FIGURE 3.1 Binary addition, showing carries from right to left. The rightmost bit adds 1 to 0, resulting in the sum of this bit being 1 and the carry out from this bit being 0. Hence, the operation for the second digit to the right is 0 + 1 + 1. This generates a 0 for this sum bit and a carry out of 1. The third digit is the sum of 1 + 1 + 1, resulting in a carry out of 1 and a sum bit of 1. The fourth bit is 1 + 0 + 0, yielding a 1 sum and no carry.

FIGURE 3.2 Overflow conditions for addition and subtraction.

FIGURE 3.3 First version of the multiplication hardware. The Multiplicand register, ALU, and Product register are all 64 bits wide, with only the Multiplier register containing 32 bits. (Appendix B describes ALUs.) The 32-bit multiplicand starts in the right half of the Multiplicand register and is shifted left 1 bit on each step. The multiplier is shifted in the opposite direction at each step. The algorithm starts with the product initialized to 0. Control decides when to shift the Multiplicand and Multiplier registers and when to write new values into the Product register.

FIGURE 3.4 The first multiplication algorithm, using the hardware shown in Figure 3.3. If the least significant bit of the multiplier is 1, add the multiplicand to the product. If not, go to the next step. Shift the multiplicand left and the multiplier right in the next two steps. These three steps are repeated 32 times.

FIGURE 3.5 Multiply example using algorithm in Figure 3.4. The bit examined to determine the next step is circled in color.

FIGURE 3.6 Refined version of the multiplication hardware. Compare with the first version in Figure 3.3. The Multiplicand register, ALU, and Multiplier register are all 32 bits wide, with only the Product register left at 64 bits. Now the product is shifted right. The separate Multiplier register also disappeared. The multiplier is placed instead in the right half of the Product register. These changes are highlighted in color. (The Product register should really be 65 bits to hold the carry out of the adder, but it’s shown here as 64 bits to highlight the evolution from Figure 3.3.)

FIGURE 3.7 Fast multiplication hardware. Rather than use a single 32-bit adder 31 times, this hardware “unrolls the loop” to use 31 adders and then organizes them to minimize delay.

FIGURE 3.8 First version of the division hardware. The Divisor register, ALU, and Remainder register are all 64 bits wide, with only the Quotient register being 32 bits. The 32-bit divisor starts in the left half of the Divisor register and is shifted right 1 bit each iteration. The remainder is initialized with the dividend. Control decides when to shift the Divisor and Quotient registers and when to write the new value into the Remainder register.

FIGURE 3.9 A division algorithm, using the hardware in Figure 3.8. If the remainder is positive, the divisor did go into the dividend, so step 2a generates a 1 in the quotient. A negative remainder after step 1 means that the divisor did not go into the dividend, so step 2b generates a 0 in the quotient and adds the divisor to the remainder, thereby reversing the subtraction of step 1. The final shift, in step 3, aligns the divisor properly, relative to the dividend for the next iteration. These steps are repeated 33 times.

FIGURE 3.10 Division example using the algorithm in Figure 3.9. The bit examined to determine the next step is circled in color.

FIGURE 3.11 An improved version of the division hardware. The Divisor register, ALU, and Quotient register are all 32 bits wide, with only the Remainder register left at 64 bits. Compared to Figure 3.8, the ALU and Divisor registers are halved and the remainder is shifted left. This version also combines the Quotient register with the right half of the Remainder register. (As in Figure 3.6, the Remainder register should really be 65 bits to make sure the carry out of the adder is not lost.)

FIGURE 3.12 MIPS core architecture. The memory and registers of the MIPS architecture are not included for space reasons, but this section added the Hi and Lo registers to support multiply and divide. MIPS machine language is listed in the MIPS Reference Data Card at the front of this book.

FIGURE 3.13 EEE 754 encoding of floating-point numbers. A separate sign bit determines the sign. Denormalized numbers are described in the Elaboration on page 232. This information is also found in Column 4 of the MIPS Reference Data Card at the front of this book.

FIGURE 3.14 Floating-point addition. The normal path is to execute steps 3 and 4 once, but if rounding causes the sum to be unnormalized, we must repeat step 3.

FIGURE 3.15 Block diagram of an arithmetic unit dedicated to floating-point addition. The steps of Figure 3.14 correspond to each block, from top to bottom. First, the exponent of one operand is subtracted from the other using the small ALU to determine which is larger and by how much. This difference controls the three multiplexors; from left to right, they select the larger exponent, the significand of the smaller number, and the significand of the larger number. The smaller significand is shifted right, and then the significands are added together using the big ALU. The normalization step then shifts the sum left or right and increments or decrements the exponent. Rounding then creates the final result, which may require normalizing again to produce the actual final result.

FIGURE 3.16 Floating-point multiplication. The normal path is to execute steps 3 and 4 once, but if rounding causes the sum to be unnormalized, we must repeat step 3.

FIGURE 3.17 MIPS floating-point architecture revealed thus far. See Appendix A, Section A.10, for more detail. This information is also found in column 2 of the MIPS Reference Data Card at the front of this book.

FIGURE 3.18 MIPS floating-point instruction encoding. This notation gives the value of a field by row and by column. For example, in the top portion of the figure, lw is found 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. Underlined text means the field is used elsewhere. For example, FlPt in row 2 and column 1 (op = 010001two) is defined in the bottom part of the figure. Hence sub.f in row 0 and column 1 of the bottom section means that the funct field (bits 5–0) of the instruction is 000001two and the op field (bits 31–26) is 010001two. Note that the 5-bit rs field, specified in the middle portion of the figure, determines whether the operation is single precision (f = s, so rs = 10000) or double precision (f = d, so rs = 10001). Similarly, bit 16 of the instruction determines if the bc1.c instruction tests for true (bit 16 = 1= > bc1.t) or false (bit 16 = 0= > bc1.f). This information is also found in column 2 of the MIPS Reference Data Card at the front of this book.

FIGURE 3.19 Summary of ARM NEON instructions for subword parallelism. We use the curly brackets {} to show optional variations of the basic operations: {S8,U8,8} stand for signed and unsigned 8-bit integers or 8-bit data where type doesn’t matter, of which 16 fit in a 128-bit register; {S16,U16,16} stand for signed and unsigned 16-bit integers or 16-bit type-less data, of which 8 fit in a 128-bit register; {S32,U32,32} stand for signed and unsigned 32-bit integers or 32-bit type-less data, of which 4 fit in a 128-bit register; {S64,U64,64} stand for signed and unsigned 64-bit integers or type-less 64-bit data, of which 2 fit in a 128-bit register; {F32} stand for signed and unsigned 32-bit floating point numbers, of which 4 fit in a 128-bit register. Vector Load reads one n-element structure from memory into 1, 2, 3, or 4 NEON registers. It loads a single n-element structure to one lane (See Section 6.6), and elements of the register that are not loaded are unchanged. Vector Store writes one n-element structure into memory from 1, 2, 3, or 4 NEON registers.

FIGURE 3.20 The SSE/SSE2 floating-point instructions of the x86. xmm means one operand is a 128-bit SSE2 register, and mem/xmm means the other operand is either in memory or it is an SSE2 register. We use the curly brackets {} to show optional variations of the basic operations: {SS} stands for Scalar Single precision floating point, or one 32-bit operand in a 128-bit register; {PS} stands for Packed Single precision floating point, or four 32-bit operands in a 128-bit register; {SD} stands for Scalar Double precision floating point, or one 64-bit operand in a 128-bit register; {PD} stands for Packed Double precision floating point, or two 64-bit operands in a 128-bit register; {A} means the 128-bit operand is aligned in memory; {U} means the 128-bit operand is unaligned in memory; {H} means move the high half of the 128-bit operand; and {L} means move the low half of the 128-bit operand.

FIGURE 3.21 Optimized version of DGEMM using C intrinsics to generate the AVX512 subword-parallel instructions for the x86. Figure 3.22 shows the assembly language produced by the compiler for the inner loop.

FIGURE 3.22 The x86 assembly language for the body of the nested loops generated by compiling the optimized C code in Figure 3.21. Note the similarities to Figure 2.44 on page 167, of Chapter 2, with the primary difference being that the original floating-point operations are now using ZMM registers and using the pd versions of the instructions for parallel double precision instead of the sd version for scalar double precision, and it is performing a single multiply–add instruction instead of a separate multiply instruction and a separate add instruction.

FIGURE 3.23 A sampling of newspaper and magazine articles from November 1994, including the New York Times, San Jose Mercury News, San Francisco Chronicle, and Infoworld. The Pentium floating-point divide bug even made the “Top 10 List” of the David Letterman Late Show on television. Intel eventually took a $500 million write-off to replace the buggy chips.

FIGURE 3.24 The MIPS instruction set. This book concentrates on the instructions in the left column. This information is also found in columns 1 and 2 of the MIPS Reference Data Card at the front of this book.

FIGURE 3.25 Remaining MIPS-32 and Pseudo MIPS instruction sets. f means single (s) or double (d) precision floating-point instructions, and s means signed and unsigned (u) versions. MIPS-32 also has FP instructions for multiply and add/sub (madd.f/ msub.f), ceiling (ceil.f), truncate (trunc.f), round (round.f), and reciprocal (recip.f). The underscore represents the letter to include to represent that datatype.

FIGURE 3.26 The frequency of the MIPS instructions for SPEC CPU2006 integer and floating point. All instructions that accounted for at least 0.2% of the instructions are included in the table. Pseudoinstructions are converted into MIPS-32 before execution, and hence do not appear here.

FIGURE 3.27 Floating-point format for IEEE 754 single precision (fp32), IEEE 754 half-precision (fp16), and Brain Float 16. Google’s TPUv3 hardware uses Brain Float 16 (see Section 6.11).