Problem 1 (Instruction Following) ★
Assume a RISC-V processor starts with the following values in the register file (and program counter):

<table>
<thead>
<tr>
<th>PC</th>
<th>x0</th>
<th>x1</th>
<th>x2</th>
<th>x3</th>
<th>x4</th>
<th>x5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0100</td>
<td>0x0000</td>
<td>0x0000</td>
<td>0x0000</td>
<td>0x0000</td>
<td>0x0000</td>
<td>0x0000</td>
</tr>
</tbody>
</table>

Fill in the state of the register file after each instruction is executed:
(assume instruction labelled jmplabel resides in memory at 0x0200)

addi x1, x0, 0xAA

<table>
<thead>
<tr>
<th>PC</th>
<th>x0</th>
<th>x1</th>
<th>x2</th>
<th>x3</th>
<th>x4</th>
<th>x5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ori x2, x1, 0x01

<table>
<thead>
<tr>
<th>PC</th>
<th>x0</th>
<th>x1</th>
<th>x2</th>
<th>x3</th>
<th>x4</th>
<th>x5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

srai x3, x2, 0x01

<table>
<thead>
<tr>
<th>PC</th>
<th>x0</th>
<th>x1</th>
<th>x2</th>
<th>x3</th>
<th>x4</th>
<th>x5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

add x5, x3, 1

<table>
<thead>
<tr>
<th>PC</th>
<th>x0</th>
<th>x1</th>
<th>x2</th>
<th>x3</th>
<th>x4</th>
<th>x5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

addi x4, x0, 0xFF

<table>
<thead>
<tr>
<th>PC</th>
<th>x0</th>
<th>x1</th>
<th>x2</th>
<th>x3</th>
<th>x4</th>
<th>x5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

beq x5, x4, jmplabel

<table>
<thead>
<tr>
<th>PC</th>
<th>x0</th>
<th>x1</th>
<th>x2</th>
<th>x3</th>
<th>x4</th>
<th>x5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

jmplabel: jalr x1, 0x100(x0)

<table>
<thead>
<tr>
<th>PC</th>
<th>x0</th>
<th>x1</th>
<th>x2</th>
<th>x3</th>
<th>x4</th>
<th>x5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Problem 2 (Pseudo-instruction Deconstruction)

RISC-V assembly provides a lot of helpful pseudo-instructions, that do not have their own encoding, however, can be written as one or more actual instructions which when executed perform the operation of the pseudo-instruction. For each of these pseudo-instructions, write one or more actual instructions that perform the same function:

(assume rX, rY, and rZ are any general registers, constantN is a constant of size N bits, and label is any general instruction label)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>mv rX, rY</strong></td>
<td></td>
</tr>
<tr>
<td><strong>neg rX</strong></td>
<td></td>
</tr>
<tr>
<td><strong>call label</strong></td>
<td></td>
</tr>
<tr>
<td><strong>ret</strong></td>
<td></td>
</tr>
<tr>
<td><strong>bgt rX, rY, label</strong></td>
<td></td>
</tr>
<tr>
<td><strong>beqz rX, label</strong></td>
<td></td>
</tr>
<tr>
<td><strong>li rX, constant12</strong></td>
<td></td>
</tr>
<tr>
<td><strong>li rX, constant32</strong></td>
<td></td>
</tr>
</tbody>
</table>
Problem 3 (Manual Code Compilation) ★

Consider the following two pieces of C code that do population counting (find the total number of ones in a binary number):

```c
int popcnt(int y) {
    int s = 0;
    while (y) {
        if (y & 0x1) s++;
        y >>= 1;
    }
    return s;
}
```

-------------

```c
int popcnt(int y) {
    int s = 0;
    while (y) {
        if (y & 0x80000000) s++;
        y <<= 1;
    }
    return s;
}
```

(A) Do both pieces of code perform the correct function? If yes, how? If no, what do we need to change in either piece of code in order to make it work properly? *Hint: think about what happens when y is negative.*
(B) Translate the first piece of code (after applying fixes, if any) into RISC-V assembly. Note that in RISC-V, by convention, the arguments are passed in the ax registers and the return value is saved in a0, so the input y will be in the a0 register and the return value s should also be saved in the a0 register. Also mention which register each variable is stored in.

```c
int s = 0;

while (y) {
    if (y & 0x1) s++;
    y >>= 1;
}
return s;
```
Problem 4 (Intermediate Representation Optimization) ★

Consider the following C code.

```c
int t = 35;
int x = 60;
int y = 2;

while (true) {
    if (x < 2*t) {
        x += 3;
    } else if (x > 2*t) {
        x -= 3;
    }
}
```

(A) Draw the IR (intermediate representation) control flow graph for this code.
(B) Perform dead code elimination on the IR control flow graph.

(C) Perform constant propagation.
(D) Perform constant folding.

(E) Perform dead code elimination again.
Problem 5 (Many Ways to Calc a Fact)

Consider the following two ways to implement the factorial function. Compile them to assembly.

```c
int fact1(int n) {
    int r = 1;
    while (n > 1) {
        r *= n;
        n--;
    }
    return r;
}

int fact2(int n) {
    if (n == 1) return 1;
    else return n * fact2(n-1);
}
```
Problem 6 (Protocols and Procedures) ★

For the following C functions, does the corresponding RISC-V assembly obey the RISC-V calling conventions? If not, rewrite the function so that it does obey the calling conventions.

(A) int function_A(int a, int b) {
    some_other_function();
    return a + b;
}

function_A:
    addi sp, sp, -8
    sw a0, 8(sp)
    sw a1, 4(sp)
    sw ra, 0(sp)
    jal some_other_function
    lw a0, 8(sp)
    lw a1, 4(sp)
    add a0, a0, a1
    lw ra, 0(sp)
    addi sp, sp, 8
    ret
(B) int function_B(int a, int b) {
    int i = foo((a + b) ^ (a - b));
    ret (i + 1) ^ i;
}

function_B:
    addi sp, sp, -4
    sw ra, 0(sp)
    add t0, a0, a1
    sub a0, a0, a1
    xor a0, t0, a0
    jal foo
    addi t0, a0, 1
    xor a0, t0, a0
    lw ra, 0(sp)
    addi sp, sp, 4
    ret
(C) int function_C(int x) {
    foo(1, x);
    bar(2, x);
    baz(3, x);
    return 0;
}

function_C:
    addi sp, sp, -4
    sw ra, 0(sp)
    mv a1, a0
    li a0, 1
    jal foo
    li a0, 2
    jal bar
    li a0, 3
    jal baz
    li a0, 0
    lw ra, 0(sp)
    addi sp, sp, 4
    ret

yes ... no
(D) int function_D(int x, int y) {
    int i = foo(1, 2);
    return i + x + y;
}

function_D:
    addi sp, sp, -4
    sw ra, 0(sp)
    mv s0, a0
    mv s1, a1
    li a0, 1
    li a1, 2
    jal foo
    add a0, a0, s0
    add a0, a0, s1
    lw ra, 0(sp)
    addi sp, sp, 4
    ret

yes ... no
Problem 7 (To Save or Not to Save)

Consider the following function in C:

```c
void do_something(int x) {
    f(x);
    g(x);
    h(x);
}
```

(A) Write this function in RISC-V assembly using the stack to save and restore the value x before each function call to f, g, or h.

(B) Write this function in RISC-V assembly using a saved register to restore the value x before each function call to f, g, or h.
(C) What are the advantages of each approach? Which should a compiler likely choose?
Problem 8 (Using the Stack)

Integer arrays `season1` and `season2` contain points Ben Bitdiddle had scored at each game over two seasons during his time at MIT Intramural Basketball Team. Please write a RISC-V assembly function `greaterthan20` which counts the number of games he scored more than 20 points. An equivalent C function and a sample use case are given below. Note that the base addresses for arrays `season1` and `season2` along with their size are passed down to function `greaterthan20`.

```c
int greaterthan20(int a[], int b[], int size) {
    int count = 0;
    for (int i = 0; i < size; ++i) {
        if (a[i] > 20) count += 1;
        if (b[i] > 20) count += 1;
    }
    return count;
}
```

```c
int main() {
    int season1[] = {18, 28, 19, 33, 25, 11, 20};
    int season2[] = {30, 12, 13, 33, 37, 19, 22};
    int result = greaterthan20(season1, season2, 7);
}
```

// Beginning of your assembly code
`greaterthan20`:
  `li t0, 0 // t0 ← count`
  `li t1, 0 // t1 ← index`
  `li t2, 20`

`loop:`