

处理器体系结构

Ch2指令集体系结构B

—MIPS指令集体系结构与汇编语言入门

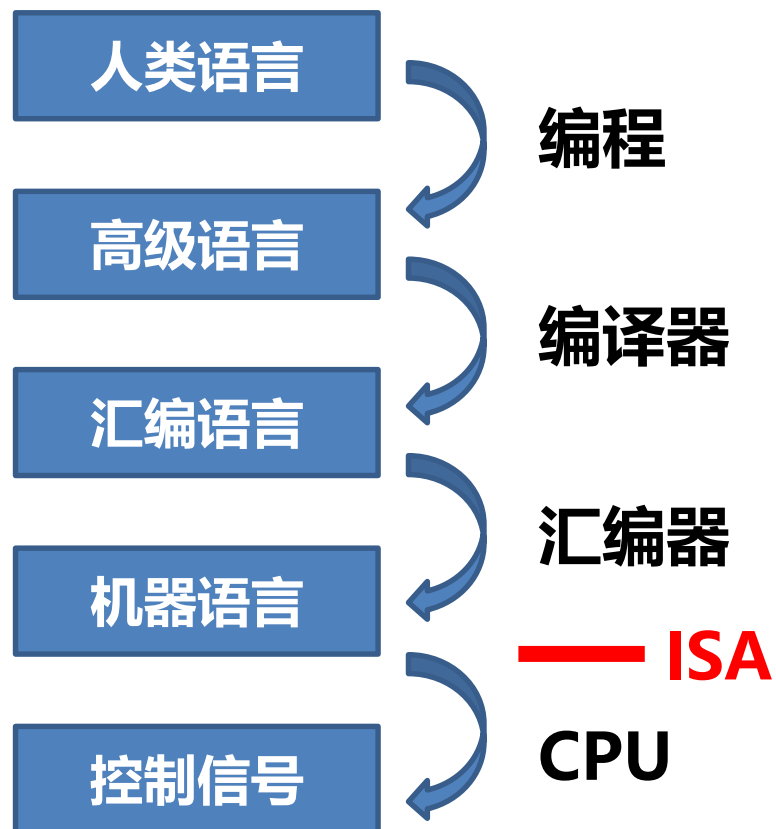
(Micro-processor Architecture)

课程回顾-- ISA的作用

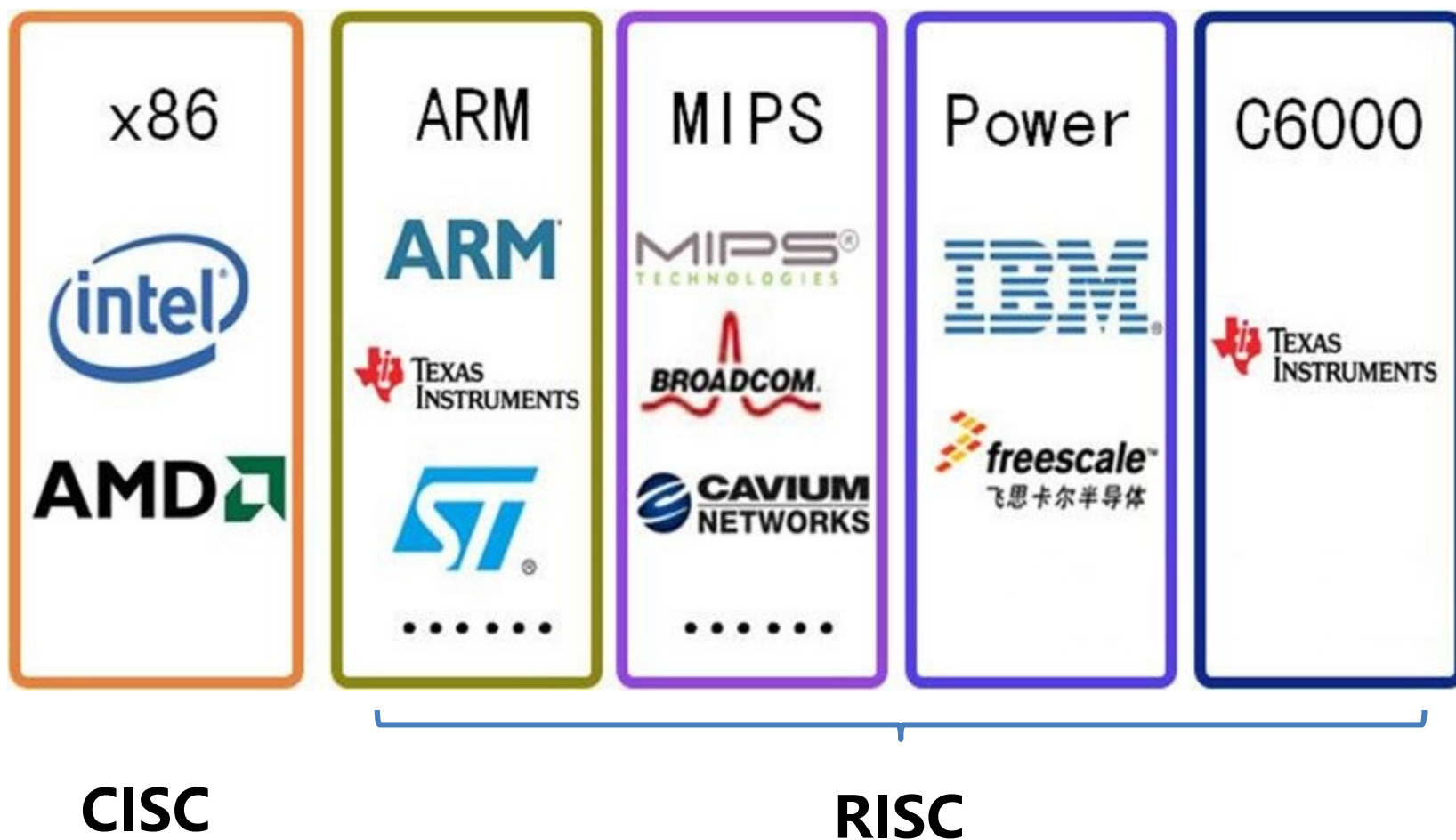
ISA在汇编器编写者（CPU软件）和
处理器设计人员（CPU硬件）之间提
供了一个抽象层

处理器设计者：依据ISA来设计处理器

处理器使用者（如：写汇编器的牛*程序员）：依据ISA就知道CPU选用的指令集，就知道自己可以使用哪些指令以及遵循哪些规范



课程回顾--主流的指令集体系结构(ISA)



本节课内容： MIPS指令集体系结构与汇编语言入门

中国留学生在国外的高速公路出车祸了，连人带车翻下悬崖，交警赶到后向下喊话道：“How are you?” 留学生答：“I’ m fine, thank you!” 然后交警走了，留学生就死了。

本节课内容： MIPS指令集体系结构与汇编语言入门

汇编语言： 机器指令的助记符，面向机器

汇编指令与机器指令一一对应（伪指令除外）

早期，不同处理器有着不同的汇编语言和汇编器

-> 程序**缺乏可移植性**

ISA: x86、ARM、MIPS、POWER、 ...

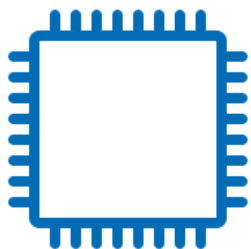
MIPS指令集体系结构->MIPS汇编语言

学习汇编语言需要 “与时俱进”



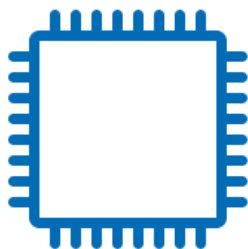
关于MIPS

官网: <https://www.mips.com/>



Current Cores

Latest family of MIPS CPUs offers best-in-class performance, power and area efficiency



Classic Cores

Widely licensed and cost-effective solutions for embedded and multimedia applications



MIPS Architecture

Highest levels of performance with clean, elegant design



32-bit embedded microcontrollers

64-bit servers and infrastructure processors

...and everything in-between



Markets ▾

Develop

Artificial Intelligence

Automotive

Consumer

Internet of Things (IoT)

Networking

关于MIPS

官网: <https://www.mips.com/>

Architectures

Based on a heritage built over more than three decades of constant innovation, the MIPS architecture is the industry's most efficient RISC architecture, delivering the best performance and lowest power consumption in a given silicon area.

- nanoMIPS Architecture
- MIPS32 Instruction Set Architecture (ISA)
- MIPS64 Architecture ISA
- microMIPS ISA
- MIPS Multi-Threading architecture module
- MIPS Virtualization architecture module
- MIPS SIMD architecture module
- MIPS DSP architecture module
- MIPS MCU architecture module
- MIPS16e architecture module

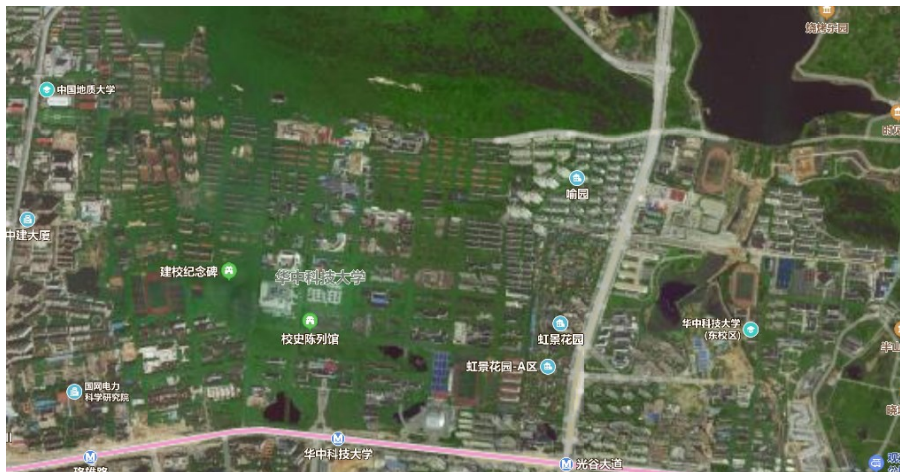


度娘了下, MIPS很简单嘛, 只有31条指令

->错误! 去看看MIPS32和MIPS64的手册

*处理器体系结构的学习方法

如果华中科技大学是一款高性能处理器，如何去了解她？



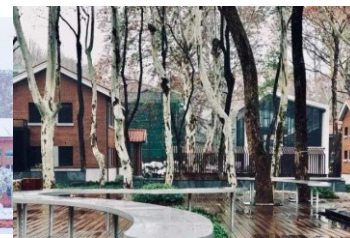
院系设置

机械科学与工程学院
电气与电子工程学院
能源与动力工程学院
水电与数字化工程学院
电子信息与通信学院
化学与化工学院
公共管理学院
人文学院
社会学院
基础医学院
公共卫生学院
生物医学研究所(原计划生育研究所)
第三临床学院
中科院清与可再生能源学院
工程科学学院(国际化学学院)
网络空间安全学院



数学与统计学院
经济学院
新闻与传播学院
法学院
药学院
医学院
第一临床学院
远程与继续教育学院
体育学院
武汉国际微电子学院

物理学院
管理学院
马克思主义学院
外国语学院
医药卫生管理学院
护理学院
第二临床学院
教育科学研究所
武汉国家研究中心
航空航天学院



坐北朝南，依山傍水；左有森林公园，右有光谷中心；
四大工学院之首，素有学在华工的美誉！

下设46个学院，103个本科专业，45个硕士学位授权
一级学科，41个博士学位授权一级学科，39个博士后
科研流动站。。。



由简入深：先有骨骼，再有血肉（天才少年请无视）
得鱼得渔：以MIPS32为鱼，以设计思想为渔

我们将以MIPS32最核心的31条指令，逐步掌握其设计思想、电路实现和优化策略

MIPS指令集体系结构与汇编语言入门

1. MIPS中的操作数
2. 指令在计算机内部的表示
3. 关于存储程序
4. 逻辑运算指令
5. 决策指令

MIPS指令集体系结构与汇编语言入门

1. MIPS中的操作数

1.1 寄存器操作数

2. 指令在计算机内部的表示

1.2 存储器操作数

3. 关于存储程序

1.3 立即数操作数

4. 逻辑运算指令

5. 决策指令

1. MIPS中的操作数--1.1 寄存器操作数

加法与减法，**有且仅有**三个操作数：两个源操作数、一个目的操作数

指令格式：add rd, rs, rt #rd=rs+rt

其它数值运算的格式同上

设计原则1：简单源自规整

规整可简化电路实现

简单可以在较低的成本下实现高性能

例：以下C程序如何编译？：

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

我们当然可以设计出一条复杂的指令，
一次性完成上述运算，但这会增加复杂度，
且降低指令的使用率



add t0, g, h

add t1, i, j

sub f, t0, t1

-> 需要频繁地访问存储器！

操作数应当放在哪里？

1. MIPS中的操作数--1.1寄存器操作数

MIPS算术运算指令的**操作数必须直接取自寄存器!**

MIPS拥有过32×32bit的寄存器 (32个字)

编号: 0~31

寄存器地址	寄存器名称	名称含义	用途
\$0	\$zero	Zero	常量0
\$1	\$at	Assembler temporary	留给汇编器作临时变量
\$2-\$3	\$v0-\$v1	Values	子函数调用返回值
\$4-\$7	\$a0-\$a3	Arguments	子函数调用参数
\$8-\$15	\$t0-\$t7	Temporaries	存放临时变量(随便用的)
\$16-\$23	\$s0-\$s7	Saved values	保存变量(子函数调用前后)
\$24-\$25	\$t8-\$t9	Temporaries	存放临时变量(随便用的)
\$26-\$27	\$k0-\$k1	Kernel	中断、异常处理保存的参数
\$28	\$gp	Global pointer	全局指针
\$29	\$sp	Stack pointer	堆栈指针
\$30	\$fp	Frame pointer	帧指针
\$31	\$ra	Return address	子函数返回地址

\$t0~\$t9

\$s0~\$s7



设计原则2: 越少越快

寄存器个数<->时钟周期、指令格式位数

1. MIPS中的操作数--1.1寄存器操作数

刚刚的例子: $f = (g + h) - (i + j);$

假设: $f \sim j$ 分别分配给 $\$s0 \sim \$s4$, 则MIPS指令为:

add t0, g, h		add \$t0, \$s1, \$s2
 add t1, i, j		add \$t1, \$s3, \$s4
sub f, t0, t1		sub \$s0, \$t0, \$t1

-> 少量运算可以直接使用寄存器数据, 但大量数据存哪儿?

1. MIPS中的操作数--1.2 存储器操作数



ALU



寄存器



存储器

数据传送指令: lw、sw

单位: 比特(Bit)?、字节(Byte)?、字(word)?

注意: 地址编号必须为**4的倍数**

寄存器非常宝贵: 必须有效地使用寄存器(性能优化)



1. MIPS中的操作数--1.2 存储器操作数

大小端问题：如何将一个字存进内存（以0x12345678为例）

内存地址	小端模式存放内容	大端模式存放内容
0x4000	0x78	0x12
0x4001	0x56	0x34
0x4002	0x34	0x56
0x4003	0x12	0x78

MIPS、PowerPC：大端模式存储

x86：小端模式存储

1. MIPS中的操作数--1.2 存储器操作数

例1：C语言代码 $A[12] = h + A[8]$

假设h存于\$*s2*，数列A的基址存于\$*s3*中，则对应的MIPS代码为：

MIPS代码：

`lw $t0, 32($s3)`

`add $t0, $s2, $t0`

`sw $t0, 48($s3)`

指令格式：

`lw rt, offset(rs)`

`sw rt, offset(rs)`

思考：32位MIPS体系架构下，最多有多大地址空间？

$2^{32} = 4\text{GB (Giga Byte)}$

1. MIPS中的操作数--1.2 存储器操作数

例2：假设A是一个数组，基址存于寄存器\$*s3*，变量*g*、*h*、*i*分别放在\$*s1*、\$*s2*、\$*s4*中，则将下面的C语言转换为MIPS汇编语言：

$g = h + A[i]$

获得A[i]的地址:

```
add  $t1, $s4, $s4
add  $t1, $t1, $t1
add  $t1, $t1, $s3
```

将A[i]取到寄存器中:

```
lw   $t0, 0($t1)
```

执行加法:

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

1. MIPS中的操作数--1.3 立即数操作数

思考：for循环中的 $k = k + 1$ 如何实现？

```
lw      $t0, AddrConstant4($s1) #假设$s1+AddrConstant4是常量4的存储器地址
add     $s3, $s3, $t0
```

->速度慢，且会消耗一个寄存器资源

解决办法：提供立即数加法=> `addi $s3, $s3, ?` **#?: 4**

设计原则3：加速常用操作

注意：没有“subi” => `addi $s3, $s3, -4`

MIPS指令集体系结构与汇编语言入门

1. MIPS中的操作数

2. 指令在计算机内部的表示

3. 关于存储程序

4. 逻辑运算指令

5. 决策指令

2.1 R型指令

2.2 I型指令

暂时不看J型指令

2. 指令在计算机内部的表示--2.1 R型指令

处理器如何解读汇编语句?

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

op: operation code (opcode)

操作码

rs: first source **register** number

第一个源操作数寄存器

rt: second source **register** number

第二个源操作数寄存器

rd: destination **register** number

目的操作数寄存器

shamt: shift amount (00000 for now)

位移量 (移位指令)

funct: function code (extends opcode)

函数码

"R" : Register, 对寄存器进行操作

寄存器命名:

rs: register source

rt: register target

rd: register destination

*关于函数码

思考：既然函数码相当于操作码的扩展，
那为什么不把它们合并？（为什么采用6
位操作码+6位函数码，而非12位操作
码？）

U2017...

想想我们的学号：

M2018...

D2016...

这样做更便于译码和加速！

CORE INSTRUCTION SET			FOR-MAT	OPERATION (in Verilog)	OPCODE / FUNCT (Hex)
Add	add	R	R	$R[rd] = R[rs] + R[rt]$	(1) 0 / 20 _{hex}
Add Immediate	addi	I	R	$R[rt] = R[rs] + \text{SignExtImm}$	(1,2) 8 _{hex}
Add Imm. Unsigned	addiu	I	R	$R[rt] = R[rs] + \text{SignExtImm}$	(2) 9 _{hex}
Add Unsigned	addu	R	R	$R[rd] = R[rs] + R[rt]$	0 / 21 _{hex}
And	and	R	R	$R[rd] = R[rs] \& R[rt]$	0 / 24 _{hex}
And Immediate	andi	I	R	$R[rt] = R[rs] \& \text{ZeroExtImm}$	(3) c _{hex}
Branch On Equal	beq	I		if($R[rs] == R[rt]$) $PC = PC + 4 + \text{BranchAddr}$	(4) 4 _{hex}
Branch On Not Equal	bne	I		if($R[rs] != R[rt]$) $PC = PC + 4 + \text{BranchAddr}$	(4) 5 _{hex}
Jump	j	J		$PC = \text{JumpAddr}$	(5) 2 _{hex}
Jump And Link	jal	J		$R[31] = PC + 8; PC = \text{JumpAddr}$	(5) 3 _{hex}
Jump Register	jr	R		$PC = R[rs]$	0 / 08 _{hex}
Load Byte Unsigned	lbu	I		$R[rt] = \{24'b0, M[R[rs] + \text{SignExtImm}](7:0)\}$	(2) 24 _{hex}
Load Halfword Unsigned	lhu	I		$R[rt] = \{16'b0, M[R[rs] + \text{SignExtImm}](15:0)\}$	(2) 25 _{hex}
Load Linked	ll	I		$R[rt] = M[R[rs] + \text{SignExtImm}]$	(2,7) 30 _{hex}
Load Upper Imm.	lui	I		$R[rt] = \{\text{imm}, 16'b0\}$	f _{hex}
Load Word	lw	I		$R[rt] = M[R[rs] + \text{SignExtImm}]$	(2) 23 _{hex}
Nor	nor	R	R	$R[rd] = \sim (R[rs] R[rt])$	0 / 27 _{hex}
Or	or	R	R	$R[rd] = R[rs] R[rt]$	0 / 25 _{hex}
Or Immediate	ori	I	R	$R[rt] = R[rs] \text{ZeroExtImm}$	(3) d _{hex}
Set Less Than	slt	R		$R[rd] = (R[rs] < R[rt]) ? 1 : 0$	0 / 2a _{hex}
Set Less Than Imm.	slti	I		$R[rt] = (R[rs] < \text{SignExtImm}) ? 1 : 0$	(2) a _{hex}
Set Less Than Imm. Unsigned	sltiu	I		$R[rt] = (R[rs] < \text{SignExtImm}) ? 1 : 0$	(2,6) b _{hex}
Set Less Than Unsig.	sltu	R		$R[rd] = (R[rs] < R[rt]) ? 1 : 0$	(6) 0 / 2b _{hex}
Shift Left Logical	sll	R		$R[rd] = R[rt] << \text{shamt}$	0 / 00 _{hex}
Shift Right Logical	srl	R		$R[rd] = R[rt] >>> \text{shamt}$	0 / 02 _{hex}
Store Byte	sb	I		$M[R[rs] + \text{SignExtImm}](7:0) = R[rt](7:0)$	(2) 28 _{hex}
Store Conditional	sc	I		$M[R[rs] + \text{SignExtImm}] = R[rt];$ $R[rt] = (\text{atomic}) ? 1 : 0$	(2,7) 38 _{hex}
Store Halfword	sh	I		$M[R[rs] + \text{SignExtImm}](15:0) = R[rt](15:0)$	(2) 29 _{hex}
Store Word	sw	I		$M[R[rs] + \text{SignExtImm}] = R[rt]$	(2) 2b _{hex}
Subtract	sub	R	R	$R[rd] = R[rs] - R[rt]$	(1) 0 / 22 _{hex}
Subtract Unsigned	subu	R	R	$R[rd] = R[rs] - R[rt]$	0 / 23 _{hex}

2. 指令在计算机内部的表示--2.1 R型指令

例：指令 “add \$t0, \$s1, \$s2” 对应的机器码？

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

\$8-\$15	\$t0-\$t7
\$16-\$23	\$s0-\$s7
\$24-\$25	\$t8-\$t9

0	\$s1	\$s2	\$t0	0	add
---	------	------	------	---	-----

0	17	18	8	0	32
---	----	----	---	---	----

000000	10001	10010	01000	00000	100000
--------	-------	-------	-------	-------	--------

$$00000010001100100100000000100000_2 = 02324020_{16}$$

R型指令的格式是否适合取字？：lw rt, 4n(rs)

->新的矛盾：若保持相同指令格式，则需要更长的指令（或变长）

2. 指令在计算机内部的表示--2.2 I型指令

设计原则4: 优秀的设计者需要适当的折中

\$8-\$15	\$t0-\$t7
\$16-\$23	\$s0-\$s7
\$24-\$25	\$t8-\$t9

“I” : Immediate, 立即数指令



Constant or address: 立即数或地址偏移量 (字节)

补码: $-2^{15} \sim +2^{15}-1$

例: 指令 “lw \$t0, 32(\$s3)” 的机器码?

43	19	8	32
----	----	---	----

I型指令的折中: 保持指令长度相同, 而不同类型的指令采用不同格式; 不同格式的指令显然增加了复杂度, 所以让格式尽可能的类似

-> 处理器可根据op码区分R和I型指令, 进而作不同处理

2. 指令在计算机内部的表示--2.2 I型指令

例：假设数组A的基址放在\$t1中，h放在寄存器\$s2中，则

“A[300]=h+A[300]” 将如何编译成机器码？

lw \$t0, 1200(\$t1)

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

sw \$t0, 1200(\$t1)

\$8-\$15	\$t0-\$t7
\$16-\$23	\$s0-\$s7
\$24-\$25	\$t8-\$t9

Op	Rs	Rt	rd-shamt-funct or constant\address		
35	9	8	1200		
0	18	8	8	0	32
43	9	8	1200		
100011	01001	01000	0000 0100 1011 0000		
000000	10010	01000	01000	00000	100000
101011	01001	01000	0000 0100 1011 0000		

习题

\$8-\$15	\$t0-\$t7
\$16-\$23	\$s0-\$s7
\$24-\$25	\$t8-\$t9

下列MIPS指令是什么含义？

00000010001010010101000000100000

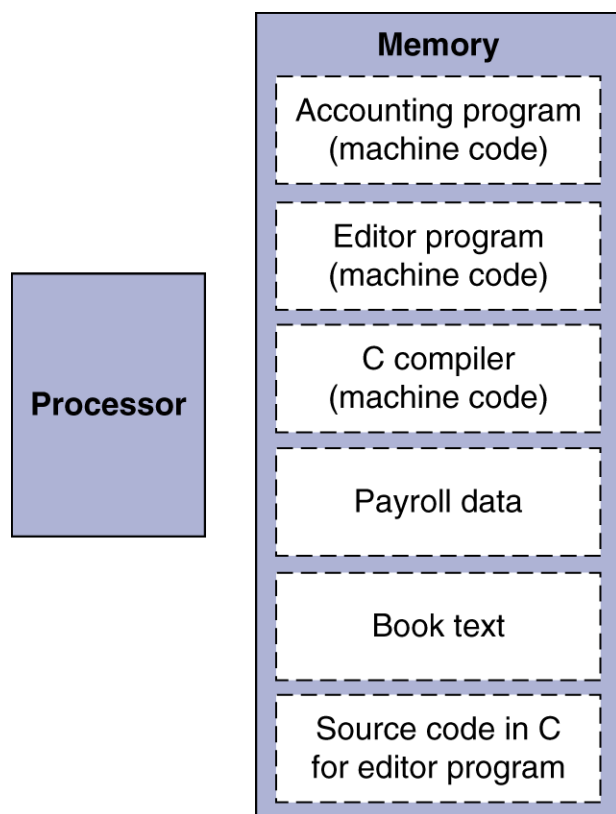
CORE INSTRUCTION SET		FOR-MAT	OPERATION (in Verilog)	OPCODE / FUNCT (Hex)
Add	add	R	$R[rd] = R[rs] + R[rt]$	(1) $0 / 20_{hex}$
Add Immediate	addi	I	$R[rt] = R[rs] + \text{SignExtImm}$	(1,2) 8_{hex}
Add Imm. Unsigned	addiu	I	$R[rt] = R[rs] + \text{SignExtImm}$	(2) 9_{hex}
Add Unsigned	addu	R	$R[rd] = R[rs] + R[rt]$	$0 / 21_{hex}$
And	and	R	$R[rd] = R[rs] \& R[rt]$	$0 / 24_{hex}$
And Immediate	andi	I	$R[rt] = R[rs] \& \text{ZeroExtImm}$	(3) c_{hex}
Branch On Equal	beq	I	if($R[rs] == R[rt]$) $PC = PC + 4 + \text{BranchAddr}$	(4) 4_{hex}
Branch On Not Equal	bne	I	if($R[rs] != R[rt]$) $PC = PC + 4 + \text{BranchAddr}$	(4) 5_{hex}
Jump	j	J	$PC = \text{JumpAddr}$	(5) 2_{hex}
Jump And Link	jal	J	$R[31] = PC + 8; PC = \text{JumpAddr}$	(5) 3_{hex}
Jump Register	jr	R	$PC = R[rs]$	$0 / 08_{hex}$
Load Byte Unsigned	lbu	I	$R[rt] = \{24'b0, M[R[rs] + \text{SignExtImm}](7:0)\}$	(2) 24_{hex}
Load Halfword Unsigned	lhu	I	$R[rt] = \{16'b0, M[R[rs] + \text{SignExtImm}](15:0)\}$	(2) 25_{hex}
Load Linked	ll	I	$R[rt] = M[R[rs] + \text{SignExtImm}]$	(2,7) 30_{hex}
Load Upper Imm.	lui	I	$R[rt] = \{\text{imm}, 16'b0\}$	f_{hex}
Load Word	lw	I	$R[rt] = M[R[rs] + \text{SignExtImm}]$	(2) 23_{hex}
Nor	nor	R	$R[rd] = \sim (R[rs] R[rt])$	$0 / 27_{hex}$
Or	or	R	$R[rd] = R[rs] R[rt]$	$0 / 25_{hex}$
Or Immediate	ori	I	$R[rt] = R[rs] \text{ZeroExtImm}$	(3) d_{hex}
Set Less Than	slt	R	$R[rd] = (R[rs] < R[rt]) ? 1 : 0$	$0 / 2a_{hex}$
Set Less Than Imm.	slti	I	$R[rt] = (R[rs] < \text{SignExtImm}) ? 1 : 0$	(2) a_{hex}
Set Less Than Imm. Unsigned	sltiu	I	$R[rt] = (R[rs] < \text{SignExtImm}) ? 1 : 0$	(2,6) b_{hex}
Set Less Than Unsig.	sltu	R	$R[rd] = (R[rs] < R[rt]) ? 1 : 0$	(6) $0 / 2b_{hex}$
Shift Left Logical	sll	R	$R[rd] = R[rt] \ll \text{shamt}$	$0 / 00_{hex}$
Shift Right Logical	srl	R	$R[rd] = R[rt] \gg \text{shamt}$	$0 / 02_{hex}$
Store Byte	sb	I	$M[R[rs] + \text{SignExtImm}](7:0) = R[rt](7:0)$	(2) 28_{hex}
Store Conditional	sc	I	$M[R[rs] + \text{SignExtImm}] = R[rt];$ $R[rt] = (\text{atomic}) ? 1 : 0$	(2,7) 38_{hex}
Store Halfword	sh	I	$M[R[rs] + \text{SignExtImm}](15:0) = R[rt](15:0)$	(2) 29_{hex}
Store Word	sw	I	$M[R[rs] + \text{SignExtImm}] = R[rt]$	(2) $2b_{hex}$
Subtract	sub	R	$R[rd] = R[rs] - R[rt]$	(1) $0 / 22_{hex}$
Subtract Unsigned	subu	R	$R[rd] = R[rs] - R[rt]$	$0 / 23_{hex}$

MIPS指令集体系结构与汇编语言入门

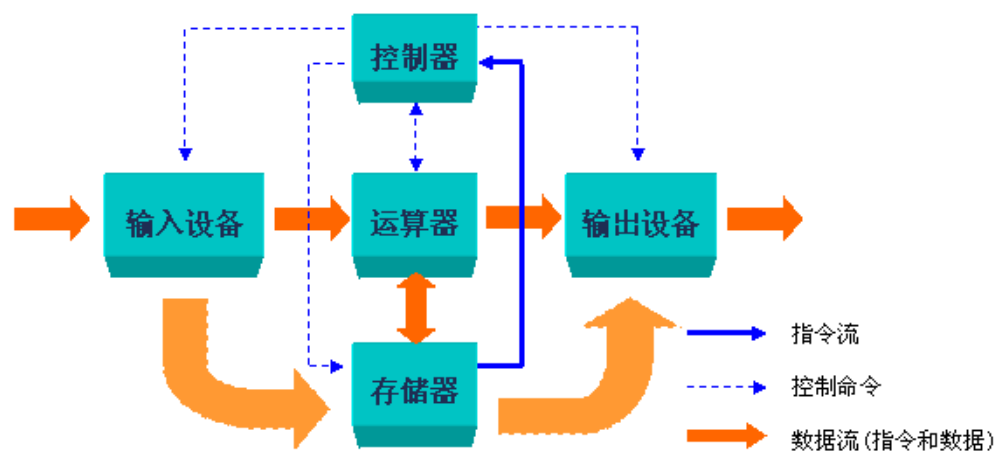
- 1. MIPS中的操作数
- 2. 指令在计算机内部的表示
- 3. 关于存储程序
 - 3.1 程序兼容性
 - 3.2 冯诺依曼架构
 - 3.3 哈佛架构
 - 3.4 混合式架构
- 4. 逻辑运算指令
- 5. 决策指令

3. 关于存储程序

1945年6月，冯诺依曼提交了著名的“关于EDVAC的报告草案”，提出“存储程序”的思想，定义EDVAC为五个部分：运算单元、控制单元、存储单元、输入单元、输出单元



指令与数据一样，都用二进制表示，都存储在内存里
程序可以生成程序（比如编译器）



3. 关于存储程序--3.1 程序兼容性

二进制兼容：编译后的程序可以在相同ISA的其他处理器上运行

(相同的二进制代码，无需重新编译)

源代码兼容：相同的源代码，编译后可以在其他处理器上运行

3. 关于存储程序--*向下兼容与向上兼容

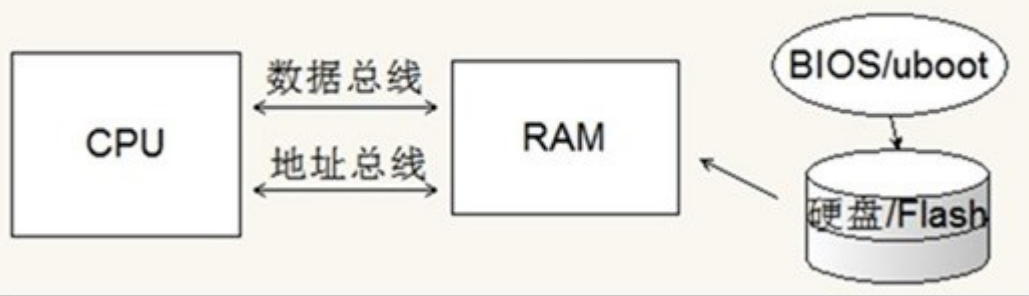
Backward compatibility is a property of a system, product, or technology that allows for interoperability with an older legacy system, or with input designed for such a system.

有什么缺点?

Forward compatibility is a design characteristic that allows a system to accept input intended for a later version of itself.

3. 关于存储程序--3.2 冯诺依曼架构

冯诺依曼结构



冯.诺伊曼结构：将程序指令存储器和数据存储器合并在一起的计算机架构

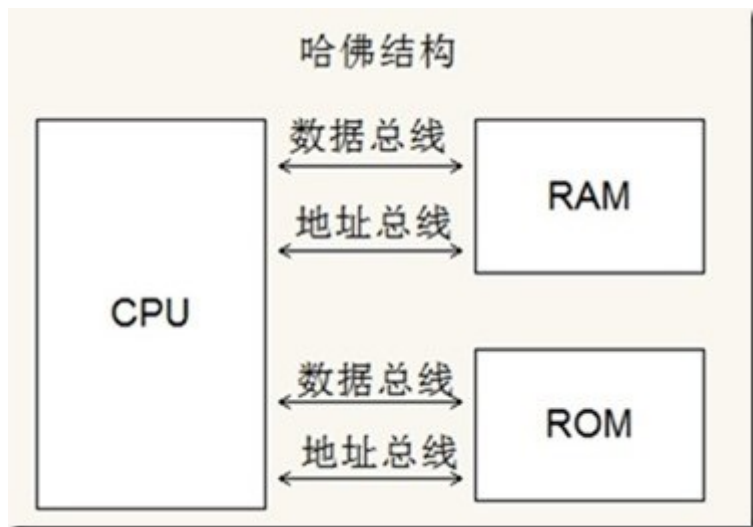
优点:

1. 指令和数据区大小可调，提高存储器利用率；
2. 可以把指令当作数据处理，便于修改指令值和软件升级；
(双刃剑：由于程序与数据具有一样的读写权限，所以出BUG时很容易死机)
3. 总线和控制简单，成本低；
4. 外设要求低（只需一个存储器和相应的总线）

劣势:

1. 不便于流水线，降低CPU效率：读指令时不能操作数据，操作数据时不能读指令；
2. 指令和数据的宽度必须相同；

3. 关于存储程序--3.3 哈佛架构



优点:

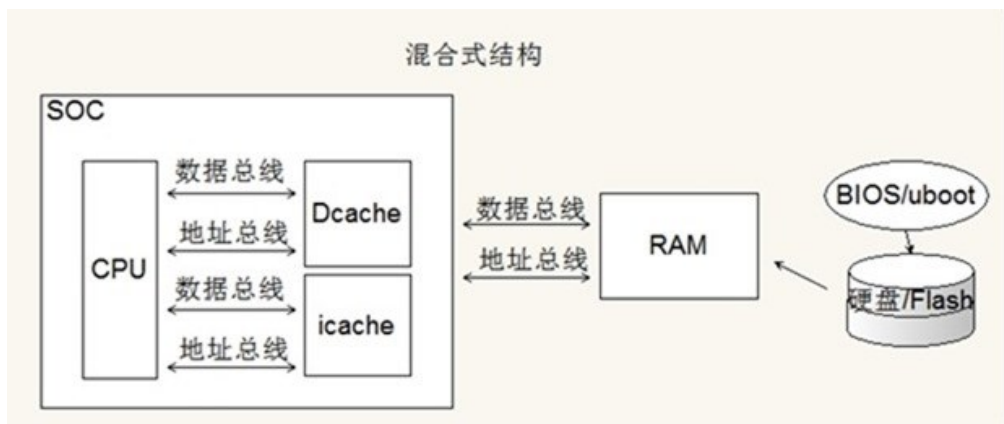
1. 便于流水线, 操作数据的同时可进行取指; (按顺序执行时效果最好, 如果程序跳来跳去, 那也没什么好处, 因此哈佛架构适合任务单调, 而需要高速执行的CPU)
2. 指令和数据不会互相干扰, 程序出bug时还能够顺序执行;

劣势:

1. 难以修改指令, 不便于软件升级;
2. 存储器利用率低;
3. 总线多, 结构复杂, 成本高;
4. 外设要求高, 不便于外围存储扩展;

适合单片机、DSP等嵌入式系统

3. 关于存储程序--3.4 混合式架构



内部采用哈佛，外部采用冯诺依曼的混合式架构

这种结构就是目前ARM的结构，将两种结构**扬其长，避其短**。其中，芯片内部的cache，表示高速缓存。Dcache用来缓存部分代码，icache用来缓存部分数据。只有需要改变时，cache才会到RAM中加载新的数据。所以大部分时间CPU都是通过哈佛结构和cache（高速缓存）通讯，这个速度是非常快的~~

这样在芯片外部，利用冯诺依曼结构，节省了外部的PCB走线资源。

3. 关于存储程序

思考：冯诺依曼架构和哈佛架构（或混合架构）与指令集架构有什么关系？

3. 关于存储程序--3.4 程序存储方式与ISA的关系

理论上将，任何指令集都可以采用冯诺依曼或哈佛结构实现

但它们又有一定的关联性（考虑流水线的实现、性能优化）

指令集架构

内核（物理实现）

x86, x64

Intels、AMDs

ARMv1

ARM1

ARMv2

ARM2

ARMv3

ARM6系列

ARMv4

ARM7系列、部分ARM9

ARMv5

部分ARM9 ARM10系列

ARMv6

ARM11系列

ARMv7

Cortex系列(A,R,M)

MIPS32

MIPS M14K, 龙芯系列, OpenMIPS

大多数DSP

冯诺依曼

哈佛架构/混合式

- 内核决定了与Cache的接口，从而决定是否区分指令Cache和数据Cache

实际上，一种ISA在设计的时候就必须考虑程序存储的影响，设计出的指令集也必然有一定的倾向性

MIPS指令集体系结构与汇编语言入门

- 1. MIPS中的操作数
- 2. 指令在计算机内部的表示
- 3. 关于存储程序
- 4. 逻辑运算指令
 - 4.1 位移指令
 - 4.2 按位与
 - 4.3 按位或
 - 4.4 按位取反
- 5. 决策指令

4. 逻辑运算指令

Operation	C	Java	MIPS
Shift left	<<	<<	sll
Shift right	>>	>>>	srl
Bitwise AND	&	&	and, andi
Bitwise OR			or, ori
Bitwise NOT	~	~	nor

可用于对数据的位操作

CORE INSTRUCTION SET			FOR-MAT	OPERATION (in Verilog)	OPCODE / FUNCT (Hex)
Add	add	R	R[rd] = R[rs] + R[rt]	(1)	0 / 20 _{hex}
Add Immediate	addi	I	R[rt] = R[rs] + SignExtImm	(1,2)	8 _{hex}
Add Imm. Unsigned	addiu	I	R[rt] = R[rs] + SignExtImm	(2)	9 _{hex}
Add Unsigned	addu	R	R[rd] = R[rs] + R[rt]		0 / 21 _{hex}
And	and	R	R[rd] = R[rs] & R[rt]		0 / 24 _{hex}
And Immediate	andi	I	R[rt] = R[rs] & ZeroExtImm	(3)	c _{hex}
Branch On Equal	beq	I	if(R[rs]==R[rt]) PC=PC+4+BranchAddr	(4)	4 _{hex}
Branch On Not Equal	bne	I	if(R[rs]!=R[rt]) PC=PC+4+BranchAddr	(4)	5 _{hex}
Jump	j	J	PC=JumpAddr	(5)	2 _{hex}
Jump And Link	jal	J	R[31]=PC+8;PC=JumpAddr	(5)	3 _{hex}
Jump Register	jr	R	PC=R[rs]		0 / 08 _{hex}
Load Byte Unsigned	lbu	I	R[rt]={24'b0,M[R[rs] +SignExtImm](7:0)}	(2)	24 _{hex}
Load Halfword Unsigned	lhu	I	R[rt]={16'b0,M[R[rs] +SignExtImm](15:0)}	(2)	25 _{hex}
Load Linked	ll	I	R[rt] = M[R[rs]+SignExtImm]	(2,7)	30 _{hex}
Load Upper Imm.	lui	I	R[rt] = {imm, 16'b0}		f _{hex}
Load Word	lw	I	R[rt] = M[R[rs]+SignExtImm]	(2)	23 _{hex}
Nor	nor	R	R[rd] = ~(R[rs] R[rt])		0 / 27 _{hex}
Or	or	R	R[rd] = R[rs] R[rt]		0 / 25 _{hex}
Or Immediate	ori	I	R[rt] = R[rs] ZeroExtImm	(3)	d _{hex}
Set Less Than	slt	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0		0 / 2a _{hex}
Set Less Than Imm.	slti	I	R[rt] = (R[rs] < SignExtImm) ? 1 : 0	(2)	a _{hex}
Set Less Than Imm. Unsigned	sltiu	I	R[rt] = (R[rs] < SignExtImm) ? 1 : 0	(2,6)	b _{hex}
Set Less Than Unsig.	sltu	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0	(6)	0 / 2b _{hex}
Shift Left Logical	sll	R	R[rd] = R[rt] << shamt		0 / 00 _{hex}
Shift Right Logical	srl	R	R[rd] = R[rt] >>> shamt		0 / 02 _{hex}
Store Byte	sb	I	M[R[rs]+SignExtImm](7:0) = R[rt](7:0)	(2)	28 _{hex}
Store Conditional	sc	I	M[R[rs]+SignExtImm] = R[rt]; R[rt] = (atomic) ? 1 : 0	(2,7)	38 _{hex}
Store Halfword	sh	I	M[R[rs]+SignExtImm](15:0) = R[rt](15:0)	(2)	29 _{hex}
Store Word	sw	I	M[R[rs]+SignExtImm] = R[rt]	(2)	2b _{hex}
Subtract	sub	R	R[rd] = R[rs] - R[rt]	(1)	0 / 22 _{hex}
Subtract Unsigned	subu	R	R[rd] = R[rs] - R[rt]		0 / 23 _{hex}

4. 逻辑运算指令--4.1位移指令

sll: shift left logical 左移并补0

srl: shift right logical 右移并补0

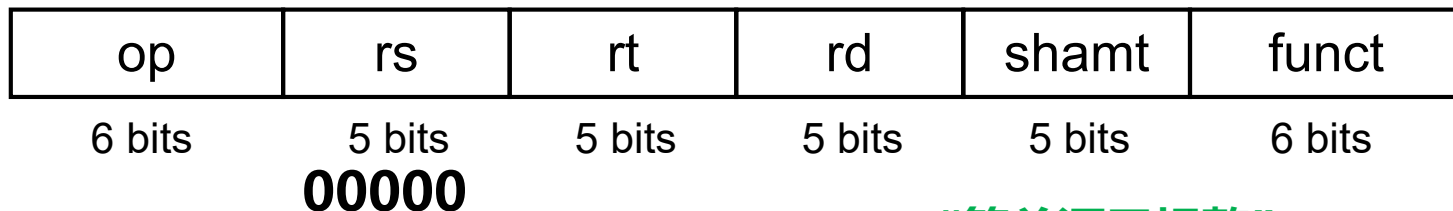
指令格式: **sll/srl rd, rt, shamt**

shamt: 左/右移的位数

例: **sll \$t2, \$s0, 4** #reg \$t2=reg \$s0<<4bits

思考: 能不能将位移指令“塞”进R型指令格式?

可以, 位移指令属于R型:



-- “简单源于规整”

4. 逻辑运算指令--4.2按位与

and: 按位与

指令格式: `and rd, rs, rt` (R型)

andi: 立即数按位与

指令格式: `andi rt, rs, immediate` (I型)

可用于对字的位操作, 如:

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

4. 逻辑运算指令--4.3按位或

or: 按位或 指令格式: or rd, rs, rt (R型)

ori: 立即数按位与 指令格式: ori rt, rs, immediate (I型)

可用于对字的位操作, 如:

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

4. 逻辑运算指令--4.4按位取反

是否需要设置指令NOT?

由于: $a \text{ NOR } b == \text{NOT} (a \text{ OR } b)$ 所以, 可用NOR实现NOT

指令格式: `nor rd, rs, rt` (R型)

如: `nor $t0, $t1, $zero`

\$t1	0000 0000 0000 0000 0011 1100 0000 0000
------	---

\$t0	1111 1111 1111 1111 1100 0011 1111 1111
------	---

Nor的主要功能就是按位取反, 常数在nor中用得很少, 所以没有 “nori”

-- “简单源于规整”

MIPS指令集体系结构与汇编语言入门

1. MIPS中的操作数
2. 指令在计算机内部的表示
3. 关于存储程序
4. 逻辑运算指令
5. 决策指令

5. 决策指令

计算机有别于计算器的关键----分支

beq rs, rt, L1 #如果\$rs==\$rt, 则跳转至L1处 (branch on equal)

bne rs, rt, L1 #如果\$rs!=\$rt, 则跳转至L1处 (branch on not equal)

指令格式: beq rs, rt, immediate } 跳转至?: 若为直接寻址, 则范围很小
 bne rs, rt, immediate }

PC相对寻址:

PC+4+(符号位扩展且左移两位的立即数)

->加速大概率事件!

分支范围: 相对下一条指令的 $\pm 2^{15}$ 个字, 可满足几乎所有的循环和if语句跳转

MIPS对所有条件分支都使用PC相对寻址, 可以跳转到比较近的分支地址

跳得远, 超过 $\pm 2^{15}$ (相对PC+4) 怎么办?

5. 决策指令

j L1 #无条件跳转至L1处

j address

R型:

op(6)	rs(5)	rt(5)	rd(5)	shamt(5)	funct(6)
-------	-------	-------	-------	----------	----------

I型:

op(6)	rs(5)	rt(5)	constant or address(16)
-------	-------	-------	-------------------------

J型:

op(6)	address(26)
-------	-------------

高4位

中间26位

低2位

保持不变

替换

00

寻址范围: $0 \sim 2^{26}$ 个字 (256MB)

J型指令: j、jal

跳转超过256MB范围怎么办?

-> 地址放在寄存器中

5. 决策指令

思考：如果分支跳转超过PC相对寻址范围怎么办？

beq \$s0, \$s1, L1



bne \$s0, \$s1, L2

j L1

L2: ...



“j” 指令完成跳转

5. 决策指令—*MIPS寻址方式总结

MIPS提供了5种寻址方式：

1、立即数寻址

操作数是指令中的立即数

2、寄存器寻址

操作数是寄存器中数据

3、基址寻址

立即数+寄存器值

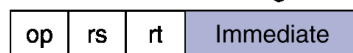
4、PC相对寻址

地址是 $PC + 4 +$ 左移的16位立即数

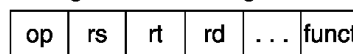
5、伪直接寻址

地址为PC高位+左移的26位立即数

1. Immediate addressing



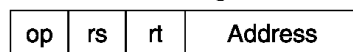
2. Register addressing



Registers

Register

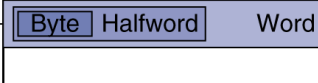
3. Base addressing



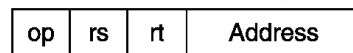
Memory

Register

+



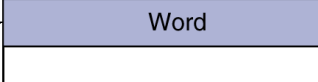
4. PC-relative addressing



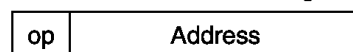
Memory

PC

+



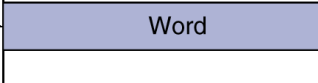
5. Pseudodirect addressing



Memory

PC

:



5. 决策指令

例：C 代码：

if (i==j)f = g+h;

else f = g-h;

f, g, ..., j分别保存在\$**s0**, \$**s1**, ...\$

编译后的 MIPS 代码：

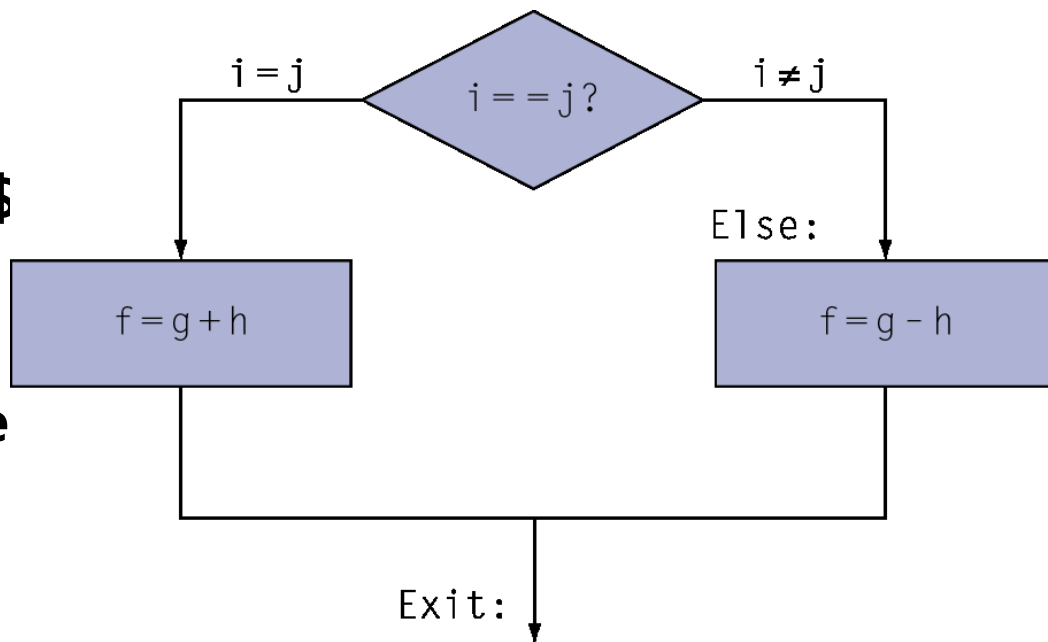
bne \$s3, \$s4, Else

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

j Exit

Else: sub \$s0, \$s1, \$s2

Exit: ...



5. 决策指令

例：C 代码：

while (save[i] == k) i += 1;

i 保存在\$*s3*, k 保存在\$*s5*, save的基址保存在\$*s6*

编译后的MIPS代码：

```
Loop: sll $t1, $s3, 2      #计算4*i  
add $t1, $t1, $s6    #$t1: save[i]的位置  
lw $t0, 0($t1)      #$t0: save[i]的值  
bne $t0, $s5, Exit  
  
addi $s3, $s3, 1      #i=i+1  
j Loop
```

Exit: ...

5. 决策指令

扩展：如果Loop的位置在内存80000处，那么这段代码对应的机器码是？

编译后的MIPS代码：

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

add \$t1, \$t1, \$s6

lw \$t0, 0(\$t1)

bne \$t0, \$s5, Exit

addi \$s3, \$s3, 1

j Loop

Exit: ...

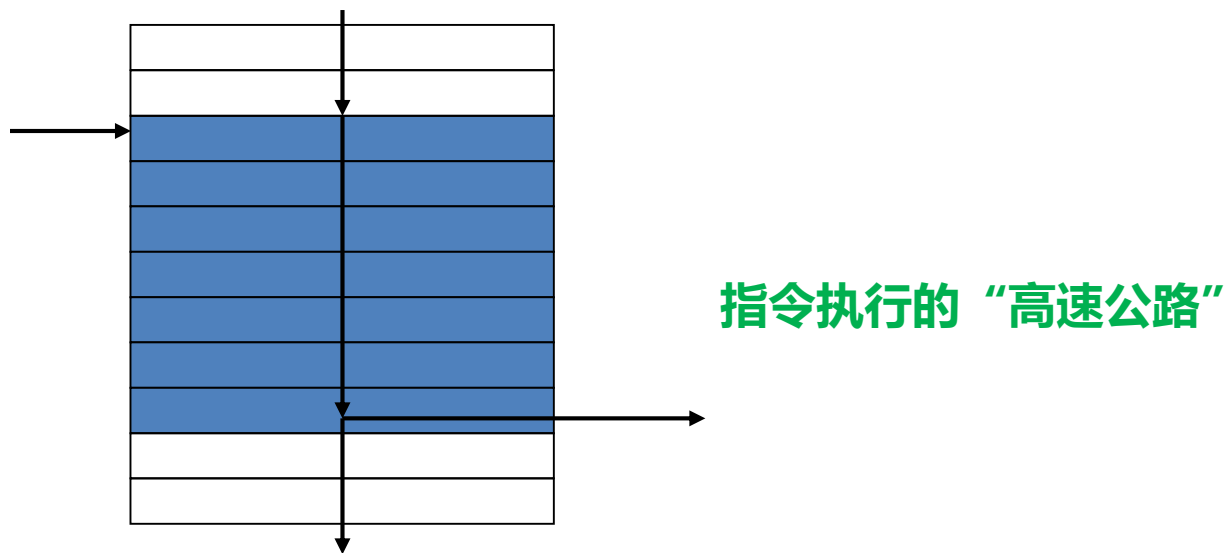
80000	0	0	19	9	2	0
80004	0	9	22	9	0	32
80008	35	9	8	0		
80012	5	8	21	2		
80016	8	19	19	1		
80020	2	20000				
80024	...					

5. 决策指令--基本块

基本块：不包含分支和标签的指令序列（头部有标签；尾部是分支指令）

编译器首先要将程序划分为若干基本块

先进的处理器可以对基本块的执行进行加速****



5. 决策指令--更多决策指令

slt: set on less than (如果小于成立, 则将结果置1, 否则置0)

例: slt rd, rs, rt #如果 $rs < rt$, 则 $rd=1$; 否则 $rd=0$

slti: set on less than immediate

例: slt rt, rs, constant #如果 $rs < \text{constant}$, 则 $rt=1$; 否则 $rt=0$

与beq、bne结合, 用于实现判定 $=$ 、 \neq 、 $<$ 、 \leq 、 $>$ 、 \geq :

slt \$t0, \$s1, \$s2 # if ($\$s1 < \$s2$)

bne \$t0, \$zero, L # branch to L

简单性原则: MIPS没有提供 “blt” 、 “bge” 指令 ->好的折中设计

5. 决策指令--更多决策指令

对于无符号数，有：sltu、sltiu

例：

\$s0 = 1111 1111 1111 1111 1111 1111 1111 1111

\$s1 = 0000 0000 0000 0000 0000 0000 0000 0001

slt \$t0, \$s0, \$s1 # signed

执行结果：-1 < +1 \$t0 = 1

sltu \$t0, \$s0, \$s1 # unsigned

执行结果：+4,294,967,295 > +1 \$t0 = 0

小结

MIPS指令集体系结构与汇编语言

1. MIPS中的操作数
2. 指令在计算机内部的表示
3. 关于存储程序
4. 逻辑运算指令
5. 决策指令

课后作业

**画出MIPS处理器中R型I型和J型指令的机器码格式，
并说明每个字段的含义和用途。**