CSc 252: Computer Organization  
Fall 18 (Lewis)  

Test 3  
Thu 4 Oct 2018

Name: ________________________________ NetID: ____________________

Person to your left: __________________ Person to your right: __________________

Allowable Instructions

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

- `add, addi, sub`
- `beq, bne, j`
- `slt, slti`
- `and, andi, or, ori, nor, nori, xor, xori`
- `sll, srl, sra`
- `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>The First Page</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>Inequality Checks</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>MIPS Code 1</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>MIPS Code 2</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>Reading MIPS</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>Total:</td>
<td>100</td>
<td></td>
</tr>
</tbody>
</table>
1. (a) (5 points) When we declare a string in MIPS, we don't need a length variable. Why not? How do we find the end of the string?

(b) (5 points) We showed, in class, that a MUX can be implemented using a simple sum-of-products. Give the sum-of-products to implement a 2-input MUX.
   - Use the variable C to represent the control bit.
   - Use the variable X as the 0 input to the MUX.
   - Use the variable Y as the 1 input to the MUX.

(c) (2 points) The I-format instructions (like ADDI) can't hold a full 32-bits of data in their constant field - but the ALU requires 32 bit inputs. How is the constant field converted to something large enough?

(d) (3 points) How many bits can an ADDI instruction store in its constant?

(e) (5 points) Suppose that you have the following array:

   ```c
   int arr[100];
   ```

   Assume that the register $t8$ has been initialized to hold an index into the array. Write a code snippet which does three things:
   - Gets the address of the array.
   - Reads element [3] into register $s0$.
   - Reads element [t8] into register $s1$. (Don’t error-check the index, just use it.)
2. Suppose that you have two variables, stored in the registers $s0$, $s1$. Implement the conditional branches shown below. Use no more than 2 instructions per part.

(a) (2 points)
   
   if (s0 != s1)
   goto GO_HERE;

(b) (6 points)
   
   if (s0 > s1)
   goto GO_HERE;

(c) (6 points)
   
   if (s0 <= s1)
   goto GO_HERE;

(d) (6 points)
   
   if (s0 >= s1)
   goto GO_HERE;
3. (25 points) Convert the following C code into MIPS assembly language.

- Use $sX$ registers for all variables that have names in the C program.
- Use $tX$ registers for all temporary values.

```c
int limit = ... ;  // s6 - this is set by previous code
int val = 1;       // choose a register to hold this
int count = 0;     // and also this

while (val < limit)
{
    val = val * 3;
    count++;
}

printf("%d", count);
```

NOTE: You’re allowed to use `mult`, if you know how to use it correctly. But doing add/shift is probably easier!
4. (15 points) (Same instructions as the previous page.)

```c
int asdf = ... ;    // s0 - this is set by previous code
int jkl = ... ;     // s1 - this is set by previous code
int qwerty = ... ;  // s2 - this is set by previous code
int uiop = ... ;    // s3 - this is set by previous code

int tmp;            // allocate a register for this
if (asdf == jkl || qwerty == uiop)
    tmp = 1;
else
    tmp = asdf+qwerty;
printf("%d", tmp);
```
5. Assume that we have the following variables, stored in the following registers:

\[
\begin{align*}
$s2 & \text{ elway} \\
$s4 & \text{ montana} \\
$s6 & \text{ kosar} \\
$s7 & \text{ brady}
\end{align*}
\]

Read the following code snippets. Then write one or two lines of code (C or pseudocode) for each, which indicates what the assembly language does. When the MIPS writes to a register, you can use names like \( sX \) to represent the variable that it is writing to, like this:

\[
\text{s3 }\text{=}\text{ foo+bar;}
\]

(a) (5 points) \quad \text{add } \text{s0, }\text{zero,}\text{s2}

(b) (5 points)
\[
\begin{align*}
\text{add } & \text{t0, }\text{s2,}\text{s4} \\
\text{add } & \text{t1, }\text{s6,}\text{s7} \\
\text{add } & \text{t2, }\text{t0,}\text{t1} \\
\text{sub } & \text{s1, zero,}\text{t2}
\end{align*}
\]

(c) (5 points)
\[
\begin{align*}
\text{addi } & \text{v0, zero,1} \\
\text{add } & \text{a0, zero,}\text{s7} \\
\text{syscall} \\
\text{addi } & \text{v0, zero,11} \\
\text{addi } & \text{a0, zero,}'\text{\newline}' \\
\text{syscall}
\end{align*}
\]

(d) (5 points)
\[
\begin{align*}
\text{add } & \text{t8, }\text{s2,}\text{s4} \\
\text{slt } & \text{t7, t8,}\text{s6} \\
\text{beq } & \text{t7,zero,}\text{AFTER} \\
\text{sub } & \text{s0, s6,}\text{s7} \\
\text{AFTER:}
\end{align*}
\]