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>0x0010</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>0x0010</td>
<td>0x0000</td>
<td>0x00AA</td>
<td>0x0000</td>
<td>0x0000</td>
<td>0x0000</td>
<td>0x0000</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>0x0010</td>
<td>0x0000</td>
<td>0x00AA</td>
<td>0x00AB</td>
<td>0x0000</td>
<td>0x0000</td>
<td>0x0000</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>0x0010</td>
<td>0x0000</td>
<td>0x00AA</td>
<td>0x00AB</td>
<td>0x0055</td>
<td>0x0000</td>
<td>0x0000</td>
</tr>
</tbody>
</table>

**add x5, x3, x1**

<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>0x0010</td>
<td>0x0000</td>
<td>0x00AA</td>
<td>0x00AB</td>
<td>0x0055</td>
<td>0x0000</td>
<td>0x00FF</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>0x0014</td>
<td>0x0000</td>
<td>0x00AA</td>
<td>0x00AB</td>
<td>0x0055</td>
<td>0x00FF</td>
<td>0x00FF</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>0x0020</td>
<td>0x0000</td>
<td>0x00AA</td>
<td>0x00AB</td>
<td>0x0055</td>
<td>0x00FF</td>
<td>0x00FF</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>0x0010</td>
<td>0x0000</td>
<td>0x0204</td>
<td>0x00AB</td>
<td>0x0055</td>
<td>0x00FF</td>
<td>0x00FF</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)

<table>
<thead>
<tr>
<th>Pseudo-instruction</th>
<th>Actual Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mv rX, rY</code></td>
<td><code>add rX, rY, x0</code></td>
</tr>
<tr>
<td><code>neg rX</code></td>
<td><code>sub rX, x0, rX</code></td>
</tr>
<tr>
<td><code>call label</code></td>
<td><code>jal ra, label</code></td>
</tr>
<tr>
<td><code>ret</code></td>
<td><code>jalr x0, 0(ra)</code></td>
</tr>
<tr>
<td><code>bgt rX, rY, label</code></td>
<td><code>blt rY, rX, label</code></td>
</tr>
<tr>
<td><code>beqz rX, label</code></td>
<td><code>beq rX, x0, label</code></td>
</tr>
<tr>
<td><code>li rX, constant12</code></td>
<td><code>addi rX, x0, constant12</code></td>
</tr>
</tbody>
</table>
  `addi rX, x0, constant32[11:0]`  
  We add constant32[11] here to offset the sign extension done by addi. If constant32[11] is 1 then constant32[11:0] will be treated as a negative number and will be sign extended by adding many 1’s in front of it. This is effectively the same as adding a -1 in the 12th bit position, so can be reversed by adding a 1 back in the 12th bit position. |
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;
}

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

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.*

The value y needs to be an unsigned int otherwise the right shifts in the first piece of code will be arithmetic, leading to additional 1s shifted into y and never-ending execution if 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.

```assembly
int s = 0;  
li a1, 0

while (y) {
    _start:
        beqz a0, _end

    if (y & 0x1) s++;  
        andi a2, a0, 0x1  
        beqz a2, _shift  
        addi a1, a1, 0x1

    _shift:
        srli a0, a0, 0x1

}  
return s;
    _end:
        mv a0, a1
        ret
```

y is stored in a0, s is stored in a1, and the intermediate result of y & 0x1 is stored in a2.

Share this Godbolt link (https://godbolt.org/z/P9rdesjqh) and go through it and the differences between handwritten assembly and compiler generated assembly (with -O1, because -O0 makes for unreadable assembly). Main difference is that you can add the result of the and operation directly to the variable s without having an extra branch. Mention that \*BRANCHES BAD\*.

Another difference is that the compiler prefers to move y out of a0 and accumulate the final result in a0 itself. Probably because having the return result in a0 always means you can return anytime with the ret instruction.

Another optimization made at higher -Ox levels is that the instructions with data dependencies are pushed apart to reduce stalling. See the difference between -O1 and -O3, where add and srli are interchanged. Also some redundant code elimination where li a0, 0 is removed right after a comparison between a0 and 0.

Mention that Godbolt is your best friend when trying to understand compilation from source to assembly. Also that compilers are smart (smarter than us), and can perform all sorts of crazy optimizations, such as recognizing a popcnt when they see one, and replacing the entire thing with a popcnt instruction if it exists on the target.
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.

![Control Flow Graph](image-url)
(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;
}
```

```assembly
fact1:
    mv a5, a0
    li a4, 1
    ble a0, a4, _end
    li a0, 1
_while:
    mul a0, a0, a5
    addi a5, a5, -1
    bne a5, a4, _while
    ret
_end:
    li a0, 1
    ret

a0: r, a5: n, a4: 1 (for comparison)
```

```c
int fact2(int n) {
    if (n == 1) return 1;
    else return n * fact2(n-1);
}
```

```assembly
fact2:
    addi sp, sp, -16
    sw ra, 12(sp)
    sw s0, 8(sp)
    mv s0, a0
    li a5, 1
    bne a0, a5, _else
_end:
    mv a0, s0
    lw ra, 12(sp)
    lw s0, 8(sp)
    addi sp, sp, 16
    ret
_else:
    addi a0, a0, -1
    call fact2
    mul s0, s0, a0
    j _end

stack saves ra and s0
s0 saves the value of a0 (i.e. n passed in)
a5 stores 1 for comparison
```

Finally, check [https://godbolt.org/z/haEr6Ez47](https://godbolt.org/z/haEr6Ez47) to see what other optimizations are possible at higher -Ox levels. Main one being that at -O2 and above, the compiler realizes what the recursion is doing and basically emits the same assembly for both.
**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

    yes ... no

    addi sp, sp, -8 only allocates two words on the stack, 0(sp) (i.e. sp + 0) and 4(sp) (i.e. sp + 4). Therefore, using the address 8(sp) violates the calling convention because we have not allocated space for a third word. We can fix it just by replacing the -8 with -12, and the 8 with 12 at the end:

function_A:
    addi sp, sp, -12
    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, 12
    ret

    Everything else is correct. We save a0 and a1 onto the stack and restore them after calling some_other_function, since they are caller saved, and that function is allowed to overwrite them. Then we add them and put the result in a0, where it is returned to the caller.
(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  

yes ... no

Nothing is wrong here. addi sp, sp, -4 allocates the address 0(sp), which we use to store and restore ra, so that it's OK when ra is overwritten by calling foo. sp is also restored to the old value at the end. All used registers a0, a1, t0 are caller-saved registers, so we are allowed to modify them. Since we also don’t assume that foo preserves any of those registers when called, as we only need its return value, which appears in a0. Thus, we do not need to restore a0, a1, t0 or save them to the stack.
The code assumes that its argument \( x \) will stay in register \( a1 \) as it calls functions \( \text{foo} \) and \( \text{bar} \), because it needs to pass the same argument to \( \text{bar} \) and \( \text{baz} \). However, those functions are allowed to overwrite \( a1 \) by calling convention, since \( a1 \) is a caller saved register. Instead, we must store \( x \) in the stack and restore it when we need it again. (Note that we only need to store \( x \) once, and we can load it twice; that part of the stack belongs to this function, so neither \( \text{foo} \) nor \( \text{bar} \) is allowed to modify it. Also, we do not need to restore \( a1 \) after returning from \( \text{baz} \), because we don’t need it anymore and we aren’t required to preserve it by calling convention.)

```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
```
(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

If we want to use callee saved registers s0 and s1, we must preserve them for our caller to abide by the calling convention. So we need to allocate additional space on the stack, store the initial values of the saved registers, and restore them before we return. Otherwise, this is a legal and reasonable use of s0 and s1 to store values (x and y) that we don’t want the call to foo to overwrite.

    function_D:
        addi sp, sp, -12
        sw ra, 0(sp)
        sw s0, 4(sp)
        sw s1, 8(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)
        lw s0, 4(sp)
        lw s1, 8(sp)
        addi sp, sp, 12
        ret

An alternative would be to forgo the usage of s0 and s1 entirely, and simply store and restore a0 and a1 to/from the stack directly. We can choose any caller-saved registers other than a0 to restore those values to; below we choose t0 and t1.

    function_D:
        addi sp, sp, -12
sw ra, 0(sp)
sw a0, 4(sp)
sw a1, 8(sp)
li a0, 1
li a1, 2
jal foo
lw t0, 4(sp)
lw t1, 8(sp)
add a0, a0, t0
add a0, a0, t1
lw ra, 0(sp)
addi sp, sp, 12
ret
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.

```assembly
do_something:  addi sp, sp, -8
                sw ra, 0(sp)
                sw a0, 4(sp)
                call f
                lw a0, 4(sp)
                call g
                lw a0, 4(sp)
                call h
                lw ra, 0(sp)
                addi sp, sp, 8
```

(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.

```assembly
do_something:  addi sp, sp, -8
                sw ra, 0(sp)
                sw s0, 4(sp)
                mv s0, a0
                call f
                mv a0, s0
                call g
                mv a0, s0
                call h
                lw ra, 0(sp)
                lw s0, 4(sp)
                addi sp, sp, 8
```
(C) What are the advantages of each approach? Which should a compiler likely choose?

Approach (A) has fewer instructions. Approach (B) has fewer loads and stores to the stack. Since interactions with memory typically take a long time, the compiler should choose approach B.
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:
    // We are translating the for loop of the original greaterthan20 code into something more easily translated
    // into assembly code, like a while loop of the form:
    // while i < size:
    //     (body of loop goes here)
    //     i = i + 1
    ble a2, t1, endloop // if a2 (= size) <= t1 (= i), jump to endloop
        // (that is, stop the loop if i < size is false)
    slli t3, t1, 2 // t3 ← 4 × index
checka:
    add t4, a0, t3 // t4 ← a0 (= base of array a) + t3
        // (a0 + t3 = address of a[i])
    lw t5, 0(t4) // t5 = value at address t4 = value of a[i]
    ble t5, t2, checkb // if a[i] <= 20, then skip to checking b[i]
    addi t0, t0, 1 // increment count
checkb:
    add t4, a1, t3 // t4 ← a1 (= base of array b) + t3
        // (a1 + t3 = address of b[i])
    lw t5, 0(t4) // t5 = contents of address t4 = value of b[i]
    ble t5, t2, endcompare // if b[i] <= 20, then go to endcompare
    addi t0, t0, 1 // increment count
endcompare:
addi t1, t1, 1 // increment index i
j loop // restart loop from the condition check
endloop:
mv a0, t0 // move count to a0, the register for holding the
           // return value
ret