K

Survey of Instruction Set Architectures

RISC: any computer announced after 1985.

Steven Przybylski
A Designer of the Stanford MIPS

K.1 Introduction

This appendix covers 13 instruction set architectures, some of which remain a vital part of the IT industry and some of which have retired to greener pastures. We keep them all in part to show the changes in fashion of instruction set architecture over time.

We start with ten RISC architectures. There are billions of dollars of computers shipped each year for ARM (including Thumb), MIPS (including MIPS16), Power, and SPARC. Indeed, ARM dominates embedded computing. However, the Digital Alpha and HP PA-RISC were both shoved aside by Itanium, and they remain primarily of historical interest.

The 80x86 remains a dominant ISA, dominating the desktop and the low-end of the server market. It has been extended more than any other ISA in this book, and there are no plans to stop it soon. Now that it has made the transition to 64-bit addressing, we expect this architecture to be around longer than your authors.

The VAX typifies an ISA where the emphasis was on code size and offering a higher level machine language in the hopes of being a better match to programming languages. The architects clearly expected it to be implemented with large amounts of microcode, which made single chip and pipelined implementations more challenging. Its successor was the Alpha, which had a short life.

The vulnerable IBM 360/370 remains a classic that set the standard for many instruction sets to follow. Among the decisions the architects made in the early 1960s were:

- 8-bit byte
- Byte addressing
- 32-bit words
- 32-bit single precision floating-point format + 64-bit double precision floating-point format
- 32-bit general-purpose registers, separate 64-bit floating-point registers
- Binary compatibility across a family of computers with different cost-performance
- Separation of architecture from implementation

As mentioned in Chapter 2, the IBM 370 was extended to be virtualizable, so it has the lowest overhead for a virtual machine of any ISA. The IBM 360/370 remains the foundation of the IBM mainframe business in a version that has extended to 64 bits.

K.2 A Survey of RISC Architectures for Desktop, Server, and Embedded Computers

Introduction

We cover two groups of Reduced Instruction Set Computer (RISC) architectures in this section. The first group is the desktop and server RISCs:

- Digital Alpha
- MIPS, Inc.
- Hewlett-Packard PA-RISC
- IBM and Motorola PowerPC
- Sun Microsystems SPARC

The second group is the embedded RISCs:

- Advanced RISC Machines ARM
- Advanced RISC Machines Thumb
- Hitachi SuperH
- Mitsubishi M32R
- MIPS, Inc. MIPS16

Although three of these architectures have faded over time—namely, the Alpha, PA-RISC, and M32R—there has never been another class of computers so similar.

There has never been another class of computers so similar. This similarity allows the presentation of 10 architectures in about 50 pages. Characteristics of the desktop and server RISCs are found in Figure K.1 and the embedded RISCs in Figure K.2.

Notice that the embedded RISCs tend to have 8 to 16 general-purpose registers, while the desktop/server RISCs have 32, and that the length of instructions is 16 to 32 bits in embedded RISCs but always 32 bits in desktop/server RISCs.

Although shown as separate embedded instruction set architectures, Thumb and MIPS16 are really optional modes of ARM and MIPS invoked by call instructions. When in this mode they execute a subset of the native architecture using 16-bit-long instructions. These 16-bit instruction sets are not intended to be full architectures, but they are enough to encode most procedures. Both machines expect procedures to be homogeneous, with all instructions in either 16-bit mode or 32-bit mode. Programs will consist of procedures in 16-bit mode for density or in 32-bit mode for performance.

One complication of this description is that some of the older RISCs have been extended over the years. We decided to describe more recent versions of the architectures: Alpha version 3, MIPS64, PA-RISC 2.0, and SPARC version 9 for the desktop/server; ARM version 4, Thumb version 1, Hitachi SuperH SH-3, M32R version 1, and MIPS16 version 1 for the embedded ones.

	Alpha	MIPS I	PA-RISC 1.1	PowerPC	SPARC v.8
Date announced	1992	1986	1986	1993	1987
Instruction size (bits)	32	32	32	32	32
Address space (size, model)	64 bits, flat	32 bits, flat	48 bits, segmented	32 bits, flat	32 bits, flat
Data alignment	Aligned	Aligned	Aligned	Unaligned	Aligned
Data addressing modes	1	1	5	4	2
Protection	Page	Page	Page	Page	Page
Minimum page size	8 KB	4 KB	4 KB	4 KB	8 KB
I/O	Memory mapped	Memory mapped	Memory mapped	Memory mapped	Memory mapped
Integer registers (number, model, size)	31 GPR × 64 bits	31 GPR × 32 bits	31 GPR × 32 bits	32 GPR × 32 bits	31 GPR × 32 bits
Separate floating-point registers	31 × 32 or 31 × 64 bits	16 × 32 or 16 × 64 bits	56 × 32 or 28 × 64 bits	32 × 32 or 32 × 64 bits	32 × 32 or 32 × 64 bits
Floating-point format	IEEE 754 single, double	IEEE 754 single, double	IEEE 754 single, double	IEEE 754 single, double	IEEE 754 single, double

Figure K.1 Summary of the first version of five recent architectures for desktops and servers. Except for the number of data address modes and some instruction set details, the integer instruction sets of these architectures are very similar. Contrast this with Figure K.34. Later versions of these architectures all support a flat, 64-bit address space.

	ARM	Thumb	SuperH	M32R	MIPS16
Date announced	1985	1995	1992	1997	1996
Instruction size (bits)	32	16	16	16/32	16/32
Address space (size, model)	32 bits, flat	32 bits, flat	32 bits, flat	32 bits, flat	32/64 bits, flat
Data alignment	Aligned	Aligned	Aligned	Aligned	Aligned
Data addressing modes	6	6	4	3	2
Integer registers (number, model, size)	15 GPR × 32 bits	8 GPR + SP, LR × 32 bits	16 GPR × 32 bits	16 GPR × 32 bits	8 GPR + SP, RA × 32/64 bits
I/O	Memory mapped	Memory mapped	Memory mapped	Memory mapped	Memory mapped

Figure K.2 Summary of five recent architectures for embedded applications. Except for number of data address modes and some instruction set details, the integer instruction sets of these architectures are similar. Contrast this with Figure K.34.

The remaining sections proceed as follows. After discussing the addressing modes and instruction formats of our RISC architectures, we present the survey of the instructions in five steps:

- Instructions found in the MIPS core, which is defined in Appendix A of the main text
- Multimedia extensions of the desktop/server RISCs

- Digital signal-processing extensions of the embedded RISCs
- Instructions not found in the MIPS core but found in two or more architectures
- The unique instructions and characteristics of each of the 10 architectures

We give the evolution of the instruction sets in the final section and conclude with a speculation about future directions for RISCs.

Addressing Modes and Instruction Formats

Figure K.3 shows the data addressing modes supported by the desktop architectures. Since all have one register that always has the value 0 when used in address modes, the absolute address mode with limited range can be synthesized using `zero` as the base in displacement addressing. (This register can be changed by arithmetic-logical unit (ALU) operations in PowerPC; it is always 0 in the other machines.) Similarly, register indirect addressing is synthesized by using displacement addressing with an offset of 0. Simplified addressing modes is one distinguishing feature of RISC architectures.

Figure K.4 shows the data addressing modes supported by the embedded architectures. Unlike the desktop RISCs, these embedded machines do not reserve a register to contain 0. Although most have two to three simple addressing modes, ARM and SuperH have several, including fairly complex calculations. ARM has an addressing mode that can shift one register by any amount, add it to the other registers to form the address, and then update one register with this new address.

References to code are normally PC-relative, although jump register indirect is supported for returning from procedures, for case statements, and for pointer function calls. One variation is that PC-relative branch addresses are shifted left

Addressing mode	Alpha	MIPS64	PA-RISC 2.0	PowerPC	SPARC v.9
Register + offset (displacement or based)	X	X	X	X	X
Register + register (indexed)	—	X (FP)	X (Loads)	X	X
Register + scaled register (scaled)	—	—	X	—	—
Register + offset and update register	—	—	X	X	—
Register + register and update register	—	—	X	X	—

Figure K.3 Summary of data addressing modes supported by the desktop architectures. PA-RISC also has short address versions of the offset addressing modes. MIPS64 has indexed addressing for floating-point loads and stores. (These addressing modes are described in Figure A.6 on page A-9.)

Addressing mode	ARM v.4	Thumb	SuperH	M32R	MIPS16
Register + offset (displacement or based)	X	X	X	X	X
Register + register (indexed)	X	X	X	—	—
Register + scaled register (scaled)	X	—	—	—	—
Register + offset and update register	X	—	—	—	—
Register + register and update register	X	—	—	—	—
Register indirect	—	—	X	X	—
Autoincrement, autodecrement	X	X	X	X	—
PC-relative data	X	X (loads)	X	—	X (loads)

Figure K.4 Summary of data addressing modes supported by the embedded architectures. SuperH and M32R have separate register indirect and register + offset addressing modes rather than just putting 0 in the offset of the latter mode. This increases the use of 16-bit instructions in the M32R, and it gives a wider set of address modes to different data transfer instructions in SuperH. To get greater addressing range, ARM and Thumb shift the offset left 1 or 2 bits if the data size is half word or word. (These addressing modes are described in Figure A.6 on page A-9.)

2 bits before being added to the PC for the desktop RISCs, thereby increasing the branch distance. This works because the length of all instructions for the desktop RISCs is 32 bits and instructions must be aligned on 32-bit words in memory. Embedded architectures with 16-bit-long instructions usually shift the PC-relative address by 1 for similar reasons.

Figure K.5 shows the format of the desktop RISC instructions, which includes the size of the address in the instructions. Each instruction set architecture uses these four primary instruction formats. Figure K.6 shows the six formats for the embedded RISC machines. The desire to have smaller code size via 16-bit instructions leads to more instruction formats.

Figures K.7 and K.8 show the variations in extending constant fields to the full width of the registers. In this subtle point, the RISCs are similar but not identical.

Instructions: The MIPS Core Subset

The similarities of each architecture allow simultaneous descriptions, starting with the operations equivalent to the MIPS core.

MIPS Core Instructions

Almost every instruction found in the MIPS core is found in the other architectures, as Figures K.9 through K.13 show. (For reference, definitions of the MIPS instructions are found in Section A.9.) Instructions are listed under four categories: data transfer (Figure K.9); arithmetic, logical (Figure K.10); control (Figure K.11); and floating point (Figure K.12). A fifth category (Figure K.13) shows conventions for register usage and pseudoinstructions on each architecture. If a MIPS

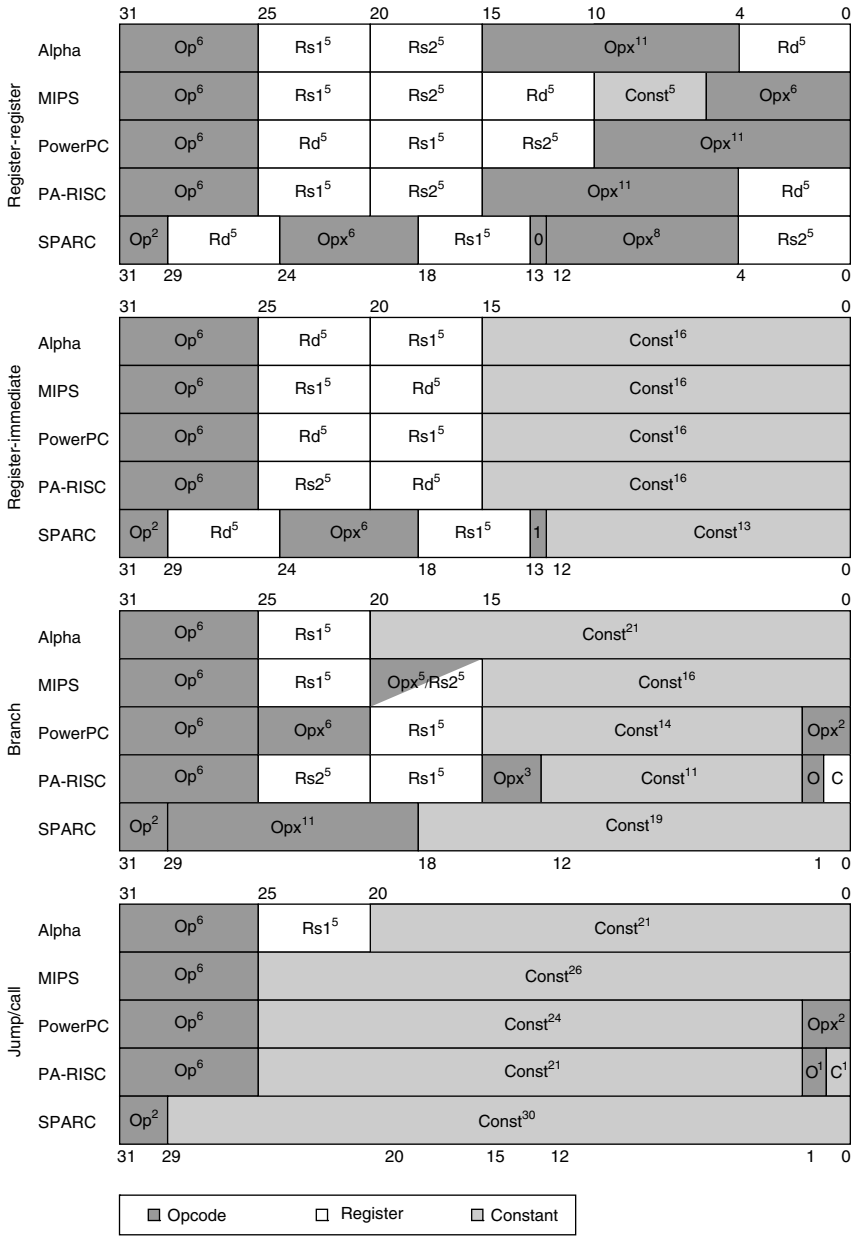

Figure K.5 Instruction formats for desktop/server RISC architectures. These four formats are found in all five architectures. (The superscript notation in this figure means the width of a field in bits.) Although the register fields are located in similar pieces of the instruction, be aware that the destination and two source fields are scrambled. Op = the main opcode, Opx = an opcode extension, Rd = the destination register, Rs1 = source register 1, Rs2 = source register 2, and Const = a constant (used as an immediate or as an address). Unlike the other RISCs, Alpha has a format for immediates in arithmetic and logical operations that is different from the data transfer format shown here. It provides an 8-bit immediate in bits 20 to 13 of the RR format, with bits 12 to 5 remaining as an opcode extension.

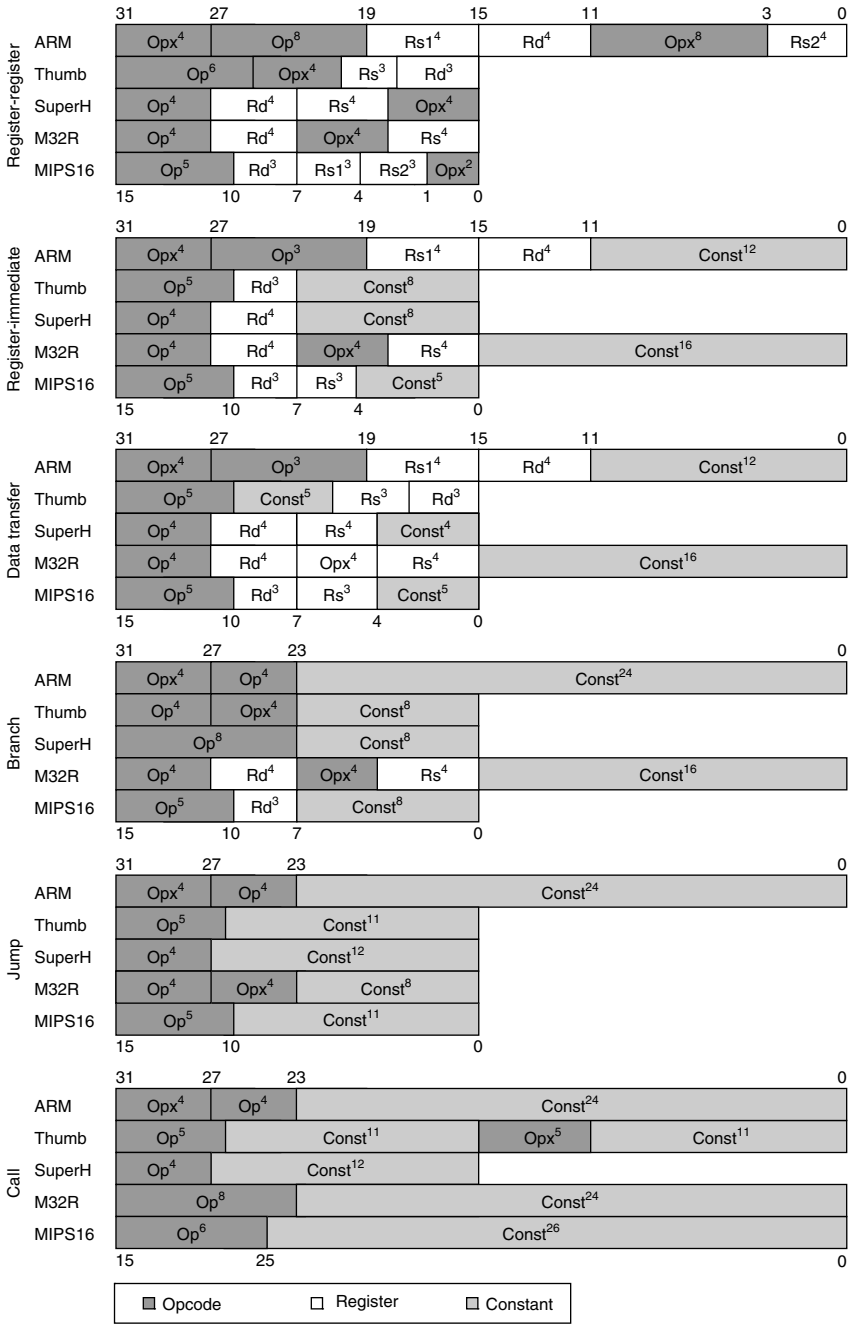

Figure K.6 Instruction formats for embedded RISC architectures. These six formats are found in all five architectures. The notation is the same as Figure K.5. Note the similarities in branch, jump, and call formats and the diversity in register-register, register-immediate, and data transfer formats. The differences result from whether the architecture has 8 or 16 registers, whether it is a 2- or 3-operand format, and whether the instruction length is 16 or 32 bits.

Format: instruction category	Alpha	MIPS64	PA-RISC 2.0	PowerPC	SPARC v.9
Branch: all	Sign	Sign	Sign	Sign	Sign
Jump/call: all	Sign	—	Sign	Sign	Sign
Register-immediate: data transfer	Sign	Sign	Sign	Sign	Sign
Register-immediate: arithmetic	Zero	Sign	Sign	Sign	Sign
Register-immediate: logical	Zero	Zero	—	Zero	Sign

Figure K.7 Summary of constant extension for desktop RISCs. The constants in the jump and call instructions of MIPS are not sign extended since they only replace the lower 28 bits of the PC, leaving the upper 4 bits unchanged. PA-RISC has no logical immediate instructions.

Format: instruction category	ARM v.4	Thumb	SuperH	M32R	MIPS16
Branch: all	Sign	Sign	Sign	Sign	Sign
Jump/call: all	Sign	Sign/Zero	Sign	Sign	—
Register-immediate: data transfer	Zero	Zero	Zero	Sign	Zero
Register-immediate: arithmetic	Zero	Zero	Sign	Sign	Zero/Sign
Register-immediate: logical	Zero	—	Zero	Zero	—

Figure K.8 Summary of constant extension for embedded RISCs. The 16-bit-length instructions have much shorter immediates than those of the desktop RISCs, typically only 5 to 8 bits. Most embedded RISCs, however, have a way to get a long address for procedure calls from two sequential half words. The constants in the jump and call instructions of MIPS are not sign extended since they only replace the lower 28 bits of the PC, leaving the upper 4 bits unchanged. The 8-bit immediates in ARM can be rotated right an even number of bits between 2 and 30, yielding a large range of immediate values. For example, all powers of 2 are immediates in ARM.

core instruction requires a short sequence of instructions in other architectures, these instructions are separated by semicolons in Figures K.9 through K.13. (To avoid confusion, the destination register will always be the leftmost operand in this appendix, independent of the notation normally used with each architecture.) Figures K.14 through K.17 show the equivalent listing for embedded RISCs. Note that floating point is generally not defined for the embedded RISCs.

Every architecture must have a scheme for compare and conditional branch, but despite all the similarities, each of these architectures has found a different way to perform the operation.

Compare and Conditional Branch

SPARC uses the traditional four condition code bits stored in the program status word: *negative*, *zero*, *carry*, and *overflow*. They can be set on any arithmetic or logical instruction; unlike earlier architectures, this setting is optional on each instruction. An explicit option leads to fewer problems in pipelined implementation. Although condition codes can be set as a side effect of an operation, explicit

Data transfer (instruction formats)	R-I	R-I	R-I, R-R	R-I, R-R	R-I, R-R
Instruction name	**Alpha**	**MIPS64**	**PA-RISC 2.0**	**PowerPC**	**SPARC v.9**
Load byte signed	LDBU; SEXTB	LB	LDB; EXTRW,S 31,8	LBZ; EXTSB	LDSB
Load byte unsigned	LDBU	LBU	LDB, LDBX, LDBS	LBZ	LDUB
Load half word signed	LDWU; SEXTW	LH	LDH; EXTRW,S 31,16	LHA	LDSH
Load half word unsigned	LDWU	LHU	LDH, LDHX, LDHS	LHZ	LDUH
Load word	LDLS	LW	LDW, LDWX, LDWS	LW	LD
Load SP float	LDS*	LWC1	FLDWX, FLDWS	LFS	LDF
Load DP float	LDT	LDC1	FLDDX, FLDDS	LFD	LDDF
Store byte	STB	SB	STB, STBX, STBS	STB	STB
Store half word	STW	SH	STH, STHX, STHS	STH	STH
Store word	STL	SW	STW, STWX, STWS	STW	ST
Store SP float	STS	SWC1	FSTWX, FSTWS	STFS	STF
Store DP float	STT	SDC1	FSTDX, FSTDS	STFD	STDF
Read, write special registers	MF_, MT_	MF, MT_	MFCTL, MTCTL	MFSPR, MF_, MTSPR, MT_	RD, WR, RDPR, WRPR, LDXFSR, STXFSR
Move integer to FP register	ITOFS	MFC1/ DMFC1	STW; FLDWX	STW; LDFS	ST; LDF
Move FP to integer register	FTTOIS	MTC1/ DMTC1	FSTWX; LDW	STFS; LW	STF; LD

Figure K.9 Desktop RISC data transfer instructions equivalent to MIPS core. A sequence of instructions to synthesize a MIPS instruction is shown separated by semicolons. If there are several choices of instructions equivalent to MIPS core, they are separated by commas. For this figure, half word is 16 bits and word is 32 bits. Note that in Alpha, LDS converts single-precision floating point to double precision and loads the entire 64-bit register.

compares are synthesized with a subtract using r0 as the destination. SPARC conditional branches test condition codes to determine all possible unsigned and signed relations. Floating point uses separate condition codes to encode the IEEE 754 conditions, requiring a floating-point compare instruction. Version 9 expanded SPARC branches in four ways: a separate set of condition codes for 64-bit operations; a branch that tests the contents of a register and branches if the value is =, not=, <, <=, >=, or <= 0 (see MIPS below); three more sets of floating-point condition codes; and branch instructions that encode static branch prediction.

PowerPC also uses four condition codes: *less than*, *greater than*, *equal*, and *summary overflow*, but it has eight copies of them. This redundancy allows the PowerPC instructions to use different condition codes without conflict, essentially giving PowerPC eight extra 4-bit registers. Any of these eight condition codes can be the target of a compare instruction, and any can be the source of a

Arithmetic/logical (instruction formats)	R-R, R-I	R-R, R-I	R-R, R-I	R-R, R-I	R-R, R-I
Instruction name	**Alpha**	**MIPS64**	**PA-RISC 2.0**	**PowerPC**	**SPARC v.9**
Add	ADDL	ADDU, ADDU	ADDL, LDO, ADDI, UADDCM	ADD, ADDI	ADD
Add (trap if overflow)	ADDLV	ADD, ADDI	ADDO, ADDIO	ADDO; MCRXR; BC	ADDcc; TVS
Sub	SUBL	SUBU	SUB, SUBI	SUBF	SUB
Sub (trap if overflow)	SUBLV	SUB	SUBTO, SUBIO	SUBF/oe	SUBcc; TVS
Multiply	MULL	MULT, MULTU	SHiADD;...; (i=1,2,3)	MULLW, MULLI	MULX
Multiply (trap if overflow)	MULLV	—	SHiADDO;...;	—	—
Divide	—	DIV, DIVU	DS;...; DS	DIVW	DIVX
Divide (trap if overflow)	—	—	—	—	—
And	AND	AND, ANDI	AND	AND, ANDI	AND
Or	BIS	OR, ORI	OR	OR, ORI	OR
Xor	XOR	XOR, XORI	XOR	XOR, XORI	XOR
Load high part register	LDAH	LUI	LDIL	ADDIS	SETHI (B fmt.)
Shift left logical	SLL	SLLV, SLL	DEPW, Z 31-i,32-i	RLWINM	SLL
Shift right logical	SRL	SRLV, SRL	EXTRW, U 31, 32-i	RLWINM 32-i	SRL
Shift right arithmetic	SRA	SRAV, SRA	EXTRW, S 31, 32-i	SRAW	SRA
Compare	CMPEQ, CMPLT, CMPLE	SLT/U, SLTI/U	COMB	CMP(I)CLR	SUBcc r0,...

Figure K.10 Desktop RISC arithmetic/logical instructions equivalent to MIPS core. Dashes mean the operation is not available in that architecture, or not synthesized in a few instructions. Such a sequence of instructions is shown separated by semicolons. If there are several choices of instructions equivalent to MIPS core, they are separated by commas. Note that in the "Arithmetic/logical" category all machines but SPARC use separate instruction mnemonics to indicate an immediate operand; SPARC offers immediate versions of these instructions but uses a single mnemonic. (Of course, these are separate opcodes!)

conditional branch. The integer instructions have an option bit that behaves as if the integer op is followed by a compare to zero that sets the first condition "register." PowerPC also lets the second "register" be optionally set by floating-point instructions. PowerPC provides logical operations among these eight 4-bit condition code registers (CRAND, CROR, CRXOR, CRNAND, CRNOR, CREQV), allowing more complex conditions to be tested by a single branch.

MIPS uses the contents of registers to evaluate conditional branches. Any two registers can be compared for equality (BEQ) or inequality (BNE), and then the branch is taken if the condition holds. The set-on-less-than instructions (SLT, SLTI, SLTU, SLTIU) compare two operands and then set the destination register to 1 if less and to 0 otherwise. These instructions are enough to synthesize the full

Control (instruction formats)	B, J/C	B, J/C	B, J/C	B, J/C	B, J/C
Instruction name	**Alpha**	**MIPS64**	**PA-RISC 2.0**	**PowerPC**	**SPARC v.9**
Branch on integer compare	B_ (<, >, <=, >=, =, not=)	BEQ, BNE, B_Z (<, >, <=, >=)	COMB, COMIB	BC	BR_Z, BPcc (<, >, <=, >=, =, not=)
Branch on floating-point compare	FB_(<, >, <=, >=, =, not=)	BC1T, BC1F	FSTWX f0; LDW t; BB t	BC	FBPfcc (<, >, <=, >=, =,...)
Jump, jump register	BR, JMP	J, JR	BL r0, BLR r0	B, BCLR, BCCTR	BA, JMPL r0,...
Call, call register	BSR	JAL, JALR	BL, BLE	BL, BLA, BCLRL, BCCTRL	CALL, JMPL
Trap	CALL_PAL GENTRAP	BREAK	BREAK	TW, TWI	Ticc, SIR
Return from interrupt	CALL_PAL REI	JR; ERET	RFI, RFIR	RFI	DONE, RETRY, RETURN

Figure K.11 Desktop RISC control instructions equivalent to MIPS core. If there are several choices of instructions equivalent to MIPS core, they are separated by commas.

set of relations. Because of the popularity of comparisons to 0, MIPS includes special compare-and-branch instructions for all such comparisons: greater than or equal to zero (BGEZ), greater than zero (BGTZ), less than or equal to zero (BLEZ), and less than zero (BLTZ). Of course, equal and not equal to zero can be synthesized using r0 with BEQ and BNE. Like SPARC, MIPS I uses a condition code for floating point with separate floating-point compare and branch instructions; MIPS IV expanded this to eight floating-point condition codes, with the floating-point comparisons and branch instructions specifying the condition to set or test.

Alpha compares (CMPEQ, CMPLT, CMPLE, CMPULT, CMPULE) test two registers and set a third to 1 if the condition is true and to 0 otherwise. Floating-point compares (CMTEQ, CMTLT, CMTLE, CMTUN) set the result to 2.0 if the condition holds and to 0 otherwise. The branch instructions compare one register to 0 (BEQ, BGE, BGT, BLE, BLT, BNE) or its least-significant bit to 0 (BLBC, BLBS) and then branch if the condition holds.

PA-RISC has many branch options, which we'll see in the section "Instructions Unique to Alpha" on page K-27. The most straightforward is a compare and branch instruction (COMB), which compares two registers, branches depending on the standard relations, and then tests the least-significant bit of the result of the comparison.

ARM is similar to SPARC, in that it provides four traditional condition codes that are optionally set. CMP subtracts one operand from the other and the difference sets the condition codes. Compare negative (CMN) *adds* one operand to the other, and the sum sets the condition codes. TST performs logical AND on the two operands to set all condition codes but overflow, while TEQ uses exclusive

Floating point (instruction formats)	R-R	R-R	R-R	R-R	R-R
Instruction name	**Alpha**	**MIPS64**	**PA-RISC 2.0**	**PowerPC**	**SPARC v.9**
Add single, double	ADDS, ADDT	ADD.S, ADD.D	FADD FADD/dbl	FADDS, FADD	FADDS, FADDD
Subtract single, double	SUBS, SUBT	SUB.S, SUB.D	FSUB FSUB/dbl	FSUBS, FSUB	FSUBS, FSUBD
Multiply single, double	MULS, MULT	MUL.S, MUL.D	FMPY FMPY/dbl	FMULS, FMUL	FMULS, FMULD
Divide single, double	DIVS, DIVT	DIV.S, DIV.D	FDIV, FDIV/dbl	FDIVS, FDIV	FDIVS, FDIVD
Compare	CMPT_ (=, <, <=, UN)	C_.S, C_.D (<, >, <=, >=, =,...)	FCMP, FCMP/dbl (<, =, >)	FCMP	FCMPS, FCMPD
Move R-R	ADDT Fd, F31, Fs	MOV.S, MOV.D	FCPY	FMV	FMOVS/D/Q
Convert (single, double, integer) to (single, double, integer)	CVTST, CVTTS, CVTTQ, CVTQS, CVTQT	CVT.S.D, CVT.D.S, CVT.S.W, CVT.D.W, CVT.W.S, CVT.W.D	FCNVFF,s,d FCNVFF,d,s FCNVXF,s,s FCNVXF,d,d FCNVFX,s,s FCNVFX,d,s	—, FRSP, —, FCTIW, —, —	FSTOD, FDTOS, FSTOI, FDTOI, FITOS, FITOD

Figure K.12 Desktop RISC floating-point instructions equivalent to MIPS core. Dashes mean the operation is not available in that architecture, or not synthesized in a few instructions. If there are several choices of instructions equivalent to MIPS core, they are separated by commas.

Conventions	Alpha	MIPS64	PA-RISC 2.0	PowerPC	SPARC v.9
Register with value 0	r31 (source)	r0	r0	r0 (addressing)	r0
Return address register	(any)	r31	r2, r31	link (special)	r31
No-op	LDQ_U r31,...	SLL r0, r0, r0	OR r0, r0, r0	ORI r0, r0, #0	SETHI r0, 0
Move R-R integer	BIS..., r31,...	ADD..., r0,...	OR..., r0,...	OR rx, ry, ry	OR..., r0,...
Operand order	OP Rs1, Rs2, Rd	OP Rd, Rs1, Rs2	OP Rs1, Rs2, Rd	OP Rd, Rs1, Rs2	OP Rs1, Rs2, Rd

Figure K.13 Conventions of desktop RISC architectures equivalent to MIPS core.

OR to set the first three condition codes. Like SPARC, the conditional version of the ARM branch instruction tests condition codes to determine all possible unsigned and signed relations. As we shall see in the section "Instructions Unique to SPARC v.9" on page K-29, one unusual feature of ARM is that every instruction has the option of executing conditionally depending on the condition

Instruction name	ARM v.4	Thumb	SuperH	M32R	MIPS16
Data transfer (instruction formats)	**DT**	**DT**	**DT**	**DT**	**DT**
Load byte signed	LDRSB	LDRSB	MOV.B	LDB	LB
Load byte unsigned	LDRB	LDRB	MOV.B; EXTU.B	LDUB	LBU
Load half word signed	LDRSH	LDRSH	MOV.W	LDH	LH
Load half word unsigned	LDRH	LDRH	MOV.W; EXTU.W	LDUH	LHU
Load word	LDR	LDR	MOV.L	LD	LW
Store byte	STRB	STRB	MOV.B	STB	SB
Store half word	STRH	STRH	MOV.W	STH	SH
Store word	STR	STR	MOV.L	ST	SW
Read, write special registers	MRS, MSR	—[1]	LDC, STC	MVFC, MVTC	MOVE

Figure K.14 Embedded RISC data transfer instructions equivalent to MIPS core. A sequence of instructions to synthesize a MIPS instruction is shown separated by semicolons. Note that floating point is generally not defined for the embedded RISCs. Thumb and MIPS16 are just 16-bit instruction subsets of the ARM and MIPS architectures, so machines can switch modes and execute the full instruction set. We use —[1] to show sequences that are available in 32-bit mode but not 16-bit mode in Thumb or MIPS16.

codes. (This bears similarities to the annulling option of PA-RISC, seen in the section "Instructions Unique to Alpha" on page K-27.)

Not surprisingly, Thumb follows ARM. Differences are that setting condition codes are not optional, the TEQ instruction is dropped, and there is no conditional execution of instructions.

The Hitachi SuperH uses a single T-bit condition that is set by compare instructions. Two branch instructions decide to branch if either the T bit is 1 (BT) or the T bit is 0 (BF). The two flavors of branches allow fewer comparison instructions.

Mitsubishi M32R also offers a single condition code bit (C) used for signed and unsigned comparisons (CMP, CMPI, CMPU, CMPUI) to see if one register is less than the other or not, similar to the MIPS set-on-less-than instructions. Two branch instructions test to see if the C bit is 1 or 0: BC and BNC. The M32R also includes instructions to branch on equality or inequality of registers (BEQ and BNE) and all relations of a register to 0 (BGEZ, BGTZ, BLEZ, BLTZ, BEQZ, BNEZ). Unlike BC and BNC, these last instructions are all 32 bits wide.

MIPS16 keeps set-on-less-than instructions (SLT, SLTI, SLTU, SLTIU), but instead of putting the result in one of the eight registers, it is placed in a special register named T. MIPS16 is always implemented in machines that also have the full 32-bit MIPS instructions and registers; hence, register T is really register 24

Arithmetic/logical (instruction formats)	R-R, R-I	R-R, R-I	R-R, R-I	R-R, R-I	R-R, R-I
Instruction name	**ARM v.4**	**Thumb**	**SuperH**	**M32R**	**MIPS16**
Add	ADD	ADD	ADD	ADD, ADDI, ADD3	ADDU, ADDIU
Add (trap if overflow)	ADDS; SWIVS	ADD; BVC .+4; SWI	ADDV	ADDV, ADDV3	—[1]
Subtract	SUB	SUB	SUB	SUB	SUBU
Subtract (trap if overflow)	SUBS; SWIVS	SUB; BVC .+1; SWI	SUBV	SUBV	—[1]
Multiply	MUL	MUL	MUL	MUL	MULT, MULTU
Multiply (trap if overflow)					—
Divide	—	—	DIV1, DIVoS, DIVoU	DIV, DIVU	DIV, DIVU
Divide (trap if overflow)	—	—			—
And	AND	AND	AND	AND, AND3	AND
Or	ORR	ORR	OR	OR, OR3	OR
Xor	EOR	EOR	XOR	XOR, XOR3	XOR
Load high part register	—	—		SETH	—[1]
Shift left logical	LSL[3]	LSL[2]	SHLL, SHLLn	SLL, SLLI, SLL[3]	SLLV, SLL
Shift right logical	LSR[3]	LSR[2]	SHRL, SHRLn	SRL, SRLI, SRL[3]	SRLV, SRL
Shift right arithmetic	ASR[3]	ASR[2]	SHRA, SHAD	SRA, SRAI, SRA[3]	SRAV, SRA
Compare	CMP,CMN, TST,TEQ	CMP, CMN, TST	CMP/cond, TST	CMP/I, CMPU/I	CMP/I[2], SLT/I, SLT/IU

Figure K.15 Embedded RISC arithmetic/logical instructions equivalent to MIPS core. Dashes mean the operation is not available in that architecture or not synthesized in a few instructions. Such a sequence of instructions is shown separated by semicolons. If there are several choices of instructions equivalent to MIPS core, they are separated by commas. Thumb and MIPS16 are just 16-bit instruction subsets of the ARM and MIPS architectures, so machines can switch modes and execute the full instruction set. We use —[1] to show sequences that are available in 32-bit mode but not 16-bit mode in Thumb or MIPS16. The superscript 2 indicates new instructions found only in 16-bit mode of Thumb or MIPS16, such as CMP/I[2]. ARM includes shifts as part of every data operation instruction, so the shifts with superscript 3 are just a variation of a move instruction, such as LSR[3].

in the full MIPS architecture. The MIPS16 branch instructions test to see if a register is or is not equal to zero (BEQZ and BNEZ). There are also instructions that branch if register T is or is not equal to zero (BTEQZ and BTNEZ). To test if two registers are equal, MIPS added compare instructions (CMP, CMPI) that compute the exclusive OR of two registers and place the result in register T. Compare was

Control (instruction formats)	B, J, C	B, J, C	B, J, C	B, J, C	B, J, C
Instruction name	**ARM v.4**	**Thumb**	**SuperH**	**M32R**	**MIPS16**
Branch on integer compare	B/cond	B/cond	BF, BT	BEQ, BNE, BC,BNC, B__Z	BEQZ[2], BNEZ[2], BTEQZ[2], BTNEZ[2]
Jump, jump register	MOV pc,ri	MOV pc,ri	BRA, JMP	BRA, JMP	B2, JR
Call, call register	BL	BL	BSR, JSR	BL, JL	JAL, JALR, JALX[2]
Trap	SWI	SWI	TRAPA	TRAP	BREAK
Return from interrupt	MOVS pc, r14	—[1]	RTS	RTE	—[1]

Figure K.16 Embedded RISC control instructions equivalent to MIPS core. Thumb and MIPS16 are just 16-bit instruction subsets of the ARM and MIPS architectures, so machines can switch modes and execute the full instruction set. We use —[1] to show sequences that are available in 32-bit mode but not 16-bit mode in Thumb or MIPS16. The superscript 2 indicates new instructions found only in 16-bit mode of Thumb or MIPS16, such as BTEQZ[2].

Conventions	ARM v.4	Thumb	SuperH	M32R	MIPS16
Return address reg.	R14	R14	PR (special)	R14	RA (special)
No-op	MOV r0,r0	MOV r0,r0	NOP	NOP	SLL r0, r0
Operands, order	OP Rd, Rs1, Rs2	OP Rd, Rs1	OP Rs1, Rd	OP Rd, Rs1	OP Rd, Rs1, Rs2

Figure K.17 Conventions of embedded RISC instructions equivalent to MIPS core.

added since MIPS16 left out instructions to compare and branch if registers are equal or not (BEQ and BNE).

Figures K.18 and K.19 summarize the schemes used for conditional branches.

Instructions: Multimedia Extensions of the Desktop/Server RISCs

Since every desktop microprocessor by definition has its own graphical displays, as transistor budgets increased it was inevitable that support would be added for graphics operations. Many graphics systems use 8 bits to represent each of the three primary colors plus 8 bits for a location of a pixel.

The addition of speakers and microphones for teleconferencing and video games suggested support of sound as well. Audio samples need more than 8 bits of precision, but 16 bits are sufficient.

Every microprocessor has special support so that bytes and half words take up less space when stored in memory, but due to the infrequency of arithmetic operations on these data sizes in typical integer programs, there is little support beyond data transfers. The architects of the Intel i860, which was justified as a graphical accelerator within the company, recognized that many graphics and

	Alpha	MIPS64	PA-RISC 2.0	PowerPC	SPARC v.9
Number of condition code bits (integer and FP)	0	8 FP	8 FP	8 × 4 both	2 × 4 integer, 4 × 2 FP
Basic compare instructions (integer and FP)	1 integer, 1 FP	1 integer, 1 FP	4 integer, 2 FP	4 integer, 2 FP	1 FP
Basic branch instructions (integer and FP)	1	2 integer, 1 FP	7 integer	1 both	3 integer, 1 FP
Compare register with register/ const and branch	—	=, not=	=, not=, <, <=, >, >=, even, odd	—	—
Compare register to zero and branch	=, not=, <, <=, >, >=, even, odd	=, not=, <, <=, >, >=	=, not=, <, <=, >, >=, even, odd	—	=, not=, <, <=, >, >=

Figure K.18 Summary of five desktop RISC approaches to conditional branches. Floating-point branch on PA-RISC is accomplished by copying the FP status register into an integer register and then using the branch on bit instruction to test the FP comparison bit. Integer compare on SPARC is synthesized with an arithmetic instruction that sets the condition codes using `r0` as the destination.

	ARM v.4	Thumb	SuperH	M32R	MIPS16
Number of condition code bits	4	4	1	1	1
Basic compare instructions	4	3	2	2	2
Basic branch instructions	1	1	2	3	2
Compare register with register/const and branch	—	—	=, >, >=	=, not=	—
Compare register to zero and branch	—	—	=, >, >=	=, not=, <, <=, >, >=	=, not=

Figure K.19 Summary of five embedded RISC approaches to conditional branches.

audio applications would perform the same operation on vectors of these data. Although a vector unit was beyond the transistor budget of the i860 in 1989, by partitioning the carry chains within a 64-bit ALU (see Section J.8), it could perform simultaneous operations on short vectors of eight 8-bit operands, four 16-bit operands, or two 32-bit operands. The cost of such partitioned ALUs was small. Applications that lend themselves to such support include MPEG (video), games like DOOM (3D graphics), Adobe Photoshop (digital photography), and teleconferencing (audio and image processing).

Like a virus, over time such multimedia support has spread to nearly every desktop microprocessor. HP was the first successful desktop RISC to include such support. As we shall see, this virus spread unevenly. IBM split multimedia support. The PowerPC offers the active instructions, but the Power version does not.

These extensions have been called *subword parallelism*, *vector*, or *single-instruction, multiple data* (SIMD) (see Appendix A). Since Intel marketing used SIMD to describe the MMX extension of the 80x86, that has become the popular name. Figure K.20 summarizes the support by architecture.

Instruction category	Alpha MAX	MIPS MDMX	PA-RISC MAX2	PowerPC ActiveC	SPARC VIS
Add/subtract		8B, 4H	4H	16B, 8H, 4W	4H, 2W
Saturating add/sub		8B, 4H	4H	16B, 8H, 4W	
Multiply		8B, 4H		16B, 8H, 4W	4B/H
Compare	8B (>=)	8B, 4H (=,<,<=)		16B, 8H, 4W	4H, 2W (=, not=, >, <=)
Shift right/left		8B, 4H	4H	16B, 8H, 4W	
Shift right arithmetic		4H	4H	16B, 8H, 4W	
Multiply and add		8B, 4H		16B, 8H, 4W	
Shift and add (saturating)			4H		
AND/OR/XOR	8B, 4H, 2W	8B, 4H, 2W	8B, 4H, 2W	16B, 8H, 4W	8B, 4H, 2W
Absolute difference	8B				8B
Max/min	8B, 4W	8B, 4H		16B, 8H, 4W	
Pack (2*n* bits --> *n* bits)	2W->2B, 4H->4B	2*2W->4H, 2*4H->8B	2*4H->8B	4W->4H, 8H->8B, 4W->4B	2W->2H, 2W->2B, 4H->4B
Unpack/merge	2B->2W, 4B->4H	2*4B->8B, 2*2H->4H		4H->4W, 8B->8H	4B->4H, 2*4B->8B
Permute/shuffle		8B, 4H	4H	16B, 8H, 4W	
Register sets	Integer	Fl. Pt. + 192b Acc.	Integer	32 ×128b	Fl. Pt.

Figure K.20 Summary of multimedia support for desktop RISCs. B stands for byte (8 bits), H for half word (16 bits), and W for word (32 bits). Thus, 8B means an operation on 8 bytes in a single instruction. Pack and unpack use the notation 2*2W to mean 2 operands each with 2 words. Note that MDMX has vector/scalar operations, where the scalar is specified as an element of one of the vector registers. This table is a simplification of the full multimedia architectures, leaving out many details. For example, MIPS MDMX includes instructions to multiplex between two operands, HP MAX2 includes an instruction to calculate averages, and SPARC VIS includes instructions to set registers to constants. Also, this table does not include the memory alignment operation of MDMX, MAX, and VIS.

From Figure K.20 you can see that in general MIPS MDMX works on 8 bytes or 4 half words per instruction, HP PA-RISC MAX2 works on 4 half words, SPARC VIS works on 4 half words or 2 words, and Alpha doesn't do much. The Alpha MAX operations are just byte versions of compare, min, max, and absolute difference, leaving it up to software to isolate fields and perform parallel adds, subtracts, and multiplies on bytes and half words. MIPS also added operations to work on two 32-bit floating-point operands per cycle, but they are considered part of MIPS V and not simply multimedia extensions (see the section "Instructions Unique to MIPS64" on page K-24).

One feature not generally found in general-purpose microprocessors is saturating operations. Saturation means that when a calculation overflows the result is set to the largest positive number or most negative number, rather than a modulo

calculation as in two's complement arithmetic. Commonly found in digital signal processors (see the next subsection), these saturating operations are helpful in routines for filtering.

These machines largely used existing register sets to hold operands: integer registers for Alpha and HP PA-RISC and floating-point registers for MIPS and Sun. Hence, data transfers are accomplished with standard load and store instructions. PowerPC ActiveC added 32 128-bit registers. MIPS also added a 192-bit (3*64) wide register to act as an accumulator for some operations. By having 3 times the native data width, it can be partitioned to accumulate either 8 bytes with 24 bits per field or 4 half words with 48 bits per field. This wide accumulator can be used for add, subtract, and multiply/add instructions. MIPS claims performance advantages of 2 to 4 times for the accumulator.

Perhaps the surprising conclusion of this table is the lack of consistency. The only operations found on all four are the logical operations (`AND`, `OR`, `XOR`), which do not need a partitioned ALU. If we leave out the frugal Alpha, then the only other common operations are parallel adds and subtracts on 4 half words.

Each manufacturer states that these are instructions intended to be used in hand-optimized subroutine libraries, an intention likely to be followed, as a compiler that works well with all desktop RISC multimedia extensions would be challenging.

Instructions: Digital Signal-Processing Extensions of the Embedded RISCs

One feature found in every digital signal processor (DSP) architecture is support for integer multiply-accumulate. The multiplies tend to be on shorter words than regular integers, such as 16 bits, and the accumulator tends to be on longer words, such as 64 bits. The reason for multiply-accumulate is to efficiently implement digital filters, common in DSP applications. Since Thumb and MIPS16 are subset architectures, they do not provide such support. Instead, programmers should use the DSP or multimedia extensions found in the 32-bit mode instructions of ARM and MIPS64.

Figure K.21 shows the size of the multiply, the size of the accumulator, and the operations and instruction names for the embedded RISCs. Machines with accumulator sizes greater than 32 and less than 64 bits will force the upper bits to remain as the sign bits, thereby "saturating" the add to set to maximum and minimum fixed-point values if the operations overflow.

Instructions: Common Extensions to MIPS Core

Figures K.22 through K.28 list instructions not found in Figures K.9 through K.17 in the same four categories. Instructions are put in these lists if they appear in more than one of the standard architectures. The instructions are defined using the hardware description language defined in Figure K.29.

	ARM v.4	Thumb	SuperH	M32R	MIPS16
Size of multiply	32B × 32B	—	32B × 32B, 16B × 16B	32B × 16B, 16B × 16B	—
Size of accumulator	32B/64B	—	32B/42B, 48B/64B	56B	—
Accumulator name	Any GPR or pairs of GPRs	—	MACH, MACL	ACC	—
Operations	32B/64B product + 64B accumulate signed/unsigned	—	32B product + 42B/32B accumulate (operands in memory); 64B product + 64B/48B accumulate (operands in memory); clear MAC	32B/48B product + 64B accumulate, round, move	—
Corresponding instruction names	MLA, SMLAL, UMLAL	—	MAC, MACS, MAC.L, MAC.LS, CLRMAC	MACHI/MACLO, MACWHI/MACWLO, RAC, RACH, MVFACHI/MVFACLO, MVTACHI/MVTACLO	—

Figure K.21 Summary of five embedded RISC approaches to multiply-accumulate.

Although most of the categories are self-explanatory, a few bear comment:

- The "atomic swap" row means a primitive that can exchange a register with memory without interruption. This is useful for operating system semaphores in a uniprocessor as well as for multiprocessor synchronization (see Section 5.5).
- The 64-bit data transfer and operation rows show how MIPS, PowerPC, and SPARC define 64-bit addressing and integer operations. SPARC simply defines all register and addressing operations to be 64 bits, adding only special instructions for 64-bit shifts, data transfers, and branches. MIPS includes the same extensions, plus it adds separate 64-bit signed arithmetic instructions. PowerPC adds 64-bit right shift, load, store, divide, and compare and has a separate mode determining whether instructions are interpreted as 32- or 64-bit operations; 64-bit operations will not work in a machine that only supports 32-bit mode. PA-RISC is expanded to 64-bit addressing and operations in version 2.0.
- The "prefetch" instruction supplies an address and hint to the implementation about the data. Hints include whether the data are likely to be read or written soon, likely to be read or written only once, or likely to be read or written many times. Prefetch does not cause exceptions. MIPS has a version that adds two registers to get the address for floating-point programs, unlike non-floating-point MIPS programs. (See Chapter 2 to learn more about prefetching.)
- In the "Endian" row, "Big/Little" means there is a bit in the program status register that allows the processor to act either as Big Endian or Little Endian

Name	Definition	Alpha	MIPS64	PA-RISC 2.0	PowerPC	SPARC v.9
Atomic swap R/M (for locks and semaphores)	Temp<---Rd; Rd<---Mem[x]; Mem[x]<---Temp	LDL/Q_L; STL/Q_C	LL; SC	— (see Fig. K.8)	LWARX; STWCX	CASA, CASX
Load 64-bit integer	Rd<---$_{64}$ Mem[x]	LDQ	LD	LDD	LD	LDX
Store 64-bit integer	Mem[x]<---$_{64}$ Rd	STQ	SD	STD	STD	STX
Load 32-bit integer unsigned	$Rd_{32..63}$<---$_{32}$ Mem[x]; $Rd_{0..31}$<---$_{32}$ 0	LDL; EXTLL	LWU	LDW	LWZ	LDUW
Load 32-bit integer signed	$Rd_{32..63}$<---$_{32}$ Mem[x]; $Rd_{0..31}$<---$_{32}$ $Mem[x]_0^{32}$	LDL	LW	LDW; EXTRD,S 63, 8	LWA	LDSW
Prefetch	Cache[x]<---*hint*	FETCH, FETCH_M*	PREF, PREFX	LDD, r0 LDW, r0	DCBT, DCBTST	PRE-FETCH
Load coprocessor	Coprocessor<--- Mem[x]	—	LWCi	CLDWX, CLDWS	—	—
Store coprocessor	Mem[x]<--- Coprocessor	—	SWCi	CSTWX, CSTWS	—	—
Endian	(Big/Little Endian?)	Either	Either	Either	Either	Either
Cache flush	(Flush cache block at this address)	ECB	CPOop	FDC, FIC	DCBF	FLUSH
Shared-memory synchronization	(All prior data transfers complete before next data transfer may start)	WMB	SYNC	SYNC	SYNC	MEMBAR

Figure K.22 Data transfer instructions not found in MIPS core but found in two or more of the five desktop architectures. The load linked/store conditional pair of instructions gives Alpha and MIPS atomic operations for semaphores, allowing data to be read from memory, modified, and stored without fear of interrupts or other machines accessing the data in a multiprocessor (see Chapter 5). Prefetching in the Alpha to external caches is accomplished with FETCH and FETCH_M; on-chip cache prefetches use LD_Q A, R31, and LD_Y A. F31 is used in the Alpha 21164 (see Bhandarkar [1995], p. 190).

(see Section A.3). This can be accomplished by simply complementing some of the least-significant bits of the address in data transfer instructions.

- The "shared-memory synchronization" helps with cache-coherent multiprocessors: All loads and stores executed before the instruction must complete before loads and stores after it can start. (See Chapter 5.)
- The "coprocessor operations" row lists several categories that allow for the processor to be extended with special-purpose hardware.

One difference that needs a longer explanation is the optimized branches. Figure K.30 shows the options. The Alpha and PowerPC offer branches that take effect immediately, like branches on earlier architectures. To accelerate branches, these machines use branch prediction (see Section 3.3). All the rest of the desktop RISCs offer delayed branches (see Appendix C). The embedded RISCs generally do not support delayed branch, with the exception of SuperH, which has it as an option.

Name	Definition	Alpha	MIPS64	PA-RISC 2.0	PowerPC	SPARC v.9
64-bit integer arithmetic ops	Rd<---$_{64}$Rs1 op$_{64}$ Rs2	ADD, SUB, MUL	DADD, DSUB DMULT, DDIV	ADD, SUB, SHLADD, DS	ADD, SUBF, MULLD, DIVD	ADD, SUB, MULX, S/UDIVX
64-bit integer logical ops	Rd<---$_{64}$Rs1 op$_{64}$ Rs2	AND, OR, XOR	AND, OR, XOR	AND, OR, XOR	AND, OR, XOR	AND, OR, XOR
64-bit shifts	Rd<---$_{64}$Rs1 op$_{64}$ Rs2	SLL, SRA, SRL	DSLL/V, DSRA/V, DSRL/V	DEPD,Z EXTRD,S EXTRD,U	SLD, SRAD, SRLD	SLLX, SRAX, SRLX
Conditional move	if (cond) Rd<---Rs	CMOV_	MOVN/Z	SUBc, n; ADD	—	MOVcc, MOVr
Support for multiword integer add	CarryOut, Rd <--- Rs1 + Rs2 + OldCarryOut	—	ADU; SLTU; ADDU, DADU; SLTU; DADDU	ADDC	ADDC, ADDE	ADDcc
Support for multiword integer sub	CarryOut, Rd <--- Rs1 Rs2 + OldCarryOut	—	SUBU; SLTU; SUBU, DSUBU; SLTU; DSUBU	SUBB	SUBFC, SUBFE	SUBcc
And not	Rd <--- Rs1 & ~(Rs2)	BIC	—	ANDCM	ANDC	ANDN
Or not	Rd <--- Rs1 | ~(Rs2)	ORNOT	—	—	ORC	ORN
Add high immediate	Rd$_{0..15}$<---Rs1$_{0..15}$ + (Const<<16);	—	—	ADDIL (R-I)	ADDIS (R-I)	—
Coprocessor operations	(Defined by coprocessor)	—	COPi	COPR,i	—	IMPDEPi

Figure K.23 Arithmetic/logical instructions not found in MIPS core but found in two or more of the five desktop architectures.

Name	Definition	Alpha	MIPS64	PA-RISC 2.0	PowerPC	SPARC v.9
Optimized delayed branches	(Branch not always delayed)	—	BEQL, BNEL, B_ZL (<, >, <=, >=)	COMBT, n, COMBF, n	—	BPcc, A, FPBcc, A
Conditional trap	if (COND) {R31<---PC; PC <---0..0#i}	—	T_,T_I (=, not=, <, >, <=, >=)	SUBc, n; BREAK	TW, TD, TWI, TDI	Tcc
No. control registers	Misc. regs (virtual memory, interrupts, . . .)	6	equiv. 12	32	33	29

Figure K.24 Control instructions not found in MIPS core but found in two or more of the five desktop architectures.

The other three desktop RISCs provide a version of delayed branch that makes it easier to fill the delay slot. The SPARC "annulling" branch executes the instruction in the delay slot only if the branch is taken; otherwise, the instruction is annulled. This means the instruction at the target of the branch can safely be copied into the delay slot since it will only be executed if the branch is taken. The

Name	Definition	Alpha	MIPS64	PA-RISC 2.0	PowerPC	SPARC v.9
Multiply and add	Fd <--- (Fs1 × Fs2) + Fs3	—	MADD.S/D	FMPYFADD sgl/dbl	FMADD/S	
Multiply and sub	Fd <--- (Fs1 × Fs2) – Fs3	—	MSUB.S/D		FMSUB/S	
Neg mult and add	Fd <--- -((Fs1 × Fs2) + Fs3)	—	NMADD.S/D	FMPYFNEG sgl/dbl	FNMADD/S	
Neg mult and sub	Fd <--- -((Fs1 × Fs2) – Fs3)	—	NMSUB.S/D		FNMSUB/S	
Square root	Fd <--- SQRT(Fs)	SQRT_	SQRT.S/D	FSQRT sgl/dbl	FSQRT/S	FSQRTS/D
Conditional move	if (cond) Fd<---Fs	FCMOV_	MOVF/T, MOVF/T.S/D	FTESTFCPY	—	FMOVcc
Negate	Fd <--- Fs ^ x80000000	CPYSN	NEG.S/D	FNEG sgl/dbl	FNEG	FNEGS/D/Q
Absolute value	Fd <--- Fs & x7FFFFFFF	—	ABS.S/D	FABS/dbl	FABS	FABSS/ D/Q

Figure K.25 Floating-point instructions not found in MIPS core but found in two or more of the five desktop architectures.

Name	Definition	ARM v.4	Thumb	SuperH	M32R	MIPS16
Atomic swap R/M (for semaphores)	Temp<---Rd; Rd<---Mem[x]; Mem[x]<---Temp	SWP, SWPB	—[1]	(see TAS)	LOCK; UNLOCK	—[1]
Memory management unit	Paged address translation	Via coprocessor instructions	—[1]	LDTLB		—[1]
Endian	(Big/Little Endian?)	Either	Either	Either	Big	Either

Figure K.26 Data transfer instructions not found in MIPS core but found in two or more of the five embedded architectures. We use —[1] to show sequences that are available in 32-bit mode but not 16-bit mode in Thumb or MIPS16.

restrictions are that the target is not another branch and that the target is known at compile time. (SPARC also offers a nondelayed jump because an unconditional branch with the annul bit set does *not* execute the following instruction.) Later versions of the MIPS architecture have added a branch likely instruction that also annuls the following instruction if the branch is not taken. PA-RISC allows almost any instruction to annul the next instruction, including branches. Its "nullifying" branch option will execute the next instruction depending on the direction of the branch and whether it is taken (i.e., if a forward branch is *not* taken or a backward branch is taken). Presumably this choice was made to optimize loops, allowing the instructions following the exit branch and the looping branch to execute in the common case.

Now that we have covered the similarities, we will focus on the unique features of each architecture. We first cover the desktop/server RISCs, ordering

Name	Definition	ARM v.4	Thumb	SuperH	M32R	MIPS16
Load immediate	Rd<---Imm	MOV	MOV	MOV, MOVA	LDI, LD24	LI
Support for multiword integer add	CarryOut, Rd <--- Rd + Rs1 + OldCarryOut	ADCS	ADC	ADDC	ADDX	—[1]
Support for multiword integer sub	CarryOut, Rd <--- Rd – Rs1 + OldCarryOut	SBCS	SBC	SUBC	SUBX	—[1]
Negate	Rd <--- 0 – Rs1		NEG[2]	NEG	NEG	NEG
Not	Rd <--- ~(Rs1)	MVN	MVN	NOT	NOT	NOT
Move	Rd <--- Rs1	MOV	MOV	MOV	MV	MOVE
Rotate right	Rd <--- Rs i, >> $Rd_{0...i-1}$ <--- $Rs_{31-i...31}$	ROR	ROR	ROTR		
And not	Rd <--- Rs1 & ~(Rs2)	BIC	BIC			

Figure K.27 Arithmetic/logical instructions not found in MIPS core but found in two or more of the five embedded architectures. We use —[1] to show sequences that are available in 32-bit mode but not in 16-bit mode in Thumb or MIPS16. The superscript 2 shows new instructions found only in 16-bit mode of Thumb or MIPS16, such as NEG[2].

Name	Definition	ARM v.4	Thumb	SuperH	M32R	MIPS16
No. control registers	Misc. registers	21	29	9	5	36

Figure K.28 Control information in the five embedded architectures.

them by length of description of the unique features from shortest to longest, and then the embedded RISCs.

Instructions Unique to MIPS64

MIPS has gone through five generations of instruction sets, and this evolution has generally added features found in other architectures. Here are the salient unique features of MIPS, the first several of which were found in the original instruction set.

Nonaligned Data Transfers

MIPS has special instructions to handle misaligned words in memory. A rare event in most programs, it is included for supporting 16-bit minicomputer applications and for doing `memcpy` and `strcpy` faster. Although most RISCs trap if you try to load a word or store a word to a misaligned address, on all architectures misaligned words can be accessed without traps by using four load byte instructions and then assembling the result using shifts and logical ors. The MIPS load and store word left and right instructions (LWL, LWR, SWL,

Notation	Meaning	Example	Meaning
<--	Data transfer. Length of transfer is given by the destination's length; the length is specified when not clear.	`Regs[R1]<--Regs[R2];`	Transfer contents of R2 to R1. Registers have a fixed length, so transfers shorter than the register size must indicate which bits are used.
M	Array of memory accessed in bytes. The starting address for a transfer is indicated as the index to the memory array.	`Regs[R1]<--M[x];`	Place contents of memory location x into R1. If a transfer starts at `M[i]` and requires 4 bytes, the transferred bytes are `M[i]`, `M[i+1]`, `M[i+2]`, and `M[i+3]`.
<--n	Transfer an *n*-bit field, used whenever length of transfer is not clear.	`M[y]<--16M[x];`	Transfer 16 bits starting at memory location x to memory location y. The length of the two sides should match.
X_n	Subscript selects a bit.	`Regs[R1]0<--0;`	Change sign bit of R1 to 0. (Bits are numbered from MSB starting at 0.)
$X_{m..n}$	Subscript selects a field.	`Regs[R3]24..31<--M[x];`	Moves contents of memory location x into low-order byte of R3.
X^n	Superscript replicates a bit field.	`Regs[R3]0..23<--024;`	Sets high-order 3 bytes of R3 to 0.
##	Concatenates two fields.	`Regs[R3]<--024## M[x]; F2##F3<-- 64M[x];`	Moves contents of location x into low byte of R3; clears upper 3 bytes. Moves 64 bits from memory starting at location x; 1st 32 bits go into F2, 2nd 32 into F3.
*, &	Dereference a pointer; get the address of a variable.	`p*<--&x;`	Assign to object pointed to by p the address of the variable x.
<<, >>	C logical shifts (left, right).	`Regs[R1] << 5`	Shift R1 left 5 bits.
==, !=, >, <, >=, <=	C relational operators; equal, not equal, greater, less, greater or equal, less or equal.	`(Regs[R1]== Regs[R2]) & (Regs[R3]!=Regs[R4])`	True if contents of R1 equal the contents of R2 and contents of R3 do not equal the contents of R4.
&, |, ^, !	C bitwise logical operations: AND, OR, XOR, and complement.	`(Regs[R1] & (Regs[R2]| Regs[R3]))`	Bitwise AND of R1 and bitwise OR of R2 and R3.

Figure K.29 Hardware description notation (and some standard C operators).

	(Plain) branch	Delayed branch	Annulling delayed branch	
Found in architectures	Alpha, PowerPC, ARM, Thumb, SuperH, M32R, MIPS16	MIPS64, PA-RISC, SPARC, SuperH	MIPS64, SPARC	PA-RISC
Execute following instruction	Only if branch *not* taken	Always	Only if branch taken	If forward branch *not* taken or backward branch taken

Figure K.30 When the instruction following the branch is executed for three types of branches.

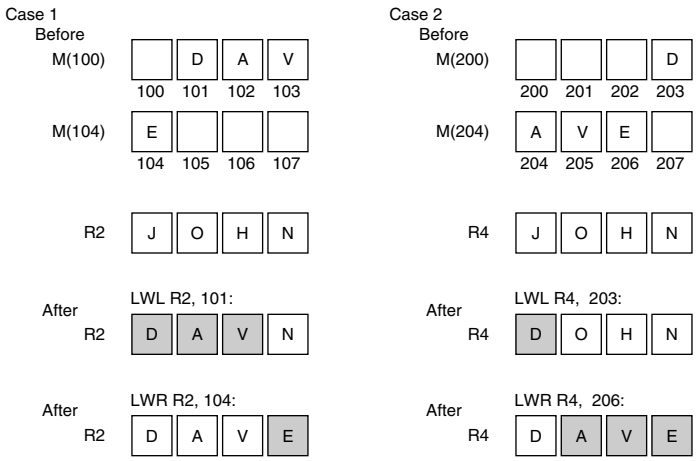

Figure K.31 MIPS instructions for unaligned word reads. This figure assumes operation in Big Endian mode. Case 1 first loads the 3 bytes 101, 102, and 103 into the left of R2, leaving the least-significant byte undisturbed. The following LWR simply loads byte 104 into the least-significant byte of R2, leaving the other bytes of the register unchanged using LWL. Case 2 first loads byte 203 into the most-significant byte of R4, and the following LWR loads the other 3 bytes of R4 from memory bytes 204, 205, and 206. LWL reads the word with the first byte from memory, shifts to the left to discard the unneeded byte(s), and changes only those bytes in Rd. The byte(s) transferred are from the first byte to the lowest-order byte of the word. The following LWR addresses the last byte, right-shifts to discard the unneeded byte(s), and finally changes only those bytes of Rd. The byte(s) transferred are from the last byte up to the highest-order byte of the word. Store word left (SWL) is simply the inverse of LWL, and store word right (SWR) is the inverse of LWR. Changing to Little Endian mode flips which bytes are selected and discarded. (If big-little, left-right, and load-store seem confusing, don't worry; they work!)

SWR) allow this to be done in just two instructions: LWL loads the left portion of the register and LWR loads the right portion of the register. SWL and SWR do the corresponding stores. Figure K.31 shows how they work. There are also 64-bit versions of these instructions.

Remaining Instructions

Below is a list of the remaining unique details of the MIPS64 architecture:

- *NOR*—This logical instruction calculates ~(Rs1 | Rs2).
- *Constant shift amount*—Nonvariable shifts use the 5-bit constant field shown in the register-register format in Figure K.5.

- *SYSCALL*—This special trap instruction is used to invoke the operating system.
- *Move to/from control registers*—`CTCi` and `CFCi` move between the integer registers and control registers.
- *Jump/call not PC-relative*—The 26-bit address of jumps and calls is not added to the PC. It is shifted left 2 bits and replaces the lower 28 bits of the PC. This would only make a difference if the program were located near a 256 MB boundary.
- *TLB instructions*—Translation lookaside buffer (TLB) misses were handled in software in MIPS I, so the instruction set also had instructions for manipulating the registers of the TLB (see Chapter 2 for more on TLBs). These registers are considered part of the "system coprocessor." Since MIPS I the instructions differ among versions of the architecture; they are more part of the implementations than part of the instruction set architecture.
- *Reciprocal and reciprocal square root*—These instructions, which do not follow IEEE 754 guidelines of proper rounding, are included apparently for applications that value speed of divide and square root more than they value accuracy.
- *Conditional procedure call instructions*—`BGEZAL` saves the return address and branches if the content of Rs1 is greater than or equal to zero, and `BLTZAL` does the same for less than zero. The purpose of these instructions is to get a PC-relative call. (There are "likely" versions of these instructions as well.)
- *Parallel single-precision floating-point operations*—As well as extending the architecture with parallel integer operations in MDMX, MIPS64 also supports two parallel 32-bit floating-point operations on 64-bit registers in a single instruction. "Paired single" operations include add (`ADD.PS`), subtract (`SUB.PS`), compare (`C.__.PS`), convert (`CVT.PS.S`, `CVT.S.PL`, `CVT.S.PU`), negate (`NEG.PS`), absolute value (`ABS.PS`), move (`MOV.PS`, `MOVF.PS`, `MOVT.PS`), multiply (`MUL.PS`), multiply-add (`MADD.PS`), and multiply-subtract (`MSUB.PS`).

There is no specific provision in the MIPS architecture for floating-point execution to proceed in parallel with integer execution, but the MIPS implementations of floating point allow this to happen by checking to see if arithmetic interrupts are possible early in the cycle (see Appendix J). Normally, exception detection would force serialization of execution of integer and floating-point operations.

Instructions Unique to Alpha

The Alpha was intended to be an architecture that made it easy to build high-performance implementations. Toward that goal, the architects originally made

two controversial decisions: imprecise floating-point exceptions and no byte or half-word data transfers.

To simplify pipelined execution, Alpha does not require that an exception act as if no instructions past a certain point are executed and that all before that point have been executed. It supplies the TRAPB instruction, which stalls until all prior arithmetic instructions are guaranteed to complete without incurring arithmetic exceptions. In the most conservative mode, placing one TRAPB per exception-causing instruction slows execution by roughly five times but provides precise exceptions (see Darcy and Gay [1996]).

Code that does not include TRAPB does not obey the IEEE 754 floating-point standard. The reason is that parts of the standard (NaNs, infinities, and denormal) are implemented in software on Alpha, as it is on many other microprocessors. To implement these operations in software, however, programs must find the offending instruction and operand values, which cannot be done with imprecise interrupts!

When the architecture was developed, it was believed by the architects that byte loads and stores would slow down data transfers. Byte loads require an extra shifter in the data transfer path, and byte stores require that the memory system perform a read-modify-write for memory systems with error correction codes since the new ECC value must be recalculated. This omission meant that byte stores require the sequence load word, replace desired byte, and then store word. (Inconsistently, floating-point loads go through considerable byte swapping to convert the obtuse VAX floating-point formats into a canonical form.)

To reduce the number of instructions to get the desired data, Alpha includes an elaborate set of byte manipulation instructions: extract field and zero rest of a register (EXTxx), insert field (INSxx), mask rest of a register (MSKxx), zero fields of a register (ZAP), and compare multiple bytes (CMPGE).

Apparently the implementors were not as bothered by load and store byte as were the original architects. Beginning with the shrink of the second version of the Alpha chip (21164A), the architecture *does* include loads and stores for bytes and half words.

Remaining Instructions

Below is a list of the remaining unique instructions of the Alpha architecture:

- *PAL code*—To provide the operations that the VAX performed in microcode, Alpha provides a mode that runs with all privileges enabled, interrupts disabled, and virtual memory mapping turned off for instructions. PAL (*privileged architecture library*) code is used for TLB management, atomic memory operations, and some operating system primitives. PAL code is called via the CALL_PAL instruction.
- *No divide*—Integer divide is not supported in hardware.
- *"Unaligned" load-store*—LDQ_U and STQ_U load and store 64-bit data using addresses that ignore the least-significant three bits. Extract instructions then

select the desired unaligned word using the lower address bits. These instructions are similar to LWL/R,SWL/R in MIPS.

- *Floating-point single precision represented as double precision*—Single-precision data are kept as conventional 32-bit formats in memory but are converted to 64-bit double-precision format in registers.
- *Floating-point register F31 is fixed at zero*—To simplify comparisons to zero.
- *VAX floating-point formats*—To maintain compatibility with the VAX architecture, in addition to the IEEE 754 single- and double-precision formats called S and T, Alpha supports the VAX single- and double-precision formats called F and G, but not VAX format D. (D had too narrow an exponent field to be useful for double precision and was replaced by G in VAX code.)
- *Bit count instructions*—Version 3 of the architecture added instructions to count the number of leading zeros (CTLZ), count the number of trailing zeros (CTTZ), and count the number of ones in a word (CTPOP). Originally found on Cray computers, these instructions help with decryption.

Instructions Unique to SPARC v.9

Several features are unique to SPARC.

Register Windows

The primary unique feature of SPARC is register windows, an optimization for reducing register traffic on procedure calls. Several banks of registers are used, with a new one allocated on each procedure call. Although this could limit the depth of procedure calls, the limitation is avoided by operating the banks as a circular buffer, providing unlimited depth. The knee of the cost-performance curve seems to be six to eight banks.

SPARC can have between 2 and 32 windows, typically using 8 registers each for the globals, locals, incoming parameters, and outgoing parameters. (Given that each window has 16 unique registers, an implementation of SPARC can have as few as 40 physical registers and as many as 520, although most have 128 to 136, so far.) Rather than tie window changes with call and return instructions, SPARC has the separate instructions SAVE and RESTORE. SAVE is used to "save" the caller's window by pointing to the next window of registers in addition to performing an add instruction. The trick is that the source registers are from the caller's window of the addition operation, while the destination register is in the callee's window. SPARC compilers typically use this instruction for changing the stack pointer to allocate local variables in a new stack frame. RESTORE is the inverse of SAVE, bringing back the caller's window while acting as an add instruction, with the source registers from the callee's window and the destination register in the caller's window. This automatically deallocates the stack frame. Compilers can also make use of it for generating the callee's final return value.

The danger of register windows is that the larger number of registers could slow down the clock rate. This was not the case for early implementations. The SPARC architecture (with register windows) and the MIPS R2000 architecture (without) have been built in several technologies since 1987. For several generations the SPARC clock rate has not been slower than the MIPS clock rate for implementations in similar technologies, probably because cache access times dominate register access times in these implementations. The current-generation machines took different implementation strategies—in order versus out of order—and it's unlikely that the number of registers by themselves determined the clock rate in either machine. Recently, other architectures have included register windows: Tensilica and IA-64.

Another data transfer feature is alternate space option for loads and stores. This simply allows the memory system to identify memory accesses to input/output devices, or to control registers for devices such as the cache and memory management unit.

Fast Traps

Version 9 SPARC includes support to make traps fast. It expands the single level of traps to at least four levels, allowing the window overflow and underflow trap handlers to be interrupted. The extra levels mean the handler does not need to check for page faults or misaligned stack pointers explicitly in the code, thereby making the handler faster. Two new instructions were added to return from this multilevel handler: `RETRY` (which retries the interrupted instruction) and `DONE` (which does not). To support user-level traps, the instruction `RETURN` will return from the trap in nonprivileged mode.

Support for LISP and Smalltalk

The primary remaining arithmetic feature is tagged addition and subtraction. The designers of SPARC spent some time thinking about languages like LISP and Smalltalk, and this influenced some of the features of SPARC already discussed: register windows, conditional trap instructions, calls with 32-bit instruction addresses, and multiword arithmetic (see Taylor et al. [1986] and Ungar et al. [1984]). A small amount of support is offered for tagged data types with operations for addition, subtraction, and hence comparison. The two least-significant bits indicate whether the operand is an integer (coded as 00), so `TADDcc` and `TSUBcc` set the overflow bit if either operand is not tagged as an integer or if the result is too large. A subsequent conditional branch or trap instruction can decide what to do. (If the operands are not integers, software recovers the operands, checks the types of the operands, and invokes the correct operation based on those types.) It turns out that the misaligned memory access trap can also be put to use for tagged data, since loading from a pointer with the wrong tag can be an invalid access. Figure K.32 shows both types of tag support.

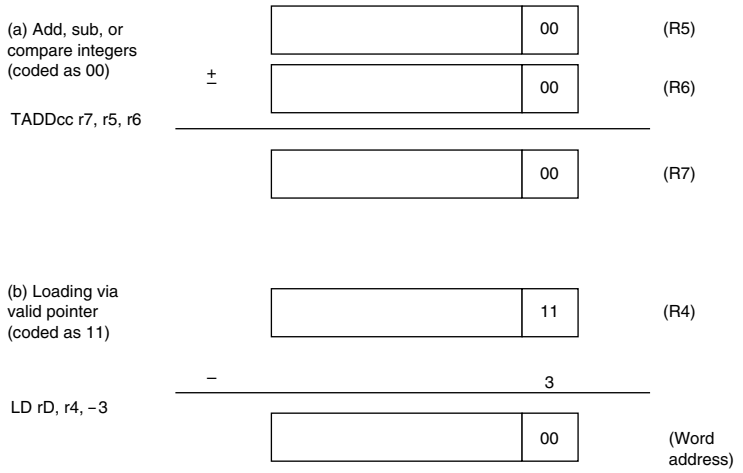

Figure K.32 SPARC uses the two least-significant bits to encode different data types for the tagged arithmetic instructions. (a) Integer arithmetic, which takes a single cycle as long as the operands and the result are integers. (b) The misaligned trap can be used to catch invalid memory accesses, such as trying to use an integer as a pointer. For languages with paired data like LISP, an offset of –3 can be used to access the even word of a pair (CAR) and +1 can be used for the odd word of a pair (CDR).

Overlapped Integer and Floating-Point Operations

SPARC allows floating-point instructions to overlap execution with integer instructions. To recover from an interrupt during such a situation, SPARC has a queue of pending floating-point instructions and their addresses. RDPR allows the processor to empty the queue. The second floating-point feature is the inclusion of floating-point square root instructions FSQRTS, FSQRTD, and FSQRTQ.

Remaining Instructions

The remaining unique features of SPARC are as follows:

- JMPL uses Rd to specify the return address register, so specifying r31 makes it similar to JALR in MIPS and specifying r0 makes it like JR.
- LDSTUB loads the value of the byte into Rd and then stores FF16 into the addressed byte. This version 8 instruction can be used to implement a semaphore (see Chapter 5).
- CASA (CASXA) atomically compares a value in a processor register to a 32-bit (64-bit) value in memory; if and only if they are equal, it swaps the value in memory with the value in a second processor register. This version 9 instruction can be used to construct wait-free synchronization algorithms that do not require the use of locks.

- `XNOR` calculates the exclusive OR with the complement of the second operand.
- `BPcc`, `BPr`, and `FBPcc` include a branch-prediction bit so that the compiler can give hints to the machine about whether a branch is likely to be taken or not.
- `ILLTRAP` causes an illegal instruction trap. Muchnick [1988] explained how this is used for proper execution of aggregate returning procedures in C.
- `POPC` counts the number of bits set to one in an operand, also found in the third version of the Alpha architecture.
- *Nonfaulting loads* allow compilers to move load instructions ahead of conditional control structures that control their use. Hence, nonfaulting loads will be executed speculatively.
- *Quadruple-precision floating-point arithmetic and data transfer* allow the floating-point registers to act as eight 128-bit registers for floating-point operations and data transfers.
- *Multiple-precision floating-point results for multiply* mean that two single-precision operands can result in a double-precision product and two double-precision operands can result in a quadruple-precision product. These instructions can be useful in complex arithmetic and some models of floating-point calculations.

Instructions Unique to PowerPC

PowerPC is the result of several generations of IBM commercial RISC machines—IBM RT/PC, IBM Power1, and IBM Power2—plus the Motorola 88x00.

Branch Registers: Link and Counter

Rather than dedicate one of the 32 general-purpose registers to save the return address on procedure call, PowerPC puts the address into a special register called the *link register.* Since many procedures will return without calling another procedure, link doesn't always have to be saved away. Making the return address a special register makes the return jump faster since the hardware need not go through the register read pipeline stage for return jumps.

In a similar vein, PowerPC has a *count register* to be used in for loops where the program iterates for a fixed number of times. By using a special register the branch hardware can determine quickly whether a branch based on the count register is likely to branch, since the value of the register is known early in the execution cycle. Tests of the value of the count register in a branch instruction will automatically decrement the count register.

Given that the count register and link register are already located with the hardware that controls branches, and that one of the problems in branch prediction is getting the target address early in the pipeline (see Appendix C), the

PowerPC architects decided to make a second use of these registers. Either register can hold a target address of a conditional branch. Thus, PowerPC supplements its basic conditional branch with two instructions that get the target address from these registers (`BCLR`, `BCCTR`).

Remaining Instructions

Unlike most other RISC machines, register 0 is not hardwired to the value 0. It cannot be used as a base register—that is, it generates a 0 in this case—but in base + index addressing it can be used as the index. The other unique features of the PowerPC are as follows:

- *Load multiple* and *store multiple* save or restore up to 32 registers in a single instruction.
- `LSW` and `STSW` permit fetching and storing of fixed- and variable-length strings that have arbitrary alignment.
- *Rotate with mask* instructions support bit field extraction and insertion. One version rotates the data and then performs logical AND with a mask of ones, thereby extracting a field. The other version rotates the data but only places the bits into the destination register where there is a corresponding 1 bit in the mask, thereby inserting a field.
- *Algebraic right shift* sets the carry bit (`CA`) if the operand is negative and any 1 bits are shifted out. Thus, a signed divide by any constant power of 2 that rounds toward 0 can be accomplished with a `SRAWI` followed by `ADDZE`, which adds `CA` to the register.
- `CBTLZ` will count leading zeros.
- `SUBFIC` computes (immediate – RA), which can be used to develop a one's or two's complement.
- *Logical shifted immediate* instructions shift the 16-bit immediate to the left 16 bits before performing AND, OR, or XOR.

Instructions Unique to PA-RISC 2.0

PA-RISC was expanded slightly in 1990 with version 1.1 and changed significantly in 2.0 with 64-bit extensions in 1996. PA-RISC has perhaps the most unusual features of any desktop RISC machine. For example, it has the most addressing modes and instruction formats and, as we shall see, several instructions that are really the combination of two simpler instructions.

Nullification

As shown in Figure K.30, several RISC machines can choose to not execute the instruction following a delayed branch in order to improve utilization of the

branch slot. This is called *nullification* in PA-RISC, and it has been generalized to apply to any arithmetic/logical instruction as well as to all branches. Thus, an add instruction can add two operands, store the sum, and cause the following instruction to be skipped if the sum is zero. Like conditional move instructions, nullification allows PA-RISC to avoid branches in cases where there is just one instruction in the then part of an if statement.

A Cornucopia of Conditional Branches

Given nullification, PA-RISC did not need to have separate conditional branch instructions. The inventors could have recommended that nullifying instructions precede unconditional branches, thereby simplifying the instruction set. Instead, PA-RISC has the largest number of conditional branches of any RISC machine. Figure K.33 shows the conditional branches of PA-RISC. As you can see, several are really combinations of two instructions.

Synthesized Multiply and Divide

PA-RISC provides several primitives so that multiply and divide can be synthesized in software. Instructions that shift one operand 1, 2, or 3 bits and then add, trapping or not on overflow, are useful in multiplies. (Alpha also includes instructions that multiply the second operand of adds and subtracts by 4 or by 8: `S4ADD`, `S8ADD`, `S4SUB`, and `S8SUB`.) Divide step performs the critical step of nonrestoring

Name	Instruction	Notation	
`COMB`	Compare and branch	`if (cond(Rs1,Rs2))`	`{PC <--- PC + offset12}`
`COMIB`	Compare imm. and branch	`if (cond(imm5,Rs2))`	`{PC <--- PC + offset12}`
`MOVB`	Move and branch	`Rs2 <--- Rs1,` `if (cond(Rs1,0))`	`{PC <--- PC + offset12}`
`MOVIB`	Move immediate and branch	`Rs2 <--- imm5,` `if (cond(imm5,0))`	`{PC <--- PC + offset12}`
`ADDB`	Add and branch	`Rs2 <--- Rs1 + Rs2,` `if (cond(Rs1 + Rs2,0))`	`{PC <--- PC + offset12}`
`ADDIB`	Add imm. and branch	`Rs2 <--- imm5 + Rs2,` `if (cond(imm5 + Rs2,0))`	`{PC <--- PC + offset12}`
`BB`	Branch on bit	`if (cond(Rsp,0))`	`{PC <--- PC + offset12}`
`BVB`	Branch on variable bit	`if (cond(Rssar,0))`	`{PC <--- PC + offset12}`

Figure K.33 The PA-RISC conditional branch instructions. The 12-bit offset is called `offset12` in this table, and the 5-bit immediate is called `imm5`. The 16 conditions are =, <, <=; odd; signed overflow; unsigned no overflow; zero or no overflow unsigned; never; and their respective complements. The `BB` instruction selects one of the 32 bits of the register and branches depending if its value is 0 or 1. The `BVB` selects the bit to branch using the shift amount register, a special-purpose register. The subscript notation specifies a bit field.

divide, adding, or subtracting depending on the sign of the prior result. Magenheimer et al. [1988] measured the size of operands in multiplies and divides to show how well the multiply step would work. Using these data for C programs, Muchnick [1988] found that by making special cases the average multiply by a constant takes 6 clock cycles and multiply of variables takes 24 clock cycles. PA-RISC has 10 instructions for these operations.

The original SPARC architecture used similar optimizations, but with increasing numbers of transistors the instruction set was expanded to include full multiply and divide operations. PA-RISC gives some support along these lines by putting a full 32-bit integer multiply in the floating-point unit; however, the integer data must first be moved to floating-point registers.

Decimal Operations

COBOL programs will compute on decimal values, stored as 4 bits per digit, rather than converting back and forth between binary and decimal. PA-RISC has instructions that will convert the sum from a normal 32-bit add into proper decimal digits. It also provides logical and arithmetic operations that set the condition codes to test for carries of digits, bytes, or half words. These operations also test whether bytes or half words are zero. These operations would be useful in arithmetic on 8-bit ASCII characters. Five PA-RISC instructions provide decimal support.

Remaining Instructions

Here are some remaining PA-RISC instructions:

- *Branch vectored* shifts an index register left 3 bits, adds it to a base register, and then branches to the calculated address. It is used for case statements.
- *Extract* and *deposit* instructions allow arbitrary bit fields to be selected from or inserted into registers. Variations include whether the extracted field is sign-extended, whether the bit field is specified directly in the instruction or indirectly in another register, and whether the rest of the register is set to zero or left unchanged. PA-RISC has 12 such instructions.
- To simplify use of 32-bit address constants, PA-RISC includes `ADDIL`, which adds a left-adjusted 21-bit constant to a register and places the result in register 1. The following data transfer instruction uses offset addressing to add the lower 11 bits of the address to register 1. This pair of instructions allows PA-RISC to add a 32-bit constant to a base register, at the cost of changing register 1.
- PA-RISC has nine debug instructions that can set breakpoints on instruction or data addresses and return the trapped addresses.
- *Load* and *clear* instructions provide a semaphore or lock that reads a value from memory and then writes zero.

- *Store bytes short* optimizes unaligned data moves, moving either the leftmost or the rightmost bytes in a word to the effective address, depending on the instruction options and condition code bits.
- Loads and stores work well with caches by having options that give hints about whether to load data into the cache if it's not already in the cache. For example, load with a destination of register 0 is defined to be software-controlled cache prefetch.
- PA-RISC 2.0 extended cache hints to stores to indicate block copies, recommending that the processor not load data into the cache if it's not already in the cache. It also can suggest that on loads and stores there is spatial locality to prepare the cache for subsequent sequential accesses.
- PA-RISC 2.0 also provides an optional branch-target stack to predict indirect jumps used on subroutine returns. Software can suggest which addresses get placed on and removed from the branch-target stack, but hardware controls whether or not these are valid.
- *Multiply/add* and *multiply/subtract* are floating-point operations that can launch two independent floating-point operations in a single instruction in addition to the fused multiply/add and fused multiply/negate/add introduced in version 2.0 of PA-RISC.

Instructions Unique to ARM

It's hard to pick the most unusual feature of ARM, but perhaps it is conditional execution of instructions. Every instruction starts with a 4-bit field that determines whether it will act as a NOP or as a real instruction, depending on the condition codes. Hence, conditional branches are properly considered as conditionally executing the unconditional branch instruction. Conditional execution allows avoiding a branch to jump over a single instruction. It takes less code space and time to simply conditionally execute one instruction.

The 12-bit immediate field has a novel interpretation. The 8 least-significant bits are zero-extended to a 32-bit value, then rotated right the number of bits specified in the first 4 bits of the field multiplied by 2. Whether this split actually catches more immediates than a simple 12-bit field would be an interesting study. One advantage is that this scheme can represent all powers of 2 in a 32-bit word.

Operand shifting is not limited to immediates. The second register of all arithmetic and logical processing operations has the option of being shifted before being operated on. The shift options are shift left logical, shift right logical, shift right arithmetic, and rotate right. Once again, it would be interesting to see how often operations like rotate-and-add, shift-right-and-test, and so on occur in ARM programs.

Remaining Instructions

Below is a list of the remaining unique instructions of the ARM architecture:

- *Block loads and stores*—Under control of a 16-bit mask within the instructions, any of the 16 registers can be loaded or stored into memory in a single instruction. These instructions can save and restore registers on procedure entry and return. These instructions can also be used for block memory copy—offering up to four times the bandwidth of a single register load-store—and today block copies are the most important use.
- *Reverse subtract*—`RSB` allows the first register to be subtracted from the immediate or shifted register. `RSC` does the same thing but includes the carry when calculating the difference.
- *Long multiplies*—Similar to MIPS, Hi and Lo registers get the 64-bit signed product (`SMULL`) or the 64-bit unsigned product (`UMULL`).
- *No divide*—Like the Alpha, integer divide is not supported in hardware.
- *Conditional trap*—A common extension to the MIPS core found in desktop RISCs (Figures K.22 through K.25), it comes for free in the conditional execution of all ARM instructions, including `SWI`.
- *Coprocessor interface*—Like many of the desktop RISCs, ARM defines a full set of coprocessor instructions: data transfer, moves between general-purpose and coprocessor registers, and coprocessor operations.
- *Floating-point architecture*—Using the coprocessor interface, a floating-point architecture has been defined for ARM. It was implemented as the FPA10 coprocessor.
- *Branch and exchange instruction sets*—The `BX` instruction is the transition between ARM and Thumb, using the lower 31 bits of the register to set the PC and the most-significant bit to determine if the mode is ARM (1) or Thumb (0).

Instructions Unique to Thumb

In the ARM version 4 model, frequently executed procedures will use ARM instructions to get maximum performance, with the less frequently executed ones using Thumb to reduce the overall code size of the program. Since typically only a few procedures dominate execution time, the hope is that this hybrid gets the best of both worlds.

Although Thumb instructions are translated by the hardware into conventional ARM instructions for execution, there are several restrictions. First, conditional execution is dropped from almost all instructions. Second, only the first 8 registers are easily available in all instructions, with the stack pointer, link register, and program counter being used implicitly in some instructions. Third, Thumb uses a two-operand format to save space. Fourth, the unique shifted immediates and

shifted second operands have disappeared and are replaced by separate shift instructions. Fifth, the addressing modes are simplified. Finally, putting all instructions into 16 bits forces many more instruction formats.

In many ways the simplified Thumb architecture is more conventional than ARM. Here are additional changes made from ARM in going to Thumb:

- *Drop of immediate logical instructions*—Logical immediates are gone.
- *Condition codes implicit*—Rather than have condition codes set optionally, they are defined by the opcode. All ALU instructions and none of the data transfers set the condition codes.
- *Hi/Lo register access*—The 16 ARM registers are halved into Lo registers and Hi registers, with the 8 Hi registers including the stack pointer (SP), link register, and PC. The Lo registers are available in all ALU operations. Variations of `ADD`, `BX`, `CMP`, and `MOV` also work with all combinations of Lo and Hi registers. SP and PC registers are also available in variations of data transfers and add immediates. Any other operations on the Hi registers require one `MOV` to put the value into a Lo register, perform the operation there, and then transfer the data back to the Hi register.
- *Branch/call distance*—Since instructions are 16 bits wide, the 8-bit conditional branch address is shifted by 1 instead of by 2. Branch with link is specified in two instructions, concatenating 11 bits from each instruction and shifting them left to form a 23-bit address to load into PC.
- *Distance for data transfer offsets*—The offset is now 5 bits for the general-purpose registers and 8 bits for SP and PC.

Instructions Unique to SuperH

Register 0 plays a special role in SuperH address modes. It can be added to another register to form an address in indirect indexed addressing and PC-relative addressing. R0 is used to load constants to give a larger addressing range than can easily be fit into the 16-bit instructions of the SuperH. R0 is also the only register that can be an operand for immediate versions of `AND`, `CMP`, `OR`, and `XOR`.

Below is a list of the remaining unique details of the SuperH architecture:

- *Decrement and test*—`DT` decrements a register and sets the T bit to 1 if the result is 0.
- *Optional delayed branch*—Although the other embedded RISC machines generally do not use delayed branches (see Appendix C), SuperH offers optional delayed branch execution for `BT` and `BF`.
- *Many multiplies*—Depending on if the operation is signed or unsigned, if the operands are 16 bits or 32 bits, or if the product is 32 bits or 64 bits, the proper multiply instruction is `MULS`, `MULU`, `DMULS`, `DMULU`, or `MUL`. The product is found in the MACL and MACH registers.

- *Zero and sign extension*—Bytes or half words are either zero-extended (EXTU) or sign-extended (EXTS) within a 32-bit register.
- *One-bit shift amounts*—Perhaps in an attempt to make them fit within the 16-bit instructions, shift instructions only shift a single bit at a time.
- *Dynamic shift amount*—These variable shifts test the sign of the amount in a register to determine whether they shift left (positive) or shift right (negative). Both logical (SHLD) and arithmetic (SHAD) instructions are supported. These instructions help offset the 1-bit constant shift amounts of standard shifts.
- *Rotate*—SuperH offers rotations by 1 bit left (ROTL) and right (ROTR), which set the T bit with the value rotated, and also have variations that include the T bit in the rotations (ROTCL and ROTCR).
- SWAP—This instruction swaps either the high and low bytes of a 32-bit word or the two bytes of the rightmost 16 bits.
- *Extract word (*XTRCT*)*—The middle 32 bits from a pair of 32-bit registers are placed in another register.
- *Negate with carry*—Like SUBC (Figure K.27), except the first operand is 0.
- *Cache prefetch*—Like many of the desktop RISCs (Figures K.22 through K.25), SuperH has an instruction (PREF) to prefetch data into the cache.
- *Test-and-set*—SuperH uses the older test-and-set (TAS) instruction to perform atomic locks or semaphores (see Chapter 5). TAS first loads a byte from memory. It then sets the T bit to 1 if the byte is 0 or to 0 if the byte is not 0. Finally, it sets the most-significant bit of the byte to 1 and writes the result back to memory.

Instructions Unique to M32R

The most unusual feature of the M32R is a slight very long instruction word (VLIW) approach to the pairs of 16-bit instructions. A bit is reserved in the first instruction of the pair to say whether this instruction can be executed in parallel with the next instruction—that is, the two instructions are independent—or if these two must be executed sequentially. (An earlier machine that offered a similar option was the Intel i860.) This feature is included for future implementations of the architecture.

One surprise is that all branch displacements are shifted left 2 bits before being added to the PC, and the lower 2 bits of the PC are set to 0. Since some instructions are only 16 bits long, this shift means that a branch cannot go to any instruction in the program: It can only branch to instructions on word boundaries. A similar restriction is placed on the return address for the branch-and-link and jump-and-link instructions: They can only return to a word boundary. Thus, for a slightly larger branch distance, software must ensure that all branch addresses and all return addresses are aligned to a word boundary. The M32R code space is probably slightly larger, and it probably executes more NOP instructions than it would if the branch address were only shifted left 1 bit.

However, the VLIW feature above means that a NOP can execute in parallel with another 16-bit instruction, so that the padding doesn't take more clock cycles. The code size expansion depends on the ability of the compiler to schedule code and to pair successive 16-bit instructions; Mitsubishi claims that code size overall is only 7% larger than that for the Motorola 680x0 architecture.

The last remaining novel feature is that the result of the divide operation is the remainder instead of the quotient.

Instructions Unique to MIPS16

MIPS16 is not really a separate instruction set but a 16-bit extension of the full 32-bit MIPS architecture. It is compatible with any of the 32-bit address MIPS architectures (MIPS I, MIPS II) or 64-bit architectures (MIPS III, IV, V). The ISA mode bit determines the width of instructions: 0 means 32-bit-wide instructions and 1 means 16-bit-wide instructions. The new `JALX` instruction toggles the ISA mode bit to switch to the other ISA. `JR` and `JALR` have been redefined to set the ISA mode bit from the most-significant bit of the register containing the branch address, and this bit is not considered part of the address. All jump and link instructions save the current mode bit as the most-significant bit of the return address.

Hence MIPS supports whole procedures containing either 16-bit or 32-bit instructions, but it does not support mixing the two lengths together in a single procedure. The one exception is the `JAL` and `JALX`: These two instructions need 32 bits even in the 16-bit mode, presumably to get a large enough address to branch to far procedures.

In picking this subset, MIPS decided to include opcodes for some three-operand instructions and to keep 16 opcodes for 64-bit operations. The combination of this many opcodes and operands in 16 bits led the architects to provide only 8 easy-to-use registers—just like Thumb—whereas the other embedded RISCs offer about 16 registers. Since the hardware must include the full 32 registers of the 32-bit ISA mode, MIPS16 includes move instructions to copy values between the 8 MIPS16 registers and the remaining 24 registers of the full MIPS architecture. To reduce pressure on the 8 visible registers, the stack pointer is considered a separate register. MIPS16 includes a variety of separate opcodes to do data transfers using SP as a base register and to increment SP: `LWSP`, `LDSP`, `SWSP`, `SDSP`, `ADJSP`, `DADJSP`, `ADDIUSPD`, and `DADDIUSP`.

To fit within the 16-bit limit, immediate fields have generally been shortened to 5 to 8 bits. MIPS16 provides a way to extend its shorter immediates into the full width of immediates in the 32-bit mode. Borrowing a trick from the Intel 8086, the `EXTEND` instruction is really a 16-bit prefix that can be prepended to any MIPS16 instruction with an address or immediate field. The prefix supplies enough bits to turn the 5-bit fields of data transfers and 5- to 8-bit fields of arithmetic immediates into 16-bit constants. Alas, there are two exceptions. `ADDIU` and `DADDIU` start with 4-bit immediate fields, but since `EXTEND` can only supply 11 more bits, the wider immediate is limited to 15 bits. `EXTEND` also extends the

3-bit shift fields into 5-bit fields for shifts. (In case you were wondering, the EXTEND prefix does *not* need to start on a 32-bit boundary.)

To further address the supply of constants, MIPS16 added a new addressing mode! PC-relative addressing for load word (LWPC) and load double (LDPC) shifts an 8-bit immediate field by 2 or 3 bits, respectively, adding it to the PC with the lower 2 or 3 bits cleared. The constant word or double word is then loaded into a register. Thus 32-bit or 64-bit constants can be included with MIPS16 code, despite the loss of LIU to set the upper register bits. Given the new addressing mode, there is also an instruction (ADDIUPC) to calculate a PC-relative address and place it in a register.

MIPS16 differs from the other embedded RISCs in that it can subset a 64-bit address architecture. As a result it has 16-bit instruction-length versions of 64-bit data operations: data transfer (LD, SD, LWU), arithmetic operations (DADDU/IU, DSUBU, DMULT/U, DDIV/U), and shifts (DSLL/V, DSRA/V, DSRL/V).

Since MIPS plays such a prominent role in this book, we show all the additional changes made from the MIPS core instructions in going to MIPS16:

- *Drop of signed arithmetic instructions*—Arithmetic instructions that can trap were dropped to save opcode space: ADD, ADDI, SUB, DADD, DADDI, DSUB.
- *Drop of immediate logical instructions*—Logical immediates are gone, too: ANDI, ORI, XORI.
- *Branch instructions pared down*—Comparing two registers and then branching did not fit, nor did all the other comparisons of a register to zero. Hence, these instructions didn't make it either: BEQ, BNE, BGEZ, BGTZ, BLEZ, and BLTZ. As mentioned in the section "Instructions: The MIPS Core Subset" on page K-6, to help compensate MIPS16 includes compare instructions to test if two registers are equal. Since compare and set-on-less-than set the new T register, branches were added to test the T register.
- *Branch distance*—Since instructions are 16 bits wide, the branch address is shifted by one instead of by two.
- *Delayed branches disappear*—The branches take effect before the next instruction. Jumps still have a one-slot delay.
- *Extension and distance for data transfer offsets*—The 5-bit and 8-bit fields are zero-extended instead of sign-extended in 32-bit mode. To get greater range, the immediate fields are shifted left 1, 2, or 3 bits depending on whether the data are half word, word, or double word. If the EXTEND prefix is prepended to these instructions, they use the conventional signed 16-bit immediate of the 32-bit mode.
- *Extension of arithmetic immediates*—The 5-bit and 8-bit fields are zero-extended for set-on-less-than and compare instructions, for forming a PC-relative address, and for adding to SP and placing the result in a register (ADDIUSP, DADDIUSP). Once again, if the EXTEND prefix is prepended to these instructions, they use the conventional signed 16-bit immediate of the 32-bit

mode. They are still sign-extended for general adds and for adding to SP and placing the result back in SP (`ADJSP`, `DADJSP`). Alas, code density and orthogonality are strange bedfellows in MIPS16!

- *Redefining shift amount of 0*—MIPS16 defines the value 0 in the 3-bit shift field to mean a shift of 8 bits.
- *New instructions added due to loss of register 0 as zero*—Load immediate, negate, and not were added, since these operations could no longer be synthesized from other instructions using `r0` as a source.

Concluding Remarks

This survey covers the addressing modes, instruction formats, and all instructions found in 10 RISC architectures. Although the later sections concentrate on the differences, it would not be possible to cover 10 architectures in these few pages if there were not so many similarities. In fact, we would guess that more than 90% of the instructions executed for any of these architectures would be found in Figures K.9 through K.17. To contrast this homogeneity, Figure K.34 gives a summary for four architectures from the 1970s in a format similar to that shown in Figure K.1. (Since it would be impossible to write a single section in this style for those architectures, the next three sections cover the 80x86, VAX, and IBM 360/370.) In the history of computing, there has never been such widespread agreement on computer architecture.

	IBM 360/370	Intel 8086	Motorola 68000	DEC VAX
Date announced	1964/1970	1978	1980	1977
Instruction size(s) (bits)	16, 32, 48	8, 16, 24, 32, 40, 48	16, 32, 48, 64, 80	8, 16, 24, 32, ... , 432
Addressing (size, model)	24 bits, flat/ 31 bits, flat	4 + 16 bits, segmented	24 bits, flat	32 bits, flat
Data aligned?	Yes 360/No 370	No	16-bit aligned	No
Data addressing modes	2/3	5	9	=14
Protection	Page	None	Optional	Page
Page size	2 KB & 4 KB	—	0.25 to 32 KB	0.5 KB
I/O	Opcode	Opcode	Memory mapped	Memory mapped
Integer registers (size, model, number)	16 GPR × 32 bits	8 dedicated data × 16 bits	8 data and 8 address × 32 bits	15 GPR × 32 bits
Separate floating-point registers	4 × 64 bits	Optional: 8 × 80 bits	Optional: 8 × 80 bits	0
Floating-point format	IBM (floating hexadecimal)	IEEE 754 single, double, extended	IEEE 754 single, double, extended	DEC

Figure K.34 Summary of four 1970s architectures. Unlike the architectures in Figure K.1, there is little agreement between these architectures in any category. (See Section K.3 for more details on the 80x86 and Section K.4 for a description of the VAX.)

This style of architecture cannot remain static, however. Like people, instruction sets tend to get bigger as they get older. Figure K.35 shows the genealogy of these instruction sets, and Figure K.36 shows which features were added to or deleted from generations of desktop RISCs over time.

As you can see, all the desktop RISC machines have evolved to 64-bit address architectures, and they have done so fairly painlessly.

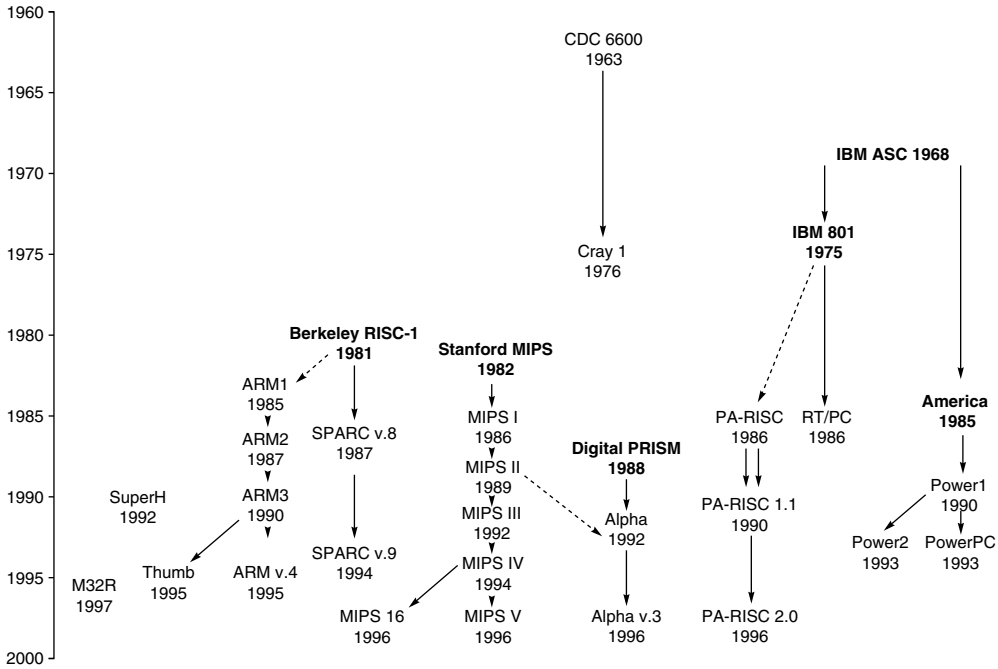

Figure K.35 The lineage of RISC instruction sets. Commercial machines are shown in plain text and research machines in bold. The CDC-6600 and Cray-1 were load-store machines with register 0 fixed at 0, and separate integer and floating-point registers. Instructions could not cross word boundaries. An early IBM research machine led to the 801 and America research projects, with the 801 leading to the unsuccessful RT/PC and America leading to the successful Power architecture. Some people who worked on the 801 later joined Hewlett-Packard to work on the PA-RISC. The two university projects were the basis of MIPS and SPARC machines. According to Furber [1996], the Berkeley RISC project was the inspiration of the ARM architecture. While ARM1, ARM2, and ARM3 were names of both architectures and chips, ARM version 4 is the name of the architecture used in ARM7, ARM8, and StrongARM chips. (There are no ARM v.4 and ARM5 chips, but ARM6 and early ARM7 chips use the ARM3 architecture.) DEC built a RISC microprocessor in 1988 but did not introduce it. Instead, DEC shipped workstations using MIPS microprocessors for three years before they brought out their own RISC instruction set, Alpha 21064, which is very similar to MIPS III and PRISM. The Alpha architecture has had small extensions, but they have not been formalized with version numbers; we used version 3 because that is the version of the reference manual. The Alpha 21164A chip added byte and half-word loads and stores, and the Alpha 21264 includes the MAX multimedia and bit count instructions. Internally, Digital names chips after the fabrication technology: EV4 (21064), EV45 (21064A), EV5 (21164), EV56 (21164A), and EV6 (21264). "EV" stands for "extended VAX."

Acknowledgments

We would like to thank the following people for comments on drafts of this survey: Professor Steven B. Furber, University of Manchester; Dr. Dileep Bhandarkar, Intel Corporation; Dr. Earl Killian, Silicon Graphics/MIPS; and Dr. Hiokazu Takata, Mitsubishi Electric Corporation.

	PA-RISC			SPARC		MIPS					Power		
Feature	**1.0**	**1.1**	**2.0**	**v. 8**	**v. 9**	**I**	**II**	**III**	**IV**	**V**	**1**	**2**	**PC**
Interlocked loads	X	"	"	X	"		+	"	"		X	"	"
Load-store FP double	X	"	"	X	"		+	"	"		X	"	"
Semaphore	X	"	"	X	"		+	"	"		X	"	"
Square root	X	"	"	X	"		+	"	"			+	"
Single-precision FP ops	X	"	"	X	"	X	"	"	"				+
Memory synchronize	X	"	"	X	"		+	"	"		X	"	"
Coprocessor	X	"	"	X	—	X	"	"	"				
Base + index addressing	X	"	"	X	"				+		X	"	"
Equiv. 32 64-bit FP registers		"	"		+			+	"		X	"	"
Annulling delayed branch	X	"	"	X	"		+	"	"				
Branch register contents	X	"	"		+	X	"	"	"				
Big/Little Endian		+	"		+	X	"	"	"				+
Branch-prediction bit					+		+	"	"		X	"	"
Conditional move					+				+		X	"	—
Prefetch data into cache			+		+				+		X	"	"
64-bit addressing/int. ops			+		+			+	"				+
32-bit multiply, divide		+	"		+	X	"	"	"		X	"	"
Load-store FP quad					+							+	—
Fused FP mul/add			+						+		X	"	"
String instructions	X	"	"								X	"	—
Multimedia support		X	"	X						X			

Figure K.36 Features added to desktop RISC machines. X means in the original machine, + means added later, " means continued from prior machine, and — means removed from architecture. Alpha is not included, but it added byte and word loads and stores, and bit count and multimedia extensions, in version 3. MIPS V added the MDMX instructions and paired single floating-point operations.

K.3 The Intel 80x86

Introduction

MIPS was the vision of a single architect. The pieces of this architecture fit nicely together and the whole architecture can be described succinctly. Such is not the case of the 80x86: It is the product of several independent groups who evolved the architecture over 20 years, adding new features to the original instruction set as you might add clothing to a packed bag. Here are important 80x86 milestones:

- 1978—The Intel 8086 architecture was announced as an assembly language–compatible extension of the then-successful Intel 8080, an 8-bit microprocessor. The 8086 is a 16-bit architecture, with all internal registers 16 bits wide. Whereas the 8080 was a straightforward accumulator machine, the 8086 extended the architecture with additional registers. Because nearly every register has a dedicated use, the 8086 falls somewhere between an accumulator machine and a general-purpose register machine, and can fairly be called an *extended accumulator* machine.
- 1980—The Intel 8087 floating-point coprocessor is announced. This architecture extends the 8086 with about 60 floating-point instructions. Its architects rejected extended accumulators to go with a hybrid of stacks and registers, essentially an *extended stack* architecture: A complete stack instruction set is supplemented by a limited set of register-memory instructions.
- 1982—The 80286 extended the 8086 architecture by increasing the address space to 24 bits, by creating an elaborate memory mapping and protection model, and by adding a few instructions to round out the instruction set and to manipulate the protection model. Because it was important to run 8086 programs without change, the 80286 offered a *real addressing mode* to make the machine look just like an 8086.
- 1985—The 80386 extended the 80286 architecture to 32 bits. In addition to a 32-bit architecture with 32-bit registers and a 32-bit address space, the 80386 added new addressing modes and additional operations. The added instructions make the 80386 nearly a general-purpose register machine. The 80386 also added paging support in addition to segmented addressing (see Chapter 2). Like the 80286, the 80386 has a mode to execute 8086 programs without change.

This history illustrates the impact of the "golden handcuffs" of compatibility on the 80x86, as the existing software base at each step was too important to jeopardize with significant architectural changes. Fortunately, the subsequent 80486 in 1989, Pentium in 1992, and P6 in 1995 were aimed at higher performance, with only four instructions added to the user-visible instruction set: three to help with multiprocessing plus a conditional move instruction.

Since 1997 Intel has added hundreds of instructions to support multimedia by operating on many narrower data types within a single clock (see Appendix A). These SIMD or vector instructions are primarily used in handcoded libraries or drivers and rarely generated by compilers. The first extension, called MMX, appeared in 1997. It consists of 57 instructions that pack and unpack multiple bytes, 16-bit words, or 32-bit double words into 64-bit registers and performs shift, logical, and integer arithmetic on the narrow data items in parallel. It supports both saturating and nonsaturating arithmetic. MMX uses the registers comprising the floating-point stack and hence there is no new state for operating systems to save.

In 1999 Intel added another 70 instructions, labeled SSE, as part of Pentium III. The primary changes were to add eight separate registers, double their width to 128 bits, and add a single-precision floating-point data type. Hence, four 32-bit floating-point operations can be performed in parallel. To improve memory performance, SSE included cache prefetch instructions plus streaming store instructions that bypass the caches and write directly to memory.

In 2001, Intel added yet another 144 instructions, this time labeled SSE2. The new data type is double-precision arithmetic, which allows pairs of 64-bit floating-point operations in parallel. Almost all of these 144 instructions are versions of existing MMX and SSE instructions that operate on 64 bits of data in parallel. Not only does this change enable multimedia operations, but it also gives the compiler a different target for floating-point operations than the unique stack architecture. Compilers can choose to use the eight SSE registers as floating-point registers as found in the RISC machines. This change has boosted performance on the Pentium 4, the first microprocessor to include SSE2 instructions. At the time of announcement, a 1.5 GHz Pentium 4 was 1.24 times faster than a 1 GHz Pentium III for SPECint2000(base), but it was 1.88 times faster for SPECfp2000(base).

In 2003, a company other than Intel enhanced the IA-32 architecture this time. AMD announced a set of architectural extensions to increase the address space for 32 to 64 bits. Similar to the transition from 16- to 32-bit address space in 1985 with the 80386, AMD64 widens all registers to 64 bits. It also increases the number of registers to 16 and has 16 128-bit registers to support XMM, AMD's answer to SSE2. Rather than expand the instruction set, the primary change is adding a new mode called *long mode* that redefines the execution of all IA-32 instructions with 64-bit addresses. To address the larger number of registers, it adds a new prefix to instructions. AMD64 still has a 32-bit mode that is backwards compatible to the standard Intel instruction set, allowing a more graceful transition to 64-bit addressing than the HP/Intel Itanium. Intel later followed AMD's lead, making almost identical changes so that most software can run on either 64-bit address version of the 80x86 without change.

Whatever the artistic failures of the 80x86, keep in mind that there are more instances of this architectural family than of any other server or desktop processor in the world. Nevertheless, its checkered ancestry has led to an architecture that is difficult to explain and impossible to love.

We start our explanation with the registers and addressing modes, move on to the integer operations, then cover the floating-point operations, and conclude with an examination of instruction encoding.

80x86 Registers and Data Addressing Modes

The evolution of the instruction set can be seen in the registers of the 80x86 (Figure K.37). Original registers are shown in black type, with the extensions of the 80386 shown in a lighter shade, a coloring scheme followed in subsequent figures. The 80386 basically extended all 16-bit registers (except the segment registers) to 32 bits, prefixing an "E" to their name to indicate the 32-bit version. The arithmetic, logical, and data transfer instructions are two-operand instructions that allow the combinations shown in Figure K.38.

To explain the addressing modes, we need to keep in mind whether we are talking about the 16-bit mode used by both the 8086 and 80286 or the 32-bit mode available on the 80386 and its successors. The seven data memory addressing modes supported are

- Absolute
- Register indirect
- Based
- Indexed
- Based indexed with displacement
- Based with scaled indexed
- Based with scaled indexed and displacement

Displacements can be 8 or 32 bits in 32-bit mode, and 8 or 16 bits in 16-bit mode. If we count the size of the address as a separate addressing mode, the total is 11 addressing modes.

Although a memory operand can use any addressing mode, there are restrictions on what registers can be used in a mode. The section "80x86 Instruction Encoding" on page K-55 gives the full set of restrictions on registers, but the following description of addressing modes gives the basic register options:

- *Absolute*—With 16-bit or 32-bit displacement, depending on the mode.
- *Register indirect*—`BX`, `SI`, `DI` in 16-bit mode and `EAX`, `ECX`, `EDX`, `EBX`, `ESI`, and `EDI` in 32-bit mode.
- *Based mode with 8-bit or 16-bit/32-bit displacement*—`BP`, `BX`, `SI`, and `DI` in 16-bit mode and `EAX`, `ECX`, `EDX`, `EBX`, `ESI`, and `EDI` in 32-bit mode. The displacement is either 8 bits or the size of the address mode: 16 or 32 bits. (Intel gives two different names to this single addressing mode, *based* and *indexed*, but they are essentially identical and we combine them. This book uses indexed addressing to mean something different, explained next.)

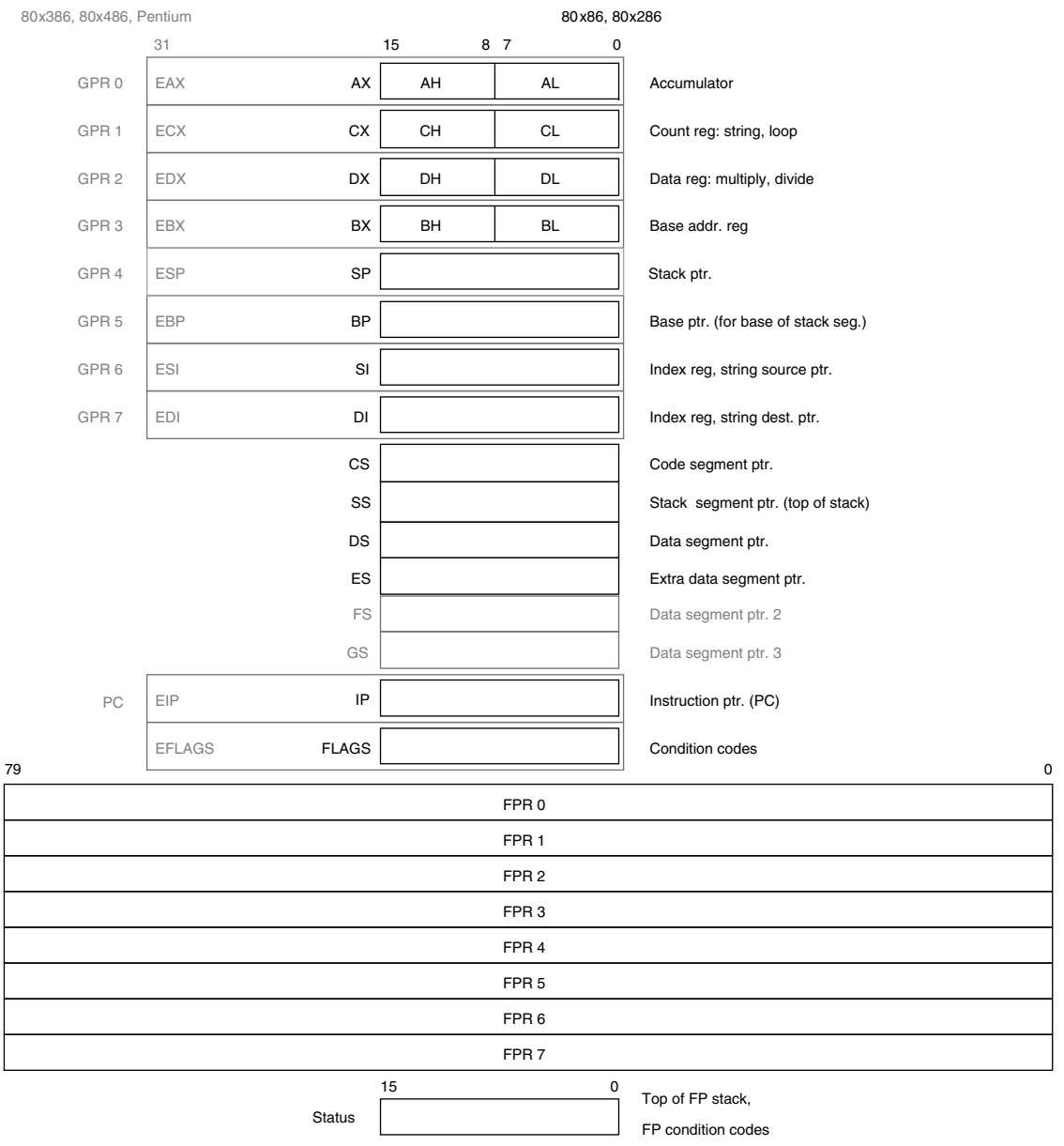

Figure K.37 The 80x86 has evolved over time, and so has its register set. The original set is shown in black and the extended set in gray. The 8086 divided the first four registers in half so that they could be used either as one 16-bit register or as two 8-bit registers. Starting with the 80386, the top eight registers were extended to 32 bits and could also be used as general-purpose registers. The floating-point registers on the bottom are 80 bits wide, and although they look like regular registers they are not. They implement a stack, with the top of stack pointed to by the status register. One operand must be the top of stack, and the other can be any of the other seven registers below the top of stack.

Source/destination operand type	Second source operand
Register	Register
Register	Immediate
Register	Memory
Memory	Register
Memory	Immediate

Figure K.38 Instruction types for the arithmetic, logical, and data transfer instructions. The 80x86 allows the combinations shown. The only restriction is the absence of a memory-memory mode. Immediates may be 8, 16, or 32 bits in length; a register is any one of the 14 major registers in Figure K.37 (not IP or FLAGS).

- *Indexed*—The address is the sum of two registers. The allowable combinations are `BX+SI`, `BX+DI`, `BP+SI`, and `BP+DI`. This mode is called *based indexed* on the 8086. (The 32-bit mode uses a different addressing mode to get the same effect.)
- *Based indexed with 8- or 16-bit displacement*—The address is the sum of displacement and contents of two registers. The same restrictions on registers apply as in indexed mode.
- *Base plus scaled indexed*—This addressing mode and the next were added in the 80386 and are only available in 32-bit mode. The address calculation is

$$\text{Base register} + 2^{\text{Scale}} \times \text{Index register}$$

  where *Scale* has the value 0, 1, 2, or 3; *Index register* can be any of the eight 32-bit general registers except `ESP`; and *Base register* can be any of the eight 32-bit general registers.
- *Base plus scaled index with 8- or 32-bit displacement*—The address is the sum of the displacement and the address calculated by the scaled mode immediately above. The same restrictions on registers apply.

The 80x86 uses Little Endian addressing.

Ideally, we would refer discussion of 80x86 logical and physical addresses to Chapter 2, but the segmented address space prevents us from hiding that information. Figure K.39 shows the memory mapping options on the generations of 80x86 machines; Chapter 2 describes the segmented protection scheme in greater detail.

The assembly language programmer clearly must specify which segment register should be used with an address, no matter which address mode is used. To save space in the instructions, segment registers are selected automatically depending on which address register is used. The rules are simple: References to instructions (`IP`) use the code segment register (`CS`), references to the stack (`BP` or `SP`) use the stack segment register (`SS`), and the default segment register for the other registers is the data segment register (`DS`). The next section explains how they can be overridden.

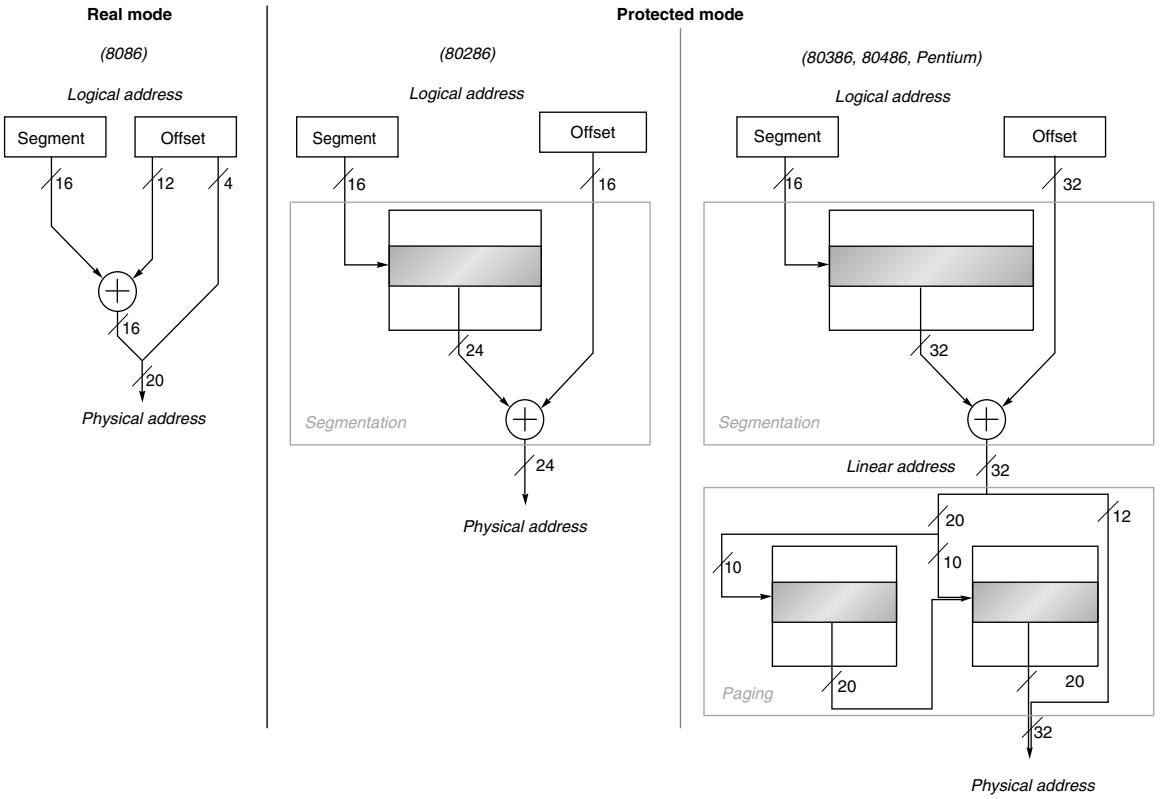

Figure K.39 The original segmented scheme of the 8086 is shown on the left. All 80x86 processors support this style of addressing, called *real mode*. It simply takes the contents of a segment register, shifts it left 4 bits, and adds it to the 16-bit offset, forming a 20-bit physical address. The 80286 (center) used the contents of the segment register to select a segment descriptor, which includes a 24-bit base address among other items. It is added to the 16-bit offset to form the 24-bit physical address. The 80386 and successors (right) expand this base address in the segment descriptor to 32 bits and also add an optional paging layer below segmentation. A 32-bit linear address is first formed from the segment and offset, and then this address is divided into two 10-bit fields and a 12-bit page offset. The first 10-bit field selects the entry in the first-level page table, and then this entry is used in combination with the second 10-bit field to access the second-level page table to select the upper 20 bits of the physical address. Prepending this 20-bit address to the final 12-bit field gives the 32-bit physical address. Paging can be turned off, redefining the 32-bit linear address as the physical address. Note that a "flat" 80x86 address space comes simply by loading the same value in all the segment registers; that is, it doesn't matter which segment register is selected.

80x86 Integer Operations

The 8086 provides support for both 8-bit (*byte*) and 16-bit (called *word*) data types. The data type distinctions apply to register operations as well as memory accesses. The 80386 adds 32-bit addresses and data, called *double words*. Almost every operation works on both 8-bit data and one longer data size. That size is determined by the mode and is either 16 or 32 bits.

Clearly some programs want to operate on data of all three sizes, so the 80x86 architects provide a convenient way to specify each version without expanding code size significantly. They decided that most programs would be dominated by either 16- or 32-bit data, and so it made sense to be able to set a default large size. This default size is set by a bit in the code segment register. To override the default size, an 8-bit *prefix* is attached to the instruction to tell the machine to use the other large size for this instruction.

The prefix solution was borrowed from the 8086, which allows multiple prefixes to modify instruction behavior. The three original prefixes override the default segment register, lock the bus so as to perform a semaphore (see Chapter 5), or repeat the following instruction until CX counts down to zero. This last prefix was intended to be paired with a byte move instruction to move a variable number of bytes. The 80386 also added a prefix to override the default address size.

The 80x86 integer operations can be divided into four major classes:

1. Data movement instructions, including move, push, and pop.
2. Arithmetic and logic instructions, including logical operations, test, shifts, and integer and decimal arithmetic operations.
3. Control flow, including conditional branches and unconditional jumps, calls, and returns.
4. String instructions, including string move and string compare.

Figure K.40 shows some typical 80x86 instructions and their functions.

The data transfer, arithmetic, and logic instructions are unremarkable, except that the arithmetic and logic instruction operations allow the destination to be either a register or a memory location.

Control flow instructions must be able to address destinations in another segment. This is handled by having two types of control flow instructions: "near" for intrasegment (within a segment) and "far" for intersegment (between segments) transfers. In far jumps, which must be unconditional, two 16-bit quantities follow the opcode in 16-bit mode. One of these is used as the instruction pointer, while the other is loaded into CS and becomes the new code segment. In 32-bit mode the first field is expanded to 32 bits to match the 32-bit program counter (EIP).

Calls and returns work similarly—a far call pushes the return instruction pointer and return segment on the stack and loads both the instruction pointer and the code segment. A far return pops both the instruction pointer and the code segment from the stack. Programmers or compiler writers must be sure to always use the same type of call *and* return for a procedure—a near return does not work with a far call, and *vice versa*.

String instructions are part of the 8080 ancestry of the 80x86 and are not commonly executed in most programs.

Figure K.41 lists some of the integer 80x86 instructions. Many of the instructions are available in both byte and word formats.

Instruction	Function
JE name	if equal(CC) {IP←name}; IP–128 ≤ name ≤ IP+128
JMP name	IP←name
CALLF name, seg	SP←SP–2; M[SS:SP]←IP+5; SP←SP–2; M[SS:SP]←CS; IP←name; CS←seg;
MOVW BX,[DI+45]	BX←$_{16}$M[DS:DI+45]
PUSH SI	SP←SP–2; M[SS:SP]←SI
POP DI	DI←M[SS:SP]; SP←SP+2
ADD AX,#6765	AX←AX+6765
SHL BX,1	BX←BX$_{1..15}$ ## 0
TEST DX,#42	Set CC flags with DX & 42
MOVSB	M[ES:DI]←$_{8}$M[DS:SI]; DI←DI+1; SI←SI+1

Figure K.40 Some typical 80x86 instructions and their functions. A list of frequent operations appears in Figure K.41. We use the abbreviation SR:X to indicate the formation of an address with segment register SR and offset X. This effective address corresponding to SR:X is (SR<<4)+X. The CALLF saves the IP of the next instruction and the current CS on the stack.

80x86 Floating-Point Operations

Intel provided a stack architecture with its floating-point instructions: loads push numbers onto the stack, operations find operands in the top two elements of the stacks, and stores can pop elements off the stack, just as the stack example in Figure A.2 on page A-4 suggests.

Intel supplemented this stack architecture with instructions and addressing modes that allow the architecture to have some of the benefits of a register-memory model. In addition to finding operands in the top two elements of the stack, one operand can be in memory or in one of the seven registers below the top of the stack.

This hybrid is still a restricted register-memory model, however, in that loads always move data to the top of the stack while incrementing the top of stack pointer and stores can only move the top of stack to memory. Intel uses the notation ST to indicate the top of stack, and ST(i) to represent the *i*th register below the top of stack.

One novel feature of this architecture is that the operands are wider in the register stack than they are stored in memory, and all operations are performed at this wide internal precision. Numbers are automatically converted to the internal 80-bit format on a load and converted back to the appropriate size on a store. Memory data can be 32-bit (single-precision) or 64-bit (double-precision) floating-point numbers, called *real* by Intel. The register-memory version of these instructions will then convert the memory operand to this Intel 80-bit format before performing

Instruction	Meaning
Control	**Conditional and unconditional branches**
JNZ, JZ	Jump if condition to IP + 8-bit offset; JNE (for JNZ) and JE (for JZ) are alternative names
JMP, JMPF	Unconditional jump—8- or 16-bit offset intrasegment (near) and intersegment (far) versions
CALL, CALLF	Subroutine call—16-bit offset; return address pushed; near and far versions
RET, RETF	Pops return address from stack and jumps to it; near and far versions
LOOP	Loop branch—decrement CX; jump to IP + 8-bit displacement if CX ¦ 0
Data transfer	**Move data between registers or between register and memory**
MOV	Move between two registers or between register and memory
PUSH	Push source operand on stack
POP	Pop operand from stack top to a register
LES	Load ES and one of the GPRs from memory
Arithmetic/logical	**Arithmetic and logical operations using the data registers and memory**
ADD	Add source to destination; register-memory format
SUB	Subtract source from destination; register-memory format
CMP	Compare source and destination; register-memory format
SHL	Shift left
SHR	Shift logical right
RCR	Rotate right with carry as fill
CBW	Convert byte in AL to word in AX
TEST	Logical AND of source and destination sets flags
INC	Increment destination; register-memory format
DEC	Decrement destination; register-memory format
OR	Logical OR; register-memory format
XOR	Exclusive OR; register-memory format
String instructions	**Move between string operands; length given by a repeat prefix**
MOVS	Copies from string source to destination; may be repeated
LODS	Loads a byte or word of a string into the A register

Figure K.41 Some typical operations on the 80x86. Many operations use register-memory format, where either the source or the destination may be memory and the other may be a register or immediate operand.

the operation. The data transfer instructions also will automatically convert 16- and 32-bit integers to reals, and *vice versa*, for integer loads and stores.

The 80x86 floating-point operations can be divided into four major classes:

1. Data movement instructions, including load, load constant, and store.
2. Arithmetic instructions, including add, subtract, multiply, divide, square root, and absolute value.

3. Comparison, including instructions to send the result to the integer CPU so that it can branch.
4. Transcendental instructions, including sine, cosine, log, and exponentiation.

Figure K.42 shows some of the 60 floating-point operations. We use the curly brackets {} to show optional variations of the basic operations: {I} means there is an integer version of the instruction, {P} means this variation will pop one operand off the stack after the operation, and {R} means reverse the sense of the operands in this operation.

Not all combinations are provided. Hence,

```
F{I}SUB{R}{P}
```

represents these instructions found in the 80x86:

```
FSUB
FISUB
FSUBR
FISUBR
FSUBP
FSUBRP
```

There are no pop or reverse pop versions of the integer subtract instructions.

Data transfer	Arithmetic	Compare	Transcendental
F{I}LD mem/ST(i)	F{I}ADD{P} mem/ST(i)	F{I}COM{P}{P}	FPATAN
F{I}ST{P} mem/ST(i)	F{I}SUB{R}{P} mem/ST(i)	F{I}UCOM{P}{P}	F2XM1
FLDPI	F{I}MUL{P} mem/ST(i)	FSTSW AX/mem	FCOS
FLD1	F{I}DIV{R}{P} mem/ST(i)		FPTAN
FLDZ	FSQRT		FPREM
	FABS		FSIN
	FRNDINT		FYL2X

Figure K.42 The floating-point instructions of the 80x86. The first column shows the data transfer instructions, which move data to memory or to one of the registers below the top of the stack. The last three operations push constants on the stack: pi, 1.0, and 0.0. The second column contains the arithmetic operations described above. Note that the last three operate only on the top of stack. The third column is the compare instructions. Since there are no special floating-point branch instructions, the result of the compare must be transferred to the integer CPU via the FSTSW instruction, either into the AX register or into memory, followed by an SAHF instruction to set the condition codes. The floating-point comparison can then be tested using integer branch instructions. The final column gives the higher-level floating-point operations.

Note that we get even more combinations when including the operand modes for these operations. The floating-point add has these options, ignoring the integer and pop versions of the instruction:

`FADD`		Both operands are in the in stack, and the result replaces the top of stack.
`FADD`	`ST(i)`	One source operand is *i*th register below the top of stack, and the result replaces the top of stack.
`FADD`	`ST(i),ST`	One source operand is the top of stack, and the result replaces *i*th register below the top of stack.
`FADD`	`mem32`	One source operand is a 32-bit location in memory, and the result replaces the top of stack.
`FADD`	`mem64`	One source operand is a 64-bit location in memory, and the result replaces the top of stack.

As mentioned earlier SSE2 presents a model of IEEE floating-point registers.

80x86 Instruction Encoding

Saving the worst for last, the encoding of instructions in the 8086 is complex, with many different instruction formats. Instructions may vary from 1 byte, when there are no operands, to up to 6 bytes, when the instruction contains a 16-bit immediate and uses 16-bit displacement addressing. Prefix instructions increase 8086 instruction length beyond the obvious sizes.

The 80386 additions expand the instruction size even further, as Figure K.43 shows. Both the displacement and immediate fields can be 32 bits long, two more prefixes are possible, the opcode can be 16 bits long, and the scaled index mode specifier adds another 8 bits. The maximum possible 80386 instruction is 17 bytes long.

Figure K.44 shows the instruction format for several of the example instructions in Figure K.40. The opcode byte usually contains a bit saying whether the operand is a byte wide or the larger size, 16 bits or 32 bits depending on the mode. For some instructions, the opcode may include the addressing mode and the register; this is true in many instructions that have the form `register ←register op immediate`. Other instructions use a "postbyte" or extra opcode byte, labeled "mod, reg, r/m" in Figure K.43, which contains the addressing mode information. This postbyte is used for many of the instructions that address memory. The based with scaled index uses a second postbyte, labeled "sc, index, base" in Figure K.43.

The floating-point instructions are encoded in the escape opcode of the 8086 and the postbyte address specifier. The memory operations reserve 2 bits to decide whether the operand is a 32- or 64-bit real or a 16- or 32-bit integer. Those same 2 bits are used in versions that do not access memory to decide whether the stack should be popped after the operation and whether the top of stack or a lower register should get the result.

Alas, you cannot separate the restrictions on registers from the encoding of the addressing modes in the 80x86. Hence, Figures K.45 and K.46 show the encoding of the two postbyte address specifiers for both 16- and 32-bit mode.

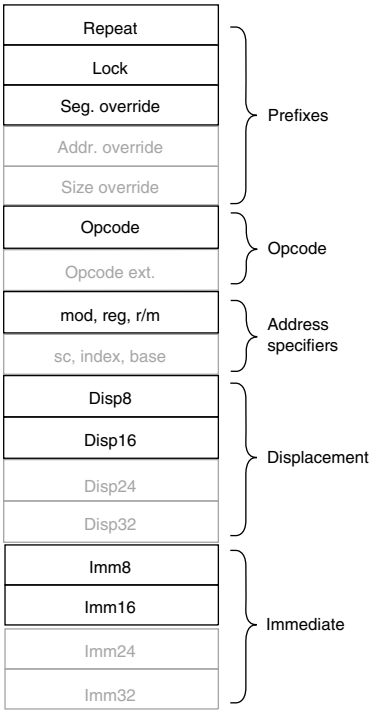

Figure K.43 The instruction format of the 8086 (black type) and the extensions for the 80386 (shaded type). Every field is optional except the opcode.

Putting It All Together: Measurements of Instruction Set Usage

In this section, we present detailed measurements for the 80x86 and then compare the measurements to MIPS for the same programs. To facilitate comparisons among dynamic instruction set measurements, we use a subset of the SPEC92 programs. The 80x86 results were taken in 1994 using the Sun Solaris FORTRAN and C compilers V2.0 and executed in 32-bit mode. These compilers were comparable in quality to the compilers used for MIPS.

Remember that these measurements depend on the benchmarks chosen and the compiler technology used. Although we feel that the measurements in this section are reasonably indicative of the usage of these architectures, other programs may behave differently from any of the benchmarks here, and different compilers may yield different results. In doing a real instruction set study, the architect would want to have a much larger set of benchmarks, spanning as wide an application range as possible, and consider the operating system and its usage

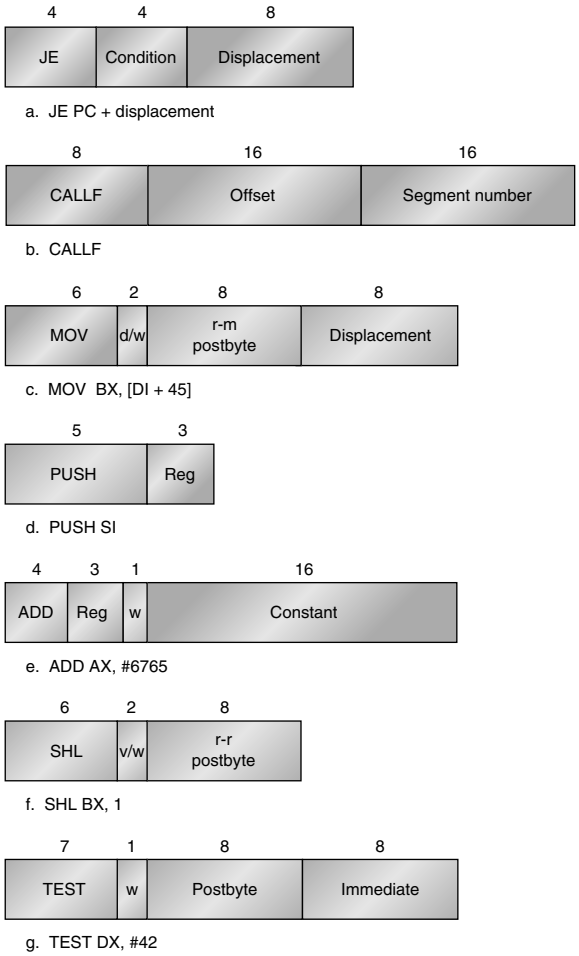

Figure K.44 Typical 8086 instruction formats. The encoding of the postbyte is shown in Figure K.45. Many instructions contain the 1-bit field w, which says whether the operation is a byte or a word. Fields of the form v/w or d/w are a d-field or v-field followed by the w-field. The d-field in `MOV` is used in instructions that may move to or from memory and shows the direction of the move. The field v in the `SHL` instruction indicates a variable-length shift; variable-length shifts use a register to hold the shift count. The `ADD` instruction shows a typical optimized short encoding usable only when the first operand is `AX`. Overall instructions may vary from 1 to 6 bytes in length.

of the instruction set. Single-user benchmarks like those measured here do not necessarily behave in the same fashion as the operating system.

We start with an evaluation of the features of the 80x86 in isolation, and later compare instruction counts with those of DLX.

		w = 1			mod = 0		mod = 1		mod = 2		
reg	w = 0	16b	32b	r/m	16b	32b	16b	32b	16b	32b	mod = 3
0	AL	AX	EAX	0	addr=BX+SI	=EAX	*same*	*same*	*same*	*same*	*same*
1	CL	CX	ECX	1	addr=BX+DI	=ECX	*addr as*	*addr as*	*addr as*	*addr as*	*as*
2	DL	DX	EDX	2	addr=BP+SI	=EDX	*mod=0*	*mod=0*	*mod=0*	*mod=0*	*reg*
3	BL	BX	EBX	3	addr=BP+SI	=EBX	*+ disp8*	*+ disp8*	*+ disp16*	*+ disp32*	*field*
4	AH	SP	ESP	4	addr=SI	=(sib)	SI+disp16	(sib)+disp8	SI+disp8	(sib)+disp32	"
5	CH	BP	EBP	5	addr=DI	=disp32	DI+disp8	EBP+disp8	DI+disp16	EBP+disp32	"
6	DH	SI	ESI	6	addr=disp16	=ESI	BP+disp8	ESI+disp8	BP+disp16	ESI+disp32	"
7	BH	DI	EDI	7	addr=BX	=EDI	BX+disp8	EDI+disp8	BX+disp16	EDI+disp32	"

Figure K.45 The encoding of the first address specifier of the 80x86, *mod, reg, r/m.* The first four columns show the encoding of the 3-bit reg field, which depends on the w bit from the opcode and whether the machine is in 16- or 32-bit mode. The remaining columns explain the mod and r/m fields. The meaning of the 3-bit r/m field depends on the value in the 2-bit mod field and the address size. Basically, the registers used in the address calculation are listed in the sixth and seventh columns, under mod = 0, with mod = 1 adding an 8-bit displacement and mod = 2 adding a 16- or 32-bit displacement, depending on the address mode. The exceptions are r/m = 6 when mod = 1 or mod = 2 in 16-bit mode selects BP plus the displacement; r/m = 5 when mod = 1 or mod = 2 in 32-bit mode selects EBP plus displacement; and r/m = 4 in 32-bit mode when mod ¦3 (sib) means use the scaled index mode shown in Figure K.46. When mod = 3, the r/m field indicates a register, using the same encoding as the reg field combined with the w bit.

Index		Base
0	EAX	EAX
1	ECX	ECX
2	EDX	EDX
3	EBX	EBX
4	No index	ESP
5	EBP	If mod = 0, disp32 If mod ¦ 0, EBP
6	ESI	ESI
7	EDI	EDI

Figure K.46 Based plus scaled index mode address specifier found in the 80386. This mode is indicated by the (sib) notation in Figure K.45. Note that this mode expands the list of registers to be used in other modes: Register indirect using ESP comes from Scale = 0, Index = 4, and Base = 4, and base displacement with EBP comes from Scale = 0, Index = 5, and mod = 0. The two-bit scale field is used in this formula of the effective address: Base register + $2^{\text{Scale}} \times$ Index register.

Measurements of 80x86 Operand Addressing

We start with addressing modes. Figure K.47 shows the distribution of the operand types in the 80x86. These measurements cover the "second" operand of the operation; for example,

```
mov EAX, [45]
```

counts as a single memory operand. If the types of the first operand were counted, the percentage of register usage would increase by about a factor of 1.5.

The 80x86 memory operands are divided into their respective addressing modes in Figure K.48. Probably the biggest surprise is the popularity of the addressing modes added by the 80386, the last four rows of the figure. They account for about half of all the memory accesses. Another surprise is the popularity of direct addressing. On most other machines, the equivalent of the direct

	Integer average	FP average
Register	45%	22%
Immediate	16%	6%
Memory	39%	72%

Figure K.47 Operand type distribution for the average of five SPECint92 programs (compress, eqntott, espresso, gcc, li) and the average of five SPECfp92 programs (doduc, ear, hydro2d, mdljdp2, su2cor).

Addressing mode	Integer average	FP average
Register indirect	13%	3%
Base + 8-bit disp.	31%	15%
Base + 32-bit disp.	9%	25%
Indexed	0%	0%
Based indexed + 8-bit disp.	0%	0%
Based indexed + 32-bit disp.	0%	1%
Base + scaled indexed	22%	7%
Base + scaled indexed + 8-bit disp.	0%	8%
Base + scaled indexed + 32-bit disp.	4%	4%
32-bit direct	20%	37%

Figure K.48 Operand addressing mode distribution by program. This chart does not include addressing modes used by branches or control instructions.

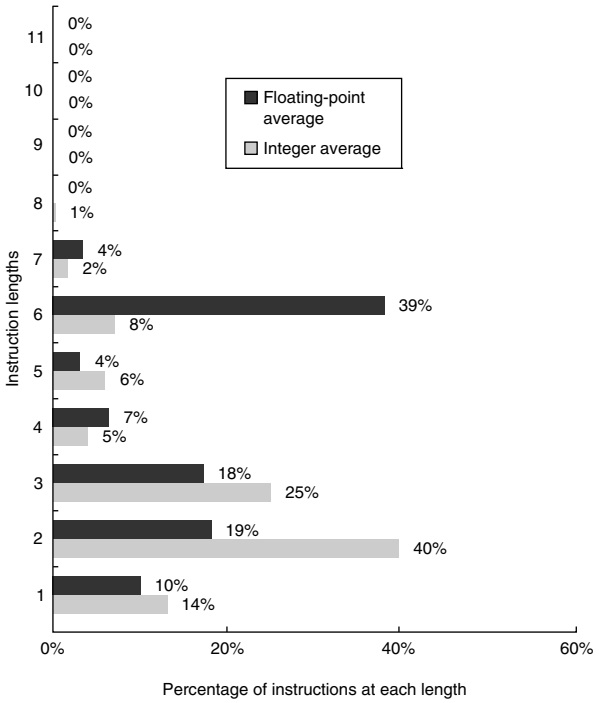

Figure K.49 Averages of the histograms of 80x86 instruction lengths for five SPECint92 programs and for five SPECfp92 programs, all running in 32-bit mode.

addressing mode is rare. Perhaps the segmented address space of the 80x86 makes direct addressing more useful, since the address is relative to a base address from the segment register.

These addressing modes largely determine the size of the Intel instructions. Figure K.49 shows the distribution of instruction sizes. The average number of bytes per instruction for integer programs is 2.8, with a standard deviation of 1.5, and 4.1 with a standard deviation of 1.9 for floating-point programs. The difference in length arises partly from the differences in the addressing modes: Integer programs rely more on the shorter register indirect and 8-bit displacement addressing modes, while floating-point programs more frequently use the 80386 addressing modes with the longer 32-bit displacements.

Given that the floating-point instructions have aspects of both stacks and registers, how are they used? Figure K.50 shows that, at least for the compilers used in this measurement, the stack model of execution is rarely followed. (See Section L.3 for a historical explanation of this observation.)

Finally, Figures K.51 and K.52 show the instruction mixes for 10 SPEC92 programs.

Option	doduc	ear	hydro2d	mdljdp2	su2cor	FP average
Stack (2nd operand ST (1))	1.1%	0.0%	0.0%	0.2%	0.6%	0.4%
Register (2nd operand ST(i), i > 1)	17.3%	63.4%	14.2%	7.1%	30.7%	26.5%
Memory	81.6%	36.6%	85.8%	92.7%	68.7%	73.1%

Figure K.50 The percentage of instructions for the floating-point operations (add, sub, mul, div) that use each of the three options for specifying a floating-point operand on the 80x86. The three options are (1) the strict stack model of implicit operands on the stack, (2) register version naming an explicit operand that is not one of the top two elements of the stack, and (3) memory operand.

Instruction	doduc	ear	hydro2d	mdljdp2	su2cor	FP average
Load	8.9%	6.5%	18.0%	27.6%	27.6%	20%
Store	12.4%	3.1%	11.5%	7.8%	7.8%	8%
Add	5.4%	6.6%	14.6%	8.8%	8.8%	10%
Sub	1.0%	2.4%	3.3%	2.4%	2.4%	3%
Mul						0%
Div						0%
Compare	1.8%	5.1%	0.8%	1.0%	1.0%	2%
Mov reg-reg	3.2%	0.1%	1.8%	2.3%	2.3%	2%
Load imm	0.4%	1.5%				0%
Cond. branch	5.4%	8.2%	5.1%	2.7%	2.7%	5%
Uncond branch	0.8%	0.4%	1.3%	0.3%	0.3%	1%
Call	0.5%	1.6%		0.1%	0.1%	0%
Return, jmp indirect	0.5%	1.6%		0.1%	0.1%	0%
Shift	1.1%		4.5%	2.5%	2.5%	2%
AND	0.8%	0.8%	0.7%	1.3%	1.3%	1%
OR	0.1%			0.1%	0.1%	0%
Other (XOR, not, . . .)						0%
Load FP	14.1%	22.5%	9.1%	12.6%	12.6%	14%
Store FP	8.6%	11.4%	4.1%	6.6%	6.6%	7%
Add FP	5.8%	6.1%	1.4%	6.6%	6.6%	5%
Sub FP	2.2%	2.7%	3.1%	2.9%	2.9%	3%
Mul FP	8.9%	8.0%	4.1%	12.0%	12.0%	9%
Div FP	2.1%		0.8%	0.2%	0.2%	0%
Compare FP	9.4%	6.9%	10.8%	0.5%	0.5%	5%
Mov reg-reg FP	2.5%	0.8%	0.3%	0.8%	0.8%	1%
Other (abs, sqrt, . . .)	3.9%	3.8%	4.1%	0.8%	0.8%	2%

Figure K.51 80x86 instruction mix for five SPECfp92 programs.

Instruction	compress	eqntott	espresso	gcc (cc1)	li	Int. average
Load	20.8%	18.5%	21.9%	24.9%	23.3%	22%
Store	13.8%	3.2%	8.3%	16.6%	18.7%	12%
Add	10.3%	8.8%	8.15%	7.6%	6.1%	8%
Sub	7.0%	10.6%	3.5%	2.9%	3.6%	5%
Mul				0.1%		0%
Div						0%
Compare	8.2%	27.7%	15.3%	13.5%	7.7%	16%
Mov reg-reg	7.9%	0.6%	5.0%	4.2%	7.8%	4%
Load imm	0.5%	0.2%	0.6%	0.4%		0%
Cond. branch	15.5%	28.6%	18.9%	17.4%	15.4%	20%
Uncond. branch	1.2%	0.2%	0.9%	2.2%	2.2%	1%
Call	0.5%	0.4%	0.7%	1.5%	3.2%	1%
Return, jmp indirect	0.5%	0.4%	0.7%	1.5%	3.2%	1%
Shift	3.8%		2.5%	1.7%		1%
AND	8.4%	1.0%	8.7%	4.5%	8.4%	6%
OR	0.6%		2.7%	0.4%	0.4%	1%
Other (XOR, not, . . .)	0.9%		2.2%	0.1%		1%
Load FP						0%
Store FP						0%
Add FP						0%
Sub FP						0%
Mul FP						0%
Div FP						0%
Compare FP						0%
Mov reg-reg FP						0%
Other (abs, sqrt, . . .)						0%

Figure K.52 80x86 instruction mix for five SPECint92 programs.

Comparative Operation Measurements

Figures K.53 and K.54 show the number of instructions executed for each of the 10 programs on the 80x86 and the ratio of instruction execution compared with that for DLX: Numbers less than 1.0 mean that the 80x86 executes fewer instructions than DLX. The instruction count is surprisingly close to DLX for many integer programs, as you would expect a load-store instruction set architecture like DLX to execute more instructions than a register-memory architecture like the 80x86. The floating-point programs always have higher counts for the 80x86, presumably due to the lack of floating-point registers and the use of a stack architecture.

	compress	eqntott	espresso	gcc (cc1)	li	Int. avg.
Instructions executed on 80x86 (millions)	2226	1203	2216	3770	5020	
Instructions executed ratio to DLX	0.61	1.74	0.85	0.96	0.98	1.03
Data reads on 80x86 (millions)	589	229	622	1079	1459	
Data writes on 80x86 (millions)	311	39	191	661	981	
Data read-modify-writes on 80x86 (millions)	26	1	129	48	48	
Total data reads on 80x86 (millions)	615	230	751	1127	1507	
Data read ratio to DLX	0.85	1.09	1.38	1.25	0.94	1.10
Total data writes on 80x86 (millions)	338	40	319	709	1029	
Data write ratio to DLX	1.67	9.26	2.39	1.25	1.20	3.15
Total data accesses on 80x86 (millions)	953	269	1070	1836	2536	
Data access ratio to DLX	1.03	1.25	1.58	1.25	1.03	1.23

Figure K.53 Instructions executed and data accesses on 80x86 and ratios compared to DLX for five SPECint92 programs.

	doduc	ear	hydro2d	mdljdp2	su2cor	FP average
Instructions executed on 80x86 (millions)	1223	15,220	13,342	6197	6197	
Instructions executed ratio to DLX	1.19	1.19	2.53	2.09	1.62	1.73
Data reads on 80x86 (millions)	515	6007	5501	3696	3643	
Data writes on 80x86 (millions)	260	2205	2085	892	892	
Data read-modify-writes on 80x86 (millions)	1	0	189	124	124	
Total data reads on 80x86 (millions)	517	6007	5690	3820	3767	
Data read ratio to DLX	2.04	2.36	4.48	4.77	3.91	3.51
Total data writes on 80x86 (millions)	261	2205	2274	1015	1015	
Data write ratio to DLX	3.68	33.25	38.74	16.74	9.35	20.35
Total data accesses on 80x86 (millions)	778	8212	7965	4835	4782	
Data access ratio to DLX	2.40	3.14	5.99	5.73	4.47	4.35

Figure K.54 Instructions executed and data accesses for five SPECfp92 programs on 80x86 and ratio to DLX.

Another question is the total amount of data traffic for the 80x86 versus DLX, since the 80x86 can specify memory operands as part of operations while DLX can only access via loads and stores. Figures K.53 and K.54 also show the data reads, data writes, and data read-modify-writes for these 10 programs. The total accesses ratio to DLX of each memory access type is shown in the bottom rows, with the read-modify-write counting as one read and one write. The 80x86

	Integer average		FP average	
Category	**x86**	**DLX**	**x86**	**DLX**
Total data transfer	34%	36%	28%	2%
Total integer arithmetic	34%	31%	16%	12%
Total control	24%	20%	6%	10%
Total logical	8%	13%	3%	2%
Total FP data transfer	0%	0%	22%	33%
Total FP arithmetic	0%	0%	25%	41%

Figure K.55 Percentage of instructions executed by category for 80x86 and DLX for the averages of five SPECint92 and SPECfp92 programs of Figures K.53 and K.54.

performs about two to four times as many data accesses as DLX for floating-point programs, and 1.25 times as many for integer programs. Finally, Figure K.55 shows the percentage of instructions in each category for 80x86 and DLX.

Concluding Remarks

Beauty is in the eye of the beholder.

Old Adage

As we have seen, "orthogonal" is not a term found in the Intel architectural dictionary. To fully understand which registers and which addressing modes are available, you need to see the encoding of all addressing modes and sometimes the encoding of the instructions.

Some argue that the inelegance of the 80x86 instruction set is unavoidable, the price that must be paid for rampant success by any architecture. We reject that notion. Obviously, no successful architecture can jettison features that were added in previous implementations, and over time some features may be seen as undesirable. The awkwardness of the 80x86 began at its core with the 8086 instruction set and was exacerbated by the architecturally inconsistent expansions of the 8087, 80286, and 80386.

A counterexample is the IBM 360/370 architecture, which is much older than the 80x86. It dominates the mainframe market just as the 80x86 dominates the PC market. Due undoubtedly to a better base and more compatible enhancements, this instruction set makes much more sense than the 80x86 more than 30 years after its first implementation.

For better or worse, Intel had a 16-bit microprocessor years before its competitors' more elegant architectures, and this head start led to the selection of the 8086 as the CPU for the IBM PC. What it lacks in style is made up in quantity, making the 80x86 beautiful from the right perspective.

The saving grace of the 80x86 is that its architectural components are not too difficult to implement, as Intel has demonstrated by rapidly improving performance of integer programs since 1978. High floating-point performance is a larger challenge in this architecture.

K.4 The VAX Architecture

VAX: the most successful minicomputer design in industry history . . . the VAX was probably the hacker's favorite machine . . . . Especially noted for its large, assembler-programmer-friendly instruction set—an asset that became a liability after the RISC revolution.

Eric Raymond
The New Hacker's Dictionary (1991)

Introduction

To enhance your understanding of instruction set architectures, we chose the VAX as the representative *Complex Instruction Set Computer* (CISC) because it is so different from MIPS and yet still easy to understand. By seeing two such divergent styles, we are confident that you will be able to learn other instruction sets on your own.

At the time the VAX was designed, the prevailing philosophy was to create instruction sets that were close to programming languages in order to simplify compilers. For example, because programming languages had loops, instruction sets should have loop instructions. As VAX architect William Strecker said ("VAX-11/780—A Virtual Address Extension to the PDP-11 Family," *AFIPS Proc.,* National Computer Conference, 1978):

> A major goal of the VAX-11 instruction set was to provide for effective compiler generated code. Four decisions helped to realize this goal: . . . 1) A very regular and consistent treatment of operators . . . . 2) An avoidance of instructions unlikely to be generated by a compiler . . . . 3) Inclusions of several forms of common operators . . . . 4) Replacement of common instruction sequences with single instructions. Examples include procedure calling, multiway branching, loop control, and array subscript calculation.

Recall that DRAMs of the mid-1970s contained less than 1/1000th the capacity of today's DRAMs, so code space was also critical. Hence, another prevailing philosophy was to minimize code size, which is de-emphasized in fixed-length instruction sets like MIPS. For example, MIPS address fields always use 16 bits, even when the address is very small. In contrast, the VAX allows instructions to be a variable number of bytes, so there is little wasted space in address fields.

Whole books have been written just about the VAX, so this VAX extension cannot be exhaustive. Hence, the following sections describe only a few of its addressing modes and instructions. To show the VAX instructions in action, later

sections show VAX assembly code for two C procedures. The general style will be to contrast these instructions with the MIPS code that you are already familiar with.

The differing goals for VAX and MIPS have led to very different architectures. The VAX goals, simple compilers and code density, led to the powerful addressing modes, powerful instructions, and efficient instruction encoding. The MIPS goals were high performance via pipelining, ease of hardware implementation, and compatibility with highly optimizing compilers. The MIPS goals led to simple instructions, simple addressing modes, fixed-length instruction formats, and a large number of registers.

VAX Operands and Addressing Modes

The VAX is a 32-bit architecture, with 32-bit-wide addresses and 32-bit-wide registers. Yet, the VAX supports many other data sizes and types, as Figure K.56 shows. Unfortunately, VAX uses the name "word" to refer to 16-bit quantities; in this text, a word means 32 bits. Figure K.56 shows the conversion between the MIPS data type names and the VAX names. Be careful when reading about VAX instructions, as they refer to the names of the VAX data types.

The VAX provides 16 32-bit registers. The VAX assembler uses the notation `r0`, `r1`, . . . , `r15` to refer to these registers, and we will stick to that notation. Alas, 4 of these 16 registers are effectively claimed by the instruction set architecture. For example, `r14` is the stack pointer (`sp`) and `r15` is the program counter (`pc`). Hence, `r15` cannot be used as a general-purpose register, and using `r14` is very difficult because it interferes with instructions that manipulate the stack. The other dedicated registers are `r12`, used as the argument pointer (`ap`), and `r13`, used as the frame pointer (`fp`); their purpose will become clear later. (Like MIPS, the VAX assembler accepts either the register number or the register name.)

Bits	Data type	MIPS name	VAX name
8	Integer	Byte	Byte
16	Integer	Half word	Word
32	Integer	Word	Long word
32	Floating point	Single precision	F_floating
64	Integer	Double word	Quad word
64	Floating point	Double precision	D_floating or G_floating
8n	Character string	Character	Character

Figure K.56 VAX data types, their lengths, and names. The first letter of the VAX type (b, w, l, f, q, d, g, c) is often used to complete an instruction name. Examples of move instructions include `movb`, `movw`, `movl`, `movf`, `movq`, `movd`, `movg`, and `movc3`. Each move instruction transfers an operand of the data type indicated by the letter following `mov`.

VAX addressing modes include those discussed in Appendix A, which has all the MIPS addressing modes: *register*, *displacement*, *immediate*, and *PC-relative*. Moreover, all these modes can be used for jump addresses or for data addresses.

But that's not all the addressing modes. To reduce code size, the VAX has three lengths of addresses for displacement addressing: 8-bit, 16-bit, and 32-bit addresses called, respectively, *byte displacement*, *word displacement*, and *long displacement* addressing. Thus, an address can be not only as small as possible but also as large as necessary; large addresses need not be split, so there is no equivalent to the MIPS `lui` instruction (see Figure A.24 on page A-37).

Those are still not all the VAX addressing modes. Several have a *deferred* option, meaning that the object addressed is only the *address* of the real object, requiring another memory access to get the operand. This addressing mode is called *indirect addressing* in other machines. Thus, *register deferred*, *autoincrement deferred*, and *byte/word/long displacement deferred* are other addressing modes to choose from. For example, using the notation of the VAX assembler, `r1` means the operand is register 1 and `(r1)` means the operand is the location in memory pointed to by `r1`.

There is yet another addressing mode. *Indexed addressing* automatically converts the value in an index operand to the proper byte address to add to the rest of the address. For a 32-bit word, we needed to multiply the index of a 4-byte quantity by 4 before adding it to a base address. Indexed addressing, called *scaled addressing* on some computers, automatically multiplies the index of a 4-byte quantity by 4 as part of the address calculation.

To cope with such a plethora of addressing options, the VAX architecture separates the specification of the addressing mode from the specification of the operation. Hence, the opcode supplies the operation and the number of operands, and each operand has its own addressing mode specifier. Figure K.57 shows the name, assembler notation, example, meaning, and length of the address specifier.

The VAX style of addressing means that an operation doesn't know where its operands come from; a VAX `add` instruction can have three operands in registers, three operands in memory, or any combination of registers and memory operands.

Example How long is the following instruction?

```
addl3 r1,737(r2),(r3)[r4]
```

The name `addl3` means a 32-bit add instruction with three operands. Assume the length of the VAX opcode is 1 byte.

Answer The first operand specifier—`r1`—indicates register addressing and is 1 byte long. The second operand specifier—`737(r2)`—indicates displacement addressing and has two parts: The first part is a byte that specifies the word displacement addressing mode and base register (`r2`); the second part is the 2-byte-long displacement (`737`). The third operand specifier—`(r3)[r4]`—also has two parts: The first byte specifies register deferred addressing mode (`(r3)`), and the second byte specifies the Index register and the use of indexed addressing (`[r4]`). Thus, the total length of the instruction is 1 + (1) + (1 + 2) + (1 + 1) = 7 bytes.

Addressing mode name	Syntax	Example	Meaning	Length of address specifier in bytes
Literal	#value	#–1	–1	1 (6-bit signed value)
Immediate	#value	#100	100	1 + length of the immediate
Register	rn	r3	r3	1
Register deferred	(rn)	(r3)	Memory[r3]	1
Byte/word/long displacement	Displacement (rn)	100(r3)	Memory[r3 + 100]	1 + length of the displacement
Byte/word/long displacement deferred	@displacement (rn)	@100(r3)	Memory[Memory [r3 + 100]]	1 + length of the displacement
Indexed (scaled)	Base mode [rx]	(r3)[r4]	Memory[r3 + r4 × *d*] (where *d* is data size in bytes)	1 + length of base addressing mode
Autoincrement	(rn)+	(r3)+	Memory[r3]; r3 = r3 + *d*	1
Autodecrement	– (rn)	–(r3)	r3 = r3 – *d;* Memory[r3]	1
Autoincrement deferred	@(rn)+	@(r3)+	Memory[Memory[r3]]; r3 = r3 + *d*	1

Figure K.57 Definition and length of the VAX operand specifiers. The length of each addressing mode is 1 byte plus the length of any displacement or immediate field needed by the mode. Literal mode uses a special 2-bit tag and the remaining 6 bits encode the constant value. If the constant is too big, it must use the immediate addressing mode. Note that the length of an immediate operand is dictated by the length of the data type indicated in the opcode, not the value of the immediate. The symbol *d* in the last four modes represents the length of the data in bytes; *d* is 4 for 32-bit add.

In this example instruction, we show the VAX destination operand on the left and the source operands on the right, just as we show MIPS code. The VAX assembler actually expects operands in the opposite order, but we felt it would be less confusing to keep the destination on the left for both machines. Obviously, left or right orientation is arbitrary; the only requirement is consistency.

Elaboration Because the PC is one of the 16 registers that can be selected in a VAX addressing mode, 4 of the 22 VAX addressing modes are synthesized from other addressing modes. Using the PC as the chosen register in each case, immediate addressing is really autoincrement, PC-relative is displacement, absolute is autoincrement deferred, and relative deferred is displacement deferred.

Encoding VAX Instructions

Given the independence of the operations and addressing modes, the encoding of instructions is quite different from MIPS.

VAX instructions begin with a single byte opcode containing the operation and the number of operands. The operands follow the opcode. Each operand begins with a single byte, called the *address specifier*, that describes the addressing mode for that operand. For a simple addressing mode, such as register

addressing, this byte specifies the register number as well as the mode (see the rightmost column in Figure K.57). In other cases, this initial byte can be followed by many more bytes to specify the rest of the address information.

As a specific example, let's show the encoding of the add instruction from the example on page K-67:

```
addl3 r1,737(r2),(r3)[r4]
```

Assume that this instruction starts at location 201.

Figure K.58 shows the encoding. Note that the operands are stored in memory in opposite order to the assembly code above. The execution of VAX instructions begins with fetching the source operands, so it makes sense for them to come first. Order is not important in fixed-length instructions like MIPS, since the source and destination operands are easily found within a 32-bit word.

The first byte, at location 201, is the opcode. The next byte, at location 202, is a specifier for the index mode using register `r4`. Like many of the other specifiers, the left 4 bits of the specifier give the mode and the right 4 bits give the register used in that mode. Since `addl3` is a 4-byte operation, `r4` will be multiplied by 4 and added to whatever address is specified next. In this case it is register deferred addressing using register `r3`. Thus, bytes 202 and 203 combined define the third operand in the assembly code.

The following byte, at address 204, is a specifier for word displacement addressing using register `r2` as the base register. This specifier tells the VAX that the following two bytes, locations 205 and 206, contain a 16-bit address to be added to `r2`.

The final byte of the instruction gives the destination operand, and this specifier selects register addressing using register `r1`.

Such variability in addressing means that a single VAX operation can have many different lengths; for example, an integer add varies from 3 bytes to 19 bytes. VAX implementations must decode the first operand before they can find the second, and so implementors are strongly tempted to take 1 clock cycle to

Byte address	Contents at each byte	Machine code
201	Opcode containing `addl3`	$c1_{hex}$
202	Index mode specifier for `[r4]`	44_{hex}
203	Register indirect mode specifier for `(r3)`	63_{hex}
204	Word displacement mode specifier using `r2` as base	$c2_{hex}$
205	The 16-bit constant 737	$e1_{hex}$
206		02_{hex}
207	Register mode specifier for `r1`	51_{hex}

Figure K.58 The encoding of the VAX instruction `addl3 r1,737(r2),(r3)[r4]`, assuming it starts at address 201. To satisfy your curiosity, the right column shows the actual VAX encoding in hexadecimal notation. Note that the 16-bit constant 737_{ten} takes 2 bytes.

decode each operand; thus, this sophisticated instruction set architecture can result in higher clock cycles per instruction, even when using simple addresses.

VAX Operations

In keeping with its philosophy, the VAX has a large number of operations as well as a large number of addressing modes. We review a few here to give the flavor of the machine.

Given the power of the addressing modes, the VAX *move* instruction performs several operations found in other machines. It transfers data between any two addressable locations and subsumes load, store, register-register moves, and memory-memory moves as special cases. The first letter of the VAX data type (b, w, l, f, q, d, g, c in Figure K.56) is appended to the acronym `mov` to determine the size of the data. One special move, called *move address*, moves the 32-bit *address* of the operand rather than the data. It uses the acronym `mova`.

The arithmetic operations of MIPS are also found in the VAX, with two major differences. First, the type of the data is attached to the name. Thus, `addb`, `addw`, and `addl` operate on 8-bit, 16-bit, and 32-bit data in memory or registers, respectively; MIPS has a single add instruction that operates only on the full 32-bit register. The second difference is that to reduce code size the add instruction specifies the number of unique operands; MIPS always specifies three even if one operand is redundant. For example, the MIPS instruction

```
add         $1, $1, $2
```

takes 32 bits like all MIPS instructions, but the VAX instruction

```
addl2       r1, r2
```

uses `r1` for both the destination and a source, taking just 24 bits: 8 bits for the opcode and 8 bits each for the two register specifiers.

Number of Operations

Now we can show how VAX instruction names are formed:

$$(\text{operation})(\text{datatype})\binom{2}{3}$$

The operation `add` works with data types byte, word, long, float, and double and comes in versions for either 2 or 3 unique operands, so the following instructions are all found in the VAX:

```
addb2    addw2    addl2    addf2    addd2
addb3    addw3    addl3    addf3    addd3
```

Accounting for all addressing modes (but ignoring register numbers and immediate values) and limiting to just byte, word, and long, there are more than 30,000 versions of integer add in the VAX; MIPS has just 4!

Another reason for the large number of VAX instructions is the instructions that either replace sequences of instructions or take fewer bytes to represent a single instruction. Here are four such examples (* means the data type):

VAX operation	Example	Meaning
`clr*`	`clrl r3`	`r3 = 0`
`inc*`	`incl r3`	`r3 = r3 + 1`
`dec*`	`decl r3`	`r3 = r3 – 1`
`push*`	`pushl r3`	`sp = sp – 4; Memory[sp] = r3;`

The *push* instruction in the last row is exactly the same as using the move instruction with autodecrement addressing on the stack pointer:

```
movl –(sp), r3
```

Brevity is the advantage of `pushl`: It is 1 byte shorter since `sp` is implied.

Branches, Jumps, and Procedure Calls

The VAX branch instructions are related to the arithmetic instructions because the branch instructions rely on *condition codes*. Condition codes are set as a side effect of an operation, and they indicate whether the result is positive, negative, or zero or if an overflow occurred. Most instructions set the VAX condition codes according to their result; instructions without results, such as branches, do not. The VAX condition codes are N (Negative), Z (Zero), V (oVerflow), and C (Carry). There is also a *compare* instruction `cmp*` just to set the condition codes for a subsequent branch.

The VAX branch instructions include all conditions. Popular branch instructions include `beql`(=), `bneq`(≠), `blss`(<), `bleq`(≤), `bgtr`(>), and `bgeq`(≥), which do just what you would expect. There are also unconditional branches whose name is determined by the size of the PC-relative offset. Thus, `brb` (*branch byte*) has an 8-bit displacement, and `brw` (*branch word*) has a 16-bit displacement.

The final major category we cover here is the procedure *call and return* instructions. Unlike the MIPS architecture, these elaborate instructions can take dozens of clock cycles to execute. The next two sections show how they work, but we need to explain the purpose of the pointers associated with the stack manipulated by `calls` and `ret`. The *stack pointer*, `sp`, is just like the stack pointer in MIPS; it points to the top of the stack. The *argument pointer*, `ap`, points to the base of the list of arguments or parameters in memory that are passed to the procedure. The *frame pointer*, `fp`, points to the base of the local variables of the procedure that are kept in memory (the *stack frame*). The VAX call and return instructions manipulate these pointers to maintain the stack in proper condition

across procedure calls and to provide convenient base registers to use when accessing memory operands. As we shall see, call and return also save and restore the general-purpose registers as well as the program counter. Figure K.59 gives a further sampling of the VAX instruction set.

An Example to Put It All Together: swap

To see programming in VAX assembly language, we translate two C procedures, `swap` and `sort`. The C code for `swap` is reproduced in Figure K.60. The next section covers `sort`.

We describe the `swap` procedure in three general steps of assembly language programming:

1. Allocate registers to program variables.
2. Produce code for the body of the procedure.
3. Preserve registers across the procedure invocation.

The VAX code for these procedures is based on code produced by the VMS C compiler using optimization.

Register Allocation for swap

In contrast to MIPS, VAX parameters are normally allocated to memory, so this step of assembly language programming is more properly called "variable allocation." The standard VAX convention on parameter passing is to use the stack. The two parameters, `v[]` and `k`, can be accessed using register `ap`, the argument pointer: The address `4(ap)` corresponds to `v[]` and `8(ap)` corresponds to `k`. Remember that with byte addressing the address of sequential 4-byte words differs by 4. The only other variable is `temp`, which we associate with register `r3`.

Code for the Body of the Procedure swap

The remaining lines of C code in `swap` are

```
temp = v[k];
v[k] = v[k + 1];
v[k + 1] = temp;
```

Since this program uses `v[]` and `k` several times, to make the programs run faster the VAX compiler first moves both parameters into registers:

```
movl        r2, 4(ap)             ;r2 = v[]
movl        r1, 8(ap)             ;r1 = k
```

Instruction type	Example	Instruction meaning
Data transfers	Move data between byte, half-word, word, or double-word operands; * is data type	
	`mov*`	Move between two operands
	`movzb*`	Move a byte to a half word or word, extending it with zeros
	`mova*`	Move the 32-bit address of an operand; data type is last
	`push*`	Push operand onto stack
Arithmetic/logical	Operations on integer or logical bytes, half words (16 bits), words (32 bits); * is data type	
	`add*_`	Add with 2 or 3 operands
	`cmp*`	Compare and set condition codes
	`tst*`	Compare to zero and set condition codes
	`ash*`	Arithmetic shift
	`clr*`	Clear
	`cvtb*`	Sign-extend byte to size of data type
Control	Conditional and unconditional branches	
	`beql, bneq`	Branch equal, branch not equal
	`bleq, bgeq`	Branch less than or equal, branch greater than or equal
	`brb, brw`	Unconditional branch with an 8-bit or 16-bit address
	`jmp`	Jump using any addressing mode to specify target
	`aobleq`	Add one to operand; branch if result ≤ second operand
	`case_`	Jump based on case selector
Procedure	Call/return from procedure	
	`calls`	Call procedure with arguments on stack (see "A Longer Example: `sort`" on page K-76)
	`callg`	Call procedure with FORTRAN-style parameter list
	`jsb`	Jump to subroutine, saving return address (like MIPS `jal`)
	`ret`	Return from procedure call
Floating point	Floating-point operations on D, F, G, and H formats	
	`addd_`	Add double-precision D-format floating numbers
	`subd_`	Subtract double-precision D-format floating numbers
	`mulf_`	Multiply single-precision F-format floating point
	`polyf`	Evaluate a polynomial using table of coefficients in F format
Other	Special operations	
	`crc`	Calculate cyclic redundancy check
	`insque`	Insert a queue entry into a queue

Figure K.59 Classes of VAX instructions with examples. The asterisk stands for multiple data types: b, w, l, d, f, g, h, and q. The underline, as in `addd_`, means there are 2-operand (`addd2`) and 3-operand (`addd3`) forms of this instruction.

```
swap(int v[], int k)
{
   int temp;
   temp = v[k];
   v[k] = v[k + 1];
   v[k + 1] = temp;
}
```

Figure K.60 A C procedure that swaps two locations in memory. This procedure will be used in the sorting example in the next section.

Note that we follow the VAX convention of using a semicolon to start a comment; the MIPS comment symbol # represents a constant operand in VAX assembly language.

The VAX has indexed addressing, so we can use index k without converting it to a byte address. The VAX code is then straightforward:

```
movl      r3, (r2)[r1]          ;r3 (temp) = v[k]
addl3     r0, #1,8(ap)          ;r0 = k + 1
movl      (r2)[r1],(r2)[r0]     ;v[k] = v[r0] (v[k + 1])
movl      (r2)[r0],r3           ;v[k + 1] = r3 (temp)
```

Unlike the MIPS code, which is basically two loads and two stores, the key VAX code is one memory-to-register move, one memory-to-memory move, and one register-to-memory move. Note that the `addl3` instruction shows the flexibility of the VAX addressing modes: It adds the constant 1 to a memory operand and places the result in a register.

Now we have allocated storage and written the code to perform the operations of the procedure. The only missing item is the code that preserves registers across the routine that calls `swap`.

Preserving Registers across Procedure Invocation of swap

The VAX has a pair of instructions that preserve registers, `calls` and `ret`. This example shows how they work.

The VAX C compiler uses a form of callee convention. Examining the code above, we see that the values in registers `r0`, `r1`, `r2`, and `r3` must be saved so that they can later be restored. The `calls` instruction expects a 16-bit mask at the beginning of the procedure to determine which registers are saved: if bit *i* is set in the mask, then register *i* is saved on the stack by the `calls` instruction. In addition, `calls` saves this mask on the stack to allow the return instruction (`ret`) to restore the proper registers. Thus, the `calls` executed by the caller does the saving, but the callee sets the call mask to indicate what should be saved.

One of the operands for `calls` gives the number of parameters being passed, so that `calls` can adjust the pointers associated with the stack: the argument

pointer (ap), frame pointer (fp), and stack pointer (sp). Of course, calls also saves the program counter so that the procedure can return!

Thus, to preserve these four registers for swap, we just add the mask at the beginning of the procedure, letting the calls instruction in the caller do all the work:

```
.word ^m<r0,r1,r2,r3> ;set bits in mask for 0,1,2,3
```

This directive tells the assembler to place a 16-bit constant with the proper bits set to save registers r0 through r3.

The return instruction undoes the work of calls. When finished, ret sets the stack pointer from the current frame pointer to pop everything calls placed on the stack. Along the way, it restores the register values saved by calls, including those marked by the mask and old values of the fp, ap, and pc.

To complete the procedure swap, we just add one instruction:

```
ret         ;restore registers and return
```

The Full Procedure swap

We are now ready for the whole routine. Figure K.61 identifies each block of code with its purpose in the procedure, with the MIPS code on the left and the VAX code on the right. This example shows the advantage of the scaled indexed

MIPS versus VAX					
Saving register					
swap:	addi	$29,$29, –12	swap:	.word ^m<r0,r1,r2,r3>	
	sw	$2, 0($29)			
	sw	$15, 4($29)			
	sw	$16, 8($29)			
Procedure body					
	muli	$2, $5,4		movl	r2, 4(a)
	add	$2, $4,$2		movl	r1, 8(a)
	lw	$15, 0($2)		movl	r3, (r2)[r1]
	lw	$16, 4($2)		addl3	r0, #1,8(ap)
	sw	$16, 0($2)		movl	(r2)[r1],(r2)[r0]
	sw	$15, 4($2)		movl	(r2)[r0],r3
Restoring registers					
	lw	$2, 0($29)			
	lw	$15, 4($29)			
	lw	$16, 8($29)			
	addi	$29,$29, 12			
Procedure return					
	jr	$31		ret	

Figure K.61 MIPS versus VAX assembly code of the procedure swap in Figure K.60 on page K-74.

addressing and the sophisticated call and return instructions of the VAX in reducing the number of lines of code. The 17 lines of MIPS assembly code became 8 lines of VAX assembly code. It also shows that passing parameters in memory results in extra memory accesses.

Keep in mind that the number of instructions executed is not the same as performance; the fallacy on page K-81 makes this point.

Note that VAX software follows a convention of treating registers `r0` and `r1` as temporaries that are not saved across a procedure call, so the VMS C compiler does include registers `r0` and `r1` in the register saving mask. Also, the C compiler should have used `r1` instead of `8(ap)` in the `addl3` instruction; such examples inspire computer architects to try to write compilers!

A Longer Example: `sort`

We show the longer example of the `sort` procedure. Figure K.62 shows the C version of the program. Once again we present this procedure in several steps, concluding with a side-by-side comparison to MIPS code.

Register Allocation for `sort`

The two parameters of the procedure `sort`, `v` and `n`, are found in the stack in locations `4(ap)` and `8(ap)`, respectively. The two local variables are assigned to registers: `i` to `r6` and `j` to `r4`. Because the two parameters are referenced frequently in the code, the VMS C compiler copies the *address* of these parameters into registers upon entering the procedure:

```
   moval       r7,8(ap)       ;move address of n into r7
   moval       r5,4(ap)       ;move address of v into r5
```

It would seem that moving the *value* of the operand to a register would be more useful than its address, but once again we bow to the decision of the VMS C compiler. Apparently the compiler cannot be sure that `v` and `n` don't overlap in memory.

```
sort (int v[], int n)
{
    int i, j;
    for (i = 0; i < n; i = i + 1) {
    for (j = i – 1; j >= 0 && v[j] > v[j + 1]; j = j – 1)
      { swap(v,j);
    }
  }
}
```

Figure K.62 A C procedure that performs a bubble sort on the array v.

Code for the Body of the `sort` Procedure

The procedure body consists of two nested *for* loops and a call to `swap`, which includes parameters. Let's unwrap the code from the outside to the middle.

The Outer Loop

The first translation step is the first for loop:

```
for (i = 0; i < n; i = i + 1) {
```

Recall that the C for statement has three parts: initialization, loop test, and iteration increment. It takes just one instruction to initialize `i` to 0, the first part of the for statement:

```
clrl        r6              ;i = 0
```

It also takes just one instruction to increment `i`, the last part of the for:

```
incl        r6              ;i = i + 1
```

The loop should be exited if `i < n` is *false,* or said another way, exit the loop if `i ≥ n`. This test takes two instructions:

```
for1tst:    cmpl    r6,(r7);compare r6 and memory[r7] (i:n)
            bgeq    exit1  ;go to exit1 if r6 ≥ mem[r7] (i ≥ n)
```

Note that `cmpl` sets the condition codes for use by the conditional branch instruction `bgeq`.

The bottom of the loop just jumps back to the loop test:

```
            brb     for1tst        ;branch to test of outer loop
exit1:
```

The skeleton code of the first for loop is then

```
            clrl    r6      ;i = 0
for1tst:    cmpl    r6,(r7) ;compare r6 and memory[r7] (i:n)
            bgeq exit1      ;go to exit1 if r6 ≥ mem[r7] (i ≥ n)
                    ...
                    (body of first for loop)
                    ...
            incl    r6      ;i = i + 1
            brb     for1tst ;branch to test of outer loop
exit1:
```

The Inner Loop

The second for loop is

```
for (j = i – 1; j >= 0 && v[j] > v[j + 1]; j = j – 1) {
```

The initialization portion of this loop is again one instruction:

```
subl3       r4,r6,#1    ;j = i – 1
```

The decrement of j is also one instruction:

```
decl        r4      ;j = j – 1
```

The loop test has two parts. We exit the loop if either condition fails, so the first test must exit the loop if it fails (j < 0):

```
for2tst:blss   exit2       ;go to exit2 if r4 < 0 (j < 0)
```

Notice that there is no explicit comparison. The lack of comparison is a benefit of condition codes, with the conditions being set as a side effect of the prior instruction. This branch skips over the second condition test.

The second test exits if v[j] > v[j + 1] is false, or exits if v[j] ≤ v[j + 1]. First we load v and put j + 1 into registers:

```
movl      r3,(r5)   ;r3 = Memory[r5] (r3 = v)
addl3     r2,r4,#1  ;r2 = r4 + 1 (r2 = j + 1)
```

Register indirect addressing is used to get the operand pointed to by r5.

Once again the index addressing mode means we can use indices without converting to the byte address, so the two instructions for v[j] ≤ v[j + 1] are

```
cmpl     (r3)[r4],(r3)[r2]  ;v[r4] : v[r2] (v[j]:v[j + 1])
bleq      exit2             ;go to exit2 if v[j] ≤ v[j + 1]
```

The bottom of the loop jumps back to the full loop test:

```
brb         for2tst        # jump to test of inner loop
```

Combining the pieces, the second for loop looks like this:

```
           subl3   r4,r6, #1       ;j = i – 1
for2tst:   blss    exit2           ;go to exit2 if r4 < 0 (j < 0)
           movl    r3,(r5)         ;r3 = Memory[r5] (r3 = v)
           addl3   r2,r4,#1        ;r2 = r4 + 1 (r2 = j + 1)
           cmpl    (r3)[r4],(r3)[r2];v[r4] : v[r2]
           bleq     exit2          ;go to exit2 if v[j] ð [j+1]
                   ...
                   (body of second for loop)
                   ...
           decl    r4              ;j = j – 1
           brb     for2tst         ;jump to test of inner loop
exit2:
```

Notice that the instruction blss (at the top of the loop) is testing the condition codes based on the new value of r4 (j), set either by the subl3 before entering the loop or by the decl at the bottom of the loop.

The Procedure Call

The next step is the body of the second for loop:

```
swap(v,j);
```

Calling swap is easy enough:

```
calls       #2,swap
```

The constant 2 indicates the number of parameters pushed on the stack.

Passing Parameters

The C compiler passes variables on the stack, so we pass the parameters to swap with these two instructions:

```
pushl       (r5)    ;first swap parameter is v
pushl        r4     ;second swap parameter is j
```

Register indirect addressing is used to get the operand of the first instruction.

Preserving Registers across Procedure Invocation of sort

The only remaining code is the saving and restoring of registers using the callee save convention. This procedure uses registers r2 through r7, so we add a mask with those bits set:

```
.word ^m<r2,r3,r4,r5,r6,r7>; set mask for registers 2-7
```

Since ret will undo all the operations, we just tack it on the end of the procedure.

The Full Procedure sort

Now we put all the pieces together in Figure K.63. To make the code easier to follow, once again we identify each block of code with its purpose in the procedure and list the MIPS and VAX code side by side. In this example, 11 lines of the sort procedure in C become the 44 lines in the MIPS assembly language and 20 lines in VAX assembly language. The biggest VAX advantages are in register saving and restoring and indexed addressing.

Fallacies and Pitfalls

The ability to simplify means to eliminate the unnecessary so that the necessary may speak.

Hans Hoffman
Search for the Real (1967)

MIPS versus VAX

			Saving registers			
	sort:	addi	$29,$29, −36	sort:	.word	^m<r2,r3,r4,r5,r6,r7>
		sw	$15, 0($29)			
		sw	$16, 4($29)			
		sw	$17, 8($29)			
		sw	$18,12($29)			
		sw	$19,16($29)			
		sw	$20,20($29)			
		sw	$24,24($29)			
		sw	$25,28($29)			
		sw	$31,32($29)			
			Procedure body			
Move parameters		move	$18, $4		moval	r7,8(ap)
		move	$20, $5		moval	r5,4(ap)
Outer loop		add	$19, $0, $0		clrl	r6
	for1tst:	slt	$8, $19, $20	for1tst:	cmpl	r6,(r7)
		beq	$8, $0, exit1		bgeq	exit1
Inner loop		addi	$17, $19, −1		subl3	r4,r6,#1
	for2tst:	slti	$8, $17, 0	for2tst:		
		bne	$8, $0, exit2		blss	exit2
		muli	$15, $17, 4		movl	r3,(r5)
		add	$16, $18, $15			
		lw	$24, 0($16)			
		lw	$25, 4($16)		addl3	r2,r4,#1
		slt	$8, $25, $24		cmpl	(r3)[r4],(r3)[r2]
		beq	$8, $0, exit2		bleq	exit2
Pass parameters and call		move	$4, $18		pushl	(r5)
		move	$5, $17		pushl	r4
		jal	swap		calls	#2,swap
Inner loop		addi	$17, $17, −1		decl	r4
		j	for2tst		brb	for2tst
Outer loop	exit2:	addi	$19, $19, 1	exit2:	incl	r6
		j	for1tst		brb	for1tst
			Restoring registers			
	exit1:	lw	$15, 0($29)			
		lw	$16, 4($29)			
		lw	$17, 8($29)			
		lw	$18,12($29)			
		lw	$19,16($29)			
		lw	$20,20($29)			
		lw	$24,24($29)			
		lw	$25,28($29)			
		lw	$31,32($29)			
		addi	$29,$29, 36			
			Procedure return			
		jr	$31	exit1:	ret	

Figure K.63 MIPS32 versus VAX assembly version of procedure `sort` in Figure K.62 on page K-76.

Fallacy *It is possible to design a flawless architecture.*

All architecture design involves trade-offs made in the context of a set of hardware and software technologies. Over time those technologies are likely to change, and decisions that may have been correct at one time later look like mistakes. For example, in 1975 the VAX designers overemphasized the importance of code size efficiency and underestimated how important ease of decoding and pipelining would be 10 years later. And, almost all architectures eventually succumb to the lack of sufficient address space. Avoiding these problems in the long run, however, would probably mean compromising the efficiency of the architecture in the short run.

Fallacy *An architecture with flaws cannot be successful.*

The IBM 360 is often criticized in the literature—the branches are not PC-relative, and the address is too small in displacement addressing. Yet, the machine has been an enormous success because it correctly handled several new problems. First, the architecture has a large amount of address space. Second, it is byte addressed and handles bytes well. Third, it is a general-purpose register machine. Finally, it is simple enough to be efficiently implemented across a wide performance and cost range.

The Intel 8086 provides an even more dramatic example. The 8086 architecture is the only widespread architecture in existence today that is not truly a general-purpose register machine. Furthermore, the segmented address space of the 8086 causes major problems for both programmers and compiler writers. Nevertheless, the 8086 architecture—because of its selection as the microprocessor in the IBM PC—has been enormously successful.

Fallacy *The architecture that executes fewer instructions is faster.*

Designers of VAX machines performed a quantitative comparison of VAX and MIPS for implementations with comparable organizations, the VAX 8700 and the MIPS M2000. Figure K.64 shows the ratio of the number of instructions executed and the ratio of performance measured in clock cycles. MIPS executes about twice as many instructions as the VAX while the MIPS M2000 has almost three times the performance of the VAX 8700.

Concluding Remarks

The Virtual Address eXtension of the PDP-11 architecture . . . provides a virtual address of about 4.3 gigabytes which, even given the rapid improvement of memory technology, should be adequate far into the future.

William Strecker
"VAX-11/780—A Virtual Address Extension to the PDP-11 Family," *AFIPS Proc.,* National Computer Conference (1978)

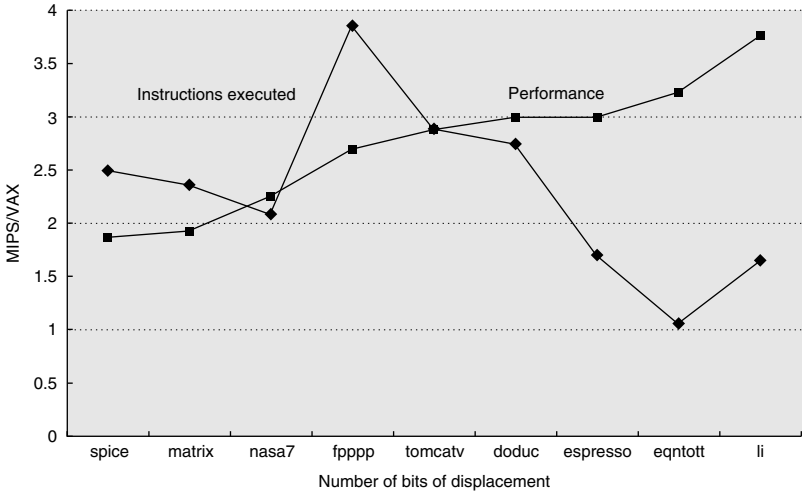

Figure K.64 Ratio of MIPS M2000 to VAX 8700 in instructions executed and performance in clock cycles using SPEC89 programs. On average, MIPS executes a little over twice as many instructions as the VAX, but the CPI for the VAX is almost six times the MIPS CPI, yielding almost a threefold performance advantage. (Based on data from "Performance from Architecture: Comparing a RISC and CISC with Similar Hardware Organization," by D. Bhandarkar and D. Clark, in *Proc. Symp. Architectural Support for Programming Languages and Operating Systems IV*, 1991.)

Program	Machine	Branch	Arithmetic/ logical	Data transfer	Floating point	Totals
gcc	VAX	30%	40%	19%		89%
	MIPS	24%	35%	27%		86%
spice	VAX	18%	23%	15%	23%	79%
	MIPS	4%	29%	35%	15%	83%

Figure K.65 The frequency of instruction distribution for two programs on VAX and MIPS.

We have seen that instruction sets can vary quite dramatically, both in how they access operands and in the operations that can be performed by a single instruction. Figure K.65 compares instruction usage for both architectures for two programs; even very different architectures behave similarly in their use of instruction classes.

A product of its time, the VAX emphasis on code density and complex operations and addressing modes conflicts with the current emphasis on easy decoding, simple operations and addressing modes, and pipelined performance.

With more than 600,000 sold, the VAX architecture has had a very successful run. In 1991, DEC made the transition from VAX to Alpha.

Orthogonality is key to the VAX architecture; the opcode is independent of the addressing modes, which are independent of the data types and even the number of unique operands. Thus, a few hundred operations expand to hundreds of thousands of instructions when accounting for the data types, operand counts, and addressing modes.

Exercises

K.1 [3] <K.4> The following VAX instruction decrements the location pointed to be register `r5`:

```
decl (r5)
```

What is the single MIPS instruction, or if it cannot be represented in a single instruction, the shortest sequence of MIPS instructions, that performs the same operation? What are the lengths of the instructions on each machine?

K.2 [5] <K.4> This exercise is the same as Exercise K.1, except this VAX instruction clears a location using autoincrement deferred addressing:

```
clrl @(r5)+
```

K.3 [5] <K.4> This exercise is the same as Exercise K.1, except this VAX instruction adds 1 to register `r5`, placing the sum back in register `r5`, compares the sum to register `r6`, and then branches to L1 if `r5 < r6`:

```
aoblss r6, r5, L1 # r5 = r5 + 1; if (r5 < r6) goto L1.
```

K.4 [5] <K.4> Show the single VAX instruction, or minimal sequence of instructions, for this C statement:

```
a = b + 100;
```

Assume `a` corresponds to register `r3` and `b` corresponds to register `r4`.

K.5 [10] <K.4> Show the single VAX instruction, or minimal sequence of instructions, for this C statement:

```
x[i + 1] = x[i] + c;
```

Assume `c` corresponds to register `r3`, `i` to register `r4`, and `x` is an array of 32-bit words beginning at memory location $4{,}000{,}000_{ten}$.

K.5 The IBM 360/370 Architecture for Mainframe Computers

Introduction

The term “computer architecture” was coined by IBM in 1964 for use with the IBM 360. Amdahl, Blaauw, and Brooks [1964] used the term to refer to the

programmer-visible portion of the instruction set. They believed that a family of machines of the same architecture should be able to run the same software. Although this idea may seem obvious to us today, it was quite novel at the time. IBM, even though it was the leading company in the industry, had five different architectures before the 360. Thus, the notion of a company standardizing on a single architecture was a radical one. The 360 designers hoped that six different divisions of IBM could be brought together by defining a common architecture. Their definition of *architecture* was

> . . . the structure of a computer that a machine language programmer must understand to write a correct (timing independent) program for that machine.

The term "machine language programmer" meant that compatibility would hold, even in assembly language, while "timing independent" allowed different implementations.

The IBM 360 was introduced in 1964 with six models and a 25:1 performance ratio. Amdahl, Blaauw, and Brooks [1964] discussed the architecture of the IBM 360 and the concept of permitting multiple object-code-compatible implementations. The notion of an instruction set architecture as we understand it today was the most important aspect of the 360. The architecture also introduced several important innovations, now in wide use:

1. 32-bit architecture
2. Byte-addressable memory with 8-bit bytes
3. 8-, 16-, 32-, and 64-bit data sizes
4. 32-bit single-precision and 64-bit double-precision floating-point data

In 1971, IBM shipped the first System/370 (models 155 and 165), which included a number of significant extensions of the 360, as discussed by Case and Padegs [1978], who also discussed the early history of System/360. The most important addition was virtual memory, though virtual memory 370s did not ship until 1972, when a virtual memory operating system was ready. By 1978, the high-end 370 was several hundred times faster than the low-end 360s shipped 10 years earlier. In 1984, the 24-bit addressing model built into the IBM 360 needed to be abandoned, and the 370-XA (eXtended Architecture) was introduced. While old 24-bit programs could be supported without change, several instructions could not function in the same manner when extended to a 32-bit addressing model (31-bit addresses supported) because they would not produce 31-bit addresses. Converting the operating system, which was written mostly in assembly language, was no doubt the biggest task.

Several studies of the IBM 360 and instruction measurement have been made. Shustek's thesis [1978] is the best known and most complete study of the 360/370 architecture. He made several observations about instruction set complexity that were not fully appreciated until some years later. Another important study of the 360 is the Toronto study by Alexander and Wortman [1975] done on an IBM 360 using 19 XPL programs.

System/360 Instruction Set

The 360 instruction set is shown in the following tables, organized by instruction type and format. System/370 contains 15 additional user instructions.

Integer/Logical and Floating-Point R-R Instructions

The * indicates the instruction is floating point, and may be either D (double precision) or E (single precision).

Instruction	Description
ALR	Add logical register
AR	Add register
A*R	FP addition
CLR	Compare logical register
CR	Compare register
C*R	FP compare
DR	Divide register
D*R	FP divide
H*R	FP halve
LCR	Load complement register
LC*R	Load complement
LNR	Load negative register
LN*R	Load negative
LPR	Load positive register
LP*R	Load positive
LR	Load register
L*R	Load FP register
LTR	Load and test register
LT*R	Load and test FP register
MR	Multiply register
M*R	FP multiply
NR	And register
OR	Or register
SLR	Subtract logical register
SR	Subtract register
S*R	FP subtraction
XR	Exclusive or register

Branches and Status Setting R-R Instructions

These are R-R format instructions that either branch or set some system status; several of them are privileged and legal only in supervisor mode.

Instruction	Description
BALR	Branch and link
BCTR	Branch on count
BCR	Branch/condition
ISK	Insert key
SPM	Set program mask
SSK	Set storage key
SVC	Supervisor call

Branches/Logical and Floating-Point Instructions—RX Format

These are all RX format instructions. The symbol "+" means either a word operation (and then stands for nothing) or H (meaning half word); for example, A+ stands for the two opcodes A and AH. The "*" represents D or E, standing for double- or single-precision floating point.

Instruction	Description
A+	Add
A*	FP add
AL	Add logical
C+	Compare
C*	FP compare
CL	Compare logical
D	Divide
D*	FP divide
L+	Load
L*	Load FP register
M+	Multiply
M*	FP multiply
N	And
O	Or
S+	Subtract
S*	FP subtract
SL	Subtract logical
ST+	Store
ST*	Store FP register
X	Exclusive or

Branches and Special Loads and Stores—RX Format

Instruction	Description
BAL	Branch and link
BC	Branch condition
BCT	Branch on count
CVB	Convert-binary
CVD	Convert-decimal
EX	Execute
IC	Insert character
LA	Load address
STC	Store character

RS and SI Format Instructions

These are the RS and SI format instructions. The symbol "*" may be A (arithmetic) or L (logical).

Instruction	Description
BXH	Branch/high
BXLE	Branch/low-equal
CLI	Compare logical immediate
HIO	Halt I/O
LPSW	Load PSW
LM	Load multiple
MVI	Move immediate
NI	And immediate
OI	Or immediate
RDD	Read direct
SIO	Start I/O
SL*	Shift left A/L
SLD*	Shift left double A/L
SR*	Shift right A/L
SRD*	Shift right double A/L
SSM	Set system mask
STM	Store multiple
TCH	Test channel
TIO	Test I/O
TM	Test under mask
TS	Test-and-set
WRD	Write direct
XI	Exclusive or immediate

SS Format Instructions

These are add decimal or string instructions.

Instruction	Description
AP	Add packed
CLC	Compare logical chars
CP	Compare packed
DP	Divide packed
ED	Edit
EDMK	Edit and mark
MP	Multiply packed
MVC	Move character
MVN	Move numeric
MVO	Move with offset
MVZ	Move zone
NC	And characters
OC	Or characters
PACK	Pack (Character → decimal)
SP	Subtract packed
TR	Translate
TRT	Translate and test
UNPK	Unpack
XC	Exclusive or characters
ZAP	Zero and add packed

360 Detailed Measurements

Figure K.66 shows the frequency of instruction usage for four IBM 360 programs.

Instruction	PLIC	FORTGO	PLIGO	COBOLGO	Average
Control	**32%**	**13%**	**5%**	**16%**	**16%**
BC, BCR	28%	13%	5%	14%	15%
BAL, BALR	3%			2%	1%
Arithmetic/logical	**29%**	**35%**	**29%**	**9%**	**26%**
A, AR	3%	17%	21%		10%
SR	3%	7%			3%
SLL		6%	3%		2%
LA	8%	1%	1%		2%
CLI	7%				2%
NI				7%	2%
C	5%	4%	4%	0%	3%
TM	3%	1%		3%	2%
MH			2%		1%
Data transfer	**17%**	**40%**	**56%**	**20%**	**33%**
L, LR	7%	23%	28%	19%	19%
MVI	2%		16%	1%	5%
ST	3%		7%		3%
LD		7%	2%		2%
STD		7%	2%		2%
LPDR		3%			1%
LH	3%				1%
IC	2%				1%
LTR		1%			0%
Floating point		**7%**			**2%**
AD		3%			1%
MDR		3%			1%
Decimal, string	**4%**			**40%**	**11%**
MVC	4%			7%	3%
AP				11%	3%
ZAP				9%	2%
CVD				5%	1%
MP				3%	1%
CLC				3%	1%
CP				2%	1%
ED				1%	0%
Total	**82%**	**95%**	**90%**	**85%**	**88%**

Figure K.66 Distribution of instruction execution frequencies for the four 360 programs. All instructions with a frequency of execution greater than 1.5% are included. Immediate instructions, which operate on only a single byte, are included in the section that characterized their operation, rather than with the long character-string versions of the same operation. By comparison, the average frequencies for the major instruction classes of the VAX are 23% (control), 28% (arithmetic), 29% (data transfer), 7% (floating point), and 9% (decimal). Once again, a 1% entry in the average column can occur because of entries in the constituent columns. These programs are a compiler for the programming language PL-I and runtime systems for the programming languages FORTRAN, PL/I, and Cobol.

K.6 Historical Perspective and References

Section L.4 (available online) features a discussion on the evolution of instruction sets and includes references for further reading and exploration of related topics.