There are some cases where pseudoinstructions must be used (for example, the `la` instruction when an actual value is not known at assembly time). In many cases, they are quite convenient and result in more readable code (for example, the `ji` and `move` instructions). If you choose to use pseudoinstructions for these reasons, please add a sentence or two to your solution stating which pseudoinstructions you have used and why.

2.1 [15] <§2.4>  For More Practice: Instruction Formats
2.2 [5] <§2.4> What binary number does this hexadecimal number represent: 7fff fff a hex? What decimal number does it represent?
2.3 [5] <§2.4> What hexadecimal number does this binary number represent: 1100 1010 1111 1110 1111 1010 1100 1110 two?
2.4 [5] <§2.4> Why doesn't MIPS have a subtract immediate instruction?
2.5 [15] <§2.5>  For More Practice: MIPS Code and Logical Operations
2.6 [15] <§2.5> Some computers have explicit instructions to extract an arbitrary field from a 32-bit register and to place it in the least significant bits of a register. The figure below shows the desired operation:

```
   31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

     |     |       |
  31 - j bits | j - i bits | i + 1 bits
      j      i   0

   31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

   0...0 0 0 0 0

   32 - (j - i) bits | j - i bits
```

Find the shortest sequence of MIPS instructions that extracts a field for the constant values $i = 5$ and $j = 22$ from register $t.3$ and places it in register $t.0$. (Hint: It can be done in two instructions.)

2.7 [10] <§2.5>  For More Practice: Logical Operations in MIPS
2.8 [20] <§2.5>  In More Depth: Bit Fields in C
2.9 [20] <§2.5>  In More Depth: Bit Fields in C
2.10 [20] <§2.5>  In More Depth: Jump Tables
2.11 [20] <§2.5>  In More Depth: Jump Tables
2.12 [20] <§2.5>  In More Depth: Jump Tables
2.20 [10] <§2.8> Compute the decimal byte values that form the null-terminated ASCII representation of the following string:

A byte is 8 bits

2.21 [30] <§§2.7, 2.8> For More Practice: MIPS Coding and ASCII Strings

2.22 [20] <§§2.7, 2.8> For More Practice: MIPS Coding and ASCII Strings

2.23 [20] <§§2.7, 2.8> [Ex. 2.22] For More Practice: MIPS Coding and ASCII Strings

2.24 [30] <§§2.7, 2.8> For More Practice: MIPS Coding and ASCII Strings

2.25 <§2.8> For More Practice: Comparing C/Java to MIPS

2.26 <§2.8> For More Practice: Translating MIPS to C

2.27 <§2.8> For More Practice: Understanding MIPS Code

2.28 <§2.8> For More Practice: Understanding MIPS Code

2.29 [5] <§§2.3, 2.6, 2.9> Add comments to the following MIPS code and describe in one sentence what it computes. Assume that $a0 and $a1 are used for the input and both initially contain the integers a and b, respectively. Assume that $v0 is used for the output.

```
add $t0, $zero, $zero

loop:
  beq $a1, $zero, finish
  add $t0, $t0, $a0
  sub $a1, $a1, 1
  j loop

finish:
  addi $t0, $t0, 100
  add $v0, $t0, $zero
```

2.30 [12] <§§2.3, 2.6, 2.9> The following code fragment processes two arrays and produces an important value in register $v0. Assume that each array consists of 2500 words indexed 0 through 2499, that the base addresses of the arrays are stored in $a0 and $a1 respectively, and that their sizes (2500) are stored in $a2 and $a3, respectively. Add comments to the code and describe in one sentence what this code does. Specifically, what will be returned in $v0?

```
sll $a2, $a2, 2
sll $a3, $a3, 2
add $v0, $zero, $zero
add $t0, $zero, $zero
outer: add $t4, $a0, $t0
```
lw $t4, 0($t4)
add $t1, $zero, $zero

inner:    add $t3, $a1, $t1
lw $t3, 0($t3)
br $t3, $t4, skip
addi $v0, $v0, 1

skip:     addi $t1, $t1, 4
bne $t1, $a3, inner
addi $t0, $t0, 4
bne $t0, $a2, outer

2.31 [10] Assume that the code from Exercise 2.30 is run on a machine with a 2 GHz clock that requires the following number of cycles for each instruction:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>add, addi, slt</td>
<td>1</td>
</tr>
<tr>
<td>lw, bne</td>
<td>2</td>
</tr>
</tbody>
</table>

In the worst case, how many seconds will it take to execute this code?

2.32 [5] Show the single MIPS instruction or minimal sequence of instructions for this C statement:

```
b = 25 / a;
```

Assume that a corresponds to register $t0 and b corresponds to register $t1.

2.33 [10] For More Practice: Translating from C to MIPS

2.34 [10] The following program tries to copy words from the address in register $a0 to the address in register $a1, counting the number of words copied in register $v0. The program stops copying when it finds a word equal to 0. You do not have to preserve the contents of registers $v1, $a0, and $a1. This terminating word should be copied but not counted.

```
addi $v0, $zero, 0 # Initialize count
loop: lw $v1, 0($a0)  # Read next word from source
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
```

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

There are multiple bugs in this MIPS program; fix them and turn in a bug-free version. Like many of the exercises in this chapter, the easiest way to write MIPS programs is to use the simulator described in Appendix A.

2.35 [10] §§2.2, 2.3, 2.6, 2.9> For More Practice: Reverse Translation from MIPS to C

2.36 §§2.9> For More Practice: Translating from C to MIPS

2.37 [25] §§2.10> As discussed on page 107 (Section 2.10, "Assembler"), pseudoinstructions are not part of the MIPS instruction set but often appear in MIPS programs. For each pseudoinstruction in the following table, produce a minimal sequence of actual MIPS instructions to accomplish the same thing. You may need to use set for some of the sequences. In the following table, big refers to a specific number that requires 32 bits to represent and small to a number that can fit in 16 bits.

<table>
<thead>
<tr>
<th>Pseudoinstruction</th>
<th>What it accomplishes</th>
</tr>
</thead>
<tbody>
<tr>
<td>move $t1, $t2</td>
<td>$t1 = $t2</td>
</tr>
<tr>
<td>clear $t50</td>
<td>$t0 = 0</td>
</tr>
<tr>
<td>beq $t1, small, L</td>
<td>if ($t1 = small) go to L</td>
</tr>
<tr>
<td>beq $t2, big, L</td>
<td>if ($t2 = big) go to L</td>
</tr>
<tr>
<td>li $t3, small</td>
<td>$t3 = small</td>
</tr>
<tr>
<td>li $t3, big</td>
<td>$t3 = big</td>
</tr>
<tr>
<td>ble $t3, $t5, L</td>
<td>if ($t3 &lt;= $t5) go to L</td>
</tr>
<tr>
<td>bgt $t4, $t5, L</td>
<td>if ($t4 &gt; $t5) go to L</td>
</tr>
<tr>
<td>bge $t5, $t3, L</td>
<td>if ($t5 &gt;= $t3) go to L</td>
</tr>
<tr>
<td>add $t0, $t2, big</td>
<td>$t0 = $t2 + big</td>
</tr>
<tr>
<td>lw $t5, big($t2)</td>
<td>$t5 = Memory($t2 - offset)</td>
</tr>
</tbody>
</table>

2.38 [5] §§2.9, 2.10> Given your understanding of PC-relative addressing, explain why an assembler might have problems directly implementing the branch instruction in the following code sequence:

```
here:  beq $s0, $s2, there
 ...
there  add $s0, $s0, $s0
```

Show how the assembler might rewrite this code sequence to solve these problems.

2.39 §§2.10> For More Practice: MIPS Pseudoinstructions

2.40 §§2.10> For More Practice: Linking MIPS Code