Hi there! Welcome back to section, we’re happy that you’re here 😊

**Control Flow and Condition Codes**

Internally, condition codes (Carry, Zero, Sign, Overflow) are set based on the result of the previous operation. The `j*` and `set*` families of instructions use the values of these “flags” to determine their effects. See the table provided on your reference sheet for equivalent conditionals.

An *indirect jump* is specified by adding an asterisk (*) in front of a memory operand and causes your program counter to load the address stored at the computed address. (e.g. `jmp *%rax`) This is useful for switch case statements.

**Procedure Basics**

The instructions `push`, `pop`, `call`, and `ret` move the stack pointer (`%rsp`) automatically.

 `%rax` is used for the return value and the first six arguments go in `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9` (*Diane’s Silk Dress Cost $89*).

**Exercises:**

1. [CSE351 Au15 Midterm] Convert the following C function into x86-64 assembly code. You are not being judged on the efficiency of your code – just the correctness.

   ```c
   long happy(long *x, long y, long z) {
     if (y > z)
       return z + y;
     else
       return *x;
   }
   ```

   ```assembly
   happy:
       cmpq %rdx, %rsi
       jle .else
       leaq (%rdx, %rsi), %rax
       ret
   .else:
       movq (%rdi), %rax
       ret
   ```

   Multiple other possibilities (*e.g.* switch ordering of if/else clauses, replace `lea` with `mov/add instruction pair`).
2. Write an equivalent C function for the following x86-64 code:

```assembly
mystery:
1   testl  %edx, %edx       # %edx is 3rd argument (z)
2   js      .L3             # jump to .L3 if z<0
3   cmpl   %esi, %edx      # %esi is 2nd argument (y)
4   jge    .L3             # jump to .L3 if y<=z
5   movslq %edx, %rdx      # sign-extend 3rd argument (z)
6   movl   (%rdi,%rdx,4), %eax # %rdi is 1st argument (x), calc *(x + z*4)
7   ret
 .L3:
8   movl  $0, %eax          # return 0
9   ret
```

```c
int mystery(int *x, int y, int z) {
    if (z >= 0 && z < y)
        return x[z];
    else
        return 0;
}
```

Notes:
- If either conditional is True, then we jump to the “else” clause, so in C we execute the “if” clause only when the complement of both of them are True.
- Line 6 indicates that the return type is 4 bytes (int). Line 8 is ambiguous since it zeros out the entire 8 bytes of %rax.
- Argument variable names are arbitrary. Based on usage, could perhaps have used x→ar, y→n, z→k.
- First argument had to point to int based on scale factor in Line 6. Both int *x and int x[] work.
3. [CSE351 Wi17 Midterm] Consider the following x86-64, (partially blank) C code, and memory diagram. Addresses and values are 64-bit. Fill in the C code based on the given assembly.

```c
int foo(long* p) {
    int result = 0;
    while (p != NULL) {
        p = *(long**)p;
        result = result + 1;
    }
    return result;
}
```

Part 2: Follow the execution of `foo` in assembly, where `0x1000` is passed in to `%rdi`

Write the values of `%rdi` and `%eax` in the columns. If the value doesn’t change, you can leave it blank.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>%rdi (hex)</th>
<th>%eax (decimal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>movl $0, %eax</td>
<td></td>
<td></td>
</tr>
<tr>
<td>testq %rdi, %rdi</td>
<td></td>
<td></td>
</tr>
<tr>
<td>je L2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>movq (%rdi), %rdi</td>
<td>0x1030</td>
<td></td>
</tr>
<tr>
<td>addl $1, %eax</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>jmp L1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>movq 0x0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>addl</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>jmp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>testq</td>
<td></td>
<td></td>
</tr>
<tr>
<td>je</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1000</td>
<td>0x1030</td>
</tr>
<tr>
<td>0x1008</td>
<td>0x1020</td>
</tr>
<tr>
<td>0x1010</td>
<td>0x1000</td>
</tr>
<tr>
<td>0x1018</td>
<td>0x0000</td>
</tr>
<tr>
<td>0x1020</td>
<td>0x1030</td>
</tr>
<tr>
<td>0x1028</td>
<td>0x1008</td>
</tr>
<tr>
<td>0x1030</td>
<td>0x0000</td>
</tr>
<tr>
<td>0x1038</td>
<td>0x1038</td>
</tr>
<tr>
<td>0x1040</td>
<td>0x1048</td>
</tr>
<tr>
<td>0x1048</td>
<td>0x1040</td>
</tr>
</tbody>
</table>