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

Time for $\frac{1}{2}$ revolution = $\frac{1}{2}$ rev $\times \frac{1}{7200} \frac{\text{minutes}}{\text{rev}} \times 60 \frac{\text{seconds}}{\text{minute}} = 4.17$ 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 × .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

1980	Die area	0.16
	Yield	0.48
	Defect density	17.04
1992	Die area	0.97
	Yield	0.48
	Defect density	1.98
1992 + 1980	Improvement	8.62

1.54 No solution provided.

1.55 No solution provided.

1.56 No solution provided.

2 Solutions

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 200×10^6 cycles per second / 20×10^6 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 / second = 600×10^6 cycles per program / 10 cycles per instruction = 60×10^6 instructions per program. For M2, we have 4 seconds $\times 300 \times 10^6$ cycles / second = 1200×10^6 cycles per program / 9.4 cycles per instruction = 127.7×10^6 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 / Execution time) $\times$ (1 / 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.

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

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

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

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. $\text{CPI for M1} = \frac{1+2+3+4}{4} = 2.5$, and $\text{CPI for M2} = \frac{2+2+4+4}{4} = 3.0$. Using $\text{CPU time} = \frac{\text{Instruction count} \times \text{CPI}}{\text{Clock rate}}$, we get the following: $\text{CPU time for M1} = \frac{\text{Instruction count} \times 2.5}{500\text{ MHz}} = \frac{\text{Instruction count}}{200\text{ million}}$, and $\text{CPU time for M2} = \frac{\text{Instruction count} \times 3}{750\text{ MHz}} = \frac{\text{Instruction count}}{250\text{ million}}$.

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 × 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, C, MIPS} {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 $= \dfrac{\text{Clock rate}}{\text{CPI} \times 10^6}$. 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 $= \dfrac{1000}{\text{CPI}} = 278$ and

MIPS for MNFP $= \dfrac{1000}{\text{CPI}} = 500$.

2.16

Instruction class	Frequency on MFP	Count on MFP in millions	Count on MNFP in millions
Floating point multiply	10%	30	900
Floating point add	15%	45	900
Floating point divide	5%	15	750
Integer instructions	70%	210	210
Totals	100%	300	2760

2.17 Execution time $= \dfrac{\text{IC} \times 10^6}{\text{MIPS}}$. So execution time is $\dfrac{300}{278} = 1.08$ seconds, and execution time on MNFP is $\dfrac{2760}{500} = 5.52$ 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 must faster is Mcomp than Mbase?

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

$$\text{CPU time Mcomp} = \frac{\text{Clock rate}}{\text{IC} \times 0.81 \times 3.1} = \frac{\text{Clock rate}}{\text{IC} \times 2.5}$$

So then

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

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

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

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

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.

$\text{GM(A)} = \sqrt{1000} = 32$, $\text{GM(B)} = \sqrt{1000} = 32$, and $\text{GM(C)} = \sqrt{400} = 20$, 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 $10x + 100 = 1.6 \times (20x + 20)$, x = 3.1 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: $x + 1000 = 1.6 \times (20x + 20)$, x = 31.2. So the mix is 97% program 1 and 3% program 2. Note that the mix is very different in each case!

2.29

Program	Weight	Computer A	Computer B	Computer C
Program 1 (seconds)	10	1	10	20
Program 2 (seconds)	1	1000	100	20
Weighted AM		9.18	18.2	20

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:

Program	Weight	Computer A	Computer B	Computer C
Program 1 (seconds)	1	1	10	20
Program 2 (seconds)	1/1000	1000	100	20
Weighted AM		2	10.1	20

This makes A the fastest.

Now with equal time on machine B:

Program	Weight	Computer A	Computer B	Computer C
Program 1 (seconds)	1	1	10	20
Program 2 (seconds)	1/10	1000	100	20
Weighted AM		91.8	18.2	20

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

Program	Computer A	Computer B	Computer C
1	10	1	0.5
2	0.1	1	5

2.39 The harmonic mean of a set of rates,

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

where AM is the arithmetic mean of the corresponding execution times.

3 Solutions

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 + ... + `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 (ceiling(`n`/2))^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 × 13 for the outer loop. The inner loop requires 1 + 2 + 2 + 1 + 1 + 2 = 9 cycles per iteration and it repeats 5000 × 5000 times, for a total of 9 × 5000 × 500 cycles. The overall execution time is thus approximately (5000 × 13 + 9 × 5000 × 5000) / (500 × 10^6) = .45 sec. Note that the execution time for the inner loop is really the only code of significance.

3.4
```
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 `lui`:

```
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)
```

3.6
```
      addi $v0,$zero,-1  # Initialize to avoid counting zero word
loop: lw   $v1,0($a0)    # Read next word from source
      addi $v0,$v0,1     # Increment count words copied
      sw   $v1,0($a1)    # Write to destination
      addi $a0,$a0,4     # Advance pointer to next source
      addi $a1,$a1,4     # Advance pointer to next dest
      bne  $v1,$zero,loop# Loop if the word copied ≠ 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

Instruction	Format	op	rs	rt	immediate
`lw $v1,0($a0)`	I	35	4	3	0
`addi $v0,$v0,1`	I	8	2	2	1
`sw $v1,0($a1)`	I	43	5	3	0
`addi $a0,$a0,1`	I	8	4	4	1
`addi $a1,$a1,1`	I	8	5	5	1
`bne $v1,$zero,loop`	I	5	3	0	–20

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
      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 iterations of the loop plus a final pass that goes to Exit at the bne instruction before updating 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]
      bne $t0, $s5, Exit   # go to Exit 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 $7 + 10 \times 3 = 37$.

3.10

Pseudoinstruction	What it accomplishes	Solution
`move $t5, $t3`	`$t5 = $t3`	`add $t5, $t3, $zero`
`clear $t5`	`$t5 = 0`	`add $t5, $zero, $zero`
`li $t5, small`	`$t5 = small`	`addi $t5, $zero, small`
`li $t5, big`	`$t5 = big`	`lui $t5, upper_half(big)` `ori $t5, $t5, lower_half(big)`

`lw $t5, big($t3)`	`$t5 = Memory[$t3 + big]`	`li $at, big`
		`add $at, $at, $t3`
		`lw $t5, 0($at)`

`addi $t5, $t3, big`	`$t5 = $t3 + big`	`li $at, big` `add $t5, $t3, $at`
`beq $t5, small, L`	`if ($t5 = small) go to L`	`li $at, small` `beq $t5, $at, L`
`beq $t5, big, L`	`if ($t5 = big) go to L`	`li $at, big` `beq $at, $zero, L`
`ble $t5, $t3, L`	`if ($t5 <= $t3) go to L`	`slt $at, $t3, $t5` `beq $at, $zero, L`
`bgt $t5, $t3, L`	`if ($t5 > $t3) go to L`	`slt $at, $t3, $t5` `bne $at, $zero, L`
`bge $t5, $t3, L`	`if ($t5 >= $t3) go to L`	`slt $at, $t5, $t3` `beq $at, $zero, L`

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

3.11 The fragment of C code is

```
for (i=0; i<=100; i=i+1) {a[i] = b[i] + c;}
```

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

```
        add $t0, $zero, $zero  # Temp reg $t0 = 0
```

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

```
        lw $t1, 0($s0)         # Temp reg $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 $t2, AddressConstant4($zero)    # Temp reg $t2 = 4
        lw $t3, AddressConstant401($zero)  # Temp reg $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 ≤ 100
      bne $t8, $zero, Loop # go to Loop if i ≤ 100
```

The number of instructions executed is 4 + 101 × 8 = 812. The number of data references made is 3 + 101 × 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 × 1.0 + .33 × 1.4 + .17 × 1.7 + .02 × 1.2) × I cycles; if we call the unmodified machine's cycle time C seconds, then the time taken on the unmodified machine is (.48 × 1.0 + .33 × 1.4 + .17 × 1.7 + .02 × 1.2) × I × 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 × I × f data transfer instructions that is changed corresponds to an arithmetic instruction that can be eliminated. So, there are now only (.48 –(.33 × f)) × I arithmetic instructions, and the modified machine, with its cycle time of 1.1 × C seconds, will take ((.48 – .33f) × 1.0 + .33 × 1.4 + .17 × 1.7 + .02 × 1.2) × I × 1.1 × 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 = $\frac{100 + \left(33 \times \frac{2}{3}\right)}{133}$ = 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 = 41 / 141 = 29%.

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

3.16 Effective CPI = $\sum_{\substack{\text{Instruction}\\ \text{classes}}} \text{CPI}_{\text{class}} \times \text{Frequency of execution}_{\text{class}}$

For gcc, CPI = 1.0 × 0.48 + 1.4 × 0.33 + 1.7 × 0.17 + 1.2 × 0.02 = 1.3. For spice, CPI = 1.0 × 0.5 + 1.4 × 0.41 + 1.7 × 0.08 + 1.2 × 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.

$\text{exec}_{\text{old}} = n \times \text{CPI} \times t$
$\text{exec}_{\text{new}} = (0.78\, n + N \times 0.22\, n) \times \text{CPI} \times 1.1t$

$\text{exec}_{\text{new}} \le \text{exec}_{\text{old}}$
$(0.78\, n + N \times 0.22\, n) \times \text{CPI} \times 1.1t \le n \times \text{CPI} \times t$
$(0.78\, n + N \times 0.22n) \times 1.1 \le n$
$(0.78 + N \times 0.22) \times 1.1 \le 1$

$0.78 + N \times 0.22 \le \frac{1}{1.1}$

$N \times 0.22 \le \frac{1}{1.1} - 0.78$

$N \le \frac{\frac{1}{1.1} - 0.78}{0.22}$

$N \le \frac{1 - 1.1 \times 0.78}{1.1 \times 0.22}$

$N \le 0.587$

We need to eliminate at least 41.3% of loads.

3.18 No solution provided.

3.19

Accumulator		
Instruction	Code bytes	Data bytes
`load b  # Acc = b;`	3	4
`add c  # Acc += c;`	3	4
`store a  # a = Acc;`	3	4
`add c  # Acc += c;`	3	4
`store b  # Acc = b;`	3	4
`neg  # Acc =- Acc;`	1	0
`add a  # Acc -= b;`	3	4
`store d  # d = Acc;`	3	4
Total:	22	28

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

Stack		
Instruction	Code bytes	Data bytes
`push b`	3	4
`push c`	3	4
`add`	1	0
`dup`	1	0
`pop a`	3	4
`push c`	3	4
`add`	1	0
`dup`	1	0
`pop b`	3	4
`neg`	1	0
`push a`	3	4
`add`	1	0
`pop d`	3	4
Total:	27	28

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

Memory-Memory		
Instruction	Code bytes	Data bytes
`add a, b, c # a=b+c`	7	12
`add b, a, c # b=a+c`	7	12
`sub d, a, b # d=a-b`	7	12
Total:	21	36

Code size is 21 bytes, and memory bandwidth is 21 + 36 = 57 bytes.

Load-Store		
Instruction	Code bytes	Data bytes
`load  $1, b        # $1 = b;`	4	4
`load  $2, c        # $2 = c;`	4	4
`add   $3, $1, $2   # $3 = $1 + $2`	3	0
`store $3, a        # a = $3;`	4	4
`add   $1, $2, $3   # $1 = $2 + $3;`	3	0
`store $1, b        # b = $1;`	4	4
`sub   $4, $3, $1   # $4 = $3 - $1;`	3	0
`store $4, d        # d = $4;`	4	4
Total:	29	20

Code size is 29 bytes, and memory bandwidth is 29 + 20 = 49 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.

A		b	y	t	e		i	s		8		b	i	t	s	
65	32	98	121	116	101	32	101	115	32	56	32	98	101	116	115	0

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 Programing Language* by Kernighan and Ritchie:

```
    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:

```
.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                       #    n /= 10;
          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    $a1, $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
```

```
main:     addi    $29, $29, -4
          sw      $31, 0($29)
          li      $v0, 4
          la      $a0, hello
          syscall
          li      $v0, 5                  # read_int
          syscall
          move    $a0, $v0                # itoa( $a0, str );
          la      $a1, str                #
          jal     itoa                    #
          la      $a0, str
          li      $v0, 4
          syscall
          la      $a0, newln
          syscall
          lw      $31, 0($29)
          addi    $29, $29, 4
          jr      $31
```

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 "\n\nThis program computes the Fibonacci
         function."
prmpt2:  .asciiz "\nEnter 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, fibne1    # if n == 1 ...
        jr $31                  # ... return 1
fibne1:                         # 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 "\n\nThis program computes the the Fibonacci
                  function."
prmpt2:   .asciiz "\nEnter 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 $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 ...
          la $a0, descr     # ... passing address of output
                            # descriptor
          syscall
          li $v0, 1         # print_int system service ...
          move $a0, $s1     # ... passing argument fib(n)
          syscall
# 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 ...
          move $v0, $a1             # ... return b
          jr $31
fibne0:                             # Assert: n != 0
          addi $a2, $a2, -1         # count = count - 1
          add $t1, $a0, $a1         # $t1 = a + b
          move $a1, $a0             # b = a
          move $a0, $t1             # a = a + old b
          j fib                     # tail call fib(a+b, a, count-1)
```

3.28 No solution provided.