Assignment 3: Function Calls and Exceptions

15-411: Compiler Design
Alex Crichton (acrichto@andrew) and Ian Gillis (igillis@andrew)

Due: Thursday, October 9, 2012 (1:30 pm)

Reminder: Assignments are individual assignments, not done in pairs. The work must be all your own.

You may hand in a handwritten solution or a printout of a typeset solution at the beginning of lecture on Tuesday, October 9. Please read the late policy for written assignments on the course web page. If you decide not to typeset your answers, make sure the text and pictures are legible and clear.

Problem 1: Calling Conventions (15 points)

(a) The function on the next page is written in x86-64 assembly (with some syntactic liberties), and it is a bit *too* concerned about preserving the calling function's register values. Without modifying the function's "work" section, rewrite the function to remove as many of the "prologue" and "epilogue" instructions as you safely can. You should also give the reasons why those instructions are unnecessary, and why the remaining ones are necessary. You should also reduce the stack frame size to the amount that is actually necessary for your new function.

(b) In addition to saving and restoring registers it doesn't need to, the program is violating the x86-64 calling conventions. Explain how.

(c) There is nothing on an x86-64 processor actually enforcing the calling conventions, and there are a couple reasons why you might be tempted to forgoe them. In some situations you could optimize away a few instructions by ignoring them, or you might simply want to use the easier to remember rule that r8 through r15 are the callee save registers. However, conventions usually exist for a good reason. What would be the negative consequences of ignoring calling conventions in your compiler? Are there any circumstances (in your compiler, or in general) when it *would* be okay to ignore calling conventions?

```
// Prologue
sub 1337, rsp
mov rbx, 108(rsp)
mov rcx, 116(rsp)
mov rdx, 124(rsp)
mov rsi, 132(rsp)
mov rdi, 140(rsp)
mov rbp, 148(rsp)
mov r8, 156(rsp)
mov r9, 164(rsp)
mov r10, 172(rsp)
mov r11, 180(rsp)
mov r12, 188(rsp)
mov r13, 196(rsp)
mov r14, 204(rsp)
mov r15, 212(rsp)

// Work
mov 8, rax
mov rsi, rbp
add rdi, rbp
mov rbx, 0(rsp)
mov rbp, 8(rsp)
add 0(rsp), r8
sub 8(rsp), r8
add r8, rax

// Epilogue
mov 108(rsp), rbx
mov 116(rsp), rcx
mov 124(rsp), rdx
mov 132(rsp), rsi
mov 140(rsp), rdi
mov 148(rsp), rbp
mov 156(rsp), r8
mov 164(rsp), r9
mov 172(rsp), r10
mov 180(rsp), r11
mov 188(rsp), r12
mov 196(rsp), r13
mov 204(rsp), r14
mov 212(rsp), r15
add 1337, rsp
ret
```

Problem 2: Tail Call Optimization (20 points)

Tail call optimization is an optimization which can be applied wherever a function *foo* makes a call to a function *bar* and then returns either nothing or the result of *bar* immediately afterward. (An important special case of this is tail recursion, where *foo* and *bar* are the same function.) Because *foo* does nothing with its local variables after the call to *bar*, it is safe to have *bar* overwrite the contents of *foo*'s stack frame instead of creating a new one, and additionally save a somewhat costly `ret` operation. Consider the following tail recursive x86-64 function:

```
foo:
  sub 40, rsp
  add 4, rdi
  cmp rdi, rdx
  je     baztime
bartime:
  call bar
  add 40, rsp
  ret
baztime:
  call baz
  mov rax, rdi
  cmp rdi, rsi
  jne   footime
  mov 34, rsi
footime:
  call foo
  add 40, rsp
  ret
```

(a) Write down a tail call optimized version of the above program.

(b) Describe the process that a compiler would need to take to perform tail call optimization in general.

(c) There is an important benefit of tail call optimization in addition to increasing program speed. Can you think of it? (If you think of multiple answers, feel free to write more than one, but please don't try the "shotgun" strategy of question answering.)

Problem 3: SSA and Neededness (25 points)

Recall from lecture that static single assignment form (SSA) is an intermediate representation in which each variable is assigned to exactly once, though possibly by a ϕ function that selects among several possible values. Consider the function on the following page.

```
foo(x, y) {
    z <- x + y
    s <- y
    q <- 0
    while (s != 0) {
        s <- s - 2
        if (s < 0) {
            q <- 9
            return q
        } else if (s < 8) {
            if (z < 5) {
                x <- x + 1
                z <- z * 2
                continue
            } else {
                break
            }
        } else {
            z <- 51 / z
        }
    }
    return q
}
```

(a) Write down the control flow graph for this program and label the nodes (basic blocks).

(b) For each node, list what nodes are in its dominance frontier.

(c) Write down the control flow graph for the program after converting it to SSA form.

(d) Recall from Lecture 5 neededness analysis. How does performing neededness analysis change when a program is in SSA form?

(e) Do neededness analysis on the above program, and write down a neededness-optimized control flow graph for the program based on this.