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.

Instruction	gap	gcc	gzip	mcf	perl	Integer average
load	26.5%	25.1%	20.1%	30.3%	28.7%	26%
store	10.3%	13.2%	5.1%	4.3%	16.2%	10%
add	21.1%	19.0%	26.9%	10.1%	16.7%	19%
sub	1.7%	2.2%	5.1%	3.7%	2.5%	3%
mul	1.4%	0.1%				0%
compare	2.8%	6.1%	6.6%	6.3%	3.8%	5%
load imm	4.8%	2.5%	1.5%	0.1%	1.7%	2%
cond branch	9.3%	12.1%	11.0%	17.5%	10.9%	12%
cond move	0.4%	0.6%	1.1%	0.1%	1.9%	1%
jump	0.8%	0.7%	0.8%	0.7%	1.7%	1%
call	1.6%	0.6%	0.4%	3.2%	1.1%	1%
return	1.6%	0.6%	0.4%	3.2%	1.1%	1%
shift	3.8%	1.1%	2.1%	1.1%	0.5%	2%
and	4.3%	4.6%	9.4%	0.2%	1.2%	4%
or	7.9%	8.5%	4.8%	17.6%	8.7%	9%
xor	1.8%	2.1%	4.4%	1.5%	2.8%	3%
other logical	0.1%	0.4%	0.1%	0.1%	0.3%	0%
load FP						0%
store FP						0%
add FP						0%
sub FP						0%
mul FP						0%
div FP						0%
mov reg-reg FP						0%
compare FP						0%
cond mov FP						0%
other FP						0%

Figure 2.32 MIPS dynamic instruction mix for five SPECint2000 programs. Note that integer register-register move instructions are included in the or instruction. Blank entries have the value 0.0%.

be issued at the same time. If there are not five independent instructions available for the compiler to schedule together—that is, the rest are dependent—then NOPs are placed in the leftover slots. This instruction coding technique is called, naturally enough, *very long instruction word* (VLIW), and it predates the Trimedia processors. VLIW is the subject of Chapter 4, so we will just give a preview of VLIW here. An example helps explain how the Trimedia TM32 CPU works.