Show your work for each problem; we are more interested in how you get the answer than whether you get the right answer.
1. (20 pts.) In MIPS assembly, implement the standard C function `strspn`:

```c
size_t strspn(const char *s, const char *accept)
```

This returns the length (in bytes) of the initial segment of the string `s` that consists entirely of bytes in `accept`.

Start from the `strspn.s` template on the class website; use the SPIM simulator.

Your function must obey MIPS calling conventions.

Turn in your solution on paper with evidence that it works. Add some test cases.

On the supplied test harness, your code should print

```
strspan("Hello World!", "Hloe") = 5
strspan("Hello World!", "H Wdelor") = 11
strspan("Hello World!", "!H Wdelor") = 12
strspan("Hello World!", "HHHHHH1llllooo") = 1
strspan("Hello World!", "HHHHHHooooeeeelll") = 5
strspan("Hello World!", "HHHHHHooooeeeelll Wrd!") = 12
strspan("", "Hello World!") = 0
strspan("Hello World!", ") = 0
strspan("Hello World!", "Hello World!") = 12
strspan("Hello World!", "HelWrd!") = 4
strspan("", ") = 0
```
strspan:
    move $t0, $a0 # Save start of string
.L1:
    lbu $t1, 0($t0) # Get character from string
    addiu $t0, $t0, 1 # Advance
    move $t2, $a1 # Start at beginning of accept string
.L2:
    lbu $t3, 0($t2) # Get character to accept
    addiu $t2, $t2, 1 # Advance
    beq $t3, $zero, .L3 # Not at end of accept: check next
    beq $t3, $t1, .L1 # Matching character found: advance
    b .L2
.L3:
    subu $v0, $t0, $a0 # How far did we go?
    subu $v0, $v0, 1 # Correct for post-increment
    jr $ra
2. (30 pts.) In MIPS assembly, implement an “eval” function that walks a tree that represents a Boolean expression and computes its meaning. Each tree node begins with a byte that indicates the node is an integer (leaf) or operator plus one or two pointers to their arguments. In C,

```c
int eval(struct expr *e)
{
    int left, right;
    if (e->op == 0) return e->pl.leaf;
    left = eval(e->pl.branch.left);
    if (e->op == '!') return ~left; /* bitwise NOT */
    right = eval(e->pl.branch.right);
    switch (e->op) {
        case '+': return left | right; /* bitwise OR */
        case '*': return left & right; /* bitwise AND */
        case '^': return left ^ right; /* bitwise XOR */
    }
    return 0;
}
```

Start from the `eval.s` template on the class website.
Your function must obey MIPS calling conventions. Use the stack to implement the recursion.
Implement your function in the SPIM simulator.
Turn in your solution on paper with evidence that it works. Add some test cases.

On the supplied test harness, your code should print

\[
\begin{align*}
1 &= 1 \\
0 &= 0 \\
4 &= 4 \\
(0+4) &= 4 \\
(0+1) &= 1 \\
(1+1) &= 1 \\
(1+4) &= 5 \\
(7*(1+1)) &= 1 \\
(2+3) &= 3 \\
((1+4)+(0+1)) &= 5 \\
!(3) &= -4 \\
(0^0) &= 0 \\
(0^1) &= 1 \\
(1^1) &= 0 \\
((0+4)^{(1+4)}) &= 1 \\
!(((0+4)^{(1+4)})) &= -2 
\end{align*}
\]
eval:
    lb  $t0, 0($a0)
    bne $t0, $0, dobranch
    lw  $v0, 4($a0)
    jr  $ra

dobranch:
    addiu $sp, $sp, -16
    sw  $ra, 0($sp)
    sw  $s0, 4($sp)
    sw  $s1, 8($sp)

    move $s1, $a0

    lw  $a0, 4($a0)
    jal eval
    move $s0, $v0

    lb  $t0, 0($s1)

    li  $t1, '!'  
    bne $t0, $t1, L0
    not $v0, $s0
    b  evalexit

L0:
    lw  $a0, 8($s1)
    jal eval

    lb  $t0, 0($s1)

    li  $t1, '+'
    bne $t0, $t1, L1
    or  $v0, $s0, $v0
    b  evalexit

L1:
    li  $t1, '*'
    bne $t0, $t1, L2
    and $v0, $s0, $v0
    b  evalexit

L2:
    li  $t1, '^'
    bne $t0, $t1, evalexit
    xor $v0, $s0, $v0

    evalexit:
    lw  $ra, 0($sp)
    lw  $s0, 4($sp)
    lw  $s1, 8($sp)
    addiu $sp, $sp, 16
    jr  $ra
3. (25 pts.) Extend the single-cycle MIPS processor to support the `xori` instruction (i-type, OP=001110).

<table>
<thead>
<tr>
<th>Inst.</th>
<th>OP</th>
<th>RegWrite</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>Branch</th>
<th>MemWrite</th>
<th>MemToReg</th>
<th>ALUOp</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>000000</td>
<td>1</td>
<td>1</td>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>lw</td>
<td>100011</td>
<td>1</td>
<td>0</td>
<td>01</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>00</td>
</tr>
<tr>
<td>sw</td>
<td>101011</td>
<td>0</td>
<td>-</td>
<td>01</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>00</td>
</tr>
<tr>
<td>beq</td>
<td>000100</td>
<td>0</td>
<td>-</td>
<td>00</td>
<td>1</td>
<td>0</td>
<td>-</td>
<td>01</td>
</tr>
<tr>
<td>addiu</td>
<td>001001</td>
<td>1</td>
<td>0</td>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>xori</td>
<td>001110</td>
<td>1</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>11</td>
</tr>
</tbody>
</table>
The ALU already almost performs the XOR operation as part of addition (each bit is the XOR of the two input bits and the XOR of the carry). So it’s easy to add XOR to the ALU by adding a “disable carry” input:

<table>
<thead>
<tr>
<th>F3</th>
<th>F2</th>
<th>F1</th>
<th>F0</th>
<th>Func.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>A &amp; B</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>A + B</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>A &amp; B̅</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>A − B</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>A &lt; B (slt)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>A ⊕ B</td>
</tr>
</tbody>
</table>
Finally, we need to extend the ALU Decoder to handle the new XOR operation:

<table>
<thead>
<tr>
<th>ALU Op</th>
<th>Funct</th>
<th>ALU Ctrl.</th>
<th>ALU Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>—</td>
<td>010</td>
<td>Add</td>
</tr>
<tr>
<td>01</td>
<td>—</td>
<td>0110</td>
<td>Subtract</td>
</tr>
<tr>
<td>10 100001</td>
<td>0010</td>
<td>Add</td>
<td></td>
</tr>
<tr>
<td>10 100011</td>
<td>0110</td>
<td>Subtract</td>
<td></td>
</tr>
<tr>
<td>10 100100</td>
<td>0000</td>
<td>AND</td>
<td></td>
</tr>
<tr>
<td>10 100101</td>
<td>0001</td>
<td>OR</td>
<td></td>
</tr>
<tr>
<td>10 101010</td>
<td>0111</td>
<td>Slt</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>—</td>
<td>1010</td>
<td>XOR</td>
</tr>
</tbody>
</table>
4. (10 pts.) Assuming the following dynamic instruction frequency for a program running on the single-cycle MIPS processor

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>addu</td>
<td>25%</td>
</tr>
<tr>
<td>addi</td>
<td>20%</td>
</tr>
<tr>
<td>beq</td>
<td>10%</td>
</tr>
<tr>
<td>lw</td>
<td>25%</td>
</tr>
<tr>
<td>sw</td>
<td>20%</td>
</tr>
</tbody>
</table>

(a) (5 pts.) In what fraction of all cycles is the data memory accessed (either read or written)?

Only for loads and stores, so 25% (lw) + 20% (sw) = 45%.

(b) (5 pts.) In what fraction of cycles is the sign extend circuit used?

- addi uses it for the immediate operand
- beq uses it to compute the PC-relative address
- lw uses it to compute the offset address
- sw uses it to compute the offset address

So, 20% + 10% + 25% + 20% = 75%.
5. (15 pts.) For each of the caches listed below, show how a 32-bit addresses breaks into tag, set index, and byte offset fields.

Cache A: 16384B, 4-way set-associative, 16B lines
128B per set, so 256 sets in cache

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

- tag (20 bits)
- set index (8 bits)
- byte offset (4 bits)

Cache B: 8192B, direct-mapped, 32B lines
32B per set, so 256 sets in cache

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

- tag (19 bits)
- set index (8 bits)
- byte offset (5 bits)