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

- **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两种设计理念**
  - 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
- **典型ISA**
  - 技术方面：基于目前硬件技术现状，存在一些已经过时的决定
  - 非技术方面：知识产权问题

review：典型ISA

	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	√	√	√		√		

影响ISA在商业上的命运：技术原因固然重要，但非技术原因同样重要。
Intel、ARM的成功经验给我们的重要启示：生态建设十分重要！

2.1 ISA的基本概念

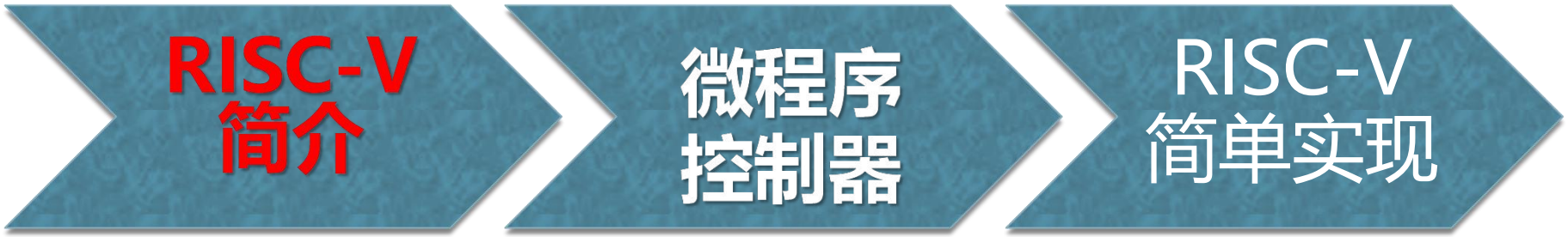
ISA中需要描述的有关问题：

- 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

2.3 ISA的实现



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：**

  一个基本的整数ISA + ISA的扩展（可选）
- **基本的整数ISA：**
  - RV32I or RV64I
  - RV32的变体 RV32E
  - RV128I
- **基本的整数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

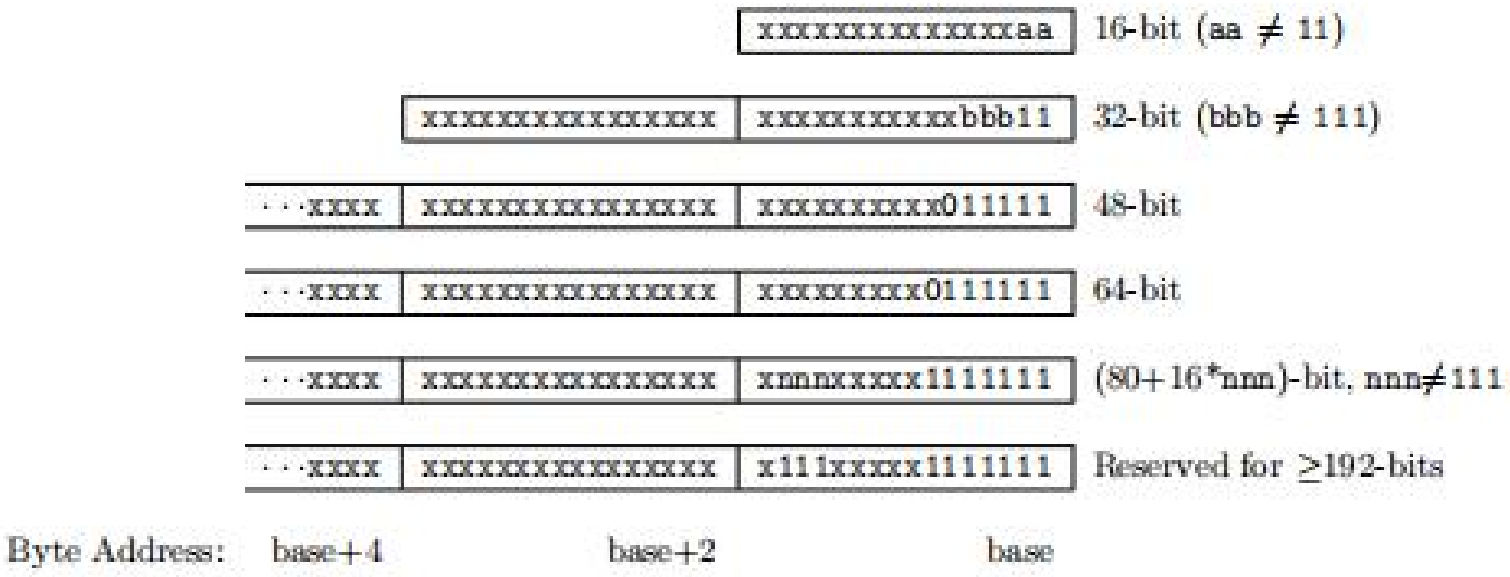


Figure 1.1: RISC-V instruction length encoding.

- **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

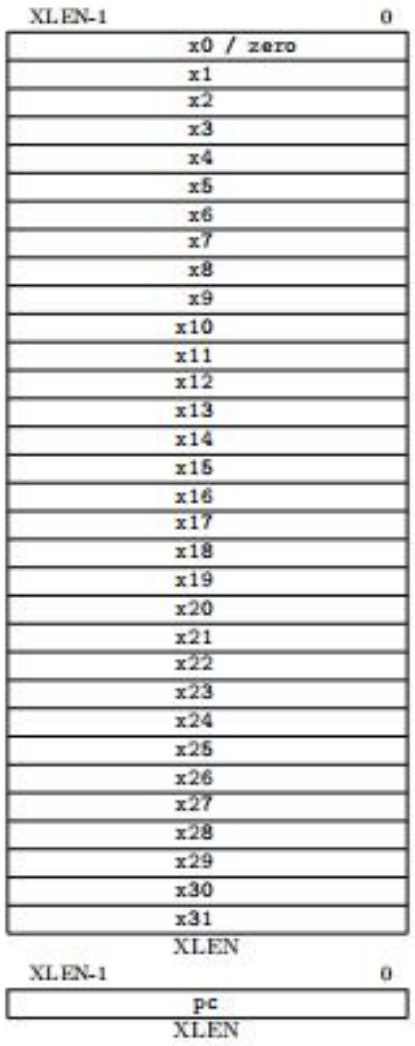
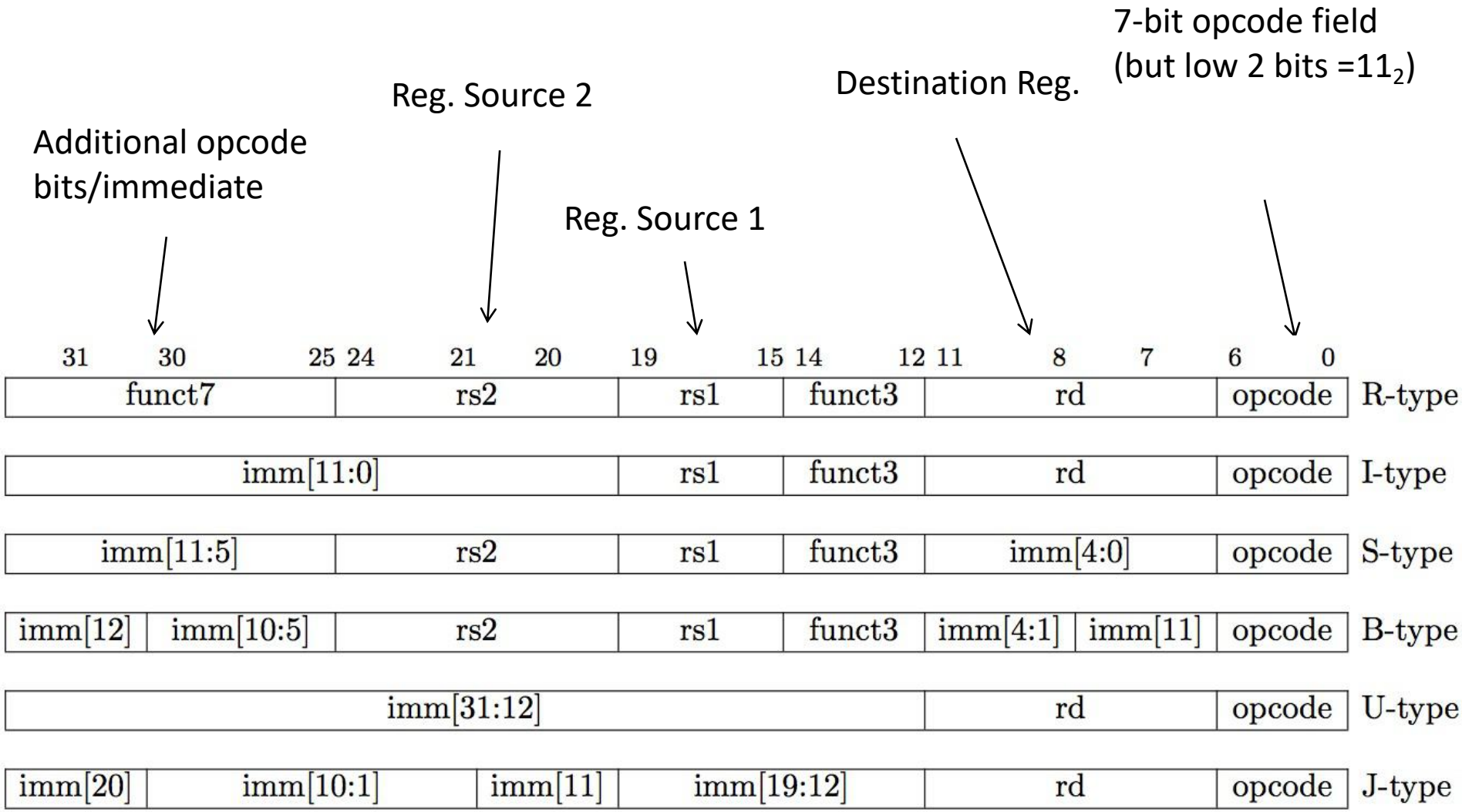


Figure 2.1: RISC-V user-level base integer register state.

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

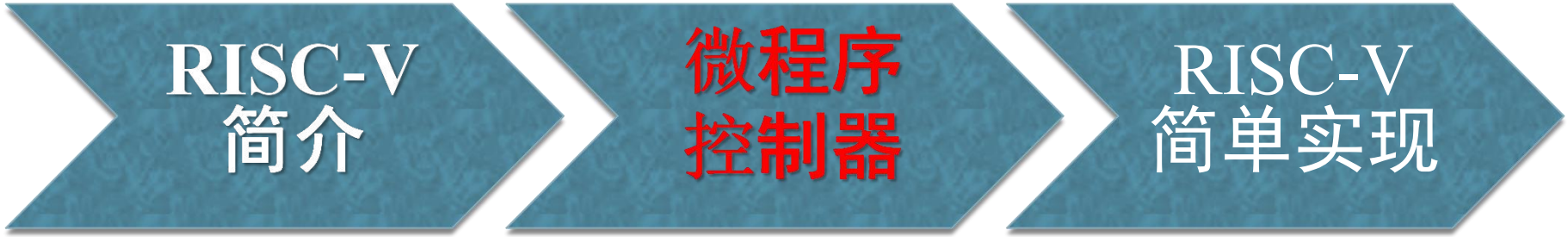
RISC-V ISA 小结

- **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

2.3 ISA的实现

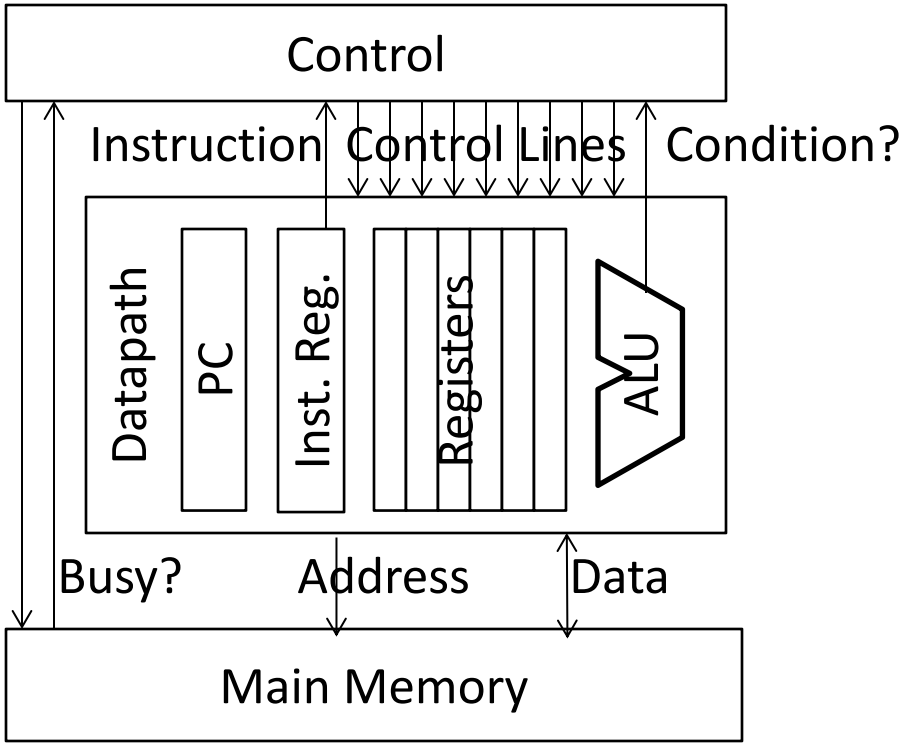


Recap：为什么学习微程序控制

- **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, IBM360, PowerPC)
- **帮助我们理解技术驱动从CISC->RISC**

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、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: Vacuum Tubes
  - Main Memory: Magnetic cores
  - Read-Only Memory: Diode matrix, punched metal cards, …
  - Cost： Logic > RAM > ROM
  - Speed: ROM > RAM

Data path 以及Control Unit

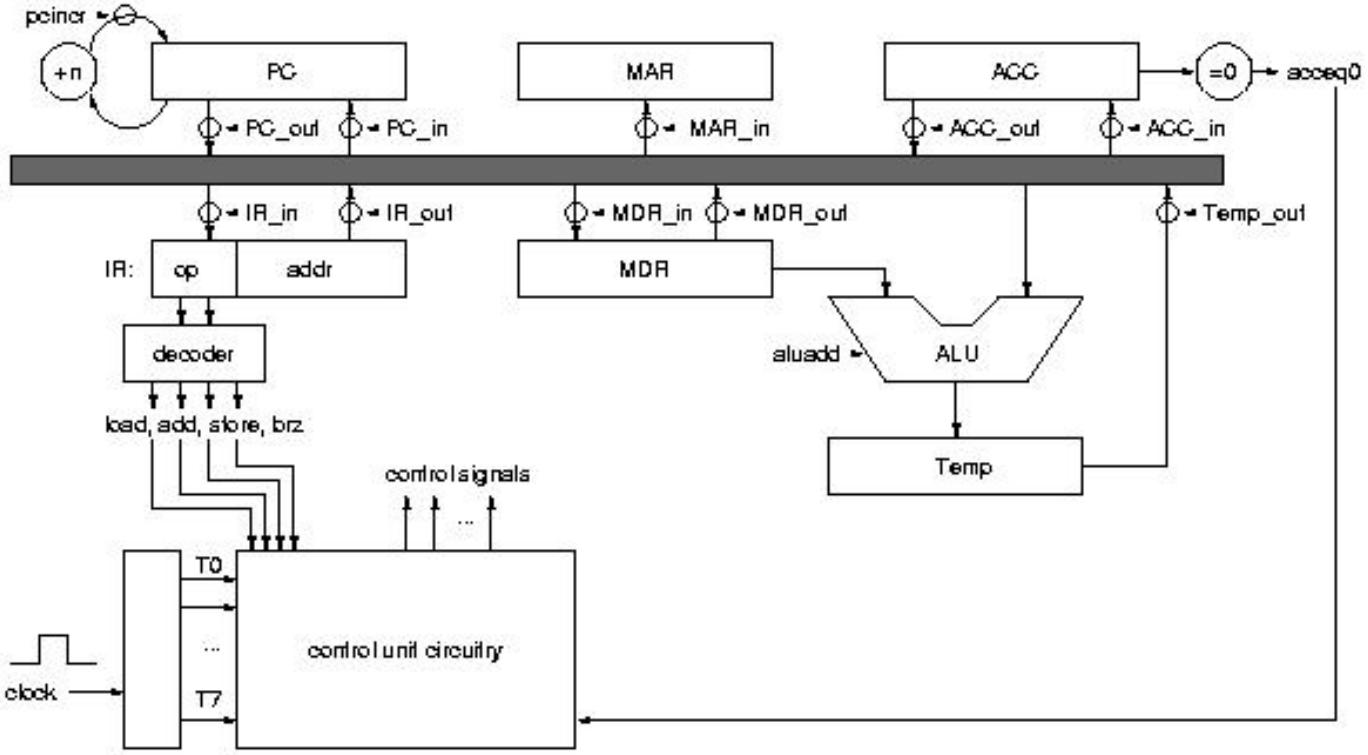


Figure 1. Simple data path for a four-instruction computer (the small circles represent control points)

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

(opcode 00) load address : ACC <- memory[address]
(opcode 01) add address : ACC <- ACC + memory[address]
(opcode 10) store address : memory[address] <- ACC
(opcode 11) brz address : if(ACC == 0) PC <- address

A Simple Example

```
(opcode 00)  load  address :  ACC <- memory[ address ]
(opcode 01)  add   address :  ACC <- ACC + memory[ address ]
(opcode 10)  store address :  memory[ address ] <- ACC
(opcode 11)  brz   address :  if( ACC == 0 ) PC <- address
```

Figure 2：Instruction definitions for the simple computer

```
ACC_in     : ACC <- CPU internal bus
ACC_out    : CPU internal bus <- ACC
aluadd     : addition is selected as the ALU operation
IR_in      : IR <- CPU internal bus
IR_out     : CPU internal bus <- address portion of IR
MAR_in     : MAR <- CPU internal bus
MDR_in     : MDR <- CPU internal bus
MDR_out    : CPU internal bus <- MDR
PC_in      : PC <- CPU internal bus
PC_out     : CPU internal bus <- PC
pcincr     : PC <- PC + 1
read       : MDR <- memory[ MAR ]
TEMP_out   : CPU internal bus <- TEMP
write      : memory[ MAR ] <- 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 (acceq0) 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

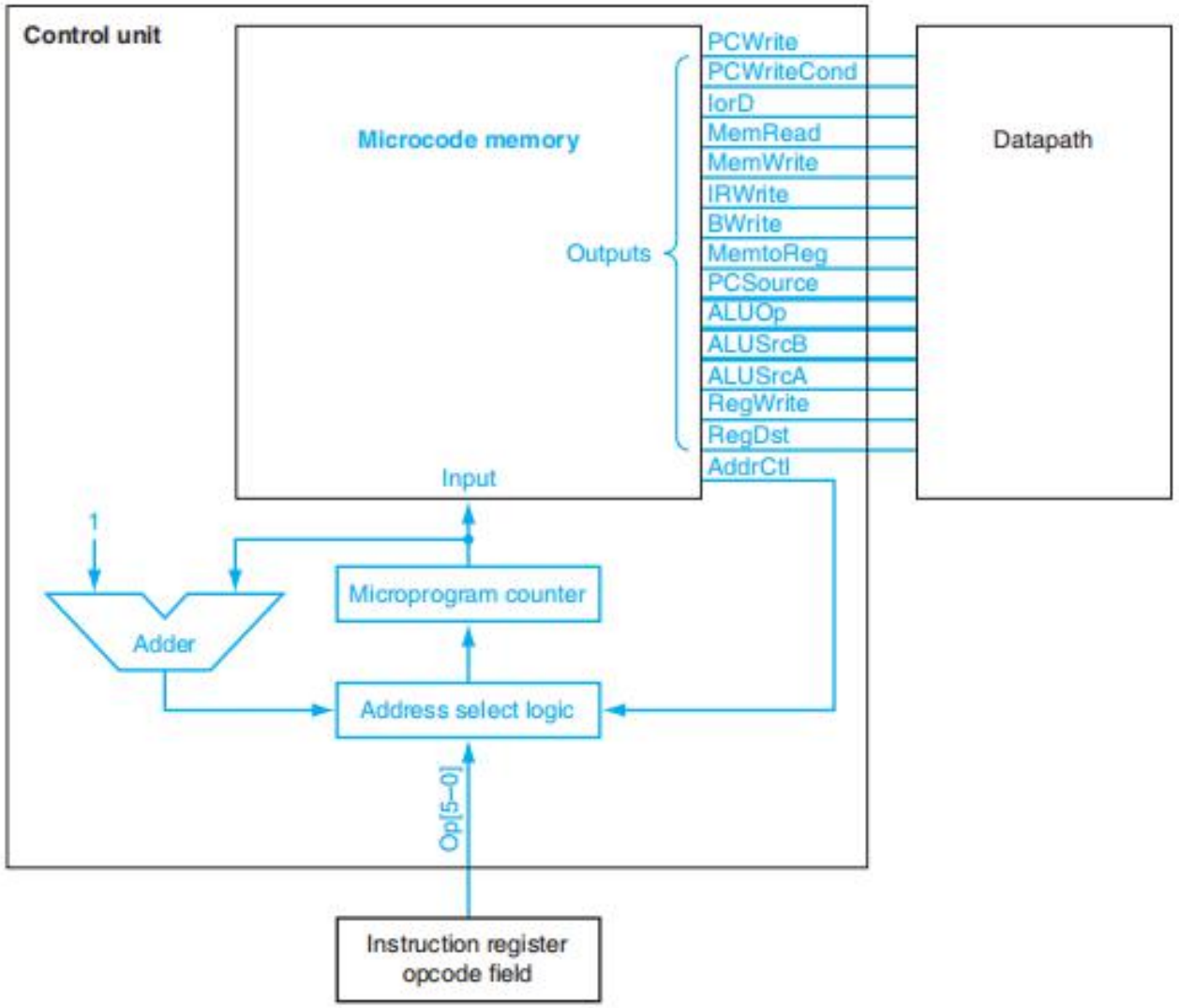


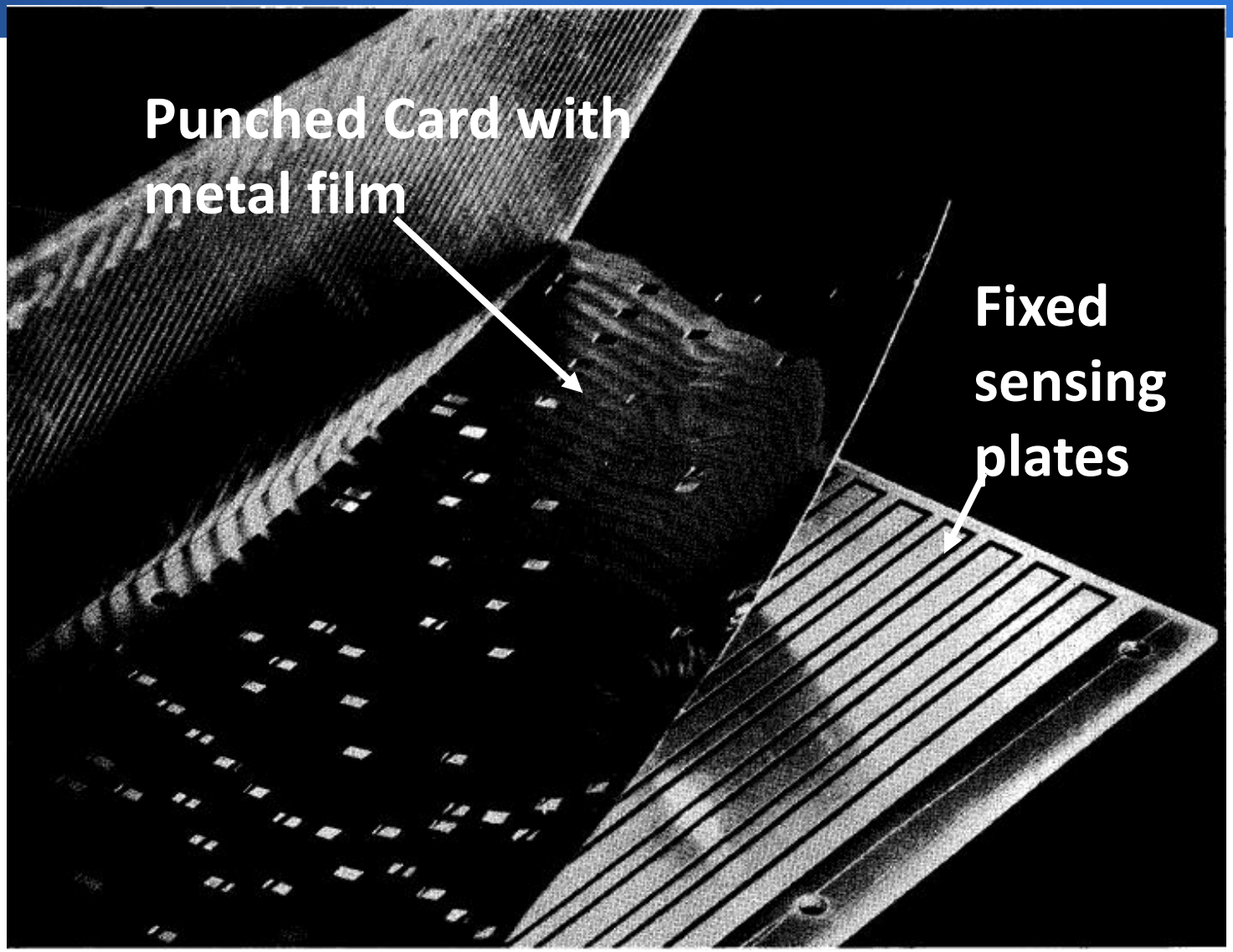
FIGURE C.4.6 The control unit as a microcode. The use of the word "micro" serves to distinguish between the program counter in the datapath and the microprogram counter, and between the microcode memory and the instruction memory.

Microprogramming in IBM 360

	M30	M40	M50	M65
Datapath width (bits)	8	16	32	64
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



[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

```
; P1WFUD.[600,1205]     MICRO2  1F(12)    26-May-81  14:58:1     VAX11/780 Microcode : PCS 01, FPLA 0D, WCS122     Page  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
```

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**

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IBM 370的简单的寄存器-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

From CISC to RISC

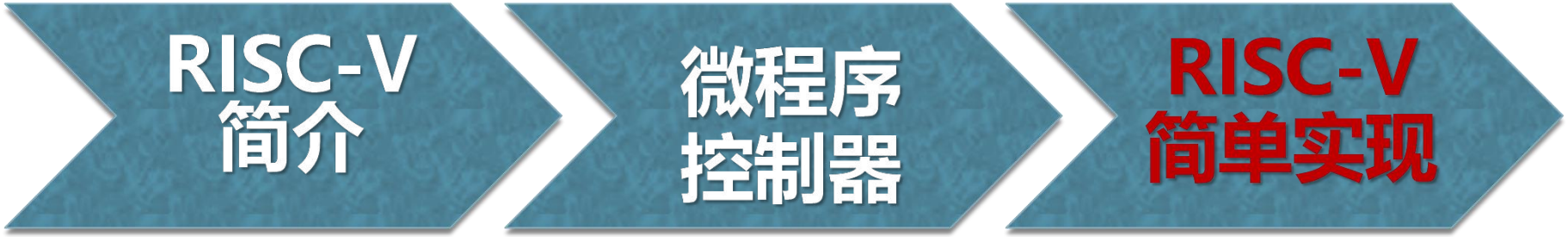
- 使用快速RAM构建用户最近要执行的指令的指令缓存，而不是固定的硬件微程序
  - 指令缓存中的内容随着程序执行不断更新，提高访存速度
  - 使用简单的ISA，以有效实现硬布线的流水线方式执行
  - 大多数编译器生成的代码只使用了一部分常用的CISC指令
  - 简单的指令格式使得流水线高效实现成为可能
- **芯片集成度的提高带来的机遇**
  - 80年代初，单芯片上已经可以集成32-bit的数据通路加上小的cache
  - 大多数情况下没有芯片间的通信，使得性能更好

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

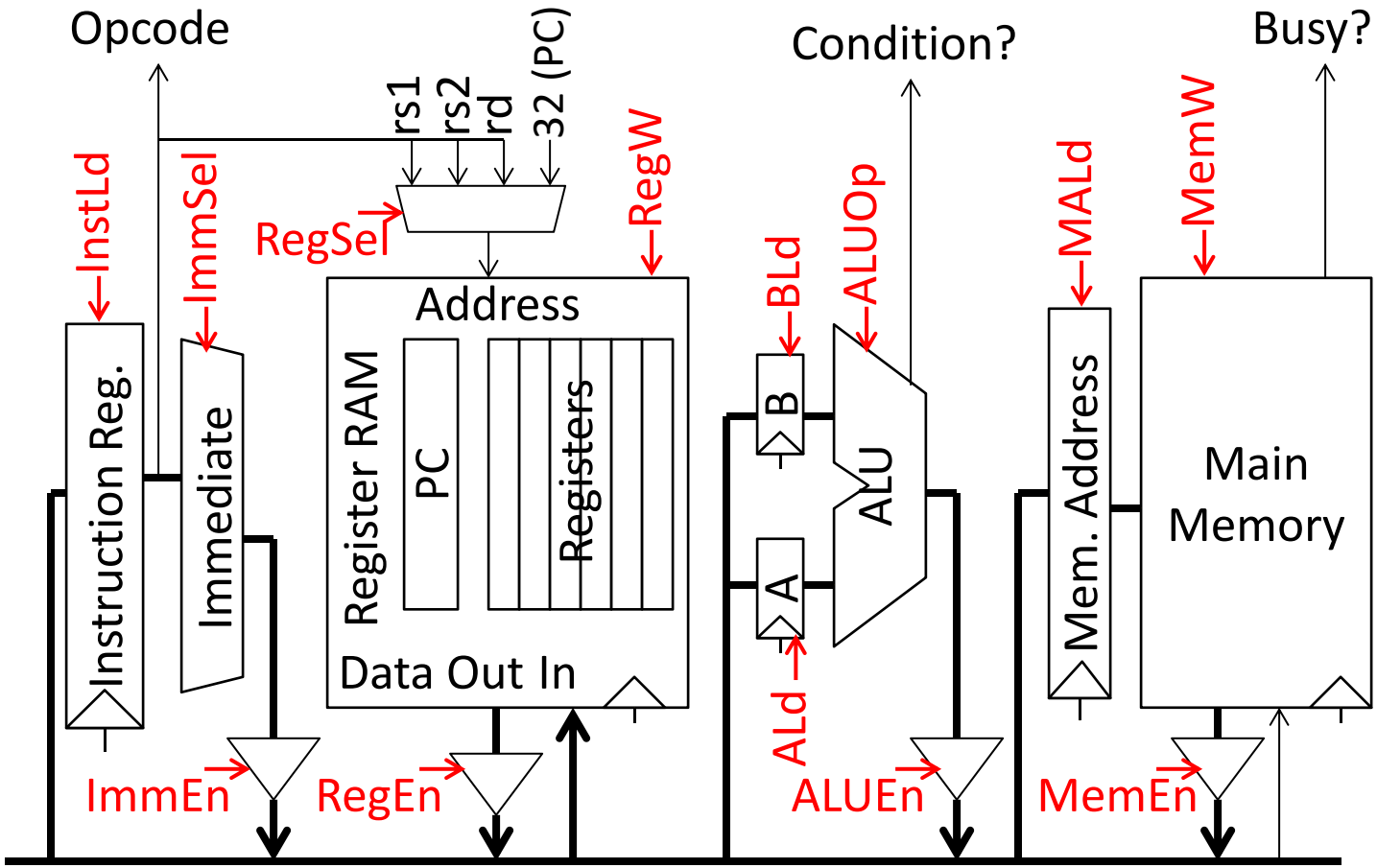
2.3 ISA的实现



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**

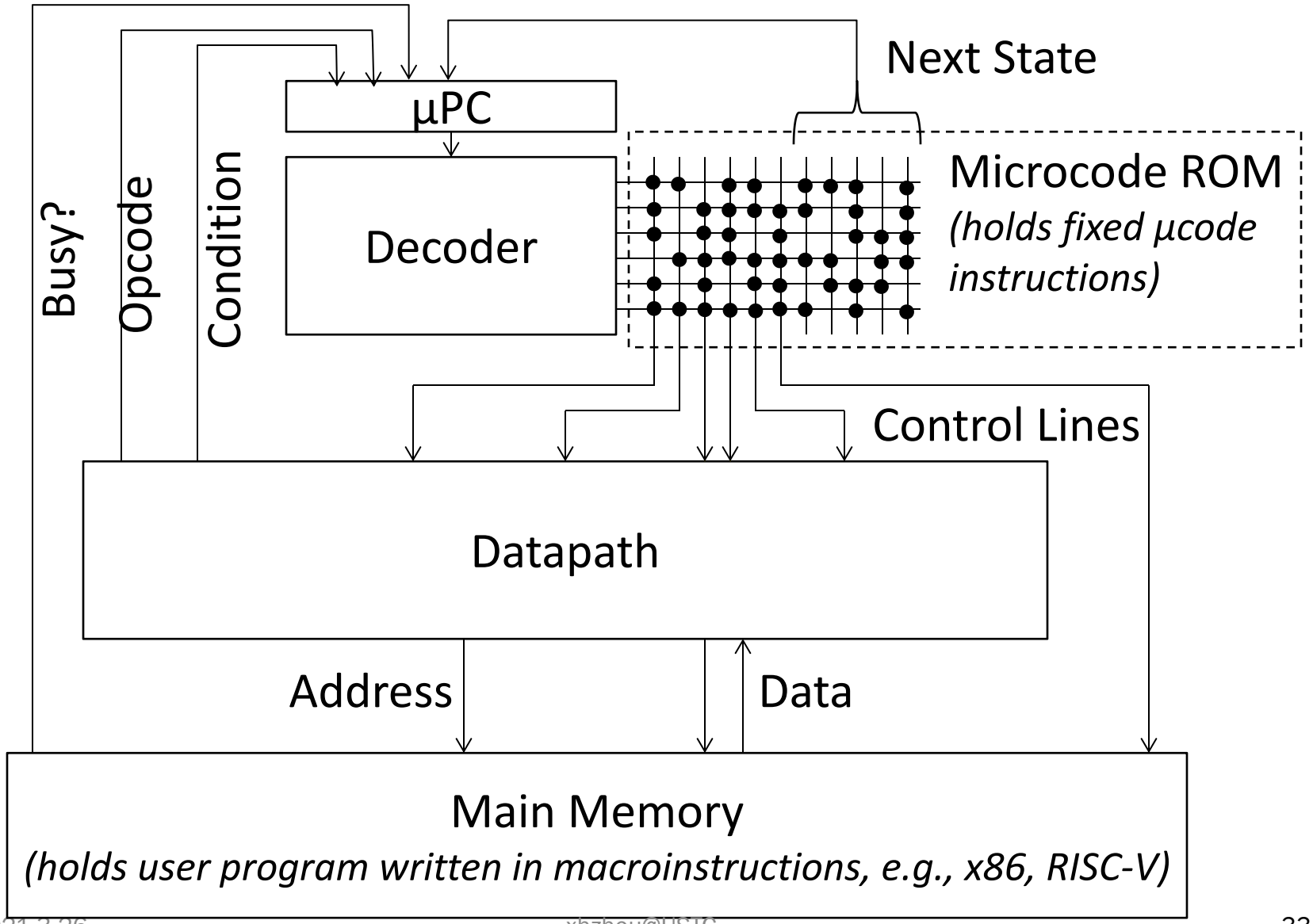
微程序控制RISC-V的单总线数据通路



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


Next State
μPC
Busy?
Opcode
Condition
Decoder
Microcode ROM
(holds fixed μcode instructions)
Control Lines
Datapath
Address
Data
Main Memory
(holds user program written in macroinstructions, e.g., x86, RISC-V)

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:=Imml //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

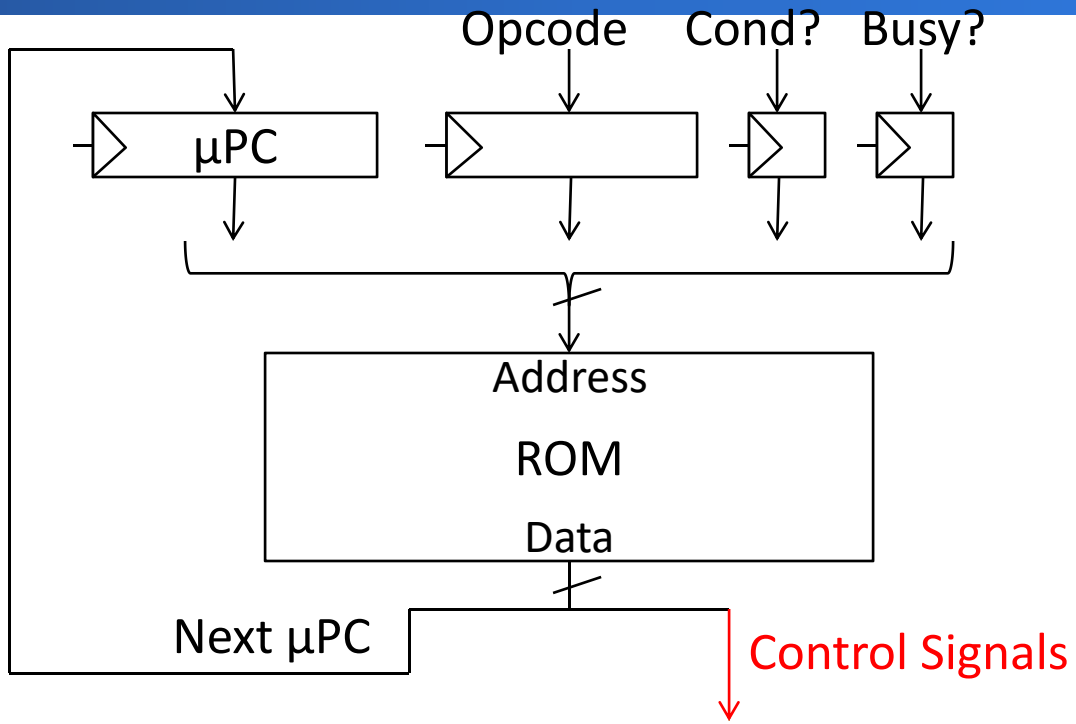
JAL:

A:=PC // 20200305
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 = $2^{|\mu address|} \times |data|$**

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

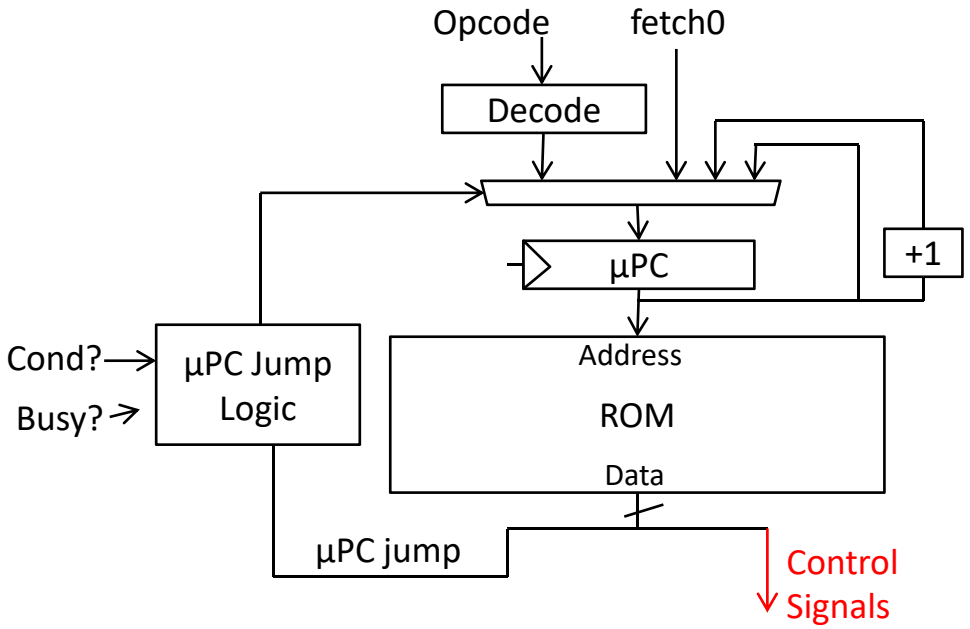
单总线数据通路结构的微程序控制存储器大小

- 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 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!

单总线 RISC-V 微程序控制引擎



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

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

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

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

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)

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

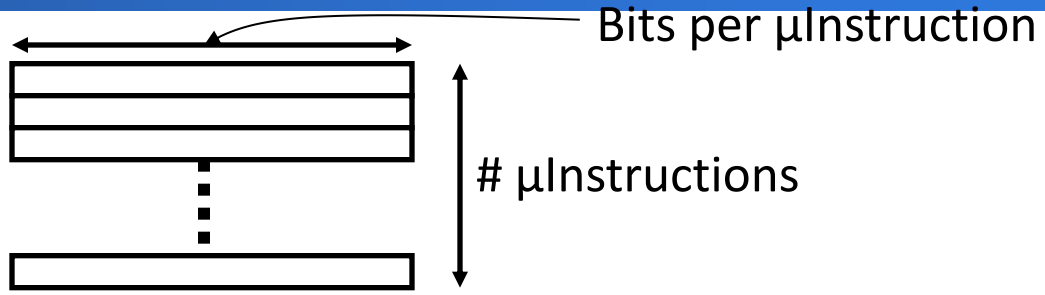
Memory-memory add: M[rd] = M[rs1] + M[rs2]

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

Horizontal vs Vertical μCode



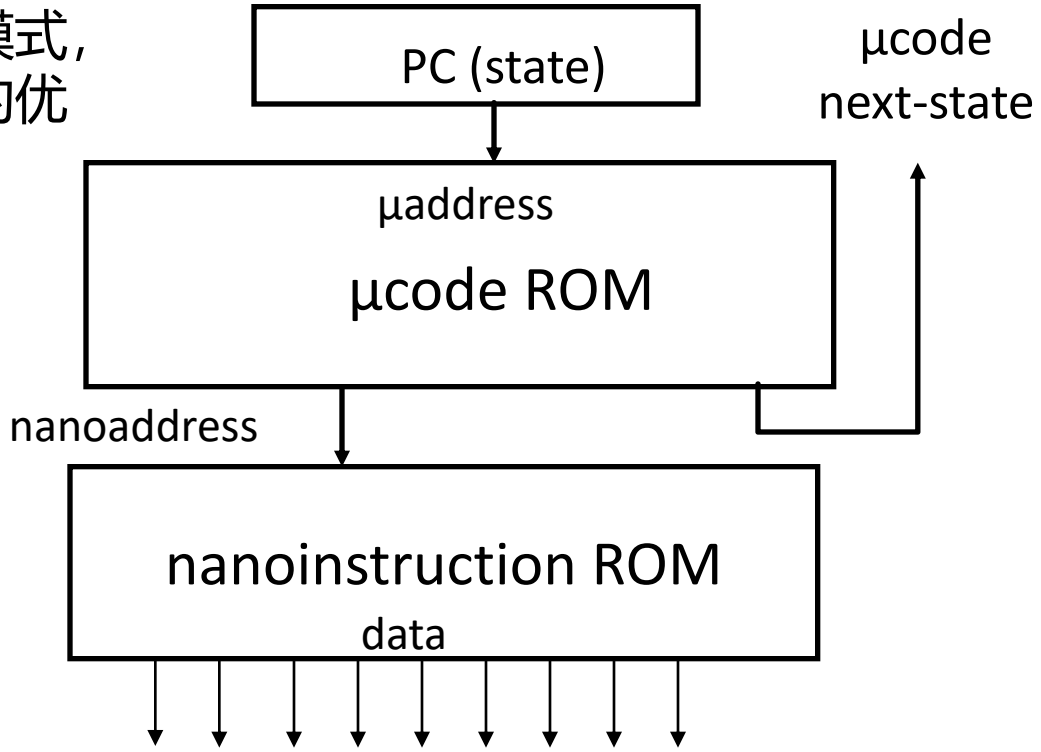
- **水平微编码的水平型微指令**
  - 每条微指令有多个微操作并行
  - 每条宏指令（指令）具有较少的微指令
  - 稀疏的（微操作）编码 ⇒ 微指令较宽（含有较多的位数）
- **垂直微编码的垂直型微指令**
  - 典型的是每条微指令代表一个数据通路操作
    - 不同的数据通路分支是独立的微指令
  - 每条宏指令（指令）需要更多的微指令
  - 紧凑的（微指令）编码 ⇒ 微指令较窄（含有较少的位数）
- **Nanocoding**
  - 水平型微指令和垂直型微指令的结合

Reduce ROM width (#data bits)

Nanocoding

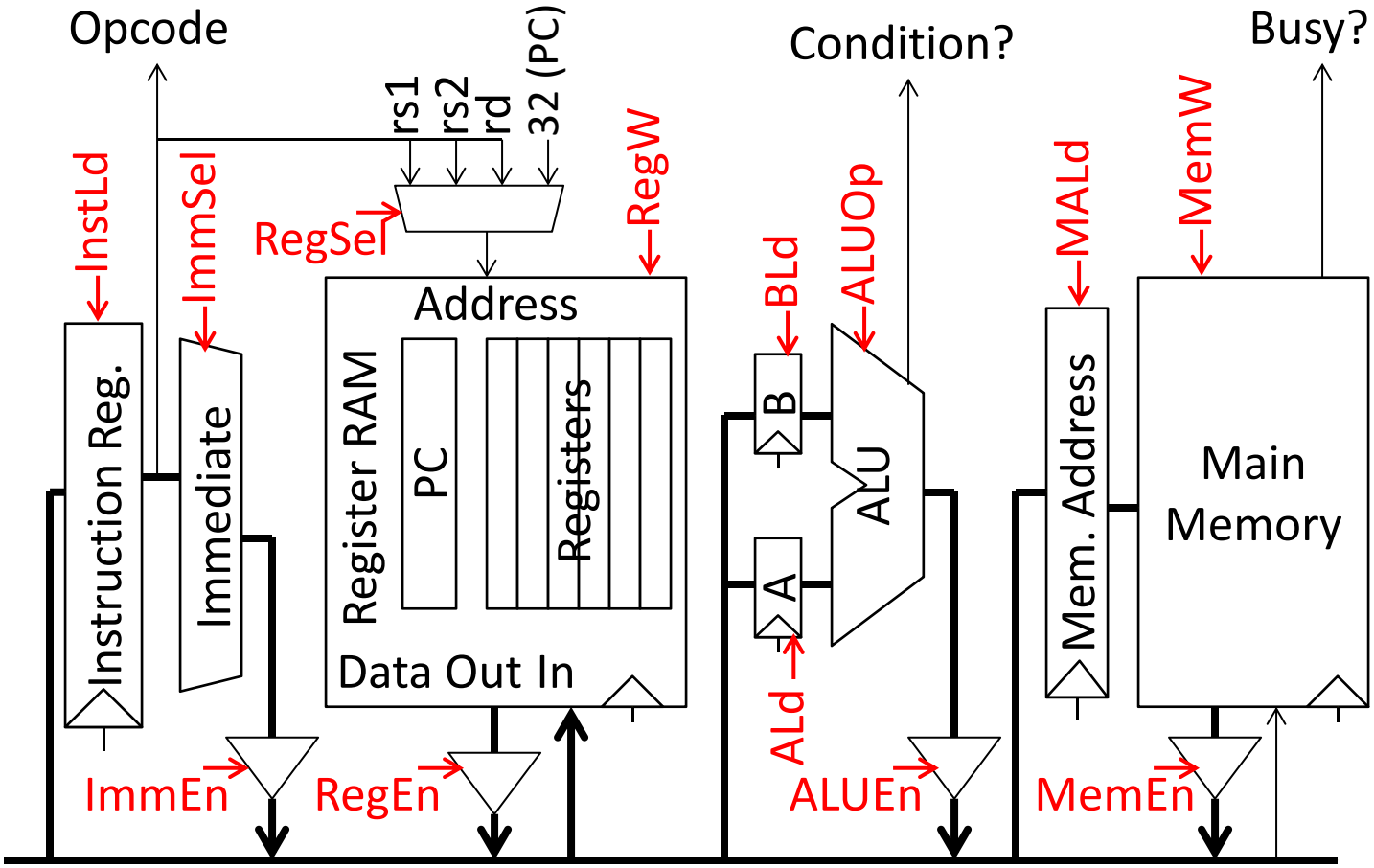
利用微代码中重复的控制信号模式，结合垂直微指令和水平微指令的优势。

```
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

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**