Appendix K
Survey of Instruction Set Architectures

	ARMv8	MIPS64 R6	Power v3.0	RV64G	SPARCv9
Original date (base ISA)	1986	1986	1990	2016	1987
Date of this ISA	2011	2014	2013	2016	2008
Instruction size (bits)	32	32	32	32	32
Address space (size, model)	64 bits (flat)	64 bits, flat	64 bits, flat	64 bits, flat	64 bits, flat
Data alignment	Aligned preferred	Aligned preferred	Unaligned	Aligned preferred	Aligned
Data addressing modes	8 (including scaled, pre/post increment)	1 (+1 for FP only)	4	1	2
Integer registers (number, model, size)	31 GPR x 64, plus stack pointer	31 GPR × 64 bits	31 GPR × 64 bits	31 GPR × 64 bits	31 GPR × 64 bits
Separate floating-point registers	32x32 or 32x64 bits	32 × 32 or 32 × 64 bits	32 × 32 or 32 × 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 most recent version of five architectures for desktop, server, and PMD use (all had earlier versions). 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.29. In ARMv8, register 31 is a 0 (like register 0 in the other architectures), but when it is used in a load or store, it is the current stack pointer, a special purpose register. We can either think of SP-based addressing as a different mode (which is how the assembly mnemonics operate) or as simply a register + offset addressing mode (which is how the instruction is encoded).

	microMIPS64	RV64GC	Thumb-2
Date announced	2009	2016	2003
Instruction size (bits)	16/32	16/32	16/32
Address space (size, model)	32/64 bits, flat	32/64 bits flat	32 bits, flat
Data alignment	Aligned	Aligned, preferred	Aligned
Data addressing modes	2	1	6
Integer registers (number, model, size)	31 GPR × 64 bits	31 GPR × 64 bits	15 GPR × 32 bits
Integer registers accessible by most 16-bit instructions (which use should specifiers)	8 GPR + SP + GP +RA GPRs: 0, 2-7, 17, or 2-7, 16, 17	8 GPRs + SP GPRs: 8-15	8 GPR + SP × 32 bits

Figure K.2 Summary of three recent architectures for embedded applications. All three use 16-bit extensions of a base instruction set. Except for number of data address modes and a number of instruction set details, the integer instruction sets of these architectures are similar. Contrast this with Figure K.29. An earlier 16-bit version of the MIPS instruction set, called MIPS16, was created in 1995 and was replaced by microMIPS32 and microMIPS64. The first Thumb architecture had only 16-bit instructions and was created in 1996. Thumb-2 is built primarily on ARMv7, the 32-bit ARM instruction set; it offers 16 registers. RISC-V also defines RV32E, which has only 16 registers, includes the 16-bit instructions, and cannot have floating point. It appears that most implementations for embedded applications opt for RV32C or RV64GC.

	ARMv8	MIPS64 R6	Power v3.0	RV64G	SPARCv9
Register + offset (displacement or based)	B, H, W, D	B, H, W, D	B, H, W, D	B, H, W, D	B, H, W, D
Register + register (indexed)	B, H, W, D		B, H, W, D		B, H, W, D
Register + scaled register (scaled)	B, H, W, D	W,D			
Register + register + offset	B, H, W, D				
Register + offset & update register to effective address (based with update)	B, H, W, D		B, H, W, D		
Register & update register to register + offset (register with update)	B, H, W, D				
Register + Register & update register to effective address (indexed with update)	B, H, W, D		B, H, W, D		
PC-relative (PC + displacement)	W, D	W, D			

Figure K.3 Summary of data addressing modes supported by the desktop architectures, where B, H, W, D indicate what datatypes can use the addressing mode. Note that ARM includes two different types of address modes with updates, one of which is included in Power.

Register specifier	microMIPS64	RV64GC	Thumb-2
3-bit	2-7,16, 17	8-15	0-7
stack pointer register	29	2	0 (when used in load/store)
global pointer register	28		
return address register	31	1	14
Using special register	stack pointer or global pointer; 5-bit offset	stack pointer; 5-bit offset	stack pointer; 8-bit offset

Figure K.4 Register encodings for the 16-bit subsets of microMIPS64, RV64GC, and Thumb-2, including the core general purpose registers, and special-purpose registers accessible by some instructions.

Addressing mode	microMIPS64	RV64GC	Thumb-2
Register + offset (displacement or based)	4-bit offset, one of 8 registers	5-bit offset, one of 8 registers	5-bit offset, one of 8 registers
PC-relative data			word only; 8-bit offset
Using special register	stack pointer or global pointer; 5-bit offset	stack pointer; 5-bit offset	stack pointer; 8-bit offset

Figure K.5 Summary of data addressing modes supported by the embedded architectures. microMIPS64, RV64c, and Thumb-2 show only the modes supported in 16-bit instruction formats. The stack pointer in RV64GC and microMIPS64 is a designed GPR; it is another version of r31 is Thumb-2. In microMIPS64, the global pointer is register 30 and is used by the linkage convention to point to the global variable data pool. Notice that typically only 8 registers are accessible as base registers (and as we will see as ALU sources and destinations).

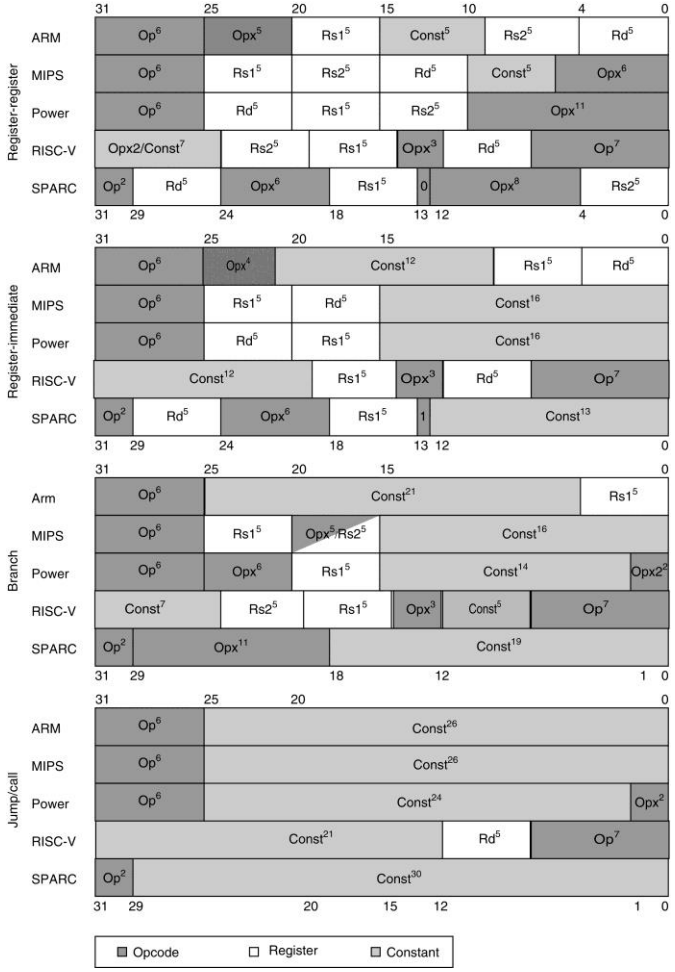

Figure K.6 Instruction formats for desktop/server RISC architectures. These four formats are found in all five architectures. (The superscript notation in this figure means the width of a field in bits.) Although the register fields are located in similar pieces of the instruction, be aware that the destination and two source fields are sometimes 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, address, mask, or sift amount). Although the labels on the instruction formats tell where various instructions are encoded, there are variations. For example, loads and stores, both use the ALU immediate form in MIPS. In RISC-V, loads use the ALU immediate format, while stores use the branch format.

Architecture	Additional instruction formats	Format function and use
ARMv8	At least 10 (many small variations); major forms are shown.	Logical immediates with 13-bit immediate field.
		Shifts with constant amount.(16-bit opcode)
		16-bit immediate form
		Exclusive operations: three register fields
		Branch register: long opcode
		Load/store with address mode bits.
MIPS64	1	A PC-relative set of load/stores using register-immediate format but with 18-bit immediates (since the other source is the PC).
Power	9 (not including a number of small variations or the vector extensions)	DQ-mode: uses the ALU immediate form but takes four bits of the displacement for other functions.
		DS-mode: uses the ALU immediate form but takes two bits of the displacement for other functions.
		DX-form: Like register-immediate, but with a register-source replaced by PC.
		MD, MDS formats: like register-register but used for shifts and rotates.
		X, XS, and several minor variations: used for indexed addressing modes, shifts, and a variety of extended purposes.
		Z22, Z23 formats: used for manipulating floating point numbers
RV64	2	SB format: a variant of the branch format with different immediate treatment
		UJ format: a variant of the jump/call format with different immediate treatment
SPARC	3	Another format for conditional branches containing 3 more bits of displacement (22 total versus 19) but no prediction hints.
		A format with 22-bit immediate used to load the upper half of a register,
		A format for conditional branches based on a register compare with zero.

Figure K.7 Other instruction formats beyond the four major formats of the previous figure. In some cases, there are formats very similar to one of the four core formats, but where a register field is used for other purposes. The Power architecture also includes a number of formats for vector operations.

Architecture	Opcode main: extended	Register specifiers x length	Immediate field length	Typical instructions
microMIPS64	6	none	10	Jumps
	6	1x5	5	Register-register operation (32 registers) and Load using SP as base register; any destination
	6	1x3	7	Branches equal/not equal zero. Loads using GP. as base.
	6:4	2x3		Register-register operation, rd/rs1, and rs2; 8 registers
	6:1	2x3	3	Register-register immediate, rd/rs1, and rs2; 8 registers
	6	2x3	4	Loads and stores; 8 registers
	6:4	2x3		Register-register operation, rd, and rs1; 8 registers
	6	2x5		Register-register operation; 32 registers.
RV64GC	2:3		11	Jumps
	2:3	1x3	7	Branch
	2:3	1x3	8	Immediate one source register.
	2:3	1x5	6	Store using SP as base.
	2:3	1x5	6	ALU immediate and load using SP as base.
	2:4	2x5		Register-register operation
	2:3	2x3	5	Loads and stores using 8 registers.
Thumb-2	3:2	2x3	5	Shift, move, load/store word/byte
	3:2	1x3	8	immediates: add, subtract, move, and compare
	4:1	1x3	8	Load/store with stack pointer as base, Add to SP or PC, Load/store multiple
	4:3	3x3		Load register indexed
	4:4		8	Conditional branch, system instruction
	4:12			Miscellaneous: 22 different instructions with 12 formats (includes compare and branch on zero, pop/push registers, adjust stack pointer, reverse bytes, IF-THEN instruction).
	5	1x3	8	Load relative to PC
	5		11	Unconditional branch
	6:1	3x3		Add/subtract
	6:3	1x4, 1x3		Special data processing
	6:4	2x3		Logical data processing
	6:6	1x4		Branch and change instruction set (ARM vs. Thumb)

Figure K.8 Instruction formats for the 16-bit instructions of microMIPS64, RV64GC, and Thumb-2. For instructions with a destination and two sources, but only two register fields, the instruction uses one of the registers as both source and destination. Note that the extended opcode field (or function field) and immediate field sometimes overlap or are identical. For RV64GC and microMIPS64, all the formats are shown; for Thumb-2, the Miscellaneous format includes 22 instructions with 12 slightly different formats; we use the extended opcode field, but a few of these instructions have immediate or register fields.

Data transfer (instruction formats)	R-I	R-I	R-I, R-R	R-I	R-I, R-R
Instruction name	**ARMv8**	**MIPS64**	**Power**	**RV64G**	**SPARC**
Load byte signed/unsigned.	LDR_B	LB_	LBZ; EXTSB	LB_	LD_B
Load halfword signed, unsigned	LDR_H	LH_	LHA/LHZ	LH_	LD_H
Load word	LDRSW/LDR	LW_	LW_	LW_	LD_W
Load double	LDRX	LD	LD	LD	LDD
Load float register SP/DP	LD_	L_C1	LF_	FL_	LD_F
Store byte	STB	SB	STB	SB	STB
Store half word	STW	SH	STH	STH	STH
Store word	STL	SW	STW	SW	ST
Store double word	STX	SD	SD	SD	STD
Store float SP/DP	ST_	S_C1	STF_	FS_	ST_F
Load reserved	LDEXB,LDEXH LDEXW,LDEXD	LL, LLD	lwarx, ldarx,	LR	
Store conditional	STEXB, STEXH, STEXW, STEXD	SC, SCD	stwcx, stdcx	SC	
Read/write spec. register	MF_, MT_	MF, MT_	M_SPR,	csrr_, csrr_i,	RD__,WR__
Move integer to FP register	ITOFS	MFC1/ DMFC1	STW; LDFS	STW; FLDWX	ST; LDF
Move FP to integer register	FTTOIS	MTC1/ DMTC1	STFS; LW	FSTWX; LDW	STF; LD
Synchronize data, instruction stream	DSB ISB	SYNC, SYNCI	SYNC, ISYNC	Fence Fence.i	MEMBAR FLUSH
Atomic operations	LDWAT, LDDAT STWAT, STDAT	LLWP, LLDP, SCWP, SCDP		AMOSWAP.W/D, AMOADD,W.D, AMOAND.W/D, AMOXOR.W/D, AMOOR.W/D, AMOMIN_.W/D, AMOMAX_.W/D	CASA, SWAP, LDSTUB

Figure K.9 Desktop RISC data transfer instructions equivalent to RV64G core. A sequence of instructions to synthesize a RV64G instruction is shown separated by semicolons. The MIPS and Power instructions for atomic operations load and conditionally store a pair of registers and can be used to implement the RV64G atomic operations with at most one intervening ALU instruction. The SPARC instructions: compare-and-swap, swap, LDSTUB provide atomic updates to a memory location and can be used to build the RV64G instructions. The Power3 instructions provide all the functionality, as the RV64G instructions, depending on a function field.

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 v8**	**MIPS64**	**Power v3**	**RISC-V**	**SPARC v.9**
Add word, immediate	ADD, ADDI	ADDU, ADDUI,	ADD, ADDI	ADDW, ADDWI	ADD
Add double word	ADDX	DADDU, DADDUI	ADD, ADDI	ADD, ADDI	ADD
Subtract	SUB, SUBI	SUBU, SUBI	SUBF	SUBW, SUBWI	SUB
Subtract double word	SUBX	DSUBU, DSUBUI	SUBF	SUB,SUBI	SUB
Multiply	MUL, SMUL	MUL, MULU, DMUL, DMULU	MULLW, MULLI	MUL, MULU, MULW, MULWU	MULX
Divide	MULX, SMULX	DIV, DIVU,DDIV, DDIVU	DIVW	DIV, DIVU, DIVW, DIVWU	DIVX
Remainder		MOD, MODU, DMOD, DMODU	MODSW, MODUW	REM, REMU, REMW, REMWU	
And	AND, ANDI	AND, ANDI	AND, ANDI	AND, ANDI	AND
Or	OR, ORI	OR, ORI	OR, ORI	OR, ORI	OR
Xor	XOR, XORI	XOR, XORI	XOR, XORI	XOR, XORI	XOR
Load bits 31..16	MOV	LUI	ADDIS	ADDIS	SETHI (Bfmt.)
Load upper bits of PC	ADR	ADDIUPC	ADDPCIS	AUIPC	
Shift left logical, double word and word versions, immediate and variable	LSL	SLLV, SLL	RLWINM	SLL, SLLI, SLLW, SLLWI	SLL
Shift right logical, double word and word version, immediate and variables	RSL	SRLV, SRL	RLWINM 32-i	SRL, SRLI, SRLW, SRLWI	SRL
Shift right arithmetic, double word and word versions, immediate and variable	RSA	SRAV, SRA	SRAW	SRA, SRAI, SRAW, SRAWI	SRA
Compare	CMP	SLT/U, SLTI/U	CMP(I)CLR	SLT/U, SLTI/U	SUBcc r0,...

Figure K.10 Desktop RISC arithmetic/logical instructions equivalent to RISC-V integer ISA. MIPS also provides instructions that trap on arithmetic overflow, which are synthesized in other architectures with multiple instructions. 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!)

Instruction name	ARMv8	MIPS64	PowerPC	RISC-V	SPARC v.9
Branch on integer compare	B.cond, CBZ, CBNZ	BEQ, BNE, B_Z (<, >, <=, >=) OR S***; BEZ	BC	BEQ, BNE, BLT, BGE, BLTU, BGEU	BR_Z, BPcc (<, >, <=, >=, =, not=)
Branch on floating-point compare	B.cond	BC1T, BC1F	BC	BEZ, BNZ	FBPfcc (<, >, <=, >=, =,...)
Jump, jump register	B, BR	J, JR	B, BCLR, BCCTR	JAL, JALR (with x0)	BA, JMPL r0,...
Call, call register	BL, BLR	JAL, JALR	BL, BLA, BCLRL, BCCTRL	JAL, JALR	CALL, JMPL
Trap	SVC, HVC, SMC	BREAK	TW, TWI	ECALL	Ticc, SIR
Return from interrupt	ERET	JR; ERET	RFI	EBREAK	DONE, RETRY, RETURN

Figure K.11 Desktop RISC control instructions equivalent to RV64G.

	ARMv8	MIPS64	PowerPC	RISC-V	SPARC v.9
Number of condition code bits (integer and FP)	16 (8 + the inverse)	none	8 × 4 both	none	2 × 4 integer, 4 × 2 FP
Basic compare instructions (integer and FP)	1 integer; 1 FP	1 integer, 1 FP	4 integer, 2 FP	2 integer; 3 FP	1 FP
Basic branch instructions (integer and FP)	1	2 integer, 1 FP	1 both	4 integer (used for FP as well)	3 integer, 1 FP
Compare register with register/ constant and branch	—	=, not=	—	=, not =, >=, <	—
Compare register to zero and branch	—	=, not=, <, <=, >, >=	—	=, not=, <, <=, >, >=	=, not=, <, <=, >, >=

Figure K.12 Summary of five desktop RISC approaches to conditional branches. Integer compare on SPARC is synthesized with an arithmetic instruction that sets the condition codes using r0 as the destination.

Floating point (instruction formats)	R-R	R-R	R-R	R-R	R-R
Instruction name	**ARMv8**	**MIPS64**	**PowerPC**	**RISC-V**	**SPARC v.9**
Add single, double	FADD	ADD.*	FADD*	FADD.*	FADD*
Subtract single, double	FSUB	SUB.*	FSUB*	FSUB.*	FSUB*
Multiply single, double	FMUL	MUL.*	FMUL*	FMUL.*	FMUL*
Divide single, double	FDIV	DIV.*	FDIV*	FDIV.*	FDIV*
Square root single, double	FSQRT	SQRT.*	FSQRT*	FSQRT.*	FSQRT*
Multiply add; Negative multiply add: single, double	FMADD, FNMADD	MADD.* NMAD.*	FMADD*, FNMADD*	FMADD.* FNMADD.*	
Multiply subtract single, double, Negative multiply subtract: single, double	FMSUB, FNMSUB	MSUB.*, NMSUB.*	FMSUB*, FNMSUB*	FMSUB.*, FNMSUB.*	
Copy sign or negative sign double or single to another FP register	FMOV, FNEG	FMOV.*, FNEG.*	FMOV*, FNEG*	FSGNJ.*, FSGNJN.*	FMOV*, FNEG*
Replace sign bit with XOR of sign bits single double	FABS	FABS.*	FABS*	FSGNJX.*	FABS*
Maximum or minimum single, double	FMAX, FMIN	MAX.*, MIN.*		FMAX.*, FMIN.*	
Classify floating point value single double		CLASS.*		FCLASS.*	
Compare	FCMP	CMP.*	FCMP*	FCMP.*	FCMP*
Convert between FP single or double and FP single or double, OR integer single or double, signed and unsigned with rounding	FCVT	CVT, CEIL, FLOOR		FCVT	F*TO*

Figure K.13 Desktop RISC floating-point instructions equivalent to RV64G ISA with an empty entry meaning that the instruction is unavailable. ARMv8 uses the same assembly mnemonic for single and double precision; the register designator indicates the precision. "*" is used as an abbreviation for S or D. For floating point compares all conditions: equal, not equal, less than, and less-then or equal are provided. Moves operate in both directions from/to integer registers. Classify sets a register based on whether the floating point quantity is plus or minus infinity, denorm, +/ − 0, etc.). The sign-injection instructions take two operands, but are primarily used to form floating point move, negate, and absolute value, which are separate instructions in the other ISAs.

Instruction name	microMIPS64 rs1;rs2/dst; offset	RV64GC rs1;rs2/dst; offset	Thumb-2 rs1;rs2/dst; offset
Load word	8;8;4	8;8;5	8;8;5
Load double word		8;8;5	
Load word with stack pointer as base register	1;32;5	1;32;6	1;3;8
Load double word with stack pointer as base register		1;32;6	
Store word	8;8;4	8;8;5	8;8;5
Store double word		8;8;5	
Store word with stack pointer as base register	1;32;5	1;32;6	1;3;8
Store double with stack pointer as base register		1;32;6	

Figure K.14 Embedded RISC data transfer instructions equivalent to RV64GC 16-bit ISA; a blank indicates that the instruction is not a 16-bit instruction. Rather than show the instruction name, where appropriate, we show the number of registers that can the base register for the address calculation, followed by the number of registers that can be the destination for a load or the source for a store, and finally, the size of the immediate used for address calculation. For example: 8; 8; 5 for a load means that there are 8 possible base registers, 8 possible destination registers for the load, and a 5-bit offset for the address calculation. For a store, 8; 8; 5, specifies that the source of the value to store comes from one of 8 registers. Remember that Thumb-2 also has 32-bit instructions (although not the full ARMv8 set) and that RV64GC and microMIPS64 have the full set of 32-bit instructions in RV64I or MIPS64.

Instruction Name/Function	microMIPS64	RV64GC	Thumb-2
Load immediate	8;7	32;6	8;8
Load upper immediate		32;6	
add immediate	32;4	32;6	8;8;3
add immediate word (32 bits) & sign extend		32;6	
add immediate to stack pointer	1;9	1;6 (adds 16x imm.)	1;7
add immediate to stack pointer store in reg.	1;8;6	1;8;6 (adds 4x imm.)	
shift left/right logical	8;8;3 (shift amt.)	8;6(shift amt.)	8;8;5 (shift amt.)
shift right arithmetic		8;6(shift amt.)	8;8;5 (shift amt.)
AND immediate	8;8;4	8;6	8;8
move	32;32	32;32	16;16
add	8;8;8	32;32	8;8;8 16;16
AND, OR, XOR	8;8	8;8	8;8
subtract	8;8;8	8;8	8;8;8
add word, subtract word (32 bits) & sign extend		8;8	

Figure K.15 ALU instructions provided in RV64GC and the equivalents, if any, in the 16-bit instructions of microMIPS64 or Thumb-2. An entry shows the number of register sources/destinations, followed by the size of the immediate field, if it exists for that instruction. The add to stack pointer with scaled immediate instructions are used for adjusting the stack pointer and creating a pointer to a location on the stack. In Thumb, the add has two forms one with three operands from the 8-register subset (Lo) and one with two operands but any of 16-registers.

	microMIPS64	RV64GC	Thumb-2
Unconditional branch	10-bit offset	11-bit offset	11-bit offset
Unconditional branch and link		11-bit offset	11-bit offset
Unconditional branch to register w/wo link	any of 32 registers	any of 32 registers	
Compare register to zero (=/!=) and branch	8 registers; 7-bit offset	8 registers; 8-bit offset	no: but see caption

Figure K.16 Summary of three embedded RISC approaches to conditional branches. A blank indicates that the instruction does not exist. Thumb-2 uses 4 condition code bits; it provides a conditional branch that tests the 4-bit condition code and has a branch offset of 8 bits.

Function	Definition	ARMv8	MIPS64	PowerPC	SPARC v.9
Load/store multiple registers	Loads or stores 2 or more registers	Load pair, store pair		Load store multiple (<=31 registers),	
Cache manipulation and prefetch	Modifies status of a cache line or does a prefetch	Prefetch	CACHE, PREFETCH	Prefetch	Prefetch

Figure K.17 Data transfer instructions not found in RISC-V core but found in two or more of the five desktop architectures. SPARC requires memory accesses to be aligned, while the other architectures support unaligned access, albeit, often with major performance penalties. The other architectures do not require alignment, but may use slow mechanisms to handle unaligned accesses.MIPS provides a set of instructions to handle misaligned accesses: LDL and LDR (load double left and load double right instructions) work as a pair to load a misaligned word; the corresponding store instructions perform the inverse. The Prefetch instruction causes a cache prefetch, while CACHE provides limited user control over the cache state.

Name	Definition	ARMv8	MIPS64	PowerPC	SPARC v.9
Delayed branches	Delayed branches with/without cancellation		BEQ, BNE, BGTZ, BLEZ, BCxEQZ, BCxNEZ		BPcc, A, FPBcc, A
Conditional trap	Traps if a condition is true		TEQ, TNE, TGE, TLT, TGEU, TLTU	TW, TD, TWI, TDI	Tcc

Figure K.18 Control instructions not found in RV64G core but found in two or more of the other architectures. MIPS64 Release 6 has nondelayed and normal delayed branches, while SPARC v.9 has delayed branches with cancellation based on the static prediction.

Instruction class	Instruction name(s)	Function
ALU	Byte align	Take a pair of registers and extract a word or double word of bytes. Used to implement unaligned byte copies.
	Align Immediate to PC	Adds the upper 16 bits of the PC to an immediate shifted left 16 bits and puts the result in a register; Used to get a PC-relative address.
	Bit swap	Reverses the bits in each byte of a register.
	No-op and link	Puts the value of PC+8 into a register
	Logical NOR	Computes the NOR of 2 registers
Control transfer	Branch and Link conditional	Compares a register to 0 and does a branch if condition is true; places the return address in the link register.
	Jump indexed, Jump and link indexed	Adds an offset and register to get new PC, w/wo link address

Figure K.19 Additional instructions provided MIPS64 R6. In addition, there are several instructions for supporting virtual machines, most are privileged.

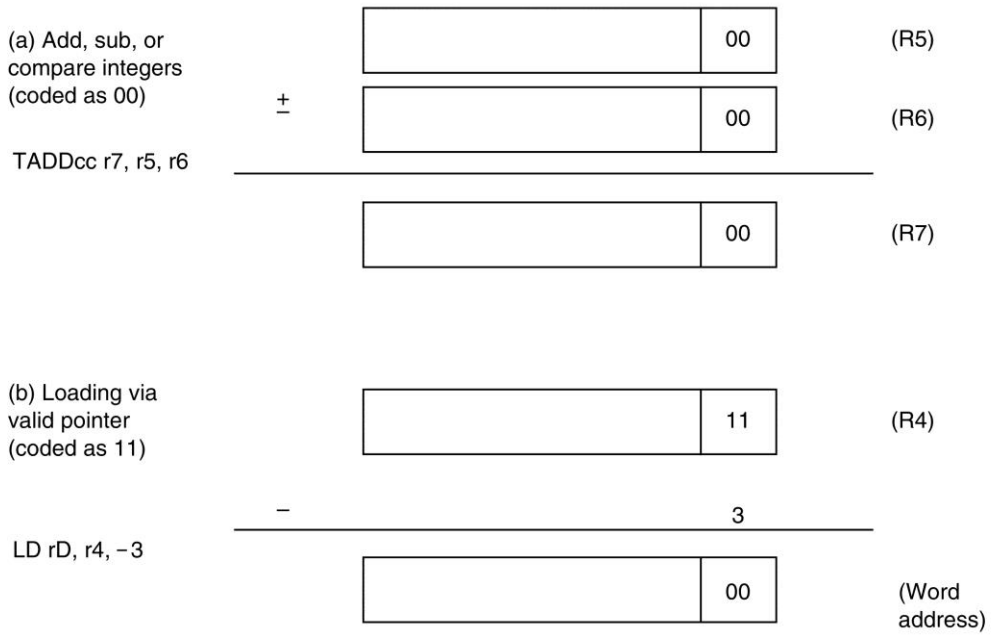

Figure K.20 SPARC uses the two least-significant bits to encode different data types for the tagged arithmetic instructions. (a) Integer arithmetic, which takes a single cycle as long as the operands and the result are integers. (b) The misaligned trap can be used to catch invalid memory accesses, such 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).

Instruction class	Instruction name(s)	Function
Data transfer	SAVE, RESTORE	Save or restore a register window
	Nonfaulting load	Version of load instructions that do not generate faults on address exceptions; allows speculation for loads.
ALU	Tagged add, Tagged subtract, with and without trap	Perform a tagged add/subtract, set condition codes, optionally trap.
Control transfer	Retry, Return, and Done	To provide handling for traps.
Floating Point Instructions	FMOVcc	Conditional move between FP registers based on integer or FP condition codes.

Figure K.21 Additional instructions provided in SPARCv9. Although register windows are by far the most significant distinction, they do not require many instructions!

Instruction class	Instruction name(s)	Function
Data transfer	Load/Store Non-temporal pair	Loads/stores a pair of registers with an indication not to cache the data. Base + scaled offset addressing mode only.
ALU	Add Extended word/double word	Add 2 registers after left shifting the second register operand and extending it.
	Add with shift; add immediate with shift	Adds with shift of the second operand.
	Address of page	Computes the address of a page based on PC (similar to ADDUIPC, which is the same as ADR in ARMv8)
	AND, OR, XOR, XOR NOT shifted register	Logical operation on a register and a shifted register.
	Bit field clear shifted	Shift operand, invert and AND with another operand
	Conditional compare, immediate, negative, negative immediate	If condition true, then set condition flags to compare result, otherwise leave condition flags untouched.
	Conditional increment, invert, negate	If condition then set destination to increment/invert/negate of source register
	CRC	Computes a CRC checksum: byte, word, halfword, double
	Multiply add, subtract	Integer multiply-add or multiply-subtract
	Multiply negate	Negate the product of two integers; word & double word
	Move immediate or inverse	Replace 16-bits in a register with immediate, possibly shifted
	Reverse bit order	Reverses the order of bits in a register
	Signed bit field move	Move a signed bit field; sign extend to left; zero extend to right
	Unsigned divide, multiple, multiply negate, multiply-add, multiply-sub	Unsigned versions of the basic instructions
Control transfer	CBNZ, CBZ	Compare branch =/!= 0, indicating this is not a call or return.
	TBNZ, TBZ	Tests bit in a register =/!= 0, and branch.

Figure K.22 Additional instructions provided in ARMv8, the AArch64 instruction set. Unless noted the instruction is available in a word and double word format, if there is a difference. Most of the ALU instructions can optionally set the condition codes; these are not included as separate instructions here or in earlier tables.

Instruction class	Instruction name(s)	Function
Data transfer	LHBRX, LWBRX, LDBRX	Loads a halfword/word/double word but reverses the byte order.
	SHBRX, SWBRX, SDBRX	Stores a halfword/word/double word but reverses the byte order
	LDQ, STQ	Load/store quadword to a register pair.
ALU	DRAN	Generate a random number in a register
	CMPB	Compares the individual bytes in a register and sets another register byte by byte.
	CMPRB	Compares a byte (x) against two other bytes (y and z) and sets a condition to indicate if the value of y<=x<=z.
	CRAND, CRNAND, CROR, CRNOR, CRXOR, CREQV, CORC, CRANDC	Logical operations on the condition register.
	ZCMPEQB	Compares a byte (x) against the eight bytes in another register and sets a condition to indicate if x = any of the 8 bytes
	EXTSWSL	Sign extend word and shift left
	POPCNTB, POPCNTW POPCNTD	Count number of 1s in each byte and place total in another byte. Count number of 1s in each word and place total in another word. Count number of 1s in a double word.
	PRTYD, PRTYW	Compute byte parity of the bytes in a word or double word.
	BPERMD	Permutes the bits in a double word, producing a permuted byte.
	CDTBCD, CDCBCD, ADDGCS	Instructions to convert from/to binary coded decimal (BCD) or operate on two BCD values
Control transfer	BA, BCA	Branches to an absolute address, conditionally & unconditionally
	BCCTR, BCCTRL	Conditional branch to address in the count register, w/wo linking
	BCTSAR, BCTARL	Conditional branch to address in the Branch Target Address register, w/wo linking
	CLRBHRB, MFBHRBE	Manipulate the branch history rolling buffer.
Floating Point Instructions	FRSQRTE	Computes an estimate of reciprocal of the square root,
	FTDIV, FTSQRT	Tests for divide by zero or square of negative number
	FSEL	Test register against zero and select one of two operands to move
	Decimal floating point operations	A series of 48 instructions to support decimal floating point.

Figure K.23 Additional instructions provided in Power3. Rotate instructions have two forms: one that sets a condition register and one that does not. There are a set of string instructions that load up to 32 bytes from an arbitrary address to a set of registers. These instructions will be phased out in future implementations, and hence we just mention them here.

	ARMv8	MIPS64 R6	Power v3.0	SPARCv9
Name of ISA extension	Advanced SIMD	MIPS64 SIMD Architecture	Vector Facility	VIS
Date of Current Version	2011	2012	2015	1995
Vector registers: # x size	32 x 128 bits	32 x 128 bits	32 x 128 bits	32 x 64 bits
Use GP/FP registers or independent set	extend FP registers doubling width	extend FP registers doubling width	Independent	Same as FP registers
Integer data sizes	8, 16, 32, 64	8, 16, 32, 64	8, 16, 32, 64, 128	8,16, 32
FP data sizes	32, 64	32, 64	32	
Immediates for integer and logical operations		5 bits arithmetic 8 bits logical		

Figure K.24 Structure of the SIMD extensions intended for multimedia support. In addition to the vector facility, The last row states whether the SIMD instruction set supports immediates (e.g, add vector immediate or AND vector immediate); the entry states the size of immediates for those ISAs that support them. Note that the fact that an immediate is present is encoded in the opcode space, and could alternatively be added to the next table as additional instructions. Power 3 has an optional Vector-Scalar Extension. The Vector-Scalar Extension defines a set of vector registers that overlap the FP and normal vector registers, eliminating the need to move data back and forth to the vector registers. It also supports double precision floating point operations.

Instruction category	ARM Advanced SIMD	MIPS SIMD	Power Vector Facility
Add/subtract	16B, 8H, 4W; 2 D	16B, 8H; 4W; 2 D	16B, 8H, 4W, 2 D, Q
Saturating add/sub	16B, 8H, 4W; 2 D	16B, 8H; 4W; 2 D	16B, 8H, 4W, 2 D, Q
Absolute value of difference	16B, 8H, 4W; 2 D	16B, 8H, 4W; 2 D	16B, 8H, 4W; 2 D; Q
Adjacent add & subtract (pairwise)	16B, 8H, 4W	16B, 8H, 4W	16B, 8H, 4W; 2 D
Average		16B, 8H, 4W; 2 D	16B, 8H, 4W; 2 D; Q
Dot product add, dot product subtract	16B, 8H, 4W	16B, 8H, 4W	16B, 8H, 4W; 2 D
Divide: signed, unsigned	16B, 8H, 4W	16B, 8H, 4W; 2 D	16B, 8H, 4W; 2 D; Q
Multiply: signed, unsigned	16B, 8H, 4W	16B, 8H, 4W	16B, 8H, 4W; 2 D
Multiply add, multiply subtract	16B, 8H, 4W	16B, 8H, 4W	16B, 8H, 4W; 2 D
Maximum, signed & unsigned	16B, 8H, 4W; 2 D	16B, 8H, 4W; 2 D	16B, 8H, 4W; 2 D; Q
Minimum, signed & unsigned	16B, 8H, 4W; 2 D	16B, 8H, 4W; 2 D	16B, 8H, 4W; 2 D; Q
Modulo, signed & unsigned		16B, 8H, 4W; 2 D	16B, 8H, 4W; 2 D; Q
Compare equal	16B, 8H, 4W; 2 D	16B, 8H, 4W; 2 D	16B, 8H, 4W; 2 D; Q
Compare <, <=, signed, unsigned	16B, 8H, 4W; 2 D	16B, 8H, 4W; 2 D	16B, 8H, 4W; 2 D; Q

Figure K.25 Summary of arithmetic SIMD instructions. B stands for byte (8 bits), H for half word (16 bits), and W for word (32 bits), D for double word (64 bits), and Q for quad word (128 bits). Thus, 8B means an operation on 8 bytes in a single instruction. Note that some instructions--such as adjacent add/subtract, or multiply--produce results that are twice the width of the inputs (e.g. multiply on 16 bytes produces 8 halfword results). Dot product is a multiply and accumulate. The SPARC VIS instructions are aimed primarily at graphics and are structured accordingly.

Instruction category	ARM Advanced SIMD	MIPS SIMD	Power Vector Facility
Shift right/left, logical, arithmetic	16B, 8H, 4W; 2 D	16B, 8H, 4W; 2 D; Q	16B, 8H, 4W; 2 D; Q
Count leading or trailing zeros	16B, 8H, 4W; 2 D	16B, 8H, 4W; 2 D	16B, 8H, 4W; 2 D; Q
and/or/xor	Q	Q	Q
Bit insert & extract	16B, 8H, 4W; 2 D	16B, 8H, 4W; 2 D	16B, 8H, 4W; 2 D; Q
Population count		16B, 8H, 4W; 2 D	16B, 8H, 4W; 2 D; Q
Interleave even/odd, left/right		16B, 8H, 4W; 2 D	6B, 8H, 4W; 2 D
Pack even/odd		16B, 8H, 4W; 2 D	6B, 8H, 4W; 2 D
Shuffle		16B, 8H, 4W; 2 D	16B, 8H, 4W; 2 D
SPLAT		16B, 8H, 4W; 2 D	16B, 8H, 4W; 2 D

Figure K.26 Summary of logical, bitwise, permute, and pack/unpack instructions, using the same format as the previous figure. When there is a single operand the instruction applies to the entire register; for logical operations there is no difference.Interleave puts together the elements (all even, odd, leftmost or rightmost) from two different registers to create one value; it can be used for unpacking. Pack moves the even or odd elements from two different registers to the leftmost and rightmost halves of the result. Shuffle creates a from two registers based on a mask that selects which source for each item. SPLAT copies a value into each item in a register.

Instruction category	ARM Advanced SIMD	MIPS SIMD	Power Vector Facility
FP add, subtract, multiply, divide	4W, 2D	4W, 2D	4W, 2D
FP multiply add/subtract	4W, 2D	4W, 2D	4W, 2D
FP maximum/minimum	4W, 2D	4W, 2D	4W, 2D
FP SQRT and 1/SQRT	4W, 2D	4W, 2D	4W, 2D
FP Compare	4W, 2D	4W, 2D	4W, 2D
FP Convert to/from integer	4W, 2D	4W, 2D	4W, 2D

Figure K.27 Summary of floating point, using the same format as the previous figure.

Function	Thumb-2	microMIPS32 DSP
Add/Subtract	4B, 2H	4B, 2Q15
Add /Subtract with saturation	4B, 2H	4B, 2Q15, Q31
Add/Subtract with Exchange (exchanges halfwords in rt, then adds first halfword and subtracts second) with optional saturation	2H	
Reduce by add (sum the values)		4B
Absolute value		2Q15, Q31
Precision reduce/increase (reduces or increases the precision of a value)		2B, Q15, 2Q15, Q31
Shifts: left, right, logical & arithmetic, with optional saturation		4B, 2H
Multiply	2H	2B, 2H, 2Q15
Multiply add/subtract (to GPR or accumulator register in MIPS)	2H	2Q15
Complex multiplication step (2 multiplies and addition/subtraction)	2H	2Q15
Multiply and accumulate (by addition or subtraction)	2H	Q15, Q31
Replicate bits		B, H
Compare: =, <, <=, sets condition field		4B, 2H
Pick (use condition bits to choose bytes or halfwords from two operands)		4B, 2H
Pack choosing a halfword from each operand		H
Extract		Q63
Move from/to accumulator		DW

Figure K.28 Summary of two embedded RISC DSP operations, showing the data types for each operation. A blank indicates that the operation is not supported as a single instruction. Byte quantities are usually unsigned. Complex multiplication step implements multiplication of complex numbers where each component is a Q15 value. ARM uses its standard condition register, while MIPS adds a set of condition bits as part of the state in the DSP extension.

	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.29 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.)

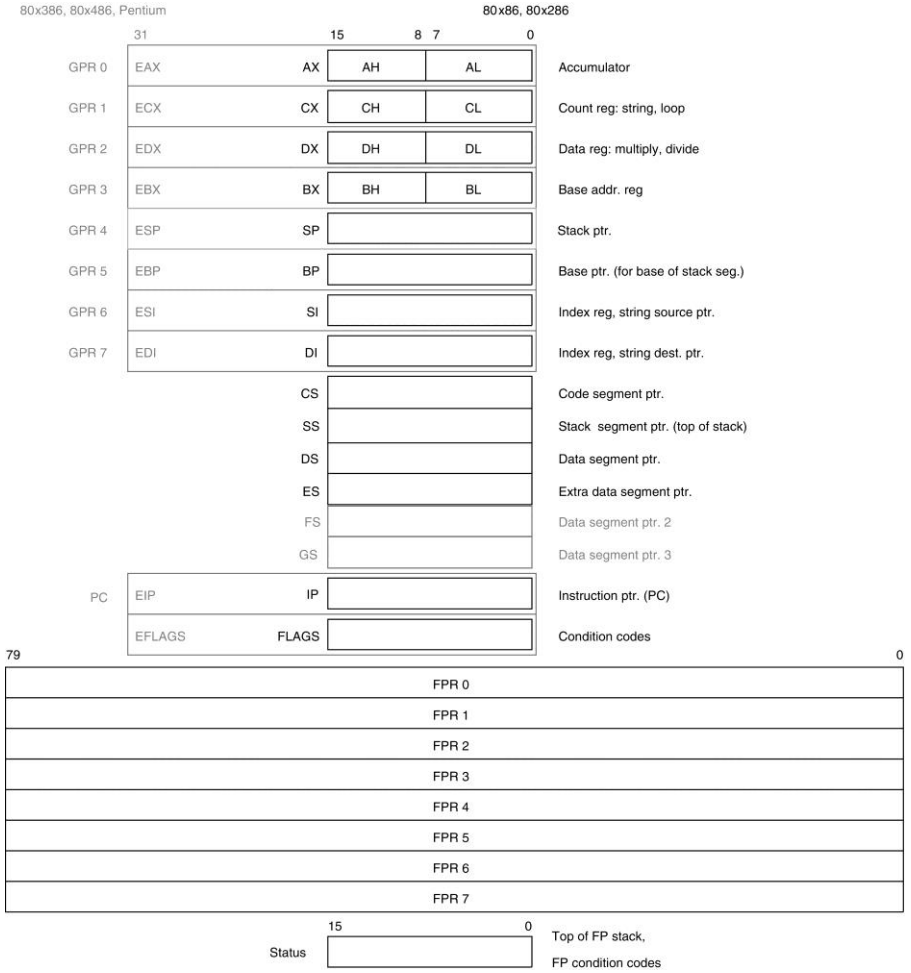

Figure K.30 The 80x86 has evolved over time, and so has its register set. The original set is shown in black and the extended set in gray. The 8086 divided the first four registers in half so that they could be used either as one 16-bit register or as two 8-bit registers. Starting with the 80386, the 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.31 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.30 (not IP or FLAGS).

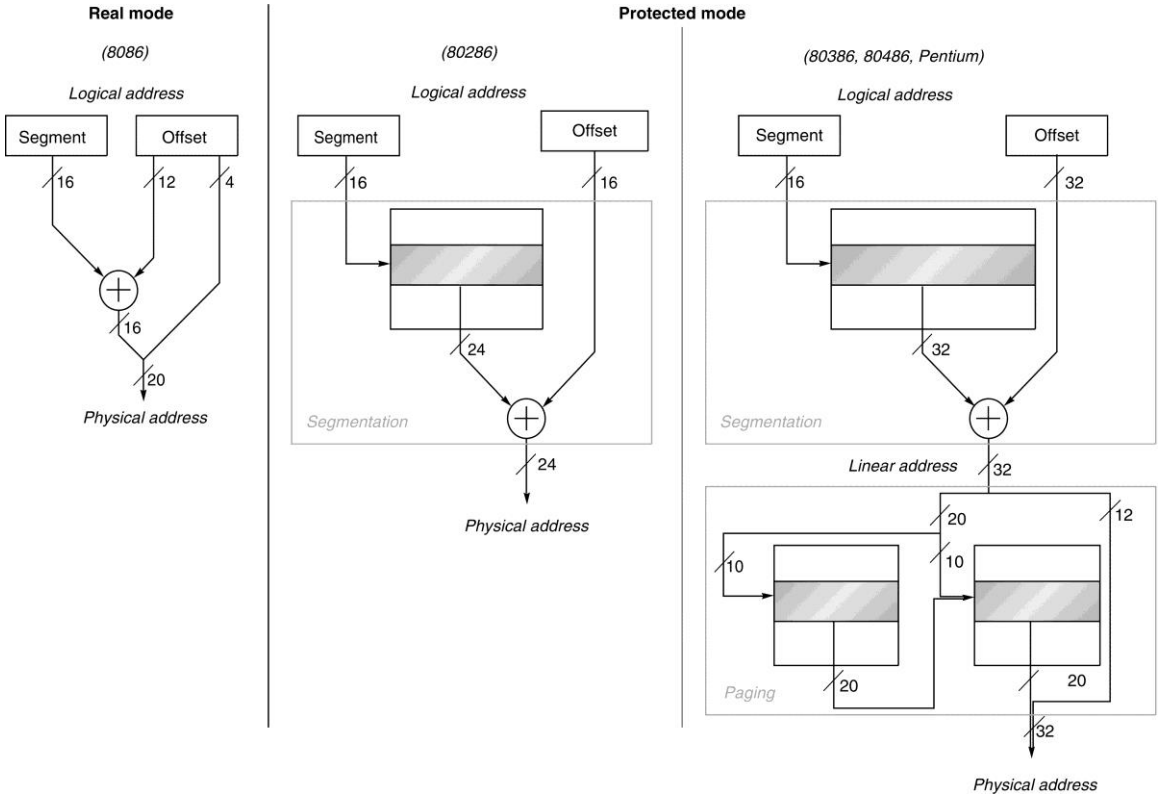

Figure K.32 The original segmented scheme of the 8086 is shown on the left. All 80x86 processors support this style of addressing, called *real mode*. It simply takes the contents of a segment register, shifts it left 4 bits, and adds it to the 16-bit offset, forming a 20-bit physical address. The 80286 (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.

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.33 Some typical 80x86 instructions and their functions. A list of frequent operations appears in Figure K.34. 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.

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.34 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.

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.35 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.

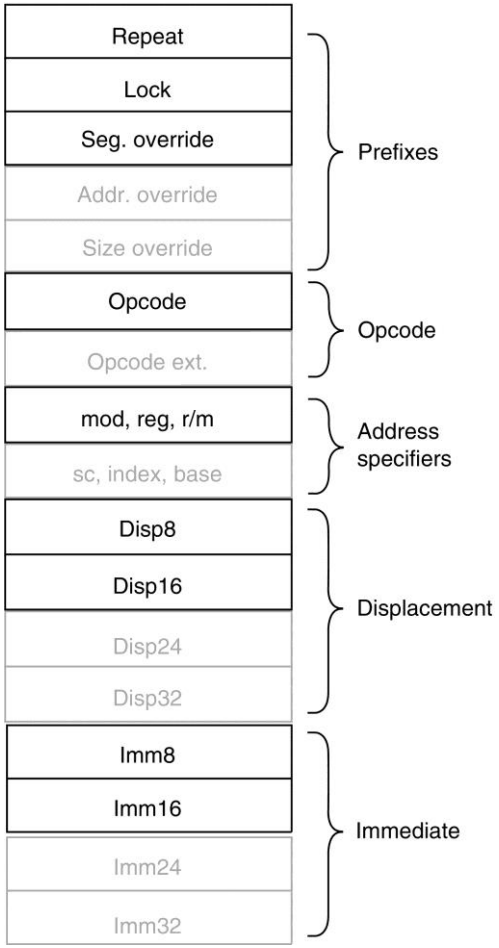

Figure K.36 The instruction format of the 8086 (black type) and the extensions for the 80386 (shaded type). Every field is optional except the opcode.

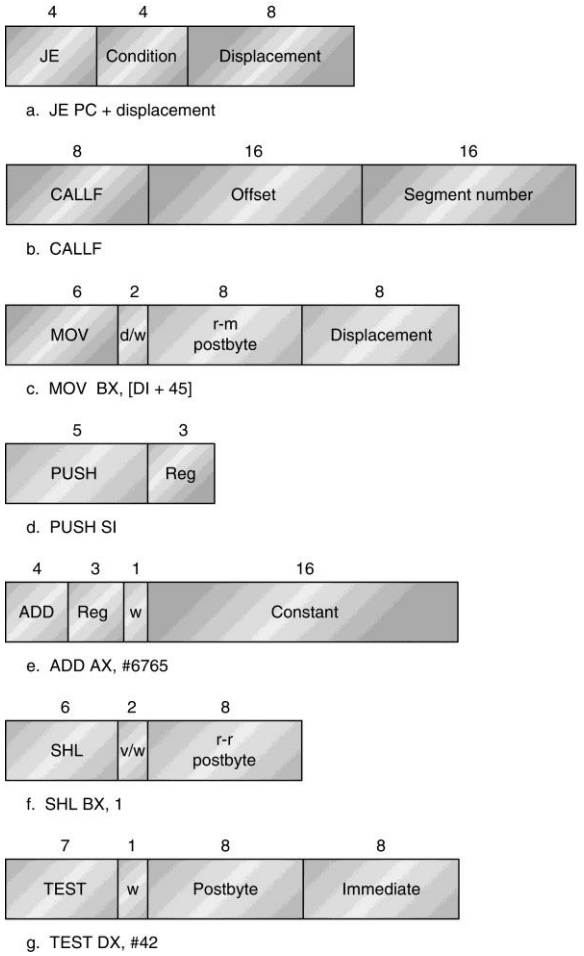

Figure K.37 Typical 8086 instruction formats. The encoding of the postbyte is shown in Figure K.38. Many instructions contain the 1-bit field w, which says whether the operation is a byte or a word. Fields of the form v/w or d/w are a d-field or v-field followed by the w-field. The d-field in `MOV` is 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.

		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	*+ disp 8*	*+ disp 8*	*+ disp1 6*	*+ disp3 2*	*field*
4	AH	SP	ESP	4	addr=SI	=(si)b	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.38 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.39. 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.39 Based plus scaled index mode address specifier found in the 80386. This mode is indicated by the (sib) notation in Figure K.38. 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^{Scale} × Index register.

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

Figure K.40 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.41 Operand addressing mode distribution by program. This chart does not include addressing modes used by branches or control instructions.

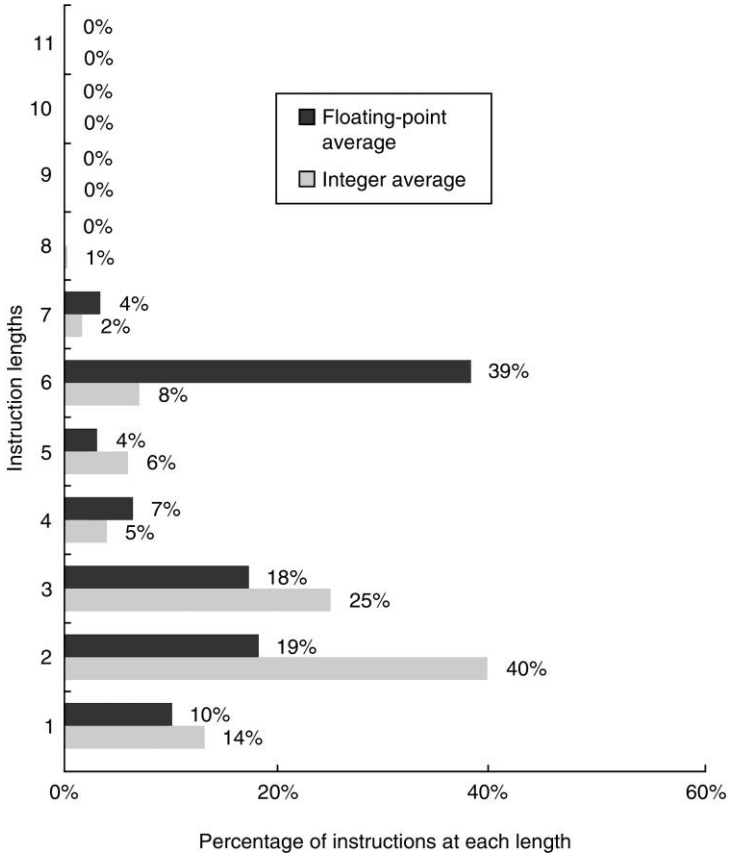

Figure K.42 Averages of the histograms of 80x86 instruction lengths for five SPECint92 programs and for five SPECfp92 programs, all running in 32-bit mode.

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.43 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.44 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.45 80x86 instruction mix for five SPECint92 programs.

	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.46 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.47 Instructions executed and data accesses for five SPECfp92 programs on 80x86 and ratio to DLX.

	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.48 Percentage of instructions executed by category for 80x86 and DLX for the averages of five SPECint92 and SPECfp92 programs of Figures K.46 and K.47.

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.49 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`.

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.50 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.

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.51 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.

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-33)
	`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.52 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.53 A C procedure that swaps two locations in memory. This procedure will be used in the sorting example in the next section.

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.54 MIPS versus VAX assembly code of the procedure swap in Figure K.53 on page K-30.

```
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.55 A C procedure that performs a bubble sort on the array v.

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.56 MIPS32 versus VAX assembly version of procedure `sort` in Figure K.55 on page K-33.

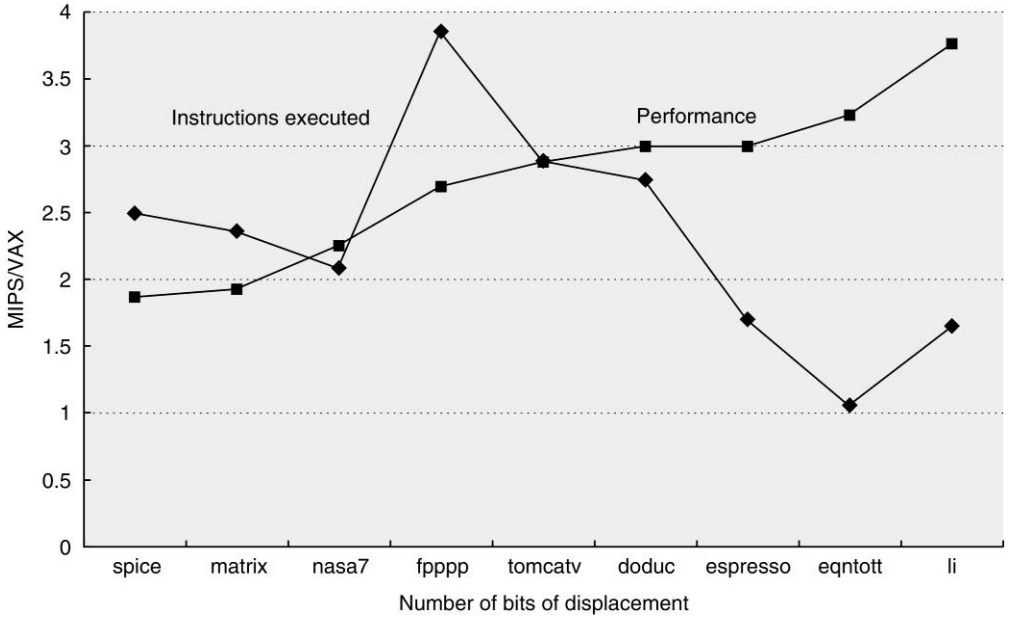

Figure K.57 Ratio of MIPS M2000 to VAX 8700 in instructions executed and performance in clock cycles using SPEC89 programs. On average, MIPS executes a little over twice as many instructions as the VAX, but the CPI for the VAX is almost six times the MIPS CPI, yielding almost a threefold performance 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.58 The frequency of instruction distribution for two programs on VAX and MIPS.

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.59 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.