CSci Spring 2020 Section 010 Problem Set 3 Solutions

Problem 1

First, let's walk through the effects of the math in the assembly code. We're looking to understand the computations in terms of the parameters i, j, and k:

```
access_multi:                               rdi = i, rsi = j, rdx = k
        imulq   $3360, %rdi, %rax           rax = 3360*i
        imulq   $280, %rsi, %rcx            rcx = 280*j
        addq    %rax, %rcx                  rcx = 3360*i + 280*j
        leaq    (%rcx,%rdx,8), %rsi         rsi = 3360*i + 280*j + 8*k
        sarq    $3, %rsi                    rsi = 420*i + 35*j + k
        movq    $-1, %rax
        cmpq    $8399, %rsi
        ja      .LBB0_2                     is rsi >= 8400?
        movq    multi(%rcx,%rdx,8), %rax    multi + 3360*i + 280*j + 8*k
.LBB0_2:
        retq
```

Each row is C elements, each two-dimensional "plane" is $B \cdot C$ elements, and the whole array is $A \cdot B \cdot C$ elements; the elements are longs, so the sizes in bytes are 8 times each of those. If m is short for the starting address of the array, the (i, j, k)th element will be at the address $m + 8 \cdot (B \cdot C \cdot i + C \cdot j + k)$. Comparing this expression with the effective address of the final `movq`, we see that $8 \cdot C$ is 280, so C is 35. $8 \cdot B \cdot C$ is 3360, so $B \cdot C$ is 420, so B must be 12. The above expression didn't include A, but there's an overflow check that compares the effective address to the size of the array which does depend on A. The source code wrote the comparison with >= but the compiled code uses >, so the bound has been adjusted by 1. Interestingly the code does most of the computation including the factor of 8, but then takes it out with the right shift by 3, so the size of the array in longs is $8399 + 1 = 8400$. We can confirm that that is a multiple of 420, specifically $8400 = 20 \cdot 12 \cdot 35$, so A is 20.

Problem 2

There are a number of possible solutions, but the computation is complicated enough that it's helpful to think about reusable patterns or abstractions. One of the most useful is the two-argument functions that compute either the smaller or the larger of their two arguments, which are commonly called `min` (from minimum) and `max` (from maximum).

To help illustrate what's going on, we'll show C code examples and then the corresponding Y86-64 examples. First off is min and max, which are often written a C preprocessor macros:

```
#define MIN(a, b) ((a) < (b) ? (a) : (b))
#define MAX(a, b) ((a) > (b) ? (a) : (b))
```

Implementing these functions with conditional moves illustrates the core technique for solving this problem.

```
min:
        rrmovq  %rdi, %rax      # default: min(a, b) = a
        rrmovq  %rdi, %rcx
```

```
        subq    %rsi, %rcx
        cmovg   %rsi, %rax      # if a > b, min(a, b) = b
        ret

max:
        rrmovq  %rdi, %rax      # default: max(a, b) = a
        rrmovq  %rdi, %rcx
        subq    %rsi, %rcx
        cmovl   %rsi, %rax      # if a < b, max(a, b) = b
        ret
```

Remember that Y86-64 doesn't have a `cmp` instruction for comparing numbers. It does have the `sub` instruction that subtracts two numbers and sets the condition code flags based on the comparison, but `sub` also computes the difference in its destination register. Mostly in this question we don't care about the value of the difference, but we need to compute the difference in a different register to be able to reuse the values we were comparing. These implementations use `rcx` as that scratch register. Following the standard calling conventions, the two arguments `a` and `b` are in `rdi` and `rsi` respectively, and the result is returned via `rax`. The basic structure is that we start off with a copy of `a` in `rax`, and then conditonally replace it with `b` if the result should be `b` instead. Looking at `min` in detail, we compute $a - b$ in `rcx`, which also sets the flags representing the comparison result of a versus b. If $a > b$, then the minimum needs to be b, but otherwise (including if they're equal) it can be a, so that's why we use `g` as the condition for the conditional move. (You can also think of `g` as representing that the difference $a - b$ is greater than 0, but it's important that it is based on the sign of the full difference not counting overflow.) After you understand `min`, `max` is the same except with the condition flipped from greater-than to less-than. This is the analogous kind of flip to the change we made in the C source versions, but the assembly versions are opposite because we're giving the condition for the result to be b whereas the C gives the condition for it to be a. There's a distinction without a difference about what the code does when $a = b$. For instance we could have used `ge` as the condition in place of `g`, but it doesn't matter which one you return in that case because they're equal.

A related and more powerful abstraction that might be familiar to you from sorting algorithms is to examine two values, determine which is larger and which is smaller, and put them in a fixed order by interchanging (swapping) them if they start out in the reverse order. We'll refer to this abstraction as "compare-and-swap". Because it needs to modify two locations, in C or the standard calling conventions the arguments need to be passed via pointers. One easy way to implement compare-and-swap is not to use swapping directly, but to use `min` and `max`. Here's a C version of that approach:

```
void compare_and_swap(long *p, long *q) {
    long min = MIN(*p, *q);
    long max = MAX(*p, *q);
    *p = min;
    *q = max;
}
```

This approach translates directly into Y86-64:

```
compare_and_swap_call:
        pushq   %r12
        pushq   %r13
        pushq   %r14
        rrmovq  %rdi, %r12      # r12 is a preserved copy of p
```

```
        rrmovq  %rsi, %r13      # r13 is a preserved copy of q
        mrmovq  (%r12), %rdi
        mrmovq  (%r13), %rsi
        call    min
        rrmovq  %rax, %r14      # r14 is min(*p, *q)
        mrmovq  (%r12), %rdi
        mrmovq  (%r13), %rsi
        call    max             # rax ia max(*p, *q)
        rmmovq  %r14, (%r12)    # *p = min
        rmmovq  %rax, (%r13)    # *q = max
        popq    %r14
        popq    %r13
        popq    %r12
        ret
```

Note how we used the preserved registers `r12` through `r14` to store copies of the arguments, since the argument registers might be modified by `min` and `max`. Even though the versions of `min` and `max` we showed above don't modify `rdi` and `rsi`, following the standard calling conventions avoids trouble if you change implementations later.

Using separate functions helps break down the complexity into more manageable pieces, but it doesn't always lead to the shortest solution, since there are more instructions moving data around. You can get a shorter implementation of this function if you use the compare-and-swap concept more directly, and do the swapping within the registers of a function:

```
compare_and_swap_inline:
        mrmovq  (%rdi), %r8     # r8 = *p
        mrmovq  (%rsi), %r9     # r9 = *q
        rrmovq  %r8, %rax
        subq    %r9, %rax
        cmovg   %r8, %rax
        cmovg   %r9, %r8
        cmovg   %rax, %r9       # swap r8 with r9 (via rax) if r8 > r9
        rmmovq  %r8, (%rdi)     # *p = min(r8, r9)
        rmmovq  %r9, (%rsi)     # *q = max(r8, r9)
        ret
```

Notice that there are three `cmovg` instructions in a row. All these are conditional on the same comparison (subtraction). The flags are like other registers in that they keep their values until replaced by another instruction, and `cmov` itself doesn't change the flags, so you can conditionally control several moves in a row. Implementing swapping in terms of copying takes three copies and a temporary location: when the condition is true, the three `cmovg` instructions swap `r8` with `r9` using `rax` for temporary storage. The swap happens if `r8` was bigger than `r9`, so after the swap `r8` is always less than or equal to `r9`.

The first complete implementation of the median-of-3 we'll show will look familiar if you remember bubble sort. We use compare-and-swap operations to put the three values in order, and then choose the middle one. Bubble sorting three elements takes only two passes with a total of three compare and swap operations. The C code shows the structure clearly:

```
long median1(long a, long b, long c) {
    compare_and_swap(&a, &b);
```

```
    compare_and_swap(&b, &c);
    compare_and_swap(&a, &b);
    return b;
}
```

Note that the three calls to `compare_and_swap` do not take all three pairs of variables. Instead the first two are like a first pass of bubble sort that moves that maximum value into the `c` position, and then the final compare-and-swap is the second pass that makes sure the two elements that are not the maximum are correctly sorted. (Some but not all other sequences of three calls would work equally well.)

Translating the compare-and-swap via pointers into Y86-64 is a little less convenient because of the register-based calling convention: you have to copy the arguments to the stack to be able to pass pointers to them. And constructing the stack addresses is slightly inconvenient because Y86-64 doesn't have an `lea` instruction. However the basic structure is straightforward: allocate 24 bytes on the stack, copy the arguments there, call `compare_and_swap` with three different sets of pointers, then copy the middle value to `rax` to return it:

```
median1_call:
        irmovq  $24, %rcx
        subq    %rcx, %rsp
        rmmovq  %rdi, 0(%rsp)
        rmmovq  %rsi, 8(%rsp)
        rmmovq  %rdx, 16(%rsp)
        rrmovq  %rsp, %rdi
        rrmovq  %rsp, %rsi
        irmovq  $8, %rcx
        addq    %rcx, %rsi
        call    compare_and_swap        # compare_and_swap(0+rsp, 8+rsp)
        rrmovq  %rsp, %rdi
        irmovq  $8, %rcx
        addq    %rcx, %rdi
        rrmovq  %rdi, %rsi
        addq    %rcx, %rsi
        call    compare_and_swap        # compare_and_swap(8+rsp, 16+rsp)
        rrmovq  %rsp, %rdi
        rrmovq  %rsp, %rsi
        irmovq  $8, %rcx
        addq    %rcx, %rsi
        call    compare_and_swap        # compare_and_swap(0+rsp, 8+rsp)
        mrmovq  8(%rsp), %rax           # return 8(%rsp)
        irmovq  $24, %rcx
        addq    %rcx, %rsp
        ret
```

You can get a shorter implementation of the same strategy by copying the code sequence from the direct compare-and-swap above, three times. Then no other data movement is needed except copying `rsi` to `rax` at the end:

```
median1_inline:
        rrmovq  %rdi, %rax
```

```
        subq    %rsi, %rax
        cmovg   %rdi, %rax
        cmovg   %rsi, %rdi
        cmovg   %rax, %rsi      # Compare and swap %rdi with %rsi
        rrmovq  %rsi, %rax
        subq    %rdx, %rax
        cmovg   %rsi, %rax
        cmovg   %rdx, %rsi
        cmovg   %rax, %rdx      # Compare and swap %rsi with %rdx
        rrmovq  %rdi, %rax
        subq    %rsi, %rax
        cmovg   %rdi, %rax
        cmovg   %rsi, %rdi
        cmovg   %rax, %rsi      # Compare and swap %rdi with %rsi
        rrmovq  %rsi, %rax      # return %rsi
        ret
```

A second implementation strategy, which gives efficient code when implemented using conditional jumps, is to divide up the different cases in a tree-like structure until the identity of the median is discovered. This a good general divide-and-conquer (at coding time) way to implement any kind of complex condition. If you start, without loss of generality, by comparing `a` with `b`, it then takes either one or two more comparisons to locate `c` into one of the three ranges between `a` and `b`. In C, this looks like:

```
long median2(long a, long b, long c) {
    if (a <= b) {
        if (c <= a)
            return a;
        else if (c <= b)
            return c;
        else
            return b;
    } else {
        /* b < a */
        if (c <= b)
            return b;
        else if (c <= a)
            return c;
        else
            return a;
    }
}
```

This structure is less advantageous when translated into conditional moves, because there you always have to do all of the comparisons. But the translation can be done systematically. To translate a chain of if-then-else statements into conditional moves, you can reverse the order by initializing a register with the else value first, and conditionally moving values to the same register based on the other conditions going in reverse order. Reversing the order gives the right behavior because an if-then-else chain stops when it reaches the first condition that is true. In a sequence of conditional moves, the execution never stops early,

but if more than one condition is true, the one that is evaluated last will take precedence. We can get the effect of nested if-then-else chains by evaluating one chain into a separate register. In the code below, the answer for the case $a \leq b$ is evaluated into `r8`, while the $a > b$ is evaluated into `rax`, and then we use a final conditional move to decide whether to move `r8` into `rax`:

```
median2:
        rrmovq  %rsi, %r8
        rrmovq  %rdx, %rcx
        subq    %rsi, %rcx
        cmovle  %rdx, %r8       # if (c <= b) r8 = c;
        rrmovq  %rdx, %rcx
        subq    %rdi, %rcx
        cmovle  %rdi, %r8       # if (c <= a) r8 = a;
        rrmovq  %rdi, %rax
        rrmovq  %rdx, %rcx
        subq    %rdi, %rcx
        cmovle  %rdx, %rax      # if (c <= a) rax = c;
        rrmovq  %rdx, %rcx
        subq    %rsi, %rcx
        cmovle  %rsi, %rax      # if (c <= b) rax = b;
        rrmovq  %rdi, %rcx
        subq    %rsi, %rcx
        cmovle  %r8, %rax       # if (a <= b) rax = r8;
        ret
```

Next let's consider a couple of different ways that you can build a median out of `min` and `max`. One clever idea comes from the fact that the median is value that is left over after you remove the smallest value and the largest value from among the three. You might informally call this a "set" approach, but if you think about what needs to happen when there are duplicate elements, the right abstraction is actually what mathematicians call a multiset, where duplicate elements count more than once. A sequence of multiset operations that ends with just one result can be implemented with an arithmetic operator that has the right properties of being associative, commutative, and having an inverse. Probably the easiest one to think about is addition and subtraction. If you add together the three elements, then subtract out the smallest element and the largest element, you'll be left with the median element. You might want to think through what happens if there are duplicate elements or if overflow occurs, but as it turns out everything works out fine. (And in fact XOR can be used too.)

We've already discussed how to compute the smallest of two elements; computing the smallest of a bigger set can be done just by repeating the operation. This leads to a short C implementation:

```
long median3(long a, long b, long c) {
    long min = MIN(a, MIN(b, c));
    long max = MAX(a, MAX(b, c));

    return a + b + c - min - max;
}
```

This C implementation also translates fairly directly into Y86-64 code that calls the previously-defined `min` and `max` functions, though with a number of instructions just to move data around:

```
median3_call:
        pushq   %r12
        pushq   %r13
        pushq   %r14
        pushq   %rbx
        rrmovq  %rdi, %r12      # %r12 is a preserved copy of a
        rrmovq  %rsi, %r13      # %r13 is a preserved copy of b
        rrmovq  %rdx, %r14      # %r14 is a preserved copy of c
        call    min             # rax = min(a, b)
        rrmovq  %rax, %rdi
        rrmovq  %r14, %rsi
        call    min
        rrmovq  %rax, %rbx      # rbx = min(a, b, c)
        rrmovq  %r12, %rdi
        rrmovq  %r13, %rsi
        call    max             # rax = max(a, b)
        rrmovq  %rax, %rdi
        rrmovq  %r14, %rsi
        call    max             # rax = max(a, b, c)
        rrmovq  %r12, %rcx
        addq    %r13, %rcx
        addq    %r14, %rcx      # rcx = a + b + c
        subq    %rbx, %rcx
        subq    %rax, %rcx
        rrmovq  %rcx, %rax      # rax = a + b + c - min - max
        popq    %rbx
        popq    %r14
        popq    %r13
        popq    %r12
        ret
```

As we saw before, inlining the minimum and maximum computations leads to a shorter version of the solution:

```
median3_inline:
        rrmovq  %rdi, %r8       # min = a
        rrmovq  %r8, %rax
        subq    %rsi, %rax
        cmovg   %rsi, %r8       # if a > b, min = b
        rrmovq  %r8, %rax
        subq    %rdx, %rax
        cmovg   %rdx, %r8       # if min > c, min = c
        rrmovq  %rdi, %r9       # max = a
        rrmovq  %r9, %rax
        subq    %rsi, %rax
        cmovl   %rsi, %r9       # if a < b, max = b
        rrmovq  %r9, %rax
        subq    %rdx, %rax
```

```
        cmovl   %rdx, %r9       # if max < c, max = c
        rrmovq  %rdi, %rax
        addq    %rsi, %rax
        addq    %rdx, %rax      # rax = a + b + c
        subq    %r8, %rax
        subq    %r9, %rax       # rax = a + b + c - min - max
        ret
```

Another intuition that leads to an implementation in terms of minimum and maximum is based on an operation that in signal processing and graphics is called "clamping". Clamping enforces both a lower bound and an upper bound on a value, so for instance if an integer is clamped between 0 and 255, any negative value will be changed to 0, any value greater than 255 will be changed to 255, and values between 0 and 255 will be unchanged. Enforcing a lower bound is the same as computing a maximum between the varying value and the lower bound, while dually enforcing an upper bound can be done as computing a minimum. You can combine a minimum and a maximum to implement clamping. On the other hand, the median of three values can also be implemented as a kind of clamping. Going with the intuition of the median being in the middle when the values are sorted, the median can also be implemented by clamping one of the values (say c) between the other two (a and b: this gives c if c is between a and b, the lower bound if c is too low, or the upper bound if c is too high. However a and b have to be put in the right order (we need to know which one is the upper bound and which one is the lower bound) for this to work; we've seen before we can do that with `min` and `max`. Putting these ideas together gives this C implementation, where the last line is the clamping:

```
long median4(long a, long b, long c) {
    long min_ab = MIN(a, b);
    long max_ab = MAX(a, b);
    return MAX(min_ab, MIN(max_ab, c));
}
```

As with the last approach, this can be translated either more directly using calls to `min` and `max`, or by inlining the minimum and maximum calculations. Here's the version with calls:

```
median4_call:
        pushq   %r12
        pushq   %r13
        pushq   %r14
        pushq   %rbx
        rrmovq  %rdi, %r12      # %r12 is a preserved copy of a
        rrmovq  %rsi, %r13      # %r13 is a preserved copy of b
        rrmovq  %rdx, %r14      # %r14 is a preserved copy of c
        call    min
        rrmovq  %rax, %rbx      # rbx = min(a, b)
        rrmovq  %r12, %rdi
        rrmovq  %r13, %rsi
        call    max
        rrmovq  %rax, %rdi      # rdi = max(a, b)
        rrmovq  %r14, %rsi
        call    min
        rrmovq  %rax, %rsi      # rsi = min(max(a, b), c)
```

```
        rrmovq  %rbx, %rdi
        call    max             # return max(min(a, b), min(max(a, b), c))
        popq    %rbx
        popq    %r14
        popq    %r13
        popq    %r12
        ret
```

And here's the version with the computations inline:

```
median4_inline:
        rrmovq  %rdi, %rcx
        rrmovq  %rdi, %r8
        subq    %rsi, %rcx
        cmovg   %rsi, %r8       # r8 = min(a, b)
        rrmovq  %rdi, %rcx
        rrmovq  %rdi, %r9
        subq    %rsi, %rcx
        cmovl   %rsi, %r9       # r9 = max(a, b)
        rrmovq  %r9, %rcx
        rrmovq  %r9, %rax
        subq    %rdx, %rcx
        cmovg   %rdx, %rax      # rax = min(r9, c)
        rrmovq  %rax, %rcx
        # rrmovq %rax, %rax     # from pattern, but skip because it's a no-op
        subq    %r8, %rcx
        cmovl   %r8, %rax       # rax = max(r8, rax)
        ret
```

You can see that this code consists entirely of four copies of the `min` and `max` computation sequences (in the order min, max, min, max), just over different variables. We've commented out one move of `rax` to itself which would be produced by following the pattern, since obviously that operation has no effect.

Last but not least, another generic approach to writing the median as a conditional is to group together all the cases under which the median would be a, and all the cases when it would be b; then in any remaining cases it would be c. A somewhat related practice problem in the textbook asks you to write the median of three values in HCL, and the textbook's solution takes this approach. Translating the textbook's answer from HCL into C gives:

```
long median5(long a, long b, long c) {
    if (a <= b && b <= c)
        return b;
    else if (c <= b && b <= a)
        return b;
    else if (b <= a && a <= c)
        return a;
    else if (c <= a && a <= b)
        return a;
    else
        return c;
```

```
}
```

Because of the complex and repetitive conditions, this is not the shortest implementation to translate into conditional moves, but it can be done with the same basic structure of converting an if-then-else chain backwards that we deployed before. The one thing that complicates things is those logical AND operators. One way to compile an AND operation that makes sense given its name is to represent true-false values as 0 or 1 (on x86-64 you could do this with `setCC`, or on Y86-64 you could use `irmovq` and `cmov`), and then to compute the AND with `andq`. That approach would be pretty cumbersome, though.

Another approach that works a bit more easily is related to the more common way of compiling logical AND: to use nested branches. In C, the branch `if (A && B) C; else D;` is equivalent to the nested branches `if (A) { if (B) C; else D; } else D;`. A compiler would commonly compile such an if statement into two conditional branches, with just one copy of the `D` code. You can use a similar approach to translate an AND condition into two conditional moves.

Another intuition is that if you have a conditional move from X to Y, followed by a conditional move from Y to Z, the net effect will only copy the value from X to Z if both the conditions are true. In translating this idea into conditional moves between registers, we initialize Y and Z with the old value of Z, and put the potential new value of Z in X; then Z will be updated to X only if both conditional moves happen. In the other three combinations, Z will either never be modified, or it will get copied from Y without Y changed, so it will stay unchanged.

This leads to a repetitive code structure:

```
median5:
        rrmovq  %rdx, %rax
        rrmovq  %rax, %r8
        rrmovq  %rdx, %rcx
        subq    %rdi, %rcx
        cmovle  %rdi, %r8       # if (c <= a) r8 = a;
        rrmovq  %rdi, %rcx
        subq    %rsi, %rcx
        cmovle  %r8, %rax       # if (c <= a && a <= b) rax = a;
        rrmovq  %rax, %r8
        rrmovq  %rsi, %rcx
        subq    %rdi, %rcx
        cmovle  %rdi, %r8       # if (b <= a) r8 = a;
        rrmovq  %rdi, %rcx
        subq    %rdx, %rcx
        cmovle  %r8, %rax       # if (b <= a && a <= c) rax = a;
        rrmovq  %rax, %r8
        rrmovq  %rdx, %rcx
        subq    %rsi, %rcx
        cmovle  %rsi, %r8       # if (c <= b) r8 = b;
        rrmovq  %rsi, %rcx
        subq    %rdi, %rcx
        cmovle  %r8, %rax       # if (c <= b && b <= a) rax = b;
        rrmovq  %rax, %r8
        rrmovq  %rdi, %rcx
        subq    %rsi, %rcx
        cmovle  %rsi, %r8       # if (a <= b) r8 = b;
```

```
        rrmovq  %rsi, %rcx
        subq    %rdx, %rcx
        cmovle  %r8, %rax       # if (a <= b && b <= c) rax = b;
        ret
```

In this code, `rax` plays the role described as Z above, the location that is getting conditionally updated. It starts out as `c`, under some conditions it is updated to `a` or `b`, and then it is the return value. Register `r8` plays the role of Y, and `rcx` is just used for subtraction.

Problem 3

You should recall here the discussion of signed overflow in twos-complement from earlier in the class. A sum that is the same size as the arguments can't always reflect the correct value of a sum if it's outside the range of representable numbers. The sum is always representable if the two numbers being added have different signs; overflow is possible if they have the same sum. The correct sum of two non-negative numbers is always non-negative, but if the sum of two positive numbers appears to be negative, that's positive overflow. Conversely the correct sum of two negative numbers is always negative, but if the sum of two negative numbers appears to be non-negative, that's negative overflow. In defining the boundary conditions you should be careful about 0. Adding 0 never overflows, and $0 + 0 = 0$ is not an overflow, but TMin + TMin $= 0$ is negative overflow.

In the HCL case notation, those conditions can be written:

```
bool overflow = [
    A >= 0 && B >= 0 && S <  0: 1;
    A <  0 && B <  0 && S >= 0: 1;
    1 : 0;
];
```

That's easiest to read, and we would accept it, but HCL case notation is only used for word values and not for Boolean values, so technically the two overflow cases should be combined with logical OR instead:

```
bool overflow =
    (A >= 0 && B >= 0 && S <  0) ||
    (A <  0 && B <  0 && S >= 0);
```

Problem 4

There are 2 set index and 2 byte offset bits each. The remaining bits are for the tag.

T	T	T	T	T	T	T	T	I	I	O	O
11	10	9	8	7	6	5	4	3	2	1	0

Using the above format we can extract the bits identifying the tag, set index, and line offset. With these we can identify the cache line and inspect the tags and valid bits to determine if there was a hit.

Address	T	I	O	Effect
0x5AA	5A	2	2	Miss, tag match but invalid
0xA62	A6	0	2	Hit, read byte 2 yields 0xA0
0x7A7	7A	1	3	Miss, tag match but invalid
0xDAC	DA	3	0	Hit, read byte 0 yields 0x8A