Part II: Solutions Guide
1 Solutions

1.1 q
1.2 u
1.3 f
1.4 a
1.5 c
1.6 d
1.7 i
1.8 k
1.9 j
1.10 o
1.11 w
1.12 p
1.13 n
1.14 r
1.15 y
1.16 s
1.17 l
1.18 g
1.19 x
1.20 z
1.21 t
1.22 b
1.23 h
1.24 m
1.25 e
1.26 v
1.27 j
1.28 b
1.29 f
1.30 j
1.31 i
1.32 e
1.33 d
1.34 g
1.35 c
1.36 g
1.37 d
1.38 c
1.39 j
1.40 b
1.41 f
1.42 h
1.43 a
1.44 a

1.45 Time for \( \frac{1}{2} \) revolution = \( \frac{1}{2} \) rev \( \times \) \( \frac{1}{5400} \) \( \frac{\text{minutes}}{\text{rev}} \) \( \times \) 60 \( \frac{\text{seconds}}{\text{minute}} \) = 5.56 ms

1.46 As discussed in section 1.4, die costs rise very fast with increasing die area. Consider a wafer with a large number of defects. It is quite likely that if the die area is very small, some dies will escape with no defects. On the other hand, if the die area is very large, it might be likely that every die has one or more defects. In general, then, die area greatly affects yield (as the equations on page 48 indicate), and so we would expect that dies from wafer B would cost much more than dies from wafer A.

1.47 The die area of the Pentium processor in Figure 1.16 is 91 mm\(^2\) and it contains about 3.3 million transistors, or roughly 36,000 per square millimeter. If we assume the period has an area of roughly .1 mm\(^2\), it would contain 3500 transistors (this is certainly a very rough estimate). Similar calculations with regard to Figure 1.26 and the Intel 4004 result in 191 transistors per square millimeter or roughly 19 transistors.

1.48 We can write Dies per wafer = \( f((\text{Die area})^{-1}) \) and Yield = \( f((\text{Die area})^{-2}) \) and thus Cost per die = \( f((\text{Die area})^3) \). More formally, we can write:

\[
\text{Cost per die} = \frac{\text{Cost per wafer}}{\text{Dies per wafer} \times \text{yield}}
\]

\[
\text{Dies per wafer} = \frac{\text{Wafer area}}{\text{Die area}}
\]

\[
\text{Yield} = \frac{1}{(1 + \text{Defect per area} \times \text{Die area}/2)^2}
\]

1.49 No solution provided.

1.50 From the caption in Figure 1.16 we have 198 dies at 100% yield. If the defect density is 1 per square centimeter, then the yield is approximated by \( 1/((1 + 1 \times .91/2)^2) = .47 \). Thus 198 \( \times \) .47 = 93 dies with a cost of $1000/93 = $10.75 per die.

1.51 Defects per area.
1.52 \ \text{Yield} = \frac{1}{(1 + \text{Defects per area} \times \text{Die area}/2)^2}

1.53

\begin{array}{|c|c|c|}
\hline
1980 & \text{Die area} & 0.16 \\
\hline
 & \text{Yield} & 0.48 \\
 & \text{Defect density} & 17.04 \\
\hline
1992 & \text{Die area} & 0.97 \\
\hline
 & \text{Yield} & 0.48 \\
 & \text{Defect density} & 1.98 \\
\hline
1992 + 1980 & \text{improvement} & 8.62 \\
\hline
\end{array}

1.54 No solution provided.
1.55 No solution provided.
1.56 No solution provided.
2.1 For program 1, M2 is 2.0 \((10/5)\) times as fast as M1. For program 2, M1 is 1.33 \((4/3)\) times as fast as M2.

2.2 Since we know the number of instructions executed and the time it took to execute the instructions, we can easily calculate the number of instructions per second while running program 1 as \((200 \times 10^6)/10 = 20 \times 10^6\) for M1 and \((160 \times 10^6)/5 = 32 \times 10^6\) for M2.

2.3 We know that Cycles per instruction = Cycles per second / Instructions per second. For M1 we thus have a CPI of \(200 \times 10^6\) cycles per second / \(20 \times 10^6\) instructions per second = 10 cycles per instruction. For M2 we have \(300/32 = 9.4\) cycles per instruction.

2.4 We are given the number of cycles per second and the number of seconds, so we can calculate the number of required cycles for each machine. If we divide this by the CPI we’ll get the number of instructions. For M1, we have 3 seconds \(\times 200 \times 10^6\) cycles / 10 cycles per instruction = \(60 \times 10^6\) instructions per program. For M2, we have 4 seconds \(\times 300 \times 10^6\) cycles / 9.4 cycles per instruction = \(127.7 \times 10^6\) instructions per program.

2.5 M2 is twice as fast as M1, but it does not cost twice as much. M2 is clearly the machine to purchase.

2.6 If we multiply the cost by the execution time, we are multiplying two quantities, for each of which smaller numbers are preferred. For this reason, cost times execution time is a good metric, and we would choose the machine with a smaller value. In the example, we get $10,000 \times 10\) seconds = $100,000 for M1 vs. $15,000 \times 5\) seconds = $75,000 for M2, and thus M2 is the better choice. If we used cost divided by execution time and assume we choose the machine with the larger value, then a machine with a ridiculously high cost would be chosen. This makes no sense. If we choose the machine with the smaller value, then a machine with a ridiculously high execution time would be chosen. This too makes no sense.

2.7 We would define cost-effectiveness as performance divided by cost. This is essentially \((1/\text{Execution time}) \times (1/\text{Cost})\), and in both cases larger numbers are more cost-effective when we multiply.

2.8 We can use the method in Exercise 2.7, but the execution time is the sum of the two execution times.

\[
\text{Executions per second per dollar for M1} = \frac{1}{13 \times 10,000} = \frac{1}{130,000}
\]

\[
\text{Executions per second per dollar for M2} = \frac{1}{9 \times 15,000} = \frac{1}{135,000}
\]

So M1 is slightly more cost-effective, specifically 1.04 times more.
2.9 We do this problem by finding the amount of time that program 2 can be run in an hour and using that for executions per second, the throughput measure.

\[
\text{Executions of P2 per hour} = \frac{3600 \text{ seconds}}{\text{hour}} - \frac{200 \times \text{seconds}}{\text{Execution of P1}}
\]

\[
\text{Executions of P2 per hour on M1} = \frac{3600 \text{ seconds}}{3} - \frac{200 \times 10}{3} = \frac{1600}{3} = 533
\]

\[
\text{Executions of P2 per hour on M2} = \frac{3600 \text{ seconds}}{4} - \frac{200 \times 5}{4} = \frac{2600}{4} = 650
\]

With performance measured by throughput for program 2, machine M2 is \( \frac{650}{533} = 1.2 \) times faster than M1. The cost-effectiveness of the machines is to be measured in units of throughput on program 2 per dollar, so

\[
\text{Cost-effectiveness of M1} = \frac{533}{10,000} = 0.053
\]

\[
\text{Cost-effectiveness of M2} = \frac{650}{15,000} = 0.043
\]

Thus, M1 is more cost-effective than M2. (Machine costs are from Exercise 2.5.)

2.10 For M1 the peak performance will be achieved with a sequence on instructions of class A, which have a CPI of 1. The peak performance is thus 500 MIPS.

For M2, a mixture of A and B instructions, both of which have a CPI of 2, will achieve the peak performance, which is 375 MIPS.

2.11 Let’s find the CPI for each machine first. CPI for M1 = \( \frac{1 + 2 + 3 + 4}{4} = 2.5 \), and

CPI for M2 = \( \frac{2 + 2 + 4 + 4}{4} = 3.0 \). Using CPU time = \( \frac{\text{Instruction count} \times \text{CPI}}{\text{Clock rate}} \), we get the following: CPU time for M1 = \( \frac{\text{Instruction count} \times 2.5}{500 \text{ MHz}} \) = \( \frac{\text{Instruction count}}{200 \text{ million}} \), and

CPU time for M2 = \( \frac{\text{Instruction count} \times 3}{750 \text{ MHz}} \) = \( \frac{\text{Instruction count}}{250 \text{ million}} \).

M2 has a smaller execution time and is thus faster by the inverse ratio of the execution time or \( 250/200 = 1.25 \).

2.12 M1 would be as fast if the clock rate were 1.25 higher, so 500 \times 1.25 = 625 MHz.

2.13 Note: There is an error in Exercise 2.13 on page 92 in the text. The table entry for row c, column 3 (“CPI on M2”) should be 3 instead of 8. This will be corrected in the first reprint of the book. With the corrected value of 3, this solution is valid. Using C1, the CPI on M1 = 5.8 and the CPI on M2 = 3.2. Because M1 has a clock rate twice as fast as that of M2, M1 is 1.10 times as fast. Using C2, the CPI on M1 = 6.4 and the CPI on M2 = 2.9. M2 is
(6.4/2)/2.9 = 1.10 times as fast. Using a third-party product, CPI on M1 = 5.4 and on M2 = 2.8. The third-party compiler is the superior product regardless of machine purchase. M1 is the machine to purchase using the third-party compiler, as it will be 1.04 times faster for typical programs.

2.14 Let I = number of instructions in program and C = number of cycles in program. The six subsets are [clock rate, C] [cycle time, C] [MIPS, I] [CPI, I, clock rate] [CPI, I, cycle time]. Note that in every case each subset has to have at least one rate [CPI, clock rate, cycle time, MIPS] and one absolute [C, I].

2.15 MIPS = \( \frac{\text{Clock rate}}{\text{CPI} \times 10^6} \). Let’s find the CPI for MFP first:

CPI for MFP = \( 0.1 \times 6 + 0.15 \times 4 + 0.05 \times 20 \times 0.7 \times 2 = 3.6 \); of course, the CPI for MNFP is simply 2. So MIPS for MFP = \( \frac{1000}{\text{CPI}} = 278 \) and

MIPS for MNFP = \( \frac{1000}{\text{CPI}} = 500 \).

2.16

<table>
<thead>
<tr>
<th>Instruction class</th>
<th>Frequency on MFP</th>
<th>Count on MFP in millions</th>
<th>Count on MNFP in millions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Floating point multiply</td>
<td>10%</td>
<td>30</td>
<td>900</td>
</tr>
<tr>
<td>Floating point add</td>
<td>15%</td>
<td>45</td>
<td>900</td>
</tr>
<tr>
<td>Floating point divide</td>
<td>5%</td>
<td>15</td>
<td>750</td>
</tr>
<tr>
<td>Integer instructions</td>
<td>70%</td>
<td>210</td>
<td>210</td>
</tr>
<tr>
<td>Totals</td>
<td>100%</td>
<td>300</td>
<td>2760</td>
</tr>
</tbody>
</table>

2.17 Execution time = \( \frac{\text{IC} \times 10^6}{\text{MIPS}} \). So execution time is \( \frac{300}{278} \) = 1.08 seconds, and execution time on MNFP is \( \frac{2760}{500} \) = 5.52 seconds.

2.18 CPI for Mbase = \( 2 \times 0.4 + 3 \times 0.25 + 3 \times 0.25 + 5 \times 0.1 = 2.8 \)

CPI for Mopt = \( 2 \times 0.4 + 2 \times 0.25 + 3 \times 0.25 + 4 \times 0.1 = 2.45 \)

2.19 MIPS for Mbase = \( 500 / 2.8 = 179 \). MIPS for Mopt = \( 600 / 2.45 = 245 \).

2.20 Since it’s the same architecture, we can compare the native MIPS ratings. Mopt is faster by the ratio 245/179 = 1.4.

2.21 This problem can be done in one of two ways. Either find the new mix and adjust the frequencies first or find the new (relative) instruction count and divide the CPI by that. We use the latter.

Ratio of instructions = \( 0.9 \times 0.4 + 0.9 \times 0.25 + 0.85 \times 0.25 + 0.1 \times 0.95 = 0.81 \).

So we can calculate CPI as

\[
\text{CPI} = \frac{2 \times 0.4 \times 0.9 + 3 \times 0.25 \times 0.9 + 3 \times 0.25 \times 0.85 + 5 \times 0.1 \times 0.95}{0.81} = 3.1
\]

2.22 How much faster is Mcomp than Mbase?

CPU time Mbase = \( \frac{\text{Clock rate}}{\text{IC} \times \text{CPI}} = \frac{\text{Clock rate}}{\text{IC} \times 2.8} \)

CPU time Mcomp = \( \frac{\text{Clock rate}}{\text{IC} \times 0.81 \times 3.1} = \frac{\text{Clock rate}}{\text{IC} \times 2.5} \)
So then

\[
\frac{\text{Performance \ Mboth}}{\text{Performance \ Mbase}} = \frac{\text{CPU time \ Mbase}}{\text{CPU time \ Mboth}} = \frac{\text{Clock rate \ IC \times 2.8}}{\text{Clock rate \ IC \times 2.5}} = \frac{2.8}{2.5} = 1.12
\]

2.23 The CPI is different from either Mbase or Mcomp; find that first:

\[
\text{Mboth CPI} = \frac{2 \times 0.4 \times 0.9 + 2 \times 0.25 \times 0.9 + 3 \times 0.25 \times 0.85 + 4 \times 0.1 \times 0.95}{0.81} = 2.7
\]

\[
\frac{\text{Performance \ Mboth}}{\text{Performance \ Mbase}} = \frac{\text{CPU time \ Mbase}}{\text{CPU time \ Mboth}} = \frac{\text{Clock rate \ IC \times 2.8}}{\text{Clock rate \ IC \times 2.2}} = \frac{2.8 \times 600\text{MHz}}{2.2 \times 500\text{MHz}} = 1.5
\]

2.24 First, compute the performance growth after 6 and 8 months. After 6 months = \(1.034^6 = 1.22\). After 8 months = \(1.034^8 = 1.31\). The best choice would be to implement either Mboth or Mopt.

2.25 No solution provided.

2.26 Total execution time of computer A is 1001 seconds; computer B, 110 seconds; computer C, 40 seconds. Computer C is fastest. It’s 25 times faster than computer A and 2.75 times faster than computer B.

2.27 We can just take the GM of the execution times and use the inverse.

\[
\text{GM}(A) = \sqrt[3]{1000} = 32, \quad \text{GM}(B) = \sqrt[3]{1000} = 32, \quad \text{GM}(C) = \sqrt[3]{400} = 20, \quad \text{so C is fastest.}
\]

2.28 A, B: B has the same performance as A. If we run program 2 once, how many times should we run program 1: \(x + 1000 = 10x + 100\), or \(x = 100\). So the mix is 99% program 1, 1% program 2.

B, C: C is faster by the ratio of \(\frac{32}{20} = 1.6\). Program 2 is run once, so we have

\[
10x + 100 = 1.6 \times (20x + 20), \quad x = 3.1 \text{\ times. So the mix is 76% program 1 and 24% program 2.}
\]

A, C: C is also faster by 1.6 here. We use the same equation, but with the proper times:

\[
x + 1000 = 1.6 \times (20x + 20), \quad x = 31.2 \text{. So the mix is 97% program 1 and 3% program 2. Note that the mix is very different in each case!}
\]

2.29

<table>
<thead>
<tr>
<th>Program</th>
<th>Weight</th>
<th>Computer A</th>
<th>Computer B</th>
<th>Computer C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program 1 (seconds)</td>
<td>10</td>
<td>1</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>Program 2 (seconds)</td>
<td>1</td>
<td>1000</td>
<td>100</td>
<td>20</td>
</tr>
<tr>
<td>Weighted AM</td>
<td>9.18</td>
<td>18.2</td>
<td>20</td>
<td>20</td>
</tr>
</tbody>
</table>

So B is fastest; it is 1.10 times faster than C and 5.0 times faster than A. For an equal number of executions of the programs, the ratio of total execution times A:B:C is 1001:110:40, thus C is 2.75 times faster than B and 25 times faster than A.
2.30 Equal time on machine A:

<table>
<thead>
<tr>
<th>Program</th>
<th>Weight</th>
<th>Computer A</th>
<th>Computer B</th>
<th>Computer C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program 1 (seconds)</td>
<td>1</td>
<td>1</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>Program 2 (seconds)</td>
<td>1/1000</td>
<td>1000</td>
<td>100</td>
<td>20</td>
</tr>
<tr>
<td>Weighted AM</td>
<td>2</td>
<td>10.1</td>
<td>20</td>
<td></td>
</tr>
</tbody>
</table>

This makes A the fastest.

Now with equal time on machine B:

<table>
<thead>
<tr>
<th>Program</th>
<th>Weight</th>
<th>Computer A</th>
<th>Computer B</th>
<th>Computer C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program 1 (seconds)</td>
<td>1</td>
<td>1</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>Program 2 (seconds)</td>
<td>1/10</td>
<td>1000</td>
<td>100</td>
<td>20</td>
</tr>
<tr>
<td>Weighted AM</td>
<td>91.8</td>
<td>18.2</td>
<td>20</td>
<td></td>
</tr>
</tbody>
</table>

Machine B is the fastest.

Comparing them to unweighted numbers, we notice that this weighting always makes the base machine fastest, and machine C second. The unweighted mean makes machine C fastest (and is equivalent to equal time weighting on C).

2.31 Assume 100 instructions, then the number of cycles will be $90 \times 4 + 10 \times 12 = 480$ cycles. Of these, 120 are spent doing multiplication, and thus 25% of the time is spent doing multiplication.

2.32 Unmodified for 100 instructions we are using 480 cycles, and if we improve multiplication it will only take 420 cycles. But the improvement increases the cycle time by 20%. Thus we should not perform the improvement as the original is $1.2(420)/480 = 1.05$ times faster than the improvement!

2.33 No solution provided.

2.34 No solution provided.

2.35 No solution provided.

2.36 No solution provided.

2.37 No solution provided.

2.38

<table>
<thead>
<tr>
<th>Program</th>
<th>Computer A</th>
<th>Computer B</th>
<th>Computer C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>1</td>
<td>0.5</td>
</tr>
<tr>
<td>2</td>
<td>0.1</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

2.39 The harmonic mean of a set of rates,

$$HM = \frac{1}{n} \sum_{i=1}^{n} \frac{1}{\text{Rate}_i} = \frac{1}{n} \sum_{i=1}^{n} \frac{1}{\text{Time}_i} = \frac{1}{\text{AM}},$$

where AM is the arithmetic mean of the corresponding execution times.
2.40 No solution provided.
2.41 No solution provided.
2.42 No solution provided.
2.43 No solution provided.

2.44 Using Amdahl's law (or just common sense) we can determine the following:

- Speedup if we improve only multiplication = \( \frac{100}{30 + 50 + 20/4} = \frac{100}{85} = 1.18 \).
- Speedup if we only improve memory access = \( \frac{100}{100 - (50 - 50/2)} = \frac{100}{75} = 1.33 \).
- Speedup if both improvements are made = \( \frac{100}{30 + 50/2 + 20/4} = \frac{100}{60} = 1.67 \).

2.45 The problem is solved algebraically and results in the equation

\[
100 / (Y + (100 - X - Y) + X/4) = 100 / (X + (100 - X - Y) + Y/2)
\]

where \( X \) = multiplication percentage and \( Y \) = memory percentage. Solving, we get memory percentage = \( 1.5 \times \) multiplication percentage. Many examples thus exist, e.g., multiplication = 20%, memory = 30%, other = 50%, or multiplication = 30%, memory = 45%, other = 25%, etc.

2.46 Speed-up = \( \frac{\text{Execution time before improvement}}{\text{Execution time after improvement}} \)

Rewrite the execution time equation:

\[
\text{Execution time after improvement} = \frac{\text{Execution time affected by improvement}}{\text{Amount of improvement}} + \text{Execution time unaffected}
\]

\[
= \frac{\text{Execution time affected}}{\text{Amount of improvement}} + \frac{\text{Amount of improvement} \times \text{Execution time unaffected}}{\text{Amount of improvement}}
\]
Rewrite execution time affected by improvement as execution time before improvement \( x f \), where \( f \) is the fraction affected. Similarly execution time unaffected.

\[
\frac{\text{Execution time before improvement} \times f}{\text{Amount of improvement}} + \text{Execution time before improvement} \times (1 - f)
\]

= \frac{\text{Execution time before improvement} \times f}{\text{Amount of improvement}} + \text{Execution time before improvement} \times (1 - f)

= \frac{\text{Execution time before improvement} \times f}{\text{Amount of improvement}} + \text{Execution time before improvement} \times (1 - f)

= \left( \frac{f}{\text{Amount of improvement}} + (1 - f) \right) \times \text{Execution time before improvement}

\text{Speedup} = \frac{\text{Execution time before improvement}}{\left( \frac{f}{\text{Amount of improvement}} + (1 - f) \right) \times \text{Execution time before improvement}}

\text{Speedup} = \frac{1}{\left( \frac{f}{\text{Amount of improvement}} + (1 - f) \right)}

The denominator has two terms: the fraction improved \((f)\) divided by the amount of the improvement and, the fraction unimproved \((1 - f)\).
3.1 The program computes the sum of odd numbers up to the largest odd number smaller than or equal to \( n \), e.g., \( 1 + 3 + 5 + \ldots + n \) (or \( n - 1 \) if \( n \) is even). There are many alternative ways to express this summation. For example, an equally valid answer is that the program calculates \( \left( \text{ceiling}(n/2) \right)^2 \).

3.2 The code determines the most frequent word appearing in the array and returns it in \$v1\) and its multiplicity in \$v0\).

3.3 Ignoring the four instructions before the loops, we see that the outer loop (which iterates 5000 times) has four instructions before the inner loop and six after in the worst case. The cycles needed to execute these are \( 1 + 2 + 1 + 1 = 5 \) and \( 1 + 2 + 1 + 1 + 1 + 2 = 8 \), for a total of 13 cycles per iteration, or \( 5000 \times 13 \) for the outer loop. The inner loop requires \( 1 + 2 + 2 + 1 + 1 + 2 = 9 \) cycles per iteration and it repeats \( 5000 \times 5000 \) times, for a total of \( 9 \times 5000 \times 500 \) cycles. The overall execution time is thus approximately \( (5000 \times 13 + 9 \times 5000 \times 5000) / (500 \times 10^6) = .45 \) sec. Note that the execution time for the inner loop is really the only code of significance.

3.4 \texttt{addi \$t0,\$t1,100} \# register \$t0 = \$t1 + 100

3.5 The base address of \( x \), in binary, is 0000 0000 0011 1101 0000 1001 0000 0000, which implies that we must use \texttt{lui}:

\begin{verbatim}
lui \$t1, 0000 0000 0011 1101
ori \$t1, \$t1, 0000 1001 0000 0000
lw \$t2, 44(\$t1)
add \$t2, \$t2, \$t0
sw \$t2, 40(\$t1)
\end{verbatim}

3.6 \texttt{addi \$v0,\$zero,-1} \# Initialize to avoid counting zero word

\texttt{loop: \texttt{lw \$v1,0($a0)} \# Read next word from source}

\texttt{addi \$v0,\$v0,1 \# Increment count words copied}

\texttt{sw \$v1,0($a1)} \# Write to destination

\texttt{addi \$a0,\$a0,4 \# Advance pointer to next source}

\texttt{addi \$a1,\$a1,4 \# Advance pointer to next dest}

\texttt{bne \$v1,\$zero,loop\# Loop if the word copied \neq zero}

Bugs:

1. Count (\$v0) is not initialized.

2. Zero word is counted. (1 and 2 fixed by initializing \$v0 to \(-1\).

3. Source pointer (\$a0) incremented by 1, not 4.

4. Destination pointer (\$a1) incremented by 1, not 4.

3.7

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{lw $v1,0($a0)}</td>
<td>I</td>
<td>35</td>
<td>4</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>\texttt{addi $v0,$v0,1}</td>
<td>I</td>
<td>8</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>\texttt{sw $v1,0($a1)}</td>
<td>I</td>
<td>43</td>
<td>5</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>\texttt{addi $a0,$a0,4}</td>
<td>I</td>
<td>8</td>
<td>4</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>\texttt{addi $a1,$a1,4}</td>
<td>I</td>
<td>8</td>
<td>4</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>\texttt{bne $v1,$zero,loop}</td>
<td>I</td>
<td>5</td>
<td>3</td>
<td>0</td>
<td>-20</td>
</tr>
</tbody>
</table>
3.8  count = -1;
    do {
        temp = *source;
        count = count + 1;
        *destination = temp;
        source = source + 1;
        destination = destination + 1;
    } while (temp != 0);

3.9  The C loop is
    while (save[i] == k)
        i = i + j;

with i, j, and k corresponding to registers $s3, $s4, and $s5 and the base of the array
save in $s6. The assembly code given in the example is

**Code before**

```
Loop: add $t1, $s3, $s3  # Temp reg $t1 = 2 * i
      add $t1, $t1, $t1  # Temp reg $t1 = 4 * i
      add $t1, $t1, $s6  # $t1 = address of save[i]
      lw  $t0, 0($t1)    # Temp reg $t0 = save[i]
      bne $t0, $s5, Exit # go to Exit if save[i] ≠ k
      add $s3, $s3, $s4  # i = i + j
Loop  # go to Loop

Exit:
```

Number of instructions executed if save[i + m * j] does not equal k for m = 10 and
does equal k for 0 ≤ m ≤ 9 is 10 × 7 + 5 = 75, which corresponds to 10 complete itera-
tions of the loop plus a final pass that goes to Exit at the bne instruction before updat-
ing i. Straightforward rewriting to use at most one branch or jump in the loop yields

**Code after**

```
add $t1, $s3, $s3  # Temp reg $t1 = 2 * i
add $t1, $t1, $t1  # Temp reg $t1 = 4 * i
add $t1, $t1, $s6  # $t1 = address of save[i]
lw  $t0, 0($t1)    # Temp reg $t0 = save[i]
beq $t0, $s5, Loop # go to Loop if save[i] = k
Loop: add $s3, $s3, $s4  # i = i + j
add $t1, $s3, $s3  # Temp reg $t1 = 2 * i
add $t1, $t1, $t1  # Temp reg $t1 = 4 * i
add $t1, $t1, $s6  # $t1 = address of save[i]
lw  $t0, 0($t1)    # Temp reg $t0 = save[i]
beq $t0, $s5, Loop # go to Loop if save[i] = k
Exit:
```

The number of instructions executed by this new form of the loop is 5 + 10 × 6 = 65. If
4 × j is computed before the loop, then further saving in the loop body is possible.

**Code after further improvement**

```
add $t2, $s4, $s4  # Temp reg $t2 = 2 * j
add $t2, $t2, $t2  # Temp reg $t2 = 4 * j
add $t1, $s3, $s3  # Temp reg $t1 = 2 * i
add $t1, $t1, $t1  # Temp reg $t1 = 4 * i
```
add $t1, $t1, $s6  # $t1 = address of save[i]
lw $t0, 0($t1)     # Temp reg $t0 = save[i]
bne $t0, $s5, Exit # go to Exit if save[i] ≠ k
Loop: add $t1, $t1, $t2 # $t1 = address of save [i + m * j]
lw $t0, 0($t1)     # Temp reg $t0 = save[i]
beq $t0, $s5, Loop # go to Loop if save[i] = k

Exit:

The number of instructions executed is now 7 + 10 × 3 = 37.

3.10

<table>
<thead>
<tr>
<th>Pseudoinstruction</th>
<th>What it accomplishes</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>move $t5, $t3</td>
<td>$t5 = $t3</td>
<td>add $t5, $t3, $zero</td>
</tr>
<tr>
<td>clear $t5</td>
<td>$t5 = 0</td>
<td>add $t5, $zero, $zero</td>
</tr>
<tr>
<td>li $t5, small</td>
<td>$t5 = small</td>
<td>add $t5, $zero, small</td>
</tr>
<tr>
<td>li $t5, big</td>
<td>$t5 = big</td>
<td>li $t5, upper_half(big)</td>
</tr>
<tr>
<td>lw $t5, big($t3)</td>
<td>$t5 = Memory[$t3 + big]</td>
<td>li $at, big</td>
</tr>
<tr>
<td>addi $t5, $t3, big</td>
<td>$t5 = $t3 + big</td>
<td>li $at, $t5, $at</td>
</tr>
<tr>
<td>beq $t5, small, L</td>
<td>if ($t5 = small) go to L</td>
<td>beq $t5, $at, L</td>
</tr>
<tr>
<td>beq $t5, big, L</td>
<td>if ($t5 = big) go to L</td>
<td>beq $t5, $zero, L</td>
</tr>
<tr>
<td>ble $t5, $t3, L</td>
<td>if ($t5 ≤ $t3) go to L</td>
<td>slt $at, $t3, $t5</td>
</tr>
<tr>
<td>bgt $t5, $t3, L</td>
<td>if ($t5 &gt; $t3) go to L</td>
<td>bne $at, $zero, L</td>
</tr>
<tr>
<td>bge $t5, $t3, L</td>
<td>if ($t5 ≥ $t3) go to L</td>
<td>slt $at, $t5, $t3</td>
</tr>
</tbody>
</table>

Note: In the solutions, we make use of the \textit{li} instruction, which should be implemented as shown in rows 3 and 4.

3.11 The fragment of C code is

\[\text{for} \ (i=0; \ i\leq100; \ i=i+1) \ \{a[i] = b[i] + c;\}\]

with \texttt{a} and \texttt{b} arrays of words at base addresses $a0$ and $a1$, respectively. First initialize \texttt{i} to 0 with i kept in $t0$:

\[
\text{add} \ \texttt{t0}, \ \texttt{zero}, \ \texttt{zero} \ # \text{Temp reg} \ \texttt{t0} = 0
\]

Assume that $s0$ holds the address of \texttt{c} (if $s0$ is assumed to hold the value of \texttt{c}, omit the following instruction):

\[
lw \ \texttt{t1}, \ 0($s0) \ # \text{Temp reg} \ \texttt{t1} = c
\]

To compute the byte address of successive array elements and to test for loop termination, the constants 4 and 401 are needed. Assume they are placed in memory when the program is loaded:

\[
lw \ \texttt{t2}, \ \text{AddressConstant4}($\text{zero}) \ # \text{Temp reg} \ \texttt{t2} = 4
\]
\[
lw \ \texttt{t3}, \ \text{AddressConstant401}($\text{zero}) \ # \text{Temp reg} \ \texttt{t3} = 401
\]
In section 3.8 the instructions `addi` (add immediate) and `slti` (set less than immediate) are introduced. These instructions can carry constants in their machine code representation, saving a load instruction and use of a register. Now the loop body accesses array elements, performs the computation, and tests for termination:

```
Loop: add $t4, $a1, $t0  # Temp reg $t4 = address of b[i]
lw  $t5, 0($t4)      # Temp reg $t5 = b[i]
add $t6, $t5, $t1    # Temp reg $t6 = b[i] + c
```

This `add` instruction would be `add $t6, $t5, $s0` if it were assumed that $s0 holds the value of \( c \). Continuing the loop:

```
add $t7, $a0, $t0  # Temp reg $t7 = address of a[i]
sw  $t6, 0($t7)    # a[i] = b[i] + c
add $t0, $t0, $t2  # i = i + 4
slt $t8, $t0, $t3  # $t8 = 1 if $t0 < 401, i.e., i \leq 100
bne $t8, $zero, Loop # go to Loop if i \leq 100
```

The number of instructions executed is \( 4 + 101 \times 8 = 812 \). The number of data references made is \( 3 + 101 \times 2 = 205 \).

3.12 The problem is that we are using PC-relative addressing, so if the address of there is too far away, we won’t be able to use 16 bits to describe where it is relative to the PC. One simple solution would be

```
here: bne $t1, $t2, skip
       j there
skip:
     ...
there: add $t1, $t1, $t1
```

This will work as long as our program does not cross the 256-MB address boundary described in the elaboration on page 150.

3.13 Let \( I \) be the number of instructions taken by gcc on the unmodified MIPS. This decomposes into \( .48I \) arithmetic instructions, \( .33I \) data transfer instructions, \( .17I \) conditional branches, and \( .02I \) jumps. Using the CPIs given for each instruction class, we get a total of \( (.48 \times 1.0 + .33 \times 1.4 + .17 \times 1.7 + .02 \times 1.2) \times I \) cycles; if we call the unmodified machine’s cycle time \( C \) seconds, then the time taken on the unmodified machine is \( (.48 \times 1.0 + .33 \times 1.4 + .17 \times 1.7 + .02 \times 1.2) \times I \times C \) seconds. Changing some fraction, \( f \) (namely \( .25 \)), of the data transfer instructions into the autoincrement or autodecrement version will leave the number of cycles spent on data transfer instructions unchanged. However, each of the \( .33 \times I \times f \) data transfer instructions that is changed corresponds to an arithmetic instruction that can be eliminated. So, there are now only \( (.48 - (.33 \times f)) \times I \) arithmetic instructions, and the modified machine, with its cycle time of \( 1.1 \times C \) seconds, will take \( ((.48 - .33f) \times 1.0 + .33 \times 1.4 + .17 \times 1.7 + .02 \times 1.2) \times I \times 1.1 \times C \) seconds to execute gcc. When \( f \) is \( .25 \), the unmodified machine is 2.8% faster than the modified one.

3.14 From Figure 3.38, 33% of all instructions executed by spice are data access instructions. Thus, for every 100 instructions there are \( 100 + 33 = 133 \) memory accesses: one to read each instruction and 33 to access data.

a. The percentage of all memory accesses that are for data = \( 33/133 = 25\% \).
b. Assuming two-thirds of data transfers are loads, the percentage of all memory accesses that are reads = \( \frac{100 + \left( \frac{33}{3} \times \frac{2}{3} \right)}{133} = 92\% \).

3.15 From Figure 3.38, 41% of all instructions executed by spice are data access instructions. Thus, for every 100 instructions there are 100 + 41 = 141 memory accesses: one to read each instruction and 41 to access data.

a. The percentage of all memory accesses that are for data = \( \frac{41}{141} = 29\% \).

b. Assuming two-thirds of data transfers are loads, the percentage of all memory accesses that are reads = \( \frac{100 + \left( \frac{41}{3} \times \frac{2}{3} \right)}{141} = 90\% \).

3.16 Effective CPI = \( \sum_{\text{Instruction classes}} \left( \frac{\text{CPI}_{\text{class}} \times \text{Frequency of execution}_{\text{class}}}{1.1} \right) \)

For gcc, CPI = 1.0 \( \times 0.48 + 1.4 \times 0.33 + 1.7 \times 0.17 + 1.2 \times 0.02 = 1.3 \). For spice, CPI = 1.0 \( \times 0.5 + 1.4 \times 0.41 + 1.7 \times 0.08 + 1.2 \times 0.01 = 1.2 \).

3.17 Let the program have \( n \) instructions.

Let the original clock cycle time be \( t \).

Let \( N \) be percent loads retained.

\[
\begin{align*}
\text{exec}_{\text{old}} &= n \times \text{CPI} \times t \\
\text{exec}_{\text{new}} &= (0.78 n + N \times 0.22 n) \times \text{CPI} \times 1.1 t \\
\text{exec}_{\text{new}} &\leq \text{exec}_{\text{old}} \\
(0.78 n + N \times 0.22 n) \times \text{CPI} \times 1.1 t &\leq n \times \text{CPI} \times t \\
(0.78 + N \times 0.22) \times 1.1 &\leq 1 \\
0.78 + N \times 0.22 &\leq \frac{1}{1.1} \\
N \times 0.22 &\leq \frac{1}{1.1} - 0.78 \\
N &\leq \frac{1}{1.1} - 0.78 \\
&\leq \frac{1}{1.1} - \frac{0.78}{0.22} \\
N &\leq \frac{1 - 1.1 \times 0.78}{1.1 \times 0.22} \\
N &\leq 0.587
\end{align*}
\]

We need to eliminate at least 41.3% of loads.

3.18 No solution provided.
3.19

<table>
<thead>
<tr>
<th>Accumulator</th>
<th>Instruction</th>
<th>Code bytes</th>
<th>Data bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>load b</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>add c</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>store a</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>add c</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>store b</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>neg</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>add a</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>store d</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Total:</td>
<td>22</td>
<td>28</td>
<td></td>
</tr>
</tbody>
</table>

Code size is 22 bytes, and memory bandwidth is 22 + 28 = 50 bytes.

<table>
<thead>
<tr>
<th>Stack</th>
<th>Instruction</th>
<th>Code bytes</th>
<th>Data bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>push b</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>push c</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>dup</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>pop a</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>push c</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>dup</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>pop b</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>neg</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>push a</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>pop d</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Total:</td>
<td>27</td>
<td>28</td>
<td></td>
</tr>
</tbody>
</table>

Code size is 27 bytes, and memory bandwidth is 27 + 28 = 55 bytes.

<table>
<thead>
<tr>
<th>Memory-Memory</th>
<th>Instruction</th>
<th>Code bytes</th>
<th>Data bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>add a, b, c</td>
<td>7</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>add b, a, c</td>
<td>7</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>sub d, a, b</td>
<td>7</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>Total:</td>
<td>21</td>
<td>36</td>
<td></td>
</tr>
</tbody>
</table>

Code size is 21 bytes, and memory bandwidth is 21 + 36 = 57 bytes.
The load-store machine has the lowest amount of data traffic. It has enough registers that it only needs to read and write each memory location once. On the other hand, since all ALU operations must be separate from loads and stores, and all operations must specify three registers or one register and one address, the load-store has the worst code size. The memory-memory machine, on the other hand, is at the other extreme. It has the fewest instructions (though also the largest number of bytes per instruction) and the largest number of data accesses.

3.20 To know the typical number of memory addresses per instruction, the nature of a typical instruction must be agreed upon. For the purpose of categorizing computers as 0-, 1-, 2-, 3-address machines, an instruction that takes two operands and produces a result, for example, add, is traditionally taken as typical.

**Accumulator**: An add on this architecture reads one operand from memory, one from the accumulator, and writes the result in the accumulator. Only the location of the operand in memory need be specified by the instruction. CATEGORY: 1-address architecture.

**Memory-memory**: Both operands are read from memory and the result is written to memory, and all locations must be specified. CATEGORY: 3-address architecture.

**Stack**: Both operands are read (removed) from the stack (top of stack and next to top of stack), and the result is written to the stack (at the new top of stack). All locations are known; none need be specified. CATEGORY: 0-address architecture.

**Load-store**: Both operands are read from registers and the result is written to a register. Just like memory-memory, all locations must be specified; however, location addresses are much smaller—5 bits for a location in a typical register file versus 32 bits for a location in a common memory. CATEGORY: 3-address architecture.

3.21 Figure 3.15 shows decimal values corresponding to ACSII characters.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Y</th>
<th>T</th>
<th>E</th>
<th>I</th>
<th>S</th>
<th>B</th>
<th>I</th>
<th>T</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>65</td>
<td>32</td>
<td>98</td>
<td>121</td>
<td>116</td>
<td>101</td>
<td>32</td>
<td>101</td>
<td>115</td>
<td>32</td>
<td>56</td>
</tr>
</tbody>
</table>

3.22 No solution provided.

3.23 No solution provided.

3.24 No solution provided.
3.25 Here is the C code for `itoa`, as taken from *The C Programming Language* by Kernighan and Ritchie:

```c
void reverse( char *s )
{
    int c, i, j;

    for( i = 0, j = strlen(s)-1; i < j; i++, j-- ) {
        c=s[i];
        s[i]=s[j];
        s[j] = c;
    }
}

void itoa( int n, char *s )
{
    int i, sign;

    if( ( sign = n ) < 0 )
    {
        n = –n;
        i = 0;
        do {
            s[i++] = n % 10 + '0';
        } while( ( n /= 10 ) > 0 );
        if( sign < 0 )
            s[i++] = '–';
        s[i] = '\0';
        reverse( s );
    }
}
```

The MIPS assembly code, along with a main routine to test it, might look something like this:

```assembly
.data
hello: .ascii  "\nEnter a number:"
newln: .asciiz "\n"
str: .space  32

.text
reverse: # Expects string to
    # reverse in $a0
    # s = i = $a0
    # j = $t2

        addi     $t2, $a0, -1 # j = s - 1;
        lbu      $t3, 1($t2) # while( *(j+1) )
        beqz     $t3, end_strlen
strlen_loop:
        addi     $t2, $t2, 1 # j++;
        lbu      $t3, 1($t2)
        bnez     $t3, strlen_loop
```
end_strlen: # now j =
    # &s[strlen(s)-1]
    bge $a0, $t2, end_reverse # while( i < j )
    # {
reverse_loop:
    lbu $t3, ($a0) #    $t3 = *i;
    lbu $t4, ($t2) #    $t4 = *j;
    sb $t3, ($t2) #    *j = $t3;
    sb $t4, ($a0) #    *i = $t4;
    addi $a0, $a0, 1 #    i++;
    addi $t2, $t2, -1 #    j--;
    blt $a0, $t2, reverse_loop # }
end_reverse:
    jr $31

.globl itoa # $a0 = n
itoa:    addi $29, $29, -4 # $a1 = s
    sw $31, 0($29)
    move $t0, $a0 # sign = n;
    move $t3, $a1 # $t3 = s;
    bgez $a0, non_neg # if( sign < 0 )
    sub $a0, $0, $a0 #    n = –n
non_neg:
    li $t2, 10
    itoa_loop: # do {
        div $a0, $t2 #    lo = n / 10;
        #    hi = n % 10:
        mfhi $t1
        mflo $a0
        addi $t1, $t1, 48 #    $t1 =
        #    '0' + n % 10;
        sb $t1, 0($a1) #    *s = $t1;
        addi $a1, $a1, 1 #    s++;
        bnez $a0, itoa_loop # } while( n )
        bgez $t0, non_neg2 # if( sign < 0 )
        # {
        li $t1, '-' #    *s = '-';
        sb $t1, 0($a1) #    s++;
        addi $al, $a1, 1 # }
non_neg2:
    sb $0, 0($a1)
    move $a0, $t3
    jal reverse # reverse( s );
    lw $31, 0($29)
    addi $29, $29, 4
    jr $31
.globl main
One common problem that occurred was to treat the string as a series of words rather than as a series of bytes. Each character in a string is a byte. One zero byte terminates a string. Thus when people stored ASCII codes one per word and then attempted to invoice the print_str system call, only the first character of the number printed out.

3.26

# Description: Computes the Fibonacci function using a recursive process.
# Function: F(n) = 0, if n = 0;
#          1, if n = 1;
#          F(n-1) + F(n-2), otherwise.
# Input: n, which must be a non-negative integer.
# Output: F(n).
# Preconditions: none
# Instructions: Load and run the program in SPIM, and answer the prompt.
# Algorithm for main program:
#   print prompt
#   call fib(read) and print result.
# Register usage:
#   $a0 = n (passed directly to fib)
#   $s1 = f(n)
.data
.align 2
# Data for prompts and output description
prmpt1: .asciiz "This program computes the Fibonacci function."
prmpt2: .asciiz "Enter value for n: 

descr: .asciiz "fib(n) = "
.text
.align 2
.globl __start
__start:
# Print the prompts
    li $v0, 4 # print_str system service ...
    la $a0, prmpt1 # ... passing address of first prompt
    syscall
    li $v0, 4 # print_str system service ...
    la $a0, prmpt2 # ... passing address of 2nd prompt
    syscall
# Read n and call fib with result
    li $v0, 5 # read_int system service
    syscall
    move $a0, $v0 # $a0 = n = result of read
    jal fib # call fib(n)
    move $s1, $v0 # $s0 = fib(n)
# Print result
    li $v0, 4 # print_str system service ...
    la $a0, descr # ... passing address of output descriptor
    syscall
    li $v0, 1 # print_int system service ...
    move $a0, $s # ... passing argument fib(n)
    syscall
# Call system – exit
    li $v0, 10
    syscall
# Algorithm for Fib(n):
#  if (n == 0) return 0
#  else if (n == 1) return 1
#  else return fib(n–1) + fib(n–2).
#
# Register usage:
#  $a0 = n (argument)
#  $t1 = fib(n–1)
#  $t2 = fib(n–2)
#  $v0 = 1 (for comparison)
#
# Stack usage:
# 1.  push return address, n, before calling fib(n–1)
# 2.  pop n
# 3.  push n, fib(n–1), before calling fib(n–2)
# 4.  pop fib(n–1), n, return address

fib:        bne $a0, $zero, fibne0 # if n == 0 ...
            move $v0, $zero # ... return 0
            jr $31
fibne0:     # Assert: n != 0
            li $v0, 1
            bne $a0, $v0, fibnel # if n == 1 ...
            jr $31 # ... return 1
fibnel:     # Assert: n > 1
## Compute fib(n–1)
```
addi $sp, $sp, -8  # push ...
sw $ra, 4($sp)     # ... return address
sw $a0, 0($sp)    # ... and n
addi $a0, $a0, -1  # pass argument n-1 ...
jal fib           # ... to fib
move $t1, $v0     # $t1 = fib(n-1)
lw $a0, 0($sp)    # pop n
addi $sp, $sp, 4  # ... from stack
```

## Compute fib(n–2)
```
addi $sp, $sp, -8  # push ...
sw $a0, 4($sp)     # ... n
sw $t1, 0($sp)    # ... and fib(n-1)
addi $a0, $a0, -2  # pass argument n-2 ...
jal fib           # ... to fib
move $t2, $v0     # $t2 = fib(n-2)
lw $t1, 0($sp)    # pop fib(n-1) ...
lw $a0, 4($sp)    # ... n
lw $ra, 8($sp)    # ... and return address
addi $sp, $sp, 12 # ... from stack
```

## Return fib(n–1) + fib(n–2)
```
add $v0, $t1, $t2  # $v0 = fib(n) = fib(n-1) + fib(n-2)
jr $31             # return to caller
```

3.27

# Description: Computes the Fibonacci function using an iterative process.
# Function: F(n) = 0, if n = 0;
# 1, if n = 1;
# F(n–1) + F(n–2), otherwise.
# Input: n, which must be a non-negative integer.
# Output: F(n).
# Preconditions: none
# Instructions: Load and run the program in SPIM, and answer the prompt.
# Algorithm for main program:
# print prompt
# call fib(1, 0, read) and print result.
# Register usage:
# $a2 = n (passed directly to fib)
# $s1 = f(n)
.data
.align 2
# Data for prompts and output description
prmpt1: .asciiz "This program computes the Fibonacci function."
prmpt2: .asciiz "Enter value for n: "
descr: .asciiz "fib(n) = "
.text
.align 2
.globl __start
__start:

# Print the prompts
li $v0, 4 # print_str system service ...
l $a0, prmpt1 # ... passing address of first prompt
syscall
li $v0, 4 # print_str system service ...
l $a0, prmpt2 # ... passing address of 2nd prompt
syscall

# Read n and call fib with result
li $v0, 5 # read_int system service
syscall
move $a2, $v0 # $a2 = n = result of read
li $a1, 0 # $a1 = fib(0)
li $a0, 1 # $a0 = fib(1)
jal fib # call fib(n)
move $s1, $v0 # $s0 = fib(n)

# Print result
li $v0, 4 # print_str system service ...
l $a0, descr # ... passing address of output
syscall
li $v0, 1 # print_int system service ...
mov $a0, $s1 # ... passing argument fib(n)
systemcall

# Call system - exit
li $v0, 10
syscall

# Algorithm for Fib(a, b, count):
# if (count == 0) return b
# else return fib(a + b, a, count – 1).
#
# Register usage:
# $a0 = a = fib(n-1)
# $a1 = b = fib(n-2)
# $a2 = count (initially n, finally 0).
# $t1 = temporary a + b

fib:  bne $a2, $zero, fibne0 # if count == 0 ...
mov $v0, $a1 # ... return b
jr $31

fibne0: # Assert: n != 0
add $a2, $a2, -1 # count = count - 1
add $t1, $a0, $a1 # $t1 = a + b
mov $a1, $a0 # b = a
mov $a0, $t1 # a = a + old b
j fib # tail call fib(a+b, a, count-1)

3.28 No solution provided.
3.29

```assembly
start: sbn temp, b, +1  # Sets temp = -b, always goes to next instruction
sbn a, temp, +1      # Sets a = a - temp = a - (-b) = a + b
```

3.30 There are a number of ways to do this, but this is perhaps the most concise and elegant:

```assembly
sbn  c, c, +1        # c = 0;
sbn  tmp, tmp, +1    # tmp = 0;
loop: sbn  b, one, end # while (b-- > 0)
sbn  tmp, a, loop    # c = a; /* always continue */
end:   sbn  c, tmp, +1 # c = -tmp; /* = a × b */
```
4.1 In the manner analogous to that used in the example on page 214, the number $512_{\text{ten}} = 5 \times 10^2 + 1 \times 10^1 + 2 \times 10^0$. Given that $1_{\text{two}} = 10^0$, $1010_{\text{two}} = 10^1$, and $1100100_{\text{two}} = 10^2$, we have

$$
512_{\text{ten}} = 5 \times 11000000 + 1 \times 1001 + 2 \times 1_{\text{two}}
= 1100100 + 1010 + 1
= 11000000
$$

The number $512_{\text{ten}}$ is positive, so the sign bit is 0, and sign extension yields the 32-bit two’s complement result $512_{\text{ten}} = 00000000000000000000001000000$.

4.2 This exercise can be solved with exactly the same approach used by the solution to Exercise 4.1, with the sign and sign extension being 1 digits. Because memorizing the base 10 representations of powers of 2 is useful for many hardware and software tasks, another conversion technique is widely used.

Let $N$ be the magnitude of the decimal number, $x[i]$ be the $i$th bit of the binary representation of $N$, and $m$ be the greatest integer such that $2^m \leq N$. Then

```plaintext
for (i = m; i >= 0; i = i - 1) {
    if ($2^i \leq N$) $x[i] = 1$;
    else $x[i] = 0$;
    $N = N - 2^i$;
}
```
Part II: Solutions Guide

For \( -1023_{\text{ten}} \), \( N = 1023 \)

\[ \begin{align*}
-512 & \Rightarrow x[9] = 1 \\
-256 & \Rightarrow x[8] = 1 \\
-128 & \Rightarrow x[7] = 1 \\
-64 & \Rightarrow x[6] = 1 \\
-32 & \Rightarrow x[5] = 1 \\
-16 & \Rightarrow x[4] = 1 \\
-8 & \Rightarrow x[3] = 1 \\
-4 & \Rightarrow x[2] = 1 \\
-2 & \Rightarrow x[1] = 1 \\
-1 & \Rightarrow x[0] = 1 \\
0 & \text{Done}
\end{align*} \]

So \( N = 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0011 \ 1111 \ 1111_{\text{two}} \)

Thus \( -1023_{\text{ten}} = 1111 \ 1111 \ 1111 \ 1111 \ 1110 \ 0000 \ 0011_{\text{two}} \)

4.3 Using the method of the solution to either Exercise 4.1 or Exercise 4.2, \(-4,000,000_{\text{ten}}\)

\( = 1111 \ 1111 \ 1110 \ 0010 \ 1111 \ 0111 \ 0000 \ 0000_{\text{two}} \)

4.4 We could substitute into the formula at the bottom of page 213, \((x_{31} \times -2^{31}) + (x_{30} \times 2^{30}) + (x_{29} \times 2^{29}) + \ldots + (x_{1} \times 2^{1}) + (x_{0} \times 2^{0})\), to get the answer. Because this two’s complement number has predominantly 1 digits, the formula will have many nonzero terms to add. The negation of this number will have mostly 0 digits, so using the negation shortcut in the Example on page 216 first and remembering the original sign will save enough work in the formula to be a good strategy. Thus,

\[ \text{Negating} \quad 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111_{\text{two}} \]

\[ = 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0001 \ 1111 \ 1100_{\text{two}} \]

Then the nonzero terms of the formula are

\[ 2^8 + 2^7 + 2^6 + 2^5 + 2^4 + 2^2 = 500 \]

and the original sign is 1 meaning negative, so

\( 1111 \ 1111 \ 1111 \ 1111 \ 1110 \ 0000 \ 1100_{\text{two}} = -500_{\text{ten}} \)

4.5 Here, negating first as in the solution for Exercise 4.4 really pays off. The negation is the all-zero string incremented by 1, yielding +1. Remembering the original sign is negative, \( 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111_{\text{two}} = -1_{\text{ten}} \)

4.6 Negating the two’s complement representation gives

\[ 1000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0001 \]

which equals

\[ (1 \times -2^{31}) + (1 \times 2^{0}) = -2,147,483,648_{\text{ten}} + 1_{\text{ten}} \]

\[ = -2,147,483,647_{\text{ten}} \]

Recalling that the original two’s complement number is positive,

\( 0111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111_{\text{two}} = 2,147,483,647_{\text{ten}} \).
4.7 By lookup using the table in Figure 4.1, page 218,
7fff ffa_{hex} = 0111 1111 1111 1111 1111 1111 1111 1010_{two}
and by the same technique used to solve Exercise 4.6
= 2,147,483,642_{ten}.

4.8 By lookup using the table in Figure 4.1, page 218,
1100 1010 1111 1110 1111 1010 1100 1110_{two} = cafe_{hex}.

4.9 Since MIPS includes add immediate and since immediates can be positive or negative, subtract immediate would be redundant.

4.10
```
addu $t2, $zero, $t3       # copy $t3 into $t2
bgez $t3, next            # if $t3 >= 0 then done
sub $t2, $zero, $t3       # negate $t3 and place into $t2
```
```
next
```

4.11 You should be quite suspicious of both claims. A simple examination yields

\[
\begin{align*}
6 &= 2 + 4 \\
12 &= 4 + 8 \\
18 &= 2 + 16 \\
24 &= 8 + 16 \\
30 &= 2 + 4 + 8 + 16 \text{ (so we know Harry is wrong)} \\
36 &= 4 + 32 \\
42 &= 2 + 8 + 32 \text{ (so we know David is wrong).}
\end{align*}
\]

4.12 The code loads the \texttt{sll} instruction at \texttt{shifter} into a register and masks off the shift amount, placing the least significant 5 bits from $s2$ in its place. It then writes the instruction back to memory and proceeds to execute it. The code is self-modifying; thus it is very hard to debug and likely to be forbidden in many modern operating systems. One key problem is that we would be writing into the instruction cache, and clearly this slows things down and would require a fair amount of work to implement correctly (see Chapters 6 and 7).

4.13 The problem is that \texttt{A\_lower} will be sign-extended and then added to \texttt{$t0$}. The solution is to adjust \texttt{A\_upper} by adding 1 to it if the most significant bit of \texttt{A\_lower} is a 1. As an example, consider 6-bit two's complement and the address 23 = 010111. If we split it up, we notice that \texttt{A\_lower} is 111 and will be sign-extended to 111111 = \texttt{-1} during the arithmetic calculation. \texttt{A\_upper\_adjusted} = 011000 = 24 (we added 1 to \texttt{010} and the lower bits are all \texttt{0}s). The calculation is then 24 + \texttt{-1} = 23.

4.14

a. The sign bit is 1, so this is a negative number. We first take its two's complement.

\[
\begin{align*}
A &= 1000 1111 1110 1111 1100 0000 0000 0000 \\
\neg A &= 0111 0000 0001 0000 0100 0000 0000 0000 \\
&= 2^{30} + 2^{29} + 2^{28} + 2^{20} + 2^{14} \\
&= 1,073,741,824 + 536,870,912 + 268,435,456 + 1,048,576 + 16,384 \\
&= 1,880,113,152 \\
A &= -1,880,113,152
\end{align*}
\]
b. 

\[ A = 1000\ 1111\ 1110\ 1110\ 1100\ 0000\ 0000\ 0000 \]
\[ = 8FEFC000 \]
\[ = 8 \times 16^7 + 15 \times 16^6 + 14 + 16^5 + 15 \times 16^4 + 12 \times 16^3 \]
\[ = 2,147,483,648 + 251,658,240 + 14,680,064 + 983,040 + 49,152 \]
\[ = 2,414,854,144 \]

c. 

\[ s = 1 \]
exponent = 0001 1111
\[ = 2^5 - 1 = 31 \]
significand = 110 1111 1100 0000 0000 0000
\[ (-1)^s \times (1 + \text{significand}) \times 2^{\text{exponent}-127} = -1 \times 1.111111111 \times 2^{-96} \]
\[ = -1 \times (1 + 13 \times 16^{-1} + 15 \times 16^{-2} + 2^{-9}) \times 2^{-96} \]
\[ = -1.873 \times 2^{-96} \]
\[ = -2.364 \times 10^{-29} \]

d. 

opcode (6 bits) = 100011 = lw
rs (5 bits) = 11111 = 31
rt (5 bits) = 01111 = 15
address (16 bits) = 1100 0000 0000 0000

Since the address is negative we have to take its two’s complement.

Two’s complement of address = 0100 0000 0000 0000
address = \(-2^{14}\)
\[ = -2^{14} \]
\[ = -16384 \]

Therefore the instruction is \( lw\ 15,\ -16384(31) \).

Notice that the address embedded within the 16-bit immediate field is a byte address unlike the constants embedded in PC-relative branch instructions where word addressing is used.

4.15

a. 0
b. 0
c. 0.0
d. \( sll\ \$0,\$0,0 \)

4.16 Figure 4.54 shows 21% of the instructions as being \( lw \). If 15% of these could take advantage of the new variant, that would be 3.2% of all instructions. Each of these presumably now has an \( addu \) instruction (that adds the two registers values together) that could be eliminated. Thus roughly 3.2% of the instructions, namely those \( addu \) instruc-
tions, could be eliminated if the addition were now done as part of the \texttt{lw}. The savings may be a bit overestimated (slightly less than 3.2\%) due to the fact that some existing instructions may be sharing \texttt{addu} instructions. Thus we might not eliminate one \texttt{addu} for every changed \texttt{lw}.

4.17 Either the instruction sequence

\begin{verbatim}
addu $t2, $t3, $t4
sltu $t2, $t2, $t4
\end{verbatim}

or

\begin{verbatim}
addu $t2, $t3, $t4
sltu $t2, $t2, $t3
\end{verbatim}

work.

4.18 If overflow detection is not required, then

\begin{verbatim}
addu $t3, $t5, $t7
sltu $t2, $t3, $t5
addu $t2, $t2, $t4
addu $t2, $t2, $t6
\end{verbatim}

is sufficient. If overflow detection is desired, then use

\begin{verbatim}
addu $t3, $t5, $t7
sltu $t2, $t3, $t5
add $t2, $t2, $t4
add $t2, $t2, $t6
\end{verbatim}

If overflow detection is desired, the last two \texttt{addu} instructions should be replaced by \texttt{add} instructions.

4.19 To detect whether \texttt{$s0 < s1$}, it’s tempting to subtract them and look at the sign of the result. This idea is problematic, because if the subtraction results in an overflow an exception would occur! To overcome this, there are two possible methods: You can subtract them as unsigned numbers (which never produces an exception) and then check to see whether overflow would have occurred (this is discussed in an elaboration on page 223). This method is acceptable, but it is lengthy and does more work than necessary. An alternative would be to check signs. Overflow can occur if \texttt{$s0$} and (\texttt{-$s1$}) share the same sign; i.e., if \texttt{$s0$} and \texttt{$s1$} differ in sign. But in that case, we don’t need to subtract them since the negative one is obviously the smaller! The solution in pseudocode would be

\begin{verbatim}
if ($s0<0$) and ($s1>0$) then
  $t0:=1$
else if ($s0>0$) and ($s1<0$) then
  $t0:=0$
else
  $t1:=$s0–$s1 \quad \# \text{ overflow can never occur here}
  if ($t1<0$) then
    $t0:=1$
  else
    $t0:=0$
\end{verbatim}

4.20 The new instruction treats \texttt{$s0$} and \texttt{$s1$} as a register pair and performs a shift, wherein the least significant bit of \texttt{$s0$} becomes the most significant bit of \texttt{$s1$} and both \texttt{$s0$} and \texttt{$s1$} are shifted right one place.
4.21

\[
\text{sll } s1, s0, 2 \\
\text{addu } s1, s0, s1
\]

4.22 No solution provided.

4.23 The ALU-supported set less than (\text{slt}) uses just the sign bit. In this case, if we try a set less than operation using the values \(-7\text{ ten} and 6\text{ ten} we would get \(-7 > 6\). This is clearly wrong. Modify the 32-bit ALU in Figure 4.11 on page 169 to handle \text{slt} correctly by factoring in overflow in the decision.

If there is no overflow, the calculation is done properly in Figure 4.17 and we simply use the sign bit (Result31). If there is overflow, however, then the sign bit is wrong and we need the inverse of the sign bit.

<table>
<thead>
<tr>
<th>Overflow</th>
<th>Result31</th>
<th>LessThan</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

\[\text{LessThan} = \text{Overflow} \oplus \text{Result31}\]

4.24 No solution provided.

4.25

\[10\text{ ten} = 1010\text{two} \]
\[= 1.01\text{two} \cdot 2^3\]

Sign = 0 Significand = .01
Single exponent = \(3 + 127 = 130\)
Double exponent = \(3 + 1023 = 1026\)
4.26

\[ 10.5_{\text{ten}} = 1010.1_{\text{two}} = 1.0101_{\text{two}} \cdot 2^3 \]

Sign = 0 Significand = .0101
Single exponent = 3 + 127 = 130
Double exponent = 3 + 1023 = 1026

The solution is the same as that for Exercise 4.25, but with the fourth bit from the left of the significand changed to a 1.

4.27

\[ 0.1_{\text{ten}} = 0.00011_{\text{two}} = 1.10011_{\text{two}} \cdot 2^{-4} \]

Sign = 0 Significand = .10011
Single exponent = -4 + 127 = 123
Double exponent = -4 + 1023 = 1019

4.28

\[ -2/3 = -1.01 \cdot 2^{-1} \]

Single exponent = -1 + 127 = 126
Double exponent = -1 + 1023 = 1022

4.29

```c
main( )
{
    float x;
    printf("> ");
    scanf("%f", &x);
    printf("%08lx\n", * (long *) &x);
}
```
4.30

```c
#include
// assumes float and long are both 32-bit numbers
main( )
{
    float x;
    cout << "> ";  // prompt
    cin >> x;
    cout << hex << (long &) x << endl;  // cast by reference (don't convert)
}
```

4.31 The IEEE number can be loaded easily into a register, \$t0, as a word since it occupies 4 bytes. To multiply it by 2, we recall that it is made up of three fields: a sign, an exponent, and a fraction. The actual stored number is the fraction (with an implied 1) multiplied by a power of two. Hence to multiply the whole number by 2, we simply add 1 to the exponent! Note that the following variations are incorrect:

- Multiply the register \$t0 by 2 (directly or thru \texttt{sll}) — this is wrong because \$t0 does not store an integer.
- Multiply the fractional part by 2. This is correct in principle but the resulting number would need to be renormalized into an IEEE format because the multiplication would shift the binary point. This would lead to adding 1 to the exponent!
- Shift the exponent left—this is wrong because it amounts to squaring the number rather than doubling it.

To add 1 to the exponent (which occupies bit positions 23 through 30) in \$t0, there are two ways (both ignore possible overflow):

1. Add \$t0 to a \$a0 where \$a0 contains a number having zero bits everywhere except at bit position 23. If a carry were to occur, this addition will change the sign bit, but we’re ignoring floating-point overflow anyway. Here’s the full sequence:
   ```
   lw      $t0, X($0)
   addi    $a0, $0, 1
   sll     $a0, $a0, 23
   addu    $t0, $t0, $a0
   sw      $t0, X($0)
   ```

2. Isolate the exponent (possibly by shifting left once then shifting 24 times to the right), add 1 to it, then shift it left 23 places to its proper place. To insert it back in its place, we start by “cleaning out” the old exponent (by ANDing with an appropriate mask) and then we OR the cleaned \$t0 with the incremented exponent. This method, albeit lengthy, is 100% acceptable. Here is a possible sequence:
   ```
   lw      $t0, X($0)
   andi    $t1, $t0, 0x7f800000
   srl     $t1, $t1, 23
   addi    $t1, $t1, 1
   sll     $t1, $t1, 23
   and     $t0, $t0, 0x807fffff
   or      $t0, $t0, $t1
   sw      $t0, X($0)
   ```
4.32

<table>
<thead>
<tr>
<th>Rank</th>
<th>Instruction</th>
<th>Percent executed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>lw</td>
<td>21%</td>
</tr>
<tr>
<td>2</td>
<td>addiu</td>
<td>17%</td>
</tr>
<tr>
<td>3</td>
<td>sw</td>
<td>12%</td>
</tr>
<tr>
<td>4</td>
<td>addu</td>
<td>9%</td>
</tr>
<tr>
<td>4</td>
<td>beq</td>
<td>9%</td>
</tr>
<tr>
<td>6</td>
<td>bne</td>
<td>8%</td>
</tr>
<tr>
<td>7</td>
<td>sll</td>
<td>5%</td>
</tr>
<tr>
<td>8</td>
<td>slt</td>
<td>2%</td>
</tr>
<tr>
<td>8</td>
<td>andi</td>
<td>2%</td>
</tr>
<tr>
<td>8</td>
<td>lui</td>
<td>2%</td>
</tr>
<tr>
<td>8</td>
<td>sra</td>
<td>2%</td>
</tr>
</tbody>
</table>

4.33

<table>
<thead>
<tr>
<th>Rank</th>
<th>Instruction</th>
<th>Percent executed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>l.s</td>
<td>24%</td>
</tr>
<tr>
<td>2</td>
<td>addu</td>
<td>10%</td>
</tr>
<tr>
<td>3</td>
<td>s.s</td>
<td>9%</td>
</tr>
<tr>
<td>4</td>
<td>lw</td>
<td>7%</td>
</tr>
<tr>
<td>5</td>
<td>lui</td>
<td>6%</td>
</tr>
<tr>
<td>6</td>
<td>mul.d</td>
<td>5%</td>
</tr>
<tr>
<td>6</td>
<td>sll</td>
<td>5%</td>
</tr>
<tr>
<td>8</td>
<td>add.d</td>
<td>4%</td>
</tr>
<tr>
<td>9</td>
<td>beq</td>
<td>3%</td>
</tr>
<tr>
<td>9</td>
<td>sub.d</td>
<td>3%</td>
</tr>
</tbody>
</table>

4.34

a. lw
   addu
   beq
   sll
   lui

b. List in Exercise 4.34a = 46% of gcc instructions executed.

c. List in Exercise 4.32 = 89% of gcc instructions executed.

d. List in Exercise 4.34a = 31% of spice instructions executed.

e. List in Exercise 4.33 = 76% of spice instructions executed.

4.35 lw
   addiu
   sw
   addu
   beq

They are the five most popular for gcc, and they represent 53% of the instructions executed for spice. The other popular spice instructions are never used in gcc, so they are used by only some programs and not others.
4.36

<table>
<thead>
<tr>
<th>Instruction category</th>
<th>Average CPI</th>
<th>Frequency (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loads and stores</td>
<td>1.4</td>
<td>21 + 12 = 33</td>
</tr>
<tr>
<td>Conditional branch</td>
<td>1.8</td>
<td>9 + 8 + 1 + 1 = 19</td>
</tr>
<tr>
<td>Jumps</td>
<td>1.2</td>
<td>1 + 1 = 2</td>
</tr>
<tr>
<td>Integer multiply</td>
<td>10.0</td>
<td>0</td>
</tr>
<tr>
<td>Integer divide</td>
<td>30.0</td>
<td>0</td>
</tr>
<tr>
<td>FP add and subtract</td>
<td>2.0</td>
<td>0</td>
</tr>
<tr>
<td>FP multiply, single precision</td>
<td>4.0</td>
<td>0</td>
</tr>
<tr>
<td>FP multiply, double precision</td>
<td>5.0</td>
<td>0</td>
</tr>
<tr>
<td>FP divide, single precision</td>
<td>12.0</td>
<td>0</td>
</tr>
<tr>
<td>FP divide, double precision</td>
<td>19.0</td>
<td>0</td>
</tr>
<tr>
<td>Integer ALU other than multiply and divide</td>
<td>1.0</td>
<td>9 + 17 + 1 + 2 + 5 + 2 + 1 + 1 + 2 + 1 + 1 + 2 + 1 = 46</td>
</tr>
</tbody>
</table>

Therefore,

\[
\text{CPI} = 1.4 \times 33\% + 1.8 \times 19\% + 1.2 \times 2\% + 1.0 \times 46\%
\]

\[
= 1.3
\]

4.37

<table>
<thead>
<tr>
<th>Instruction category</th>
<th>Average CPI</th>
<th>Frequency (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loads and stores</td>
<td>1.4</td>
<td>7 + 2 + 24 + 9 = 42</td>
</tr>
<tr>
<td>Conditional branch</td>
<td>1.8</td>
<td>3 + 2 + 1 + 1 + 1 = 8</td>
</tr>
<tr>
<td>Jumps</td>
<td>1.2</td>
<td>1 + 1 = 2</td>
</tr>
<tr>
<td>Integer multiply</td>
<td>10.0</td>
<td>0</td>
</tr>
<tr>
<td>Integer divide</td>
<td>30.0</td>
<td>0</td>
</tr>
<tr>
<td>FP add and subtract</td>
<td>2.0</td>
<td>4 + 3 = 7</td>
</tr>
<tr>
<td>FP multiply, single precision</td>
<td>4.0</td>
<td>0</td>
</tr>
<tr>
<td>FP multiply, double precision</td>
<td>5.0</td>
<td>5</td>
</tr>
<tr>
<td>FP divide, single precision</td>
<td>12.0</td>
<td>0</td>
</tr>
<tr>
<td>FP divide, double precision</td>
<td>19.0</td>
<td>2</td>
</tr>
<tr>
<td>Integer ALU other than multiply and divide</td>
<td>1.0</td>
<td>10 + 1 + 1 + 5 + 1 + 6 + 1 + 2 + 2 + 1 = 31</td>
</tr>
</tbody>
</table>

Therefore,

\[
\text{CPI} = 1.4 \times 42\% + 1.8 \times 8\% + 1.2 \times 2\% + 2.0 \times 7\% + 5.0 \times 5\% + 19.0 \times 2\% + 1.0 \times 31\%
\]

\[
= 1.8
\]

Note that the percentages given in Figure 4.54 for spice add up to 97%.

4.38 No solution provided.

4.39 No solution provided.

4.40

\[
xor \ s0, \ s0, \ s1
\]
\[
xor \ s1, \ s0, \ s1
\]
\[
xor \ s0, \ s0, \ s1
\]
4.41

```assembly
nor $s0, $zero, $zero
oxor $s0, $s0, $s1
```

4.42 Given that a number that is greater than or equal to zero is termed positive and a number that is less than zero is negative, inspection reveals that the last two rows of Figure 4.44 restate the information of the first two rows. Because \( A - B = A + (-B) \), the operation \( A - B \) when \( A \) is positive and \( B \) negative is the same as the operation \( A + B \) when \( A \) is positive and \( B \) is positive. Thus the third row restates the conditions of the first. The second and fourth rows refer also to the same condition.

Because subtraction of two’s complement numbers is performed by addition, a complete examination of overflow conditions for addition suffices to show also when overflow will occur for subtraction. Begin with the first two rows of Figure 4.44 and add rows for \( A \) and \( B \) with opposite signs. Build a table that shows all possible combinations of Sign and CarryIn to the sign bit position and derive the CarryOut, Overflow, and related information. Thus,

<table>
<thead>
<tr>
<th>Sign A</th>
<th>Sign B</th>
<th>Carry In</th>
<th>Carry Out</th>
<th>Sign of result</th>
<th>Correct sign of result</th>
<th>Overflow?</th>
<th>Carry In XOR Carry Out</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>No</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Yes</td>
<td>1</td>
<td>Carries differ</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>No</td>
<td>0</td>
<td>(</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>No</td>
<td>0</td>
<td>(</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>No</td>
<td>0</td>
<td>(</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>No</td>
<td>0</td>
<td>(</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Yes</td>
<td>1</td>
<td>Carries differ</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>No</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

From this table an Exclusive OR (XOR) of the CarryIn and CarryOut of the sign bit serves to detect overflow. When the signs of \( A \) and \( B \) differ, the value of the CarryIn is determined by the relative magnitudes of \( A \) and \( B \), as listed in the Notes column.
4.43 Here is the equation:

\[
\text{Sum} = (a \cdot b \cdot \text{CarryIn}) + (\overline{a} \cdot b \cdot \overline{\text{CarryIn}}) + (\overline{a} \cdot b \cdot \overline{\text{CarryIn}}) + (a \cdot b \cdot \text{CarryIn})
\]

4.44 \(C_1 = c_4, C_2 = c_8, C_3 = c_{12}, \) and \(C_4 = c_{16}.\)

\(c_4 = G_{3,0} + (P_{3,0} \cdot c_0).\)

\(c_8\) is given in the exercise.

\(c_{12} = G_{11,8} + (P_{11,8} \cdot G_{7,4}) + (P_{11,8} \cdot P_{7,4} \cdot G_{3,0}) + (P_{11,8} \cdot P_{7,4} \cdot P_{3,0} \cdot c_0).\)

\(c_{16} = G_{15,12} + (P_{15,12} \cdot G_{11,8}) + (P_{15,12} \cdot P_{11,8} \cdot G_{7,4})
\quad + (P_{15,12} \cdot P_{11,8} \cdot P_{7,4} \cdot G_{3,0}) + (P_{15,12} \cdot P_{11,8} \cdot P_{7,4} \cdot P_{3,0} \cdot c_0).\)

4.45 The equations for \(c_4, c_8,\) and \(c_{12}\) are the same as those given in the solution to Exercise 4.44. Using 16-bit adders means using another level of carry-lookahead logic to construct the 64-bit adder. The second level generate, \(G_0',\) and propagate, \(P_0',\) are

\(G_0' = G_{15,0} = G_{15,12} + P_{15,12} \cdot G_{11,8} + P_{15,12} \cdot P_{11,8} \cdot G_{7,4} + P_{15,12} \cdot P_{11,8} \cdot P_{7,4} \cdot G_{3,0}\)

and

\(P_0' = P_{15,0} = P_{15,12} \cdot P_{11,8} \cdot P_{7,4} \cdot P_{3,0}\)

Using \(G_0'\) and \(P_0'\) we can write \(c_{16}\) more compactly as

\(c_{16} = G_{15,0} + P_{15,0} \cdot c_0\)

and

\(c_{32} = G_{31,16} + P_{31,16} \cdot c_{16}\)
\(c_{48} = G_{47,32} + P_{47,32} \cdot c_{32}\)
\(c_{64} = G_{63,48} + P_{63,48} \cdot c_{48}\)

A 64-bit adder diagram in the style of Figure 4.24 is shown on the following page.
4.46 No solution provided.
4.47 No solution provided.
4.48 No solution provided.
4.49 No solution provided.
4.50 The longest paths through the top (ripple carry) adder organization in Figure 4.56 all start at input $a_0$ or $b_0$ and pass through seven full adders on the way to output $s_4$ or $s_5$. There are many such paths, all with a time delay of $7 \times 2T = 14T$. The longest paths through the bottom (carry save) adder all start at input $b_0$, $e_0$, $f_0$, $b_1$, $e_1$, or $f_1$ and proceed through six full adders to outputs $s_4$ or $s_5$. The time delay for this circuit is only $6 \times 2T = 12T$.
4.51 No solution provided.
4.52 No solution provided.
4.53

<table>
<thead>
<tr>
<th>Current bits</th>
<th>Prev. bits</th>
<th>Operation</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>ai + 1</td>
<td>ai</td>
<td>ai – 1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Add the multiplicand</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Add the multiplicand</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Add twice the multiplicand</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Subtract twice the multiplicand</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Subtract the multiplicand</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Subtract the multiplicand</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>None</td>
</tr>
</tbody>
</table>

One example of 6-bit operands that run faster when Booth’s algorithm looks at 3 bits at a time is 21\text{ten} \times 27\text{ten} = 567\text{ten}.

Two-bit Booth’s algorithm:

\[
\begin{align*}
010101 & \quad = 21\text{ten} \\
\times 011011 & \quad = 27\text{ten} \\
- 010101 & \quad 10 \text{ string (always start with padding 0 to right of LSB)} \\
000000 & \quad 11 \text{ string, middle of a string of 1s, no operation} \\
+ 010101 & \quad 01 \text{ string, add multiplicand} \\
- 010101 & \quad 10 \text{ string, subtract multiplicand} \\
000000 & \quad 11 \text{ string} \\
+ 010101 & \quad 01 \text{ string} \\
11111010111 & \quad \text{two’s complement with sign extension as needed} \\
0000000000 & \quad \text{zero with sign extension shown} \\
000010101 & \quad \text{positive multiplicand with sign extension} \\
1110101 & \quad 11 \text{ string} \\
000000 & \quad \text{positive multiplicand with sign extension} \\
+ 010101 & \quad 01 \text{ string} \\
01000110111 & \quad = 567\text{ten}
\end{align*}
\]

Don’t worry about the carry out of the MSB here; with additional sign extension for the addends, the sum would correctly have an extended positive sign. Now, using the 3-bit Booth’s algorithm:

\[
\begin{align*}
010101 & \quad = 21\text{ten} \\
\times 011011 & \quad = 27\text{ten} \\
- 010101 & \quad 110 \text{ string (always start with padding 0 to right of LSB)} \\
- 010101 & \quad 101 \text{ string, subtract the multiplicand} \\
+ 01010101 & \quad 011 \text{ string, add twice the multiplicand (i.e, shifted left 1 place)} \\
111111010111 & \quad \text{two’s complement of multiplicand with sign extension} \\
1111010111 & \quad \text{two’s complement of multiplicand with sign extension} \\
+ 01010101 & \quad \text{two’s complement of multiplicand with sign extension} \\
010001101111 & \quad = 567\text{ten}
\end{align*}
\]
Using the 3-bit version gives only 3 addends to sum to get the product versus 6 addends using the 2-bit algorithm.

Booth’s algorithm can be extended to look at any number of bits b at a time. The amounts to add or subtract include all multiples of the multiplicand from 0 to $2^{(b-1)}$. Thus, for $b > 3$ this means adding or subtracting values that are other than powers of two multiples of the multiplicand. These values do not have a trivial “shift left by the power of 2 number of bit positions” method of computation.

4.54

1. Subtract the Divisor register from the Remainder register, and place the result in the left half of the remainder register

Remainder $\geq 0$

Test Remainder

Remainder $< 0$

2a. Shift the Remainder register to the left and set the new rightmost bit to 1

2b. Shift the Remainder register to the left and set new rightmost bit to 0

32nd repetition?

Yes: 32 repetitions

No: < 32 repetitions

3a. Subtract Divisor to left half of the Remainder

32nd repetition?

Yes: 32 repetitions

No: < 32 repetitions

3b. Add Divisor to left half of the Remainder

Done: Shift left half of Remainder right 1 bit

Subtract Divisor

Done: Shift left half of Remainder right 1 bit
Note that the shaded boxes are necessary when \( \text{Remainder} < 0 \) to correct the redundant addition in step 3b should the last step end up in this branch of the flow chart. As shown by the table, 7 divided by 2 is 3 with a remainder of 1.

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Step</th>
<th>Divisor</th>
<th>Remainder</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Initial Values</td>
<td>0010</td>
<td>0000 1110</td>
</tr>
<tr>
<td>1</td>
<td>1: ( \text{Rem} - \text{Rem} - \text{Div} )</td>
<td>0010</td>
<td>1110 1110</td>
</tr>
<tr>
<td></td>
<td>2b: ( \text{Rem} &lt; 0 \Rightarrow \text{sll R, R0 = 0} )</td>
<td>0010</td>
<td>1101 1100</td>
</tr>
<tr>
<td></td>
<td>3b: ( \text{Rem} = \text{Rem} + \text{Div} )</td>
<td>0010</td>
<td>1111 1100</td>
</tr>
<tr>
<td>2</td>
<td>2b: ( \text{Rem} &lt; 0 \Rightarrow \text{sll R, R0 = 0} )</td>
<td>0010</td>
<td>1111 1000</td>
</tr>
<tr>
<td></td>
<td>3b: ( \text{Rem} = \text{Rem} + \text{Div} )</td>
<td>0010</td>
<td>0001 1000</td>
</tr>
<tr>
<td>3</td>
<td>2a: ( \text{Rem} &gt; 0 \Rightarrow \text{sll R, R0 = 1} )</td>
<td>0010</td>
<td>0011 0001</td>
</tr>
<tr>
<td></td>
<td>3a: ( \text{Rem} = \text{Rem} - \text{Div} )</td>
<td>0010</td>
<td>0001 0001</td>
</tr>
<tr>
<td>4</td>
<td>2a: ( \text{Rem} &gt; 0 \Rightarrow \text{sll R, R0 = 1} )</td>
<td>0010</td>
<td>0010 0011</td>
</tr>
<tr>
<td>Done</td>
<td>Shift Rem right 1</td>
<td>0010</td>
<td>0001 0011</td>
</tr>
</tbody>
</table>

4.55 Various rounding modes are possible. Round To Nearest is the rounding mode that is most familiar, with the exception of handling one special case. In the event that the number to round is exactly halfway between two values with the desired number of digits, the Rounding to Nearest tie is broken by rounding to the value with an even least significant digit. This removes any consistent bias in the direction of rounding—half the ties are broken by rounding up and half are broken by rounding down. Other modes of rounding in IEEE 754 are Round Toward Positive Infinity, Round Toward Negative Infinity, and Round Toward Zero (commonly known as Truncation). When no digits to the right of the least significant digit are kept in the arithmetic circuitry, then right-shifting the significand to align the exponents results in rounding via Truncation by default.

With guard and round digits (assume Round to Nearest mode is chosen):

- Extend with guard and round digits \( 9.5100 \times 10^2 \)
- Shift smaller number to match larger exponent \(+ 0.6420 \times 10^2\)
- Add (circuit must hold extra high-order digit) \(10.1520 \times 10^2\)
- Normalize the sum \(1.0152 \times 10^2\)
- Round to three significant digits \(1.02 \times 10^2\)

Without guard and round digits (Truncation occurs):

- Shift smaller number, truncate, keep only 3 digits \(9.51 \times 10^2\)
- Add (circuit must hold extra high-order digit) \(+ 0.64 \times 10^2\)
- Normalize the sum (truncation occurs) \(10.15 \times 10^2\)

Truncation saves space—no guard and rounding digits are needed—and it saves time—the shifting by normalization yields the rounded result. The disadvantage is the quality of the math, as shown by comparison with the more accurate result obtained using the more space (circuitry) and time-intensive Round to Nearest mode.
4.56 See solution to Exercise 4.55 for an explanation of rounding modes. With guard and round digits (assume Round to Nearest mode is chosen):

<table>
<thead>
<tr>
<th>Operation</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extend with guard and round digits</td>
<td>$1.4700 \times 10^2$</td>
</tr>
<tr>
<td>Shift smaller number to match larger exponent</td>
<td>$+ 0.8760 \times 10^2$</td>
</tr>
<tr>
<td>Add</td>
<td>$2.3460 \times 10^2$</td>
</tr>
<tr>
<td>Normalize sum (no shift needed, in normal form)</td>
<td>$2.3460 \times 10^2$</td>
</tr>
<tr>
<td>Round to three significant digits</td>
<td>$2.35 \times 10^2$</td>
</tr>
</tbody>
</table>

Without guard and round digits (Truncation occurs):

<table>
<thead>
<tr>
<th>Operation</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shift smaller number, truncate, keep only 3 digits</td>
<td>$1.47 \times 10^2$</td>
</tr>
<tr>
<td>Add</td>
<td>$2.34 \times 10^2$</td>
</tr>
<tr>
<td>Normalize sum (no shift needed, in normal form)</td>
<td>$2.34 \times 10^2$</td>
</tr>
</tbody>
</table>

4.57 Step 1. Unlike multiplication, we calculate the exponent of the product by just subtracting the exponents of the operands.

New exponent = $10 - (-5) = 15$

Step 2. Next comes the division of the significands:

```
   1001
  1.100 | 1.1100000
  -1.100
    100
    1000
   10000
  - 1100
    100
    Remainder
```

1.001 with a remainder of 0.100/1.100. Hence the quotient is $1.001 \times 10^{15}$. Of course we must check for division by zero.

Step 3. This product is normalized, so we need do nothing more.

Step 4. Check to see if the exponents overflow or underflow.

Step 5. The sign of the quotient depends on the signs of the original operands. If they are both the same, the sign is positive, otherwise it’s negative. Hence the quotient is $+1.001 \times 10^{15}$.
1. Subtract the exponent of the divisor from the dividend

2. Divide the significands. Check for divide by zero

3. Normalize the product if necessary, shifting it right and incrementing the exponent

   Overflow or underflow?

   Yes

   Interrupt

   No

4. Round significand to the appropriate number of bits

   Still normalized?

   No

5. Set the sign of the product to positive if the signs of the original operands are the same. If they differ make the sign negative

   Yes

Done
4.58 The guard bit is present to preserve the full precision (number of significant bits) of the significand during the operation. It can be shown that a single bit is sufficient for +, −, ×, and / to preserve the precision of a result that can be expressed in the number of bits available in the significand. If the result has more bits of precision than available in the floating-point format, then rounding is used to yield an expressible approximation (fits in the available number of bits) of the result.

The bits guard (g), round (r), and sticky (s) all participate in right shifts of the significand to make the exponent of the smaller number match that of the larger number. Right shifting moves the LSB into g, g into r, and the OR of bit(s) shifted through r into s. During left shifts, s does not participate, zeros are shifted into r as needed, r shifts into g, and g shifts into the LSB.

The following tables list the actions needed for each rounding mode.

Round Toward Negative Infinity:

<table>
<thead>
<tr>
<th>g</th>
<th>r</th>
<th>s</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Exact result; significand correctly rounded</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>1</td>
<td>Inexact result, correctly rounded if Sum is positive, round by adding 1 to LSB if Sum is negative (makes Sum more negative, moving toward negative infinity)</td>
</tr>
<tr>
<td>X</td>
<td>1</td>
<td>X</td>
<td>Inexact result, correctly rounded if Sum is positive, round by adding 1 to LSB if Sum is negative (makes Sum more negative, moving toward negative infinity)</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
<td>Inexact result, correctly rounded if Sum is positive, round by adding 1 to LSB if Sum is negative (makes Sum more negative, moving toward negative infinity)</td>
</tr>
</tbody>
</table>

Round Toward Positive Infinity:

<table>
<thead>
<tr>
<th>g</th>
<th>r</th>
<th>s</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Exact result; significand correctly rounded</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>1</td>
<td>Inexact result, correctly rounded if Sum is negative, round by adding 1 to LSB if Sum is positive (makes Sum more positive, moving toward positive infinity)</td>
</tr>
<tr>
<td>X</td>
<td>1</td>
<td>X</td>
<td>Inexact result, correctly rounded if Sum is negative, round by adding 1 to LSB if Sum is positive (makes Sum more positive, moving toward positive infinity)</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
<td>Inexact result, correctly rounded if Sum is negative, round by adding 1 to LSB if Sum is positive (makes Sum more positive, moving toward positive infinity)</td>
</tr>
</tbody>
</table>

Truncate:

<table>
<thead>
<tr>
<th>g</th>
<th>r</th>
<th>s</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>Significand correctly rounded</td>
</tr>
</tbody>
</table>
(Unbiased) Round to Nearest:

<table>
<thead>
<tr>
<th>$\text{Sum}_p$</th>
<th>g</th>
<th>r</th>
<th>s</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Exact result; significand correctly rounded</td>
</tr>
<tr>
<td>X</td>
<td>0</td>
<td>X</td>
<td>1</td>
<td>Inexact result, but significand is correctly rounded</td>
</tr>
<tr>
<td>X</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Inexact result, but significand is correctly rounded</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Tie case with even significand; significand correct</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Tie case with odd significand; round to nearest even by adding 1 to LSB</td>
</tr>
<tr>
<td>X</td>
<td>1</td>
<td>1</td>
<td>X</td>
<td>Inexact result, round to nearest by adding 1 to LSB</td>
</tr>
<tr>
<td>X</td>
<td>1</td>
<td>X</td>
<td>1</td>
<td>Inexact result, round to nearest by adding 1 to LSB</td>
</tr>
</tbody>
</table>

Note that unbiased round to nearest is the only mode that requires the value of the LSB. Neither unbiased round to nearest nor truncate predicate their action on the sign of the significand. Round to plus and minus infinity are the only modes that require the sign bit.

Filling in the table given in the exercise:

<table>
<thead>
<tr>
<th>Rounding mode</th>
<th>$\text{Sum} \geq 0$</th>
<th>$\text{Sum} &lt; 0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Toward Negative Infinity</td>
<td>blank</td>
<td>NAND($g,r,s$)</td>
</tr>
<tr>
<td>Toward Positive Infinity</td>
<td>NAND($g,r,s$)</td>
<td>blank</td>
</tr>
<tr>
<td>Truncate</td>
<td>blank</td>
<td>blank</td>
</tr>
<tr>
<td>(Unbiased) Round to Nearest</td>
<td>$g(\text{Sum}_p + r + s)$</td>
<td>$g(\text{Sum}_p + r + s)$</td>
</tr>
</tbody>
</table>

4.59 No solution provided.

4.60 No solution provided.
5.1 If Regdst = 0, all R-format instructions will not work properly because we will specify the wrong register to write. If ALUSrc = 0, then all I-format instructions except branch will not work because we will not be able to get the sign-extended 16 bits into the ALU. If MemtoReg = 0, then loads will not work. If Zero = 0, the branch instruction will never branch, even when it should.

5.2 If Regdst = 1, loads will not work (bad destination). If ALUSrc = 1, then all R-format instructions as well as branch will not work properly because the second register read will not get into the ALU. If MemtoReg = 1, then all R-format instructions will not write the correct data into the register file. If Zero = 1, all branches will be taken all of the time, even if they shouldn’t be.

5.3 If RegDst = 0, all R-format instructions will be unable to write into the proper register (rd). If MemtoReg = 0 or IorD = 0, then loads will not work. If ALUSrcA = 0, none of the instructions will work properly because they all require A to be operated on by the ALU at some point.

5.4 If RegDst = 1, load instructions will not work. If MemtoReg = 1, all R-format instructions will not work. If IorD = 1, no instructions will work because the PC will never be used to get an instruction. If ALUSrcA = 1, the program counter will not be properly incremented by 4 and essentially all instructions will not work.

5.5 No additions to the datapath are required. A new row should be added to the truth table in Figure 5.20. The new control is similar to load word because we want to use the ALU to add the immediate to a register (and thus RegDst = 0, ALUSrc = 1, ALUOp = 00). The new control is also similar to an R-format instruction, because we want to write the result of the ALU into a register (and thus MemtoReg = 0, RegWrite = 1) and of course we aren’t branching or using memory (Branch = 0, MemRead = 0, MemWrite = 0).

5.6 We already have a way to change the PC based on the specified address (using the datapath for the jump instruction), but we’ll need a way to put PC+4 into register $ra (31), and this will require changing the datapath. We can expand the multiplexor controlled by RegDst to include 31 as a new input. We can expand the multiplexor controlled by MemToReg to have PC+4 as an input. Because we expand these multiplexors, the entire columns for RegDst and MemtoReg must change appropriately in the truth table. The jal instruction doesn’t use the ALU, so ALUSrc and ALUOp can be don’t cares. We’ll have Jump = 1, RegWrite = 1, and we aren’t branching or using memory (Branch = 0, MemRead = 0, MemWrite = 0).

5.7 One possible solution is to add a new control signal called “Invzero” that selects whether Zero or inverted Zero is an input to the AND gate used for choosing what the new PC should be (thus a new multiplexor is introduced). The new control signal Invzero would be a don’t care whenever the control signal Branch is zero. Many other solutions are possible.
Note to instructors: Sometimes different solutions submitted by students will in fact be identical (e.g., one student draws a multiplexer, the other draws gates, but in fact they both implemented it the same way—just at different levels of abstraction). You may want to look for solutions that you can use to emphasize this point to your students. For example, an alternative solution would be to use Zero as an input to a new multiplexer that selects between Branch and a new control signal Bnequal. In this case, there would be no don’t cares because if we aren’t branching, both Branch and Bnequal need to be zero. Some students may present this exact solution in gate form instead of in mux form.

5.8 No changes are needed in the datapath. The new variant is just like \textit{lw} except that the ALU will use the Read data 2 input instead of the sign-extended immediate. Of course the instruction format will need to change: the register write will need to be specified by the \textit{rd} field instead of the \textit{rt} field. However, all needed paths are already in place for the sake of R-format instructions. To modify the control, we simply need to add a new row to the existing truth table. For the new instruction, RegDst = 1, ALUSrc = 0, MemtoReg = 1, RegWrite = 1, MemtoReg = 1, MemWrite = 0, ALUop = 00.

5.9 There are no temporary registers that can be made to hold intermediate (temporary) data. There are no intermediate edges to clock them. There is no state machine to keep track of whether \textit{rs} has been updated, or both \textit{rs} and \textit{rt} have been updated. In essence, we can’t order things.

5.10 Looking at Figure 5.20, we see that MemtoReg and MemRead are identical except for \textit{sw} and \textit{beq}, for which MemtoReg is a don’t care. Thus, the modification will work for the single-cycle datapath. The modification will also work on the multiple-cycle datapath assuming that the finite state machine is changed so that MemRead is asserted whenever MemtoReg is.

5.11 Using Figure 5.20, we discover that MemtoReg could be replaced by ALUSrc, RegDst could be replaced by ALUOp1, and either Branch or ALUOp0 could be replaced in favor of the other (their signals are identical). Note that in reality there would likely be additional rows present in the truth table to support other instructions, and it is quite likely that no control signals could be eliminated.

5.12 The key is recognizing that we no longer have to go through the ALU and then to memory. We would not want to add zero using the ALU, instead we want to provide a path directly from the Read data 1 output of the register file to the read/write address lines of the memory (assuming the instruction format does not change). The output of the ALU would no longer connect to memory. The control does not need to change, but some of the control signals now are don’t cares. Assuming we are not implementing \textit{addi} or \textit{addiu}, it is possible to remove AluSrc and the multiplexer it controls by having just the data from Read data 2 going into the ALU. This results in additional optimizations to ALU control.
5.13 The cycle time goes down, from 8 ns to 6 ns, because we no longer need to go through the ALU to calculate an address. Consider executing 10 instructions, one of which is a load word with an offset. Previously this would have taken \(10 \times 8 = 80\) ns. Now we require an additional \texttt{addi} instruction to compute the offset, and it takes \((10 + 1) \times 6 = 66\) ns. So clearly we can tolerate more than 10%. Algebraically, \(8 \times I = 6 \times (X + 1)I\), where \(I\) is the number of instructions and \(X\) is the percentage we are solving for. Thus, \(X = .33\) and we can tolerate 33% of the instructions being loads and stores with offsets. This shouldn’t be surprising, because we sped up the cycle time by a factor of \(8/6 = 1.33\).

5.14 The key is to understand that it is the length of the longest path in the combinational logic that is determining the cycle time. Essentially, we compute the length of the longest path for each instruction and then must take the one with maximum value. At present, the \texttt{lw} instruction is providing the longest path of length 8 ns.

a. The changes do not increase any paths beyond current maximums. Thus the cycle time is still 8 ns.

b. Consider the \texttt{beq} instruction. We now have a path which requires 10 ns to compute the branch address \((X + Y = 10)\) ns, and this is maximum. Thus, the cycle time increases to 10 ns.

c. You might be tempted to again conclude that the branch instruction will require \(X + Y = 9\) ns for branch address computation. This is not quite correct, however, because the second adder (requiring \(Y\) time) has two inputs, one of which is not available until after the instruction is read (the 16 immediate bits), and this takes 2 ns. Thus, the actual maximum length path is again 10 ns, and the cycle time is 10 ns.

5.15 The existing datapath is sufficient. The execution steps for \texttt{addi} would be

- Instruction fetch (unchanged)
- Instruction decode and register fetch (unchanged)
- Execution: \(ALUOut = A + \text{sign-extend (IR[15-0])};\)
- Instruction Completion: \(\text{Reg[IR[20-16]]} = ALUOut\)

The first three steps are identical to those performed for memory access instructions, so we can use them (we must add \(Op = \text{‘addi’}\) to the transition from state 1 to state 2). The fourth step is different and not like any existing step; thus we must add a new state to our finite state machine (i.e., state 10), a transition from state 2 to state 10 if \(Op = \text{‘addi’}\), and a transition from state 10 back to state 0. In state 10 we have \(\text{RegDst} = 0, \text{RegWrite}, \text{MemtoReg} = 0\).

5.16 We need to change Dispatch 1 and Dispatch 2 and introduce a new entry (perhaps inserted after SW2) with a label of “ADDI2” and a register control of “Write ALU rt,” which is similar to “Write ALU” except that it uses the rt field instead of rd. You could argue that further changes should be made to existing labels to make things easier to understand.
5.17 The existing datapath is insufficient. As described in the solution to 5.6 (above), we’ll need to expand the two multiplexors controlled by RegDst and MemtoReg. The execution steps would be:

- Instruction fetch (unchanged)
- Instruction decode and register fetch (unchanged)

\[ \text{jal: Reg}[31] = \text{PC}; \text{PC} = \text{PC}[31-28] || (\text{IR}[25-0] \ll 2) \]

Note that in this case we are writing the PC after it has already been incremented by 4 (in the Instruction fetch step) into register $ra$ (thus the datapath requires that PC be an input to the MemtoReg multiplexor and 31 needs to be an input to the RegDst multiplexor). We need to modify existing states to show proper values of RegDst and MemtoReg and add a new state that performs the jal instruction (and then returns to state 0) via PCWrite, PCSource=10, RegWrite, and appropriate values for MemtoReg and RegDst.

5.18 The solution is dependent on the instruction format chosen, and there are many implementation options. One approach for the format of the swap instruction might be to name in both the rs and rd fields one of the two registers being swapped. The other register would be named in rt. The outputs of A and B in Figure 5.33 should both be directed to the multiplexor controlled by MemtoReg. The MemtoReg control signal now becomes two bits (e.g., 00 = write ALUout, 01 = write MDR, 10 = write A, 11 = write B).

Because the existing multicycle control of the datapath causes A to be written on every clock cycle (see page 386, step 2), we should add a new state to those in Figure 5.42 for clock cycle 3 of the swap instruction that accomplishes Reg[IR[15-11]] = A, that is rt = rs, and add another new state for clock cycle 4 of the swap instruction that accomplishes Reg[IR[15-11]] = B, that is rd = rt. Because we wrote the swap instruction with rs and rd referring to the same register, the step in cycle 4 does rs = rt, completing the swap. In the new state for cycle 3 we need control signals set as RegDst = 1, RegWrite, MemtoReg = 11, and in the new state for cycle 4 we need RegDst = 0, RegWrite, MemtoReg = 10. In Figure 5.42, state 4 should be revised to have MemtoReg = 01 and state 7 should have MemtoReg = 00.

5.19 There are a variety of implementation options. In state 0 the PC is incremented by 4, so we must subtract 4 from the PC and put this value in a register. This requires two new states (10 and 11). There would be an arrow from state 1 to state 10 labeled Op = 'wait', and there should be unconditional arrows from state 10 to state 11 and from state 11 to state 0. State 10 accomplishes ALUout = PC - 4 (ALUSrcA=0, ALUSrcB=01, and ALUop=01) and state 11 accomplishes Reg[IR[20-16]] = ALUout (RegDst=0, MemtoReg=0, RegWrite). An alternative solution relies on the format assumption that a -1 is put in the 16-bit immediate value, then the calculation of the branch address actually computes the PC. An alternative solution relies on the idea that wait could be handled like a branch instruction. If a -1 occupied the 16-bit immediate value field of a wait instruction, then calculation of the branch target address actually computes the PC of the wait instruction itself. This clever solution reduces the number of needed new control states by one.

5.20 The jump memory instruction behaves much like a load word until the memory is read. The data coming out of memory needs to be deposited into the PC. This will require a new input to the multiplexor controlled by PCSource. We add a new state coming off of state 3 that checks for Op = 'jump memory' (and modify the transition to state 4 to ensure Op = 'lw'). The new state has PCWrite, PCSource = 11. The transitions to states 2 and 3 need to also include Op = 'jump memory'.
5.21 The instruction will require using the ALU twice. A multiplexer should be added to permit the ALU to use the previous result produced by the ALU (stored in ALUOut) as an input for the second addition. Note that changes will also be needed for register read. The instruction can be implemented in 5 cycles by reading the third register source during the cycle in which the first addition is taking place. Clearly a multiplexer and a new control signal will be needed for the inputs to one of the register read ports.

5.22 The rt field of the jump register instruction contains 0. So, along with rs we read $0. To get rs to the output of the ALU, we can perform a dummy addition of $rs + $0. We already have a path from ALUOut to the PC. We can use this path for loading the PC with rs. The finite state machine will need a new state for when the opcode is 000000 and the function code is 001000. The current condition for entering state 6 needs to be modified to make sure the function code is not 001000. In the new state, we want the appropriate combination of states 0 and 6: ALUSrcA = 1, ALUSrcB = 00, PCWrite, PC-Source = 00, and of course we make sure an addition is performed with ALUOp = 00.

5.23 Many solutions are possible. In all of them, a multiplexer will be needed as well as a new control signal (e.g., RegRead) to select which register is going to be read (i.e., using IR[25-11] or IR[20-16]). One simple solution is simply to add a write signal to A and break up state 1 into two states, in which A and B are read. It is possible to avoid adding the write signal to A if B is read first. Then A is read and RegRead is held stable (because A always writes). Alternatively, you could decide to read A first because it may be needed to calculate an address. You could then postpone reading B until state 2 and avoid adding an extra cycle for the load and store instructions. An extra cycle would be needed for the branch and R-type instructions.

5.24 The instruction mix is

<table>
<thead>
<tr>
<th>Instruction class</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loads</td>
<td>21%</td>
</tr>
<tr>
<td>Stores</td>
<td>12%</td>
</tr>
<tr>
<td>R-type</td>
<td>46%</td>
</tr>
<tr>
<td>Jump/branch</td>
<td>21%</td>
</tr>
</tbody>
</table>
The CPIs for each machine by instruction class are

<table>
<thead>
<tr>
<th>Instruction class</th>
<th>M1 CPIs</th>
<th>M2 CPIs</th>
<th>M3 CPIs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loads</td>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>Stores</td>
<td>4</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>R-type</td>
<td>4</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Jump/branch</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Effective CPI</td>
<td>4.00</td>
<td>3.33</td>
<td>3.0</td>
</tr>
</tbody>
</table>

\[
\text{MIPS}_{M1} = \frac{500}{4.00} = 125.0
\]

\[
\text{MIPS}_{M2} = \frac{400}{3.33} = 120.1
\]

\[
\text{MIPS}_{M3} = \frac{250}{3.0} = 83.3
\]

So the original machine M1 is fastest for this mix. M3 would be the fastest for an instruction mix that contained only loads.

5.25 The gcc data from Chapter 4 has the following instruction mix:

<table>
<thead>
<tr>
<th>Instruction class</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loads</td>
<td>21%</td>
</tr>
<tr>
<td>Stores</td>
<td>12%</td>
</tr>
<tr>
<td>R-type</td>
<td>46%</td>
</tr>
<tr>
<td>Jump/branch</td>
<td>21%</td>
</tr>
</tbody>
</table>

Here are the CPIs:

<table>
<thead>
<tr>
<th>Instruction class</th>
<th>CPI on 500-MHz machine</th>
<th>CPI on 750-MHz machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loads</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>Stores</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>R-type</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Jump/branch</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

\[
\text{CPI}_{500\text{MHz}} = 0.21 \times 5 + 0.12 \times 4 + 0.46 \times 4 + 0.21 \times 3 = 4.00
\]

\[
\text{CPI}_{750\text{MHz}} = 0.21 \times 6 + 0.12 \times 5 + 0.46 \times 4 + 0.21 \times 3 = 4.33
\]

The 750 MHz is faster by

\[
\frac{\text{CPI}_{750\text{MHz}}}{\text{CPI}_{500\text{MHz}}} = \frac{4.33}{4.00} = \frac{173.21}{125.00} = 1.39.
\]
5.26 Here is a possible code sequence:

```
beq $t3, $zero, done
move: lw $t4, 0($t1)
       sw $t4, 0($t2)
       addi $t1, $t1, 4
       addi $t2, $t2, 4
       addi $t3, $t3, -1
       bne $t3, $zero, move
done:    ...
```

The number of cycles required is determined by computing the number of instructions required. To copy 100 words we’ll perform 100 loads, 100 stores, 300 adds, and 101 branches. We then use the number of cycles for each instruction to arrive at 500 cycles for the loads, 400 cycles for the stores, 1200 cycles for the adds, and 303 cycles for the branches. Grand total is 2403 cycles.

5.27 No solution provided.
5.28 No solution provided.
5.29 No solution provided.
5.30 No solution provided.
5.31 No solution provided.
5.32 No solution provided.
5.33 No solution provided.
5.34 No solution provided.
5.35 No solution provided.
5.36 No solution provided.