Wiecek, C. [1982]. "A case study of the VAX 11 instruction set usage for compiler execution," *Proc. Symposium on Architectural Support for Programming Languages and Operating Systems* (March), IEEE/ACM, Palo Alto, Calif., 177–184.

Wulf, W. [1981]. "Compilers and computer architecture," *Computer* 14:7 (July), 41–47.

Exercises

Solutions to the "starred" exercises appear in Appendix B.

2.1 [10/15] <2.2> For the following assume that values A, B, C, D, and E reside in memory. Also assume that instruction operation codes are represented in 8 bits, memory addresses are 64 bits, and register addresses are 6 bits.

a. [10] <2.2> For each instruction set architecture shown in Figure 2.2, how many addresses, or names, appear in each instruction for the code to compute C = A + B, and what is the total code size?

b. [15] <2.2> Some of the instruction set architectures in Figure 2.2 destroy operands in the course of computation. This loss of data values from processor internal storage has performance consequences. For each architecture in Figure 2.2, write the code sequence to compute C = A + B followed by D = A – E. In your code, mark each operand that is destroyed during execution and mark each "overhead" instruction that is included just to overcome this loss of data from processor internal storage. What is the total code size, the number of bytes of instructions and data moved to or from memory, the number of overhead instructions, and the number of overhead data bytes for each of your code sequences?

✪ 2.2 [15] <2.2> Some operations on two operands (subtraction, for example) are not commutative. What are the advantages and disadvantages of the stack, accumulator, and load-store architectures when executing noncommutative operations?

2.3 [15/15/10/10] <2.3> The value represented by the hexadecimal number 434F 4D50 5554 4552 is to be stored in an aligned 64-bit double word.

a. [15] <2.3> Using the physical arrangement of the first row in Figure 2.5, write the value to be stored using Big Endian byte order. Next, interpret each byte as an ASCII character and below each byte write the corresponding character, forming the character string as it would be stored in Big Endian order.

b. [15] <2.3> Using the same physical arrangement as in part (a), write the value to be stored using Little Endian byte order and below each byte write the corresponding ASCII character.

c. [10] <2.3> What are the hexadecimal values of all misaligned 2-byte words that can be read from the given 64-bit double word when stored in Big Endian byte order?

d. [10] <2.3> What are the hexadecimal values of all misaligned 4-byte words that can be read from the given 64-bit double word when stored in Little Endian byte order?

2.4 [20/15/15/20] <2.2, 2.3, 2.10> Your task is to compare the memory efficiency of four different styles of instruction set architectures. The architecture styles are

1. *Accumulator*—All operations occur between a single register and a memory location.
2. *Memory-memory*—All instruction addresses reference only memory locations.
3. *Stack*—All operations occur on top of the stack. Push and pop are the only instructions that access memory; all others remove their operands from the stack and replace them with the result. The implementation uses a hardwired stack for only the top two stack entries, which keeps the processor circuit very small and low cost. Additional stack positions are kept in memory locations, and accesses to these stack positions require memory references.
4. *Load-store*—All operations occur in registers, and register-to-register instructions have three register names per instruction.

To measure memory efficiency, make the following assumptions about all four instruction sets:

- All instructions are an integral number of bytes in length.
- The opcode is always 1 byte (8 bits).
- Memory accesses use direct, or absolute, addressing.
- The variables A, B, C, and D are initially in memory.

✪ a. [20] <2.2, 2.3> Invent your own assembly language mnemonics (Figure 2.2 provides a useful sample to generalize), and for each architecture write the best equivalent assembly language code for this high-level language code sequence:

```
A = B + C;
B = A + C;
D = A - B;
```

b. [15] <2.3> Label each instance in your assembly codes for part (a) where a value is loaded from memory after having been loaded once. Also label each instance in your code where the result of one instruction is passed to another instruction as an operand, and further classify these events as involving storage within the processor or storage in memory.

c. [15] <2.10> Assume the given code sequence is from a small, embedded computer application, such as a microwave oven controller, that uses 16-bit memory addresses and data operands. If a load-store architecture is used, assume it has 16 general-purpose registers. For each architecture answer the following questions: How many instruction bytes are fetched? How many bytes of data are transferred from/to memory? Which architecture is most efficient as measured by code size? Which architecture is most efficient as measured by total memory traffic (code + data)?

d. [20] <2.10> Now assume a processor with 64-bit memory addresses and data operands. For each architecture answer the questions of part (c). How have the relative merits of the architectures changed for the chosen metrics?

2.5 [20/20/20] <2.3> We are designing instruction set formats for a load-store architecture and are trying to decide whether it is worthwhile to have multiple offset lengths for branches and memory references. The length of an instruction would be equal to 16 bits + offset length in bits, so ALU instructions will be 16 bits.

Figure 2.42 contains data on offset size for the Alpha architecture with full optimization for SPEC CPU2000. For instruction set frequencies, use the data for MIPS from the average of the five benchmarks for the load-store machine in Figure 2.32. Assume that the miscellaneous instructions are all ALU instructions that use only registers.

Number of offset magnitude bits	Cumulative data references	Cumulative branches
0	30.4%	0.1%
1	33.5%	2.8%
2	35.0%	10.5%
3	40.0%	22.9%
4	47.3%	36.5%
5	54.5%	57.4%
6	60.4%	72.4%
7	66.9%	85.2%
8	71.6%	90.5%
9	73.3%	93.1%
10	74.2%	95.1%
11	74.9%	96.0%
12	76.6%	96.8%
13	87.9%	97.4%
14	91.9%	98.1%
15	100%	98.5%
16	100%	99.5%
17	100%	99.8%
17	100%	99.9%
19	100%	100%
20	100%	100%
21	100%	100%

Figure 2.42 The second and third columns contain the cumulative percentage of the data references and branches, respectively, that can be accommodated with the corresponding number of bits of magnitude in the displacement. These are the average distances of all the integer and floating-point programs in Figure 2.8.

a. [20] <2.3> Suppose offsets are permitted to be 0, 8, 16, or 24 bits in length, including the sign bit. What is the average length of an executed instruction?

b. [20] <2.3> Suppose we want a fixed-length instruction and we chose a 24-bit instruction length (for everything, including ALU instructions). For every offset of longer than 8 bits, additional instruction(s) are required. Determine the number of instruction bytes fetched in this machine with fixed instruction size versus those fetched with a byte-variable-sized instruction as defined in part (a).

c. [20] <2.3> Now suppose we use a fixed offset length of 24 bits so that no additional instruction is ever required. How many instruction bytes would be required? Compare this result to your answer to part (b).

2.6 [15/10] <2.3> Several researchers have suggested that adding a register-memory addressing mode to a load-store machine might be useful. The idea is to replace sequences of

```
LOAD      R1,0(Rb)
ADD       R2,R2,R1
```

by

```
ADD       R2,0(Rb)
```

Assume the new instruction will cause the clock cycle to increase by 5%. Use the instruction frequencies for the gcc benchmark on the load-store machine from Figure 2.32. The new instruction affects only the clock cycle and not the CPI.

a. [15] <2.3> What percentage of the loads must be eliminated for the machine with the new instruction to have at least the same performance?

b. [10] <2.3> Show a situation in a multiple instruction sequence where a load of R1 followed immediately by a use of R1 (with some type of opcode) could not be replaced by a single instruction of the form proposed, assuming that the same opcode exists.

2.7 [25] <2.2–2.5> Find an instruction set manual for some older machine (libraries and private bookshelves are good places to look). Summarize the instruction set with the discriminating characteristics used in Figures 2.3 and 2.4. Write the code sequence for this machine for the statements in Exercise 2.1(b). The size of the data need not be the same as in Exercise 2.1(b) if the word size is smaller in the older machine.

2.8 [20] <2.2, 2.12> Consider the following fragment of C code:

```
for (i=0; i<=100; i++)
        {A[i] = B[i] + C;}
```

Assume that A and B are arrays of 64-bit integers, and C and i are 64-bit integers. Assume that all data values and their addresses are kept in memory (at addresses 0, 5000, 1500, and 2000 for A, B, C, and i, respectively) except when they are operated on. Assume that values in registers are lost between iterations of the loop.

Write the code for MIPS. How many instructions are required dynamically? How many memory-data references will be executed? What is the code size in bytes?

2.9 [20] <2.2, 2.12> For this question use the code sequence of Exercise 2.8, but put the scalar data—the value of `i`, the value of `C`, and the addresses of the array variables (but not the actual array)—in registers and keep them there whenever possible.

Write the code for MIPS. How many instructions are required dynamically? How many memory-data references will be executed? What is the code size in bytes?

2.10 [15] <2.12> When designing memory systems it becomes useful to know the frequency of memory reads versus writes and also accesses for instructions versus those for data. Using the average instruction mix information for MIPS in Figure 2.32, find

- the percentage of all memory accesses for data
- the percentage of data accesses that are reads
- the percentage of all memory accesses that are reads

Ignore the size of a datum when counting accesses.

2.11 [18] <2.12> Compute the effective CPI for MIPS using Figure 2.32. Suppose we have made the following measurements of average CPI for instructions:

Instruction	Clock cycles
All ALU instructions	1.0
Loads-stores	1.4
Conditional branches	
Taken	2.0
Not taken	1.5
Jumps	1.2

Assume that 60% of the conditional branches are taken and that all instructions in the "other" category of Figure 2.32 are ALU instructions. Average the instruction frequencies of gap and gcc to obtain the instruction mix.

2.12 [20/10] <2.3, 2.12> Consider adding a new index addressing mode to MIPS. The addressing mode adds two registers and an 11-bit signed offset to get the effective address.

Our compiler will be changed so that code sequences of the form

```
ADD R1, R1, R2
LW  Rd, 100(R1)    (or store)
```

will be replaced with a load (or store) using the new addressing mode. Use the overall average instruction frequencies from Figure 2.32 in evaluating this addition.

a. [20] <2.3, 2.12> Assume that the addressing mode can be used for 10% of the displacement loads and stores (accounting for both the frequency of this type of address calculation and the shorter offset). What is the ratio of instruction count on the enhanced MIPS compared to the original MIPS?

b. [10] <2.3, 2.12> If the new addressing mode lengthens the clock cycle by 5%, which machine will be faster and by how much?

2.13 [30] <2.7> Many computer manufacturers now include tools or simulators that allow you to measure the instruction set usage of a user program. Among the methods in use are machine simulation, hardware-supported trapping, and a compiler technique that instruments the object code module by inserting counters. Find a processor available to you that includes such a tool. Use it to measure the instruction set mix for one of the SPEC CPU2000 benchmarks reported on in this chapter. Compare the results to those shown in this chapter.

✪ 2.14 [10/10] <2.8> One use of saturating arithmetic is for real-time applications that may fail their response time constraints if processor effort is diverted to handling arithmetic exceptions. Another benefit is that the result may be more desirable. Take, for example, an image array of 24-bit picture elements (pixels), each comprised of three 8-bit unsigned integers, representing red, green, and blue color brightness, that represent an image. Larger values are brighter.

a. [10] <2.8> Brighten the two pixels E5F1D7 and AAC4DE by adding 20 to each color component using unsigned arithmetic and ignoring overflow to maintain a fixed total instruction-processing time. The values are given in hexadecimal. What are the resulting pixel values? Are the pixels brightened?

b. [10] <2.8> Repeat part (a) but use saturating arithmetic instead. What are the resulting pixel values? Are the pixels brightened?

2.15 [20] <2.9> A condition code is a bit of processor state updated each time certain ALU operation(s) execute to reflect some aspect of the execution. For example, a subtract instruction may set a bit if the result is negative and reset it for a positive result. A later operation can refer to this specific "result sign" condition code bit to glean information about the subtract result, provided no other instruction of the set that updates the result sign condition code has executed in the meantime. The concept of dedicated condition codes can be generalized to an array of general-purpose condition bits. An instruction is encoded to use any one of the general-purpose condition bits, as selected by the compiler. What are the advantages and disadvantages of a collection of general-purpose condition bits as compared to those of dedicated condition codes (see Figure 2.21)?

2.16 [25/15] <2.7, 2.11> Find a C compiler and compile the code shown in Exercise 2.8 for one of the machines covered in this book. Compile the code both optimized and unoptimized.

a. [25] <2.7, 2.11> Find the instruction count, dynamic instruction bytes fetched, and data accesses done for both the optimized and unoptimized versions.

b. [15] <2.7, 2.11> Try to improve the code by hand and compute the same measures as in part (a) for your hand-optimized version.

2.17 [30/30] <2.7, 2.11> Small synthetic benchmarks can be very misleading when used for measuring instruction mixes. This is particularly true when these benchmarks are optimized. In this exercise, we want to explore these differences. This programming exercise can be done with any load-store machine.

a. Compile Whetstone with optimization. Compute the instruction mix for the top 20 most frequently executed instructions. How do the optimized and unoptimized mixes compare? How does the optimized mix compare to the mix for 171.swim from SPEC2000 on the same or a similar machine?

b. [30] <2.7, 2.11> Follow the same guidelines as for part (a), but this time use Dhrystone and compare it with gcc.

2.18 [10] <2.11> Consider this high-level language code sequence of three statements:

```
A = B + C;
B = A + C;
D = A - B;
```

Use the technique of copy propagation (see Figure 2.25) to transform the code sequence to the point where no operand is a computed value. Note the instances in which the transformation has reduced the computational work of a statement and those cases where the work has increased. What does this suggest about the technical challenge faced in trying to satisfy the desire for optimizing compilers?

2.19 [20] <2.2, 2.10, 2.12> The design of MIPS provides for 32 general-purpose registers and 32 floating-point registers. If registers are good, are more registers better? List and discuss as many trade-offs as you can that should be considered by instruction set architecture designers examining whether to, and how much to, increase the numbers of MIPS registers.

2.20 [30] <2.3, 2.10, 2.12> MIPS has only a three-address format for its R-type register-register instructions. Many operations might use the same destination register as one of the sources. We could introduce a new instruction format into MIPS called R_2 that has only two addresses and is a total of 24 bits in length. By using this instruction type whenever an operation had only two different register operands, we could reduce the instruction bandwidth required for a program. Modify the MIPS simulator to count the frequency of register-register operations with only two different register operands. Using the benchmarks that come with the simulator, determine how much more instruction bandwidth MIPS requires than MIPS with the R_2 format.

2.21 [40] <2.2–2.12> Very long instruction word (VLIW) computers are discussed in Chapter 4, but increasingly DSPs and media processors are adopting this style of instruction set architecture. One example is the TI TMS320C6203. See if you can compare code size of VLIW to more traditional computers. One attempt would be to code a common kernel across several machines. Another would be to get access to compilers for each machine and compare code sizes. Based on your data, is VLIW an appropriate architecture for embedded applications? Why or why not?

2.22 [35] <2.2–2.8> GCC targets most modern instruction set architectures (see *www.gnu.org/software/gcc/gcc.html*). Create a version of gcc for several architectures that you have access to, such as 80x86, Alpha, MIPS, PowerPC, and SPARC. Then compile a subset of SPEC CPU2000 integer benchmarks and create a table of code sizes. Which architecture is best for each program?

2.23 [25] <App. C> How much do the instruction set variations among the RISC machines discussed in Appendix C affect performance? Choose at least three small programs (e.g., a sorting routine), and code these programs in MIPS and two other assembly languages. What is the resulting difference in instruction count?

2.24 [Discussion] <2.2–2.12> Where do instruction sets come from? Since the earliest computers date from just after World War II, it should be possible to derive the ancestry of the instructions in modern computers. This project will take a good deal of delving into libraries and perhaps contacting pioneers, but see if you can derive the ancestry of the instructions in, say, MIPS.

2.25 [Discussion] <2.2–2.15> What are the *economic* arguments (i.e., more machines sold) for and against changing instruction set architecture in desktop and server markets? What about embedded markets?

2.26 [Discussion] <1, 2> As we shall see in Chapter 3, many desktop microprocessors have a microinstruction set architecture (internal) that is different from the instruction set architecture (external) that software uses. For most such microprocessors, hardware translates each external instruction to internal instruction(s) when the instruction is fetched. However, at least one microprocessor uses an interpreter to translate instructions one at a time for the hardware and, when it detects frequently used code segments, invokes a compiler on the fly to compile those segments into optimized hardware instruction sequences and saves the compiled segments for reuse. List the pros and cons of each approach, commenting on at least the following issues: impact on clock cycle time, die size, code size, CPI, software compatibility, and execution time of the program.