Solutions for Chapter 2 Exercises

2.2 By lookup using the table in Figure 2.5 on page 62,

7fff fffa$_{hex}$ = 0111 1111 1111 1111 1111 1111 1111 1010$_{two}$

= 2,147,483,642$_{ten}$.

2.3 By lookup using the table in Figure 2.5 on page 62,

1100 1010 1111 1110 1111 1010 1100 1110$_{two}$ = cafe face$_{hex}$.

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

2.6

```
sll  $t0, $t3, 9    # shift $t3 left by 9, store in $t0
srl  $t0, $t0, 15   # shift $t0 right by 15
```

2.8 One way to implement the code in MIPS:

```
sll    $s0, $s1, 22      # shift receiver left by 22, store in data
srl    $s0, $s0, 24      # shift data right by 24 (data = receiver.receivedByte)
andi   $s1, $s1, 0xfffe  # receiver.ready = 0;
ori    $s1, $s1, 0x0002  # receiver.enable = 1;
```

Another way:

```
srl   $s0, $s1, 2        # data = receiver.receivedByte
andi  $s0, $s0, 0x00ff
andi  $s1, $s1, 0xfffe  # receiver.ready = 0;
ori   $s1, $s1, 0x0002  # receiver.enable = 1;
```

2.9

```
lb   $s0, 0($s1)              # load the lower 8 bytes of a into bits
sll  $t0, $s0, 8              # $t0 = bits << 8
or   $s0, $s0, $t0            # bits.data1 = bits.data0
lui  $s0, 0000 0000 0110 0100  # bits.data2 = 'd'
lui  $t0, 0000 0001 0000 0000  # load a 1 into the upper bits of $t0
or   $s0, $s0, $t0            # bits.valid = 1
```

2.10

```
     slt $t3, $s5, $zero       # test k < 0
     bne $t3, $zero, Exit      # if so, exit
     slt $t3, $s5, $t2         # test k < 4
     beq $t3, $zero, Exit      # if not, exit
     sll $t1, $s5, 2           # $t1 = 4*k
     add $t1, $t1, $t4         # $t1 = &JumpTable[k]
     lw $t0, 0($t1)            # $t0 = JumpTable[k]
     jr $t0                    # jump register
L0:  add $s0, $s3, $s4         # k == 0
     j Exit                    # break
L1:  add $s0, $s1, $s2         # k == 1
     j Exit                    # break
L2:  sub $s0, $s1, $s2         # k == 2
     j Exit                    # break
L3:  sub $s0, $s3, $s4         # k == 3
     j Exit                    # break
Exit:
```

2.11

a.

```
if (k==0) f = i + j;
else if (k==1) f = g + h;
else if (k==2) f = g - h;
else if (k==3) f = i - j;
```

b.

```
          bne   $s5, $0, C1      # branch k != 0
          add   $s0, $s3, $s4    # f = i + j
          j     Exit             # break
     C1:  addi  $t0, $s5, -1     # $t0 = k - 1
          bne   $t0, $0, C2      # branch k != 1
          add   $s0, $s1, $s2    # f = g + h
          j     Exit             # break
     C2:  addi  $t0, $s5, -2     # $t0 = k - 2
          bne   $t0, $0, C3      # branch k != 2
          sub   $s0, $s1, $s2    # f = g - h
          j     Exit             # break
     C3:  addi  $t0, $s5, -3     # $t0 = k - 3
          bne   $t0, $0, Exit    # branch k != 3
          sub   $s0, $s3, $s4    # f = i - j
     Exit:
```

c. The MIPS code from the previous problem would yield the following results:

(5 arithmetic)1.0 + (1 data transfer)1.4 + (2 conditional branch)1.7 + (2 jump)1.2 = 12.2 cycles

while the MIPS code from this problem would yield the following:

(4 arithmetic)1.0 + (0 data transfer)1.4 + (4 conditional branch)1.7 + (0 jump)1.2 = 10.8 cycles

2.12 The technique of using jump tables produces MIPS code that is independent of N, and always takes the following number of cycles:

(5 arithmetic)1.0 + (1 data transfer)1.4 + (2 conditional branch)1.7 + (2 jump)1.2 = 12.2 cycles

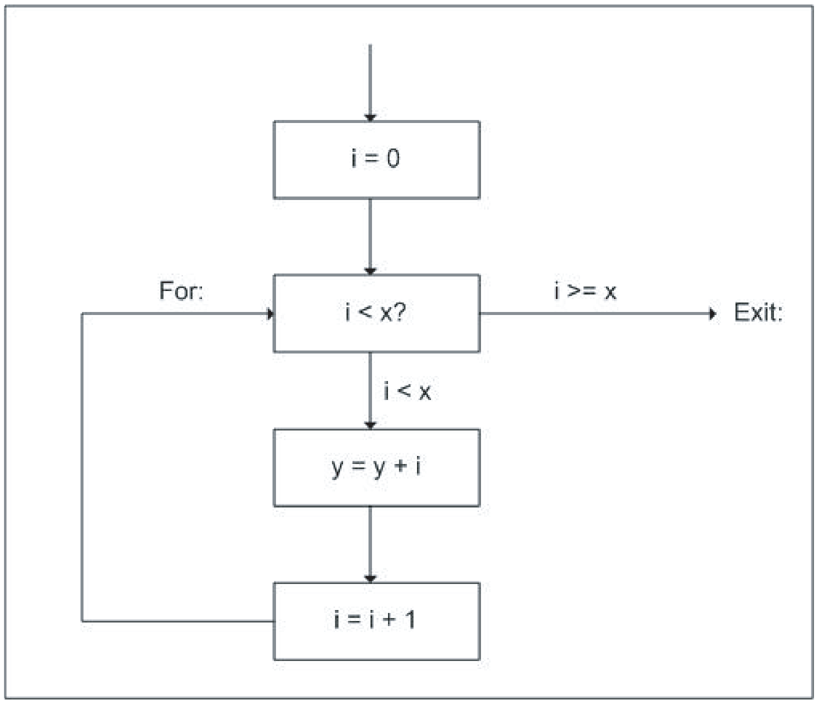
However, using chained conditional jumps takes the following number of cycles in a worst-case scenario:

(N arithmetic)1.0 + (0 data transfer)1.4 + (N conditional branch)1.7 + (0 jump)1.2 = $2.7N$ cycles

Hence, jump tables are faster for the following condition:

$N > 12.2/2.7 = 5$ *case* statements

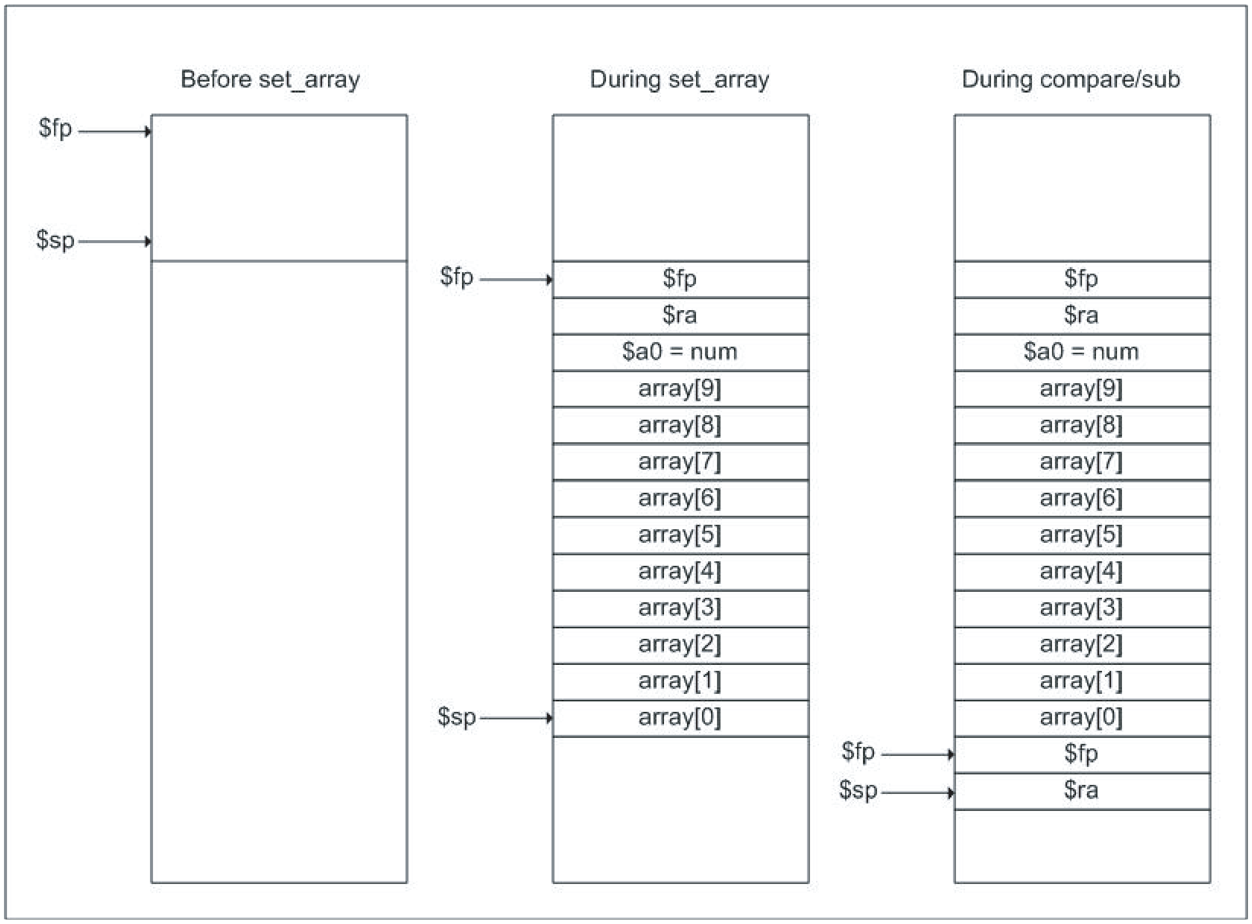
2.13



2.15 Hence, the results from using *if-else* statements are better.

```
set_array:  addi  $sp, $sp, -52      # move stack pointer
            sw    $fp, 48($sp)       # save frame pointer
            sw    $ra, 44($sp)       # save return address
            sw    $a0, 40($sp)       # save parameter (num)
            addi  $fp, $sp, 48       # establish frame pointer

            add   $s0, $zero, $zero  # i = 0
            addi  $t0, $zero, 10     # max iterations is 10
loop:       sll   $t1, $s0, 2        # $t1 = i * 4
            add   $t2, $sp, $t1      # $t2 = address of array[i]
            add   $a0, $a0, $zero    # pass num as parameter
            add   $a1, $s0, $zero    # pass i as parameter
            jal   compare            # call compare(num, i)
            sw    $v0, 0($t2)        # array[i] = compare(num, i);
            addi  $s0, $s0, 1
            bne   $s0, $t0, loop     # loop if i<10

            lw    $a0, 40($sp)       # restore parameter (num)
            lw    $ra, 44($sp)       # restore return address
            lw    $fp, 48($sp)       # restore frame pointer
            addi  $sp, $sp, 52       # restore stack pointer
            jr    $ra                # return

compare:    addi  $sp, $sp, -8       # move stack pointer
            sw    $fp, 4($sp)        # save frame pointer
            sw    $ra, 0($sp)        # save return address
            addi  $fp, $sp, 4        # establish frame pointer

            jal   sub                # can jump directly to sub
            slt   $v0, $v0, $zero    # if sub(a,b) >= 0, return 1
            slti  $v0, $v0, 1

            lw    $ra, 0($sp)        # restore return address
            lw    $fp, 4($sp)        # restore frame pointer
            addi  $sp, $sp, 8        # restore stack pointer
            jr    $ra                # return

sub:       sub    $v0, $a0, $a1      # return a-b
            jr    $ra                # return
```

The following is a diagram of the status of the stack:



2.16

```
# 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 nonnegative 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     # $s1 = 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(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
```

2.17

```
# 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 nonnegative 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      # $s1 = 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)
```

2.18 No solution provided.

2.19 Iris in ASCII:	73 114 105 115
Iris in Unicode:	0049 0072 0069 0073
Julie in ASCII:	74 117 108 105 101
Julie in Unicode:	004A 0075 006C 0069 0065

2.20 Figure 2.21 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

2.29

```
         add  $t0, $zero, $zero    # initialize running sum $t0 = 0
loop:    beq  $a1, $zero, finish   # finished when $a1 is 0
         add  $t0, $t0, $a0        # compute running sum of $a0
         sub  $a1, $a1, 1          # compute this $a1 times
         j    loop
finish:  addi $t0, $t0, 100        # add 100 to a * b
         add  $v0, $t0, $zero      # return a * b + 100
```

The program computes a * b + 100.

2.30

```
        sll  $a2, $a2, 2         # max i= 2500 * 4
        sll  $a3, $a3, 2         # max j= 2500 * 4
        add  $v0, $zero, $zero   # $v0 = 0
        add  $t0, $zero, $zero   # i = 0
outer:  add  $t4, $a0, $t0       # $t4 = address of array 1[i]
        lw   $t4, 0($t4)         # $t4 = array 1[i]
        add  $t1, $zero, $zero   # j = 0
inner:  add  $t3, $a1, $t1       # $t3 = address of array 2[j]
        lw   $t3, 0($t3)         # $t3 = array 2[j]
        bne  $t3, $t4, skip      # if (array 1[i] != array 2[j]) skip $v0++
        addi $v0, $v0, 1         # $v0++
skip    addi $t1, $t1, 4         # j++
        bne  $t1, $a3, inner     # loop if j != 2500 * 4
        addi $t0, $t0, 4         # i++
        bne  $t0, $a2, outer     # loop if i != 2500 * 4
```

The code determines the number of matching elements between the two arrays and returns this number in register `$v0`.

2.31 Ignoring the four instructions before the loops, we see that the outer loop (which iterates 2500 times) has three instructions before the inner loop and two after. The cycles needed to execute these are 1 + 2 + 1 = 4 cycles and 1 + 2 = 3 cycles, for a total of 7 cycles per iteration, or 2500×7 cycles. The inner loop requires 1 + 2 + 2 + 1 + 1 + 2 = 9 cycles per iteration and it repeats 2500×2500 times, for a total of $9 \times 2500 \times 2500$ cycles. The total number of cycles executed is therefore $(2500 \times 7) + (9 \times 2500 \times 2500) = 56{,}267{,}500$. The overall execution time is therefore $(56{,}267{,}500) / (2 \times 10^9) = 28$ ms. Note that the execution time for the inner loop is really the only code of significance.

2.32 `ori  $t1, $t0, 25    # register $t1 = $t0 | 25;`

2.34

```
      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 destination
      bne  $v1, $zero, loop  # Loop if word copied != zero
```

Bug 1: Count (`$v0`) is initialized to zero, not –1 to avoid counting zero word.

Bug 2: Count (`$v0`) is not incremented.

Bug 3: Loops if word copied is equal to zero rather than not equal.

2.37

Pseudoinstruction	What it accomplishes	Solution
`move $t1, $t2`	`$t1 = $t2`	`add   $t1, $t2, $zero`
`clear $t0`	`$t0 = 0`	`add   $t0, $zero, $zero`

`beq $t1, small, L`	if (`$t1 == small`) go to `L`	`li    $at, small`
		`beq   $t1, $at, L`

`beq $t2, big, L`	if (`$t2 == big`) go to `L`	`li    $at, big`
		`beq   $at, $zero, L`

`li $t1, small`	`$t1 = small`	`addi  $t1, $zero, small`

`li $t2, big`	`$t2 = big`	`lui   $t2, upper(big)`
		`ori   $t2, $t2, lower(big)`

`ble $t3, $t5, L`	if (`$t3 <= $t5`) go to `L`	`slt   $at, $t5, $t3`
		`beq   $at, $zero, L`

`bgt $t4, $t5, L`	if (`$t4 > $t5`) go to `L`	`slt   $at, $t5, $t4`
		`bne   $at, $zero, L`

`bge $t5, $t3, L`	if (`$t5 >= $t3`) go to `L`	`slt   $at, $t5, $t3`
		`beq   $at, $zero, L`

`addi $t0, $t2, big`	`$t0 = $t2 + big`	`li    $at, big`
		`add   $t0, $t2, $at`

`lw $t5, big($t2)`	`$t5 =` Memory[`$t2 + big`]	`li    $at, big`
		`add   $at, $at, $t2`
		`lw    $t5, $t2, $at`

Note: In the solutions, we make use of the `li` instruction, which should be implemented as shown in rows 5 and 6.

2.38 The problem is that we are using PC-relative addressing, so if that address 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    $s0, $s2, skip
      j      there
skip:
      …
there: add   $s0, $s0, $s0
```

This will work as long as our program does not cross the 256MB address boundary described in the elaboration on page 98.

2.42 Compilation times and run times will vary widely across machines, but in general you should find that compilation time is greater when compiling with optimizations and that run time is greater for programs that are compiled without optimizations.

2.45 Let I be the number of instructions taken on the unmodified MIPS. This decomposes into $0.42I$ arithmetic instructions (24% arithmetic and 18% logical), $0.36I$ data transfer instructions, $0.18I$ conditional branches, and 0.03I jumps. Using the CPIs given for each instruction class, we get a total of $(0.42 \times 1.0 + 0.36 \times 1.4 + 0.18 \times 1.7 + 0.03 \times 1.2) \times I$ cycles; if we call the unmodified machine's cycle time C seconds, then the time taken on the unmodified machine is $(0.42 \times 1.0 + 0.36 \times 1.4 + 0.18 \times 1.7 + 0.03 \times 1.2) \times I \times C$ seconds. Changing some fraction, f (namely 0.25) 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 $0.36 \times I \times f$ data transfer instructions that are changed corresponds to an arithmetic instruction that can be eliminated. So, there are now only $(0.42 - (0.36 \times f)) \times I$ arithmetic instructions, and the modified machine, with its cycle time of $1.1 \times C$ seconds, will take $((0.42 - 0.36f) \times 1.0 + 0.36 \times 1.4 + 0.18 \times 1.7 + 0.03 \times 1.2) \times \mathrm{I} \times 1.1 \times C$ seconds to execute. When f is 0.25, the unmodified machine is 2.2% faster than the modified one.

2.46 *Code before:*

```
        lw   $t2, 4($s6)                 # temp reg $t2 = length of array save
Loop:   slt  $t0, $s3, $zero             # temp reg $t0 = 1 if i < 0
        bne  $t0, $zero, IndexOutOfBounds   # if i< 0, goto Error
        slt  $t0, $s3, $t2               # temp reg $t0 = 0 if i >= length
        beq  $t0, $zero, IndexOutOfBounds   # if i >= length, goto Error
        sll  $t1, $s3, 2                 # temp reg $t1 = 4 * i
        add  $t1, $t1, $s6               # $t1 = address of save[i]
        lw   $t0, 8($t1)                 # temp reg $t0 = save[i]
        bne  $t0, $s5, Exit              # go to Exit if save[i] != k
        addi $s3, $s3, 1                 # i = i + 1
        j    Loop
Exit:
```

The number of instructions executed over 10 iterations of the loop is $10 \times 10 + 8 + 1 = 109$. This corresponds to 10 complete iterations of the loop, plus a final pass that goes to Exit from the final `bne` instruction, plus the initial `lw` instruction. Optimizing to use at most one branch or jump in the loop in addition to using only at most one branch or jump for out-of-bounds checking yields:

Code after:

```
        lw   $t2, 4($s6)                 # temp reg $t2 = length of array save
        slt  $t0, $s3, $zero             # temp reg $t0 = 1 if i < 0
        slt  $t3, $s3, $t2               # temp reg $t3 = 0 if i >= length
        slti $t3, $t3, 1                 # flip the value of $t3
        or   $t3, $t3, $t0               # $t3 = 1 if i is out of bounds
        bne  $t3, $zero, IndexOutOfBounds  # if out of bounds, goto Error
        sll  $t1, $s3, 2                 # tem reg $t1 = 4 * 1
        add  $t1, $t1, $s6               # $t1 = address of save[i]
        lw   $t0, 8($t1)                 # temp reg $t0 = save[i]
        bne  $t0, $s5, Exit              # go to Exit if save[i] != k
Loop:   addi $s3, $s3, 1                 # i = i + 1
        slt  $t0, $s3, $zero             # temp reg $t0 = 1 if i < 0
        slt  $t3, $s3, $t2               # temp reg $t3 = 0 if i >= length
        slti $t3, $t3, 1                 # flip the value of $t3
        or   $t3, $t3, $t0               # $t3 = 1 if i is out of bounds
        bne  $t3, $zero, IndexOutOfBounds  # if out of bounds, goto Error
        addi $t1, $t1, 4                 # temp reg $t1 = address of save[i]
        lw   $t0, 8($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 $10 + 10 * 9 = 100$.

2.47 To test for loop termination, the constant 401 is needed. Assume that it is placed in memory when the program is loaded:

```
       lw   $t8, AddressConstant401($zero) # $t8 = 401
       lw   $t7, 4($a0)                    # $t7 = length of a[]
       lw   $t6, 4($a1)                    # $t6 = length of b[]
       add  $t0, $zero, $zero              # initialize i = 0
Loop:  slt  $t4, $t0, $zero                # $t4 = 1 if i < 0
       bne  $t4, $zero, IndexOutOfBounds   # if i< 0, goto Error
       slt  $t4, $t0, $t6                  # $t4 = 0 if i >= length
       beq  $t4, $zero, IndexOutOfBounds   # if i >= length, goto Error
       slt  $t4, $t0, $t7                  # $t4 = 0 if i >= length
       beq  $t4, $zero, IndexOutOfBounds   # if i >= length, goto Error
       add  $t1, $a1, $t0                  # $t1 = address of b[i]
       lw   $t2, 8($t1)                    # $t2 = b[i]
       add  $t2, $t2, $s0                  # $t2 = b[i] + c
       add  $t3, $a0, $t0                  # $t3 = address of a[i]
       sw   $t2, 8($t3)                    # a[i] = b[i] + c
       addi $t0, $t0, 4                    # i = i + 4
       slt  $t4, $t0, $t8                  # $t8 = 1 if $t0 < 401, i.e., i <= 100
       bne  $t4, $zero, Loop               # goto Loop if i <= 100
```

The number of instructions executed is $4 + 101 \times 14 = 1418$. The number of data references made is $3 + 101 \times 2 = 205$.

2.48

```
compareTo: sub $v0, $a0, $a1   # return v[i].value - v[j+1].value
           jr  $ra             # return from subroutine
```

2.49 From Figure 2.44 on page 141, 36% of all instructions for SPEC2000int are data access instructions. Thus, for every 100 instructions there are 36 data accesses, yielding a total of 136 memory accesses (1 to read each instruction and 36 to access data).

a. The percentage of all memory accesses that are for data = 36/136 = 26%.

b. Assuming two-thirds of data transfers are loads, the percentage of all memory accesses that are reads = $(100 + (36 \times 2/3))/136 = 91\%$.

2.50 From Figure 2.44, 39% of all instructions for SPEC2000fp are data access instructions. Thus, for every 100 instructions there are 39 data accesses, yielding a total of 139 memory accesses (1 to read each instruction and 39 to access data).

a. The percentage of all memory accesses that are for data = 39/139 = 28%.

b. Assuming two-thirds of data transfers are loads, the percentage of all memory accesses that are reads = (100 + (39 × 2/3))/139 = 91%.

2.51 Effective CPI = Sum of (CPI of instruction type × Frequency of execution)

The average instruction frequencies for SPEC2000int and SPEC2000fp are 0.47 arithmetic (0.36 arithmetic and 0.11 logical), 0.375 data transfer, 0.12 conditional branch, 0.015 jump. Thus, the effective CPI is 0.47 × 1.0 + 0.375 × 1.4 + 0.12 × 1.7 + 0.015 × 1.2 = 1.2.

2.52

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

Stack		
Instruction	**Code bytes**	**Data bytes**
`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.

2.53 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.

2.54

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

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

```
        sbn  c, c, .+1       # c = 0;
        sbn  tmp, tmp, .+1   # tmp = 0;
 loop:  sbn  b, one, end     # while (--b >= 0)
        sbn  tmp, a, loop    # tmp -= a; /* always continue */
 end:   sbn  c, tmp, .+1     # c = -tmp; /* = a × b */
```

2.56 Without a stored program, the programmer must physically configure the machine to run the desired program. Hence, a nonstored-program machine is one where the machine must essentially be rewired to run the program. The problem with such a machine is that much time must be devoted to reprogramming the machine if one wants to either run another program or fix bugs in the current program. The stored-program concept is important in that a programmer can quickly modify and execute a stored program, resulting in the machine being more of a general-purpose computer instead of a specifically wired calculator.

2.57

MIPS:

```
      add   $t0, $zero, $zero   # i = 0
      addi  $t1, $zero, 10      # set max iterations of loop
loop: sll   $t2, t0, 2          # $t2 = i * 4
      add   $t3, $t2, $a1       # $t3 = address of b[i]
      lw    $t4, 0($t3)         # $t4 = b[i]
      add   $t4, $t4, $t0       # $t4 = b[i] + i
      sll   $t2, t0, 4          # $t2 = i * 4 * 2
      add   $t3, $t2, $a0       # $t3 = address of a[2i]
      sw    $t4, 0($t3)         # a[2i] = b[i] + i
      addi  $t0, $t0, 1         # i++
      bne   $t0, $t1, loop      # loop if i != 10
```

PowerPC:

```
      add   $t0, $zero, $zero   # i = 0
      addi  $t1, $zero, 10      # set max iterations of loop
loop: lwu   $t4, 4($a1)         # $t4 = b[i]
      add   $t4, $t4, $t0       # $t4 = b[i] + i
      sll   $t2, t0, 4          # $t2 = i * 4 * 2
      sw    $t4, $a0+$t2        # a[2i] = b[i] + i
      addi  $t0, $t0, 1         # i++
      bne   $t0, $t1, loop      # loop if i != 10
```

2.58

MIPS:

```
        add  $v0, $zero, $zero    # freq = 0
        add  $t0, $zero, $zero    # i = 0
        addi $t8, $zero, 400      # $t8 = 400
outer:  add  $t4, $a0, $t0        # $t4 = address of a[i]
        lw   $t4, 0($t4)          # $t4 = a[i]
        add  $s0, $zero, $zero    # x = 0
        addi $t1, $zero, 400      # j = 400
inner:  add  $t3, $a0, $t1        # $t3 = address of a[j]
        lw   $t3, 0($t3)          # $t3 = a[j]
        bne  $t3, $t4, skip       # if (a[i] != a[j]) skip x++
        addi $s0, $s0, 1          # x++
skip:   addi $t1, $t1, -4         # j--
        bne  $t1, $zero, inner    # loop if j != 0
        slt  $t2, $s0, $v0        # $t2 = 0 if x >= freq
        bne  $t2, $zero, next     # skip freq = x if
        add  $v0, $s0, $zero      # freq = x
next:   addi $t0, $t0, 4          # i++
        bne  $t0, $t8, outer      # loop if i != 400
```

PowerPC:

```
        add  $v0, $zero, $zero    # freq = 0
        add  $t0, $zero, $zero    # i = 0
        addi $t8, $zero, 400      # $t8 = 400
        add  $t7, $a0, $zero      # keep track of a[i] with update addressing
outer:  lwu  $t4, 4($t7)          # $t4 = a[i]
        add  $s0, $zero, $zero    # x = 0
        addi $ctr, $zero, 100     # j = 100
        add  $t6, $a0, $zero      # keep track of a[j] with update addressing
inner:  lwu  $t3, 4($t6)          # $t3 = a[j]
        bne  $t3, $t4, skip       # if (a[i] != a[j]) skip x++
        addi $s0, $s0, 1          # x++
skip:   bc   inner, $ctr!=0      # j--, loop if j!=0
        slt  $t2, $s0, $v0       # $t2 = 0 if x >= freq
        bne  $t2, $zero, next    # skip freq = x if
        add  $v0, $s0, $zero     # freq = x
next:   addi $t0, $t0, 4         # i++
        bne  $t0, $t8, outer     # loop if i != 400
```

2.59

```
xor $s0, $s0, $s1
xor $s1, $s0, $s1
xor $s0, $s0, $s1
```