1 Powerful RISC-V Functions

1. Write a function `double` in RISC-V that, when given an integer \( x \), returns \( 2x \).

   ```
   double: 
   add a0, a0, a0
   jr ra
   ```

2. Write a function `power` in RISC-V that takes in two numbers \( x \) and \( n \), and returns \( x^n \). You may assume that \( n \geq 0 \) and that multiplication will always result in a 32-bit number.

   ```
   power: 
   li t0, 0 # Set t0 to be a 0 (counter variable)
   addi t1, a0, 0 # Set t1 to be a0, which represents x
   addi a0, x0, 1 # Set a0, the return value, to 1
   loop: 
   bge t0, a1, end # End the loop if the counter is greater than or equal to a1 (represents n)
   mul a0, a0, t1 # Multiply the running product a0 by t1 (which holds x)
   addi t0, t0, 1 # Increment the counter
   jal x0, loop # Jump back to the while condition
   end: 
   jr ra # Return to caller
   ```

3. want more power?

   ```
   power: 
   addi t0 x0 1 # iterator starts at 1 since we are using blt a1 and t0
   addi t1 a0 0
   addi a0 x0 1
   loop: 
   blt a1 t0 end
   mul a0 a0 t1
   addi t0 t0 1
   jal x0 loop
   end: 
   jalr x0 ra 0
   ```

2 RISC-V with Arrays and Lists

Comment each snippet with what the snippet does. Assume that there is an array, `int arr[6] = {3, 1, 4, 1, 5, 9}`, which is starts at memory address 0xBFFFFF00, and a linked list struct (as defined below), `struct ll* lst;`, whose first element is located at address 0xABCD0000. \( s0 \) then contains \( arr \)'s address, 0xBFFFFF00, and \( s1 \) contains \( lst \)'s address, 0xABCD0000. You may assume integers and pointers are 4 bytes and that structs are tightly packed.

```
struct ll { 
    int val;
    struct ll* next;
}
```

1. `lw t0, 0(s0)` # Loads \( arr[0] \) into register \( t0 \)
`lw t1, 8(s0)` # Loads \( arr[2] \) into register \( t1 \)
`add t2, t0, t1` # Sets \( t2 \) equal to \( t0 \) plus \( t1 \)
`sw t2, 4(s0)` # Sets \( arr[1] \) equal to value in \( t2 \)

Sets \( arr[1] \) to \( arr[0] + arr[2] \)
2. 

```assembly
add t0, x0, x0  # Sets register t0 to 0
loop: slti t1, t0, 6  # Sets t1 to 1 if t0 < 6, 0 otherwise
   beq t1, x0, end  # Branches to the end if t1 is 1 (t0 >= 6)
   slli t2, t0, 2  # Sets t2 to t0 * 4 (4 is number of bytes in an integer)
   add t3, s0, t2  # Sets t3 to the address of arr[t0] (added t2 bytes to arr)
   lw t4, 0(t3)  # Load arr[t0] into register t4
   sub t4, x0, t4  # Sets t4 to its negative
   sw t4, 0(t3)  # Stores this updated value back at arr[t0]
   addi t0, t0, 1  # Increments t0 to move to the next element
   jal x0, loop  # Jump back to the loop label
end:
```

Negates all elements in arr
3. loop: beq s1, x0, end  # Branch to the end if struct pointer (s1) is NULL
lw t0, 0(s1)    # Load the value of the node into t0
addi t0, t0, 1  # Increment t0 by 1
sw t0, 0(s1)    # Store the incremented value back into the node
lw s1, 4(s1)    # Load the address of the next element into s1
jal x0, loop    # Jump back to the loop label

end:

Increments all values in the linked list by 1.

3 Translating between C and RISC-V

Translate between the C and RISC-V code. You may want to use the RISC-V Green Card as a reference. We show you how the different variables map to registers – you don’t have to worry about the stack or any memory-related issues.

<table>
<thead>
<tr>
<th>C</th>
<th>RISC-V</th>
</tr>
</thead>
<tbody>
<tr>
<td>// Nth_Fibonacci(n):</td>
<td>...</td>
</tr>
<tr>
<td>// s0 -&gt; n, s1 -&gt; fib</td>
<td>beq s0, x0, Ret0</td>
</tr>
<tr>
<td>// t0 -&gt; i, t1 -&gt; j</td>
<td>addi t2, x0, 1</td>
</tr>
<tr>
<td>// Assume fib, i, j are already these values</td>
<td>beq s0, t2, Ret1</td>
</tr>
<tr>
<td>int fib, i, j are already these values</td>
<td>addi s0, s0, -2</td>
</tr>
<tr>
<td>if (n==0) return 0;</td>
<td>Loop: beq s0, x0, RetF</td>
</tr>
<tr>
<td>else if (n==1) return 1;</td>
<td>addi s1, t0, t1</td>
</tr>
<tr>
<td>n -= 2;</td>
<td>addi t1, t0, 0</td>
</tr>
<tr>
<td>while (n != 0) {</td>
<td>addi t0, s1, 0</td>
</tr>
<tr>
<td>fib = i + j;</td>
<td>addi s0, s0, -1</td>
</tr>
<tr>
<td>j = i;</td>
<td>jal x0, Loop</td>
</tr>
<tr>
<td>i = fib;</td>
<td>Ret0: addi a0, x0, 0</td>
</tr>
<tr>
<td>n--;</td>
<td>Ret1: addi a0, x0, 1</td>
</tr>
<tr>
<td>} return fib;</td>
<td>jal x0, Done</td>
</tr>
<tr>
<td></td>
<td>RetF: add a0, x0, s1</td>
</tr>
<tr>
<td></td>
<td>Done: ...</td>
</tr>
</tbody>
</table>

Just how many times do we have to do fibonacci??????

beq s0, x0, Ret0
addi a1, x0, 1 #using a1 instead of t2 coz why not
beq s0, a1, Ret1
addi s0, s0, -2
Loop: beq s0, x0, RetF
addi s1, t0, t1
addi t1, t0, 0
addi t0, s1, 0
addi s0, s0, -1
jal x0, Loop
Ret0: addi a0, x0, 0
jal x0, Done
Ret1: addi a0, x0, 1
jal x0, Done
RetF: add a0, x0, s1
Done: jalr x0 ra 0
4 RISC-V Calling Conventions

1. How do we pass arguments into functions?
   Use the 8 arguments registers \texttt{a0 - a7}.

2. How are values returned by functions?
   Use \texttt{a0} and \texttt{a1} as the return value registers.

3. What is \texttt{sp} and how should it be used in the context of RISC-V functions?
   \texttt{sp} stands for stack pointer. We subtract from \texttt{sp} to create more space and add to free space. The stack is mainly used to save (and later restore) the value of registers that may be overwritten.

4. Which values need to saved before using \texttt{jal}?
   Registers \texttt{a0 - a7, t0 - t6, and ra}.

5. Which values need to be restored before using \texttt{jr} to return from a function?
   Registers \texttt{sp, gp, gp, and s0 - s11}.

5 Writing RISC-V Functions

Write a function \texttt{sumSquare} in RISC-V that, when given an integer \( n \), returns the summation below. If \( n \) is not positive, then the function returns 0.

\[
 n^2 + (n - 1)^2 + (n - 1)^2 + \ldots + 1^2 
\]

For this problem, you are given a RISC-V function called \texttt{square} that takes in an integer and returns its square. Implement \texttt{sumSquare} using \texttt{square} as a subroutine.

```assembly
sumSquare: addi sp, sp -12  # Make space for 3 words on the stack
        sw ra, 0(sp)       # Store the return address
        sw s0, 4(sp)       # Store register s0
        sw s1, 8(sp)       # Store register s1
        add s0, a0, x0     # Set s0 equal to the parameter n
        add s1, x0, x0     # Set s1 equal to 0 (this is where we accumulate the sum)
        loop: bge x0, s0, end # Branch if s0 is not positive
        add a0, s0, x0      # Set a0 to the value in s0 to prepare for the function square
        jal ra, square      # Call the function square
        add s1, s1, a0      # Add the returned value into the accumulator s1
        addi s0, s0, -1     # Decrement s0 by 1
        jal x0, loop        # Jump back to the loop label
        end: add a0, s1, x0  # Set a0 to s1, which is the desired return value
        lw ra, 0(sp)        # Restore ra
        lw s0, 4(sp)        # Restore s0
        lw s1, 8(sp)        # Restore s1
        addi sp, sp, 12     # Free space on the stack for the 3 words
        jr ra                # Return to the caller
```