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

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>25</td>
<td></td>
</tr>
<tr>
<td>Carry Lookahead</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>Misc</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>Tracking a Stack</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>Total:</td>
<td>100</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) (5 points) The following registers have special uses in MIPS function calls. Give a very short explanation about what each is used for:
       $v0

       $a2

   (b) (5 points) In a Carry Lookahead Adder, explain how to calculate the generate and propagate bits from the various inputs:

   (c) (5 points) What is the bNegate input to the ALU used for?

   (d) (5 points) Suppose that you have a 32-bit ALU, using the Ripple Carry design. If you changed it to a 64-bit design, how would this affect the clock speed of your processor, and why?

   (e) (5 points) Give the general formula for a certain carry bit $c_{i+1}$, using as inputs only the input bits $a_i, b_i$ and the previous carry bit $c_i$.
       You may use a sum-of-products expression, or a formula that includes parentheses.
2. (a) (10 points) Consider a Carry Lookahead Adder. Give the formula for \( c_2 \) as a **sum of products**, using only generate and propagate bits \((g_i, p_i)\), and the carry-in to the entire ALU \((c_0)\).

**HINT:** Your answer must include four terms.

**Instructor’s Note:** I accidentally mentioned \( c_2 \) above, but gave a solution for \( c_3 \) below. And it is the solution for \( c_3 \) which has four terms! So we’ll allow solutions for either \( c_2 \) or \( c_3 \).

(b) (20 points) Given the following pair of 16-bit numbers, calculate the propagate and generate bits; then calculate the super-propagate and super-generate bits for each nibble. Finally, calculate the carry-in for each nibble; the carry-in to the entire ALU has been provided.

**NOTE:** You do not need show the formulas; just show the values you calculated.

\[
\begin{align*}
a & : \ 1110 \ 1001 \ 1010 \ 0110 \\
b & : \ 1100 \ 1000 \ 1100 \ 1101 \ c_0 = 0
\end{align*}
\]

Generate Bits:

Propagate Bits:

Super Generate Bits:

Super Propagate Bits:

**Nibble** Carry-In Bits:
3. (a) (10 points) Simulate the multiplication algorithm. Show each step; give the multiplicand, multiplier, and result at each step. Use one row for each step; you may stop when the multiplier reaches zero.

Give all numbers in decimal.

<table>
<thead>
<tr>
<th>Multiplicand</th>
<th>Multiplier</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>19</td>
<td>0</td>
</tr>
</tbody>
</table>

(b) (15 points) Convert the C snippet below to MIPS assembly. You may assume that every function that you call returns `int`; functions that take parameters always have `int` parameters.

Write just this code snippet; **do not** write an entire function!

Follow these rules:
- Use `tX` registers only; do not use any `sX` registers.
- You may assume that no `tX` registers are in use before this code runs.
- If you use `tX` registers, then make sure to save them.

```c
hasbro(mattel(-1), milton(10));
```
4. (20 points) In this problem, **adam**() calls **jamie**(). The first column shows the state of the stack while **adam**() is running, just before its startup code.

In the second column, show the state of the stack after the startup code in **adam**() has completed, but before the **jal** instruction. In the third column show the state of the stack after **jamie**() 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 **jamie**() has saved all necessary registers.

**Make sure to mark:**

- The positions of $fp$, $sp$
- 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 $a0$.

**Notes:**

- When this problem begins, **adam**() is using (and wants to preserve) the registers $t8$, $t7$, $t6$, $s5$, $s4$, $s3$.
- **jamie**() takes 3 parameters. It will need to store the third on the stack.
- **jamie**() will be using the following registers somewhere in its code: $t8$, $t7$, $t6$, $s5$, $s4$, $s3$. 

<table>
<thead>
<tr>
<th>stack at beginning</th>
<th>stack at beginning</th>
<th>stack at beginning</th>
<th>stack at beginning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$fp$</td>
<td>$sp$</td>
<td>$sp$</td>
<td>$sp$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>