Figure 2.1 MIPS assembly language revealed in this chapter. This information is also found in Column 1 of the MIPS Reference Data Card at the front of this book.

Figure 2.2 Memory addresses and contents of memory at those locations. If these elements were words, these addresses would be incorrect, since MIPS actually uses byte addressing, with each word representing four bytes. Figure 2.3 shows the memory addressing for sequential word addresses.

Figure 2.3 Actual MIPS memory addresses and contents of memory for those words. The changed addresses are highlighted to contrast with Figure 2.2. Since MIPS addresses each byte, word addresses are multiples of 4: there are 4 bytes in a word.

Figure 2.4 The hexadecimal-binary conversion table. Just replace one hexadecimal digit by the corresponding four binary digits, and vice versa. If the length of the binary number is not a multiple of 4, go from right to left.

Figure 2.5 MIPS instruction encoding. In the table above, “reg” means a register number between 0 and 31, “address” means a 16-bit address, and “n.a.” (not applicable) means this field does not appear in this format. Note that add and sub instructions have the same value in the op field; the hardware uses the funct field to decide the variant of the operation: add (32) or subtract (34).

Figure 2.6 MIPS architecture revealed through Section 2.5. The two MIPS instruction formats so far are R and I. The first 16 bits are the same: both contain an op field, giving the base operation; an rs field, giving one of the sources; and the rt field, which specifies the other source operand, except for load word, where it specifies the destination register. R-format divides the last 16 bits into an rd field, specifying the destination register; the shamt field, which Section 2.6 explains; and the funct field, which specifies the specific operation of R-format instructions. I-format combines the last 16 bits into a single address field.

Figure 2.7 The stored-program concept. Stored programs allow a computer that performs accounting to become, in the blink of an eye, a computer that helps an author write a book. The switch happens simply by loading memory with programs and data and then telling the computer to begin executing at a given location in memory. Treating instructions in the same way as data greatly simplifies both the memory hardware and the software of computer systems. Specifically, the memory technology needed for data can also be used for programs, and programs like compilers, for instance, can translate code written in a notation far more convenient for humans into code that the computer can understand.

Figure 2.8 C and Java logical operators and their corresponding MIPS instructions. MIPS implements NOT using a NOR with one operand being zero.

Figure 2.9 Illustration of the options in the if statement above. The left box corresponds to the then part of the if statement, and the right box corresponds to the else part.

Figure 2.10 The values of the stack pointer and the stack (a) before, (b) during, and (c) after the procedure call. The stack pointer always points to the “top” of the stack, or the last word in the stack in this drawing.

Figure 2.11 What is and what is not preserved across a procedure call. If the software relies on the frame pointer register or on the global pointer register, discussed in the following subsections, they are also preserved.

Figure 2.12 Illustration of the stack allocation (a) before, (b) during, and (c) after the procedure call. The frame pointer ($fp) points to the first word of the frame, often a saved argument register, and the stack pointer ($sp) points to the top of the stack. The stack is adjusted to make room for all the saved registers and any memory-resident local variables. Since the stack pointer may change during program execution, it’s easier for programmers to reference variables via the stable frame pointer, although it could be done just with the stack pointer and a little address arithmetic. If there are no local variables on the stack within a procedure, the compiler will save time by not setting and restoring the frame pointer. When a frame pointer is used, it is initialized using the address in $sp on a call, and $sp is restored using $fp. This information is also found in Column 4 of the MIPS Reference Data Card at the front of this book.

Figure 2.13 The MIPS memory allocation for program and data. These addresses are only a software convention, and not part of the MIPS architecture. The stack pointer is initialized to 7fff fffchex and grows down toward the data segment. At the other end, the program code (“text”) starts at 0040 0000hex. The static data starts at 1000 0000hex. Dynamic data, allocated by malloc in C and by new in Java, is next. It grows up toward the stack in an area called the heap. The global pointer, $gp, is set to an address to make it easy to access data. It is initialized to 1000 8000hex so that it can access from 1000 0000hex to 1000 ffffhex using the positive and negative 16-bit offsets from $gp. This information is also found in Column 4 of the MIPS Reference Data Card at the front of this book.

Figure 2.14 MIPS register conventions. Register 1, called $at, is reserved for the assembler (see Section 2.12), and registers 26–27, called $k0–$k1, are reserved for the operating system. This information is also found in Column 2 of the MIPS Reference Data Card at the front of this book.

Figure 2.15 ASCII representation of characters. Note that upper- and lowercase letters differ by exactly 32; this observation can lead to shortcuts in checking or changing upper- and lowercase. Values not shown include formatting characters. For example, 8 represents a backspace, 9 represents a tab character, and 13 a carriage return. Another useful value is 0 for null, the value the programming language C uses to mark the end of a string. This information is also found in Column 3 of the MIPS Reference Data Card at the front of this book.\

Figure 2.16 Example alphabets in Unicode. Unicode version 4.0 has more than 160 “blocks,” which is their name for a collection of symbols. Each block is a multiple of 16. For example, Greek starts at 0370hex, and Cyrillic at 0400hex. The first three columns show 48 blocks that correspond to human languages in roughly Unicode numerical order. The last column has 16 blocks that are multilingual and are not in order. A 16-bit encoding, called UTF-16, is the default. A variable-length encoding, called UTF-8, keeps the ASCII subset as eight bits and uses 16 or 32 bits for the other characters. UTF-32 uses 32 bits per character. New Unicode versions are released every June, with version 13.0 in 2020. Versions 9.0 to 13.0 added various Emojis, while earlier versions added new language blocks and hieroglyphs. The total is almost 150,000 characters. To learn more, see www.unicode.org.

Figure 2.17 The effect of the lui instruction. The instruction lui transfers the 16-bit immediate constant field value into the leftmost 16 bits of the register, filling the lower 16 bits with 0s.

Figure 2.18 Illustration of the five MIPS addressing modes. The operands are shaded in color. The operand of mode 3 is in memory, whereas the operand for mode 2 is a register. Note that versions of load and store access bytes, halfwords, or words. For mode 1, the operand is 16 bits of the instruction itself. Modes 4 and 5 address instructions in memory, with mode 4 adding a 16-bit address shifted left 2 bits to the PC and mode 5 concatenating a 26-bit address shifted left 2 bits with the 4 upper bits of the PC. Note that a single operation can use more than one addressing mode. Add, for example, uses both immediate (addi) and register (add) addressing.

Figure 2.19 MIPS instruction encoding. This notation gives the value of a field by row and by column. For example, the top portion of the figure shows load word in row number 4 (100two for bits 31–29 of the instruction) and column number 3 (011two for bits 28–26 of the instruction), so the corresponding value of the op field (bits 31–26) is 100011two. An underlined name means the field is used elsewhere. For example, R-format in row 0 and column 0 (op = 000000two) is defined in the bottom part of the figure. Hence, subtract in row 4 and column 2 of the bottom section means that the funct field (bits 5–0) of the instruction is 100010two and the op field (bits 31–26) is 000000two. The floating point value in row 2, column 1 is defined in Figure 3.18 in Chapter 3. Bltz/gez is the opcode for four instructions found in Appendix A: bltz, bgez, bltzal, and bgezal. This chapter describes instructions given in full name using color, while Chapter 3 describes instructions given in mnemonics using color. Appendix A covers all instructions.

Figure 2.20 MIPS instruction formats.

Figure 2.21 A translation hierarchy for C. A high-level language program is first compiled into an assembly language program and then assembled into an object module in machine language. The linker combines multiple modules with library routines to resolve all references. The loader then places the machine code into the proper memory locations for execution by the processor. To speed up the translation process, some steps are skipped or combined. Some compilers produce object modules directly, and some systems use linking loaders that perform the last two steps. To identify the type of file, UNIX follows a suffix convention for files: C source files are named x.c, assembly files are x.s, object files are named x.o, statically linked library routines are x.a, dynamically linked library routes are x.so, and executable files by default are called a.out. MS-DOS uses the suffixes .C, .ASM, .OBJ, .LIB, .DLL, and .EXE to the same effect.

Figure 2.22 Dynamically linked library via lazy procedure linkage. (a) Steps for the first time a call is made to the DLL routine. (b) The steps to find the routine, remap it, and link it are skipped on subsequent calls. As we will see in Chapter 5, the operating system may avoid copying the desired routine by remapping it using virtual memory management.

Figure 2.23 A translation hierarchy for Java. A Java program is first compiled into a binary version of Java bytecodes, with all addresses defined by the compiler. The Java program is now ready to run on the interpreter, called the Java Virtual Machine (JVM). The JVM links to desired methods in the Java library while the program is running. To achieve greater performance, the JVM can invoke the JIT compiler, which selectively compiles methods into the native machine language of the machine on which it is running.

Figure 2.24 A C procedure that swaps two locations in memory. This subsection uses this procedure in a sorting example.

Figure 2.25 MIPS assembly code of the procedure swap in **Figure 2.24**.

Figure 2.26 A C procedure that performs a sort on the array v.

Figure 2.27 MIPS assembly version of procedure sort in Figure 2.26.

Figure 2.28 Comparing performance, instruction count, and CPI using compiler optimization for Bubble Sort. The programs sorted 100,000 words with the array initialized to random values. These programs were run on a Pentium 4 with a clock rate of 3.06 GHz and a 533 MHz system bus with 2 GB of PC2100 DDR SDRAM. It used Linux version 2.4.20.

Figure 2.29 Performance of two sort algorithms in C and Java using interpretation and optimizing compilers relative to unoptimized C version. The last column shows the advantage in performance of Quicksort over Bubble Sort for each language and execution option. These programs were run on the same system as Figure 2.28. The JVM is Sun version 1.3.1, and the JIT is Sun Hotspot version 1.3.1.

Figure 2.30 Two C procedures for setting an array to all zeros. Clear1 uses indices, while clear2 uses pointers. The second procedure needs some explanation for those unfamiliar with C. The address of a variable is indicated by &, and the object pointed to by a pointer is indicated by \*. The declarations declare that array and p are pointers to integers. The first part of the for loop in clear2 assigns the address of the first element of array to the pointer p. The second part of the for loop tests to see if the pointer is pointing beyond the last element of array. Incrementing a pointer by one, in the last part of the for loop, means moving the pointer to the next sequential object of its declared size. Since p is a pointer to integers, the compiler will generate MIPS instructions to increment p by four, the number of bytes in a MIPS integer. The assignment in the loop places 0 in the object pointed to by p.

Figure 2.31 Similarities in ARM and MIPS instruction sets.

Figure 2.32 ARM register-register and data transfer instructions equivalent to MIPS core. Dashes mean the operation is not available in that architecture or not synthesized in a few instructions. If there are several choices of instructions equivalent to the MIPS core, they are separated by commas. ARM includes shifts as part of every data operation instruction, so the shifts with superscript 1 are just a variation of a move instruction, such as lsr1. Note that ARM has no divide instruction.

Figure 2.33 Summary of data addressing modes. ARM has separate register indirect and register + offset addressing modes, rather than just putting 0 in the offset of the latter mode. To get greater addressing range, ARM shifts the offset left 1 or 2 bits if the data size is halfword or word.

Figure 2.34 Instruction formats for ARM, MIPS, and RISC-V. The differences result from whether the architecture has 16 registers like ARM or 32 registers like MIPS and RISC-V.

Figure 2.35 ARM arithmetic/logical instructions not found in MIPS.

Figure 2.36 The 80386 register set. Starting with the 80386, the top eight registers were extended to 32 bits and could also be used as general-purpose registers.

Figure 2.37 Instruction types for the arithmetic, logical, and data transfer instructions. The x86 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 2.36 (not EIP or EFLAGS).

Figure 2.38 x86 32-bit addressing modes with register restrictions and the equivalent MIPS code. The Base plus Scaled Index addressing mode, not found in ARM or MIPS, is included to avoid the multiplies by 4 (scale factor of 2) to turn an index in a register into a byte address (see Figures 2.25 and 2.27). A scale factor of 1 is used for 16-bit data, and a scale factor of 3 for 64-bit data. A scale factor of 0 means the address is not scaled. If the displacement is longer than 16 bits in the second or fourth modes, then the MIPS equivalent mode would need two more instructions: a lui to load the upper 16 bits of the displacement and an add to sum the upper address with the base register $s1. (Intel gives two different names to what is called Based addressing mode—Based and Indexed—but they are essentially identical and we combine them here.)

Figure 2.39 Some typical x86 instructions and their functions. A list of frequent operations appears in Figure 2.40. The CALL saves the EIP of the next instruction on the stack. (EIP is the Intel PC.)

Figure 2.40 Some typical operations on the x86. 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 2.41 Typical x86 instruction formats. Figure 2.42 shows the encoding of the postbyte. Many instructions contain the 1-bit field w, which says whether the operation is a byte or a double word. The d field in MOV is used in instructions that may move to or from memory and shows the direction of the move. The ADD instruction requires 32 bits for the immediate field, because in 32-bit mode, the immediates are either 8 bits or 32 bits. The immediate field in the TEST is 32 bits long because there is no 8-bit immediate for test in 32-bit mode. Overall, instructions may vary from 1 to 15 bytes in length. The long length comes from extra 1-byte prefixes, having both a 4-byte immediate and a 4-byte displacement address, using an opcode of 2 bytes, and using the scaled index mode specifier, which adds another byte.

Figure 2.42 The encoding of the first address specifier of the x86: 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-bit mode (8086) or 32-bit mode (80386). 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-bit or 32-bit displacement, depending on the address mode. The exceptions are 1) r/m = 6 when mod = 1 or mod = 2 in 16-bit mode selects BP plus the displacement; 2) r/m = 5 when mod = 1 or mod = 2 in 32-bit mode selects EBP plus displacement; and 3) r/m = 4 in 32-bit mode when mod does not equal 3, where (sib) means use the scaled index mode shown in Figure 2.38. When mod = 3, the r/m field indicates a register, using the same encoding as the reg field combined with the w bit.

Figure 2.43 C version of a double precision matrix multiply, widely known as DGEMM for Double-precision GEneral Matrix Multiply (GEMM).

Figure 2.44 The x86 assembly language for the body of the nested loops generated by compiling the unoptimized C code in Figure 2.43 using gcc with −O3 optimization flags.

Figure 2.45 Performance over the Python program on page 49 for the C program in Figure 2.43 as we increase the level of optimization of the GCC C compiler, with −O0 doing no code size or performance optimization, thereby improving compile time and −O3 being most aggressive for run time and code size. In this case, −O2 and −O3 produce the same x86 code. Most programmers use −O2 as the default compiler flag. GCC also offers a −Os optimization option, which aims at smallest code size.

Figure 2.46 Growth of x86 instruction set over time. While there is clear technical value to some of these extensions, this rapid change also increases the difficulty for other companies to try to build compatible processors.

Figure 2.47 The MIPS instruction set covered so far, with the real MIPS instructions on the left and the pseudoinstructions on the right. Appendix A (Section A.10) describes the full MIPS architecture. Figure 2.1 shows more details of the MIPS architecture revealed in this chapter. The information given here is also found in Columns 1 and 2 of the MIPS Reference Data Card at the front of the book.

Figure 2.48 MIPS instruction classes, examples, correspondence to high-level program language constructs, and percentage of MIPS instructions executed by category for the average integer and floating point SPEC CPU2006 benchmarks. Figure 3.24 in Chapter 3 shows average percentage of the individual MIPS instructions executed.