WARNING
This test is double-sided. Make sure to notice the problems on the back of each page!

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:

$\text{Solution:}$ It holds the return value (if any).

$\text{Solution:}$ It holds the third parameter (if there are that many).

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

$\text{Solution:}$ The generate bit is the AND of the input bits for that column.
The propagate bit is the OR of the same two bits.

(c) (5 points) What is the $b\text{Negate}$ input to the ALU used for?

$\text{Solution:}$ It configures the adder to perform subtraction instead.

$\text{Instructor’s Note:}$
(I’ll also allow more mechanical explanations, about how the wires are connected.)

(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?

$\text{Solution:}$ The clock speed would be cut roughly in half, because the path length through the ALU would be twice as long.

(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.

$\text{Solution:}$ $c_{i+1} = a_i b_i + (a_i + b_i) c_i$
$\quad \text{or} \quad c_{i+1} = a_i b_i + a_i c_i + b_i c_i$
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 \( c_3 \).

**Solution:**
\[
\begin{align*}
  c_3 &= g_2 + p_2g_1 + p_2p_1g_0 + p_2p_1p_0c_0 \\
  c_2 &= g_1 + p_1g_0 + p_1p_0c_0
\end{align*}
\]

(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.

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

Generate Bits:

Propagate Bits:

Super Generate Bits:

Super Propagate Bits:

Nibble Carry-In Bits:

**Solution:**
\[
\begin{align*}
  a: & \quad 1110 \ 1001 \ 1010 \ 0110 \\
  b: & \quad 1100 \ 1000 \ 1100 \ 1101 \\
  \text{generate:} & \quad 1100 \ 1000 \ 1000 \ 0100 \\
  \text{propagate:} & \quad 1110 \ 1001 \ 1110 \ 1111 \\
  \text{super-generate:} & \quad 1 \ 1 \ 1 \ 1 \\
  \text{super-propagate:} & \quad 0 \ 0 \ 0 \ 1 \\
  \text{carry-in:} & \quad 1 \ 1 \ 1 \ 0
\end{align*}
\]
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>
<tr>
<td>30</td>
<td>9</td>
<td>15</td>
</tr>
<tr>
<td>60</td>
<td>4</td>
<td>45</td>
</tr>
<tr>
<td>120</td>
<td>2</td>
<td>45</td>
</tr>
<tr>
<td>240</td>
<td>1</td>
<td>45</td>
</tr>
<tr>
<td>480</td>
<td>0</td>
<td>285</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));
```

Solution:
# mattel(-1)
addi $a0, $zero,-1
jal mattel

# save the retval onto the stack. Note that it
# is *NOT* necessary to bounce it through a tX
# register first - although that's allowed if
# you want.
addiu $sp, $sp,-4
sw $v0, 0($sp)

# milton(10)
addi $a0, $zero,10
jal milton

# restore the retval from mattel() into a0. Again,
# we could bounce it through a tX register if you
# want, but it's not necessary.
lw $a0, 0($sp)
addiu $sp, $sp,4

# move the retval from milton() into a1, as it's
# the second arg.
add $a1, $v0,$zero
jal hasbro
4. (20 points) In this problem, \texttt{adam()} calls \texttt{jamie()}. The first column shows the state of the stack while \texttt{adam()} is running, just before its startup code.

In the second column, show the state of the stack after the startup code in \texttt{adam()} has completed, but before the \texttt{jal} instruction. In the third column show the state of the stack after \texttt{jamie()} has run its function prologue, but before it saved anything other than the prologue (not even \texttt{aX} registers). In the fourth column show the state of the stack after \texttt{jamie()} 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{adam()} is using (and wants to preserve) the registers $\texttt{t8}$, $\texttt{t7}$, $\texttt{t6}$, $\texttt{s5}$, $\texttt{s4}$, $\texttt{s3}$.
- \texttt{jamie()} takes 3 parameters. It will need to store the third on the stack.
- \texttt{jamie()} will be using the following registers somewhere in its code: $\texttt{t8}$, $\texttt{t7}$, $\texttt{t6}$, $\texttt{s5}$, $\texttt{s4}$, $\texttt{s3}$.

\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}
**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.