# Problem Set 5: Code Generation for x86. Register Allocation. Solution Key

C S 395T

29 September 2021

This is the problem set for week 5 of C S 395T SIMPL. It is intended to help you learn the material by working out more examples and exercises than is possible to cover in the videos. Feel free to work individually or in groups. Ask questions on Piazza. You are not required to submit anything, and the problem set doesn't count directly towards your course grade. The solution key for the problem set will be made available one week after its release.

**Problem 1.** *C's "short-circuit evaluation" for conditionals.*
Generate x86-64 assembly code for the following boxed fragments of C code, obeying C's semantics for `&&` and `||`. You are given the location of the variables. If the condition evaluates to `TRUE` (respectively, `FALSE`), have the generated code branch to the label `Ltrue` (respectively, `Lfalse`).

(a) `if ((year % 4 == 0 && year % 100 != 0) || year % 400 == 0)`,
with `year` (of type `unsigned`) in register `%rdi`.

(b) `if (s[n] != ' ' && s[n] != '\t' && s[n] != '\n')`,
with `s` (of type `char *`) in register `%rdi` and `n` (of type `int`) in register `%rbx`.

(c) `if (p >= allocbuf && p < allocbuf + ALLOCSIZE)`,
with `p` (of type `char *`) in register `%r11` and `allocbuf` (of type `static char *`) in register `%r12`.

(d) `if (p >= p->s.ptr && (bp > p || bp < p->s.ptr))`,
with `p` (of type `Mystery *`) in register `%r8` and `bp` (also of type `Mystery *`) in register `%r9`.
The derived type `Mystery` is defined as follows.

```
typedef union mystery {
  struct {
    union mystery *ptr;
    unsigned size;
  } s;
  long x;
} Mystery;
```

**Solution:**

(a)
```
        movl %edi, %eax
        andl $3, %eax
        cmpl $0, %eax  # checking year % 4
        jne L_2
        movl %edi, %eax
        xorl %edx, %edx
```

```
        movl $100, %ecx
        divl %ecx
        cmpl $0, %edx # checking year % 100
        jne Ltrue
L_2:
        movl %edi, %eax
        xorl %edx, %edx
        movl $400, %ecx             ## imm = 0x190
        divl %ecx
        cmpl $0, %edx # checking year % 400
        jne Lfalse
Ltrue:
```

(b)

```
        movq %rdi, %rax
        movslq %ebx, %rcx
        movsbl (%rax,%rcx), %edx
        cmpl $32, %edx    # ' '
        je Lfalse
        cmpl $9, %edx    #'\t'
        je Lfalse
        cmpl $10, %edx    # '\n'
        je Lfalse
Ltrue:
```

(c)

```
        movq %r11, %rax
        cmpq %r12, %rax
        jb Lfalse
        movq %r12, %rcx
        addq $ALLOCSIZE, %rcx
        cmpq %rcx, %rax
        jae Lfalse
Ltrue:
```

(d)

```
        movq %r8, %rax
        movq %r8, %rcx
        cmpq (%rcx), %rax
        jb Lfalse
        movq %r9, %rax
        cmpq %r8, %rax
        ja Ltrue
        movq %r9, %rax
        movq %r8, %rcx
        cmpq (%rcx), %rax
        jae Lfalse
Ltrue:
```

**Problem 2.** *Stack frame for a complicated procedure.*
In this problem, we will examine the issue of creating stack frames for a complicated C procedure, that is intended to model default arguments or variadic functions that you may have seen in languages such as C++, Java, and Python.

```
extern int _foo1(int a, int b, int c, int d, bool w, bool x, bool y, bool z);
extern int _foo2(int a, int b, int c, int d, bool w, bool x);
extern int _foo3(int a, int b, int c, int d);
```

```
int foo(int argc, int argv[]) {
  switch (argc) {
    case 4: return _foo3(argv[0], argv[1], argv[2], argv[3]);
    case 6: return _foo2(argv[0], argv[1], argv[2], argv[3], argv[4], argv[5]);
    case 8: return _foo1(argv[0], argv[1], argv[2], argv[3], argv[4], argv[5], argv[6], argv[7]);
    default: return -1;
  }
}
```

(a) Show the layout of the stack frame for `foo`. Indicate each of the four areas of the stack frame, how much each area takes, and the total size of the stack frame.

(b) Generate x86-64 assembly code for `foo`, following the code generation templates discussed earlier. Remember that you do not know the internal structure of the procedures `_foo1`, `_foo2`, and `_foo3`.

(c) How can you make the generated code more compact?

**Solution:**

(a) There shouldn't be any callee-saved registers to worry about. Since there's one call and an immediate return, there aren't any caller-saved registers either. There will be a local variable needed for doing the bounds-checking on `argc` for the `switch` statement. You will need to allocate two slots in the argument build area, because you may need to pass eight parameters in one case. This should give you enough information to lay out the stack frame.

(b) Left as an exercise for the student. You will definitely want to use a jump table to implement the `switch` statement (something we haven't discussed in this course).

(c) I can think of two possible optimizations.

- Since `argv[0]` through `argv[3]` have to be loaded into the same registers in all three non-default cases, you could pull that code out of the case arms into a common code sequence.
- The other optimization would be to create a "procedure table" analogous to the jump table that we use for the switch statement, and to use the indirect form of the `callq` instruction to call the appropriate routine after loading all of the arguments in their correct locations.

**Problem 3.** *Register allocation, straight-line code.*
Perform register allocation by graph coloring for the following code.

```
x = 2;
y = 4;
w = x + y;
z = x + 1;
u = x * y;
x = z * 2;
```

(a) Rewrite it with symbolic registers substituted for the variables.

(b) Draw the interference graph for the rewritten code.

(c) Show an allocation for it with three registers, assuming that variables `y` and `w` are dead on exit from this code.

**Solution:** I will write up and post this solution separately during the week.

**Problem 4.** *Register allocation with control flow.*
Perform register allocation by graph coloring for the following program.

```
a = 2;
b = 3;
d = c;
e = a;
g = c + 1;
if (a < d) {
  do {
    b = b + 1;
    d = 2 * d;
  } while (b < 10);
 } else {
   d = d+1;
   f = a + b;
   g = e + g;
 }
 print(b, d, e, g);
```

(a) Draw the control flow graph for this program.

(b) Rewrite it with symbolic registers substituted for the variables.

(c) Draw the interference graph for the rewritten code.

(d) Show an allocation for it with five registers. (You will need to perform coalescing.)

**Solution:** I will write up and post this solution separately during the week.