Homework Information: Some of the problems are probably too long to be done the night before the due date, so plan accordingly. Late homework will not be accepted. Feel free to get help from others, but the work you hand in should be your own.

Problem 1. “Compiler Appreciation”

Translate the following code fragments (written in C) to MIPS assembly language. Use the general approach shown in lecture (allocate variables into low, directly addressable, memory addresses). You don’t have to write optimized assembly language unless you are into that sort of thing. Just show the executable code—you can assume that the necessary storage allocation for variables and arrays has already been done (meaning that you can refer to them by their label). Furthermore, you can assume that all variables have been allocated into the lower 32K bytes of memory, and that all variables and arrays are C integers, i.e., 32-bit values.

Example:

\[
x = 0 \\
\text{for ( i = 0; i < 10; i++)} \\
\quad x = x + a[i]; \\
\text{for:} \\
\quad \text{sw } 0, x \\
\quad \text{sw } 0, i \\
\quad \text{lw } t0, x \\
\quad \text{lw } t1, i \\
\quad \text{sll } t2, t1, 2 \\
\quad \text{lw } t2, a(t2) \\
\quad \text{addu } t0, t0, t2 \\
\quad \text{sw } t0, x \\
\quad \text{addiu } t1, t1, 1 \\
\quad \text{sw } t1, i \\
\quad \text{slt } t2, t1, 10 \\
\quad \text{bne } t2, 0, for
\]

(A) \[y = -y - x;\]

(B) \[a[i] = a[i+1] + a[i-1];\]

(C) if (x > y)
   \[x = x - y;\]
   else
   \[x = y - x;\]

(D) while ((i & 1) == 0)
   \[i = i >> 1;\]

(E) for (i = 1; i < 10; i++)
   \[a[i] = i + i + 1;\]

(F) \[a[x] = a[a[x]];\]
Problem 2. “MIPS Calisthenics”

Write MIPS code fragments to perform the following simple tasks.

(A) Clear registers t0-t8
(B) Clear memory locations (words) 0x100 to 0x1fc, inclusive
(C) Swap the contents of registers $t0 and $t1
(D) Count how many memory locations (words) in the address range from 0x100 to 0x1fc have a contents of 0

Problem 3. “Stack Detective”

Consider the following recursive C function to compute the $n^{th}$ Fibonacci number.

```c
int fib(int n) {
    if (n < 2)
        return n;
    else
        return fib(n-1) + fib(n-2);
}
```

After compiling, the following assembly code is generated:

```assembly
fib:    addi  $sp,$sp,-12
L01:    sw    $ra,8($sp)
L02:    sw    $a0,4($sp)
L03:    slti  $t0,$a0,2
L04:    beq   $t0,$0,L07
L05:    add   $v0,$0,$a0
L06:    beq   $0,$0,L15
L07:    addi  $a0,$a0,-1
L08:    jal   fib
L09:    sw    $v0,($sp)
L10:    lw    $a0,4($sp)
L11:    addi  $a0,$a0,-2
L12:    jal   fib
L13:    lw    $t0,($sp)
L14:    addu  $v0,$v0,$t0
L15:    lw    $ra,8($sp)
L16:    addi  $sp,$sp,12
L17:    jr     $ra
```

a) Explain how each of the 3 words allocated on the stack are used? Could this number be reduced? If so, explain how, if not explain why.

b) Suppose that the statement labeled L09 was replaced with `add $a1,$0,$v0`, and the two statements labeled L13 and L14 were replaced with the single statement `add $v0,$v0,$a1`. Would the resulting fragment still work? Explain.

c) Write an iterative version of the fib() function based on the following Fibonacci code fragment:
int x, y;

main() {
    x = 0;
    y = 1;
    while (y < 100) {
        int t = x;
        x = y;
        y = t + y;
    }
}

d) Discuss the differences between your iterative fib() implementation and the given recursive one. Which is faster? Shorter? Uses less memory? Easier to understand?

Suppose that at some point during the execution of the given recursive fib() function the computer is interrupted and the stack is examined and found to contain the following:

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Memory Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7fffefec</td>
<td>0x00380008</td>
</tr>
<tr>
<td>0x7fffe8e8</td>
<td>0x00030002</td>
</tr>
<tr>
<td>0x7fffe4e4</td>
<td>0x00000000f1</td>
</tr>
<tr>
<td>0x7fffe0e0</td>
<td>0x0040007c</td>
</tr>
<tr>
<td>0x7fffff2dc</td>
<td>0x00000007</td>
</tr>
<tr>
<td>0x7fffff2d8</td>
<td>0x00400048</td>
</tr>
<tr>
<td>0x7fffff2d0</td>
<td>0x00000006</td>
</tr>
<tr>
<td>0x7fffff2cc</td>
<td>0x00000005</td>
</tr>
<tr>
<td>0x7fffff2c8</td>
<td>0x000400048</td>
</tr>
<tr>
<td>0x7fffff2bc</td>
<td>0x0000000005</td>
</tr>
<tr>
<td>0x7fffff2b8</td>
<td>0x00000004</td>
</tr>
<tr>
<td>0x7fffff2b4</td>
<td>0x00400048</td>
</tr>
<tr>
<td>0x7fffff2b0</td>
<td>0x000400048</td>
</tr>
<tr>
<td>0x7fffff2ac</td>
<td>0x00000003</td>
</tr>
<tr>
<td>0x7fffff2a8</td>
<td>0x00000002</td>
</tr>
<tr>
<td>0x7fffff2a4</td>
<td>0x00400048</td>
</tr>
<tr>
<td>0x7fffff2a0</td>
<td>0x00000001</td>
</tr>
<tr>
<td>0x7fffff29c</td>
<td>0x00000001</td>
</tr>
<tr>
<td>0x7fffff298</td>
<td>0x00400058</td>
</tr>
<tr>
<td>0x7fffff29a</td>
<td>0x00000001</td>
</tr>
<tr>
<td>$sp @ 0x7fffff29c</td>
<td>0x5c4ee709</td>
</tr>
</tbody>
</table>

If you use the MIPS simulator/assembler as an aid in answering the following questions (which might be a good idea, though it is not necessary), you need to be aware of the following caveat. The simulator assumes that all of memory, outside the loaded .text and .data segments is filled with zeros. In reality, this is usually not the case. Upon power up, memory locations are filled
with apparently random values, and over the lifetime of a program the uninitialized values on the stack reflect the activation records of previously called procedures. Therefore, you need to consider that some of the values shown in the above stack dump may reflect uninitialized memory locations.

e) At what memory address can the function `fib()` be found?

f) What argument (value of `n`) was passed to the originating call?

g) What is the label of the last executed instruction before the machine was interrupted?

h) What would have been the lowest memory location referenced by the stack pointer during this particular invocation?