For More Practice

Instruction Formats

2.1 [15] <§2.4> Using the MIPS program in Exercise 2.34 (with bugs intact), determine the instruction format for each instruction and the decimal values of each instruction field.

MIPS Code and Logical Operations

2.5 [15] <§2.5> Consider the following code used to implement the instruction

```
sllv $s0, $s1, $s2
```

which uses the least significant 5 bits of the value in register `$s2` to specify the amount register `$s1` should be shifted left:

```
          .data
mask:     .word  0xfffff83f
          .text
start:    lw     $t0, mask
          lw     $s0, shifter
          and    $s0,$s0,$t0
          andi   $s2,$s2,0x1f
          sll    $s2,$s2,6
          or     $s0,$s0,$s2
          sw     $s0, shifter
shifter:  sll    $s0,$s1,0
```

Add comments to the code and write a paragraph describing how it works. Note that the two `lw` instructions are pseudoinstructions that use a label to specify a memory address that contains the word of data to be loaded. Why do you suppose that writing "self-modifying code" such as this is a bad idea (and oftentimes not actually allowed)?

Logical Operations in MIPS

2.7 [10] <§2.5> The following MIPS instruction sequence could be used to implement a new instruction that has two register operands. Give the instruction a name and describe what it does. Note that register `$t0` is being used as a temporary.

```
srl   $s1, $s1, 1    #
sll   $t0, $s0, 31   #  These 4 instructions accomplish
srl   $s0, $s0, 1    #  "new $s0 $s1"
or    $s1, $s1, $t0  #
```

Writing Assembly Code

2.14 [10] <§2.6> The MIPS translation of the C segment

```
while (save[i] == k)
  i = i += 1;
```

on page 72 uses both a conditional branch and an unconditional jump each time through the loop. Only poor compilers would produce code with this loop overhead. Rewrite the assembly code so that it uses at most one branch or jump each time through the loop. How many instructions are executed before and after the optimization if the number of iterations of the loop is 10 (i.e., `save[i + 10 * j]` do not equal `k` and `save[i], . . . , save[i + 9 * j]` equal `k`)?

MIPS Coding and ASCII Strings

2.21 [30] <§§2.7, 2.8> Write a program in MIPS assembly language to convert an ASCII decimal string to an integer. Your program should expect register `$a0` to hold the address of a null-terminated string containing some combination of the digits 0 through 9. Your program should compute the integer value equivalent to this string of digits, then place the number in register `$v0`. Your program need not handle negative numbers. If a nondigit character appears anywhere in the string, your program should stop with the value –1 in register `$v0`. For example, if register `$a0` points to a sequence of three bytes 50_{ten}, 52_{ten}, 0_{ten} (the null-terminated string "24"), then when the program stops, register `$v0` should contain the value 24_{ten}. (The subscript "ten" means base 10.)

2.22 [20] <§§2.7, 2.8> Write a procedure, `bcount`, in MIPS assembly language. The `bcount` procedure takes a single argument, which is a pointer to a string in register `$a0`, and it returns a count of the total number of `b` characters in the string in register `$v0`. You must use your `bfind` procedure in Exercise 2.36 in your implementation of `bcount`.

2.23 [20] <§§2.7, 2.8> Write a procedure, `bfind`, in MIPS assembly language. The procedure should take a single argument that is a pointer to a null-terminated string in register `$a0`. The `bfind` procedure should locate the first `b` character in the string and return its address in register `$v0`. If there are no `b`'s in the string, then `bfind` should return a pointer to the null character at the end of the string. For example, if the argument to `bfind` points to the string "`imbibe`," then the return value will be a pointer to the third character of the string.

2.24 [30] <§§2.7, 2.8> Write a procedure, `itoa`, in MIPS assembly language that will convert an integer argument into an ASCII decimal string. The procedure should take two arguments: the first is an integer in register `$a0`; the sec-

ond is the address at which to write a result string in register `$a1`. Then `itoa` should convert its first argument to a null-terminated decimal ASCII string and store that string at the given result location. The return value from `itoa`, in register `$v0`, should be a count of the number of nonnull characters stored at the destination.

Comparing C/Java to MIPS

2.25 Some C programmers do not understand the distinction between `x = y` and `*x = *y`. Assume `x` is associated with register `$s0`, `y` with `$s1`. Here are six MIPS instructions, labeled L1 to L6:

- `L1: add $s0, $s1, zero`
- `L2: add $s1, $s0, zero`
- `L3: lw $s0, 0($s1)`

Which is true? ("`L4; L5`" means `L4` then `L5`)

- A: `L1` is `x = y;` `L6` is `*x = *y`?
- B: `L2` is `x = y;` `L3` is `*x = *y`?
- C: `L4`; `L5` is `x = y;` `L3` is `*x = *y`?
- D: `L2` is `x = y;` `L4` is `*x = *y`?
- E: `L2` is `x = y;` `L4`; `L5` is `*x = *y`?
- F: `L1` is `x = y;` `L4`; `L5` is `*x = *y`

Translating MIPS to C

2.26 C/Java versus MIPS: Which statements (if any) are false?

1. Assignment statements: One variable on left-hand side in C/Java; one variable (register) is destination in MIPS.

2. Assignment statements: Any number of variables on right-hand side in C/Java; 1 or 2 (registers) source in MIPS.

3. Comments: /* ... */ in C/Java; // to end of line in Java; and
# to end of line in MIPS

4. Variables: declared in C/Java; no declaration in MIPS.

5. Types: Associated with declaration in C/Java (normally); associated with instruction (operator) in MIPS.

Understanding MIPS Code

2.27 MIPS to C. Assume `$s3 = i, $s4 = j, $s5 = @A`. Below is the MIPS code:

```
Loop:  addi $s4,$s4,1       # j = j + 1?
       add $t1,$s3,$s3      # $t1 = 2 * i
       add $t1,$t1,$t1      # $t1 = 4 * i
       add $t1,$t1,$s5      # $t1 = @ A[i]
       lw  $t0,0($t1)       # $t0 = A[i]
       addi $s3,$s3,4       # i = i + 1?
       slti $t1,$t0,10      # $t1 = $t0 < 10?
       beq $t0,$0, Loop     # goto Loop if >=
       slti $t1,$t0, 0      # $t1 = $t0 <  0?
       bne $t0,$0, Loop     # goto Loop if <
```

Below is part of the corresponding C code:

```
do j = j + 1
    while (_______);
```

What C code properly fills in the blank in loop on right?

A: `[i++] >= 10?`

B: `A[i++] >= 10 | A[i] < 0?`

C: `A[i++] >= 10 & A[i] < 0?`

D: `A[i++] >= 10 || A[i] < 0?`

E: `A[i++] >= 10 && A[i] < 0?`

F: None of the above

2.28 Here is some stylized MIPS code associated with procedure call:

```
r:...    # R/W $s0,$v0,$t0,$a0,$sp,$ra,mem
... ### PUSH REGISTER(S) TO STACK?
jal e  # Call e
...    # R/W $s0,$v0,$t0,$a0,$sp,$ra,mem?
jr $ra # Return to caller of r?

e:... # R/W $s0,$v0,$t0,$a0,$sp,$ra,mem?
jr $ra # Return to r
```

What does `r` have to push on the stack before `jal e`?

A: Nothing

B: 1 of (`$s0,$sp,$v0,$t0,$a0,$ra`)

C: 2 of (`$s0,$sp,$v0,$t0,$a0,$ra`)

D: 3 of (`$s0,$sp,$v0,$t0,$a0,$ra`)

E: 4 of (`$s0,$sp,$v0,$t0,$a0,$ra`)

F: 5 of (`$s0,$sp,$v0,$t0,$a0,$ra`)

Translating from C to MIPS

2.33 [10] <§2.9> Show the single MIPS instruction or minimal sequence of instructions for this C statement:

```
x[4] = x[5] + a;
```

Assume that `a` corresponds to register `$t3` and the array `x` has a base address of $6{,}400{,}000_{ten}$.

Reverse Translation from MIPS to C

2.35 [10] <§§2.2, 2.3, 2.6, 2.9> Starting with the corrected program in the answer to Exercise 2.34, write the C code segment that might have produced this code. Assume that variable `source` corresponds to register `$a0`, variable `destination` corresponds to register `$a1`, and variable `count` corresponds to register `$v0`. Show variable declarations, but assume that `source` and `destination` have been initialized to the proper addresses.

2.36 Consider the following fragment of C code:

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

Assume that `a` and `b` are arrays of words and the base address of `a` is in `$a0` and the base address of `b` is in `$a1`. Register `$t0` is associated with variable `i` and register `$s0` with `c`. Write the code for MIPS. How many instructions are executed during the running of this code? How many memory data references will be made during execution?

MIPS Pseudoinstructions

2.39 Suppose

```
lb $s0, 100($zero) #byte@100= 0x0F?
lb $s1, 200($zero) #byte@200= 0xFF
```

What are the values of `$s0` and `$s1`?

A: 15 255

B: 15 –1

C: 15 –255

D: –15 255

E: –15 –1

F: –15 –255

Linking MIPS Code

2.40 Which of the codes below are pseudoinstructions in MIPS assembly language (that is, they are not found directly in the machine language)?

i. `addi $t0, $t1, 40000`

ii. `beq $s0, 10, Exit`

iii. `sub $t0, $t1, 1`

2.41 Which of the following instructions may need to be edited during link phase?

```
Loop:
      lui $at, 0xABCD     # a
      ori $a0,$at, 0xFEDC # b
      jal add_link        # c
      bne $a0,$v0, Loop   # d?
```

Enhancing MIPS Addressing Modes

2.43 [20] <§2.13> In this exercise, we'll examine quantitatively the pros and cons of adding an addressing mode to MIPS that allows arithmetic instructions to directly access memory, as is found on the IA-32. The primary benefit is that fewer instructions will be executed because we won't have to first load a register. The primary disadvantage is that the cycle time will have to increase to account for the additional time to read memory. Consider adding a new instruction:

```
addm $t2, 100($t3) # $t2 = $t2 + Memory[$t3+100]
```

Assume that the new instruction will cause the cycle time to increase by 10%. Use the instruction frequencies for the SPEC2000int from Figure 2.48, and assume that two-thirds of the data transfers are loads and the rest are stores. Assume that the new instruction affects only the clock speed, not the CPI. What percentage of loads must be eliminated for the machine with the new instruction to have at least the same performance?

2.44 [10] <§2.12> Using the information in Exercise 2.26, write a multiple-instruction sequence in which a load of `$t0` followed immediately by the use of `$t0`—in, say, an add—could *not* be replaced by a single instruction of the form proposed.