Implementing setjmp/longjmp

In this question we'll look at how the setjmp/longjmp exception mechanism can be implemented. The goal is to give you more exposure to calling conventions and assembly programming. Consider a 32-bit computer architecture with 5 registers ($r0, ..., r4$): $r0$-$2$ are used for function parameters, $r3$ is used for the return address, and $r4$ is used to store the return code. Implement setjmp($jmp\_buf*$) and longjmp($jmp\_buf*$, int) in assembly. Assume that the $jmp\_buf$ is large enough, and that the second argument to longjmp is never 0.

setjmp:

```
movl r0, 0(r0) ; Hint: setjmp() cannot modify any registers until
movl r1, 4(r0) ; it has saved them.
movl r2, 8(r0) ;
movl r3, 12(r0) ;
xorl r4, r4 ; Don't save the return code register. Set it to 0.
retl ; Return to the address stored in r3.
```

longjmp:

```
movl r1, r4 ; Set the return code up front before r1 disappears.
movl 4(r0), r1 ; Skip recovering r0 to preserve the jmp_buf pointer.
movl 8(r0), r2 ;
movl 12(r0), r3 ; Recover the return address passed into setjmp().
movl 0(r0), r0 ; Finally, recover r0. We handled the others already.
retl ; Return "again" from setjmp() by jumping to (r3).
```

1. Is it possible to longjmp to the same jmp_buf multiple times?

   Yes.

2. Write a program which throws an exception, handles it, and returns to the exception site from the exception handler using setjmp/longjmp.

```
void f() {
    jmp_buf jb1, jb2;
    if (setjmp(&jb1) == 0) {
        if (setjmp(&jb2) == 0) {
            longjmp(&jb1, 1);
        } else {
            return;
        }
    } else {
        longjmp(&jb2, 2);
    }
}
```
Implementing `apply()`

In many functional programming languages, `apply(Function, ListOfArgs)` is considered to be a basic primitive. We will implement a very limited version of `apply` for the architecture defined in the previous question. To avoid passing arguments on the stack, we will cap the number of arguments in `ListOfArgs` to 2. Write the assembly for the function `apply(void *func, void **args, int argc)`.

```
apply:
    movl r0, r4          ; Save the function pointer in r4 for now.
    beq r2, $0, L0      ; If argc == 0, goto L0.
    beq r2, $1, L1      ; If argc == 1, goto L1.
    beq r2, $2, L2      ; If argc == 2, goto L2.
    j error             ; Otherwise, goto error.
L2:
    movl 0(r1), r0      ; Load the first argument.
    movl 4(r1), r1      ; Load the second argument.
    j L0
L1:
    movl 0(r1), r0      ; Load the first argument.
L0:
    call *r4            ; Call f.
    retl                ; Pass the return code of f to the caller.
error:
    movl $1, r4         ; Set the return code to 1.
    retl
```

1. What are we missing to handle passing arguments on the stack?
   There is no dedicated register for the stack pointer which is preserved by this calling convention.

2. Can we just use a loop to iterate through the `args` array instead of handling arguments one by one?
   It’s possible to use a loop to copy arguments into the stack for the callee, but it’s awkward to do this for register arguments.