

Appendix K

Authors: John Hennessy & David Patterson

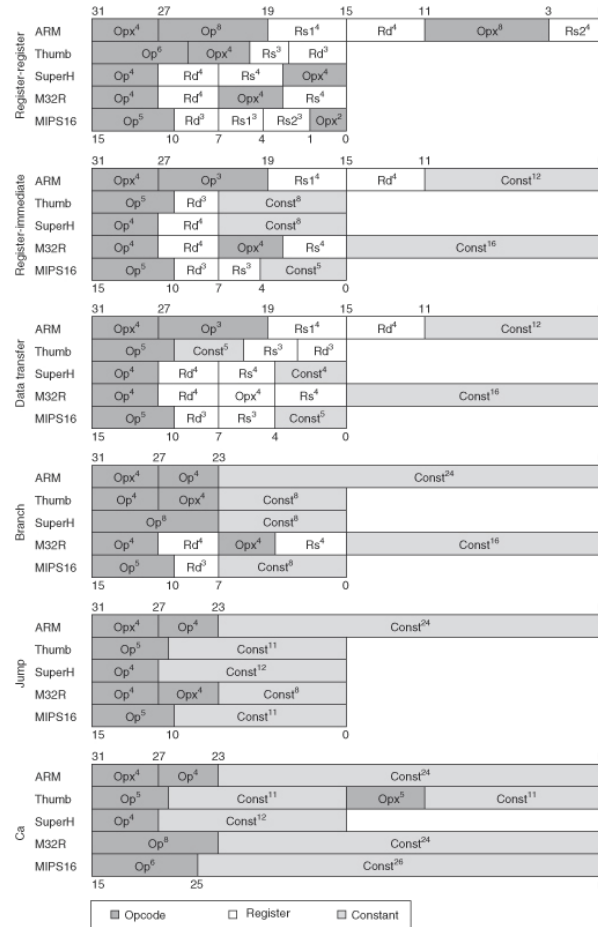


Figure K.6 Instruction formats for embedded RISC architectures. These six formats are found in all five architectures. The notation is the same as Figure K.5. Note the similarities in branch, jump, and call formats and the diversity in register-register, register-immediate, and data transfer formats. 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.

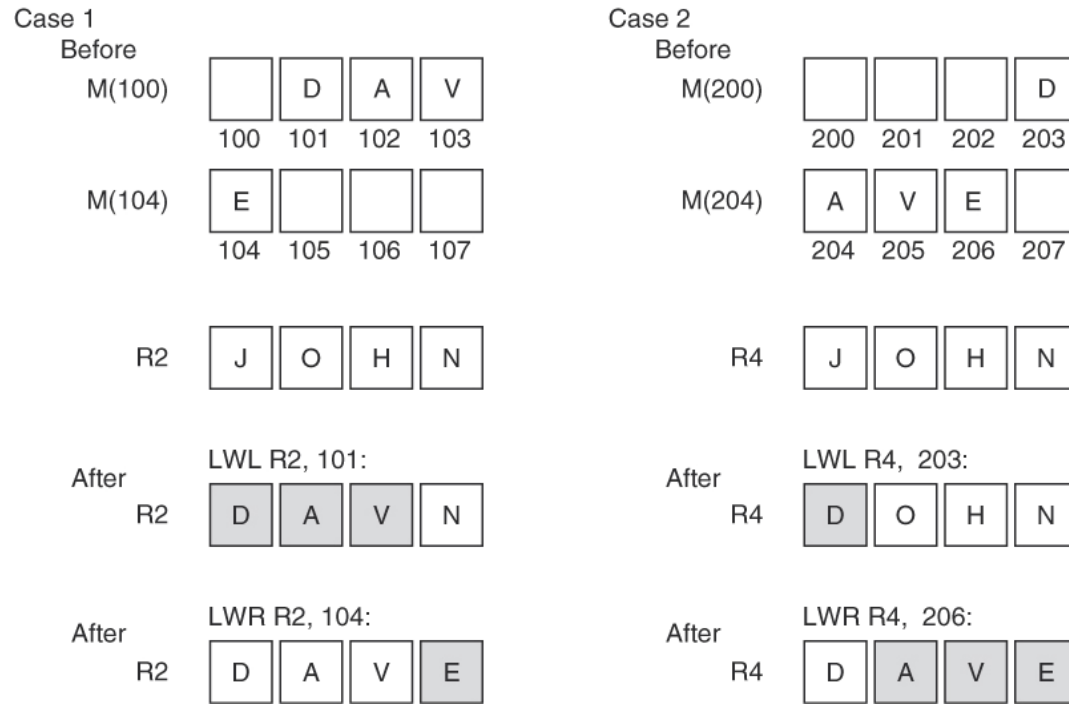


Figure K.31 MIPS instructions for unaligned word reads. This figure assumes operation in Big Endian mode. Case 1 first loads the 3 bytes 101, 102, and 103 into the left of R2, leaving the least-significant byte undisturbed. The following LWR simply loads byte 104 into the least-significant byte of R2, 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, load-store seem confusing, don't worry; they work!)

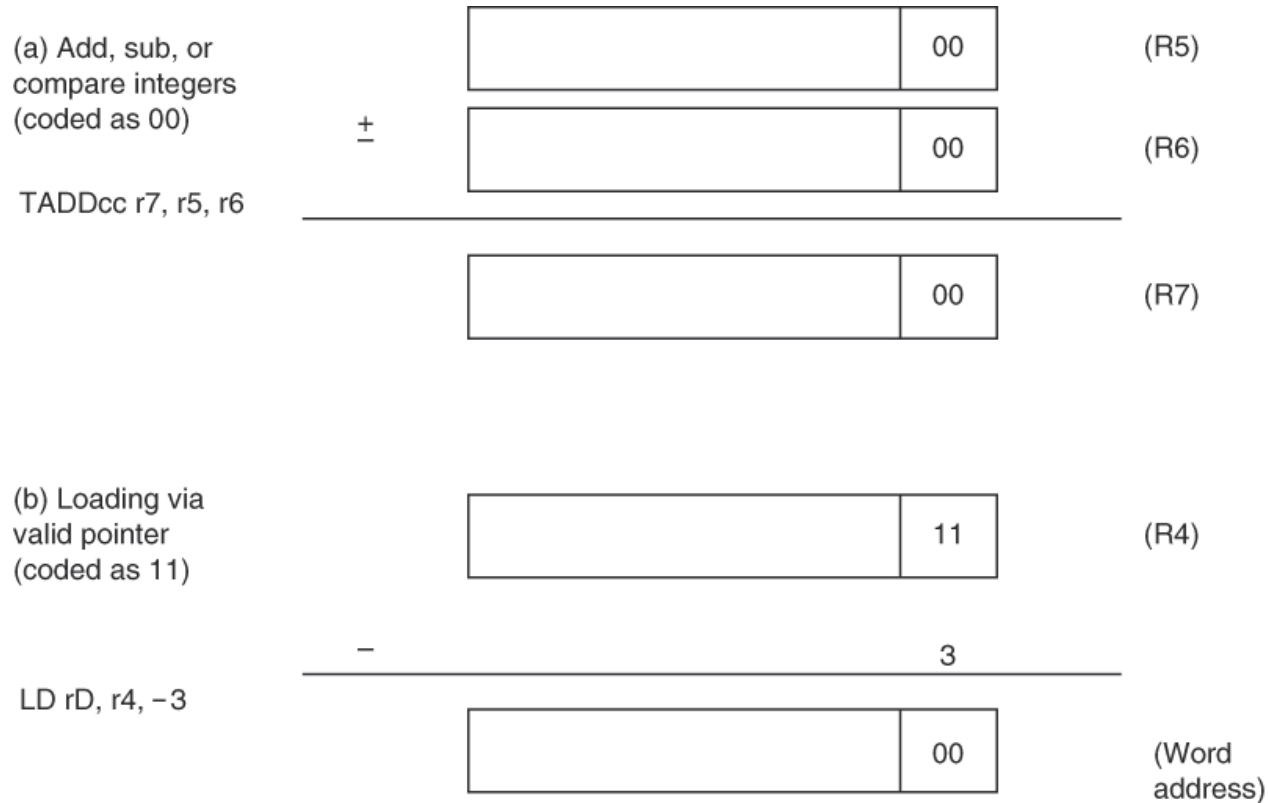


Figure K.32 SPARC uses the two least-significant bits to encode different data types for the tagged arithmetic instructions. (a) Integer arithmetic, which takes a single cycle as long as the operands and the result are integers. (b) The misaligned trap can be used to catch invalid memory accesses, such 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).

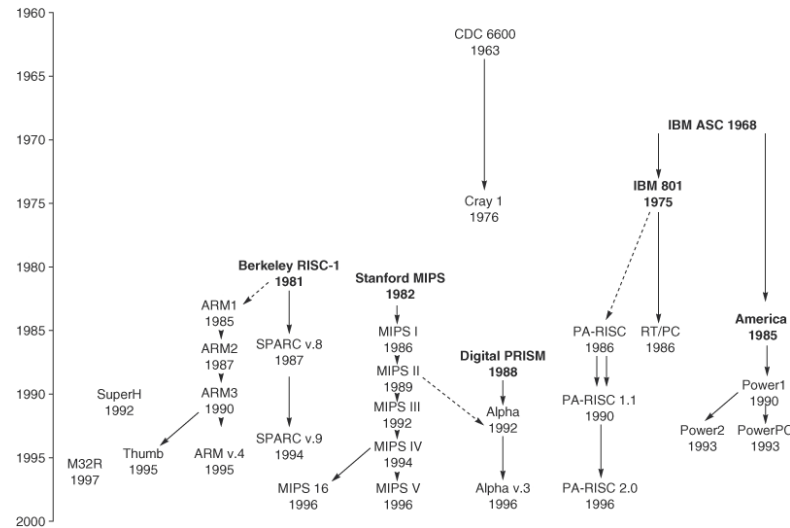


Figure K.35 The lineage of RISC instruction sets. Commercial machines are shown in plain text and research machines in bold. The CDC-6600 and Cray-1 were load-store machines with register 0 fixed at 0, and separate integer and floating-point registers. Instructions could not cross word boundaries. An 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.”

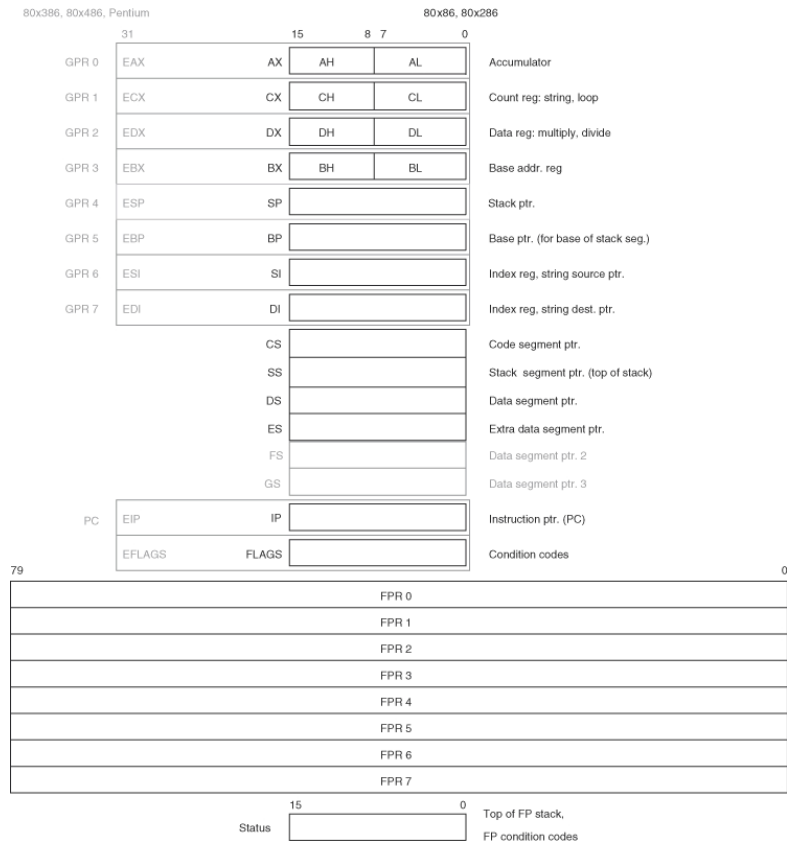


Figure K.37 The 80x86 has evolved over time, and so has its register set. The original set is shown in black and the extended set in gray. The 8086 divided the first four registers in half so that they could be used either as one 16-bit register or as two 8-bit registers. Starting with the 80386, the 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.

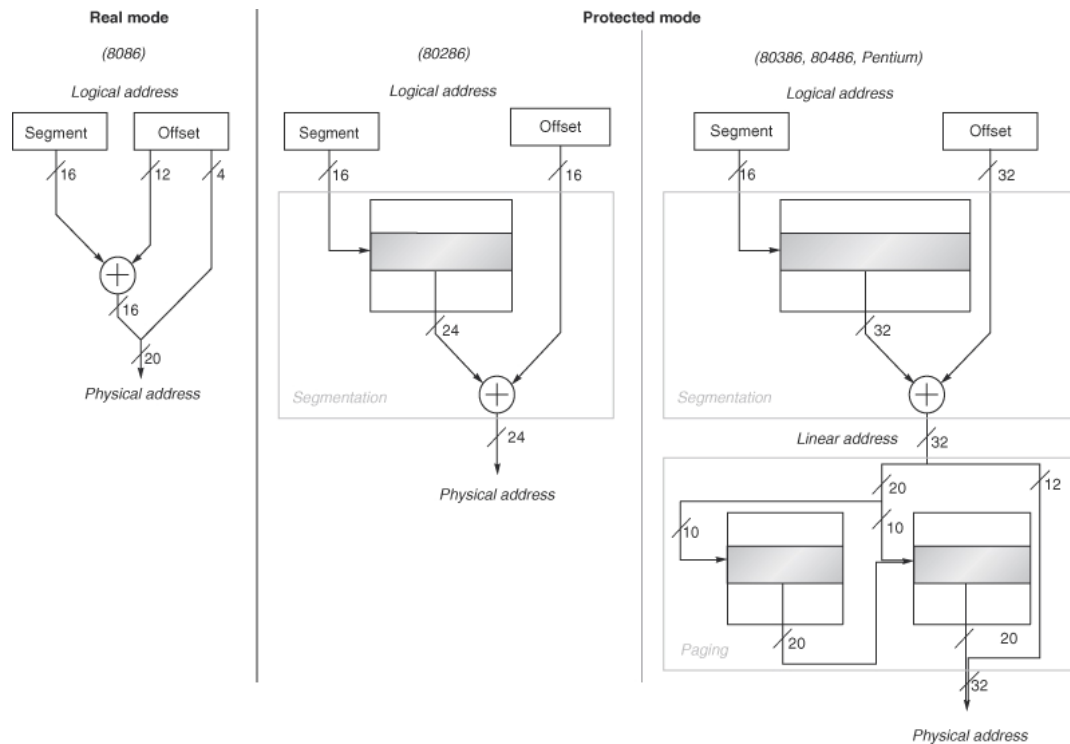


Figure K.39 The original segmented scheme of the 8086 is shown on the left. All 80x86 processors support this style of addressing, called *real mode*. It simply takes the contents of a segment register, shifts it left 4 bits, and adds it to the 16-bit offset, forming a 20-bit physical address. The 80286 (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.

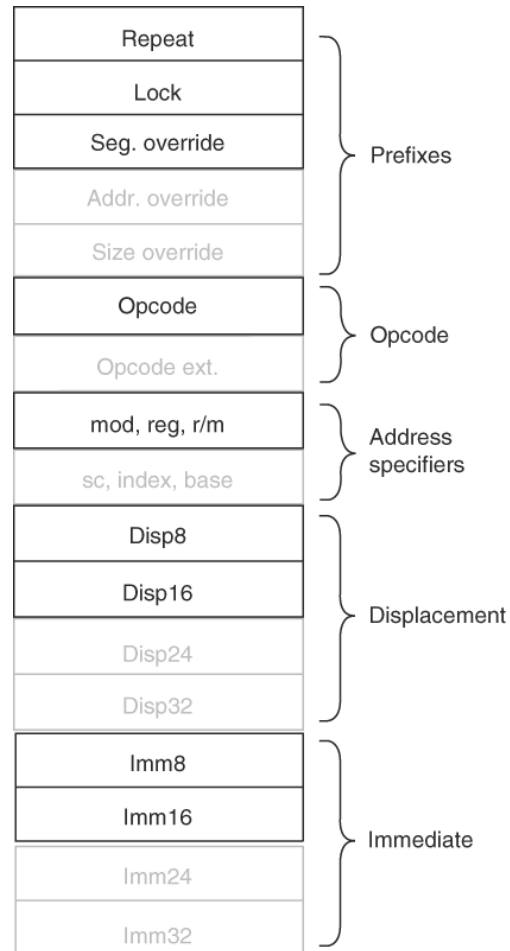


Figure K.43 The instruction format of the 8086 (black type) and the extensions for the 80386 (shaded type). Every field is optional except the opcode.

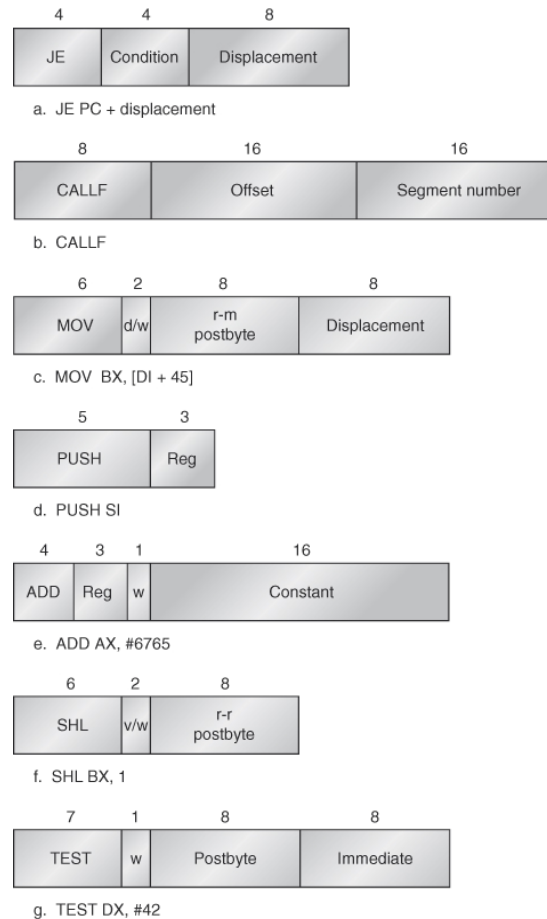


Figure K.44 Typical 8086 instruction formats. The encoding of the postbyte is shown in Figure K.45. Many instructions contain the 1-bit field *w*, which says whether the operation is a byte or a word. Fields of the form *v/w* or *d/w* are a *d*-field or *v*-field followed by the *w*-field. The *d*-field 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.

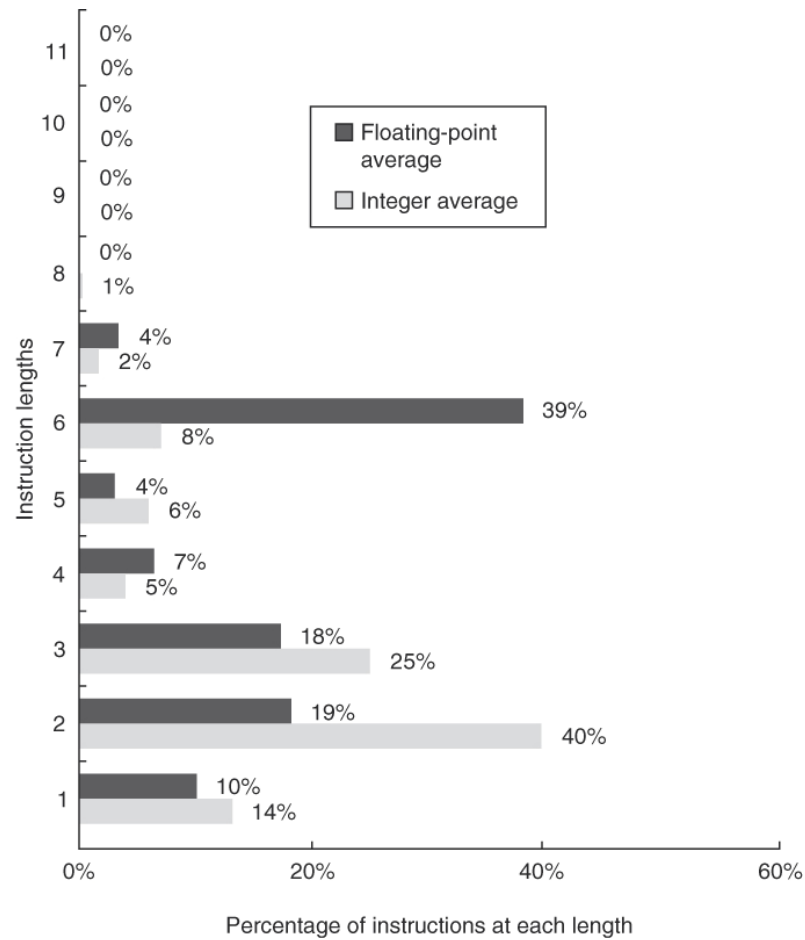


Figure K.49 Averages of the histograms of 80x86 instruction lengths for five SPECint92 programs and for five SPECfp9 programs, all running in 32-bit mode.

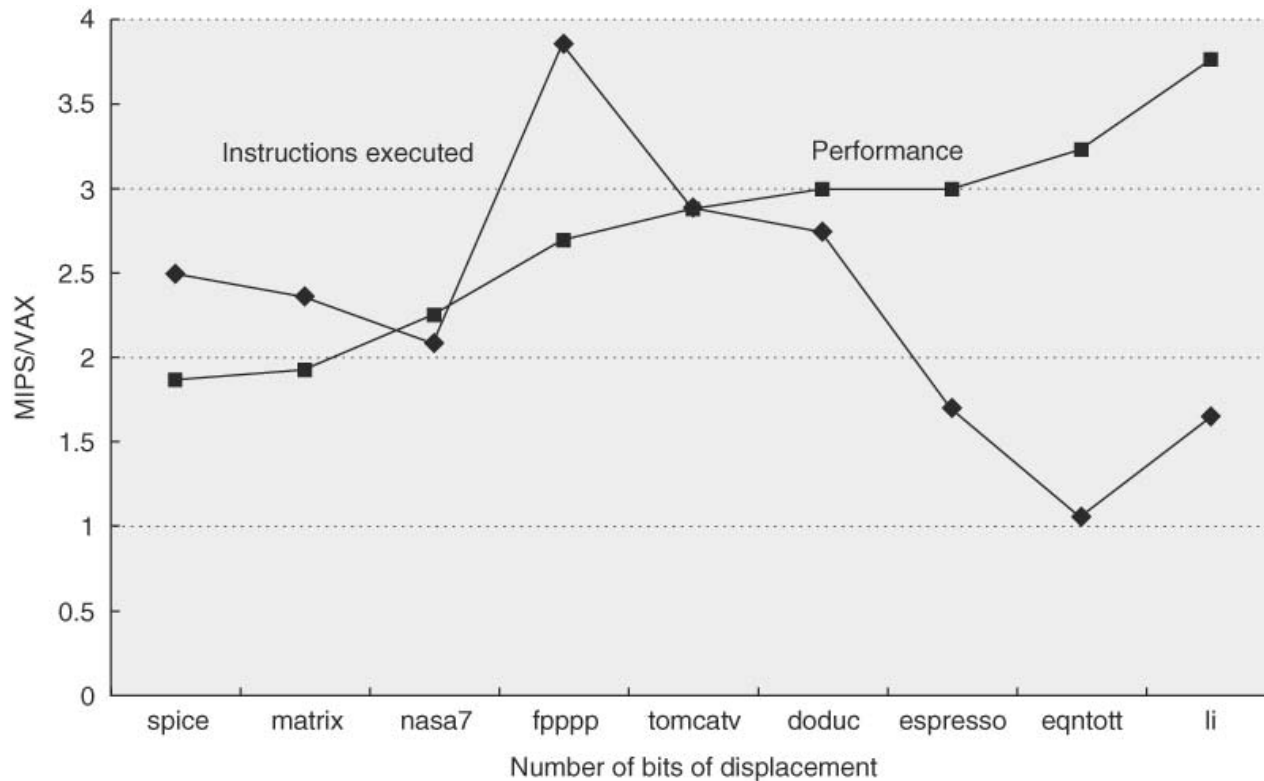


Figure K.64 Ratio of MIPS M2000 to VAX 8700 in instructions executed and performance in clock cycles using SPEC89 programs. On average, MIPS executes a little over twice as many instructions as the VAX, but the CPI for the VAX is almost six times the MIPS CPI, yielding almost a threefold performance 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.)