Allowable Instructions

When writing MIPS assembly, the only instructions that you are allowed to use (so far) are:

- `add`, `addi`, `sub`, `addu`, `addiu`
- `and`, `andi`, `or`, `ori`, `xor`, `xori`, `nor`
- `lui`
- `beq`, `bne`, `j`
- `jal`, `jr`
- `slt`, `slti`
- `sll`, `sra`, `srl`
- `lw`, `lh`, `lb`, `sw`, `sh`, `sb`
- `la`
- `syscall`
- `mult`, `div`, `mfhi`, `mflo`

While MIPS has many other useful instructions (and the assembler recognizes many pseudo-instructions), do not use them! We want you to learn the **fundamentals** of how assembly language works - you can use fancy tricks after this class is over.

<table>
<thead>
<tr>
<th>Question</th>
<th>Points</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short Answer</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>Calling Functions</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>ALU Simulation</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>Multiple Function Calls</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>Tracking a Stack</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>MIPS Assembly</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td><strong>Total:</strong></td>
<td><strong>100</strong></td>
<td></td>
</tr>
</tbody>
</table>
1. For each question below, give a short answer - a few words or symbols, maybe a sentence or two.

(a) (4 points) Give the name of the four key parts of code that surround a function call:
   In the subroutine, at the very beginning:
   **Solution:** Prologue
   In the subroutine, as it is ending:
   **Solution:** Epilogue
   In the caller, before the call:
   **Solution:** Startup code
   In the caller, after the call:
   **Solution:** Cleanup code

(b) (5 points) Our Ripple Carry Adder gives correct answers. Why then do we want to replace it with a newer, more advanced version?
   **Solution:** It’s too slow, because the calculation has to proceed through all of the ALU Element.

(c) (5 points) The following registers have special uses in MIPS function calls. Give a very short explanation about what each is used for:
   \$v0 $a1
   **Solution:** \$v0 holds the return value (if any).
   \$a1 holds the second parameter (if there are that many).

(d) (10 points) For each of the instructions below, give the proper control bits for the ALU. You may consult the diagram of the ALU element at the end of this exam.
   If the instruction does not use the ALU output, then say that the Operation is “ANY”. Likewise, if the instruction doesn’t care what the bNegate setting is, then say that it is “ANY”.

<table>
<thead>
<tr>
<th>bNegate</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td></td>
</tr>
<tr>
<td>sw</td>
<td></td>
</tr>
<tr>
<td>beq</td>
<td></td>
</tr>
<tr>
<td>j</td>
<td></td>
</tr>
<tr>
<td>slt</td>
<td></td>
</tr>
</tbody>
</table>

**Solution:**
Either mnemonics (AND, OR, ADD, LESS) or integers will be accepted for the Operation.

<table>
<thead>
<tr>
<th>bNegate</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>0</td>
</tr>
<tr>
<td>sw</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td>1</td>
</tr>
<tr>
<td>j</td>
<td>any</td>
</tr>
<tr>
<td>slt</td>
<td>1</td>
</tr>
</tbody>
</table>
2. For each snippet below, give the code to implement this function call. When I use a variable name like \( s_0 \), then assume that this is a register that you need to pass as a parameter.

Assume that you need to save all of the \( sX \) registers, but none of the \( tX \) registers.
You do not need to read any return value (for this set of questions).

(a) (4 points) \texttt{postmodernism}(s0, 10, s1)

Solution:

```
add $a0, $s0,$zero
addi $a1, $zero,10
add $a2, $s1,$zero
jal postmodernism
```

(b) (6 points) \texttt{truth}(1, 2, 3, 4, s7)

Solution:

```
addi $a0, $zero,1
addi $a1, $zero,2
addi $a2, $zero,3
addi $a3, $zero,4
sw $s7, -4($sp)
jal truth
```

3. In this problem, simulate a 4-bit ALU. Given the inputs

\[ a = 1100 \]
\[ b = 1101 \]

give each of the 4-bit values below.

(a) (4 points) The 4 AND bits. (Assume that \( bNegate=0 \).)

Solution: 1100

GRADING NOTE: All parts of this question will be graded as follows:

- All bits correct - 4 pts
- One bit incorrect - 3 pts
- More than one bit incorrect - 0 pts

(b) (4 points) The 4 OR bits. (Assume that \( bNegate=0 \).)

Solution: 1101

(c) (4 points) The 4 ADD bits. (Assume that \( bNegate=1 \).)

Solution: 1111

(d) (4 points) The 4 LESS bits. (Assume that \( bNegate=1 \).)
4. (20 points) Convert each C snippet below to MIPS assembly. You may assume that every function that you call returns \texttt{int}; functions that take parameters always have \texttt{int} parameters.

Write just this code snippet; \textbf{do not} write an entire function!

Follow these rules:

- Write to \texttt{tX} registers only; do not modify any \texttt{sX} registers unless instructed.
- You may assume that no \texttt{tX} registers are in use before this code runs.
- If you use \texttt{tX} registers, then make sure to save them.

\begin{verbatim}
printf("%d", outer(inner()));
\end{verbatim}

Solution:

\begin{verbatim}
jal inner
add $a0, $v0,$zero
jal outer
add $a0, $v0,$zero
addi $v0, $zero,1
syscall
\end{verbatim}
5. (20 points) In this problem, \texttt{caller()} calls \texttt{sub()}. The first column shows the state of the stack while \texttt{caller()} is running, just before its startup code.

In the second column, show the state of the stack after the startup code in \texttt{caller()} has completed, but before the \texttt{jal} instruction. In the third column show the state of the stack after \texttt{sub()} has run its function prologue, but before it saved anything other than the prologue (not even aX registers). In the fourth column show the state of the stack after \texttt{sub()} has saved all necessary registers.

\textbf{Make sure to mark:}

- The positions of $\texttt{fp}$, $\texttt{sp}$
- All values which have been written to stack
  
  Use arg1, arg2, etc. for the various parameters. Use arg1 for the first parameter - which is stored in $\texttt{a0}$.

\textbf{Notes:}

- When this problem begins, \texttt{caller()} is using (and wants to preserve) the registers $\texttt{t1}$, $\texttt{t3}$, $\texttt{t6}$, $\texttt{s0}$, $\texttt{s1}$, $\texttt{s6}$.
- \texttt{sub()} takes 6 parameters. It will need to store the first and second on the stack.
- \texttt{sub()} will be using the following registers somewhere in its code: $\texttt{t0}$, $\texttt{t3}$, $\texttt{t4}$, $\texttt{t5}$, $\texttt{t6}$, $\texttt{s1}$, $\texttt{s5}$.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|}
\hline
$\texttt{fp}$ & stack at beginning & stack at beginning & stack at beginning \\
$\texttt{sp}$ & stack at beginning & stack at beginning & stack at beginning \\
\hline
\end{tabular}
\end{table}
Solution:

**NOTE:** I don’t care about the order in which \( tX, aX \) registers are saved, provided that the right registers are saved, in the correct (collective) location.

On the other hand, the \( aX \) registers, as well as \( $ra, $fp \) must be in precisely the correct locations.
6. (10 points) **DO THIS QUESTION LAST.** The other code questions are simpler but worth more.

Convert the following C snippet to MIPS assembly. (You may assume that all parameters are `int`, and that every function returns `int`.)

You may assume that the function is not yet using the registers `$s0-$s3`, nor any of the `tX` registers; you may use some or all of them to store variables or temporary values.

**You must not modify the registers `$s4-$s7`.**

```c
int count = 1;
int val = 12345;
while (check(val) != 0)
{
    val = update(val >> 1);
    count++;
}
report(count);
```

**Solution:**

```mips
addi $s0, $zero,1      # count=1
addi $s1, $zero,12345  # val =12345

LOOP:
    add $a0, $s1,$zero
    jal check
    beq $v0,$zero, DONE

    # srl also OK!
    sra $a0, $s1,1
    jal update
    add $s1, $v0,$zero

    addi $s0, $s0,1
    j LOOP

DONE:
    add $a0, $s0,$zero
    jal report
```