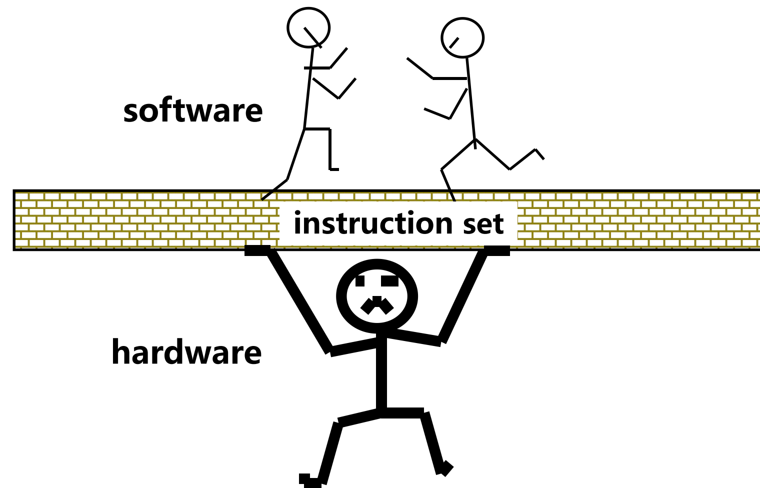
计算机体系结构

Topic II: ISA

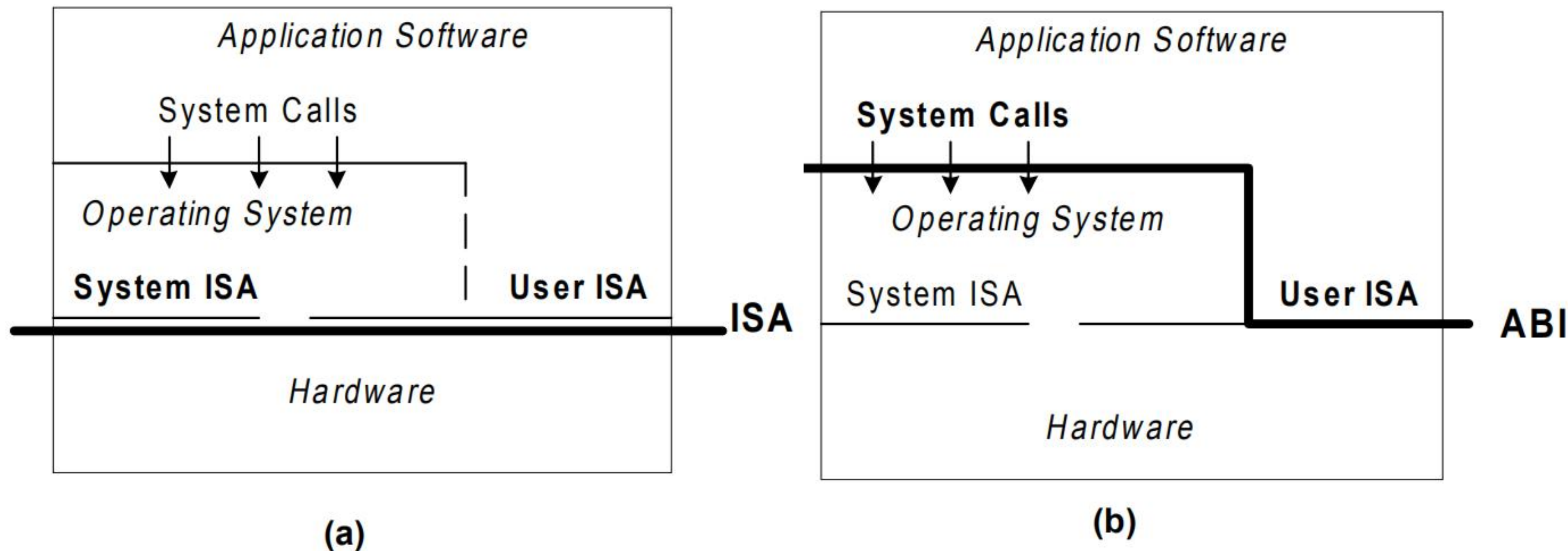
Fundamental Considerations in Designing an Instruction Set Architecture

指令集架构 (ISA)



- **软件子系统与硬件子系统的关键界面**
- **一组直接由硬件执行的指令，包括**
  - 程序员可见的机器状态
  - 程序员可见的指令集合(操作机器状态的指令)
- **应具备的特性**
  - 成本
  - 简洁性
  - 架构和具体实现分离：可持续多代，以保持向后（backward) 兼容
  - 可扩展空间：可用于不同应用领域（desktops, servers, embedded applications）
  - 易于编程/编译/链接：为高层软件的设计与开发提供方便的功能
  - 性能：方便低层硬件子系统高效实现
- **IBM 360 是第一个将ISA与其实现分离的系列机**
  - today you can buy AMD or Intel processors that run the x86-64 ISA
  - many cellphones use the ARM ISA with implementations from many different companies including TI, Qualcomm, Samsung, Marvell, etc

用户级ISA和特权级ISA



- **重要的系统界面（System Interface)**
  - ISA界面（Instruction Set Architecture)
  - ABI界面（Application Binary Interface)
- **ISA：用户级ISA+特权级ISA**
  - 用户级ISA 适用于操作系统和应用程序
  - 特权级ISA 适用于硬件资源的管理（操作系统）比如hardware thread. A key feature of RISC-V

ISA的实现

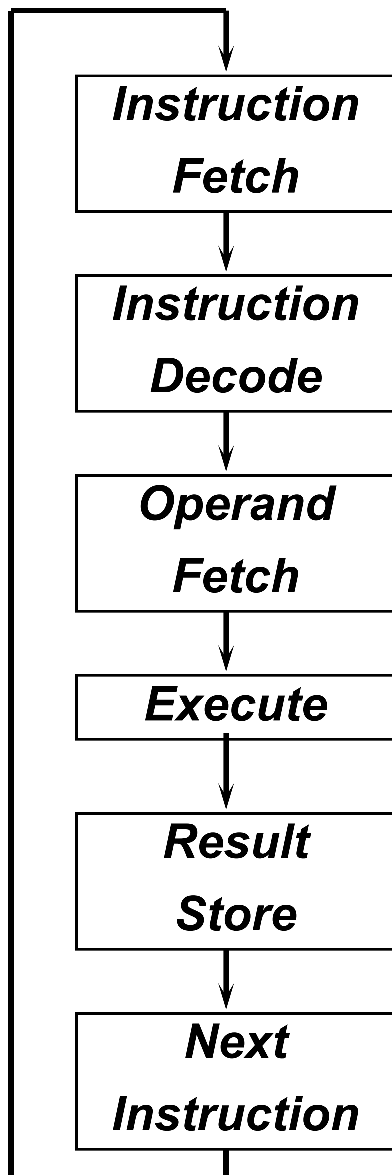
- **ISA 通常设计时会考虑特定的微体系结构（实现）方式**
  - Accumulator ⇒ hardwired, unpipelined
  - CISC ⇒ microcoded （微程序)
  - RISC ⇒ hardwired, pipelined（硬布线、流水线)
  - VLIW ⇒ fixed-latency in-order parallel pipelines (固定延时、顺序执行、多条流水线并行)
  - JVM ⇒ software interpretation（软件解释)
- **ISA 理论上可以用任何微体系结构（实现）方式**
  - Intel Ivy Bridge: hardwired pipelined CISC (x86) machine (with some microcode support)
  - Simics: Software-interpreted SPARC RISC machine
  - ARM Jazelle: A hardware JVM processor
  - RISC-V

ISA 的演进

	Accumulator	Stack	Register-Memory	Register-Register
C=A+B	Load A Add B Store C	Push A Push B Add Pop C	Load R1, A Add R1, B Store R1, C	Load R1, A Load R2, B Add R3, R1, R2 Store R3, C

几个操作数？其存储位置

ISA必须specify哪些东西?



- **指令格式或编码方式。即如何编码?**
- **操作数和操作结果的存放位置**
  - 存放位置?
  - 多少个显式操作数?
  - 存储器操作数如何定位?
  - 哪些操作数可以或不可以放到存储器中?
  - 寻址方式
- **数据类型和大小**
- **支持哪些操作**
- **下一条指令地址**
  - jumps, conditions, branches
  - fetch-decode-execute is implicit!

有关ISA的若干问题

- **存储器寻址**
- 操作数的类型与大小
- 所支持的操作
- 控制转移类指令
- 指令格式

Processor – Memory Interconnections

- Memory is viewed as a large, single-dimension array, with an address
- A memory address is an index into the array

存储器寻址

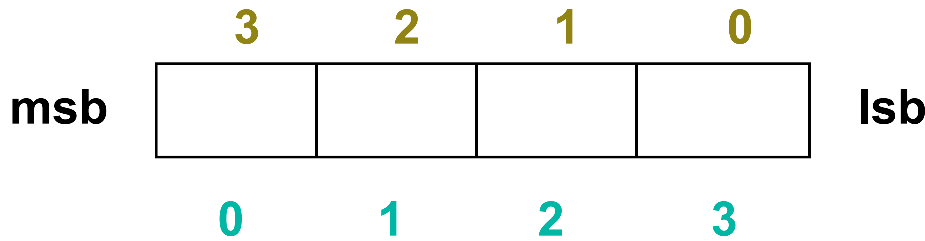
- **80年以来几乎所有机器的存储器都是按字节编址**
- **一个存储器地址可以访问：**
  - 一个字节、2个字节、4个字节、更多字节…..
- **不同体系结构对字word的定义是不同的**
  - 16位字（Intel X86）32位字（MIPS）
- **如何读32位字，两种方案**
  - 每次一个字节，四次完成；每次一个字，一次完成
- **问题：**

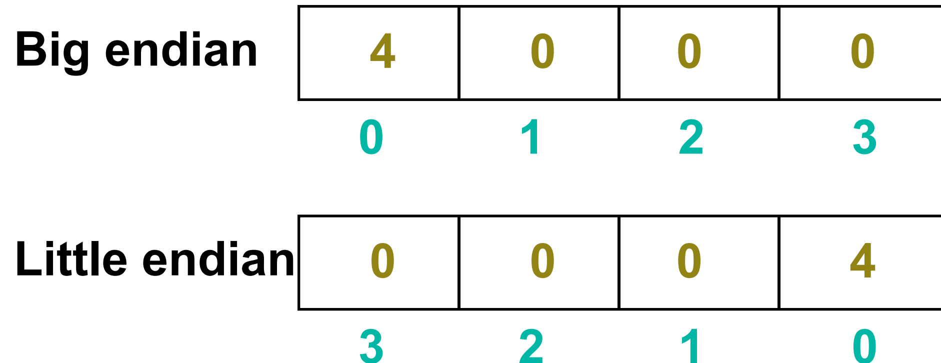
（1）如何将字节地址映射到字地址 (尾端问题 endian)

（2）一个字是否可以存放在任何字节边界上(对齐问题 alignment)

Addressing Objects: Endianess and Alignment

- Big Endian: leftmost byte is word address
  IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA
- Little Endian: rightmost byte is word address
  Intel 80x86, DEC Vax, DEC Alpha (Windows NT)





尾端问题 Endian

- **little endian, big endian, 在一个字内部的字节顺序问题**

1. Little Endian byte ordering

x+3	x+2	x+1	x	
Byte 3	Byte 2	Byte 1	Byte 0	32-bit Register

✧ Memory address X = address of least-significant byte (Intel x86)

2. Big Endian byte ordering

x	x+1	x+2	x+3	
Byte 0	Byte 1	Byte 2	Byte 3	32-bit Register

✧ Memory address X = address of most-significant byte (SPARC)

- **如地址xxx00指定了一个字（int），存储器中从xxx00处连续存放ffff0000, 则有两种方式：**
  - Little endian 方式下xxx00位置是字的最低字节, 四个字节的内容分别为00 00 ff ff, Intel 80x86, DEC Vax, DEC Alpha (Windows NT)
  - Big endian 方式下xxx00位置是字的最高字节，四个字节的内容分别为ff ff 00 00, IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA

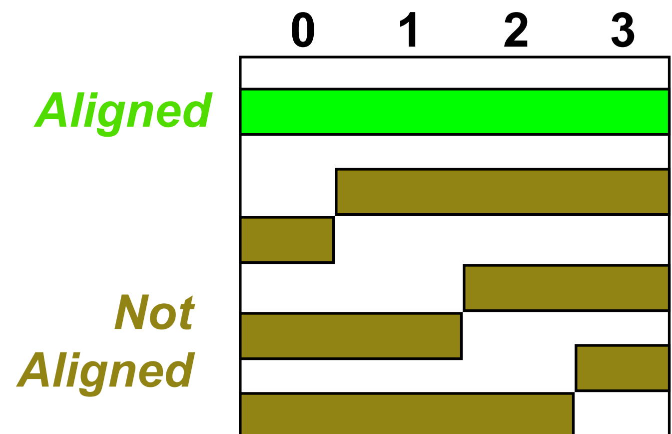
Byte Addresses

- Since 8-bit bytes are so useful, most architectures address individual bytes in memory

  memory: 2^{32} bytes = 2^{30} words

- Therefore, the memory address of a word must be a multiple of 4 (alignment restriction)

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



Recap: 对齐问题

- 对s字节的对象访问地址为A，如果A mod s =0 称为边界对齐。
- 边界对齐的原因是存储器本身读写的要求，存储器本身读写通常就是边界对齐的，对于不是边界对齐的对象的访问可能要导致存储器的两次访问，然后再拼接出所需要的数。（或发生异常）

Address mod 8	0	1	2	3	4	5	6	7
Byte	Aligned	Aligned	Aligned	Aligned	Aligned	Aligned	Aligned	Aligned
2 Bytes	Aligned		Aligned		Aligned		Aligned	
2 Bytes		Misaligned		Misaligned		Misaligned		Misalign
4 Bytes	Aligned				Aligned			
4 Bytes		Misaligned				Misaligned		
4 Bytes			Misaligned				Misaligned	
4 Bytes				Misaligned				Misalign
8 Bytes	Aligned							
8 Bytes		Misaligned						

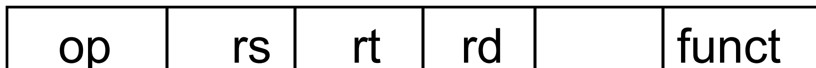
Recap: 寻址方式

- **寻址方式：如何说明要访问的对象地址**
- **有效地址：由寻址方式说明的某一存储单元的实际存储器地址。有效地址 vs. 物理地址**

Mode	Example	Meaning	When used
Register	Add R1, R2	R1 ← R1 + R2	Values in registers
Immediate	Add R1, 100	R1 ← R1 + 100	For constants
Register Indirect	Add R1, (R2)	R1 ← R1 + Mem(R2)	R2 contains address
Displacement	Add R1, (R2+16)	R1 ← R1 + Mem(R2+16)	Address local variables
Absolute	Add R1, (1000)	R1 ← R1 + Mem(1000)	Address static data
Indexed	Add R1, (R2+R3)	R1 ← R1 + Mem(R2+R3)	R2=base, R3=index
Scaled Index	Add R1, (R2+s*R3)	R1 ← R1 + Mem(R2 + s*R3)	s = scale factor = 2, 4, or 8
Post-increment	Add R1, (R2)+	R1 ← R1 + Mem(R2) R2 ← R2 + s	Stepping through array s = element size
Pre-decrement	Add R1, -(R2)	R2 ← R2 – s R1 ← R1 + Mem(R2)	Stepping through array s = element size

MIPS寻址方式图示

1. Register addressing



Register

word operand

2. Base addressing

Memory

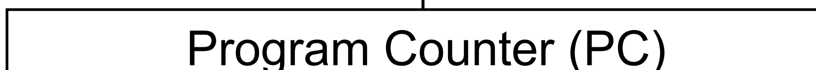
word or byte operand

3. Immediate addressing

4. PC-relative addressing

Memory

branch destination instruction



5. Pseudo-direct addressing

Memory

jump destination instruction

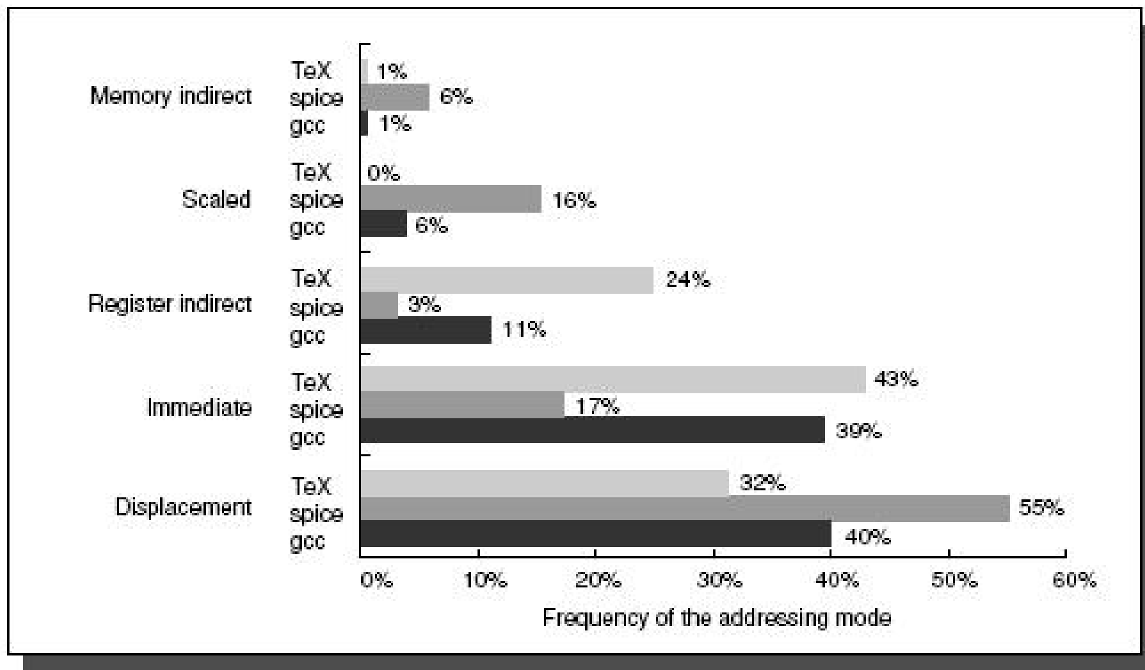
Naming Conventions for Registers

No.	Register	Description
0	$zero	constant 0
1	$at	reserved for assembler
2	$v0	expression evaluation &
3	$v1	function results
4	$a0	arguments
5	$a1	
6	$a2	
7	$a3	
8	$t0	temporary: caller saves
. . .		(callee can clobber)
15	$t7	
16	$s0	callee saves
. . .		(caller can clobber)
23	$s7	
24	$t8	temporary (cont'd)
25	$t9	
26	$k0	reserved for OS kernel
27	$k1	
28	$gp	pointer to global area
29	$sp	stack pointer
30	$fp	frame pointer
31	$ra	return address

MIPS寄存器

- **MIPS的寄存器**
  - 32个64位通用寄存器（GPRs）
    - R0，R1，…，R31
    - 也被称为整数寄存器
    - R0的值永远是0
  - 32个64位浮点数寄存器（FPRs）
    - F0，F1，…，F31
    - 用来存放32个单精度浮点数（32位），也可以用来存放32个双精度浮点数（64位）。
    - 存储单精度浮点数（32位）时，只用到FPR的一半，其另一半没用
  - 一些特殊寄存器
    - 它们可以与通用寄存器交换数据。
    - 例如，浮点状态寄存器用来保存有关浮点操作结果的信息。

各种寻址方式的使用情况? (忽略寄存器直接寻址)



三个SPEC89程序在VAX结构上的测试结果:

立即寻址，偏移寻址使用较多

偏移寻址

- **主要问题：偏移的范围（偏移量的大小）**

lw $1, ? ($2)

64bit

Alpha Architecture with full optimization for Spec CPU2000, showing the average of integer programs(CINT2000) and the average of floating-point programs (CFP2000)

立即数寻址

Alpha Architecture with full optimization for Spec CPU2000, showing the average of integer programs(CINT2000) and the average of floating-point programs (CFP2000)

立即数的大小

The distribution of immediate values. About 20% were negative for CINT2000 and about 30% were negative for CFP2000. These measurements were taken on a Alpha, where the maximum immediate is 16 bits, for the spec cpu2000 programs. A similar measurement on the VAX, which supported 32-bit immediates, showed that about 20% to 25% of immediates were longer than 16 bits.

Small constants are the most common

寻址方式小结

- **重要的寻址方式**
  - 偏移寻址方式, 立即数寻址方式, 寄存器间址方式
  - SPEC测试表明，使用频度达到 75%--99%
- **偏移字段的大小应该在 12 - 16 bits**
  - 可满足75%-99%的需求
- **立即数字段的大小应该在 8 -16 bits**
  - 可满足50%-80%的需求

Recap: 有关ISA的若干问题

- 存储器寻址
- **操作数的类型与大小**
- 所支持的操作
- 控制转移类指令
- 指令格式

操作数的类型、表示和大小

- **操作数类型和操作数表示是软硬件的主要界面之一。**
- **操作数类型：是面向应用、面向软件系统所处理的各种数据类型。**
  - 整型、浮点型、字符、字符串、向量类型等
  - 操作数类型由操作码确定
  - 操作数的表示：操作数在机器中的表示，硬件结构能够识别，指令系统可以直接使用的表示格式
  - 整型：原码、反码、补码 (2’s complement)
  - 浮点：IEEE 754标准
  - 十进制：BCD码/二进制十进制表示

常用操作数类型

- ASCII character = 1 byte (64-bit register can store 8 characters
- Unicode character or Short integer = 2 bytes = 16 bits   (half word)
- Integer = 4 bytes = 32 bits (word size on many RISC Processors)
- Single-precision float = 4 bytes = 32 bits (word size)
- Long integer = 8 bytes = 64 bits (double word)
- Double-precision float = 8 bytes = 64 bits (double word)
- Extended-precision float = 10 bytes = 80 bits (Intel architecture)
- Quad-precision float = 16 bytes = 128 bits

操作数的大小

	applu	equake	gzip	perl
Double word (64 bits)	60%	94%	31%	62%
Word (32 bits)	40%	6%	28%	18%
Half word (16 bits)	0%	0%	19%	3%
Byte (8 bits)	0%	0%	22%	18%

0% 20% 40% 60% 80% 100%

基准测试的结论：（1）对单字、双字的数据访问具有较高的频率（2）支持64位双字操作，更具有一般性

MIPS的数据表示

- **整数**
  - 字节（8位）　半字（16位）字（32位）　双字（64位）
- **浮点数**
  - 单精度浮点数（32位）　双精度浮点数（64位）

- **字节、半字或者字在装入32/64位寄存器时，用零扩展或者用符号位扩展来填充该寄存器的剩余部分。装入以后，对它们将按照32/64位整数的方式进行运算。**

Recap: 有关ISA的若干问题

- 存储器寻址
- 操作数的类型与大小
- **所支持的操作**
- 控制转移类指令
- 指令格式

典型操作类型

Operator type	Examples
Arithmetic and logical	Integer arithmetic and logical operations: add, subtract, and, or, multiple, divide
Data transfer	Loads-stores (move instructions on computers with memory addressing)
Control	Branch, jump, procedure call and return, traps
System	Operating system call, virtual memory management instructions
Floating point	Floating-point operations: add, multiply, divide, compare
Decimal	Decimal add, decimal multiply, decimal-to-character conversions
String	String move, string compare, string search
Graphics	Pixel and vertex operations, compression/decompression operations

- **一般计算机都支持前三类所有的操作**；
- **不同计算机系统 对系统支持程度不同，但都支持基本的系统功能。**
- **对最后四类操作的支持程度差别也很大，有些机器不支持，有些机器还在此基础上做一些扩展，这些指令有时作为可选的指令。**

Top 10 80x86 Instructions

Rank	instruction	Integer Average Percent total executed
1	load	22%
2	conditional branch	20%
3	compare	16%
4	store	12%
5	add	8%
6	and	6%
7	sub	5%
8	move register-register	4%
9	call	1%
10	return	1%
	Total	96%

- Simple instructions dominate instruction frequency

ISA对操作类型的选择

- **需考虑的因素：速度、价格和灵活性**
- **基本要求：指令系统的完整性、规整性、高效率和兼容性**
  - 完整性设计：具备基本指令种类
  - 兼容性：系列机
  - 高效率：指令执行速度快、使用频度高
  - 规整性
    - 让所有运算部件都能对称、均匀的在所有数据存储单元之间进行操作。
    - 对所有数据存储单元都能同等对待，无论是操作数或运算结果都可以无约束地存放到任意数据存储单元中
  - 正交性
    - 数据类型独立于寻址方式
    - 寻址方式独立于所要完成的操作
- **当前对这一问题的处理有两种截然不同的方向**
  - CISC和RISC

CISC vs RISC

- **ISA的功能设计**
  - 任务：确定硬件支持哪些操作
  - 方法：统计的方法
  - 两种类型：CISC和RISC
- **CISC（Complex Instruction Set Computer)**
  - 目标：强化指令功能，减少运行的指令条数，提高系统性能
  - 方法：面向目标程序的优化，面向高级语言和编译器的优化
- **RISC（Reduced Instruction Set Computer)**
  - 目标：通过简化指令系统，用高效的方法实现最常用的指令
  - 方法：充分发挥流水线的效率，降低（优化）CPI

Recap: 有关ISA的若干问题

- 存储器寻址
- 操作数的类型与大小
- 所支持的操作
- **控制转移类指令**
- 指令格式

控制流类指令中的寻址方式

- **PC-relative 方式 （相对寻址）**
  - 例如：条件转移
- **说明动态的转移地址方式(通过寄存器间接跳转）：**
  - 编译时不知道目标地址，程序执行时动态确定
    - Case or switch statements
    - Virtual function or methods
    - High-order functions or function pointers
    - Dynamically shared libraries
  - 转移地址放到某一寄存器中，通过寄存器间接跳转

转移目标地址与当前指令的距离

Integer average

Floating-point average

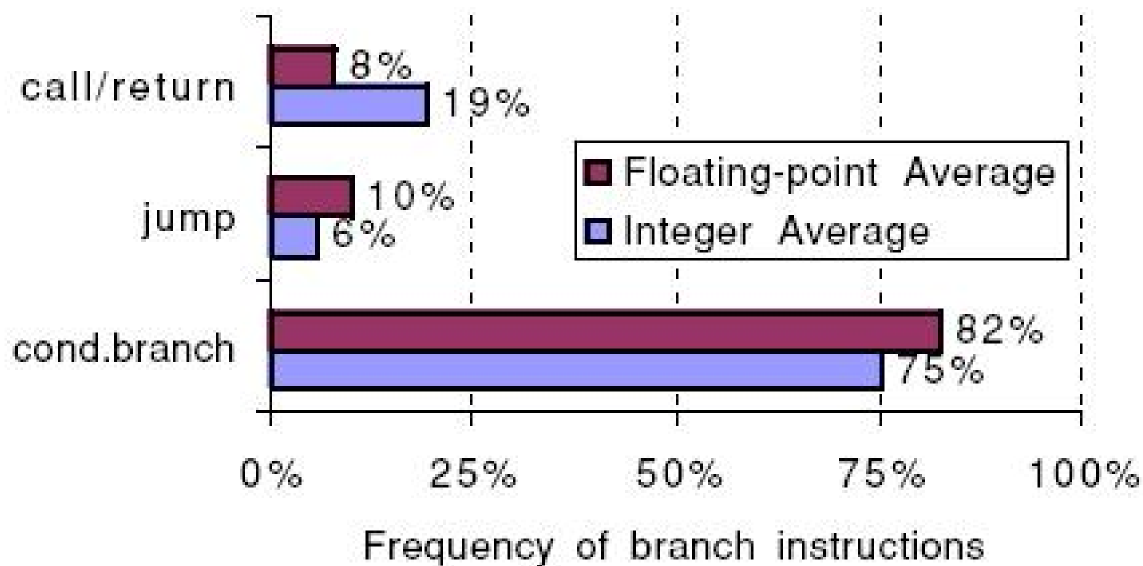
Bits of branch displacement

Alpha Architecture with full optimization for Spec CPU2000, showing the average of integer programs(CINT2000) and the average of floating-point programs (CFP2000)

建议：shorter offsets are the most common

控制类指令

- 四种类型的控制流改变：
  - 条件分支(Conditional branch) 、跳转(Jump)、过程调用(Procedure calls)、过程返回(Procedure returns)



Alpha Architecture with full optimization for Spec CPU2000, showing the average of integer programs(CINT2000) and the average of floating-point programs (CFP2000)

Conditional branches dominate

Recap：跳转指令

- **根据跳转(jump)指令确定目标地址的方式不同以及跳转时是否链接，可以把跳转指令分成4种。**
- **确定目标地址的方式**
  - 把指令中的26位偏移量左移2位后，替换程序计数器的低28位。
  - 间接跳转：由指令中指定的一个寄存器来给出转移目标地址。
- **跳转的两种类型**
  - 简单跳转：把目标地址送入程序计数器。
  - 跳转并链接：把目标地址送入程序计数器，把返回地址（即顺序下一条指令的地址）放入寄存器R31。

分支比较类型比较

Comparison type	Floating-point Average	Integer Average
Not equal	5%	2%
Equal	16%	18%
Greater than or Equal	0%	11%
Greater than	0%	0%
Less than or equal	44%	33%
Less than	34%	35%

Frequency of comparison types in branches (0%–50%)

Alpha Architecture with full optimization for Spec CPU2000, showing the average of integer programs(CINT2000) and the average of floating-point programs (CFP2000)

Less than (or equal) conditions dominate

Recap：分支指令（条件转移）

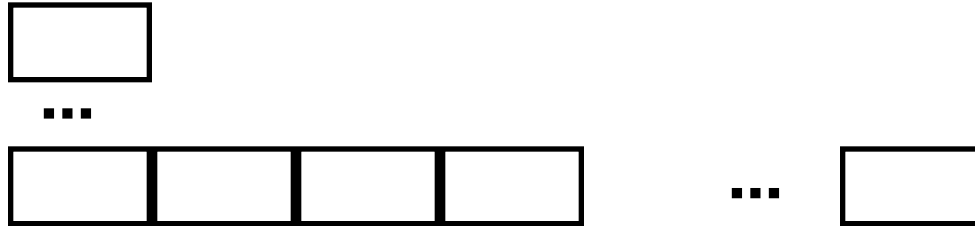
- **分支条件由指令确定。**
  - 例如：测试某个寄存器的值是否为零
- **提供一组比较指令，用于比较两个寄存器的值。**
  - 例如：“置小于”指令
- **有的分支指令可以直接判断寄存器内容是否为负，或者比较两个寄存器是否相等。**
- **分支的目标地址。**
  - 由16位带符号偏移量左移两位后和PC相加的结果来决定
- **一条浮点条件分支指令：通过测试浮点状态寄存器来决定是否进行分支。**

Recap: 有关ISA的若干问题

- 存储器寻址
- 操作数的类型与大小
- 所支持的操作
- 控制转移类指令
- **指令格式**

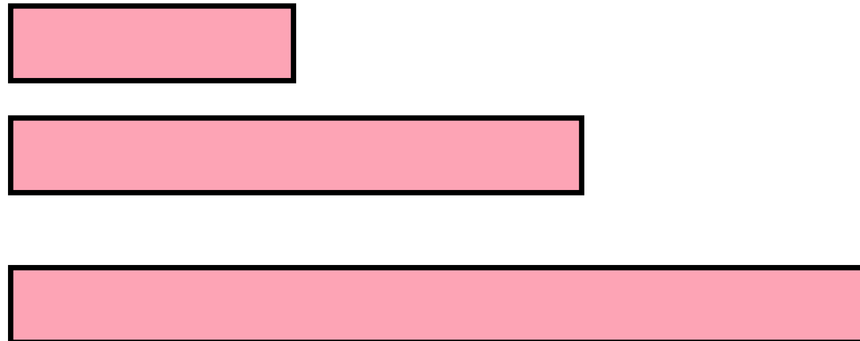
指令编码

Variable:



Fixed:

Hybrid:



指令格式选择策略

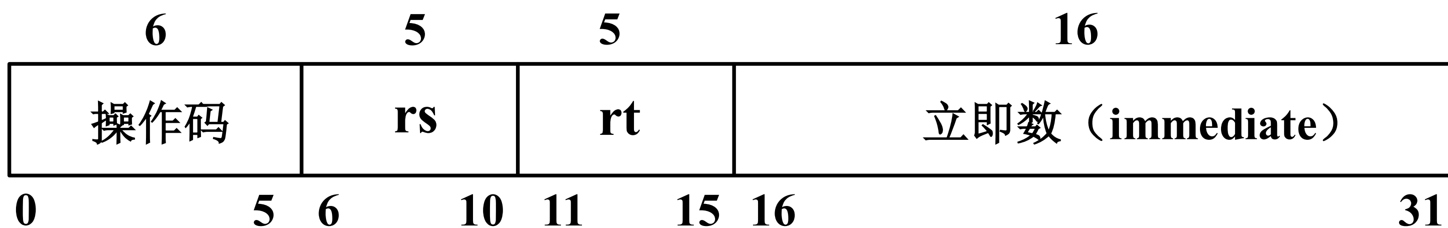
- **如果代码规模最重要，那么使用变长指令格式**
- **如果性能至关重要，使用固定长度指令**
- **有些嵌入式CPU附加可选模式，由每一应用自主选择性能还是代码量**
  - 窄指令支持更少的操作、更短的地址和立即数字段、更少的寄存器以及双地址格式，而不是传统的RISC计算机的三地址格式
  - 例如：RISC-V 的 RV32IC， C表示压缩表示
  - ARM Thumb ， microMIPS
- **有些机器使用边执行边解压的方式**
  - 例如 IBM 的CodePack PowerPC

MIPS的指令格式

- **寻址方式编码到操作码中**
- **所有指令都是32位的**
- **操作码占6位**
- **3种指令格式**
  - I类
  - R类
  - J类

I类指令

- 包括所有的load和store指令、立即数指令，分支指令。
- 立即数字段为16位，用于提供立即数或偏移量。

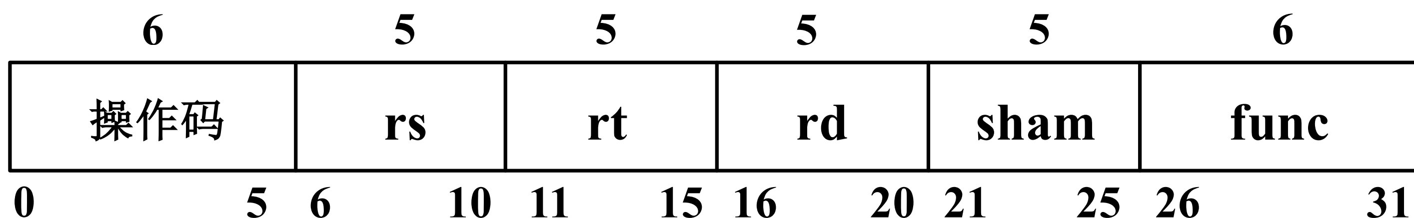


具体的I类指令

- **load指令**
  - 访存有效地址：Regs[rs] + immediate
  - 从存储器取来的数据放入寄存器rt
- **store指令**
  - 访存有效地址：Regs[rs] + immediate
  - 要存入存储器的数据放在寄存器rt中
- **立即数指令**
  - Regs[rt] ← Regs[rs] op immediate
- **分支指令**
  - 转移目标地址：PC ← PC+4 + 4*immediate

R类指令

- **包括ALU指令、专用寄存器读/写指令、寄存器跳转指令、move指令等**
- **ALU指令**
  - Regs[rd]← Regs[rs] funct Regs[rt]
  - func为具体的运算操作编码



J类指令

- **包括跳转指令、跳转并链接指令、自陷指令、异常返回指令**
- **在这类指令中，指令字的低26位是偏移量，它与PC值相加形成跳转的地址**

6	26
操作码	与 PC 相加的偏移量
0　　5	6　　31

Recap：MIPS的操作

- MIPS指令可以分为四大类
  - load和store（内存访问）、ALU操作、分支与跳转、浮点操作

Recap：MIPS（64位）控制类指令

指令举例	指令名称	含义
J name	跳转	$PC_{36..63}$ ← name<<2
JAL name	跳转并链接	Regs[R31]←PC+4；$PC_{36..63}$←name<<2； ((PC+4)—2^{27})≤name<((PC+4)+2^{27})
JALR R3	寄存器跳转并链接	Regs[R31]←PC+4；PC← Regs[R3]
JR R5	寄存器跳转	PC← Regs[R5]
BEQZ R4，name	等于零时分支	if(Regs[R4]== 0) PC←name ； ((PC+4)—2^{17})≤name<((PC+4)+2^{17})
BNE R3，R4，name	不相等时分支	if(Regs[R3]!= Regs[R4]) PC←name ((PC+4)—2^{17})≤name<((PC+4)+2^{17})
MOVZ R1，R2，R3	等于零时移动	if(Regs[R3]==0) Regs[R1]← Regs[R2]

小结：指令集架构

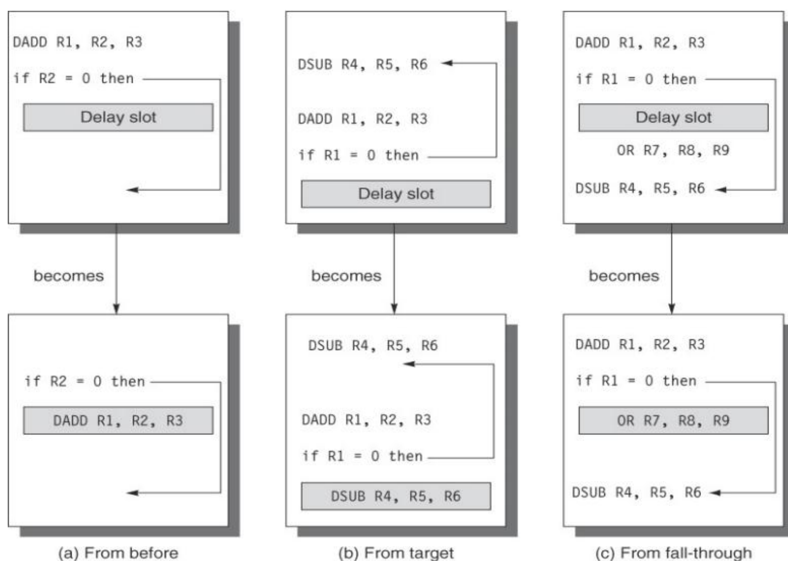
- **ISA需考虑的问题**
  - Class of ISA
  - Memory addressing
  - Types and sizes of operands
  - Operations
  - Control flow instructions
  - Encoding an ISA
  - ……
- **ISA的类型**
  - 通用寄存器型占主导地位
- **寻址方式**
  - 重要的寻址方式: 偏移寻址方式, 立即数寻址方式, 寄存器间址方式
    - SPEC测试表明，使用频度达到 75%--99%
  - 偏移字段的大小应该在 12 - 16 bits, 可满足75%-99%的需求
  - 立即数字段的大小应该在 8 -16 bits, 可满足50%-80%的需求
- **操作数的类型和大小**
  - 对单字、双字的数据访问具有较高的频率
  - 支持64位双字操作，更具有一般性

MIPS

- **MIPS是最典型的RISC 指令集架构**
  - Stanford， 1980年提出，主要受到IBM801 小型机的影响
  - 第一个商业实现是R2000（1986）
  - 最初的设计中，其整数指令集仅有**58条指令**，直接实现单发射(single-issued)、顺序流水线
  - 30年来，逐步增加到约**400条指令**。
- **主要特征：**
  - Load/Store型结构，专门的指令完成存储器与寄存器之间的传送
  - ALU类指令的操作数来源于寄存器或立即数（指令中的特定区域）
  - 降低了指令集和硬件的复杂性，依赖于优化编译技术，方便了简单流水线的实现

MIPS Shortcomings

- **主要缺陷：**
  - 针对特定的微体系架构的实现方式（5级流水、单发射、顺序流水线）进行过度的优化设计
    - 延迟转移 (delayed branch)问题导致超标量等复杂流水线的实现难度，当无法有效填充延迟槽时会导致代码尺寸变大
    - MIPS-I中暴露出其他流水线冲突（load、乘除引起的冲突）采用简单的Interlocking 简单又高效，但为了保持兼容性，仍然保留了延迟转移



MIPS Shortcomings

- **主要缺陷：**
  - ISA对位置无关的代码（position-independent code, PIC)支持不足。
    - 直接跳转没有提供PC相对寻址，需要通过间接跳转方式实现PIC，增加了代码尺寸，降低了性能
    - PIC代码支持动态链接、安全性和加载的灵活性
    - 2014年MIPS的修订，改进了PC-相对寻址(针对数据)，但仍然要多条指令才能完成

Please give an example instruction that is position dependent

MIPS Shortcomings

- **主要缺陷：**
  - 16位位宽立即数消耗了大量编码空间，只有少量的编码空间可供扩展指令
    - 2014修订版，保存有1/64的编码空间供扩展
    - 架构师如果想采用压缩指令编码来降低代码空间，就不得不采用新的指令编码
  - 乘除指令使用了特殊的寄存器（HI,LO)，导致上下文切换内容、指令条数、代码尺寸增加，微架构实现复杂

MIPS Shortcomings

- ISA假设浮点操作部件是一个独立的协处理器，使得单芯片实现无法最优
  - 例如，整型数与浮点数的转换结果写到浮点数寄存器，使用结果时，需要额外的mov指令，更糟糕的是浮点数寄存器文件与整型数寄存器文件之间的传输，需要有显式的延迟槽
- 在标准的ABI中，保留两个整型寄存器用于内核程序，减少了用户程序可用的寄存器数

Load word left (LWL), load word right (LWR)

- 使用特殊指令处理未对齐的load和store会消耗大量的操作码空间，并使除了最简单的实现之外的其他实现复杂化。
- 时钟速率/CPI 的权衡使得架构师省略了整数大小比较和分支指令。随着分支预测和静态CMOS逻辑的出现，这种权衡在今天已经不太合适了。

- 除了技术方面，MIPS是非开放的专属指令集，不能自由使用

SPARC (Scalable Processor Architecture)

- **Sun Microsystems的专属指令集**
  - 可追溯到Berkeley RISC-I和RISC-II项目 (early 1980)
  - One of the most successful early commercial RISC
- **SPARC V8 （1990） 主要特征**
  - 用户级 整型ISA **90条指令**； 硬件支持IEEE 754-1985标准的浮点数(50条)； 特权级指令**20条**
- **主要问题**
  - SPARC使用了寄存器窗口(register windows)来加速函数调用
    - The number of registers is very limited. At function calls, the registers often need to be saved, thus very slow.
    - Registers windows enable a set of registers to be swamped during function calls.
    - 对于所有的实现来说，寄存器窗口都消耗很大的面积和功耗
    - 更多细节：http://icps.u-strasbg.fr/people/loechner/public_html/enseignement/SPARC/sparcstack.html

SPARC (Scalable Processor Architecture)

- **主要问题**
  - 分支使用条件码
  - For signed numbers, SPARC uses one of the three condition codes- the Z, N, and V bits - to regulate conditional branching
  - 这些条件码由于在一些指令之间创建了额外的依赖关系，增加了体系结构状态并使实现复杂化

Abbreviations Used in Instruction Descriptions	
r[X]	The contents of register X. The instruction descriptions view r as an array of ints.
mem[X]	The contents of memory at location X. The instruction descriptions view mem as an array of chars.
constX	Constant that fits into X bits.
Z	Zero condition code. The instruction descriptions view Z as an int whose value is either 0 (FALSE) or 1 (TRUE).
N	Negative condition code. The instruction descriptions view N as an int whose value is either 0 (FALSE) or 1 (TRUE).
V	oVerflow condition code. The instruction descriptions view V as an int whose value is either 0 (FALSE) or 1 (TRUE).
C	Carry condition code. The instruction descriptions view C as an int whose value is either 0 (FALSE) or 1 (TRUE).

The condition code register on the SPARC has four bits: **Z (Zero), N (Negative), C (Carry), and V (oVerflow)**. The standard arithmetic operations (e.g., addition and subtraction) do no update the bits in the condition code register. Instead, there are special operations that update the condition code register.

Arithmetic Mnemonics (Format 3)	

add rs1,rs2,rd
add rs1,const13,rd
	Add
	r[rd] = r[rs1] + r[rs2];
	r[rd] = r[rs1] + const13;

addcc rs1,rs2,rd
 	
addcc rs1,const13,rd
	Add, and set condition codes
	r[rd] = r[rs1] + r[rs2];
	N = r[rd]<0; Z = r[rd]==0;
	V = (r[rs1]<0 & r[rs2]<0 & r[rd]>0)
	| (r[rs1]>0 & r[rs2]>0 & r[rd]<0);
	C = (r[rs1]<0 & r[rs2]<0) | (r[rd]>0 & (r[rs1]<0 | r[rs2]<0));
	r[rd] = r[rs1] + const13;
	N = r[rd]<0; Z = r[rd]==0;
	V = (r[rs1]<0 & const13<0 & r[rd]>0)
	| (r[rs1]>0 & const13>0 & r[rd]<0);
	C = (r[rs1]<0 & const13<0) | (r[rd]>0 & (r[rs1]<0 | const13<0));

SPARC (Scalable Processor Architecture)

- **主要问题**
  - load和store**相邻寄存器对**的指令
    - Ldd （需要一对相邻寄存器存放双字，并且必须是偶数寄存器；高字移入偶数寄存器，低字移入奇数寄存器）
    - 对于简单的微体系结构很有吸引力，可以在很少增加硬件复杂性的情况下提高吞吐量。
    - 遗憾的是当使用寄存器重命名使实现复杂化，因为在寄存器文件中数据在物理上可能不再相邻
  - 浮点寄存器文件和整数寄存器文件之间的移动必须使用内存系统作为中介，限制了系统性能

SPARC

- ISA通过体系结构公开的延迟陷阱队列支持非精确浮点异常，该队列向系统监控程序提供信息，以恢复此类异常上的处理器状态
- 唯一的原子内存操作是fetch-and-store，这对于实现许多无等待的数据结构是不够的

- **SPARC与其他80年代RISC结构的许多有缺陷的特性**
  - ISA设计面向单发射、顺序、五级流水线的微体系架构；
  - SPARC具有分支延迟插槽和许多显式的数据和控制冲突，这些冲突使代码生成复杂化，无助于更积极的实现；
  - 缺乏位置无关的寻址方式（相对寻址）
  - 由于SPARC缺乏足够的自由编码空间，因此不能方便地对其进行改进以支持压缩ISA扩展

Alpha （DEC)

- **DEC公司的架构师在20世纪90年代初定义了他们的RISC ISA, Alpha**
  - 摒弃了当时非常吸引人的特性，如分支延迟、条件码、寄存器窗口等
  - 创建了64位寻址空间、设计简洁、实现简单、高性能的ISA
  - Alpha架构师仔细地将特权体系结构和硬件平台的大部分细节隔离在抽象接口(特权体系结构库)后面(PALcode)
- **主要问题：尽管如此，DEC对顺序微架构的Alpha进行了过度优化，并添加了一些不太适合现代实现的特性**
  - 为了追求高时钟频率，ISA的原始版本避免了8位和16位的加载和存储，实际上创建了一个字寻址的内存系统。为了在广泛使用这些操作的应用程序上性能，添加了特殊的未对齐的加载和存储指令以及一些整数指令，以加速重新组合过程。
  - 为了方便长延迟浮点指令的乱序完成，Alpha 有一个非精确的浮点陷阱模型。这个决定可能是可以单独接受的，但是ISA还定义了异常标记和默认值(如果需要的话)必须由软件例程提供。
  - Alpha缺少整数除法指令，建议使用软件牛顿迭代法实现，导致浮点除法速度高于整数除法

Alpha （DEC）

- 与它的前辈RISC一样，没有预先考虑可能的压缩指令集扩展，因此没有足够的操作码空间来进行更新
- ISA包含有条件的移动，这使得微架构与寄存器重命名复杂化
  - cmoveq R1, R2, R3 (if R1==0 R2 = R3)

  a = max (a, b) ➔ if (a>= b) {c = a}; else c=b;

- 使用商业Alpha ISAs的一个重要风险:它们可能会被摒弃。康柏在上世纪90年代末收购了摇摇欲坠的DEC后不久，他们选择逐步淘汰Alpha，转而采用英特尔的安腾架构。康柏将Alpha的知识产权出售给了英特尔，此后不久，惠普收购了康柏，并在2004年完成了Alpha的最终实现

ARMv7 (Advanced RISC Machine)

- **32位 RISC ISA**
  - 目前世界上使用最广的体系结构。当我们权衡是否要设计自己的指令集时，ARMv7是一个自然的选择，大量的软件已经被移植到该ISA上，而且它在嵌入式和移动设备中无处不在。
  - 是一个封闭的标准，剪裁或扩充是不允许的，即使是微架构的创新也仅限于那些能够获得ARM所称的架构许可的人
  - ARMv7十分庞大复杂。整型类指令**600+条**
- **即使知识产权不是问题，它仍然存在一些技术缺陷**
  - 不支持64位地址， ISA缺乏硬件支持IEEE754-2008标准（ ARMv8 纠正了这些缺陷）
  - 特权体系结构的细节渗透到用户级体系结构的定义中

ARMv7 (Advanced RISC Machine)

- ARMv7附带一个压缩ISA，具有固定宽度的16位指令，称为Thumb。
  - Thumb虽然提供了有竞争力的代码尺寸，但性能较差
  - Thumb-2 虽然提供了较高的性能，但32位的Thumb-2编码方式与基本的ISA编码方式不同,16位的Thumb-2的编码方式与基本的16位编码方式也不同。导致译码器需要理解三种编码格式，使得能耗、延迟以及设计成本增加
- ISA中包含了许多实现复杂的特性。
  - 程序计数器(PC)是可寻址寄存器之一，这意味着几乎任何指令都可以改变控制流。
  - 更糟糕的是，程序计数器的最低有效位反映ISA当前正在执行(ARM或Thumb)哪个ISA——简单的ADD指令可以更改ISA当前在处理器上执行的指令!
  - 分支使用条件码以及谓词指令进一步使高性能实现复杂化。

ARMv8

- **2011年，ARM发布新的ISA ARMv8**
  - 64位地址; 扩展了整型寄存器组；32位指令。
  - 摒弃了ARMv7中实现复杂的一些特性
    - PC不再能直接访问;
    - 不再有谓词指令
    - 删除了load-multiple和store-multiple 指令
    - 指令编码归一化
- **主要问题**
  - 使用条件码 (比以前少了很多)
  - 存在许多特殊的寄存器 (zero, PC, sp, program status registers ,etc)

ARMv8

- 增加了一些缺陷，包括大量的subword-SIMD架构
- 指令集更加厚重：**1070条指令，53种格式，8种寻址方式。说明文档达到了5778页**
- 与其他大多数ISA一样，通常以暴露底层实现的方式将用户和特权架构紧密地结合在一起
- 此外，随着ARMv8的引入，ARM不再支持压缩指令编码
- 最后，和它的以前版本一样，ARMv8也是一个封闭的标准

OpenRISC

- **OpenRISC项目是一个开放源码处理器设计项目**
  - 来源于Hennessy和Patterson的体系结构教科书。
  - 适用于教学、科研和工业界的实现。
- **主要问题** (Pease read https://riscv.org/2014/10/why-not-build-on-openrisc/)
  - OpenRISC项目主要是开源处理器设计项目，而不是开源的ISA 规格说明，ISA和实现是紧密耦合的
  - 固定的32位编码与16位立即数阻碍了压缩ISA扩展
  - 没有64位的支持
  - Delay slots
  - 不能支持压缩指令
  - 值得一提的是：2010年这两个问题都得到了解决:延迟插槽已经成为可选的，64位版本已经定义(但是，据我们所知，从未实现过)。最终，我们（UCB）认为最好从头开始，而不是相应地修改OpenRISC。

80x86

- 在过去的四十年里，英特尔的8086架构已经成为笔记本电脑、台式机和服务器市场上最流行的指令集。
  - 在嵌入式系统领域之外，几乎所有流行的软件都被移植到x86上，或者是为x86开发的
  - 它受欢迎的原因有很多：该架构在IBM PC诞生之初的偶然可用性; 英特尔专注于二进制兼容性； 它们积极的卓有成效的微结构实现; 以及他们的前沿制造技术
  - **指令集设计质量并不是它流行的原因之一。**
- 主要问题：
  - **1300条指令**，许多寻址方式，很多特殊寄存器，多种地址翻译方式，**从AMD K5微架构开始，所有的Intel支持乱序执行的微结构，都是动态地将x86指令翻译为内部的RISC-风格的指令集。**
  - ISA不利于虚拟化，因为一些特权指令在用户模式下会无声地失败，而不是被捕获。VMware的工程师们用复杂的动态二进制翻译软件解决了这一缺陷
  - ISA的指令长度为任意整数字节数，最多为15个字节，但是数量较少的短操作码已经被随意使用

80x86

- ISA有数量极少的寄存器组
- 大多数整数寄存器在ISA中执行特殊功能，这加剧了体系结构寄存器的不足
- 一些ISA特性，包括隐式条件代码和带有谓词的移动操作，在微架构中实现复杂

- 这些ISA决策对静态代码大小有深刻的影响。尽管存在所有这些缺陷，x86通常比RISC体系结构使用更少的动态指令完成相同的功能，因为x86指令可以编码多个基本操作。
- 最后，80x86是一个专有指令集

ISA Summary

	MIPS	SPARC	Alpha	ARMv7	ARMv8	OpenRISC	80x86
Free and Open		√				√	
64-bit Address	√	√	√		√	√	√
Compressed Instructions	√			√			Partial
Separate Privileged ISA			√				
Position-Indep. Code	Partial			√	√		√
IEEE 754-2008					√		√
Classically Virtualizable	√	√	√		√		

ISA的功能设计

CISC vs RISC

- **ISA的功能设计**
  - 任务：确定硬件支持哪些操作
  - 方法：统计的方法
  - 两种类型：CISC和RISC
- **CISC（Complex Instruction Set Computer)**
  - 目标：强化指令功能，减少运行的指令条数，提高系统性能
  - 方法：面向目标程序的优化，面向高级语言和编译器的优化
- **RISC（Reduced Instruction Set Computer)**
  - 目标：通过简化指令系统，用高效的方法实现最常用的指令
  - 方法：充分发挥流水线的效率，降低（优化）CPI

CISC计算机ISA的功能设计

- **目标：强化指令功能，减少指令条数，以提高系统性能**
- **基本优化方法**

1. 面向目标程序的优化是提高计算机系统性能最直接方法

- 优化目标
  - 缩短程序的长度
  - 缩短程序的执行时间
- 优化方法
  - 统计分析目标程序执行情况，找出使用频度高，执行时间长的指令或指令串
  - 优化使用频度高的指令
  - 用新的指令代替使用频度高的指令串

优化目标程序的主要途径（1/2）

1）增强运算型指令的功能

如sin(x), cos(x), SQRT(X)，甚至多项式计算

如用一条三地址指令完成

$P(X) = C(0)+C(1)X+C(2)X^2+C(3)X^3+\ldots\ldots$

2) 增强数据传送类指令的功能

主要是指数据块传送指令

- R-R, R-M, M-M之间的数据块传送可有效的支持向量和矩阵运算，如IBM370
- R-Stack之间设置数据块传送指令，能够在程序调用和程序中断时，快速保存和恢复程序现场，如 VAX-11

优化目标程序的主要途径（2/2)

3) 增强程序控制指令的功能

在CISC中，一般均设置了多种程序控制指令，正常仅需要转移指令和子程序控制指令

2. 面向高级语言和编译程序改进指令系统

优化目标：主要是缩小HL-ML之间的差距

优化方法：

1） 增强面向HL和Compiler支持的指令功能

- 统计分析源程序中各种语句的使用频度和执行时间
- 增强相关指令的功能，优化使用频度高、执行时间长的语句
- 增加专门指令，以缩短目标程序长度，减少目标程序执行时间，缩短编译时间

FORTRAN语言和COBOL语言中各种主要语句的使用频度

语言	一元赋值	其他赋值	IF	GOTO	I/O	DO	CALL	其他
FORTRAN	31.0	15.0	11.5	10.5	6.5	4.5	6.0	15.0
COBOL	42.1	7.5	19.1	19.1	8.46	0.17	0.17	3.4

观察结果：

（1）一元赋值在其中比例最大，增强数据传送类指令功能，缩短这类指令的执行时间是对高级语言非常有力的支持，

（2）其他赋值语句中，增1操作比例较大，许多机器都有专门的增1指令

（3）条件转移和无条件转移占22%，38.2%,因此增强转移指令的功能，增加转移指令的种类是必要的

2）高级语言计算机系统

缩小HL和ML的差别时，走到极端，就是把HL和ML合二为一，即所谓的高级语言计算机。在这种机器中，高级语言不需要经过编译，直接由机器硬件来执行。如LISP机，PROLOG机

3）支持操作系统的优化实现 - 些特权指令

任何一种计算机系统通常需要操作系统，而OS又必须用指令系统来实现，指令系统对OS的支持主要有

- 处理器工作状态和访问方式的转换
- 进程的管理和切换
- 存储管理和信息保护
- 进程同步和互斥，信号量的管理等

RISC的定义和特点

- **RISC是一种计算机体系结构的设计思想，它不是一种产品。RISC是近代计算机体系结构发展史中的一个里程碑，直到现在，RISC没有一个确切的定义**
- **早期对RISC特点的描述**
  - 大多数指令在单周期内完成
  - 采用Load/Store结构
  - 硬布线控制逻辑
  - 减少指令和寻址方式的种类
  - 固定的指令格式
  - 注重代码的优化
- **从目前的发展看，RISC体系结构还应具有如下特点：**
  - 面向寄存器结构
  - 十分重视流水线的执行效率 - 尽量减少断流
  - 重视优化编译技术
- **减少指令平均执行周期数 CPI 是RISC思想的精华**

问题

RISC的指令系统精简了，CISC中的一条指令可能由一串指令才能完成，那么为什么RISC执行程序的速度比CISC还要快?

ExecuteTime = CPI X IC X T

	IC	CPI	T
CISC	1	2~15	33ns~5ns
RISC	1.3~1.4	1.1~1.4	10ns~2ns

IC：实际统计结果，RISC的IC只比CISC 长30%~40%

CPI: CISC CPI一般在4~6之间，RISC 一般CPI =1，Load/Store 为2

T: RISC采用硬布线逻辑，指令要完成的功能比较简单

RISC为什么会减少CPI

- **硬件方面：**
  - 硬布线控制逻辑
  - 减少指令和寻址方式的种类
  - 使用固定格式
  - 采用Load/Store
  - 指令执行过程中设置多级流水线。
- **软件方面：十分强调优化编译的作用**

小结

- **ISA的功能设计：任务为确定硬件支持哪些操作。方法是统计的方法。存在CISC和RISC两种类型**
  - CISC（Complex Instruction Set Computer)
    - 目标：强化指令功能，减少指令的指令条数，以提高系统性能
    - 基本方法：面向目标程序的优化，面向高级语言和编译器的优化
  - RISC（Reduced Instruction Set Computer)
    - 目标：通过简化指令系统，用最高效的方法实现最常用的指令
    - 主要手段：充分发挥流水线的效率，降低（优化）CPI
- **控制转移类指令**
- **指令编码（指令格式）**

ISA的演进

Single Accumulator (EDSAC 1950)

Accumulator + Index Registers (Manchester Mark I, IBM 700 series 1953)

Separation of Programming Model from Implementation

High-level Language Based (B5000 1963)

Concept of a Family (IBM 360 1964)

General Purpose Register Machines

Complex Instruction Sets，CISC (Vax, Intel 8086 1977-80)

Load/Store Architecture (CDC 6600, Cray 1 1963-76)

VLIW

Cydra-5 1983, Multiflow 1984, TMSC6X 1996, Intel EPIC 1997

RISC

(Mips,Sparc, IBM RS6000, . . .1987+)

RISC-V ISA

- UC Berkeley 设计的第5代RISC指令集
- 设计理念（指导思想）：通用的ISA
  - 能适应从最袖珍的嵌入式控制器，到最快的高性能计算机等各种规模的处理器。
  - 能兼容各种流行的软件栈和编程语言。
  - 适应所有实现技术，包括现场可编程门阵列（FPGA）、专用集成电路（ASIC）、全定制芯片，甚至未来的技术。
  - 对所有微体系结构实现方式都有效。例如：
    - 微编码或硬连线控制；顺序或乱序执行流水线；单发射或超标量等等。
  - 支持广泛的定制化，成为定制加速器的基础。随着摩尔定律的消退，加速器的重要性日益提高。
  - 基础的指令集架构是稳定的。不能像以前的专有指令集架构一样被弃用，例如AMD Am29000、Digital Alpha、Digital VAX、Hewlett Packard PA-RISC、Intel i860、Intel i960、Motorola 88000、以及Zilog Z8000。
  - 完全开源

技术目标

- **将ISA分成基础ISA和可选的扩展部分**
  - ISA的基础足够简单，可以用于教学和许多嵌入式处理器，包括定制加速器的控制单元。它足够完整，可以运行现代软件栈。
  - 扩展部分提高计算的性能，并支持多处理机并行
  - 支持32位和64位地址空间
- **方便定制ISA扩展**
  - 包括紧耦合功能单元和松耦合协处理器
- **支持变长指令集扩展**
  - 既为了提高代码密度，也为了扩展可能的自定义ISA扩展的空间
- **提供对现代标准的有效硬件支持**
- **用户级ISA和特权级ISA是正交的（相互独立，互不依赖）**
  - 在保持用户应用程序二进制接口(ABI)兼容性的同时，允许完全虚拟化，并允许在特权ISA中进行实验测试

RISC-V ISA的特点

- **完全开源：**
  - 它属于一个开放的，非营利性质的RISC-V基金会。
  - 开源采用BSD协议（企业完全自由免费使用，允许企业添加自有指令而不必开放共享以实现差异化发展）
- **架构简单**
  - 没有针对某一种微体系结构实现方式做过度的架构设计
  - 新的指令集，没有向后（backward）兼容的包袱
  - 说明书的页数.....（图1.6）
- **模块化的指令集架构**
  - RV32I和RV64I是基础的ISA。可扩展增加其他特性的支持
  - 面向教育或科研，易于扩充或剪裁
  - 支持32位和64位地址空间
- **面向多核并行**

ISA	Pages	Words	Hours to read	Weeks to read
RISC-V	236	76,702	6	0.2
ARM-32	2736	895,032	79	1.9
x86-32	2198	2,186,259	182	4.5

图1.6：ISA手册的页数和字数来自[Waterman and Asanovi'c 2017a]，[Waterman and Asanovi'c 2017b]，[Intel Corporation 2016]，[ARM Ltd. 2014]。读完需要的时间按每分钟读200个单词，每周读40小时计算。 基于[Baumann 2017]的图1的一部分。

RISC-V子集命名约定

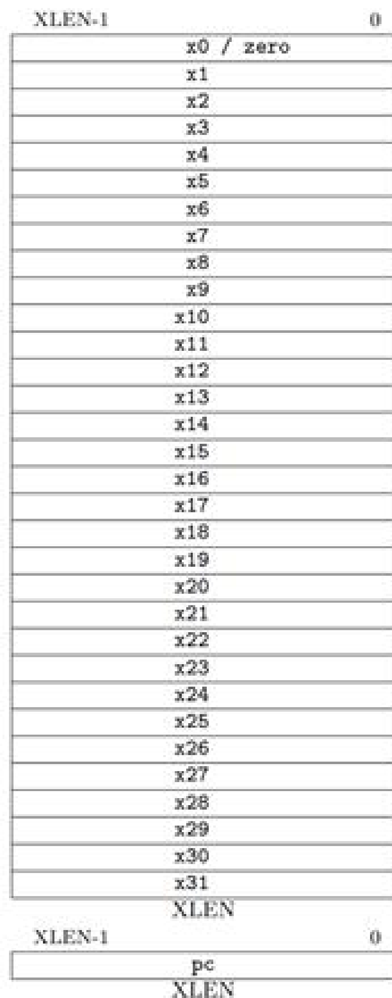
ISA base and extensions

Name	Description	Version	Status[a]
Base			
RV32I	Base Integer Instruction Set, 32-bit	2.1	Frozen
RV32E	Base Integer Instruction Set (embedded), 32-bit, 16 registers	1.9	Open
RV64I	Base Integer Instruction Set, 64-bit	2.0	Frozen
RV128I	Base Integer Instruction Set, 128-bit	1.7	Open
Extension			
M	Standard Extension for Integer Multiplication and Division	2.0	Frozen
A	Standard Extension for Atomic Instructions	2.0	Frozen
F	Standard Extension for Single-Precision Floating-Point	2.0	Frozen
D	Standard Extension for Double-Precision Floating-Point	2.0	Frozen
G	Shorthand for the base and above extensions	N/A	N/A
Q	Standard Extension for Quad-Precision Floating-Point	2.0	Frozen
L	Standard Extension for Decimal Floating-Point	0.0	Open
C	Standard Extension for Compressed Instructions	2.0	Frozen
B	Standard Extension for Bit Manipulation	0.92	Open
J	Standard Extension for Dynamically Translated Languages	0.0	Open
T	Standard Extension for Transactional Memory	0.0	Open
P	Standard Extension for Packed-SIMD Instructions	0.1	Open
V	Standard Extension for Vector Operations	0.8	Open
N	Standard Extension for User-Level Interrupts	1.1	Open
H	Standard Extension for Hypervisor	0.4	Open

a. ^ Frozen parts are expected to have their final feature set and to receive only clarifications before being ratified.

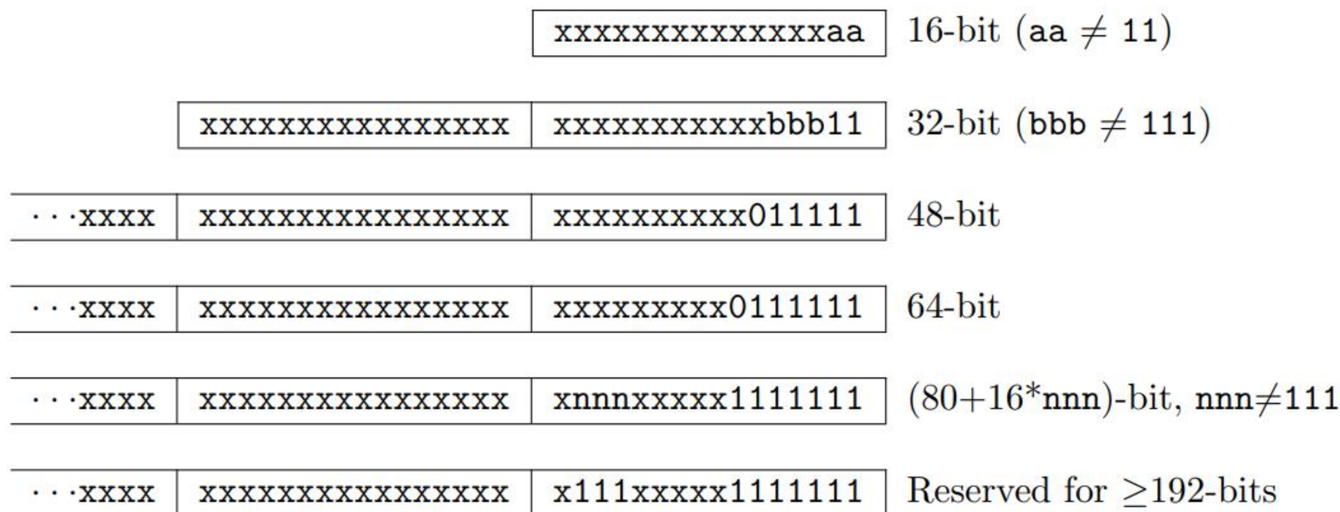
RV32 处理器状态

- 32x32-bit 整型数寄存器 (**x0-x31**)
  - **x0 总是** 0
  - 原则上其他寄存器可以通用
  - 不过software convention如右图所示



Register name	Symbolic name	Description	Owner
32 integer registers			
x0	Zero	Always zero	
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	
x4	tp	Thread pointer	
x5	t0	Temporary / alternate return address	Caller
x6–7	t1–2	Temporary	Caller
x8	s0/fp	Saved register / frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function argument / return value	Caller
x12–17	a2–7	Function argument	Caller
x18–27	s2–11	Saved register	Callee
x28–31	t3–6	Temporary	Caller
32 floating-point extension registers			
f0–7	ft0–7	Floating-point temporaries	Caller
f8–9	fs0–1	Floating-point saved registers	Callee
f10–11	fa0–1	Floating-point arguments/return values	Caller
f12–17	fa2–7	Floating-point arguments	Caller
f18–27	fs2–11	Floating-point saved registers	Callee
f28–31	ft8–11	Floating-point temporaries	Caller

RISC-V 指令编码



- 可支持变长指令
- 基础指令集 (RV32) 总是固定的32-bit指令（lowest two bits = 11_2)
- 所有的条件转移和无条件转移的转移地址16-bit对齐

RV32I 指令格式

Additional opcode bits/immediate

Reg. Source 2

Reg. Source 1

Destination Reg.

7-bit opcode field (but low 2 bits =11_2)

31	30 … 25	24 … 21	20	19 … 15	14 … 12	11 … 8	7	6 … 0	
funct7		rs2		rs1	funct3	rd		opcode	R-type
imm[11:0]				rs1	funct3	rd		opcode	I-type
imm[11:5]		rs2		rs1	funct3	imm[4:0]		opcode	S-type
imm[12]	imm[10:5]	rs2		rs1	funct3	imm[4:1]	imm[11]	opcode	B-type
imm[31:12]						rd		opcode	U-type
imm[20]	imm[10:1]		imm[11]	imm[19:12]		rd		opcode	J-type

RV32I Integer-Immediate (I)

31 – 20	19 – 15	14 – 12	11 – 7	6 – 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
I-immediate[11:0]	src	ADDI/SLTI[U]	dest	OP-IMM
I-immediate[11:0]	src	ANDI/ORI/XORI	dest	OP-IMM

`ADDI` adds the sign-extended 12-bit immediate to register `rs1`. Arithmetic overflow is ignored and the result is simply the low `XLEN` bits of the result.

`ADDI rd, rs1, 0` is used to implement the `MV rd, rs1` assembler pseudo-instruction.

RV32I Integer-Immediate (I)

31 — 25	24 — 20	19 — 15	14 — 12	11 — 7	6 — 0
imm[11:5]	imm[4:0]	rs1	funct3	rd	opcode
7	5	5	3	5	7
0000000	shamt[4:0]	src	SLLI	dest	OP-IMM
0000000	shamt[4:0]	src	SRLI	dest	OP-IMM
0100000	shamt[4:0]	src	SRAI	dest	OP-IMM

Shifts by a constant are encoded as a specialization of the I-type format. The operand to be shifted is in `rs1`, and the `shift amount` is encoded in the lower 5 bits of the `I-immediate` field. The right shift type is encoded in bit 30.

`SLLI` is a logical left shift (zeros are shifted into the lower bits);
`SRLI` is a logical right shift (zeros are shifted into the upper bits);
`SRAI` is an arithmetic right shift (the original sign bit is copied into the vacated upper bits).

RV32I Register-Register (R)

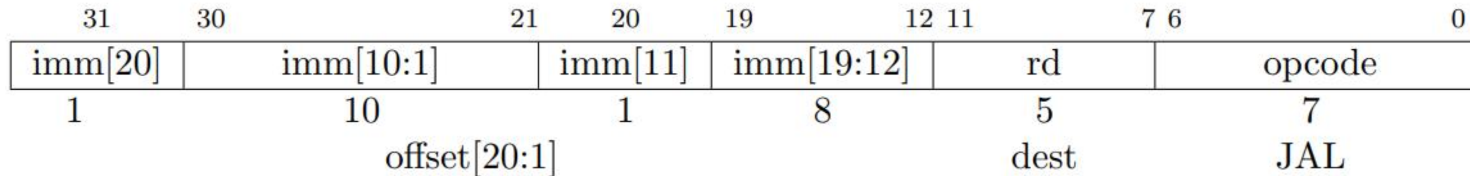
31–25	24–20	19–15	14–12	11–7	6–0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
0000000	src2	src1	ADD/SLT/SLTU	dest	OP
0000000	src2	src1	AND/OR/XOR	dest	OP
0000000	src2	src1	SLL/SRL	dest	OP
0100000	src2	src1	SUB/SRA	dest	OP

`ADD` performs the addition of `rs1` and `rs2`. `SUB` performs the subtraction of `rs2` from `rs1`. Overflows are ignored and the low `XLEN` bits of results are written to the destination `rd`.

`SLT` and `SLTU` perform signed and unsigned compares respectively, writing `1` to `rd` if `rs1 < rs2`, `0` otherwise.

`AND`, `OR`, and `XOR` perform bitwise logical operations.
`SLL`, `SRL`, and `SRA` perform logical left, logical right, and arithmetic right shifts on the value in register `rs1` by the shift amount held in the lower 5 bits of register `rs2`.

RV32I Unconditional Jump (J)

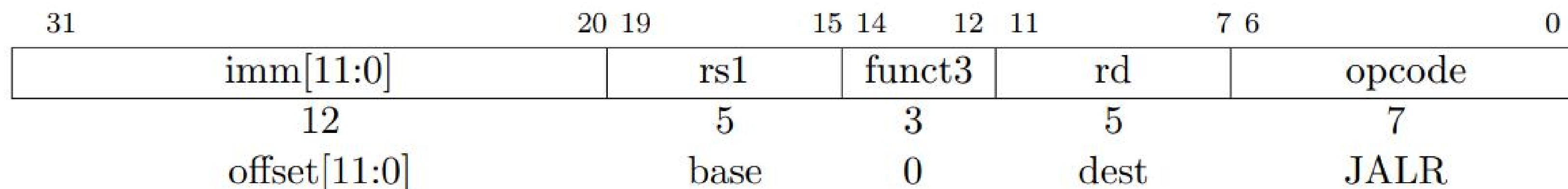


Plain unconditional jumps (assembler pseudoinstruction J) are encoded as a `JAL` with `rd=x0`.

The jump and link (`JAL`) instruction uses the `J`-type format, where the `J` -immediate encodes a signed offset in multiples of 2 bytes. The offset is sign-extended and added to the address of the jump instruction to form the jump target address. Jumps can therefore target a ±1 MiB range.

Position Independent!

RV32I Indirect Jump (I)



The indirect jump instruction `JALR` (jump and link register) uses the `I`-type encoding. The target address is obtained by adding the sign-extended 12-bit `I-immediate` to the register `rs1`, then setting the least-significant bit of the result to zero. The address of the instruction following the jump (`pc+4`) is written to register `rd`. Register `x0` can be used as the destination if the result is not required.

No architecturally visible delay slots!
MIPS Jal会执行延迟槽中的指令，然后再跳转

RV32I Conditional Branches (B)

31	30 25	24 20	19 15	14 12	11 8	7	6 0
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode
1	6	5	5	3	4	1	7
offset[12|10:5]		src2	src1	BEQ/BNE	offset[11|4:1]		BRANCH
offset[12|10:5]		src2	src1	BLT[U]	offset[11|4:1]		BRANCH
offset[12|10:5]		src2	src1	BGE[U]	offset[11|4:1]		BRANCH

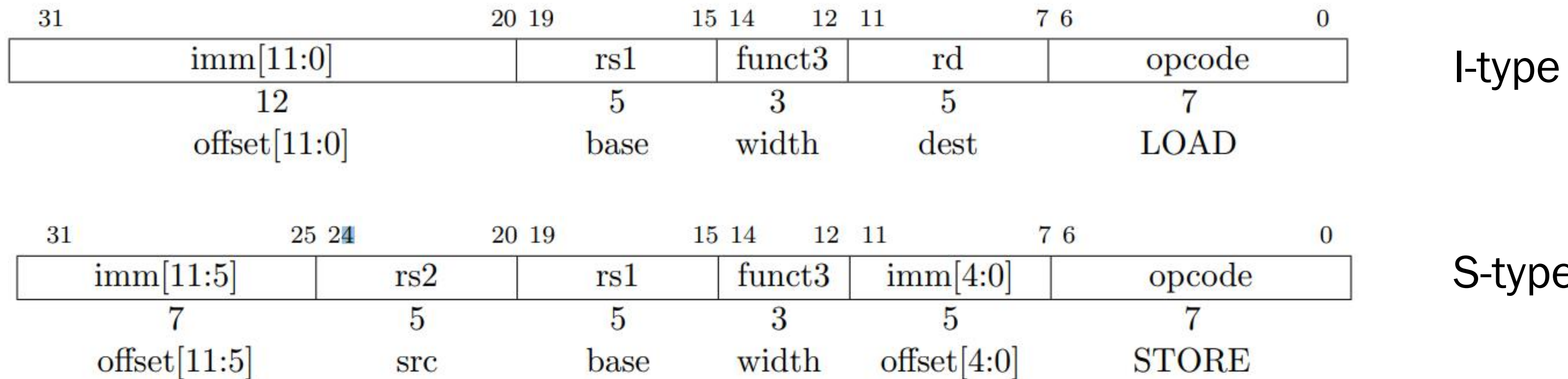
The 12-bit `B-immediate` encodes signed offsets in multiples of 2 bytes. The offset is sign-extended and added to the address of the branch instruction to give the target address. The conditional branch range is ±4 KiB.

Branch instructions compare two registers. `BEQ` and `BNE` take the branch if registers `rs1` and `rs2` are equal or unequal respectively.

`BLT` and `BLTU` take the branch if `rs1` is less than `rs2`, using signed and unsigned comparison respectively.

`BGE` and `BGEU` take the branch if `rs1` is greater than or equal to `rs2`, using signed and unsigned comparison respectively.

Integer Load/Store Instructions



`Load` and `store` instructions transfer a value between the registers and memory. Loads are encoded in the `I`-type format and stores are `S`-type. The effective address is obtained by adding register `rs1` to the sign-extended 12-bit offset. Loads copy a value from memory to register `rd`. Stores copy the value in register `rs2` to memory.

RV32I 带有指令布局，操作码，格式类型和名称的操作码映射

31 　 25	24 　 20	19 　 15	14 　 12	11 　 7	6 　 0	
imm[31:12]				rd	0110111	U lui
imm[31:12]				rd	0010111	U auipc
imm[20|10:1|11|19:12]				rd	1101111	J jal
imm[11:0]		rs1	000	rd	1100111	I jalr
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	B beq
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	B bne
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	B blt
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	B bge
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	B bltu
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	B bgeu
imm[11:0]		rs1	000	rd	0000011	I lb
imm[11:0]		rs1	001	rd	0000011	I lh
imm[11:0]		rs1	010	rd	0000011	I lw
imm[11:0]		rs1	100	rd	0000011	I lbu
imm[11:0]		rs1	101	rd	0000011	I lhu
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	S sb
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	S sh
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	S sw
imm[11:0]		rs1	000	rd	0010011	I addi
imm[11:0]		rs1	010	rd	0010011	I slti
imm[11:0]		rs1	011	rd	0010011	I sltiu
imm[11:0]		rs1	100	rd	0010011	I xori
imm[11:0]		rs1	110	rd	0010011	I ori
imm[11:0]		rs1	111	rd	0010011	I andi

RV32I 带有指令布局，操作码，格式类型和名称的操作码映射

31		25 24	20 19	15 14	12 11	7 6	0
0000000		shamt	rs1	001	rd	0010011	I slli
0000000		shamt	rs1	101	rd	0010011	I srli
0100000		shamt	rs1	101	rd	0010011	I srai
0000000		rs2	rs1	000	rd	0110011	R add
0100000		rs2	rs1	000	rd	0110011	R sub
0000000		rs2	rs1	001	rd	0110011	R sll
0000000		rs2	rs1	010	rd	0110011	R slt
0000000		rs2	rs1	011	rd	0110011	Rsltu
0000000		rs2	rs1	100	rd	0110011	R xor
0000000		rs2	rs1	101	rd	0110011	R srl
0100000		rs2	rs1	101	rd	0110011	R sra
0000000		rs2	rs1	110	rd	0110011	R or
0000000		rs2	rs1	111	rd	0110011	R and
0000	pred	succ	00000	000	00000	0001111	I fence
0000	0000	0000	00000	001	00000	0001111	I fence.i
000000000000			00000	00	00000	1110011	I ecall
000000000000			00000	000	00000	1110011	I ebreak
csr			rs1	001	rd	1110011	I csrrw
csr			rs1	010	rd	1110011	I csrrs
csr			rs1	011	rd	1110011	I csrrc
csr			zimm	101	rd	1110011	I csrrwi
csr			zimm	110	rd	1110011	I cssrrsi
csr			zimm	111	rd	1110011	I csrrci

微程序实现

思考

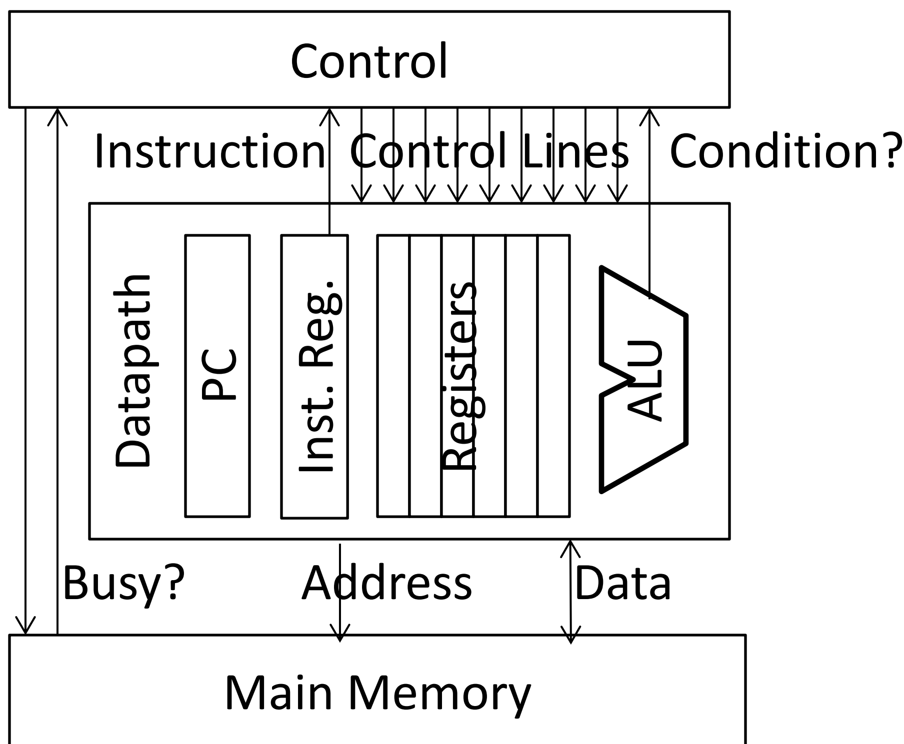
- **认真对照MIPS的每条指令与RV32I的指令。发现RV32I做出的一系列改动，并解释为什么做出这些改动。**
- **如果卡在某条指令上面，立即联系助教或者老师**

RISC-V 指令执行阶段

- **Instruction Fetch**
- **Instruction Decode**
- **Register Fetch**
- **ALU Operations**
- **Optional Memory Operations**
- **Optional Register Writeback**
- **Calculate Next Instruction Address**

控制部分与数据通路

- 处理器设计可以分为datapath和Control设计两部分
  - datapath, 存储数据、算术逻辑运算单元
  - control, 控制数据通路上的一系列操作



- 早期的计算机设计者的最大挑战是控制逻辑的正确性
- Maurice Wilkes 提出了微程序设计的概念来设计处理器的控制逻辑（EDSAC-II, 1958)
- 当时的技术水平
  - Logic: Vacuum Tubes
  - Main Memory: Magnetic cores
  - Read-Only Memory: Diode matrix, punched metal cards, …
  - Cost： Logic > RAM > ROM
  - Speed: ROM > RAM

微程序控制RISC-V的单总线数据通路

Opcode
rs1 rs2 rd 32 (PC)
Condition?
Busy?
InstLd ImmSel RegSel RegW BLd ALUOp MALd MemW
Instruction Reg.
Immediate
Address
Register RAM
PC
Registers
Data Out In
B
A
ALU
Mem. Address
Main Memory
ImmEn RegEn ALd ALUEn MemEn

微指令的寄存器传输级表示:

- MA:=PC means RegSel=PC; RegW=0; RegEn=1; MALd=1

微程序控制RISC-V的单总线数据通路

微指令的寄存器传输级表示:

- B:=Reg[rs2] means RegSel=rs2; RegW=0; RegEn=1; BLd=1

微程序控制RISC-V的单总线数据通路

Opcode
rs1
rs2
rd
32 (PC)
Condition?
Busy?
InstLd
ImmSel
RegSel
RegW
BLd
ALUOp
MALd
MemW
Address
Instruction Reg.
Immediate
Register RAM
PC
Registers
B
A
ALU
Mem. Address
Main Memory
Data Out In
ALd
ImmEn
RegEn
ALUEn
MemEn

微指令的寄存器传输级表示:

- Reg[rd]:=A+B means ALUop=Add; ALUEn=1; RegSel=rd; RegW=1

微程序控制 CPU

Microcode示意 (1)

Instruction Fetch:	**MA,A:=PC** **PC:=A+4** ***wait for memory*** **IR:=Mem** ***dispatch on opcode***
ALU:	**A:=Reg[rs1]** **B:=Reg[rs2]** **Reg[rd]:=ALUOp(A,B)** ***goto instruction fetch***
ALUI:	**A:=Reg[rs1]** **B:=ImmI** **//Sign-extend 12b immediate** **Reg[rd]:=ALUOp(A,B)** ***goto instruction fetch***

Microcode 示意 (2)

LW:

A:=Reg[rs1]
B:=ImmI //Sign-extend 12b immediate
MA:=A+B
wait for memory
Reg[rd]:=Mem
goto instruction fetch

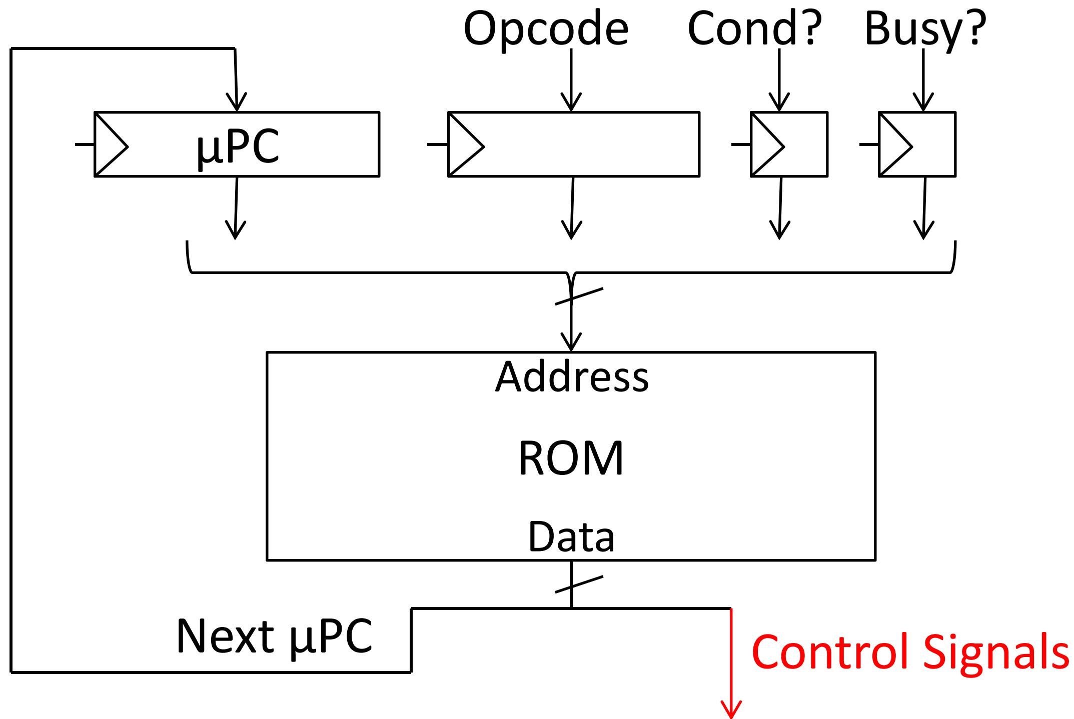
JAL:

Reg[rd]:=A // Store return address
A:=A-4 // Recover original PC
B:=ImmJ // Jump-style immediate
PC:=A+B
goto instruction fetch

Branch:

A:=Reg[rs1]
B:=Reg[rs2]
if (!ALUOp(A,B)) ***goto instruction fetch*** **//Not taken**
A:=PC //Microcode fall through if branch taken
A:=A-4
B:=ImmB// Branch-style immediate
PC:=A+B
goto instruction fetch

采用 ROM 实现微程序控制



- **How many address bits?**
  |μaddress| = |μPC|+|opcode|+ 1 + 1
- **How many data bits?**
  |data| = |μPC|+|control signals| = |μPC| + 18
- **Total ROM size = $2^{|\mu address|}$x|data|**

ROM 中的内容

Address				Data	
μPC	**Opcode**	**Cond?**	**Busy?**	**Control Lines**	**Next μPC**
fetch0	X	X	X	MA,A:=PC	fetch1
fetch1	X	X	1		fetch1
fetch1	X	X	0	IR:=Mem	fetch2
fetch2	ALU	X	X	PC:=A+4	ALU0
fetch2	ALUI	X	X	PC:=A+4	ALUI0
fetch2	LW	X	X	PC:=A+4	LW0
….					
ALU0	X	X	X	A:=Reg[rs1]	ALU1
ALU1	X	X	X	B:=Reg[rs2]	ALU2
ALU2	X	X	X	Reg[rd]:=ALUOp(A,B)	fetch0

单总线数据通路结构的微程序控制存储器大小

- 取指阶段有3个common steps
- 指令分为12 组
- 完成一条指令需要5条微指令（包括dispatch）
- 共计 3+12*5 = 63条微指令,因此 μPC需要6位

- 指令操作码（Opcode)5 位, 每条微指令~18个控制信号

- 微控制器的大小 = $2^{(6+5+2)}$x(6+18)=2^{13}x24 = ~25KiB!

Reducing Control Store Size

- Reduce ROM height (#address bits)
  - Use external logic to combine input signals
  - Reduce #states by grouping opcodes
- Reduce ROM width (#data bits)
  - Restrict µPC encoding (next,dispatch,wait on memory,...)
  - Encode control signals (vertical µcoding, nanocoding)

单总线 RISC-V 微程序控制引擎

Reducing Control Store Size

$|\mu address| = |\mu PC| + |opcode| + 1 + 1$

$|data| = |\mu PC| + |control\ signals|$

Total ROM size = $2^{|\mu address|} \times |data|$

μPC jump = next | spin | fetch | dispatch | ftrue | ffalse

μPC Jump 类型

- ***next***： **increments μPC**
- ***spin***： **waits for memory**
- ***fetch***： **jumps to start of instruction fetch**
- ***dispatch***： **jumps to start of decoded opcode group**
- ***ftrue/ffalse***： **jumps to fetch if Cond? true/false**

微程序控制存储器ROM中的内容

Address	Data	
µPC	Control Lines	Next µPC
fetch0	MA,A:=PC	next
fetch1	IR:=Mem	spin
fetch2	PC:=A+4	dispatch
ALU0	A:=Reg[rs1]	next
ALU1	B:=Reg[rs2]	next
ALU2	Reg[rd]:=ALUOp(A,B)	fetch
Branch0	A:=Reg[rs1]	next
Branch1	B:=Reg[rs2]	next
Branch2	A:=PC	ffalse
Branch3	A:=A-4	next
Branch4	B:=ImmB	next
Branch5	PC:=A+B	fetch

示例：实现一条复杂指令

Memory-memory add: M[rd] = M[rs1] + M[rs2]

Address	Data	
µPC	Control Lines	Next µPC
MMA0	MA:=Reg[rs1]	next
MMA1	A:=Mem	spin
MMA2	MA:=Reg[rs2]	next
MMA3	B:=Mem	spin
MMA4	MA:=Reg[rd]	next
MMA5	Mem:=ALUOp(A,B)	spin
MMA6		fetch

复杂指令的实现通常不需要修改数据通路，仅仅需要编写相应的微程序（可能会占用更多的控存）

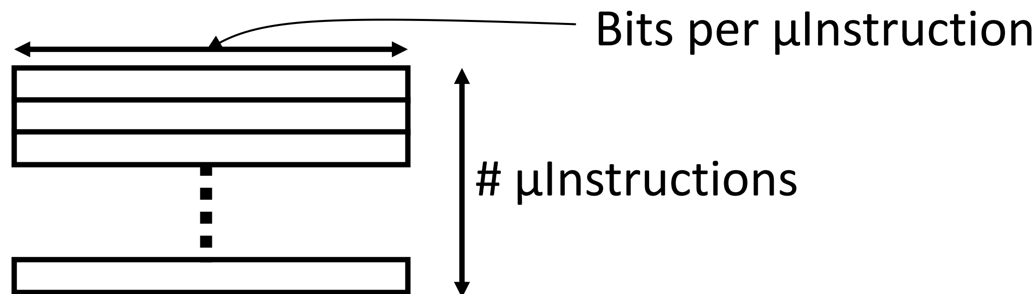
采用硬布线控制器而不修改数据通路来实现这些指令是非常困难的

Single-Bus Datapath for Microcoded RISC-V

Opcode
rs1
rs2
rd
32 (PC)
Condition?
Busy?
InstLd
ImmSel
RegSel
RegW
BLd
ALUOp
MALd
MemW
Instruction Reg.
Immediate
Address
Register RAM
PC
Registers
Data Out In
B
A
ALU
Mem. Address
Main Memory
ALd
ImmEn
RegEn
ALUEn
MemEn

Datapath unchanged for complex instructions!

Horizontal vs Vertical µCode



- **水平微编码的水平型微指令**
  - 每条微指令有多个微操作并行
  - 每条宏指令（指令）具有较少的微指令
  - 稀疏的（微操作）编码 ⇒ 微指令较宽（含有较多的位数）
- **垂直微编码的垂直型微指令**
  - 典型的是每条微指令代表一个数据通路操作
    - 不同的数据通路分支是独立的微指令
  - 每条宏指令（指令）需要更多的微指令
  - 紧凑的（微指令）编码 ⇒ 微指令较窄（含有较少的位数）
- **Nanocoding**
  - 水平型微指令和垂直型微指令的结合

Reduce ROM width (#data bits)

Nanocoding

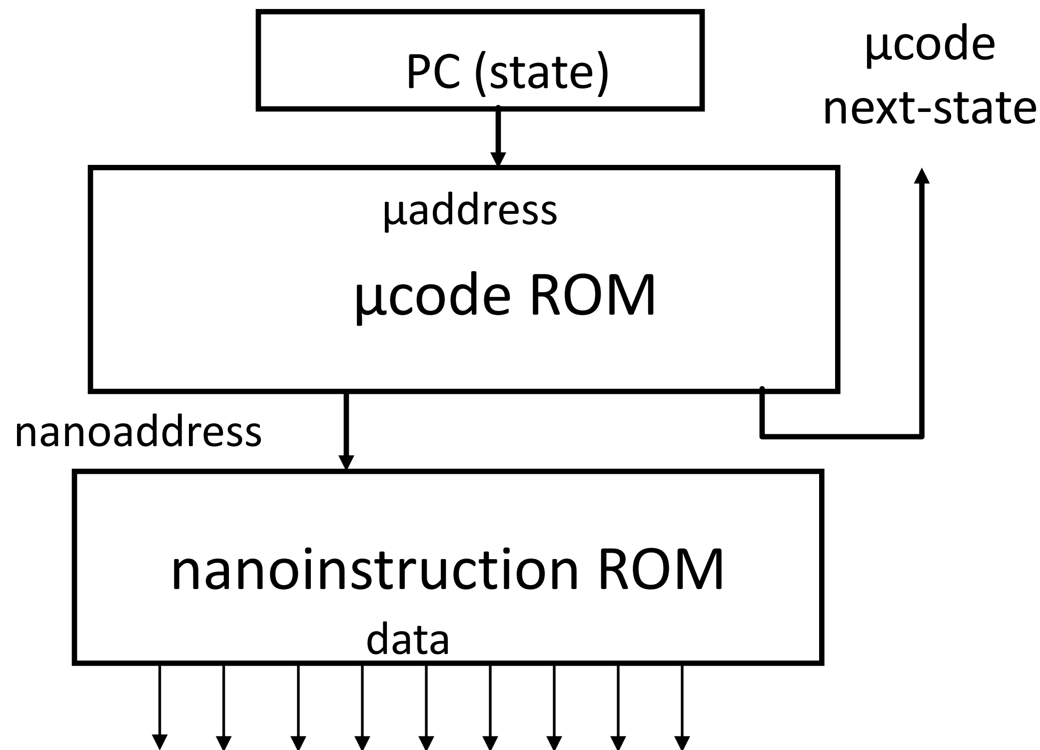
利用微代码中重复的控制信号 e.g.

ALU0 A   Reg[rs1]

...

ALUI0 A   Reg[rs1]

...



- **Motorola 68000 had 17-bit μcode containing either 10-bit μjump or 9-bit nanoinstruction pointer**
  - Nanoinstructions were 68 bits wide, decoded to give 196 control signals

Microprogramming in IBM 360

	M30	M40	M50	M65
Datapath width (bits)	8	16	32	64
µinst width (bits)	50	52	85	87
µcode size (K µinsts)	4	4	2.75	2.75
µstore technology	CCROS	TCROS	BCROS	BCROS
µstore cycle (ns)	750	625	500	200
memory cycle (ns)	1500	2500	2000	750
Rental fee ($K/month)	4	7	15	35

- **Only the fastest models (75 and 95) were hardwired**

IBM Card-Capacitor Read-Only Storage

Punched Card with metal film

Fixed sensing plates

[IBM Journal, January 1961]

Microcode Emulation

- **在推出IBM 360系列机之前，IBM最初错误地估计了与早期机器的软件兼容性的重要性**
- **Honeywell为H200系列机器提供翻译软件（“Liberator”），抢走了一些IBM 1401客户**
- **IBM对此进行了反击，为360系列增加了可选的微代码，该微代码可以模拟IBM 1401 ISA，后来又扩展到IBM 7000系列**
  - 1401上一个常用的程序是650模拟器，一些客户在1401上模拟运行650的多个程序
  - （650在1401上模拟，1401在360上模拟）

60到70年代微程序盛行

- **ROM比DRAM要快的多**
  - 逻辑器件（电子管）、主存（磁芯存储器）、ROM（二极管）
  - ROM和RAM速度之间的差异导致了额外的复杂指令
- **对于复杂的指令集，datapath和controller更便宜、更简单**
  - 新的指令（例如 floating point）可以在不修改数据通路的情况下增加
  - 修改控制器的bug更容易
- **不同型号的机器实现ISA的兼容性更简单、成本更低**

除了低档的或者性能最高机器，所有计算机都采用微程序控制

80年代初的微程序技术

- **微程序技术的进展孕育了更复杂的微程序控制的机器**
  - 复杂指令集导致μcode需要子程序和调用堆栈
  - 需要修复控制程序中的bug与μROM 的只读属性冲突
  - ➔Writable Control Store (WCS) (B1700, QMachine, Intel i432, …)
- **随着超大规模集成电路技术的出现，有关ROM和RAM速度的假设变得无效**
  - 逻辑部件、存储部件（RAM, ROM）均采用MOS晶体管实现
  - 半导体RAM与ROM的存取速度相同
- **随着编译器技术的进步复杂指令变得不再那么重要**
- **随着微结构技术的进步（pipeling, caches and buffers）,使得多周期执行reg-reg指令失去了吸引力**

Writable Control Store (WCS)

- **使用RAM实现控制存储**
  - MOS SRAM内存几乎和控制存储一样快
    - 过去 磁芯存储器/DRAM 要比控制存储器慢2-10倍
  - 要写出没有bug的微程序是很困难的
  - 使用可修改的RAM存储微程序有利于微程序的维护
  - 一些小型机提供了User-WCS选项。即允许用户修改微指令
- **User-WCS 失败的原因**
  - 几乎没有编程工具支持
  - 很难将应用软件装入很小的WCS中
  - 在用户级别使用微程序控制来模拟原来的ISA，对其他人用处不大
  - 大量的WCS空间用来保存处理器状态导致上下文切换代价昂贵
  - 如果用户改变微代码环境保护很困难
  - 虚拟存储要求微代码具有restartable能力

VAX 11-780 Microcode

```
; P1WFUD.!  )[600,1205]    MICRO2  1F(12)   26-May-81  14:58:1     VAX11/780 Microcode : PCS 01, FPLA 0D, WCS122      age  771
; CALL2 .MIC [600,1205]    Procedure call     : CALLG, CALLS

                                              ;29744  ;HERE FOR CALLG OR CALLS, AFTER PROBING THE EXTENT OF THE STACK
                                              ;29745
                                              ;29746  =0      ;------------------------------;CALL SITE FOR MPUSH
                                              ;29747  CALL.7: D_Q.AND.RC[T2],                 ;STRIP MASK TO BITS 11-0
6557K     0  U 11F4, 0811,2035,0180,F910,0000,0CD8  ;29748        CALL,J/MPUSH                      ;PUSH REGISTERS
                                              ;29749
                                              ;29750          ;------------------------------;RETURN FROM MPUSH
                                              ;29751          CACHE_D[LONG],                  ;PUSH PC
6557K 7763K U 11F5, 0000,003C,0180,3270,0000,134A  ;29752        LAB_R[SP]                         ; BY SP
                                              ;29753
                                              ;29754          ;------------------------------;
6856K     0  U 134A, 0018,0000,0180,FAF0,0200,134C  ;29755  CALL.8: R[SP]&VA_LA-K[.8]              ;UPDATE SP FOR PUSH OF PC &
                                              ;29756
                                              ;29757          ;------------------------------;
6856K     0  U 134C, 0800,003C,0180,FA68,0000,11F8  ;29758        D_R[FP]                           ;READY TO PUSH FRAME POINTER
                                              ;29759
                                              ;29760  =0      ;------------------------------;CALL SITE FOR PSHSP
                                              ;29761          CACHE_D[LONG],                  ;STORE FP,
                                              ;29762          LAB_R[SP],                      ; GET SP AGAIN
                                              ;29763          SC_K[.FFF0],                    ;-16 TO SC
6856K   21M  U 11F8, 0000,003D,6D80,3270,0084,6CD9  ;29764        CALL,J/PSHSP
                                              ;29765
                                              ;29766          ;------------------------------;
                                              ;29767          D_R[AP],                        ;READY TO PUSH AP
6856K     0  U 11F9, 0800,003C,3DF0,2E60,0000,134D  ;29768        Q_ID[PSL]                         ; AND GET PSW FOR COMBINATIO
                                              ;29769
                                              ;29770          ;------------------------------;
                                              ;29771          CACHE_D[LONG],                  ;STORE OLD AP
                                              ;29772          Q_Q.ANDNOT.K[.1F],              ;CLEAR PSW<T,N,Z,V,C>
6856K   21M  U 134D, 0019,2024,8DC0,3270,0000,134E  ;29773        LAB_R[SP]                         ;GET SP INTO LATCHES AGAIN
                                              ;29774
                                              ;29775          ;------------------------------;
6856K     0  U 134E, 2010,0038,0180,F909,4200,1350  ;29776        PC&VA_RC[T1], FLUSH.IB            ; LOAD NEW PC AND CLEAR OUT
                                              ;29777
                                              ;29778          ;------------------------------;
                                              ;29779          D_DAL.SC,                       ;PSW TO D<31:16>
                                              ;29780          Q_RC[T2],                       ;RECOVER MASK
                                              ;29781          SC_SC+K[.3],                    ;PUT -13 IN SC
6856K     0  U 1350, 0D10,0038,0DC0,6114,0084,9351  ;29782        LOAD.IB, PC_PC+1                  ;START FETCHING SUBROUTINE I
                                              ;29783
                                              ;29784          ;------------------------------;
                                              ;29785          D_DAL.SC,                       ;MASK AND PSW IN D<31:03>
                                              ;29786          Q_RC[T4],                       ;GET LOW BITS OF OLD SP TO Q<1:0>
6856K     0  U 1351, 0D10,0038,F5C0,F920,0084,9352  ;29787        SC_SC+K[.A]                       ;PUT -3 IN SC
                                              ;29788
```

Acknowledgements

- **These slides contain material developed and copyright by:**
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)

- **MIT material derived from course 6.823**
- **UCB material derived from course CS252**
- **KFUPM material derived from course COE501、COE502**