1. (30 pts.) Using the single cycle MIPS implementation (mips-uarch.pdf, slide 14) as a reference, indicate the value on the wires listed in the table below for each of the three instructions. Use use as specific a descriptor as possible. For example the value in memory at address A could be indicated with “Mem[A]” or the value of register $ra might be “Reg[ra]”.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>SrcA</th>
<th>SrcB</th>
<th>ALUControl</th>
<th>WriteReg</th>
<th>SignImm</th>
<th>PCBBranch</th>
<th>WriteData</th>
<th>PCSrc</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>beq $0, $0, 8</td>
<td>0</td>
<td>0</td>
<td>110</td>
<td>000000</td>
<td>0x0000</td>
<td>PCPlus4</td>
<td>00000008</td>
<td>1</td>
<td>unable to determine</td>
</tr>
<tr>
<td>lw $s1, 12($sp)</td>
<td>$Reg[sp]</td>
<td>0x0000000C</td>
<td>$Reg[t3]</td>
<td></td>
<td>0x00004020</td>
<td></td>
<td></td>
<td>0</td>
<td>$Reg[t0] + $Reg[t3]</td>
</tr>
<tr>
<td>add $t0, $t3, $t0</td>
<td></td>
<td></td>
<td>010</td>
<td>010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2. Why are the following considered pseudo instructions

All of these answers have some aspects in common – mainly that researchers and designers have tested a variety of pseudo instructions in scenarios where they incorporate them as actual instructions and create the support necessary to implement them. They run tests on that modified processor, and compare results with the original design. If they see a significant improvement in performance with very little penalty, they would accept it into the ISA. Otherwise, they would not. The following answers build up on it.

- Branch greater than or equal to (bge) – while implementing such a branch is simple (e.g. take the sign bit of the result, see if it is equal to 1, make sure there was no overflow, and that’ll tell you to branch). However, keep in mind that such additional logic could have slowed execution time, as it has to go through that comparator, then an OR gate that ORs all possible branch conditions, etc. This slowdown would have brought the processor to speeds deemed undesirable to the ISA architects, who figured it would be easier to set it as a pseudo-instruction. On top of that, its another instruction they need to take into account, and its additional hardware (2 gates might seem low, but it could make a difference)
- Unconditional Branch (b) – no need for an additional instruction while beq $0, $0, X does the same thing.
- Multiplication (mul) – R-instructions support only 1 32-bit result, where as multiplication is liable to take on 64-bit answers
3. Speeds ups or slow downs
   a. \( T_c = 30 + 250 + 75 + 400 + 250 + 25 + 10 = 1040 \) ps, 0.92 GHz, 0.89x as fast, 1.12x slower
   b. \( T_c = 30 + 250 + 150 + 200 + 250 + 25 + 100 = 1005 \) ps, 1.00 GHz, 0.92x as fast, 1.08x slower
   c. \( T_c = 30 + 250 + 150 + 200 + 250 + 5 + 20 = 905 \) ps, 1.10 GHz, 1.02x as fast

4. Multiple solutions exist. The way to go about this is figuring how each of the provided instructions are affected by this, and how Processor would branch (i.e. under what conditions would the processor branch, regardless of what the intentions are of the instruction being executed). Remember that at all times, PCBranch IS being calculated – whether or not it is used depends on the instruction being executed, but regardless it is calculated. So, if you are on an add instruction, and the result is 0, then it will branch, even though it is not what you meant. In the picture below, the red wire is the one that is stuck. You need to figure out how to detect that. Notice that you will branch only of the zero flag of the ALU is also set. So you know that any instruction that results in a RESULT=0 will cause a branch to occur.

The following is one such program:

```assembly
Check_branch_stuck:    addi $t0, $0, -2
                       addi $t0, $t0, 2
                       addi $v0, $0, 0
                       jr $ra
                       addi $v0, $0, 1
                       jr $ra
```

Why does this work? In the first case, adding -2 to 0 results in -2, which is not 0, and therefore the zero flag is not set (it will not jump). In the second instruction, you add 1 to the t0 register, which already equals -2. Thus, the result is 0, setting the zero flag high (to 1). Now, if the branch wire is indeed stuck, it will try to branch to \( \text{PC+4+(immediate<<2)} = \text{PC+4+8 = PC+12} \) (i.e. it'll skip two instructions and go to the
next). If it is not stuck, it'll just go to the next instruction. We can use that information to set the appropriate output in $v0