处理器体系结构

指令集体系结构

(Micro-processor Architecture)

内容概述

ISA简介

1. 处理器指令集体系结构(ISA)及其重要性
2. 从CISC到RISC

RISC-V 指令集体系结构与汇编语言入门

1. RISC-V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

硬件对过程的支持

1. 指令和寄存器
2. 叶过程(leaf procedures)
3. 嵌套过程(non-leaf procedures)

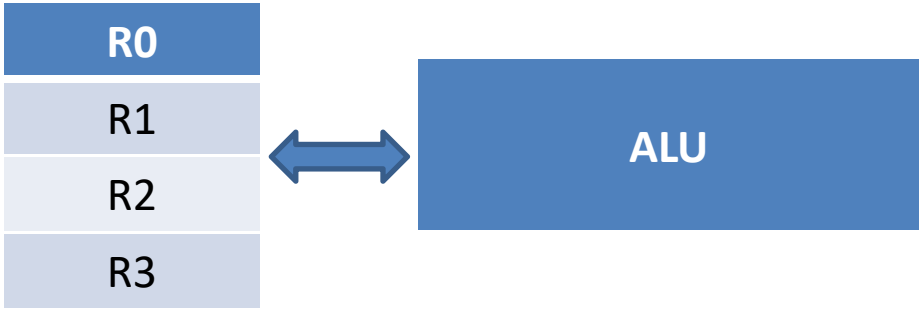
什么是ISA?

处理器如何理解人类意图？如何作运算？　　C=A+B

Memory

地址	#0	#1	#2	#3	#4	#5	…
数值	A	B	C	…	…	…	…

Registers



load R3, #0;　　从内存地址0处取A值，放入R3寄存器

load R2,#1;　　从内存地址1处取B值，放入R2寄存器

add R0, R3, R2;　　把R3和R2相加，放入R0寄存器

Store R0, #2;　　将R0中的值存入内存#2位置

什么是ISA?

load R3, #0;

load R2,#1;

add R0, R3, R2;

Store R0, #2;

操作码：

Load	Add	Store	…
00	01	10	11

寄存器：

R0	R1	R2	R3
00	01	10	11

定义add指令对应的机器码格式：

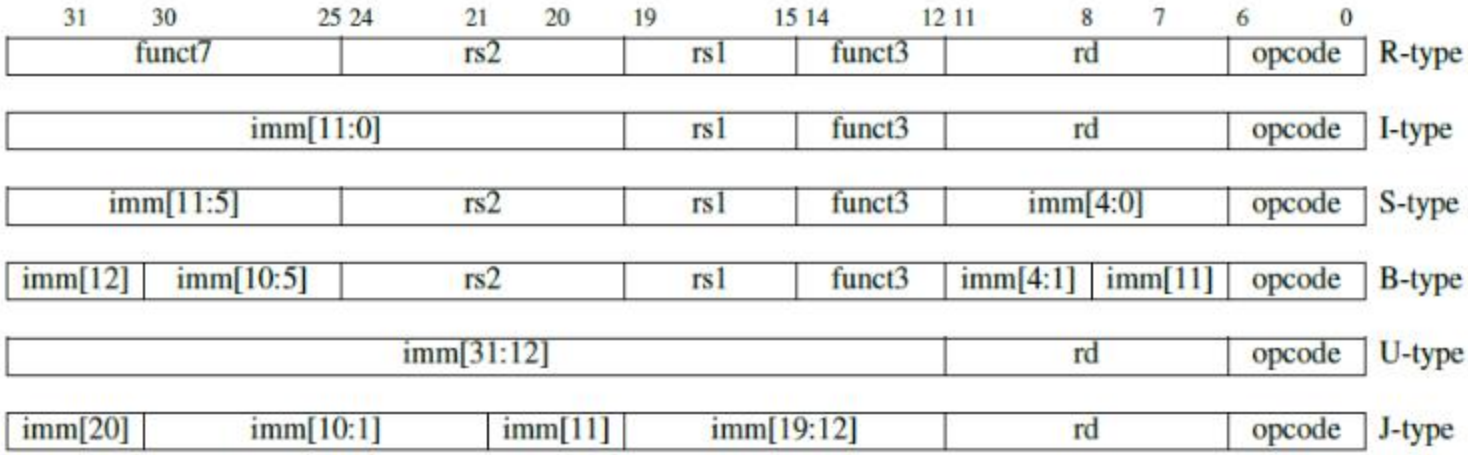
Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
操作码		第1个操作数		第2个操作数		第3个操作数	

Add R0, R3, R2 -> 01001110

一个处理器**支持的指令**和**指令的字节级编码**就是这个处理器的ISA：

指令集、指令集对应的编码、数据类型、寄存器、寻址方式、存储体系…

RISC-V 各类指令



操作码与函数码为什么不合并?

RISC-V 各类指令

RISC-V assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	Add	add x5, x6, x7	x5 = x6 + x7	Three register operands; add
	Subtract	sub x5, x6, x7	x5 = x6 - x7	Three register operands; subtract
	Add immediate	addi x5, x6, 20	x5 = x6 + 20	Used to add constants
Data transfer	Load doubleword	ld x5, 40(x6)	x5 = Memory[x6 + 40]	Doubleword from memory to register
	Store doubleword	sd x5, 40(x6)	Memory[x6 + 40] = x5	Doubleword from register to memory
	Load word	lw x5, 40(x6)	x5 = Memory[x6 + 40]	Word from memory to register
	Load word, unsigned	lwu x5, 40(x6)	x5 = Memory[x6 + 40]	Unsigned word from memory to register
	Store word	sw x5, 40(x6)	Memory[x6 + 40] = x5	Word from register to memory
	Load halfword	lh x5, 40(x6)	x5 = Memory[x6 + 40]	Halfword from memory to register
	Load halfword, unsigned	lhu x5, 40(x6)	x5 = Memory[x6 + 40]	Unsigned halfword from memory to register
	Store halfword	sh x5, 40(x6)	Memory[x6 + 40] = x5	Halfword from register to memory
	Load byte	lb x5, 40(x6)	x5 = Memory[x6 + 40]	Byte from memory to register
	Load byte, unsigned	lbu x5, 40(x6)	x5 = Memory[x6 + 40]	Byte unsigned from memory to register
	Store byte	sb x5, 40(x6)	Memory[x6 + 40] = x5	Byte from register to memory
	Load reserved	lr.d x5, (x6)	x5 = Memory[x6]	Load; 1st half of atomic swap
	Store conditional	sc.d x7, x5, (x6)	Memory[x6] = x5; x7 = 0/1	Store; 2nd half of atomic swap
	Load upper immediate	lui x5, 0x12345	x5 = 0x12345000	Loads 20-bit constant shifted left 12 bits
Logical	And	and x5, x6, x7	x5 = x6 & x7	Three reg. operands; bit-by-bit AND
	Inclusive or	or x5, x6, x8	x5 = x6 | x8	Three reg. operands; bit-by-bit OR
	Exclusive or	xor x5, x6, x9	x5 = x6 ^ x9	Three reg. operands; bit-by-bit XOR
	And immediate	andi x5, x6, 20	x5 = x6 & 20	Bit-by-bit AND reg. with constant
	Inclusive or immediate	ori x5, x6, 20	x5 = x6 | 20	Bit-by-bit OR reg. with constant
	Exclusive or immediate	xori x5, x6, 20	x5 = x6 ^ 20	Bit-by-bit XOR reg. with constant
Shift	Shift left logical	sll x5, x6, x7	x5 = x6 << x7	Shift left by register
	Shift right logical	srl x5, x6, x7	x5 = x6 >> x7	Shift right by register
	Shift right arithmetic	sra x5, x6, x7	x5 = x6 >> x7	Arithmetic shift right by register
	Shift left logical immediate	slli x5, x6, 3	x5 = x6 << 3	Shift left by immediate
	Shift right logical immediate	srli x5, x6, 3	x5 = x6 >> 3	Shift right by immediate
	Shift right arithmetic immediate	srai x5, x6, 3	x5 = x6 >> 3	Arithmetic shift right by immediate
Conditional branch	Branch if equal	beq x5, x6, 100	if (x5 == x6) go to PC+100	PC-relative branch if registers equal
	Branch if not equal	bne x5, x6, 100	if (x5 != x6) go to PC+100	PC-relative branch if registers not equal
	Branch if less than	blt x5, x6, 100	if (x5 < x6) go to PC+100	PC-relative branch if registers less
	Branch if greater or equal	bge x5, x6, 100	if (x5 >= x6) go to PC+100	PC-relative branch if registers greater or equal
	Branch if less, unsigned	bltu x5, x6, 100	if (x5 < x6) go to PC+100	PC-relative branch if registers less, unsigned
	Branch if greater or equal, unsigned	bgeu x5, x6, 100	if (x5 >= x6) go to PC+100	PC-relative branch if registers greater or equal, unsigned
Unconditional branch	Jump and link	jal x1, 100	x1 = PC+4; go to PC+100	PC-relative procedure call
	Jump and link register	jalr x1, 100(x5)	x1 = PC+4; go to x5+100	Procedure return; indirect call

RISC-V 寄存器

寄存器地址	寄存器名称	名称含义	用途
x0	zero	Zero	常量0
x1	ra	Return address	函数返回地址
x2	sp	Stack pointer	栈指针
x3	gp	Global pointer	全局指针
x4	tp	Thread pointer	线指针
x5-7	t0-2	Temporaries	存放临时变量(随便用的)
x8	s0/fp	Saved values	保存变量(子函数调用前后)/帧指针
x9	s1	Saved values	保存变量(子函数调用前后)
x10-11	a0-1	Arguments/return values	函数参数值/返回值
x12-17	a2-7	Function arguments	函数参数值
x18-27	s2-11	Saved values	保存变量(子函数调用前后)
x28-31	t3-6	Temporaries	存放临时变量(随便用的)

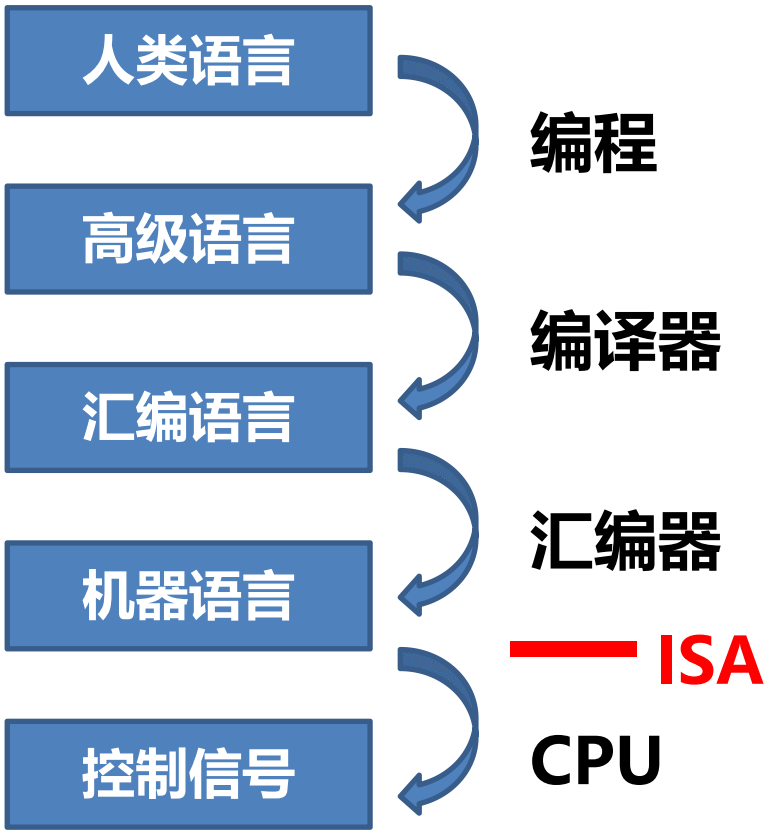
为什么是32个?

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



ISA的发展 — 从CISC到RISC

CISC: Complex Instruction Set Computer

RISC: Reduced Instruction Set Computer

“存在即是合理” — 黑格尔

CISC存在的理由: (粗放式扩张)

早期，人们使用汇编语言编程，强大的CISC指令集**便于编程**；

早期，存储器昂贵而缓慢，变长编码可以**节约存储**，**减少内存访问**。

RISC存在的理由: (量变引起质变)

70年代，人们逐渐认识到CISC指令集的**“二八定理”**

精简、定长的RISC指令可以改善**处理器频率、流水线设计、指令译码**

内容概述

ISA简介

1. 处理器指令集体系结构(ISA)及其重要性
2. 从CISC到RISC

RISC-V 指令集体系结构与汇编语言入门

1. RISC-V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

硬件对过程的支持

1. 指令和寄存器
2. 叶过程(leaf procedures)
3. 嵌套过程(non-leaf procedures)

内容概述

RISC-V指令集体系结构与汇编语言入门

1. **RISC-V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**

1. 寄存器操作数
2. 存储器操作数
3. 立即数操作数

1. RISC-V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. RISC-V中的操作数--1.1寄存器操作数

RISC-V算术运算指令的操作数必须直接取自寄存器！

RISC-V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
x0	zero	Zero	常量0
x1	ra	Return address	函数返回地址
x2	sp	Stack pointer	栈指针
x3	gp	Global pointer	全局指针
x4	tp	Thread pointer	线指针
x5-7	t0-2	Temporaries	存放临时变量(随便用的)
x8	s0/fp	Saved values	保存变量(子函数调用前后)/帧指针
x9	s1	Saved values	保存变量(子函数调用前后)
x10-11	a0-1	Arguments/return values	函数参数值/返回值
x12-17	a2-7	Function arguments	函数参数值
x18-27	s2-11	Saved values	保存变量(子函数调用前后)
x28-31	t3-6	Temporaries	存放临时变量(随便用的)

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. RISC-V中的操作数--1.1寄存器操作数

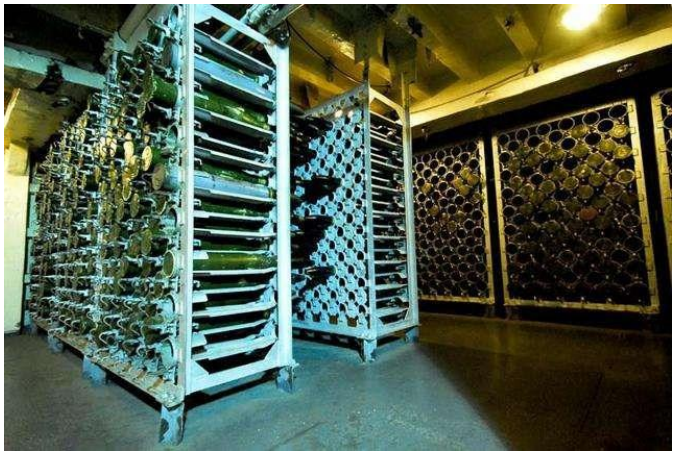
刚刚的例子：**f=(g+h)-(i+j);**

假设：**f~j分别分配给s0~s4，则RISC-V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. RISC-V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. RISC-V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

1. RISC-V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RISC-V代码为：

RISC-V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RISC-V体系架构下，最多有多大地址空间？

2^32=4GB (Giga Byte)

1. RISC-V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RISC-V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. RISC-V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**

内容概述

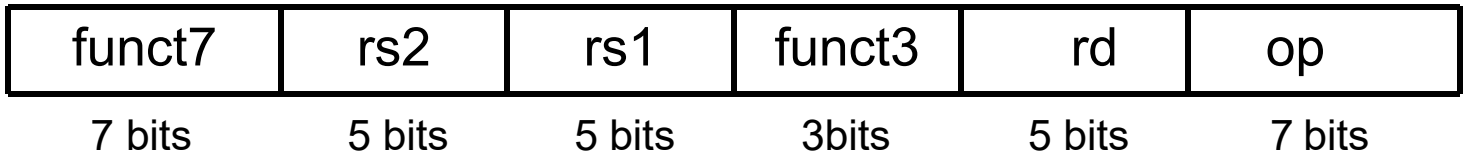
RISC-V指令集体系结构与汇编语言入门

1. RISC-V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

1. R型指令
2. I型指令
3. S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: operation code (opcode) 操作码

rs1: first source register number 第一个源操作数寄器

rs2: second source register number 第二个源操作数寄器

rd: destination register number 目的操作数寄存器

funct: function code 函数码

*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？（为什么采用位操作码+位函数码，而直接17位操作码？ ）

这样做更便于译码和加速！

CORE INSTRUCTION SET

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

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？

funct7	rs2	rs1	funct3	rd	op
7 bits	5 bits	5 bits	5bits	5 bits	7 bits
0	s1	s2	0	t0	add
0	9	18	0	5	51
0000000	01001	10010	000	00101	0110011

$$0000000\,01001\,10010\,000\,00101\,0110011_2 = 009902b3_{16}$$

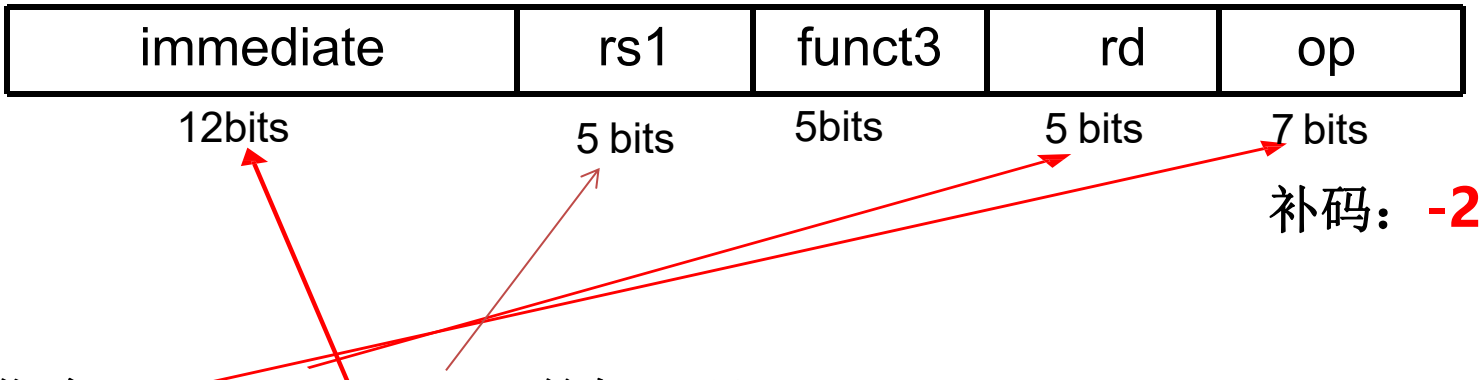
R型指令的格式是否适合取字？ ：lw rt, 4n(rs)

->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

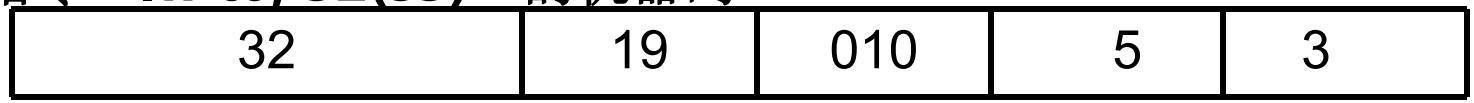
设计原则4：优秀的设计者需要适当的折中

“I”：Immediate，立即数指令



补码：$-2^{12} \sim +2^{12}-1$

例：指令“lw t0, 32(s3)”的机器码？



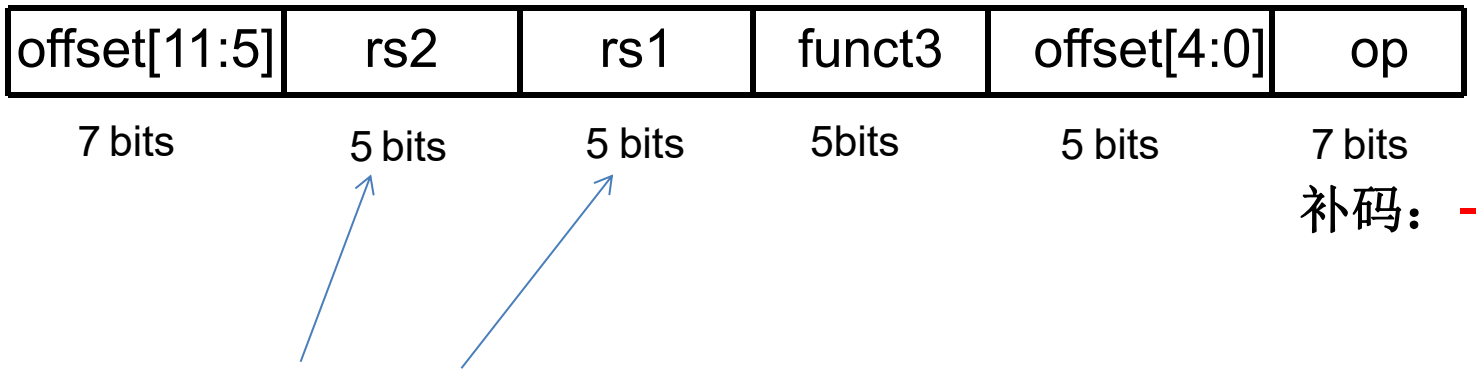
I型指令的折中：保持指令长度相同，而不同类型的指令采用不同格式；不同格 式的指令显然增加了复杂度，所以让格式尽可能的类似

->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--2.3 S型指令

设计原则1：简单源于规整

“I”：Immediate，立即数指令



补码：$-2^{12} \sim +2^{12}-1$

例：指令“**sw t0, 32(s3)**”的机器码？

1	5	18	2	0	67

->处理器可根据op码区分指令I类型，进而作不同处理

内容概述

RISC-V指令集体系结构与汇编语言入门

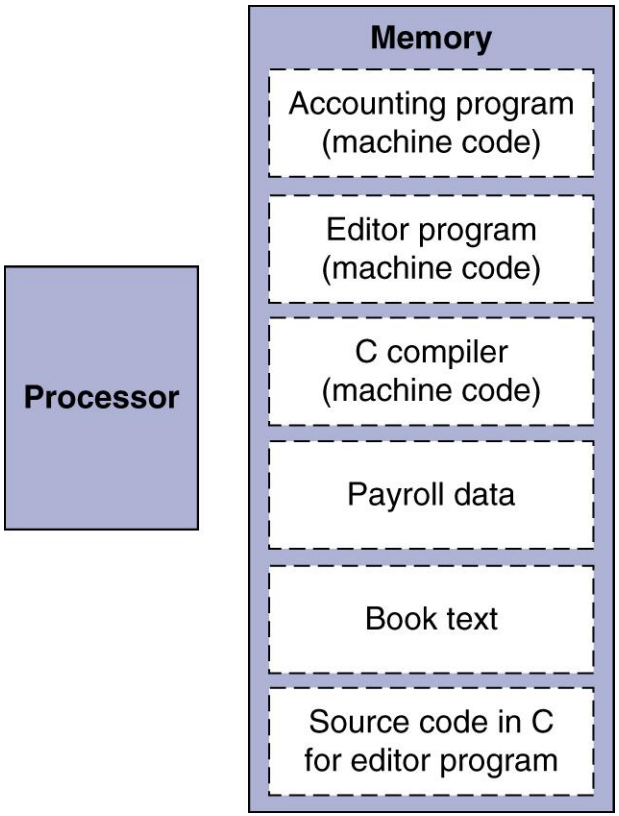
1. RISC-V中的操作数
2. 指令在计算机内部的表示
3. 关于存储程序
4. 逻辑运算指令
5. 决策指令

1. 冯诺依曼架构
2. 哈佛架构
3. 混合式架构

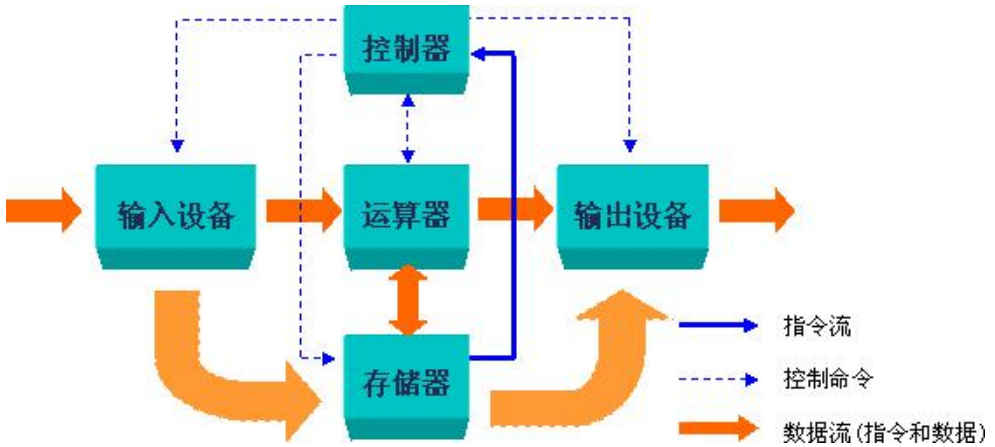
3. 关于存储程序

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

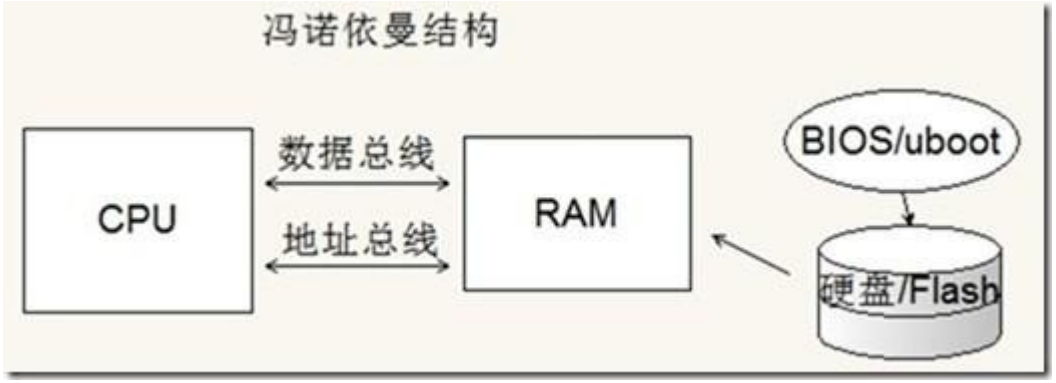
思想，定义EDVAC为五个部分：运算单元、控制单元、存储单元、输入单元、输出单元



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 冯诺依曼架构



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；

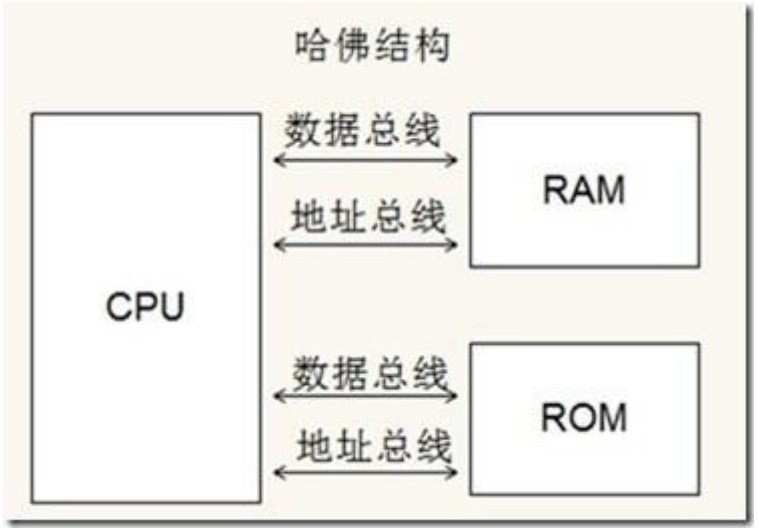
（双刃剑：由于程序与数据具有一样的读写权限，所以出BUG时很容易死机）

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--3.2 哈佛架构



哈佛结构(**Harvard architecture**)：将程序指令储存和数据储存分开的计算机架构

优点：

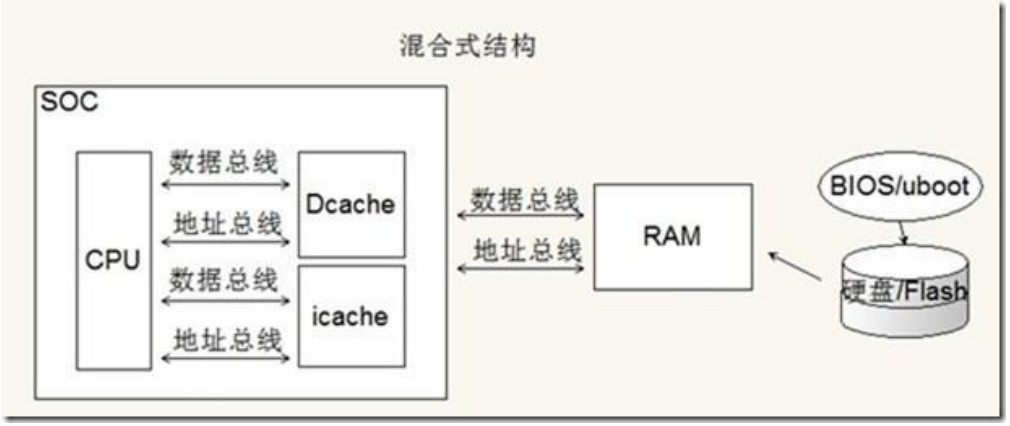
1. 便于流水线，操作数据的同时可进行取指；（按顺序执行时效果最好，如果程序跳来跳去，那也没什么好处，因此哈佛架构适合任务单调，而需要高速执行的CPU）
2. 指令和数据不会互相干扰，程序出bug时还能够顺序执行；

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--3.3 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。

内容概述

RISC-V指令集体系结构与汇编语言入门

1. RISC-V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

1. 位移指令
2. 按位与
3. 按位或
4. 按位取反

4. 逻辑运算指令

Logical operations	C operators	Java operators	RISC-V instructions
Shift left	<<	<<	sll, slli
Shift right	>>	>>>	srl, srli
Shift right arithmetic	>>	>>	sra, srai
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR	|	|	or, ori
Bit-by-bit XOR	^	^	xor, xori
Bit-by-bit NOT	~	~	xori

可用于对数据的位操作

CORE INSTRUCTION SET

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

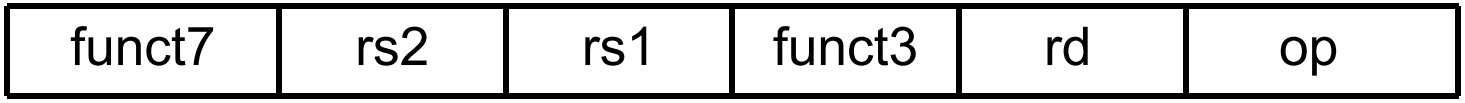
指令格式：sll/srl　rd, rt,　shamt

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”

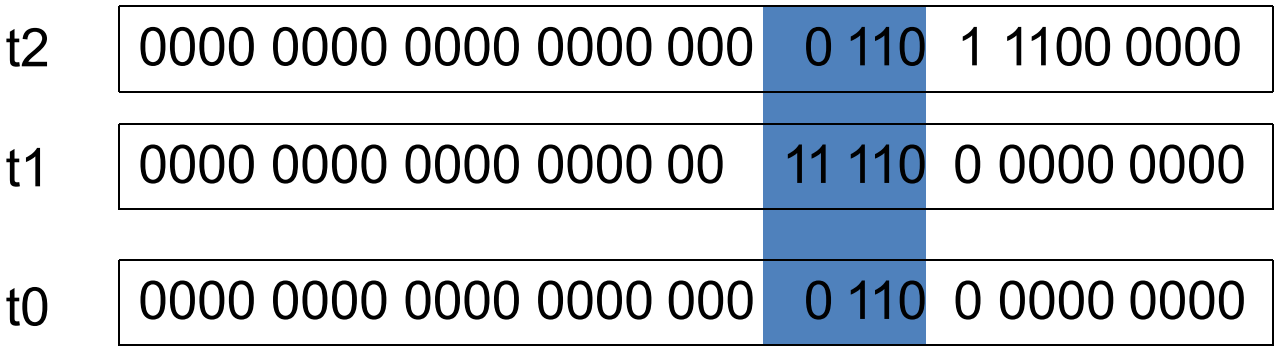
4. 逻辑运算指令--4.2按位与

and：按位与 **andi**：　　指令格式：**and rd, rs, rt　（R型）**

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



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 000	0 110	1 1100 0000
t1	0000 0000 0000 0000 00	11 110	0 0000 0000
t0	0000 0000 0000 0000 00	11 110	1 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 NOR b == NOT (a 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”

内容概述

RISC-V指令集体系结构与汇编语言入门

1. RISC-V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**
bne rs, rt, immediate
} 跳转至?：若为直接寻址，则范围很小

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±2^{15}个字，可满足几乎所有的循环和**if**语句跳转

RISC-V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±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：0~2^{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—*RISC-V寻址方式总结

RISC-V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

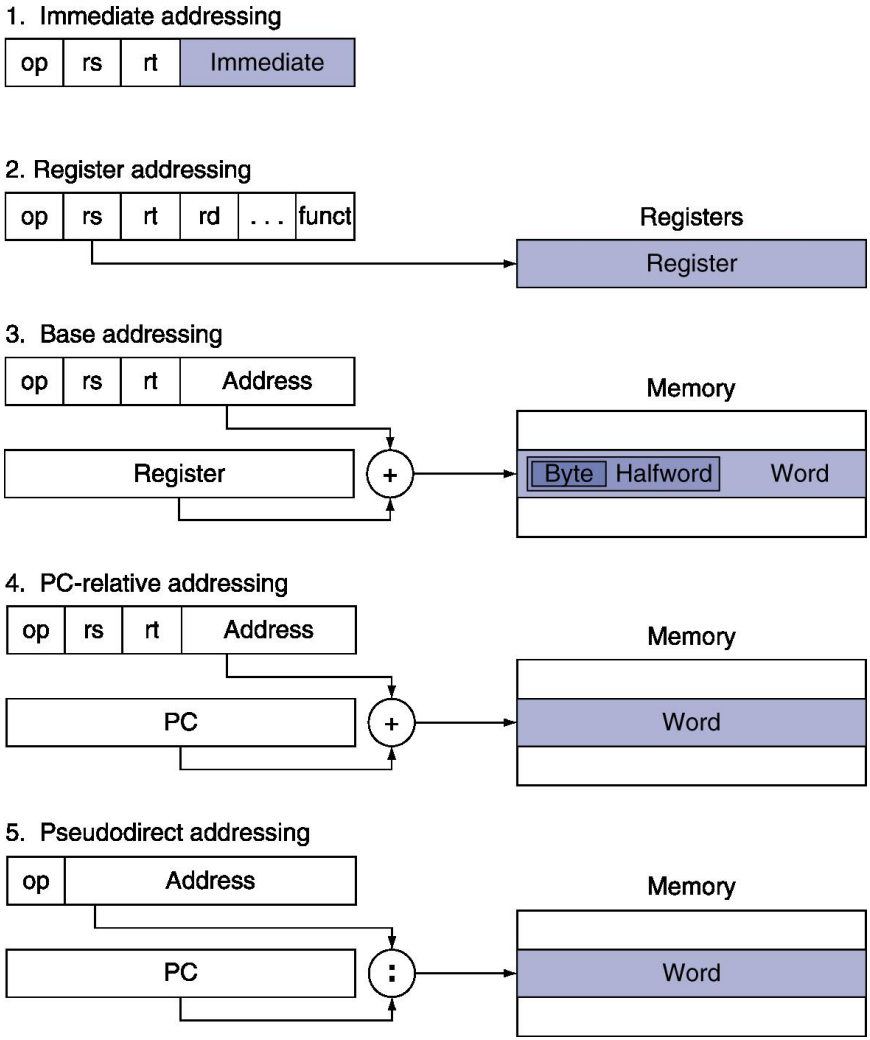
立即数+寄存器值

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



5. 决策指令

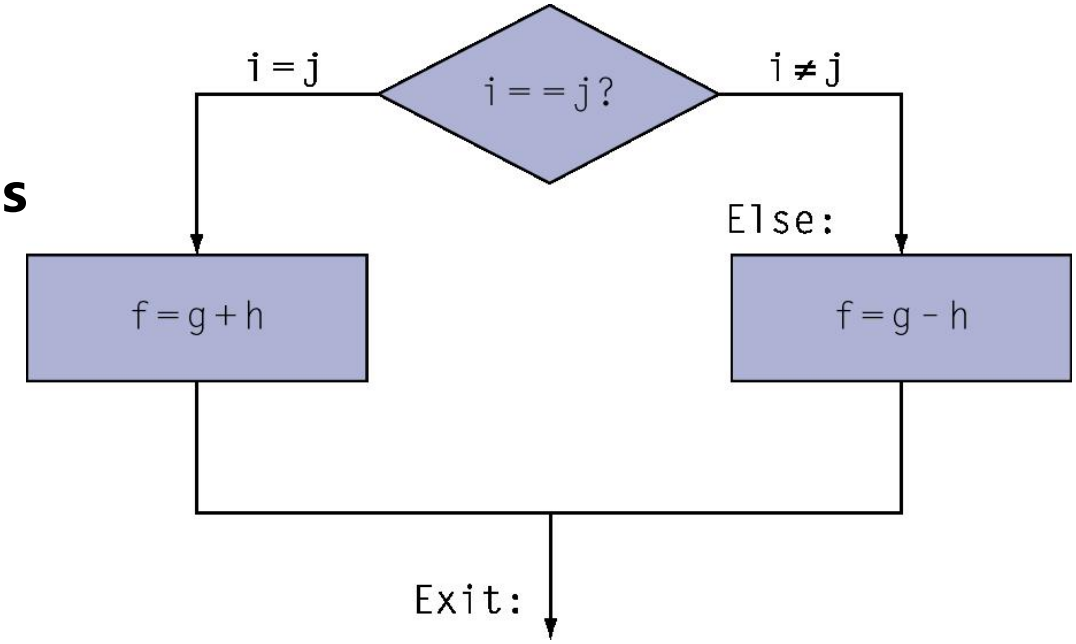
例：C 代码:

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

f, g, .., j分别保存在s0, s1, … s

编译后的 RISC-V 代码:

```
        bne s3, s4, Else
        add s0, s1, s2
        j   Exit
Else:   sub s0, s1, s2
Exit:   …
```



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<constant，则rt=1；否则rt=0**

blt: branch if less than

例：**blt rt, rs, L**

bge: Branch if greater or equal

例：**bge rt,rs,L**

slt,slti与beq、bne结合，用于实现判定==、!=、<、<=、>、>=：

```
slt t0, s1, s2        # if (s1 < s2)
bne t0, zero, L       #    branch to L
```

内容概述

ISA简介

1. 处理器指令集体系结构(ISA)及其重要性
2. 从CISC到RISC

RISC-V 指令集体系结构与汇编语言入门

1. RISC-V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

硬件对过程的支持

1. 指令和寄存器
2. 叶过程(leaf procedures)
3. 嵌套过程(non-leaf procedures)

内容概述

硬件对过程的支持

1. 指令和寄存器
2. 叶过程(leaf procedures)
3. 嵌套过程(non-leaf procedures)
4. 二进制接口规范

硬件对过程的支持

过程：根据提供的参数执行一定任务的子程序　　—实现抽象的一种方法

使用过程的好处：使程序结构化、易懂、易重复使用

1) 将参数放在过程可以访问的位置：参数传递

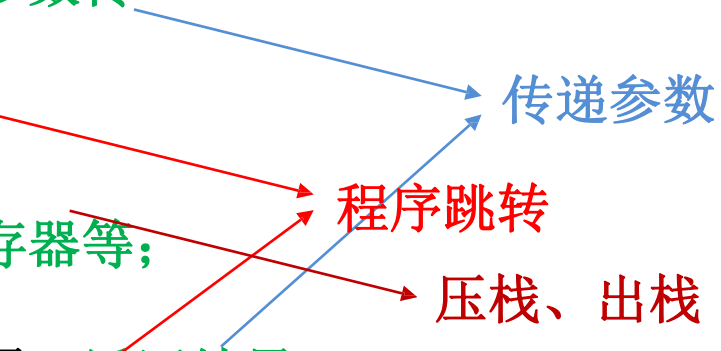
2) 将控制转交给过程：调用子程序；

3) 获得过程所需的存储资源：保护寄存器等；

4) 执行过程任务：执行子程序体；

5) 将结果放在调用程序可以访问的位置：返回结果；

6) 将控制返回初始点：返回主程序。



Caller：过程调用者

Callee：过程被调用者

硬件对过程的支持--1. 指令和寄存器

jal：jump and link，将PC+4的值存入ra，并跳转到指定地址 (子程序入口)

jr：jump register，复制ra到PC，也可用于其它跳转 (比如：转移表)

寄存器地址	寄存器名称	名称含义	用途
x0	zero	Zero	常量0
x1	ra	Return address	函数返回地址
x2	sp	Stack pointer	栈指针
x3	gp	Global pointer	全局指针
x4	tp	Thread pointer	线指针
x5-7	t0-2	Temporaries	存放临时变量(随便用的)
x8	s0/fp	Saved values	保存变量(子函数调用前后)/帧指针
x9	s1	Saved values	保存变量(子函数调用前后)
x10-11	a0-1	Arguments/return values	函数参数值/返回值
x12-17	a2-7	Function arguments	函数参数值
x18-27	s2-11	Saved values	保存变量(子函数调用前后)
x28-31	t3-6	Temporaries	存放临时变量(随便用的)

->栈(stack)

压栈(push)、出栈(pop)

子函数调用/返回的参数 超过寄存器数怎么办？

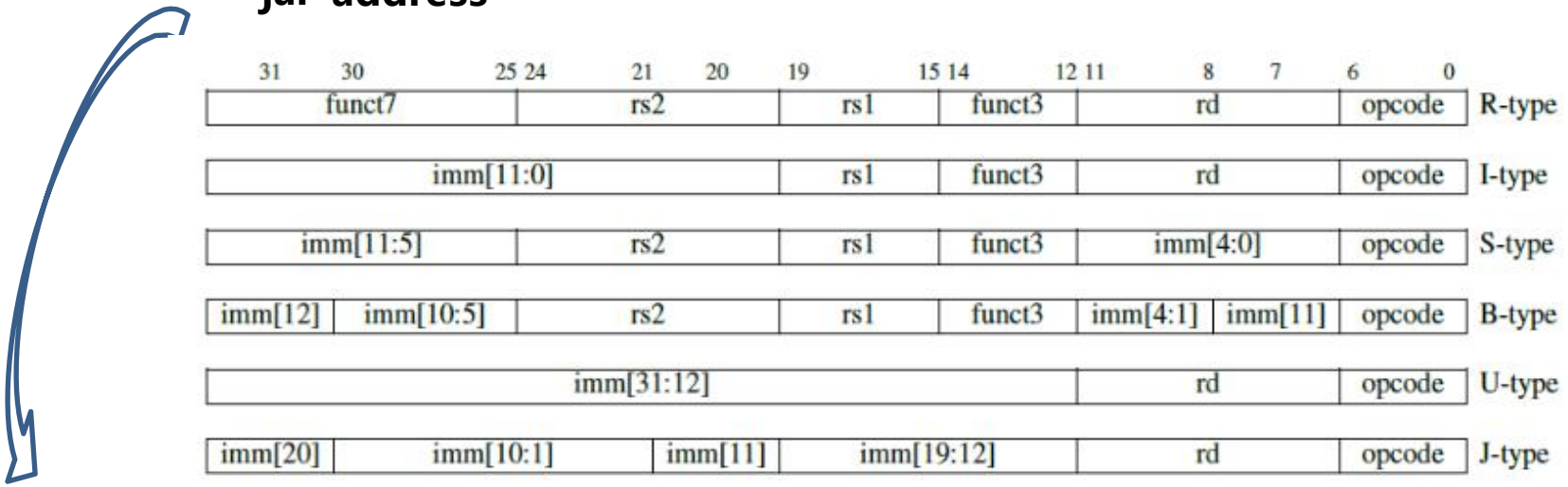
硬件对过程的支持--1. 指令和寄存器

jal：jump and link，将PC+4的值存入ra，并跳转到指定地址 (子程序入口) （J型指令）

jr：jump register，复制ra到PC，也可用于其它跳转 (比如：转移表)　　（R型指令）

指令格式：

jal address



->需要跳转到指定的32位立即数地址处怎么办？

硬件对过程的支持--2.叶过程(leaf procedures)

叶（子）过程：**leaf procedure**，不调用其他过程的过程

C code:

```
int leaf_example (int g, h, i, j)
{ int f;
  f = (g + h) - (i + j);
  return f;
}
```

f存于s0

参数g, …, j存于a0, …, a3

结果保存在v0

硬件对过程的支持--2.叶过程(leaf procedures)

叶（子）过程：**leaf procedure**，不调用其他过程的过程

C code:

```
int leaf_example (int g, h, i, j)
{ int f;
  f = (g + h) - (i + j);
  return f;
}
```

f存于s0

参数g, …, j存于a0, …, a3

结果保存在v0

子函数内如果使用s0~s7，则必须对其进行保存/恢复

RISC-V code:

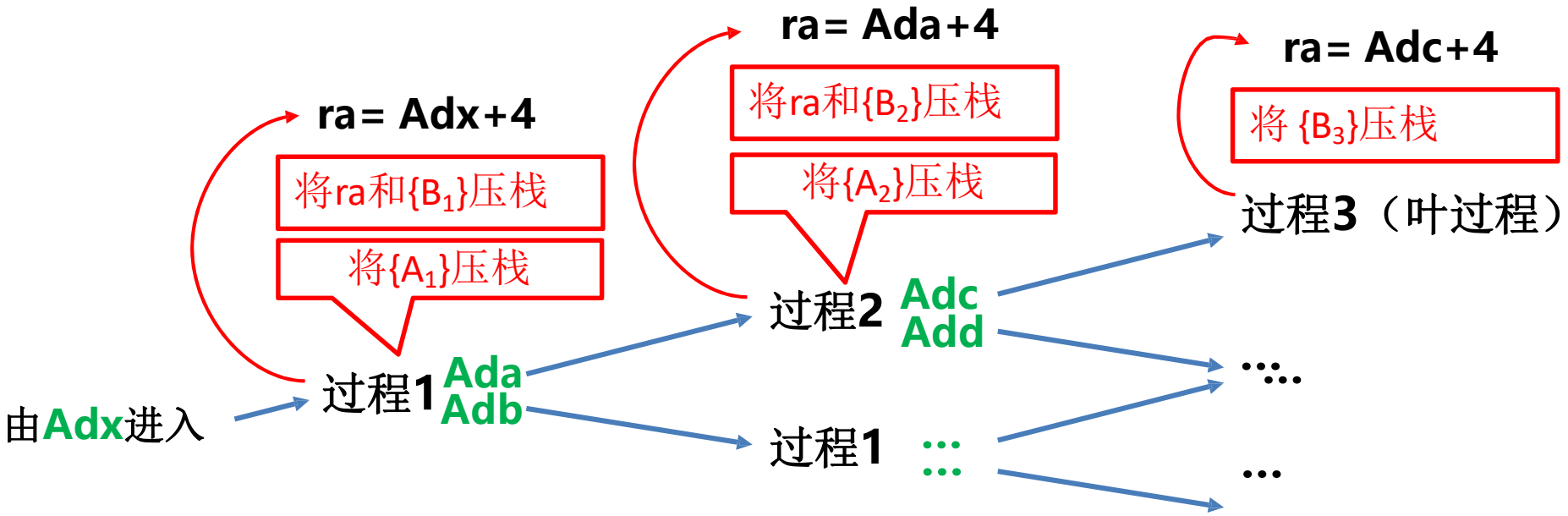
```
leaf_example:  addi sp, sp, -4      #s0入栈
               sw   s0, 0(sp)
               add  t0, a0, a1
               add  t1, a2, a3
               sub  s0, t0, t1
               add  v0, s0, zero    #保存结果  lw
                    s0, 0(sp)       #s0出栈
               addi sp, sp, 4
               jr   ra              #返回
```

硬件对过程的支持--3.嵌套过程(non-leaf procedures)

过程中调用新的过程

Caller需要保存调用后仍需使用(且callee需要写入)的a和t系列寄存器 $\{A_i\}$

Callee需要保存返回地址寄存器(叶过程除外)ra和需要使用的s系列寄存器$\{B_i\}$



别忘了出栈^-^：过程执行结束后，将先前压栈的数据恢复

小结

硬件对过程的支持

1. 指令和寄存器

2. 叶过程(leaf procedures)

3. 嵌套过程(non-leaf procedures)

4. 接口规范

小结

ISA简介

1. 处理器指令集体系结构(ISA)及其重要性
2. 从CISC到RISC

RISC-V 指令集体系结构与汇编语言入门

1. RISC-V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

硬件对过程的支持

1. 指令和寄存器
2. 叶过程(leaf procedures)
3. 嵌套过程(non-leaf procedures)