



中国科学技术大学
University of Science and Technology of China

计算机体系结构

周学海

xhzhou@ustc.edu.cn

0551-63606864

中国科学技术大学



review: 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

- **控制转移类指令**

- **指令编码 (指令格式)**

操作码 (OPCODE)	若干操作数 (OPRANDS)
-----------------	-----------------



第2章 ISA

2.1 ISA的基本概念

ISA中需要描述的有关问题：

如何访问操作数

需要支持哪些操作

如何控制指令执行的顺序

指令的编码问题

2.2 ISA的功能设计

2.3 ISA的实现



2.2 ISA的功能设计

功能设计

典型ISA

要回答的主要问题：

- 1、从体系结构设计者角度出发，硬件系统应该提供哪些基本操作（算子）？
- 2、介绍一些典型ISA在当时是具有特色的功能设计



ISA的功能设计

- **功能设计**

- 任务：确定硬件支持哪些操作
- 方法：统计的方法
- **两种不同的设计理念**：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：①面向目标程序的优化，②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



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

- **基本原则：满足图灵完备性**
- **一般计算机都支持前三类所有的操作；**
- **不同ISA 对软件系统支持程度不同，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。**



CISC计算机ISA的功能设计

$$\text{CPUtime} = \text{IC} \times \text{CPI} \times \text{T}$$

- **思路与目标:**

- 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 (Instruction Counts)
- 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) 增强运算型指令的功能

如 $\sin(x)$, $\cos(x)$, $\text{SQRT}(X)$, 甚至多项式计算

如用一条三地址指令完成

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

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

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。



面向高级语言和编译程序的优化 (1/3)

2. 面向高级语言和编译器改进指令系统

优化目标：主要是缩小HL-ML之间的差距

优化方法：

1) 增强面向高级语言和编译器支持的指令功能

- 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



面向高级语言和编译程序的优化 (2/3)

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) 一元赋值最多→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操作较多→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%→增强转移指令的功能, 增加转移指令的种类是必要的



面向高级语言和编译程序的优化 (3/3)

2) 高级语言计算机系统：缩小HL和ML的差别

极端：HL=ML，即所谓的高级语言计算机

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：特权指令

指令系统对OS的支持主要包括：

- 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计算机ISA的功能设计

Top 10 80x86 (CISC ISA) 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		<hr/> 96%

- Simple instructions dominate instruction frequency

简单指令占比高



RISC计算机ISA的功能设计

- **采用RISC设计理念的微处理器**

- SUN Microsystem: SPARC, SuperSPARC, Ultra SPARC
- SGI: MIPS R4000, R5000, R10000,
- IBM: Power PC
- Intel: 80860, 80960
- DEC: Alpha
- Motorola 88100
- HP HP300/930系列, 950系列
- ARM, MIPS
- **RISC-V**
- **LoongArch**



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
- 早期对RISC特点的描述
 - 采用Load/Store结构：存储器访问与运算分离
 - 大多数指令在单周期内完成
 - 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？

$$\text{CPUtime} = \text{IC} \times \text{CPI} \times T$$

	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：**
 - 确定硬件支持哪些操作
 - 设计方法是统计的方法
- **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
 - 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**
 - 手段：充分发挥流水线的效率，降低（优化）CPI



2.2 ISA的功能设计



功能设计

The diagram consists of two large, blue, chevron-shaped arrows pointing from left to right. The first arrow contains the text '功能设计' (Functional Design) in white. The second arrow contains the text '典型ISA' (Typical ISA) in red. The arrows are connected by a small gap, indicating a sequential process.

典型ISA



MIPS

- **MIPS是最典型的RISC 指令集架构**

- Stanford (1980)提出，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、顺序流水线
- 30年来，逐步增加到约**400条指令**。

- **部分指令特色：**

- 当确实需要访问不对齐数据时，采用对齐访存指令需要较复杂的地址计算、移位和拼接等操作 (Motivation)
- MIPS实现了不对齐访存指令LWL/LWR，兼顾了使用的便利性和硬件实现的简单性

0	1	2	3	4
	0x11	0x22	0x33	0x44

Big Endian

- R1
- ① LWL R1, 1
 - ② LWR R1, 4

31			0
0x11	0x22	0x33	
			0x44

0	1	2	3	4
	0x11	0x22	0x33	0x44

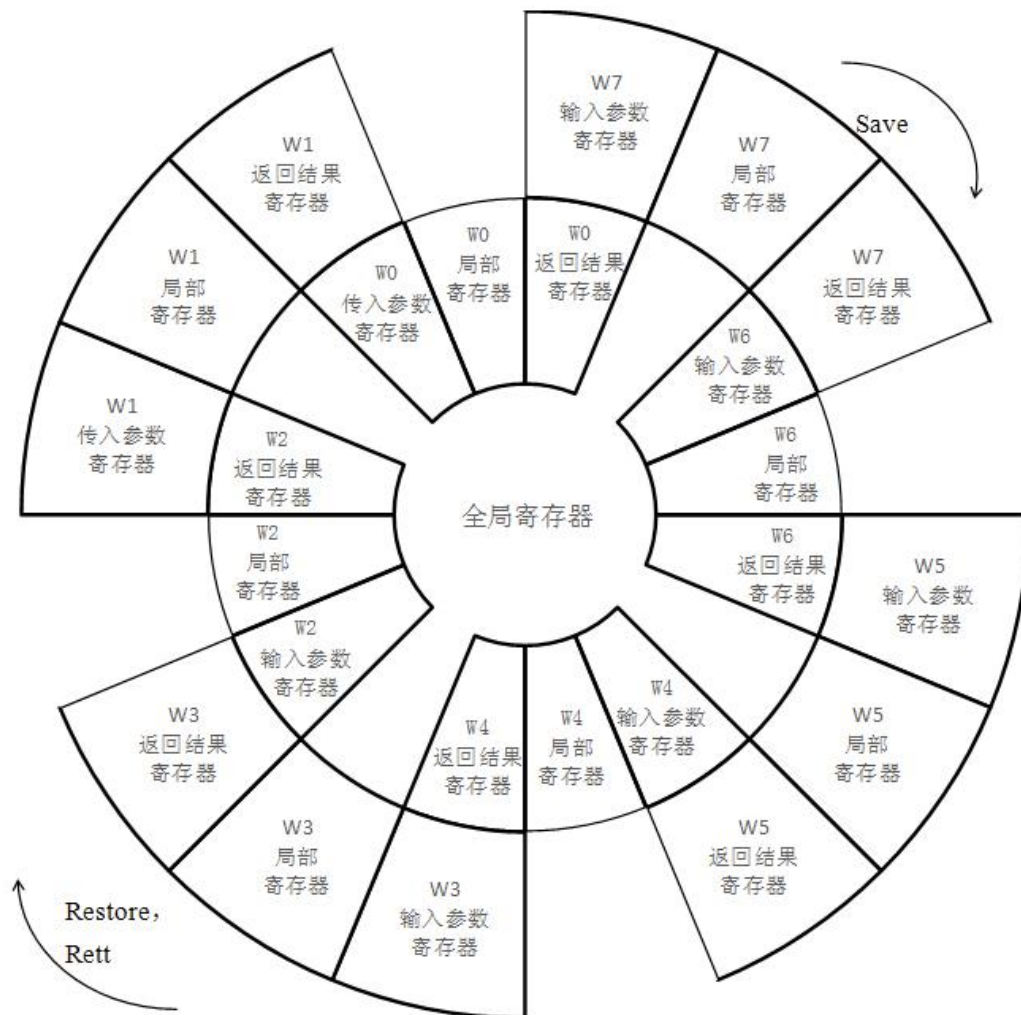
Little Endian

- R1
- ① LWR R1, 1
 - ② LWL R1, 4

31			0
	0x33	0x22	0x11
0x44			

SPARC

- **Sun Microsystems的专属指令集**
 - 可追溯到Berkeley RISC-I和RISC-II项目；例如：32位版本的ISA SPARC V8/V9
- **SPARC V8 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**
- **主要特色**
 - SPARC使用了寄存器窗口来加速函数调用和返回
 - 通过寄存器窗口重叠，传递输入参数和返回结果





PA-RISC

- **1986年，HP公司推出**
 - 第一款芯片的型号为PA-8000，后续PA-8200、PA- 8500和PA-8600等型号
- **64位微处理器PA-8700于2001年上半年正式投入服务器和工作站的使用**
- **部分指令特色：Nullification指令**
 - 可以选择不执行延时跳转后的指令，来更好地利用跳转指令的延迟槽
 - 普遍用于各种算术和逻辑指令中
 - 例如，ADDDBF(add and branch if False), 保存相加结果，如果结果为0，则可以跳过之后的一条指令。通过这样的方式，可以省略掉只有一条执行指令的分支



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
- **部分特色：**
 - Alpha架构师仔细地将特权级ISA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)
 - PAL code指令提供了一个关闭中断和虚拟地址映射的特权模式。PAL（特权指令库）可以用作TLB管理，内存原子操作，操作系统原语。通过call_pal指令调用PAL码
 - 为了高性能，不支持字节/半字的数据访问，只支持字访问
 - 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

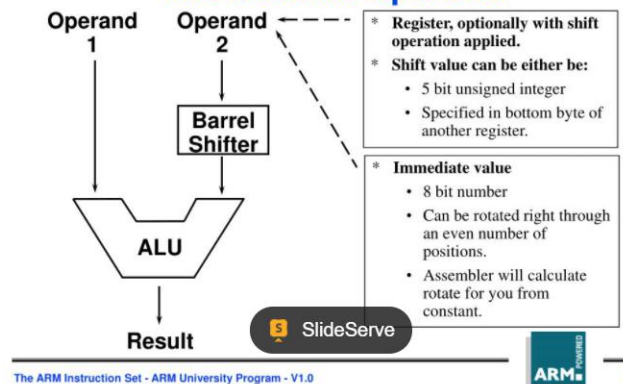
• ARM 使用最广的体系结构

- 是一个封闭的标准，不允许剪裁或扩充。即使是微架构的创新也仅限于那些能够获得ARM所称的架构许可的组织
- ARMv7, 32位 RISC ISA
 - ARMv7十分庞大复杂。整型类指令**600+条**
- 2011年，ARM发布新的ISA ARMv8
 - 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
- 2021年 3 月 30，发布ARMv9
 - 对安全的支持，V9版本引入了用于机密计算的Realms模块
 - 对AI的支持，将SVE升级到SVE2，能够显著改善CPU的AI性能

• 主要特色：

- 大部分指令可以条件式地执行，降低在分支时产生的负重
- 32-bit筒型位移器 (barrel shifter) 可用来执行大部分的算术运算指令和寻址计算而不会损失效能
- ARMv7 Thumb指令集(16位)

Using the Barrel Shifter: The Second Operand



```
MOV r0, r0, LSL #1
    //Multiply R0 by two.
MOV r1, r1, LSR #2
    //Divide R1 by four (unsigned).
MOV r2, r2, ASR #2
    //Divide R2 by four (signed).
MOV r3, r3, ROR #16
    //Swap the top and bottom halves of R3.
ADD r4, r4, r4, LSL #4
    //Multiply R4 by 17. (N = N + N * 16)
RSB r5, r5, r5, LSL #5
    //Multiply R5 by 31. (N = N * 32 - N)
```



80x86

- **Intel 8086架构是笔记本电脑、台式机和服务器的市场上最流行的指令集。**
 - 除了嵌入式系统领域，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。**
- **优势：**
 - 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指令可以完成多个基本操作。
- **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的指令长度为任意整数字节数，最多为15个字节，数量较少的短操作码已经被随意使用



2.3 ISA的实现

微程序
控制器

RISC-V
简介

RISC-V
简单实现



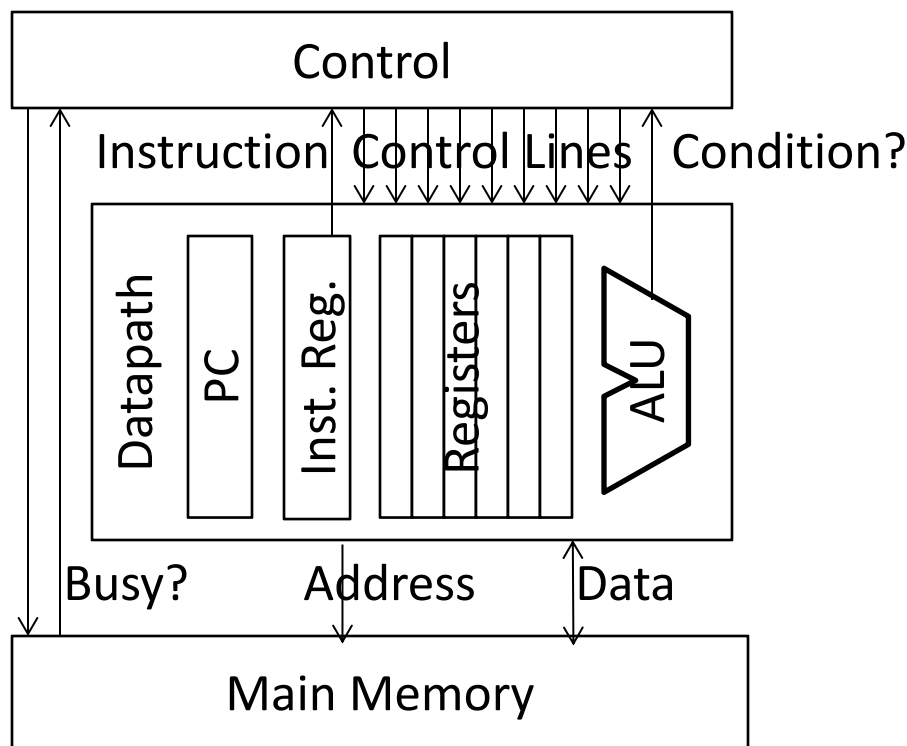
Recap: 为什么学习微程序控制

- 如何用一个结构简单的处理器实现复杂ISA
- CISC机器怎么发展起来的
 - CISC指令集仍然广泛使用 (x86)
- 帮助我们理解从CISC- \rightarrow RISC的技术驱动力



控制部分与数据通路

- 处理器设计可以分为datapath和控制设计两部分
 - **datapath**, 存储数据、算术逻辑运算单元、内部处理器总线
 - **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: 电子管
 - Main Memory: 磁芯存储器
 - Read-Only Memory: 二极管阵列, 穿孔金属卡片, ...
 - Cost: Logic > RAM > ROM
 - Speed: ROM > RAM

Data path 以及 Control Unit

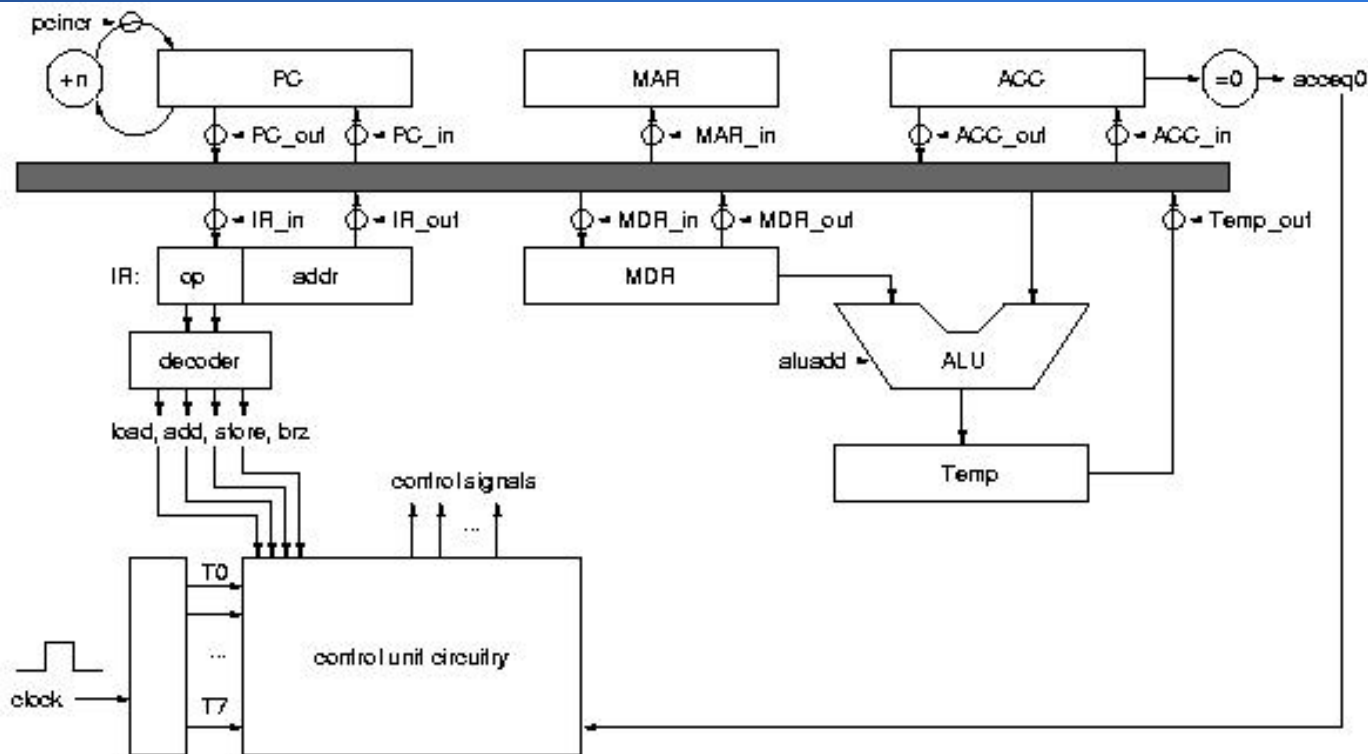


Figure 1. Simple data path for a four-instruction computer (the small circles represent control points)

- Data path、Control Unit
- Register Transfer
- Control Signal , Control Point (logic gate)
- Macroinstruction, Microinstruction, Microoperation
- 组合逻辑控制器和微程序控制器

(opcode 00) load address : $ACC \leftarrow \text{memory}[\text{address}]$
 (opcode 01) add address : $ACC \leftarrow ACC + \text{memory}[\text{address}]$
 (opcode 10) store address : $\text{memory}[\text{address}] \leftarrow ACC$
 (opcode 11) brz address : if($ACC == 0$) $PC \leftarrow \text{address}$



A Simple Example

(opcode 00) load address : $ACC \leftarrow \text{memory}[\text{address}]$
(opcode 01) add address : $ACC \leftarrow ACC + \text{memory}[\text{address}]$
(opcode 10) store address : $\text{memory}[\text{address}] \leftarrow ACC$
(opcode 11) brz address : if($ACC == 0$) $PC \leftarrow \text{address}$

Figure 2: Instruction definitions for the simple computer

ACC_in : $ACC \leftarrow \text{CPU internal bus}$
ACC_out : $\text{CPU internal bus} \leftarrow ACC$
aluadd : addition is selected as the ALU operation
IR_in : $IR \leftarrow \text{CPU internal bus}$
IR_out : $\text{CPU internal bus} \leftarrow \text{address portion of IR}$
MAR_in : $MAR \leftarrow \text{CPU internal bus}$
MDR_in : $MDR \leftarrow \text{CPU internal bus}$
MDR_out : $\text{CPU internal bus} \leftarrow MDR$
PC_in : $PC \leftarrow \text{CPU internal bus}$
PC_out : $\text{CPU internal bus} \leftarrow PC$
pcincr : $PC \leftarrow PC + 1$
read : $MDR \leftarrow \text{memory}[\text{MAR}]$
TEMP_out : $\text{CPU internal bus} \leftarrow TEMP$
write : $\text{memory}[\text{MAR}] \leftarrow MDR$

Figure 3: Control signal definitions for the simple datapath

time steps T0-T3 for each instruction **fetch**:

T0: PC_out, MAR_in

T1: read, pcincr

T2: MDR_out, IR_in

T3: time step (if needed) for decoding the opcode in the IR

time steps T4-T6 for the **load** instruction:

T4: IR_out(addr part), MAR_in

T5: read

T6: MDR_out, ACC_in, reset to T0

time steps T4-T7 for the **add** instruction:

T4: IR_out(addr part), MAR_in

T5: read

T6: ACC_out, aluadd

T7: TEMP_out, ACC_in, reset to T0

time steps T4-T6 for the **store** instruction:

T4: IR_out(addr part), MAR_in

T5: ACC_out, MDR_in

T6: write, reset to T0

time steps T4-T5 for the **brz** (branch on zero) instruction:

T4: if (acc_{eq}0) then { IR_out(addr part), PC_in }

T5: reset to T0

Figure 4. Control sequences for the four instructions



微程序控制器

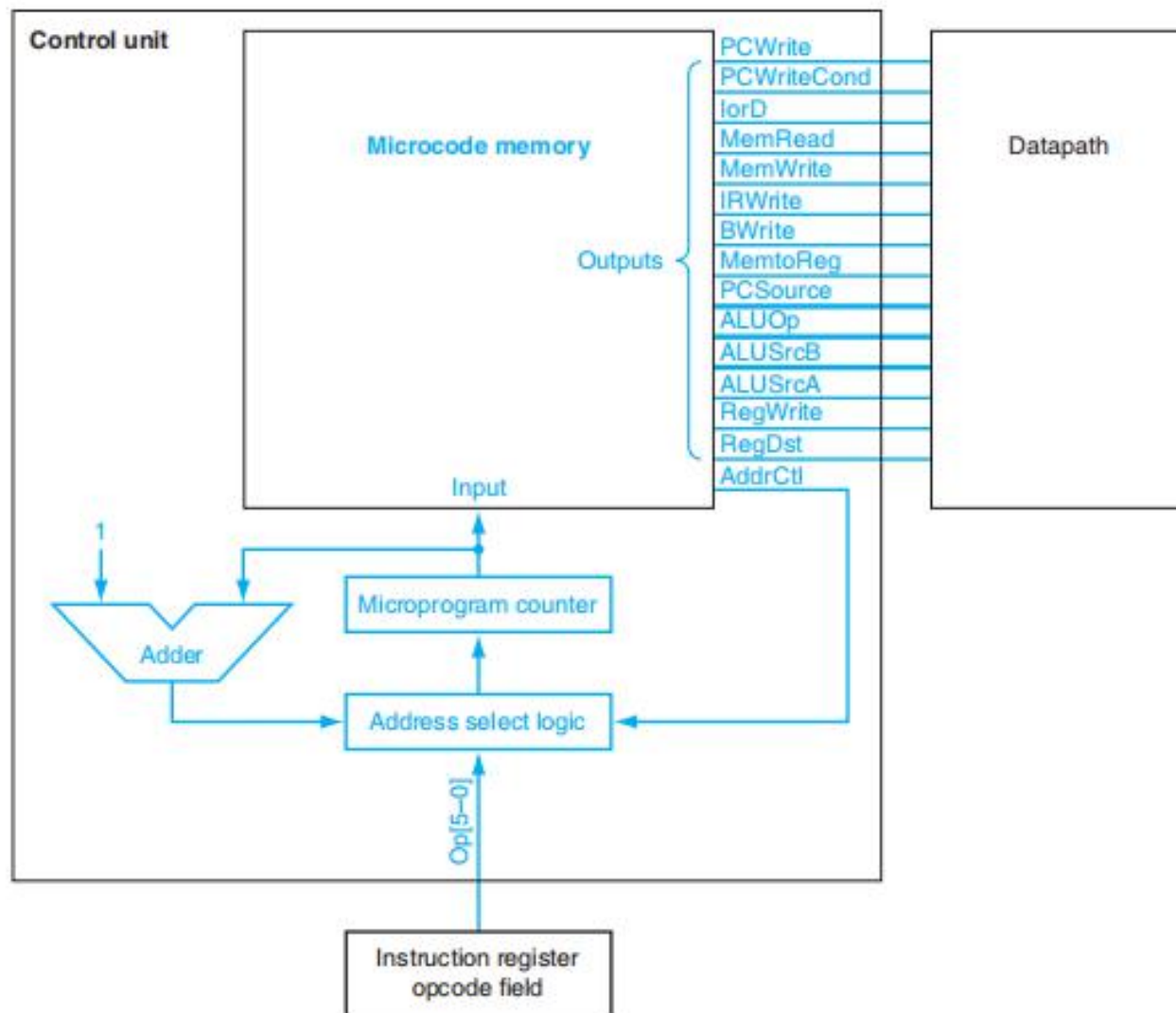


FIGURE C.4.6 The control unit as a microcode. The use of the word “micro” serves to distinguish between the program counter in the datapath and the microprogram counter, and between the microcode memory and the instruction memory.



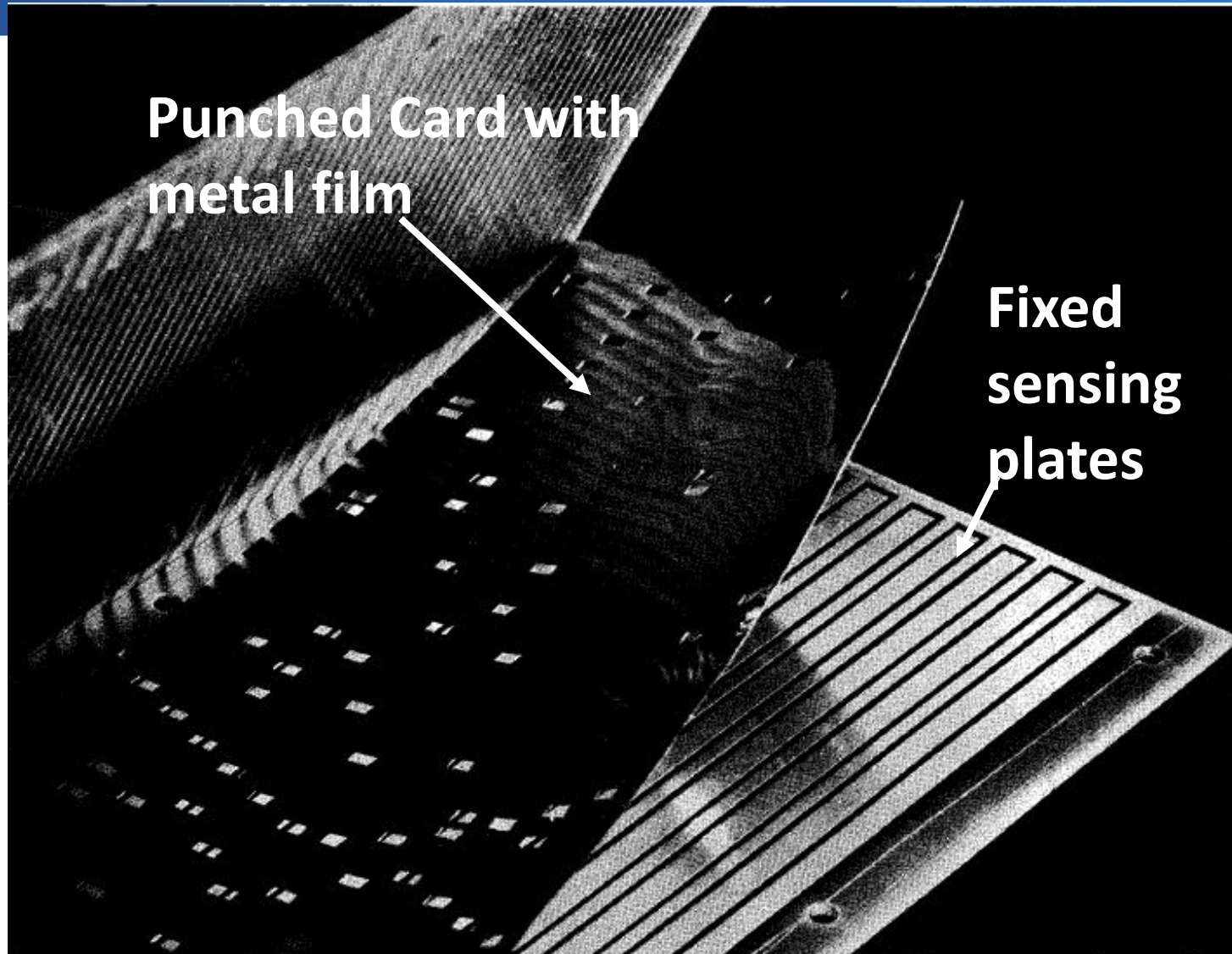
Microprogramming in IBM 360

	M30	M40	M50	M65
Datapath width (bits)	8	16	32	64
μ inst width (bits)	50	52	85	87
μ code size (K μ insts)	4	4	2.75	2.75
μ store technology	CCROS	TCROS	BCROS	BCROS
μ store 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



[IBM Journal, January 1961]



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）
 - 微程序存放在ROM中实现扩展的复杂指令
- **对于复杂的指令集，采用微程序设计技术使得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



VAX 11-780 Microcode

; PIWFUD, [600,1205]
; CALL2 ,MIC [600,1205]

MICRO2 1F(12)
Procedure call

26-May-81 14:58:11
; CALLG, CALLS

VAX11/780 Microcode : PCS 01, FPLA 0D, WCS122

Page 771

```
;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
;29748 CALL,J/MPUSH ;PUSH REGISTERS
;29749
;29750 ;-----;RETURN FROM MPUSH
;29751 CACHE_D[LONG], ;PUSH PC
;29752 LAB_R[SP] ; BY SP
;29753
;29754 ;-----;
;29755 CALL.8: R[SP]&VA_LA-K[,8] ;UPDATE SP FOR PUSH OF PC &
;29756
;29757 ;-----;
;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
;29764 CALL,J/PSHSP
;29765
;29766 ;-----;
;29767 D_R[AP], ;READY TO PUSH AP
;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,2,V,C>
;29773 LAB_R[SP] ;GET SP INTO LATCHES AGAIN
;29774
;29775 ;-----;
;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
;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_PC[T4], ;GET LOW BITS OF OLD SP TO Q<1:0>
;29787 SC_SC+K[,A] ;PUT -3 IN SC
;29788
```



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 μ code需要子程序和调用堆栈
 - 需要修复控制程序中的bug与 μ ROM 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



80年代初：微程序控制机器分析

- **用高级语言编程成为主流：关键问题：编译器会生成什么指令？**
- **IBM的John Cocke团队**
 - 为小型计算机801 (ECL Server) 开发了更简单的 ISA 和编译器，移植到 IBM370，仅使用简单的寄存器-寄存器及load/store指令，发现：与原IBM 370相比，性能提高3X
- **80年代初，Emer和Clark (DEC) 发现**
 - VAX 11/180 CPI = 10!，虽然声称是1MIPS的机器，实际测试其性能是0.5MIPS
 - VAX ISA 的 20%指令（占用了60%的微码）仅占用了 0.2%的执行时间
- **VAX8800**
 - 控制存储: 16K*147b RAM, Unified Cache: 64K*8b RAM, 微程序控制存储是cache容量的4.5x
- **结论：**
 - 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（pipelining, caches and buffers）,使得多周期执行reg-reg指令失去了吸引力



From CISC to RISC

- **使用快速RAM构建用户最近要执行的指令的指令缓存，而不是固定的硬件微程序**
 - 指令缓存中的内容随着程序执行不断更新，提高访存速度
 - 使用简单的ISA，以有效实现硬布线的流水线方式执行
 - 大多数编译器生成的代码只使用了一部分常用的CISC指令
 - 简单的指令格式使得流水线高效实现成为可能
- **芯片集成度的提高带来的机遇**
 - 80年代初，单芯片上可以在32位的数据通路加上小的cache
 - 大多数情况下没有芯片间的通信，使得性能更好



Microprogramming is far from extinct

- **80年代微程序控制起到了关键作用**
 - DEC uVAX, Motorola 68K series, Intel 286/386
- **现代微处理器中微程序控制扮演辅助的角色**
 - e.g., AMD Bulldozer, Intel Ivy Bridge, Intel Atom, IBM PowerPC, ...
 - 大多数指令采用硬布线逻辑控制
 - 不常用的指令或者复杂的指令采用微程序控制
- **芯片bug的修复（打补丁） 例如Intel处理器在bootup阶段可装载微代码方式的patches**
 - 英特尔不得不重新启用微代码工具，并寻找原来的微代码工程师来修补**熔毁/幽灵安全漏洞**



2.3 ISA的实现

微程序
控制器

RISC-V
简介

RISC-V
简单实现



Recap: 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：
 - 微程序或硬布线控制；顺序或乱序执行流水线；单发射或超标量等等。
 - 支持定制化，成为定制专用加速器的基础。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。避免被弃用，如过去的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的基础部分足够简单、完整**，可以用于教学和嵌入式处理器，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
- **方便根据应用需求扩展ISA（指令集扩展）**
 - 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**
- **有效的指令编码方式**

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

Subset	Name
Standard General-Purpose ISA	
Integer	I
Integer Multiplication and Division	M
Atomics	A
Single-Precision Floating-Point	F
Double-Precision Floating-Point	D
General	G = IMAFD
Standard User-Level Extensions	
Quad-Precision Floating-Point	Q
Decimal Floating-Point	L
16-bit Compressed Instructions	C
Bit Manipulation	B
Dynamic Languages	J
Transactional Memory	T
Packed-SIMD Extensions	P
Vector Extensions	V
User-Level Interrupts	N
Non-Standard User-Level Extensions	
Non-standard extension “abc”	Xabc
Standard Supervisor-Level ISA	
Supervisor extension “def”	Sdef
Non-Standard Supervisor-Level Extensions	
Supervisor extension “ghi”	SXghi

- **RISC-V ISA定义:**
一个基本的整型类ISA + ISA的扩展(可选)
- **基本的整型类ISA:**
 - RV32I or RV64I
 - RV32的变体 RV32E
 - RV128I
- **基本的整型类ISA不可以修改**
- **扩展的ISA分为: 标准扩展和非标准扩展**
 - 标准扩展通常是通用的, 不应当与其他标准扩展有冲突
 - 非标准扩展可能是非常专用的, 可能与其他标准或非标准扩展冲突。



RISC-V的指令编码

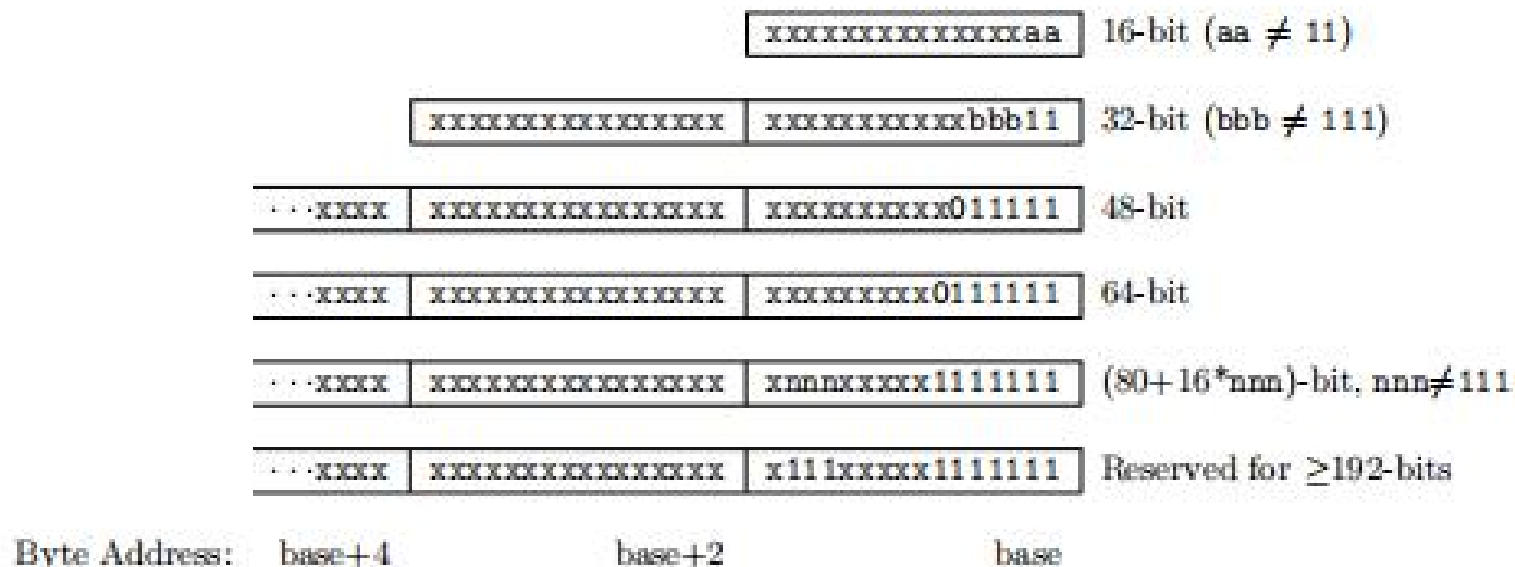


Figure 1.1: RISC-V instruction length encoding.

- 基本的32位整型类ISA指令长度32位，并且指令地址必须字对齐
- 扩展的ISA支持变长指令格式，每条指令长度可以是2字节的倍数，指令地址按2字节对齐



RISC-V 基本整型ISA编程模型

- Program counter (**pc**)
- 32个整型数寄存器 (**x0-x31**)
 - **x0 总是 0**
- 寄存器的位数: XLEN
- 操作数寻址方式:
 - 立即数寻址
 - 偏移寻址

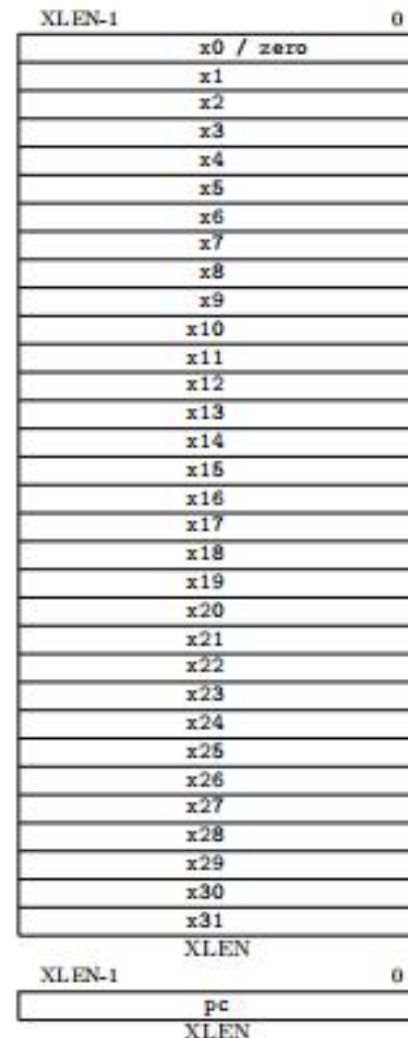
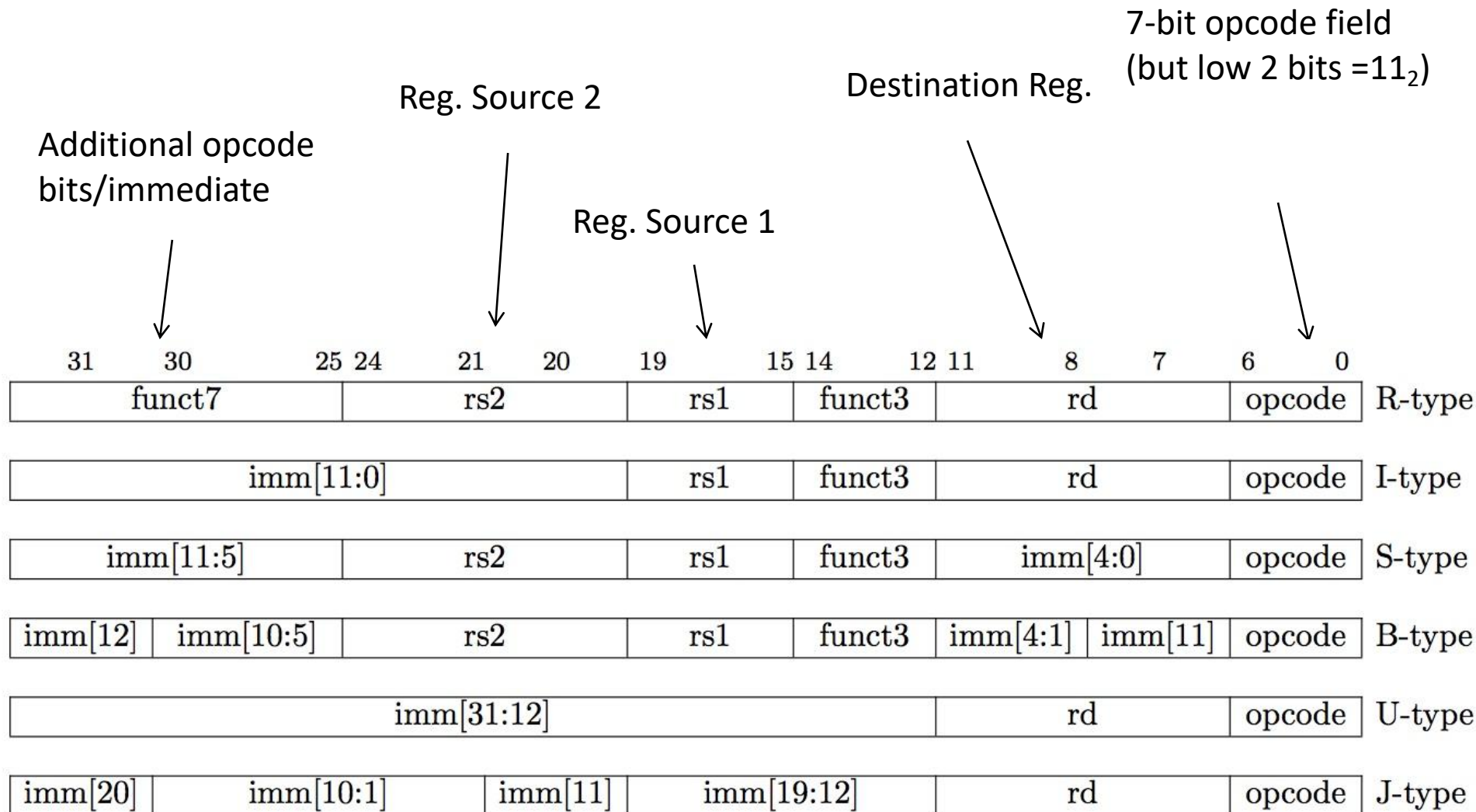


Figure 2.1: RISC-V user-level base integer register state.



RISC-V 32I基本指令格式





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



RISC-V ISA 小结

- 模块化的指令集
- 规整的指令编码
- 可定制的扩展
- 优雅的压缩指令子集
- 方便硬件设计与编译器实现
 - 简化的分支跳转，不使用分支延迟槽，不使用指令条件码
 - 存储器访问指令一次只访问一个元素
 - **立即数的最高位总是在指令的最高位**
 - 较多的寄存器，专门的Load/Store指令
 - 整型类的ALU指令在一个时钟周期完成，有利于预测指令串的执行时间
 - **ISA 支持位置无关代码** (Position Independent Code)



2.3 ISA的实现

RISC-V
简介

微程序
控制器

RISC-V
简单实现

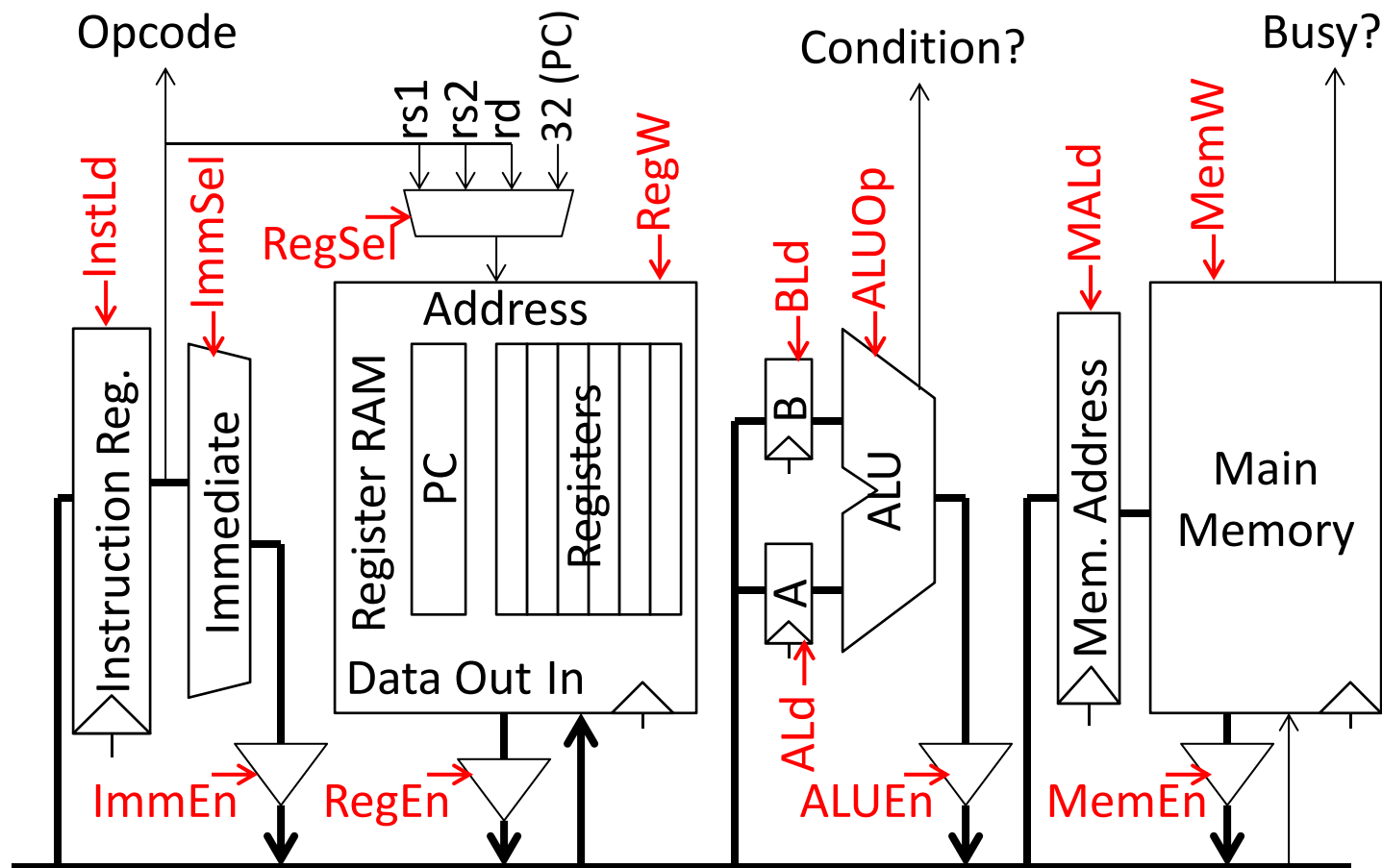


RISC-V 指令执行阶段

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



微程序控制RISC-V的单总线数据通路

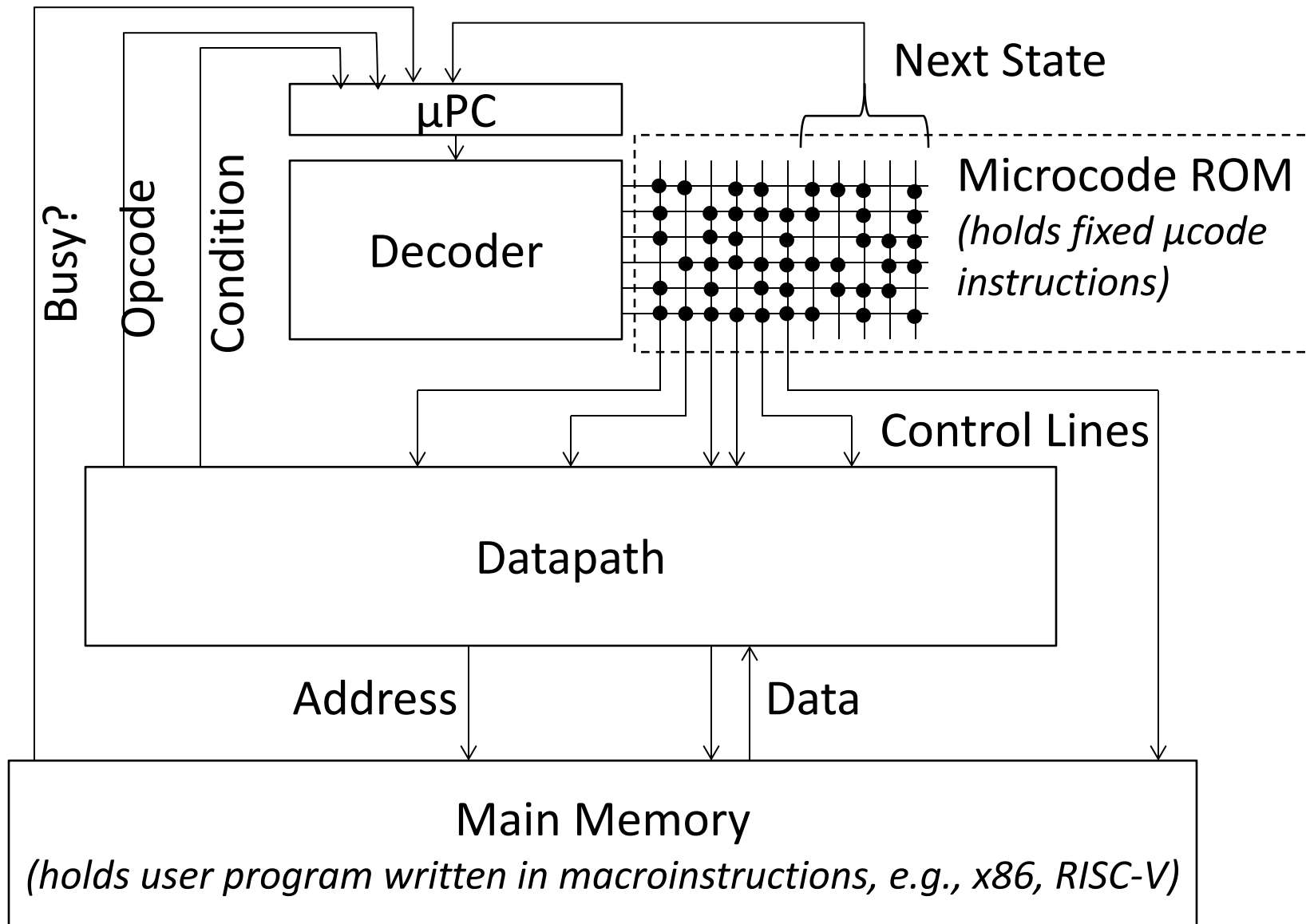


微指令的寄存器传输级(RTL)表示:

- **MA:=PC** means RegSel=PC; RegW=0; RegEn=1; MALd=1
- **B:=Reg[rs2]** means RegSel=rs2; RegW=0; RegEn=1; BLd=1
- **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:=Imm1 //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:

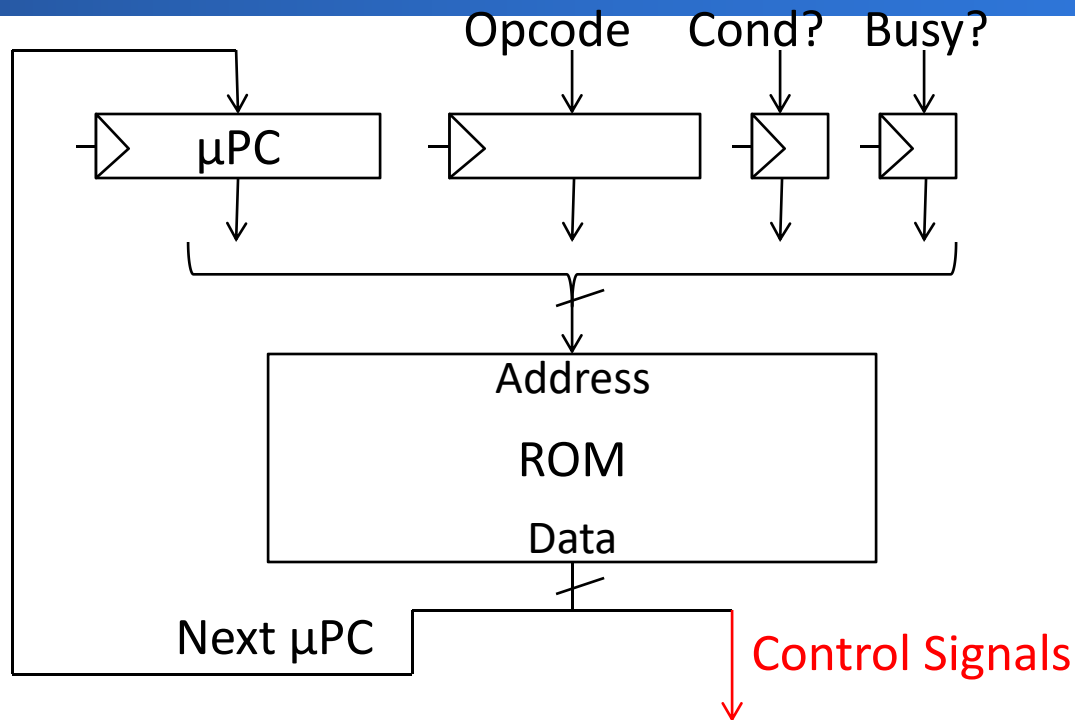
Reg[rd]:=A // Store return address
A:=A-4 // Recover original PC
B:=ImmJ // Jump-style immediate
PC:=A+B
goto instruction fetch

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?**

$$|\mu\text{address}| = |\mu\text{PC}| + |\text{opcode}| + 1 + 1$$

- **How many data bits?**

$$|\text{data}| = |\mu\text{PC}| + |\text{control signals}| = |\mu\text{PC}| + 18$$

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



ROM 中的内容

Address				Data	
μ PC	Opcode	Cond?	Busy?	Control Lines	Next μ PC
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

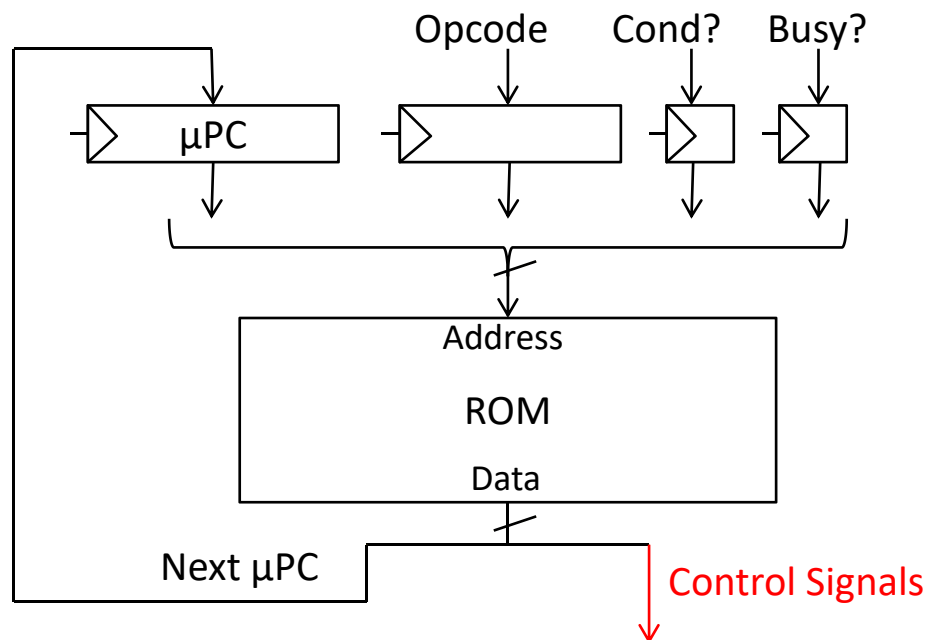


单总线数据通路结构的微程序控制存储器大小

- 取指阶段有3个公操作
- RISC-V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 μ PC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 = $2^{(6+5+2)} \times (6+18) = 2^{13} \times 24 = \sim 25\text{KiB!}$



单总线 RISC-V 微程序控制引擎



$$|\mu\text{address}| = |\mu\text{PC}| + |\text{opcode}| + 1 + 1$$

$$|\text{data}| = |\mu\text{PC}| + |\text{control signals}|$$

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

Reducing Control Store Size



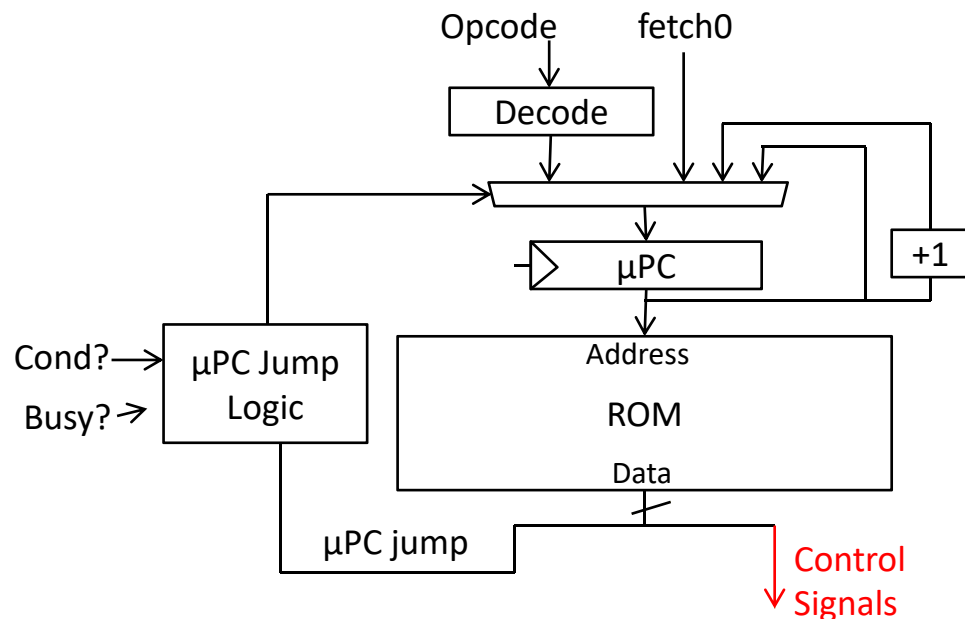
Reduce ROM height (#address bits)

使用外部逻辑来组合#input
通过分组操作码减少#state

Reduce ROM width (#data bits)

μPC 编码

控制信号编码(vertical μcoding, nanocoding)



$$\mu\text{PC jump} = \text{next} \mid \text{spin} \mid \text{fetch} \mid \text{dispatch} \mid \text{ftrue} \mid \text{ffalse}$$



μ PC Jump 类型

- ***next*** : increments μ PC
- ***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	
μ PC	Control Lines	Next μ PC
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	false
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: $M[rd] = M[rs1] + M[rs2]$

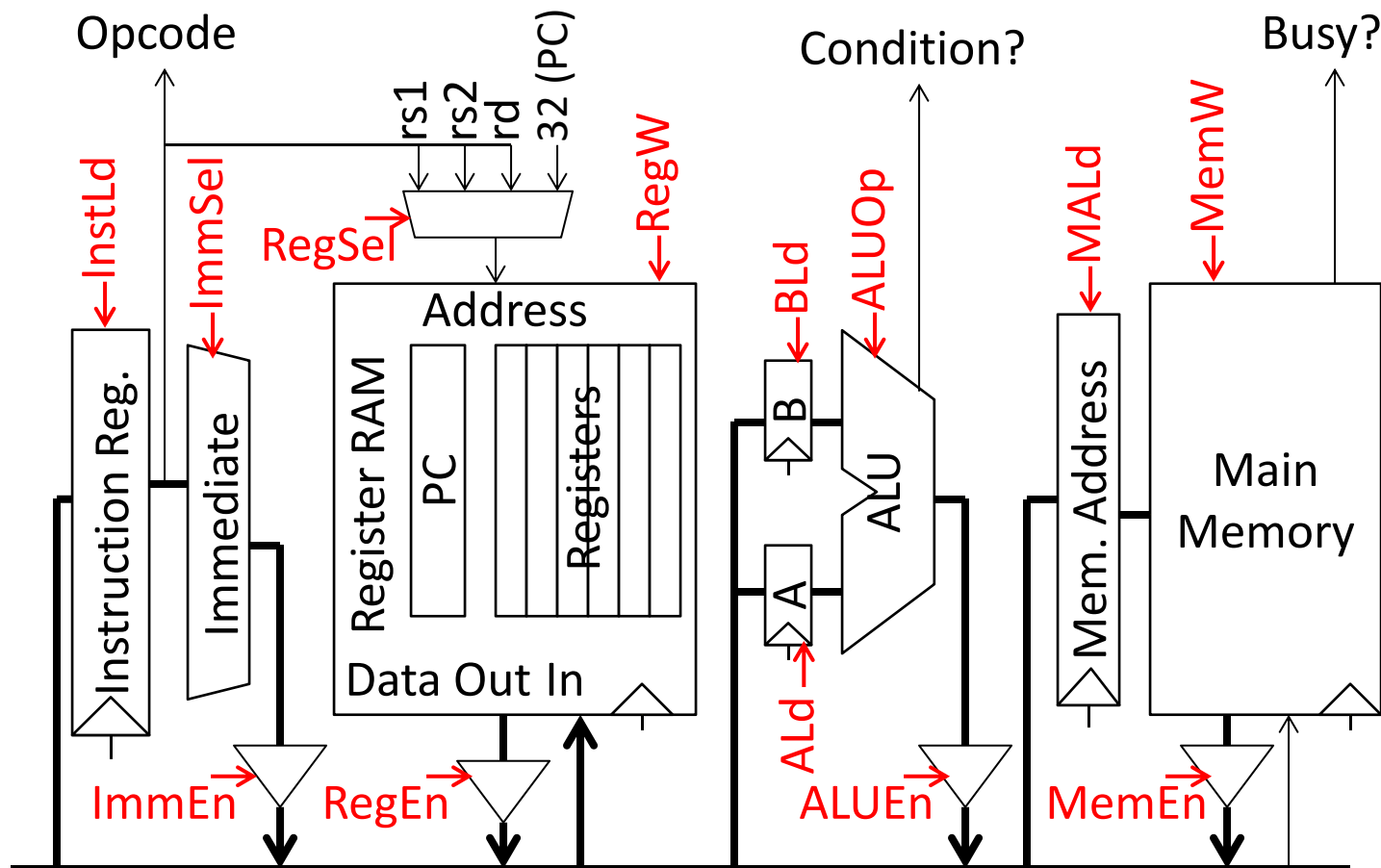
Address	Data	
μPC	Control Lines	Next μPC
MMA0	MA:=Reg[rs1]	next
MMA1	A:=Mem	spin
MMA2	MA:=Reg[rs2]	next
MMA3	B:=Mem	spin
MMA4	MA:=Reg[rd]	next
MMA5	Mem:=ALUOp(A,B)	spin
MMA6		fetch

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比较困难（特别在早期无EDA工具支持的情况下）



Single-Bus Datapath for Microcoded RISC-V



Datapath unchanged for complex instructions!



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**