		
31	<code>\$ra</code>	返回地址

101

计算机组成与实现

函数调用指令

- **Jump and Link** (`jal`)
 - ◆ `jal label`
 - ◆ 把jal的下一条指令的地址保存在`$ra`，然后跳转到`label` (即函数地址)
 - ◆ `jal`的用途是调用函数
- **Jump Register** (`jr`)
 - ◆ `jr src`
 - ◆ 无条件跳转到保存在`src`的地址 (通常就是`$ra`)
 - ◆ `jr`的用途是从函数返回

102

计算机组成与实现

PC

- PC (program counter) 是一个特殊寄存器，用于保存当前正在指令的地址
 - ◆ PC是冯式体系结构计算机的关键环节
 - ◆ 在MIPS中，PC对于程序员不可见，但可以被jal访问
- 注意：保存在\$ra的是PC+4，而不是PC
 - ◆ 否则当函数返回的时候，就会再次返回到jal本身了

函数调用示例

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

```

			C
	1000	addi \$a0,\$s0,0	# x = a
	1004	addi \$a1,\$s1,0	# y = b
	1008	jal sum	# \$ra=1012, goto sum
地	1012		
址	...		
	2000	sum: add \$v0,\$a0,\$a1	
	2004	jr \$ra	# return

实现函数的6个步骤

- 1、调用者把参数放置在某个地方以便函数能访问
 - ◆ \$a0~\$a3
- 2、调用者转移控制给被调用的函数
 - ◆ jal
- 3、函数获取局部变量对应的空间??
- 4、函数执行具体功能
- 5、函数把返回值放置在某个地方，然后恢复使用的资源
 - ◆ \$v0~\$v1
- 6、返回控制给调用者

105

计算机组成与实现

保存和恢复寄存器

- Q：为什么需要保存寄存器？
 - ◆ 理由1：寄存器数量太少，不可能只用寄存器就能编写实用程序
 - ◆ 理由2：如果被调用函数继续调用函数，会发生什么？
(`$ra` 被覆盖了！)
- Q：寄存器保存在什么地方？
 - ◆ 栈！
- `$sp`：栈指针寄存器。指针指向栈底

编号	名称	用途
0	\$zero	常量0
1	\$at	汇编器保留
2-3	\$v0~\$v1	返回值
4-7	\$a0~\$a4	参数
8-15	\$t0~\$t7	临时变量
16-23	\$s0~\$s7	程序变量
24-25	\$t8~\$t9	临时变量
28		
29	\$sp	栈指针
30		
31	\$ra	返回地址

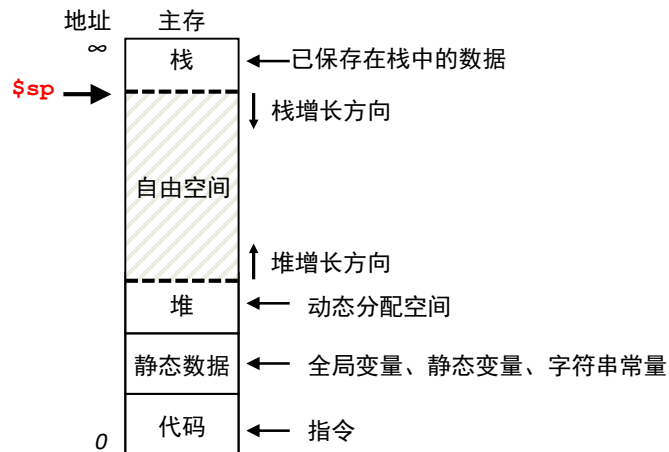
stack~栈

106

计算机组成与实现

主存分配的基本方案

- 栈帧：函数为获得保存寄存器而将\$sp向0方向调整的空间
 - ◆ 栈帧容量 = 要保存的寄存器数量 × 4



stack frame/栈帧

107

计算机组成与实现

函数示例

```
int sumSquare(int x, int y) {
    return mult(x, x) + y; }
```

- 都需要保存哪些寄存器？
 - ◆ 1) \$ra: 由于sumSquare要调用mult, 因此\$ra会被覆盖
 - ◆ 2) \$a1: 给sumSquare传递y, 但还给mult传递x
 - 关键是在sumSquare中, 会先传递x然后才引用y, 因此y就被x覆盖了
- 为了保存这2个寄存器, 首先需要将\$sp 向下移动8个字节
 - ◆ 栈帧容量为8字节: 要保存2个寄存器, 共需要8个字节

108

计算机组成与实现

函数示例

```
int sumSquare(int x, int y) {
    return mult(x,x)+ y; }
```

sumSquare:

push	{	addi \$sp,\$sp,-8	# make space on stack
		sw \$ra, 4(\$sp)	# save ret addr
		sw \$a1, 0(\$sp)	# save y
		move \$a1,\$a0	# set 2 nd mult arg
		jal mult	# call mult
pop	{	lw \$a1, 0(\$sp)	# restore y
		add \$v0,\$v0,\$a1	# ret val = mult(x,x)+y
		lw \$ra, 4(\$sp)	# restore ret addr
		addi \$sp,\$sp,8	# restore stack
		jr \$ra	

mult: ...

109

计算机组成与实现

函数的架构

□ $\text{framesize} = \text{要保存的寄存器个数} \times 4$

Prologue

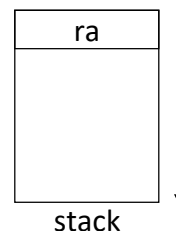
```
func_label:
addi $sp,$sp, -framesize
sw $ra, [framesize-4]($sp)
save other regs if need be
```

Body (可以调用其他函数...)

...

Epilogue

```
restore other regs if need be
lw $ra, [framesize-4]($sp)
addi $sp,$sp, framesize
jr $ra
```



110

计算机组成与实现

局部变量和数组

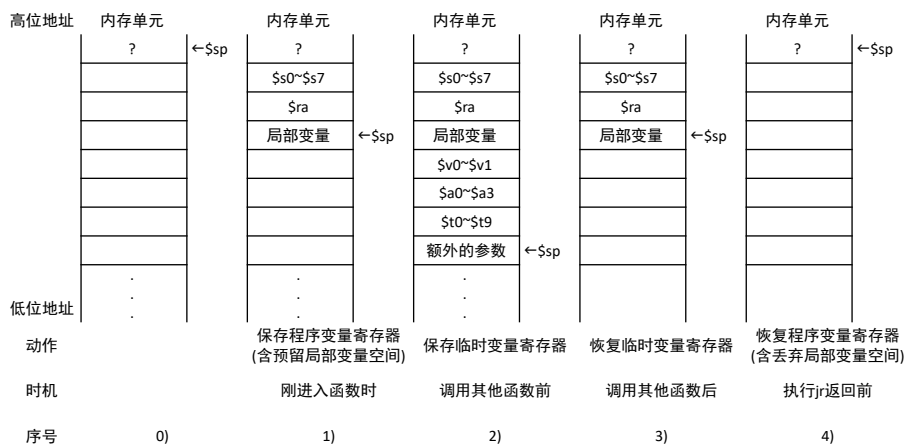
- ❑ 由于寄存器只有32个，因此编译器绝大多数情况下不可能把函数需要的所有局部变量都分配在寄存器
- ❑ 局部变量存放在函数自己的栈帧中
 - ◆ 这样当函数返回时，随着\$sp的回调，局部变量自然就被释放了
- ❑ 其他局部变量，如数组、结构等，同样也是存储在栈帧中
- ❑ 为局部变量分配空间的方法是与保存寄存器是完全相同
 - ◆ 将\$sp向下调整，产生的空间用于存储局部变量

111

计算机组成与实现

局部变量和数组

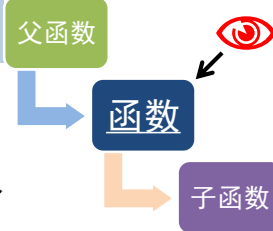
- 由于寄存器只有32个，因此编译器绝大多数情况下不可能把函数需要的所有局部变量都分配在寄存器



112

计算机组成与实现

寄存器的保护分析



- $\$s0 \sim \$s7$: 程序员变量
 - ◆ 分析: 所有函数都要使用程序员变量
 - 程序员变量的生命周期与函数生命周期相同, 即进入函数就有效, 退出函数后无效
 - ◆ 保护前提: 用哪些, 保护哪些
 - ◆ 保护动作: 刚进入函数时保存; 退出函数前恢复
- $\$ra$: 函数返回地址
 - ◆ 保护前提: 如果继续调用子函数, 那么就必须保护
 - 否则函数就不能返回至父函数
 - ◆ 保护动作: 刚进入函数时保存; 退出函数前恢复

113

计算机组成与实现

寄存器的保护分析

- $\$t0 \sim \$t9$: 临时变量 (前提: MIPS约定其服务于表达式计算)
 - ◆ 无需保护: 表达式中不调用子函数, 故不存在被子函数修改的可能


```
e = a + b + c + d ; // $t0 = a + b, $t1 = c + d
f1() ;
```
 - ◆ 需要保护: 表达式中调用子函数, 故存在被子函数修改的可能


```
z = x + y + f2() ; // $t0 = x + y
                  // $t0有可能被f2()修改
```
 - ◆ 保护前提: 如果其值在子函数调用前后必须保持一致
 - ◆ 保护动作: 调用子函数前保存; 在调用子函数后恢复

114

计算机组成与实现

寄存器的保护分析

- \$a0~\$a3, \$v0~\$v1: 参数, 返回值
 - ◆ \$a0~\$a3: 传递给了子函数, 故子函数可自由使用
 - ◆ \$v0~\$v1: 因为子函数会设置返回值, 故子函数可自由使用
 - ◆ 保护前提: 如果其值在调用子函数前后必须保持一致, 则:
 - ◆ 保护动作: 调用子函数前保存; 在调用子函数后恢复
- \$at: 汇编器使用, 故无需保护
- \$k0~\$k1: 操作系统使用 (现阶段可以不使用)
- \$sp: 栈切换时才需要保护 (现阶段可以不使用)
 - ◆ 栈切换: 通常属于操作系统范畴
- \$fp, \$gp: 在生成复杂的存储布局时使用 (属于编译范畴)

115

计算机组成与实现

寄存器的保护分析

序号	名称	MIPS汇编约定的用途	分类	保护的前提条件	何时保护与恢复
16~23	\$s0~\$s7	程序员变量	强保护	如果需要使用	进入函数时保存 退出函数前恢复
31	\$ra	函数返回地址		如果调用子函数	
2~3	\$v0~\$v1	函数返回值	弱保护	如果要求其值在调用子函数前后必须保持一致	调用子函数前保存 子函数返回后恢复
4~7	\$a0~\$a3	函数参数			
8~15, 24~25	\$t0~\$t9	临时变量			

- Saved Register: \$s0~\$s7, \$ra
 - ◆ 英文: preserved register; 中文: 保护寄存器, 保存寄存器
- Volatile Register: \$t0~\$t9, \$a0~\$a3, \$v0~\$v1
 - ◆ 英文: non-preserved register; 中文: 非保护寄存器, 非保存寄存器

116

计算机组成与实现

强保护寄存器的保护机制代码框架

- 进入函数后，立刻调整栈并保存；在函数返回时才恢复

<pre> 1 函数_label : 2 addiu \$sp, \$sp, -framesize 3 sw \$ra, [framesize-4](\$sp) 4 sw \$s0, [framesize-8](\$sp) 5 sw \$s1, [framesize-12](\$sp) 6 # ... 7 sw \$s7, 0(\$sp) 8 9 # 使用\$s0~\$s7 10 # 调用其他子函数 11 12 lw \$s7, 0(\$sp) 13 # ... 14 lw \$s1, [framesize-12](\$sp) 15 lw \$s0, [framesize-8](\$sp) 16 lw \$ra, [framesize-4](\$sp) 17 addiu \$sp, \$sp, framesize 18 jr \$ra </pre>	<div>分配栈帧 保存\$ra 保存\$s0~\$s7中后续要使用的寄存器</div> <div>恢复函数保护的寄存器</div> <div>回收栈帧</div>
--	--

117

计算机组成与实现

弱保护寄存器的保护机制代码框架

- 在调用其他子函数时才调整栈并保存；在子函数返回时立即恢复

- ◆ 有N次子函数调用，就可能N次保护与恢复；不调用子函数，就无需保护

<pre> 1 函数_label : 2 # 其他函数代码 3 4 addiu \$sp, \$sp, -tmpsize 5 sw \$t[i], [tmpsize-4](\$sp) 6 sw \$t[i+1], [tmpsize-8](\$sp) 7 # 其他需保存的寄存器 8 sw \$t7, 0(\$sp) 9 10 jal 子函数 11 12 lw \$t7, 0(\$sp) 13 # ... 14 lw \$t[i+1], [tmpsize-8](\$sp) 15 lw \$t[i], [tmpsize-4](\$sp) 16 addiu \$sp, \$sp, tmpsize 17 18 # 其他函数代码 </pre>	<div>在调用子函数前分配栈帧 保存寄存器</div> <div>调用子函数</div> <div>恢复寄存器</div> <div>回收栈帧</div>
--	--

计算机组成与实现

小测试

- 下列哪个陈述是错误的？
 - MIPS使用jal指令来调用函数，并使用jr指令函数返回
 - jal保存PC+1到\$ra
 - 函数如果不担心\$ti值在调用子函数后发生改变，则无需保存和恢复它们
 - 函数如果要使用(\$si)，就必须保存和恢复它们

119

计算机组成与实现

小结：函数调用

- 通过组合beq与slt指令，可以实现各种比较判断
- 伪指令使得代码更易读
- 函数机制
 - ◆ 用jal实现函数调用，用jr实现函数返回
 - ◆ 用\$a0-\$a3传递参数，用\$v0-\$v1传递返回值
 - ◆ 寄存器保护
 - 从用途看，寄存器可以分为强保护和弱保护两大类
 - 强保护寄存器：一进函数就保护，退出时恢复
 - 弱保护寄存器：调用子函数时保护，子函数一返回就恢复
 - ◆ 栈用于保存寄存器、局部变量等

120

计算机组成与实现

目录

- 程序执行的基本原理
- 指令格式及其操作数
- 指令集与汇编程序
 - ◆ 空操作
- 指令编码
- 汇编与反汇编实战

空操作

- 空操作（NOP）是一条特殊指令
 - ◆ CPU执行该指令时，不会产生任何实质性动作
- 出于同步等目的，有时需在程序中插入NOP
 - ◆ 在本书里，NOP主要用于解决流水线冲突
- 在MIPS中，NOP指令是不存在
 - ◆ 当汇编器发现NOP时，会将其转换为


```
sll $0,$0,0
```
 - ◆ CPU执行这条指令时，只会解析指令语义，但不会产生任何实质操作
 - 1) \$0值恒为0：故任何移位均无意义
 - 2) 移位数为0：故不会产生移位操作
 - 3) \$0不可写：故写入操作不会发生

目录

- 程序执行的基本原理
- 指令格式及其操作数
- 指令集与汇编程序
- 指令编码
 - ◆ 存储型程序概念
- 汇编与反汇编实战

程序存储概念

- 指令以二进制方式被编码
- 程序存储在存储器中；可以从存储器中读取程序也可以写入程序
 - ◆ 存储方式与数据存储完全相同
- 简化了计算机系统的软件/硬件设计
- 存储器技术既可以存储数据，也可以存储程序
- 由于存储在存储器单元中，因此指令和数据都有地址

二进制兼容

- 程序是以二进制形式发布的
 - ◆ 指令集与程序之间是强相关
- **新机器**不仅能运行基于**新指令**编译产生的**新程序**，同样也最好能运行**老程序**
- 上述特性被称为向后兼容（backward compatible）
 - ◆ 示例：今天的i7处理器仍然能运行1981年在8086处理器上编译产生的程序

125

计算机组成与实现

把指令当做数看待^{1/2}

- 假设：所有的数据都是以字为单位的(32位)
 - ◆ 每个寄存器是字宽度的
 - ◆ lw和sw读写主存的单位是字
- 问题：如何用二进制表示指令？
 - ◆ 计算机只能理解0和1，无法理解“add \$t0,\$0,\$0”
- 回答：数据以字为单位，每条指令同样被编码为一个字！
 - ◆ MIPS的所有指令的二进制编码宽度均为32位

126

计算机组成与实现

把指令当做数看待^{2/2}

- 指令的32位被划分为若干域
 - ◆ 域：占据若干特定位；代表特定含义
 - ◆ 同一个域在不同指令的含义大体是相同的
- 指令的32位解读与数据的32位数是不同的
 - ◆ 数据的32位是作为一个整体被解读
 - ◆ 指令的各个域分别表示指令的不同信息

T
设计越规则
实现越简单

field~域

MIPS的3类指令格式

- **I型指令**：指令中包含立即数
 - ◆ lw/sw的偏移是立即数；beq/bne同样包含有偏移
 - ◆ srl等移位指令：也有5位立即数（移位位数），但不属于I型指令
- **J型指令**：j和jal
 - ◆ jr：不是J型指令
- **R型指令**：所有其他的指令

srl 与 jr 都是 R 型指令！

目录

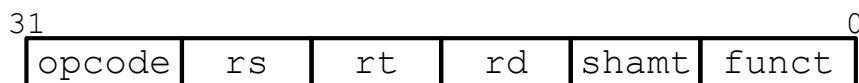
- 程序执行的基本原理
- 指令格式及其操作数
- 指令集与汇编程序
- 指令编码
 - ◆ R型指令
- 汇编与反汇编实战

R型指令^{1/3}

- 指令包含6个域：6 + 5 + 5 + 5 + 5 + 6 = 32



- 为便于理解，每个域都有一个名字



- 每个域都被视为无符号整数

- ◆ 5位域表示范围为：0~31
- ◆ 6位域表示范围为：0~63

R型指令^{2/3}

- **opcode**(6): 代表指令操作
 - ◆ R型指令的opcode固定为0b000000
- **funct**(6): 与opcode组合, 精确定义指令的具体操作
 - ◆ 主要是服务于R型指令
- Q: MIPS最多可以有多少条R型指令?
 - ◆ 由于opcode固定为0, 因此funct的编码数决定了最大条数: 64
- Q: 为什么不将opcode和funct合并为一个12位的域呢?
 - ◆ 后续内容将回答这个问题

131

计算机组成与实现

R型指令^{3/3}

- **rs** (5): 指定1st操作数 (source寄存器)
- **rt** (5): 指定2nd操作数 (target寄存器)
- **rd** (5): 指定结果回写的寄存器 (destination寄存器)
- MIPS的寄存器个数是32, 因此5位无符号数就可以表示
 - ◆ 这种编码方式非常直观: `add dst,src1,src2` → `add rd,rs,rt`
 - ◆ 注意: 与具体指令相关, 有的域是无效的
- **shamt** (5): 移位指令中的移位位数
 - ◆ 由于寄存器只有32位, 因此移位位数大于31没有意义
 - 移位位数大于31, 结果必然为0。既然如此, 直接赋值为0而无需移位
 - ◆ 注意: 除了移位指令, 该域固定为0

指令类型及各域详细描述请阅读指令手册

R型指令示例^{1/2}

□ MIPS指令

```
add    $8,$9,$10
```

□ 伪代码

```
add    R[rd] = R[rs] + R[rt]
```

□ 构造各域

```
opcode = 0          (查手册)
funct = 32          (查手册)
rd = 8              (目的寄存器)
rs = 9              (1st寄存器)
rt = 10             (2nd寄存器)
shamt = 0           (不是移位指令)
```

指令类型及各域详细描述请阅读指令手册

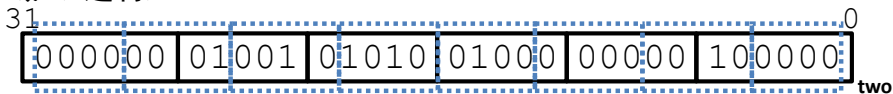
R型指令示例^{2/2}

□ 指令: `add $8,$9,$10`

域 (10进制)



域 (2进制)



16进制: 0x 012A 4020

10进制: 19,546,144

二进制编码: 机器码

通常不这么
表示指令

NOP

- 0x00000000是什么指令？
 - ◆ opcode: 0, 所以这是一条R型指令
- 根据指令手册，机器码对应的指令是


```
sll $0, $0, 0
```
- 该指令的功能就是“空操作”

135

计算机组成与实现



北京航空航天大学计算机学院
School of Computer Science and Engineering, Beihang University

目录

- 程序执行的基本原理
- 指令格式及其操作数
- 指令集与汇编程序
- 指令编码
 - ◆ I型指令
- 汇编与反汇编实战

I型指令^{3/4}

- **opcode**(6): 代表指令操作
 - ◆ I型指令的opcode为非零编码: 总共可以编码64 (2^6) 条指令
 - ◆ R型指令用opcode和funct两个独立域有助于保持格式的兼容性
- **rs** (5): 指定1st操作数 (source寄存器)
- **rt** (5): 指定2nd操作数 (target寄存器)
 - ◆ target并非都是“目的”。例如: sw的rt就是“读”
- **immediate** (16): 无符号or有符号
 - ◆ 无符号: 位运算指令 (如and/or/nor等)、小于置位指令 (如slti等)
 - `zero_ext()`: 运算前需要进行无符号扩展
 - ◆ 有符号: 分支指令 (如beq/bne等)、访存指令 (如lw/sw等)
 - 以word为单位
 - `sign_ext()`: 运算前需要进行符号扩展

139

Q
对于lw/sw, 16位立即数是否够用?

I型指令示例^{1/2}

- **MIPS指令**

```
addi $21,$22,-50
```
- **伪代码**

```
addi R[rt] = R[rs] + sign_ext(immediate)
```
- **构造各域**

<code>opcode = 8</code>	(查手册)
<code>rs = 22</code>	(1 st 寄存器)
<code>rt = 21</code>	(2 nd 寄存器)
<code>immediate = -50</code>	(10进制或16进制表示均可)

T
`addi $rt, $rs, immediate`

I型指令示例^{2/2}

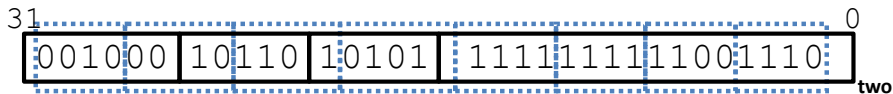
□ 指令:

addi \$21,\$22,-50

域 (10进制)



域 (2进制)



16进制: 0x 22D5 FFCE

T

addi \$rt, \$rs, immediate

如何计算32位立即数?

□ 32位立即数的应用场景, 如:

- ◆ addi/slti/andi/ori: 等需要计算2个32位数
- ◆ lw/sw: 在使用前, 需要先设置基地址寄存器的值 (32位)

□ 解决方案: 不改变指令格式, 而是增加一条新指令

□ Load Upper Immediate (lui)

- ◆ lui reg,imm
- ◆ reg的高16位写入imm, 低16位写入0
 - RTL: $R[\text{reg}] \leftarrow \text{imm} \parallel 0^{16}$

lui示例

- 需求：addi \$t0,\$t0,0xABABCD



这是一条伪指令！

- 会被assembler转换为3条指令

```
lui $at,0xABAB    # 高16位
ori $at,$at,0xCDCD # 低16位
add $t0,$t0,$at    # 赋值
```

T
手工编写汇编时，
尽量不适\$at；只
应由assembler使
用\$at

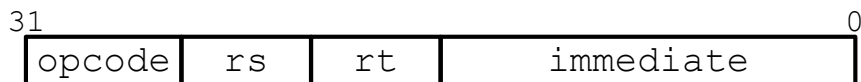
- 通过增加lui，MIPS可以用16位立即数来处理任意大小的数据

143

计算机组成与实现

分支指令（B类指令）

- beq和bne



- 比较rs和rt，并根据比较结果决定是否转移



并非所有的B类指令的rt域都是寄存器！

- 需要指定要转移的地址

- 分支指令主要用于构造：if-else，while，for

- 转移的范围通常很小（<50指令）
- 函数调用和无条件跳转用J型指令

- Q：如何用immediate表示地址？

- 由于指令存储在主存中，而主存单元可以由基地址+偏移的方式定位
- B类指令的基地址就是PC（即当前这条B类指令的PC值）

144

计算机组成与实现

PC相对寻址

- **PC相对寻址**：PC为基地址，**immediate**为偏移（二进制补码）
 - ◆ 基本计算方法： $PC \leftarrow PC + \text{偏移}$
 - ◆ 关键是：偏移值如何得到
- 下一条指令的PC值的计算方法
 - ◆ 比较结果为**假**： $PC = PC + 4$
 - ◆ 比较结果为**真**： $PC = (PC + 4) + (\text{immediate} * 4)$
- Q1：为什么**immediate**乘以4？
 - ◆ 存储器是以字节为单位的
 - ◆ 指令都是32位长，且指令是字对齐，这意味着最低2位恒为0
 - ◆ **immediate**没有必要记录最低2位，乘以4后就得到了对应字节地址！
- Q2：为什么基地址是PC+4，而不是PC？
 - ◆ 这与硬件设计有关，后续内容讲解

145

计算机组成与实现

immediate表示的转移范围是否足够大？

- **immediate**为16位符号数，其表示范围是正负 $\pm 2^{15}$
- 由于转移范围为 $\pm 2^{15}$ 字，意味着转移的指令数为 $\pm 2^{15}$ 条
 - ◆ **immediate**省略了最低2位，即量纲为字
- 按照1行C代码对应10条指令，则可转移的C代码块大小为3000行！

146

计算机组成与实现

B类指令示例^{1/2}

□ MIPS指令

```

1  Loop: beq  $9, $0, End
2          addu $8, $8, $10
3          addiu $9, $9, -1
4          j    Loop
5  End:

```

从beq后面的那条指令 (addu) 计算

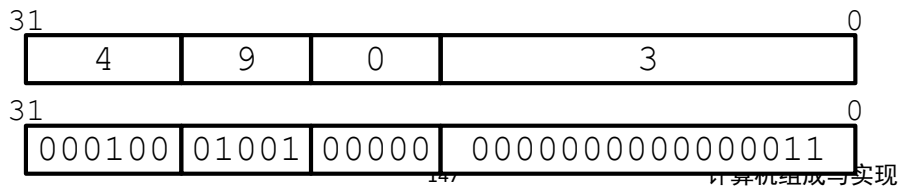
□ 构造各域

opcode = 4 (查手册)

rs = 9 (1st寄存器)

rt = 0 (2nd寄存器)

immediate = 3



更深入的思考

- 如果移动代码，是否会导致B类指令的immediate域的变化？
 - ◆ 会变化：如果只移动某行，而不是移动B指令所涉及的整块代码
 - ◆ 无变化：如果移动B指令所涉及的整块代码
- 如果跳转的目的地址超出了2¹⁵条指令，该怎么办？
 - ◆ 组合B类指令与J类指令

```

beq $s0,$0,far          bne $s0,$0,next
# next instr          --> j    far
                        next: # next instr

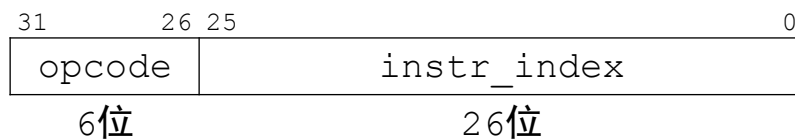
```

目录

- 程序执行的基本原理
- 指令格式及其操作数
- 指令集与汇编程序
- 指令编码
 - ◆ J型指令
- 汇编与反汇编实战

J型指令

- J指令只定义了2个域



- 要点
 - ◆ opcode的位置及位数与R型、I型指令保持一致
 - ◆ 其他域合并在一起构造大的地址范围

J型指令

$$PC \leftarrow PC_{31..28} || instr_index || 0^2$$

□ Q: J指令的转移范围有多大?

□ 分析

- ◆ PC的高4位来自当前指令的高4位
- ◆ PC的低28位用于寻址
- ◆ 意味着目的地址与当前指令必在同一区段{ X000_0000 , XFFF_FFFF }
- ◆ 故J指令的转移范围是256MB

□ 由此可知, 程序员无法用J指令跳转到256MB以外的地址空间 (2^{28})

🔴 只有jr可以跳转到4GB内任意地址

- ◆ jr: R型指令

区段15	F000_0000~FFFF_FFFF
区段14	E000_0000~EFFF_FFFF
.....	
区段2	2000_0000~2FFF_FFFF
区段1	1000_0000~1FFF_FFFF
区段0	0000_0000~0FFF_FFFF

151

计算机组成与实现

小结: 指令格式

□ 现代计算机都是程序存储型的

- ◆ 指令与数据一样存储在主存中
- ◆ 读取指令与读取数据完全可以使用相同的硬件机制
- ◆ 指令与数据位于不同区域
 - 通过PC读取的“32位01串”都被CPU当做指令
 - 通过Load/Store指令读写的“32位01串”都被CPU当做数据

□ 3类指令格式

R:	opcode	rs	rt	rd	shamt	funct
I:	opcode	rs	rt	immediate		
J:	opcode	target address				

□ B类指令使用PC相对寻址, J指令使用绝对地址寻址

152

计算机组成与实现

目录

- 程序执行的基本原理
- 指令格式及其操作数
- 指令集与汇编程序
- 指令编码
- 汇编与反汇编实战
 - ◆ 汇编实战

汇编

- 汇编是将汇编程序转换成二进制机器码的过程
- 每条指令的汇编基本步骤
 - ◆ S1: 标识出指令类型 (R/I/J)
 - ◆ S2: 标识出正确的域
 - ◆ S3: 用10进制表示各个域的值
 - ◆ S4: 把各个域的10进制转换为2进制
 - ◆ S5: 用16进制表示整个机器码

汇编实例

- 将下列汇编代码转换为相应的2进制机器码

地址 指令

800 Loop: sll \$t1,\$s3,2

804 addu \$t1,\$t1,\$s6

808 lw \$t0,0(\$t1)

812 beq \$t0,\$s5, Exit

816 addiu \$s3,\$s3,1

820 j Loop

Exit:

...

汇编实现

汇编实例

- S1: 标识出指令类型

地址 指令

800 Loop: sll \$t1,\$s3,2

R

--	--	--	--	--	--

804 addu \$t1,\$t1,\$s6

R

--	--	--	--	--	--

808 lw \$t0,0(\$t1)

I

--	--	--	--

812 beq \$t0,\$s5, Exit

I

--	--	--	--

816 addiu \$s3,\$s3,1

I

--	--	--	--

820 j Loop

J

--	--	--	--

Exit:

...

汇编实现

汇编实例

□ S2: 标示出正确的域

地址 指令

800 Loop: sll \$t1,\$s3,2

R

opcode	rs	rt	rd	shamt	funct
--------	----	----	----	-------	-------

804 addu \$t1,\$t1,\$s6

R

opcode	rs	rt	rd	shamt	funct
--------	----	----	----	-------	-------

808 lw \$t0,0(\$t1)

I

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

812 beq \$t0,\$s5, Exit

I

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

816 addiu \$s3,\$s3,1

I

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

820 j Loop

J

opcode	target address
--------	----------------

Exit:

实现

汇编实例

□ S3: 用10进制表示各个域的值

地址 指令

800 Loop: sll \$t1,\$s3,2

R

0	0	19	9	2	0
---	---	----	---	---	---

804 addu \$t1,\$t1,\$s6

R

0	9	22	9	0	33
---	---	----	---	---	----

808 lw \$t0,0(\$t1)

I

35	9	8	0
----	---	---	---

812 beq \$t0,\$s5, Exit

I

4	8	21	2
---	---	----	---

816 addiu \$s3,\$s3,1

I

8	19	19	1
---	----	----	---

820 j Loop

J

2	200
---	-----

Exit:

实现

汇编实例

□ S4: 把10进制转换为2进制

地址 指令

800	Loop: sll \$t1,\$s3,2					
R	000000	00000	10011	01001	00010	000000
804	addu \$t1,\$t1,\$s6					
R	000000	01001	10110	01001	00000	100001
808	lw \$t0,0(\$t1)					
I	100011	01001	01000	0000	0000	0000 0000
812	beq \$t0,\$s5, Exit					
I	000100	01000	10101	0000	0000	0000 0010
816	addiu \$s3,\$s3,1					
I	001000	00000	10011	0000	0000	0000 0001
820	j Loop					
J	000010	00	0000	0000	0000	0000 1100 1000

Exit:

实现

...

11 11 11 11 11 11 11 实现

汇编实例

□ S5: 转成16进制

地址	指令
800	Loop: sll \$t1,\$s3,2
R	0x 0013 4880
804	addu \$t1,\$t1,\$s6
R	0x 0136 4821
808	lw \$t0,0(\$t1)
I	0x 8D28 0000
812	beq \$t0,\$s5, Exit
I	0x 1115 0002
816	addiu \$s3,\$s3,1
I	0x 2273 0001
820	j Loop
J	0x 0800 00C8
Exit:	

...

11 11 11 11 11 11 11 实现

目录

- 程序执行的基本原理
- 指令格式及其操作数
- 指令集与汇编程序
- 指令编码
- 汇编与反汇编实战
 - ◆ 反汇编实战

反汇编

- 反汇编是汇编的逆过程，即将指令二进制代码转换为汇编代码的过程
- 反汇编基本步骤
 - ◆ S1：用2进制表示指令
 - ◆ S2：根据opcode标识出指令类型（R/I/J）
 - ◆ S3：用10进制表示各个域的值
 - ◆ S4：用标识符表示各域，并添加相应的标号
 - ◆ S5：用汇编格式书写代码
 - ◆ S6：将汇编代码翻译为C
 - 通常很难翻译为可读性C代码！需要创造性！

反汇编实例

- 把下来机器码翻译为汇编程序

地址	指令
0x00400000	0x00001025
0x00400004	0x0005402A
0x00400008	0x11000003
0x0040000C	0x00441020
0x00400010	0x20A5FFFF
0x00400014	0x08100001

163

计算机组成与实现

反汇编实例

- S1: 用2进制表示指令

地址	指令
0x00400000	0000000000000000000010000000100101
0x00400004	000000000000001010100000000101010
0x00400008	000100010000000000000000000000011
0x0040000C	00000000010001000001000000100000
0x00400010	00100000101001011111111111111111
0x00400014	000010000001000000000000000000001

R:	opcode	rs	rt	rd	shamt	funct
I:	opcode	rs	rt	immediate		
J:	opcode	target address				

计算机组成与实现

反汇编实例

- S2: 根据opcode标示出指令类型

地址	指令
0x00400000	R 000000 000000 000000 00010 000000 100101
0x00400004	R 000000 000000 00101 01000 000000 101010
0x00400008	I 000100 01000 00000 000000000000000011
0x0040000C	R 000000 00010 00100 00010 000000 100000
0x00400010	I 001000 00101 00101 1111111111111111
0x00400014	J 000010 00000100000000000000000000001

R:	opcode	rs	rt	rd	shamt	funct
I:	opcode	rs	rt	immediate		
J:	opcode	target address				

计算机组成与实现

反汇编实例

- S3: 用10进制表示各个域的值

地址	指令
0x00400000	R 0 0 0 2 0 37
0x00400004	R 0 0 5 8 0 42
0x00400008	I 4 8 0 +3
0x0040000C	R 0 2 4 2 0 32
0x00400010	I 8 5 5 -1
0x00400014	J 2 0x0100001

T

J指令的地址域保持为16进制

R:	opcode	rs	rt	rd	shamt
I:	opcode	rs	rt	immediate	
J:	opcode	target address			

计算机组成与实现

计算机组成与实现

反汇编实例

□ S4: 用标识符表示各域

地址	指令
0x00400000	R 0 \$0 \$0 \$2 0 or
0x00400004	R 0 \$0 \$5 \$8 0 slt
0x00400008	I beq \$8 \$0 +3
0x0040000C	R 0 \$2 \$4 \$2 0 add
0x00400010	I addi \$5 \$5 -1
0x00400014	J j 0x0100001

R:	opcode	rs	rt	rd	shamt	funct
I:	opcode	rs	rt	immediate		
J:	opcode	target address				

计算机组成与实现

反汇编实例

□ S5-1: 用汇编格式书写

地址	指令
0x00400000	or \$v0, \$0, \$0
0x00400004	slt \$t0, \$0, \$a1
0x00400008	beq \$t0, \$0, 3
0x0040000C	add \$v0, \$v0, \$a0
0x00400010	addi \$a1, \$a1, -1
0x00400014	j 0x0100001 # addr: 0x0400004

R:	opcode	rs	rt	rd	shamt	funct
I:	opcode	rs	rt	immediate		
J:	opcode	target address				

计算机组成与实现

反汇编实例

- S5-1: 用汇编格式书写（用寄存器名有助于提高可读性）

地址	指令
0x00400000	or \$v0,\$0,\$0
0x00400004	slt \$t0,\$0,\$a1
0x00400008	beq \$t0,\$0,3
0x0040000C	add \$v0,\$v0,\$a0
0x00400010	addi \$a1,\$a1,-1
0x00400014	j 0x0100001 # addr: 0x0400004

169

计算机组成与实现

反汇编实例

- S5-2: 用汇编格式书写（添加标号）

地址	指令
0x00400000	or \$v0,\$0,\$0
0x00400004	Loop: slt \$t0,\$0,\$a1
0x00400008	beq \$t0,\$0,Exit
0x0040000C	add \$v0,\$v0,\$a0
0x00400010	addi \$a1,\$a1,-1
0x00400014	j Loop
	Exit:

170

计算机组成与实现

反汇编实例

□ S5-2: 用汇编格式书写（添加标号）

```

        or    $v0,$0,$0    # initialize $v0 to 0
Loop:   slt   $t0,$0,$a1    # $t0 = 0 if 0 >= $a1
        beq   $t0,$0,3     # exit if $a1 <= 0
        add   $v0,$v0,$a0  # $v0 += $a0
        addi  $a1,$a1,-1   # decrement $a1
        j     Loop

```

171

计算机组成与实现

反汇编实例

□ S6: 将汇编程序转换为C程序

- ◆ 可以有很多种C代码对应到同一段汇编代码

```

/* a→$v0, b→$a0, c→$a1 */
a = 0;
while(c > 0) {
    a += b;
    c--;
}

```

- 代码分析：循环c次，每次累加b，即b被累加c次
- 代码功能：a = b × c

172

计算机组成与实现