计算机组成与体系结构

第5章 指令系统

赵庆行、张骏鹏

人工智能学院

本次课重点

- CISC
- RISC

指令系统结构的发展

5.5 指令系统结构的发展

- 在已进入计算机软硬件协同设计的今天，计算机的每一项设计决定都应该综合考虑以下因素：
  - 技术支持
  - 计算机体系结构
  - 编程语言
  - 编译技术
  - 操作系统
- 同时所做出的设计决定也会对这些因素产生影响。
- 指令系统结构发展中的标志性事件都与计算机体系结构的进步密不可分。

5.5 指令系统结构的发展

- 指令系统是计算机软、硬件的交界面，无论是计算机硬件技术的发展还是计算机软件技术的发展，都必然会引起指令系统的演变和发展。
- 随着计算机软、硬件的发展，指令系统经历了从简单到复杂（CISC），又由复杂到简单（RISC）的演变过程，而且还会继续不断地演变和发展下去。
- RISC与CISC是目前指令系统发展的两支主流，采用两种不同的设计思路。CISC设计更关注功能，RISC设计更关注速度；CISC指令系统崇尚大而全的设计理念，RISC指令系统崇尚少而精、精而快的设计理念。

5.5 指令系统结构的发展

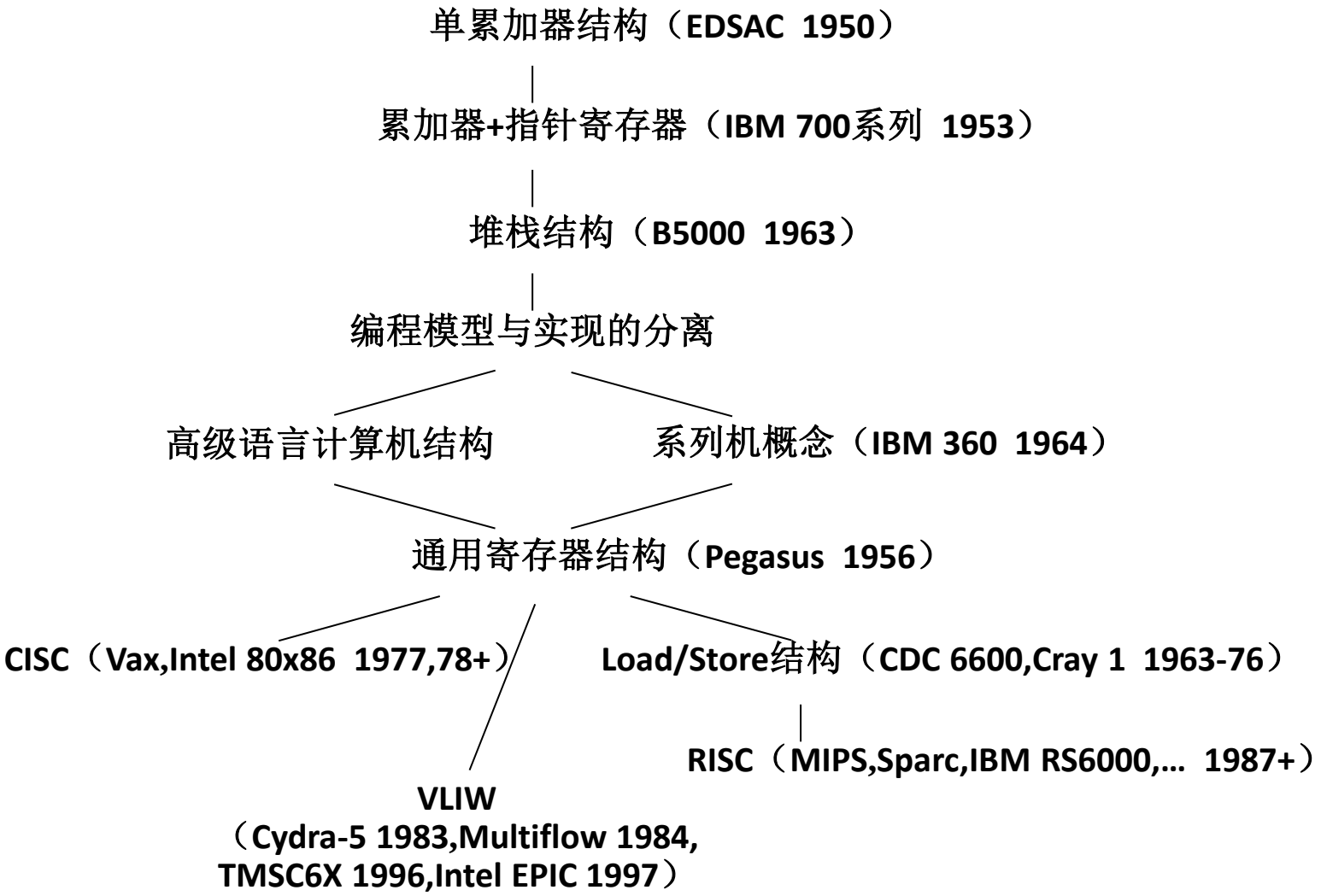


图5.17 指令系统结构的发展

CISC

复杂指令集计算机

Complicated Instruction Set Computer

5.5.1 CISC

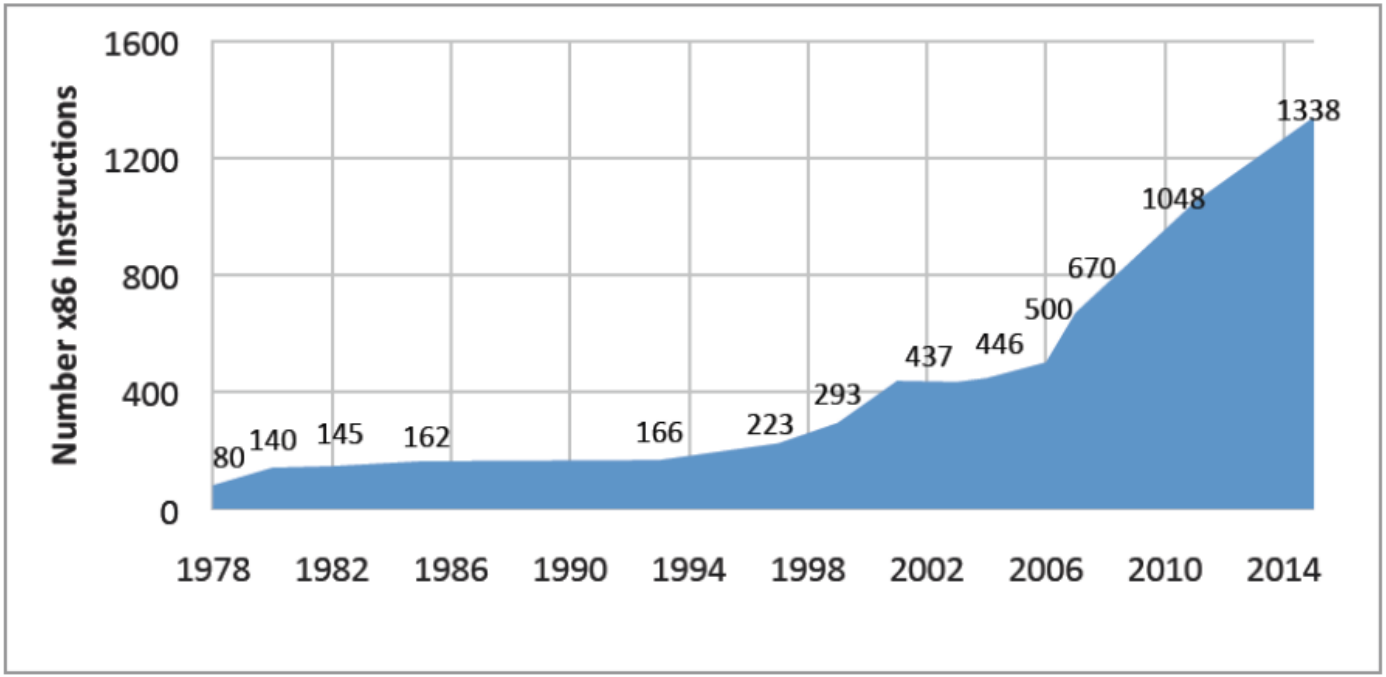
- 由于硬件实现的限制，早期的计算机是简单的。随着大规模集成电路的发展和对计算机功能日趋强大的需求，指令系统受到硬件可以实现相当复杂的结构和成本一路下降的支持，变得越来越丰富和完备，形成了复杂指令集计算机，即CISC。
- 早期CISC设计有如下特点：
  - 指令系统复杂，即指令数多、寻址方式多、指令格式多；
  - 绝大多数指令执行需要多个时钟周期；
  - 有多种指令可以访问存储器(非load/store结构)；
  - CPU控制器采用微程序控制方式实现；
  - 寄存器数量有限。

5.5.1 CISC

- CISC设计初衷之一是通过强化指令功能来简化编译器。实际结果是CISC并非实质简化了编译器，也并不一定总是可以生成更小、更快的程序。
- 规模庞大的指令系统在提供丰富指令、有助于灵活程序设计的同时，也存在诸多弊端：
  - 为适应丰富的指令系统，CPU控制器必须很复杂，相应的程序控制存储装置必须有更大空间，这会增加指令执行的时间，抵消复杂指令预期带来的速度优势。
  - 繁多的指令会使计算机研制周期变长，调试和维护难度变大。
  - 复杂指令系统必然增加硬件设计和制造的复杂性，增加了研制成本。

5.5.1 CISC

- x86指令集指令数量增长情况



–对数量庞大的指令的支持使得CPU复杂度很高，同时成本也很高

Ref: Patterson and Waterman, 《RISC-V手册》

5.5.1 CISC

- 统计结果表明，典型程序中80%的语句仅使用CPU中20%的指令，且大多数是运算、取数、转移等简单指令。这就是经典的80-20规律。

  例如，80x86中最常用的10种指令(如加、减、乘、调用、条件转移、存储器访问、寄存器间传送、比较等)占到程序的95%。

- 80-20规律的一个重要启示是：能否仅使用最常用的20%的简单指令来重新组合不常用的80%的指令功能？这实际上成为引发RISC技术的又一原因。

RISC

精简指令集计算机

Reduced Instruction Set Computer

5.5.2 RISC

- 因计算机硬、软件在逻辑上具有等效性，使指令系统的精简成为可能。
- RISC希望指令系统留下最常用的20%的简单指令，指令系统被精简掉的部分可用其它硬件或软件（包括编译程序）的功能来代替，通过优化硬件设计，提高时钟频率，以速度取胜。

5.5.2 RISC

- RISC结构计算机具有如下特点：
  - 只设置使用频度高的简单指令，所以指令的操作种类少，寻址方式少；
  - 指令格式规则，长度固定，便于简单统一的译码，可使控制器简化、硬件结构精简；
  - 仅通过load和store指令访问主存；
  - 通用寄存器数量多，一般有几十甚至几百个，大多数操作在寄存器之间进行；
  - 在非流水线RISC中，单条指令可在单机器周期内完成；在流水线RISC中，对于大多数指令有*CPI*=1；
  - 采用硬布线控制器，不使用微代码（即微程序），有利于提高时钟频率和CPU速度，能够更好地响应中断；
  - 可简化硬件设计，降低成本及便于超大规模集成电路实现；
  - 有利于多流水线、多核CPU实现；
  - 适宜高度优化编译器（即编译程序）；
  - 精简的指令使程序阅读、分析难度加大；
  - 不能同CISC兼容。解决办法一是将源程序在CISC机器上重新编译，办法二是用目标代码翻译器将CISC代码翻译为RISC代码。

5.5.2 RISC

- 随着芯片密度和硬件速度的提高，RISC系统已经越来越复杂（采用多流水线、多核结构），同时CISC设计也在关注与RISC相同的技术焦点，如增加通用寄存器数量和更加强调指令流水线设计等。
- 实践表明，RISC设计中包括某些CISC特色会有好处，CISC设计吸纳RISC优点更有利于增强自身性能。
- 目前展现的发展趋势是RISC和CISC正逐渐融合。例如RISC代表之一的PowerPC处理器设计融入了CISC技术，而CISC代表之一的Pentium处理器设计则采纳了RISC技术。
- 未来，RISC系统是否会完全取代CISC系统，或RISC-V指令系统结构是否会成为主流，我们将拭目以待。

指令系统实例

5.6 指令系统实例

Intel CPU指令集

- Intel CPU是CISC的典型代表，其指令系统丰富而强大，具有完全向后兼容能力，指令格式多样、长度可变，一直以最能体现CISC特性而著称。

RISC-V指令集

- RISC-V指令系统是RISC的最新、优化的代表，它保持了其他RISC指令集具有的指令格式少、长度固定、规模较小、速度快等特性，摒弃或修改了其他指令集的缺陷。

指令系统实例

-- RISC-V指令格式及示例

5.6.1 RISC-V指令系统

1. RISC-V简介

- 2011年推出，受到很多芯片制造商、科研机构与大学、个人的关注。
- 是第一款开源的指令集架构，开源让RISC-V的免费、共享成为可能，从而可以降低成本。
- 与以往的ISA不同，RISC-V是简约、模块化的。它的核心模块是固定不变的基础整数ISA，即RV32/64I，简洁到仅有47条指令。
- 适合多种计算机，RISC-V架构允许其指令系统由基本模块RV32/64I和零到若干个扩展模块（由RISC-V基金会认定的）组成。
- RISC-V基金会保证基本指令集和经核准的可选扩展部分是不变的ISA，这种稳定性使得软件开发者确信为RISC-V编写的软件可以永远运行在所有RISC-V核上，而软件管理者可以依赖它来保护他们的软件投资。
- RISC-V开放、可扩展的方式在体系结构上更便于软件和硬件的自由组合，也支持为需要特定加速或特殊功能的设计提供定制指令。
- 正是由于RISC-V属于一个开放的、非营利性质的基金会，所以RISC-V完全有可能挑战主流专有ISA的主导地位。

RISC-V基金会成员

riscv.org/membership/

5.6.1 RISC-V指令系统

2. RISC-V指令格式

- RISC-V ISA有6种基本指令格式，R型用于寄存器-寄存器运算，I型用于立即数运算和访存load操作，S型用于访存store操作，B型用于条件分支操作，U型用于立即数加载和加PC，J型用于无条件跳转。

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

- opcode: 操作码，表示指令的基本操作
- rd: 目的操作数寄存器
- funct3, funct7: 附加操作码/功能码
- rs1/rs2: 源操作数寄存器
- imm: 立即数或地址偏移

5.6.1 RISC-V指令系统

2. RISC-V指令格式

- RISC-V ISA有6种基本指令格式，R型用于寄存器-寄存器运算，I型用于立即数运算和访存load操作，S型用于访存store操作，B型用于条件分支操作，U型用于立即数加载和加PC，J型用于无条件跳转。

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

- RISC-V所有指令采用固定32位长，指令的简单规整极大简化了其获取；
- 使用三个操作数字段，避免了一个源操作数和目的操作数的冲突；
- 所有指令的寄存器地址字段总是在指令同一位置，这可以实现在译码指令之前先访问寄存器，而无需添加额外的译码逻辑，有助于简化设计降低成本；
- 立即数是带符号数，且立即数字段总是在指令的高位，这使得立即数符号扩展可以在指令译码之前进行。

RV32I指令

RV32I

Integer Computation

- **add** { – / immediate }
- subtract
- { **and** / **or** / exclusive **or** } { – / immediate }
- { shift left logical / shift right arithmetic / shift right logical } { – / immediate }
- load upper immediate
- add upper immediate to **pc**
- set less than { – / immediate } { – / unsigned }

Control transfer

- branch { equal / not equal }
- branch { greater than or equal / less than } { – / unsigned }
- jump and link { – / register }

Loads and Stores

- { load / store } { byte / halfword / word }
- load { byte / halfword } unsigned

Miscellaneous instructions

- **fence** loads & stores
- **fence**.instruction & data
- environment { **break** / **call** }
- control status register { read & clear bit / read & set bit / read & write } { – / immediate }

set less than { – / immediate } { – / unsigned }

表示四条指令slt, slti, sltu, sltiu.

RV32I指令及映射

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

图5.19

RV32I指令及映射

31 25	24 20	19 15	14 12	11 7	6 0	
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
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

图5.19

5.6.1 RISC-V指令系统

3. RISC-V指令示例

- 为了给ISA扩展留出足够的空间，最基础的RV32I指令集只使用了32位指令字中编码空间的不到八分之一。
- 架构师们也仔细挑选了RV32I操作码（见图5.19），使拥有共同数据通路的指令操作码有尽可能多的位编码是一样的，以此简化控制逻辑。
- 有限且尽可能相似的指令格式可以降低硬件复杂度，加快指令译码速度。

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

图5.19

对比MIPS指令格式

31 26	25 21	20 16	15 11	10 6	5 0	指令说明
SPECIAL 000000	rs	rt	rd	00000	ADD 100000	32位整数加法：ADD rd, rs, rt GPR[rd] ← GPR[rs] + GPR[rt]
31 26	25 21	20 16	15 6		5 0	
SPECIAL 000000	rs	rt	00 0000 0000		DDIV 011110	64位带符号整数除法：DDIV rs, rt (LO, HI) ← GPR[rs] / GPR[rt]
31 26	25 21	20 11	10 6		5 0	
SPECIAL 000000	rs	00 0000 0000	hint		JR 001000	无条件跳转：JR rs PC ← GPR[rs]
31 26	25 16	15 11	10 6		5 0	
SPECIAL 000000	00 0000 0000	rd	00000		MFHI 010000	传送：MFHI rd GPR[rd] ← HI，HI为专用寄存器
31 26	25 21	20 6			5 0	
SPECIAL 000000	rs	000 0000 0000 0000			MTHI 010001	传送：MTHI rs HI ← GPR[rs]
31 26	25 11	10 6			5 0	
SPECIAL 000000	00 0000 0000 0000 0	stype			SYNC 001111	定制（order）同步加载和存储操作： SYNC (stype = 0 implied)
31 26	25 6				5 0	
SPECIAL 000000	code				BREAK 001101	断点异常：BREAK
31 26	25 21	20 16	15 11	10 3	2 0	
COP0 010000	DMF 00001	rt	rd	0000 0000	sel	传送：DMFC0 rt, rd, sel GPR[rt] ← CPR[0,rd,sel]，协处理器0寄存器内容传送到通用寄存器中
31 26	25 24		6		5 0	
COP0 010000	CO 1	000 0000 0000 0000 0000			DERET 011111	Debug异常返回：DERET
31 26	25 21	20 18	17 16	15	0	
COP1 010001	BC 01000	cc	nd 0	tf 0	offset	条件分支：BC1F cc, offset if FPConditionCode(cc) = 0 then branch 测试浮点处理器FP的条件码，目标地址由PC相对寻址确定
31 26	25 21	20 16	15 11	10	0	
COP1 010001	CF 00010	rt	fs	000 0000 0000		传送：CFC1 rt, fs GPR[rt] ←FP_Control[FPR[fs]]
31 26	25 24			0		
COP2 010010	CO 1	cofun				操作协处理器2：COP2 func CoprocessorOperation(2, cofun)
31 26	25 21	20 16	15	0		
REGIMM 000001	rs	TEQI 01100	immediate			比较：TEQI rs, immediate if GPR[rs] = immediate then Trap
31 26	25	0				
J 000010	instr_index					无条件跳转：J target 目标地址为现行的边界对齐的256 MB范围

指令系统实例

--Intel指令格式及示例

5.6.2 Intel CPU指令系统

1. Intel CPU指令系统发展

- 随着Intel CPU功能不断提升，其指令格式也在不断扩展，指令数量也在不断扩大，但从一而终坚守向后兼容特性。

最先使用新指令的CPU	新指令发布时间	新增指令	指令总数(条)
8086	1978.6	89条	89
8087	1978	77条	166
80286	1982.2	新增24条	190
80386	1986.20	新增14条	204
80387	1985	新增7条	211
80486	1989.8	新增5条	216
Pentium	1993.3	新增6条	222
Pentium Pro	1996.21	新增8条	230
Pentium MMX	1997.1	新增MMX指令57条	287
Pentium Ⅱ	1997.5	新增4条	291
Pentium Ⅲ	1999.2	新增SSE指令70条	361
Pentium 4	2001.7	新增SSE2指令144条	505
Pentium 4 (Prescott核心) Core微架构	2004年初	新增SSE3指令13条 SSE3补充版本SSSE3指令16条	518 534
Core 2(45nm Penryn核心) Core i7 (Nehalem微架构)	2007.4	新增SSE 4.1指令47条 新增SSE 4.2指令7条	581 588
Sandy Bridge架构(32nm)	2008.4	新AVX指令集	

5.6.2 Intel CPU指令系统

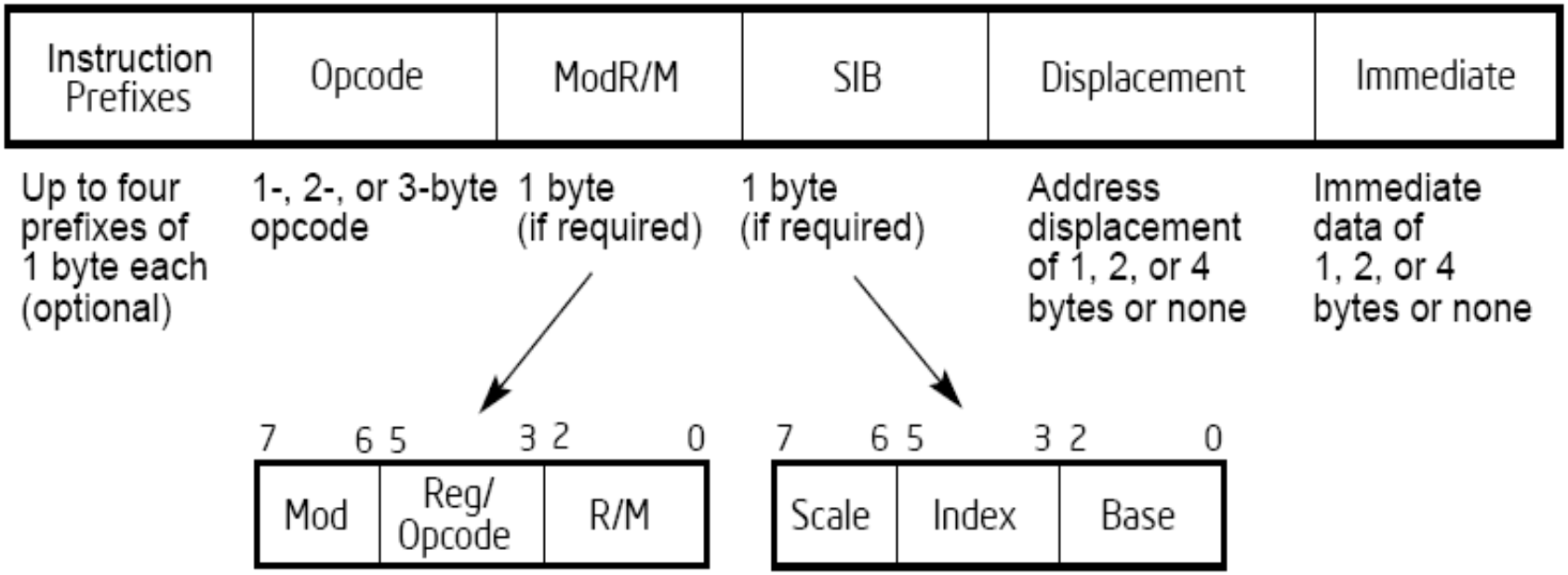
2. Intel 64和IA-32体系结构的指令格式

- IA-32汇编语句（指令）格式为：

  label: mnemonic argument1, argument2, argument3

  例如：LoadReg: MOV EAX, Date

- Intel 64和IA-32体系结构指令编码格式如图5.20所示，它是对8086指令格式的扩展。指令长度最短1字节，最长17字节。



5.6.2 Intel CPU指令系统

3. IA-32e模式的指令格式

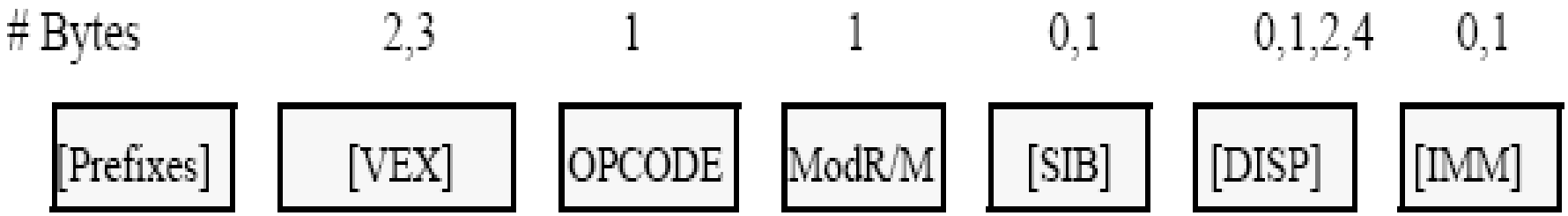
- IA-32e模式有两个子模式：
  - 兼容模式（Compatibility Mode）。允许64位操作系统运行大多数继承的未修改的保护模式软件。
  - 64位模式（64-Bit Mode）。允许64位操作系统运行访问64位地址空间的应用程序。
- 64位模式的指令格式见图5.21，与图5.20比较，它在IA-32指令格式的基础上又增加了1字节的REX前缀。

Legacy Prefixes	REX Prefix	Opcode	ModR/M	SIB	Displacement	Immediate
Grp 1, Grp 2, Grp 3, Grp 4 (optional)	(optional)	1-, 2-, or 3-byte opcode	1 byte (if required)	1 byte (if required)	Address displacement of 1, 2, or 4 bytes	Immediate data of 1, 2, or 4 bytes or none

5.6.2 Intel CPU指令系统

4. Intel AVX(advanced vector extensions)指令格式

- Intel AVX指令的编码机制是将前缀字节、操作码扩展域、操作数编码域、向量长度编码能力组合到新的前缀VEX中。
- 图5.22示意了具有VEX前缀支持的Intel 64指令编码格式。



5.6.2 Intel CPU指令系统

5. Intel 64 and IA-32典型指令示例

机器指令(16进制编码) (指令长度*)	助记符(汇编)指令	指令功能
8B */r* (2字节)	MOV *r32,r/m32*	*r/m32*指定的32位寄存器或存储单元内容传送至*r32*指定的32位寄存器中。
REX.W + 89 */r* (3字节)	MOV *r/m64,r64*	*r64*指定的64位寄存器内容传送至*r/m64*指定的64位寄存器或存储单元中。
REX.W + 81 /0 *id* (7字节)	ADD *r/m64,imm32*	将32位立即数*imm32*符号扩展，与64位*r/m64*相加，结果存入*r/m64*中。
VEX.NDS.256.66.0F.WIG 58 /r (5字节)	VADDPD ymm1, ymm2,ymm3/m256	将来自ymm3/mem的打包的双精度浮点数与ymm2相加，结果存于ymm1中。（AVX指令）
0F 84 *cd* (6字节)	JZ *rel32*	如果结果为0(ZF=1)，则实现近程跳转，目标地址为： EIP←EIP + SignExtend(DEST) 汇编指令中*rel32*是目标地址标号；机器指令中*rel32*表示相对偏移地址DEST。
EA *cp* (7字节)	JMP *ptr16:32*	远程无条件跳转，绝对地址寻址，目标地址由指令操作数*ptr16:32*直接提供确定。
F2 0F 5F */r* (4字节)	MAXSD *xmm1, xmm2/m64*	返回在*xmm2/mem64*和*xmm1*之间的最大标量双精度浮点数值。（SSE2指令）
E5 *ib* (2字节)	IN EAX, i*mm8*	从*imm8*确定的I/O端口地址输入双字数据到寄存器EAX中。
0F 01 /2 (3字节)	LGDT *m16&64*	将*m*装入到GDTR(global descriptor table register)中。*m*指定存储单元，它内含8字节基地址和2字节GDT界限(limit)。该指令仅供操作系统使用。
F4 (1字节)	HLT	停止指令执行。
F0 (1字节)	LOCK	在LOCK前缀伴随的指令执行期间，使LOCK#信号生效。
*不含非强制性前缀。		

本章要求

- 了解计算机的指令格式，掌握指令扩展操作码技术
- 掌握指令设计方法
- 掌握寻址方式
- 了解CISC与RISC的概念，了解理解精简指令系统计算机RISC的特点