Figure E.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 E.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).

Figure E.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 E.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.

Figure E.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.

Figure E.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.

Figure E.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 micro-MIPS64 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 E.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 ¼ the main opcode, Opx = an opcode extension, Rd = the destination register, Rs1 ¼ 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.

Figure E.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.

Figure E.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.

Figure E.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.

Figure E.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!).

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

Figure E.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.

Figure E.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.

Figure E.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.

Figure E.15 ALU instructions provided in RV64GC and the equivalents, if any, in the 16-bit instructions of micro-MIPS64 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.

Figure E.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.

Figure E.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.

Figure E.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.

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

Figure E.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).

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

Figure E.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.

Figure E.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.

Figure E.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.

Figure E.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.

Figure E.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.

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

Figure E.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.

Figure E.29 Summary of four 1970s architectures. Unlike the architectures in Figure E.1, there is little agreement between these architectures in any category. (See Section E.3 for more details on the 80x86 and Section E.4 for a description of the VAX.)

Figure E.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..

Figure E.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 E.30 (not IP or FLAGS).

Figure E.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.

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

Figure E.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.

Figure E.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 E.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 E.37 Typical 8086 instruction formats. The encoding of the postbyte is shown in Figure E.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.

Figure E.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 E.39. When mod = 3, the r/m field indicates a register, using the same encoding as the reg field combined with the w bit.

Figure E.39 Based plus scaled index mode address specifier found in the 80386. This mode is indicated by the (sib) notation in Figure E.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 + 2Scale × Index register.

Figure E.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).

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

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

Figure E.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.

Figure E.44 80x86 instruction mix for five SPECfp92 programs.

Figure E.45 80x86 instruction mix for five SPECint92 programs.

Figure E.46 Instructions executed and data accesses on 80x86 and ratios compared to DLX for five SPECint92 programs.

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

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

Figure E.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.

Figure E.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.

Figure E.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 737ten takes 2 bytes.

Figure E.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.

Figure E.53 A C procedure that swaps two locations in memory. This procedure will be used in the sorting example in the next section.

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

Figure E.55 MIPS versus VAX assembly code of the procedure swap in Figure E.53 on page E-30.

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

Figure E.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.)

Figure E.58 The frequency of instruction distribution for two programs on VAX and MIPS.

Figure E.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.