For More Practice

Number Representations

3.8 [10] <§3.2> Two friends, Harry and David, are arguing. Harry says, "All integers greater than zero and exactly divisible by six have exactly two 1s in their binary representation." David disagrees. He says, "No, but all such numbers have an even number of 1s in their representation." Do you agree with Harry or with David, or with neither? (Hint: Look for counterexamples.)

Writing MIPS Code to Perform Arithmetic

3.11 [15] <§3.3> Find the shortest sequence of MIPS instructions to perform double precision integer subtraction. Assume that one 64-bit, two's complement integer is in registers `$t4` and `$t5` and another is in registers `$t6` and `$t7`. The result is to be placed in registers `$t2` and `$t3`. In this example, the most significant word of the 64-bit integer is found in the even-numbered registers, and the least significant word is found in the odd-numbered registers. (Hint: It can be done in four instructions.)

3.14 [5] <§3.6> Instead of using a special hardware multiplier, it is possible to multiply using shift and add instructions. This is particularly attractive when multiplying by small constants. Suppose we want to put nine times the value of `$s0` into `$s1`, ignoring any overflow that may occur. Show a minimal sequence of MIPS instructions for doing this without using a multiply instruction.

3.15 [20] <§3.6> Find the shortest sequence of MIPS instructions to perform double precision integer multiplication. Try to do it in 35 instructions or less. Assume that one 64-bit, *unsigned* integer is in registers `$t4` and `$t5` and another is in registers `$t6` and `$t7`. The 128-bit product is to be placed in registers `$t0`, `$t1`, `$t2`, and `$t3`. In this example, the most significant word is found in the lower-numbered registers, and the least significant word is found in the higher-numbered registers. (Hint: Write out the formula for $(a \times 2^{32} + b) \times (c \times 2^{32} + d)$.)

Simulating MIPS Machines

3.16 [2 weeks] <§3.4> Write a simulator for a subset of the MIPS instruction set using MIPS instructions and the SPIM simulator described in Appendix A. Your simulator should execute hand-assembled programs that are located in the data segment of the SPIM simulator and should use `$v0` and `$v1` for input and output. Other portions of the data segment can be used for storing the memory contents and register values of your virtual machine. Your implementation can use any of the MIPS instructions, but your simulator need only support a smaller subset of the instruction set (e.g., the instructions appearing in Chapters 5 and 6).

3.17 [1 week] <§3.4> Add an exception handler to the simulator you developed for Exercise 3.16. Your simulator should generate a simulated exception if a mis-

aligned word is accessed via an `lw`, `sw`, or `jr` instruction. The exception handler should print out an error message identifying the offending address (within the simulation) and then realign the access, perform the instruction, and resume executing the simulated program.

Carry Lookahead Adders

3.18 [5] <§3.4> Rewrite the equations on page B-40 for a carry-lookahead logic for a 16-bit adder using a new notation. First, use the names for the CarryIn signals of the individual bits of the adder. That is, use c4, c8, c12, . . . instead of C1, C2, C3, ... In addition, let $P_{i,j}$ mean a propagate signal for bits *i* to *j,* and $G_{i,j}$ mean a generate signal for bits *i* to *j.* For example, the equation

$$C2 = G1 + (P1 \cdot G0) + (P1 \cdot P0 \cdot c0)$$

can be rewritten as

$$c8 = G_{7,4} + (P_{7,4} \cdot G_{3,0}) + (P_{7,4} \cdot P_{3,0} \cdot c0)$$

This more general notation is useful in creating wider adders.

3.19 [15] <§3.4> Write the equations for the carry-lookahead logic for a *64-bit* adder using the new notation from Exercise 3.18 and using 16-bit adders as building blocks. Include a drawing similar to Figure B.6.3 in your solution.

Relative Performance of Adders

3.20 [10] <§3.4> Now calculate the relative performance of adders. Assume that hardware corresponding to any equation containing only OR or AND terms, such as the equations for p*i* and g*i* on page B-39, takes one time unit T. Equations that consist of the OR of several AND terms, such as the equations for c1, c2, c3, and c4 on page B-40, would thus take two time units, 2T. The reason is it would take T to produce the AND terms and then an additional T to produce the result of the OR. Calculate the numbers and performance ratio for 4-bit adders for both ripple carry and carry lookahead. If the terms in equations are further defined by other equations, then add the appropriate delays for those intermediate equations, and continue recursively until the actual input bits of the adder are used in an equation. Include a drawing of each adder labeled with the calculated delays and the path of the worst-case delay highlighted.

3.21 [15] <§3.4> This exercise is similar to Exercise 3.20, but this time calculate the relative speeds of a 16-bit adder using ripple carry only, ripple carry of 4-bit groups that use carry lookahead, and the carry-lookahead scheme on page B-39.

3.22 [15] <§3.4> This exercise is similar to Exercises 3.20 and 3.21, but this time calculate the relative speeds of a 64-bit adder using ripple carry only, ripple carry of 4-bit groups that use carry lookahead, ripple carry of 16-bit groups that use carry lookahead, and the carry-lookahead scheme from Exercise 3.19.

3.24 [30] <§3.4> If you have access to a computer containing a MIPS processor, write a loop in assembly language that sets registers `$k0` (`$26`) and `$k1` (`$27`) to an initial value, and then loop for several seconds, checking the contents of these registers. Print the values if they change. See the elaboration on page 174 for an explanation of why they change. Can you find a reason for the particular values you observe?

Floating Point Number Representations

3.32 [10] <§3.6> Write a simple C program that inputs a floating-point number and shows its bit representation in hexadecimal.

3.33 [10] <§3.6> Write a simple Java program that inputs a floating-point number and shows its bit representation in hexadecimal.

Writing MIPS Code to Perform FP Arithmetic

3.34 [10] <§3.6> A single precision IEEE number is stored in memory at address X. Write a sequence of MIPS instructions to multiply the number at X by 4 and store the result back at X. Accomplish this without using any floating-point instructions (don't worry about overflow).

Floating Point on Algorithms

3.47 [25] <§3.8> Derive the floating-point algorithm for division as we did for addition and multiplication on pages 197 through 205. First divide $1.010_{ten} \times 10^{8}$ by $1.010_{ten} \times 10^{-4}$, showing the same steps that we did in the example starting on page 204. Then derive the floating-point division algorithm using a format similar to the multiplication algorithm in Figure 3.18 on page 205.

3.48 [30] <§3.8> The elaboration on page 217 explains the four rounding modes of IEEE 754 and the extra bit, called the *sticky bit*, needed in addition to the 2 bits called *guard* and *round*. Guard is the first bit, round is the second bit, and sticky represents whether the remaining bits are 0 or not. Fill in the following table with logical equations that are functions of guard (g), round (r), and sticky (s) for the result of a floating-point addition that creates Sum. Let p be the proper number of bits in the significand for a given precision and Sum_p be the pth most significant

bit of Sum. A blank box means that the p most significant bits of the sum are correctly rounded. If you place an equation in a box, a false equation means that the p bits are correctly rounded; a true equation means add 1 to the pth most significant bit of Sum.

Rounding mode	Sum ≧ 0	Sum < 0
Toward $-\infty$		
Toward $+\infty$		
Truncate		
Nearest even		

Denormalized Numbers

3.49 [30] <§3.8> The elaboration on page 193 mentions that IEEE 754 has two special symbols that are floating-point operands: infinity and Not a Number (NaN). There are also small numbers called *denorms,* which are not normalized. Because these special symbols and numbers are not used very frequently, implementations that employ a mix of both hardware and software techniques are sometimes used. For example, instead of using complicated hardware to handle these special cases, an exception is generated and they are handled in software. Many implementation options exist, each of which has unique performance characteristics. Your task is to benchmark several different machines for floating-point operations as the operands vary from normal numbers to these special cases. Be sure to state your conclusions by comparing the performance of different machines with one another and describing their similarities and differences. What impact are your results likely to have on software designers who must choose whether or not to make use of the special features in the IEEE 754 standard?

Evaluating Instruction Frequencies

3.50 [10] <§3.9> For the average of the SPEC2000 integer benchmarks (Figure 3.26 on page 228), find the 10 most frequently executed MIPS instructions. List them in order of popularity, from most used to least used. Show the rank, name, and percentage of instructions executed for each instruction. If there is a tie for a given rank, list all instructions that tie with the same rank, even if this results in more than 10 instructions.

3.51 [10] <§3.9> This exercise is similar to Exercise 3.50, but this time replace the SPECint averages with the SPECfp averages.

3.52 <§3.9> These questions examine the relative frequency of instructions in different programs.

a. [5] Which instructions are found both in the answer to Exercise 3.50 and in the answer to Exercise 3.51?

b. [5] What percentage of SPECint instructions executed is due to the instructions identified in Exercise 3.52?

c. [5] What percentage of SPECint instructions executed is due to the instructions identified in Exercise 3.50?

d. [5] What percentage of SPECfp instructions executed is due to the instructions identified in Exercise 3.52?

e. [5] What percentage of SPECfp instructions executed is due to the instructions identified in Exercise 3.51?

3.53 [10] <§3.9> If you were designing a machine to execute the MIPS instruction set, what are the five instructions that you would try to make as fast as possible, based on the answers to Exercises 3.50 through 3.52? Give your rationale.

Evaluating Performance

3.54 [15] <§3.9> Using Figure 3.26 on page 228, calculate the average clock cycles per instruction (CPI) for the SPECint benchmarks. Figure 3.11.1 gives the average CPI per instruction category, taking into account cache misses and other effects. Assume that instructions omitted from the table have a CPI of 1.0.

Instruction category	Average CPI
Loads and stores	1.3
Conditional branch	1.6
Jumps	1.2
Integer multiply	5.0
Integer divide	12.0
Floating-point add and subtract	1.0
Floating-point multiply, single precision	2.0
Floating-point multiply, double precision	3.0
Floating-point divide, single precision	6.0
Floating-point divide, double precision	10.0

FIGURE 3.11.1 CPI for MIPS instruction categories.

3.55 [15] <§3.9> This exercise is similar to Exercise 3.54, but this time replace SPECint with SPECfp.