Problem 1

In MIPS assembly (documented thoroughly in Appendix A), write an assembly language version of the following C code segment:

```c
for(i = 0; i < 98; i ++)
{  
}
```

Arrays A, B and C start at memory location A000\text{hex}, B000\text{hex}, and C000\text{hex} respectively. Try to reduce the total number of instructions and the number of expensive instructions such as multiplies.

Problem 2

[Taken from P&H] Implement the following C code in MIPS, assuming that `setarray` is the first function called:

```c
int i;

void setarray (int num) {
    int array[10];
    for (i=0; i<10; i++) {
        array[i] = compare(num, i);
    }
}

int compare(int a, int b) {
    if (sub(a, b) >= 0)
        return 1;
    else
        return 0;
}

int sub (int a, int b) {
    return a-b;
}
```

Be sure to handle the stack and frame pointers appropriately. The variable code font is allocated on the stack, and \textit{i} corresponds to $s0$. Draw the status of the stack before calling `setarray` and during each function call. Indicate the names of registers and variables stored on the stack and mark the location of $sp$ and $fp$. 

Problem 3

Pseudo-instructions are not part of the MIPS instruction set but often appear in MIPS programs. For each pseudo-instruction in the following table, produce a minimal sequence of actual MIPS instructions to accomplish the same thing. You may need to use $at 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 be expressed using 16 bits.

<table>
<thead>
<tr>
<th>Pseudo-instruction</th>
<th>What it accomplishes</th>
</tr>
</thead>
<tbody>
<tr>
<td>move $t5, $t3</td>
<td>$t5 = $t3</td>
</tr>
<tr>
<td>clear $t5</td>
<td>$t5 = 0</td>
</tr>
<tr>
<td>li $t5, small</td>
<td>$t5 = small</td>
</tr>
<tr>
<td>li $t5, big</td>
<td>$t5 = big</td>
</tr>
<tr>
<td>lw $t5, big($t3)</td>
<td>$t5 = Memory[$t3 + big]</td>
</tr>
<tr>
<td>addi $t5, big($t3)</td>
<td>$t5 = $t3 + big</td>
</tr>
<tr>
<td>beq $t5, small, L</td>
<td>if($t5 = small) go to L</td>
</tr>
<tr>
<td>beq $t5, big, L</td>
<td>if($t5 = big) go to L</td>
</tr>
<tr>
<td>ble $t5, $t3, L</td>
<td>if($t5 &lt;= $t3) go to L</td>
</tr>
<tr>
<td>bgt $t5, $t3, L</td>
<td>if($t5 &gt; $t3) go to L</td>
</tr>
<tr>
<td>bge $t5, $t3, L</td>
<td>if($t5 &gt;= $t3) go to L</td>
</tr>
</tbody>
</table>

Problem 4

[Taken from P&H] 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 $t1, $t2, there
...
there: add $t1, $t1, $t1

Problem 5

Assume the following instruction mix for a MIPS-like RISC instruction set: 10% stores, 30% loads, 15% branches, 35% integer arithmetic, 5% integer shift, and 5% integer multiply. Given that load instructions require 2 cycles, branches require 4 cycles, integer ALU and store instructions require one cycle, and integer multiplies require 10 cycles, compute the overall CPI.

Problem 6

A designer wants to improve the overall performance of a given machine with respect to a target benchmark suite and is considering an enhancement X that applies to 55% of the original dynamically-executed instructions, and speeds each of them up by a factor of 3. The designer’s manager has some concerns about the complexity and the cost-effectiveness of X and suggests that the designer should consider an alternative enhancement Y. Enhancement Y, if applied only to some (as yet unknown) fraction of the original dynamically-executed instructions, would make them only 75% faster. Determine what percentage of all dynamically-executed instructions should be optimized using enhancement Y in order to achieve the same overall speedup as obtained using enhancement X.