Name: ____________________________________________________________

Person to your left: _______________________________  Person to your right: __________________________________

<table>
<thead>
<tr>
<th>Question</th>
<th>Points</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary Negation</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>Zeroing a Register</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>la, lw</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>Fixing a Shift</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>Constructing a Mask</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>Writing MIPS</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>Encoding an Instruction</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>Total:</td>
<td>100</td>
<td></td>
</tr>
</tbody>
</table>
Allowed MIPS Instructions
For all problems in this Test, you may use the following MIPS instructions and no others:

- `add`, `addi`, `sub`
- `sll`, `srl`, `sra`, `and`, `andi`, `or`, `ori`, `nor`, `xor`, `xori`
- `beq`, `bne`, `j`
- `slt`, `slti`
- `lw`, `lh`, `lb`, `sw`, `sh`, `sb`
- `la`
- `syscall`

1. (5 points) MIPS does not have a NOT instruction, and yet it is possible to perform binary negation in a single instruction. Consult the list of allowable instructions above. Using one of those instructions, write a single instruction which will take the value in `$s1`, perform bitwise negation, and store the result into `$t3`.

**Solution:**

```assembly
nor $t3, $s1, $s1
```

**Alternate solution:**

```assembly
nor $t3, $s1, $zero
```

2. There are many ways to set a register to zero in MIPS. For each part below, give a single instruction that sets the register `$s7` to zero.

(a) (5 points) Use only the `add` or `sub` instructions, but do not use the register `$zero`.

**Solution:**

```assembly
sub $s7, $s0, $s0
```

**NOTE:** You could use any input register. But you have to use the same register in both inputs for it to work, of course!

(b) (5 points) Use only the `xor` instruction. Do not use the register `$zero`.

**Solution:**

```assembly
xor $s7, $s0, $s0
```

**NOTE:** You could use any input register. But you have to use the same register in both inputs for it to work, of course!

(c) (5 points) Use the `addi` instruction; you may use the register `$zero`.

**Solution:**

```assembly
addi $s7, $zero, 0
```

3. (15 points) Explain the difference between the `la` and the `lw` instructions. What is contained in the register after each one? Then give the correct sequence of instructions to load the word at `tempVar` into register `$s5`. (Do not modify any other register while you load it.)

**Solution:**

`la` loads the address of a label (also OK: address of variable, word, etc.) into a register. It does not load the value itself. On the other hand, `lw` loads a value from memory, when the address is already known.
la $s5, tempVar
lw $s5, 0($s5)
4. (5 points) Suppose that we want to take bits 20-31 in $t4$, shift them to the bottom of the register (with zeroes above), and then store the result into $t5$. Modify the (incorrect) instruction below so that it works properly, and explain the difference between your instruction and the original.

```
    sra  $t5, $t4, 20
```

**Solution:**
```
    srl  $t5, $t4, 20
```

The **sra** instruction sign-extends the value, meaning that sometimes it places 1’s in the high bits. The **srl** instruction always zero-pads the instruction.

5. (20 points) Using only the allowed instructions (see Page 2), construct a mask which has all of the even bits turned on: that is, bits 0, 2, 4, 6, 8...30. Place the mask in register $t3$; do not modify any other registers. You should be able to do this in three instructions.

**Solution:**
```
    ori  $t3, $zero, 0x5555  # t3 = 0x0000 5555, addi is also OK
    sll  $t3, $t3, 16       # t3 = 0x5555 0000
    ori  $t3, $t3, 0x5555    # t3 = 0x5555 5555, addi is also OK
```

**NOTE:** **addi** works here because, in both cases, the high bit of the immediate field is 0. But **ori** is used more commonly, because it works even when the high bit is 1.

Interesting Trivia: MIPS has another instruction, which you are not allowed to use, which can do this in two instructions. We may cover this instruction later - if we have time.
6. For the following questions, consult the MIPS .data section on the last page of this Test, and then write MIPS assembly to perform the required action. Assume that each of your answers is independent of the others. **Pay attention to types.**

In these questions, you may modify any 't' register ($t0-$t9), but you must not modify any 's' register ($s0-$s7), unless specifically instructed.


Solution:

```assembly
la $t0, arr1
lw $t1, 8($t0)  # an addi/lw sequence is also acceptable
lw $t2, 12($t0) # an addi/lw sequence is also acceptable
add $s2, $t1, $t2
```

(b) (10 points) Assume that $s2 contains the variable i. Load the address of arr2[i] into $s3.

Solution:

```assembly
la $s3, arr2 # s3 = &arr2[0]
sll $t0, $s2, 1 # t0 = 2*i
add $s3, $s3, $t0 # s3 = &arr2[i]
```
7. (20 points) Encode the following MIPS instruction into a word, then convert the word to hexadecimal. Use the next two pages of this Test to help you.

    lw  $s0, -1($t0)

Reminder: In the lw instruction, rs is the register to store, and rt is the base address.

UPDATE: The instruction above was backwards. The correct statement is that rt is the register to store, and rs is the base address. In the solution below, the original (incorrect) answer is in black, and the corrected answer is in red.

Solution: lw is an I-format instruction, with opcode hex 0x23 = binary 10 0011.

rs: s0 = 16 decimal = 10000 binary
rs: t0 = 8 decimal = 01000 binary
rt: t0 = 8 decimal = 01000 binary
rt: s0 = 16 decimal = 10000 binary
immediate field: -1 is all 1’s in binary

Opcode: 1000 11
rs: 10 000
rt: 0 1000
immed: 1111 1111 1111 1111
Binary: 1000 1110 0000 1000 1111 1111 1111 1111
Hex: 8 e 0 8 f f f f

rs: 01 000
rt: 1 0000
Binary: 1000 1101 0000 1000 1111 1111 1111 1111
Hex: 8 d 1 0 f f f f
the stack pointer. The executing procedure uses the frame pointer to quickly access values in its stack frame. For example, an argument in the stack frame can be loaded into register $v0 with the instruction

\[ \text{lw } \$v0, 0(\$fp) \]

<table>
<thead>
<tr>
<th>Register name</th>
<th>Number</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>constant 0</td>
</tr>
<tr>
<td>$at</td>
<td>1</td>
<td>reserved for assembler</td>
</tr>
<tr>
<td>$v0</td>
<td>2</td>
<td>expression evaluation and results of a function</td>
</tr>
<tr>
<td>$v1</td>
<td>3</td>
<td>expression evaluation and results of a function</td>
</tr>
<tr>
<td>$a0</td>
<td>4</td>
<td>argument 1</td>
</tr>
<tr>
<td>$a1</td>
<td>5</td>
<td>argument 2</td>
</tr>
<tr>
<td>$a2</td>
<td>6</td>
<td>argument 3</td>
</tr>
<tr>
<td>$a3</td>
<td>7</td>
<td>argument 4</td>
</tr>
<tr>
<td>$t0</td>
<td>8</td>
<td>temporary (not preserved across call)</td>
</tr>
<tr>
<td>$t1</td>
<td>9</td>
<td>temporary (not preserved across call)</td>
</tr>
<tr>
<td>$t2</td>
<td>10</td>
<td>temporary (not preserved across call)</td>
</tr>
<tr>
<td>$t3</td>
<td>11</td>
<td>temporary (not preserved across call)</td>
</tr>
<tr>
<td>$t4</td>
<td>12</td>
<td>temporary (not preserved across call)</td>
</tr>
<tr>
<td>$t5</td>
<td>13</td>
<td>temporary (not preserved across call)</td>
</tr>
<tr>
<td>$t6</td>
<td>14</td>
<td>temporary (not preserved across call)</td>
</tr>
<tr>
<td>$t7</td>
<td>15</td>
<td>temporary (not preserved across call)</td>
</tr>
<tr>
<td>$s0</td>
<td>16</td>
<td>saved temporary (preserved across call)</td>
</tr>
<tr>
<td>$s1</td>
<td>17</td>
<td>saved temporary (preserved across call)</td>
</tr>
<tr>
<td>$s2</td>
<td>18</td>
<td>saved temporary (preserved across call)</td>
</tr>
<tr>
<td>$s3</td>
<td>19</td>
<td>saved temporary (preserved across call)</td>
</tr>
<tr>
<td>$s4</td>
<td>20</td>
<td>saved temporary (preserved across call)</td>
</tr>
<tr>
<td>$s5</td>
<td>21</td>
<td>saved temporary (preserved across call)</td>
</tr>
<tr>
<td>$s6</td>
<td>22</td>
<td>saved temporary (preserved across call)</td>
</tr>
<tr>
<td>$s7</td>
<td>23</td>
<td>saved temporary (preserved across call)</td>
</tr>
<tr>
<td>$t8</td>
<td>24</td>
<td>temporary (not preserved across call)</td>
</tr>
<tr>
<td>$t9</td>
<td>25</td>
<td>temporary (not preserved across call)</td>
</tr>
<tr>
<td>$k0</td>
<td>26</td>
<td>reserved for OS kernel</td>
</tr>
<tr>
<td>$k1</td>
<td>27</td>
<td>reserved for OS kernel</td>
</tr>
<tr>
<td>$gp</td>
<td>28</td>
<td>pointer to global area</td>
</tr>
<tr>
<td>$sp</td>
<td>29</td>
<td>stack pointer</td>
</tr>
<tr>
<td>$fp</td>
<td>30</td>
<td>frame pointer</td>
</tr>
<tr>
<td>$ra</td>
<td>31</td>
<td>return address (used by function call)</td>
</tr>
</tbody>
</table>

FIGURE A.6.1  MIPS registers and usage convention.
FIGURE A.10.2 MIPS opcode map. The values of each field are specified elsewhere: if \( z = 1 \) or \( z = 2 \), then the operations are in the last field with \( \text{f} \); if \( z = 1 \), then the operations are in the last field with \( \text{d} \); if \( rs = 17 \) and \( op = 17 \). The second field (rs) uses \( 17, 18, \) and 19. These operations are determined by other fields, identified by pointers. The last field (funct) uses \( 0 \) to \( 7 \).
This gives the `.data` section of a program; in some of the questions above, you will write code to load or store these variables.

Note that the values of the variables are not given!

```
.data

arr1:
    .word  ???
    .word  ???
    .word  ???
    .word  ???

arr2:
    .half  ???
    .half  ???
    .half  ???

foo:      .word  ???
bar:      .half  ???
baz:      .word  ???
```