Computer Science 237
Assignment 3
Due next Tuesday, before Assignment 4.

This week I would like you to convert the code you wrote for last week's lab into assembly. That code, you will recall, took a pointer to a sokoban `level` structure and returned a true value (probably 1) if (and only if) each box in the level was stowed away. As an alternative to translating your code, I have included my own implementation that you may choose to translate at no penalty.

You can pick up the distribution of this lab from the course `kits` directory as `sokoban2.tar.gz`. To unzip and untar this package type

```
tar xvfz /usr/cs-local/share/cs237/kits/sokoban2.tar.gz
```

The result is a directory, `sokoban2`, that contains the source, this lab handout, and a number of data files. The structure of this directory is very similar to last week's directory. You can, as usual, build sokoban with

```
make sokoban
```

There are a couple of small changes:

1. The attached version of `win.c` is included.
2. I've changed the make target in this directory to assemble the `win` function from the source `win.s`, instead of `win.c`. (The old target can be made with `make csokoban`.)
3. The current `win.s` file has a minimal implementation that will compile, but will never detect a puzzle solution.
4. I've included, in the `examples` directory, a couple of C routines and their translations into assembly. You should investigate these.

If you've made changes to the screen directory or the source in `sokoban.c`, you can copy those files over without worry.

Preliminaries.
We must remember that we don't know how to access memory, so we are limited to using the registers to get our computation done. Because these registers are used by (1) the procedure that calls us (a complex procedure, `play`) as well as (2) the functions we call (for example, `highlight`), we must all agree who "owns" the contents of a register at any particular time.

Recall that the table on page 5 of the x86-64 handout lists all the registers that are available for our use. In particular:

1. The registers `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, and `%r9` are used to pass the first six parameters to a function, *in that order*. This means, for example, that the `level` pointer passed as your sole argument in the `win` function will appear in register `%rdi`. If a procedure uses fewer than 6 arguments, you should feel free to make use of the remaining argument registers, without concern.

   If you don't call another procedure (you're a "leaf" procedure), then you have nothing to worry about; the six registers may be freely used.

   Be aware, however, that if you *do* call a procedure from the `win` method, you will have to surrender the use of those registers to what we refer to as *the callee routine* (for example, `highlight`, or `get`). Surrendering those registers means moving the values someplace else and restoring them after the call. This can be a tedious process if you call a lot of functions, so ideally you should find some way of saving and restoring values once in the procedure. Keep reading.

2. As we saw on Friday and Monday, the register `%rax` is used to return the value from a function. Since many functions are called, even during he life of `win`, you probably want to keep long-needed values out of that register. Of course, your true-or-false return value should be stored in `%rax`, just before you execute the `ret` instruction.

3. Several registers, namely `%rbp`, `%rsp`, `%r11`, and `%r12` maintain a hierarchy or *stack* of procedures and functions that are currently active. How this works will unfold over the next few weeks, but you should think of `%rsp` as pointing to the last item pushed onto a general purpose stack of values. Other instructions make use of this, as well. For example, when you call a procedure with

```
x:  call highlight
y:
```

The address of the next instruction (`y`) is pushed on top of a memory based stack pointed to by `%rsp`, and the instruction pointer, `%rip`, is loaded with the address of `highlight`. When `highlight` returns, it performs a return:

```
ret
```

This causes the return address (sitting on top of the stack) to be popped off and re-placed back in the instruction pointer, `%rip`. The effect is to pick up in the calling routine just after the call to `highlight`. **It is vitally important** that if, in `highlight`, we add even more things to the stack that we make sure they are removed before the `ret` instruction is executed. If the stack is not "kept level", the state of the machine is usually irrevocably lost.

The other registers (`%rbp`, et al.) are interpreted in similarly complex ways. We don't modify these registers unless we fully understand what we're doing. (We don't. Yet.)

4. The remaining registers `%rbx`, and `%r12` through `%r15` are "callee saved". You may use these registers, if you want, but their values are owned by the procedure that called you. You must save the register contents before you make use of the register, and you must restore the value before you return. If you use these registers, the values are typically pushed on the stack at the top of your procedure, and they are popped from the stack just before you return.

   Suppose, for example, you wish to make use of registers `%r13` and `%r14` to store your local variables. You would push the registers on the stack at the top of your procedure:

```
win:
    pushq   %r13
    pushq   %r14
     . . .
```

   and you would restore them after you are done with them, typically just before you return:

```
    popq    %r14
    popq    %r13
    ret
```

   Remember: these are pushed onto, and popped off of a stack, so they must always be popped off in opposite order they were pushed on.

   The great advantage to these registers is that you need not save them before you call a function: their preservation is the responsibility of the callee, not the caller routine.

I am guessing that your win method makes quite a few function calls, so you might be best advised to use the callee saved registers to store your local variables.

Data sizing.
The `win` method confronts you with a variety of data sizes:

1. First, all pointers are stored as 8-byte (quad-word) quantities. They use a full-width register, typically starting with `%r`.
2. When you save and restore registers, always use the quadword versions: `pushq` and `popq`. This ensures that you protect all the bits of the register, no matter how many you *think* you're using.
3. Several values in `win` and the methods it calls are declared as `int` values. These values are represented as 4-byte (long- or double-word). They use 32-bit register names: `%eax`, `%r16d`, etc. Notice that registers *sometimes* end with the letter `d` and *not* `l`.
4. We won't need word values, but if we did, they would be stored in 16-bit registers, like `%ax`, `%bx`, and `%r8w`.
5. Characters (such as the value returned by the `get` routine) are represented as bytes. These values can be manipulated in registers like `%al`, `%bl`, and `%r8b`. The `l` in `%al` and `%bl` stands for *low*, not long. (Don't get me started.)

I will require (even if the assembler does not) that every data-manipulation instruction include an instruction size extension. These are the letters `q`, `l`, `w`, or `b`. For example, if we want to move an 8-bit value from the `%rax` register to the `%r8` register, we would use the instruction

```
movb    %al,%r8b
```

In many cases, including this specific case, the assembler can deduce that you want to move a byte, even if you use `mov` instead of `movb`. Such dependencies lead to headaches and other withdrawal symptoms later in life.

What is to be done.
To complete this lab, you must flush out the body of the `win` routine in the `win.s` file. Your assembly should be a direct, straightforward translation of *someone's* C-language code—either yours or mine. Each executable statement should be commented. A typical approach is to use the C-language code that you're translating as the comment. At this level, C looks beautiful and informative. Raw assembly, on the other hand, is not worth the bits used to represent it.

When you are finished, your program should compile, and the winning ways of sokoban should return. Turn in (only) your assembly language, `win.s`:

```
turnin -c 237 win.s
```

A possible C implementation of Sokoban's win function.

```
// A routine for detecting a win.  This is a straightforward implementation.
// (c) 2011-2014 duane a. bailey, but freely available for academic use.
#include "sokoban.h"

// Your birthday (you should change):
int BDay[] = { 12, 15, 1960 };

// Check for a win.
// Return 1 if all the BOX locations are also STORE locations.
int win(level *l)
{
    int r,c,d;
    int result;
    int h = height(l), w = width(l);

    // check for an unstowed box (an indication of an unsolved puzzle)
    for (r = 0; r < h; r++) {
        for (c = 0; c < w; c++) {
            d = get(l,r,c);
            if ((d & BOX) && !(d & STORE)) { result = 0; goto fini; } // ugly goto
        }
    }
    result = 1;

    // assertion: we have won; highlight all boxes
    for (r = 0; r < h; r++) {
        for (c = 0; c < w; c++) {
            d = get(l,r,c);
            if (d & BOX) {
                highlight(l,r,c);
            }
        }
    }
  fini:
    return result;
}
```