CSCI 3240: Project 2

Brent Yelle

March 3, 2023

General Notes

Since this assembly code makes use of `r`-registers, we can safely assume this is a 64-bit program and thus that all instances of `long` refer to 8-byte (qword) integers.

I also assume that for all arguments passed via 32-bit registers, the high 32 bits of their corresponding 64-bit registers have been properly zero-extended.

Mystery Function 1

We are provided with the declaration:

```
long MysteryFunction1(long a, int b);
```

By convention, `rdi` must contain the value of `a` (qword, 8 bytes), and `esi` must contain the value of `b` (dword, 4 bytes). Because the return type is a `long` (qword, 8 bytes), the return value must be in `rax`.

Execution Steps

We begin by copying `1` into `eax` (which also zeroes-out the whole `rax` register). This is the initialization of the variable that is to be returned, which I will call `product`; thus, this represents a declaration `long product = 0;`. (Despite this instruction using `eax` and not `rax`, we know that the return type will need all of `rax` because the function declaration specifies `long` as the return type.)

This is followed by an unconditional jump to `.L2`. We then evaluate `b` (in `rsi`) minus 1 and copy that difference `b-1` into `edx`[1]. Then, it is checked whether `b` (in `esi`[2], not the `b-1` in `edx`) is > 0. If it is, then we loop back

[1] Although this would cause an "underflow" for `b` = T_{min}, since the resulting value `0x0000000100000000` would be trimmed to `0x00000000` when stored in `edx`, a bug is avoided here due to the following conditional jump catching all non-positive `b`.

[2] This also narrowly avoids a bug, since if `rsi` (not `esi`) were invoked again here, then low-magnitude negative numbers in `esi` would be interpreted as very large positive numbers in `rsi`, e.g. `0xFFFFFFFF` would be -1 in `esi`, while the zero-extended `0x00000000FFFFFFFF` would be $+4,294,967,295 = 2^{32} - 1$ in `rsi`.

to `.L3`, which will ultimately lead back into `.L2`–this can be interpreted as `while (b > 0) { ... }`.

If the condition is not fulfilled (i.e., once `b > 0` is no longer true), then we return the current value of `rax`, which is `1` by default. This can be interpreted as a final `return product;` after the conclusion of the `while` loop.

As for the interior of the `while` loop, we follow the jump up to `.L3`, where the first step is to multiply `product` (in `rax`, starts as 1) by `a` (in `rdi`), storing the result in the former. This is quite simply `product *= a;`.

Next, we copy the contents of `edx` (`b-1`) into `esi`, overwriting its previous value of `b`; this is `b--;`. Finally, we return back to `.L2`, where we again set up to decrement `b` (in `esi`) by first copying `b-1` into `edx`. We then test whether `b > 0`, repeating the loop if true, or breaking the loop if false. Note that `b` always decrements permanently *just before* the test case of every non-initial pass through the loop, which indicates that the `b--;` is located at the end of the while loop.

"Decompiled" C Code & Interpretation

This function's purpose is to *calculate the value of* a^b:

```
long MysteryFunction1(long a, int b) {
    long product = 1;
    while (b > 0) {
        product *= a;
        b--;
    }
    return product;
}
```

When `b` > 0, or when `a` $\neq 0$ and `b` $= 0$, the return value will be equal to a^b (barring any overflows). In all other cases, it will return 1.

Mystery Function 2

We are provided with the declaration:

```
unsigned int MysteryFunction2(unsigned int num);
```

By convention, `edi` must contain the value of `num` (dword, 4 bytes). Because the return type is an `unsigned int`, the return value must be in `eax`.

Execution Steps

The function immediately begins by initializing the values in both `edx` and `ecx` to `0`; these are variables that are used during the function's execution, and I will name them `reversed` and `place`, respectively. Therefore, these represent declarations `unsigned int place = 0;` and `unsigned int reversed = 0;`.

Although it is out-of-order when reading the assembly code, we also see that later in `.L8`, the exact same kind of initialization occurs for `eax`, which is assigned the value `1` there and then further manipulated; I call this value `flipper`. This too is a declaration: `unsigned int flipper = 0;`. Since common practice when writing C would be to have declarations at the top of the function, I have mentioned it here.

Following the declarations of `reversed` and `place` (both initially 0), the function jumps to `.L5`, where there is a further jump to `.L8` if `place > 31`. Within `.L8`, the program returns `reversed`, and this is in fact the only way to possibly escape from the loop. This therefore indicates a conditional loop: `while (place <= 31);`, or to use a nice power of 2, `while (place < 32) { ... };`.

Assuming the loop is entered successfully (which will always be the case at first, since `place` is initialized to `0`, then we continue by (re-)initializing `flipper` to `1` – which is importantly `00000000000000000000000000000001` in binary: `flipper = 1;`. We then shift `flipper` to the left by `place`: `flipper = flipper << place;`. (Recall that `place` can never be above `31` due to the conditions of the loop.)

The bits of `num` (in `edi`) are then compared with this shifted value of `flipper` (in `eax`). If the bits are the same, then we set `flipper` equal to -2147483648, which is `0x80000000` in hexadecimal and `10000000000000000000000000000000` in binary: `flipper = 0x80000000;`. We then shift this right by `place` (padding with zeroes): `flipper = flipper >> place;`. The effect of this is that if there was a match with a bit in `num`, then `flipper` is bitwise *reversed*. After being reversed, the reversed-matching bit is then flipped to `1` in `reversed`.

Finally, `place` is incremented by `1`: `place++`. The loop will continue over and over again until `place` reaches `32`, at which point it will break, and `reversed` will be moved into `eax` to be returned: `return reversed;`

"Decompiled" C Code & Interpretation

This function's purpose is to *reverse the order of the bits of* `num`:

```
unsigned int MysteryFunction2(unsigned int num) {
    unsigned int flipper=1;
    unsigned int place=0;
    unsigned int reversed=0;
    while (place < 32) {
        flipper = 1;
        flipper = flipper << place;
```

```
        if (flipper & num) {
            flipper = 0x80000000;
            flipper = flipper >> place;
            reversed |= flipper;
        }
        place++;
    }
    return reversed;
}
```

The names that I have given to the local variables `flipper`, `place`, and `reversed` are intended to be indicative of their function:

The variable `reversed` starts as all `0`s, and its bits are flipped to `1` (using OR) in the opposite positions to where `num`'s are `1`s.

The variable `place` keeps track of which bit of `num` the current iteration is checking, counting from the rightmost (least significant) bit; it ranges from 0 to 31, and if it exceeds 31, then the loop `break`s, since `num` is at most 32 bits long. Every loop, it increments by 1, moving right-to-left along the binary representation of `num`.

Lastly, the variable `flipper` is the tool by which the bits in `num` are checked (with AND) and the bits in `reversed` are changed to `1` (with OR). Every loop, it is reset to `0x00000001` (31 zeroes followed by a `1` in bindary) and then shifted into the correct place using `place`. Then, the `if`-condition `(flipper & num)` triggers if the corresponding bit in `num` is also `1`, and if successful, then `flipper` is set to `0x80000000` (a `1` followed by 31 zeroes in binary) and shifted right into the reverse position. Finally, `reversed |= flipper` sets the reversed bit in `reversed` equal to 1.

Mystery Function 3

We are provided with the declaration:

```
long MysteryFunction3(long *a, int n)
```

By convention, `rdi` must contain a pointer `a` (qword, 8 bytes) to the memory address that stores one or more `long` integers (qwords, 8 bytes each), and `esi` must contain the value of `n` (dword, 4 bytes). Because the return type is a `long` (qword, 8 bytes), the return value must be in `rax`.

Furthermore, since we later use parentheses-operations to dereference different offsets around `rdi`, we can infer that `a` is in fact a pointer to an *array* of `long`s.

Execution Steps

We begin by dereferencing our first pointer `*a`, giving us the `long` value `a[0]` and storing it in `rcx`. This register `rcx` will continually be updated with new values and ultimately copied to `rax` for returning, meaning that it holds an important value: I will call this value `maximum`. This is therefore a declaration `long maximum = a[0];`.

We also set a value in `eax` to be `1` before entering a loop (by jumping to `.L10`). This value `eax` is overwritten as soon as we exit the loop (at `.L14`), meaning that it is only valuable within the loop; this tips us off that it must be some kind of iterator or counter, which I will label `i`. This, as well as the fact that it is incremented by `1` every time we go back to the start of the loop *except* for the first loop, tells me that we are dealing with a `for`-loop: `for (int i=1; ?????; i++)`, with the condition to be determined next:

The condition to exit the loop (by jumping to `.L14`) is for `i >= n`, meaning that the loop will only continue so long as `i < n`. Therefore, we can complete our loop statement: `for (int i = 1; i < n; i++)`. This also tips us off that `n` likely refers to the length of the array `a`... and that guess will be proven correct immediately.

Next, the subsequent statements grab the pointer `a`, then get an offset of `8*i` and dereference it; since `a` is an array of `longs` (8 bytes each), this is precisely the same as doing `a[i]` in C. This new value `a[i]` is compared against `maximum`, and `a[i]` is smaller or equal, then we just continue to the next iteration of the loop, but otherwise–i.e., if `a[i]` is bigger than `maximum`–the value of `maximum` is overridden with the value of `a[i]`. This is a simple `if` statement: `if (a[i] > maximum) { maximum = a[i]; }`.

Finally, the loop resets, jumping back up to `.L11`, where `i` is incremented again (as discussed before). Once the loop is finally ended, `maximum` is copied into `rax` in order to be returned: `return maximum;`.

"Decompiled" C Code & Interpretation

This function's purpose is to *find the maximum value in a given array* `a[]` *of length* `n`:

```
long MysteryFunction3(long *a, int n) {
    long maximum = a[0];
    for (int i = 1; i < n; i++) {
        if (a[i] > maximum)
            maximum = a[i];
    }
    return maximum;
}
```

The declaration `long maximum = a[0];` starts with `a[0]` as the first guess for the maximum, and then moves along the whole array, comparing each `a[i]`

with the `maximum` and updating the latter until it is the true maximum of the array.

Mystery Function 4

We are provided with the declaration:

```
int MysteryFunction4(unsigned long n);
```

By convention, `rdi` must contain the value of `num` (qword, 8 bytes). Because the return type is an `int` (dword, 4 bytes), the return value must be in `eax`.

Execution Steps

We begin by copying 0 into `eax` (which also zeroes-out the whole `rax` register). This is the initialization of the variable that is to be returned, which I will call `ones`; thus, this represents a declaration `int ones = 0;`.

We then jump to `.L16`, which immediately has a test of whether `n` is zero or not. If it is zero, then we return `ones` (0 by default). If `n` isn't zero, then we jump up to `.L17`, which will eventually funnel back down into `.L16`. This indicates a loop: `while (n != 0)`, and a return statement `return ones;` after the loop's conclusion.

`.L17` contains the body of while `while` loop: The first two commands make a copy of `edi` (the lower bits of `n`, meaning `n` % 2^{32}) and perform a bitwise `and` of it and `1`, yielding (`n` % 2^{32}) % 2, which is equal to just `n` % 2 = `n & 1`–which is the rightmost (least significant) bit of `n`. This result–0 or 1–is then added to `ones` (in `eax`). This can be interpreted as `ones += (n & 1);`. Finally, `n` (in `rdi`) is shifted right by 1 (i.e., divided by 2 while discarding the remainder): `n = n >> 1;`.

The code then continues down back to `.L16`, checking again whether `n` > 0 or not, looping again if nonzero. Since we used `shr`, and `n` is an `unsigned long`, the left side of `n` will get padded with `0`s, guaranteeing that we will end up with `n` $= 0$ after at most 64 iterations of the loop.

"Decompiled" C Code & Interpretation

This function's purpose is to *count the number of* `1`*s in the binary representation of* `n`:

```
int MysteryFunction4(unsigned long n) {
    int ones = 0;
    while (n != 0) {
        ones += (n & 1);
        n = n >> 1;
```

```
    }
    return ones;
}
```

The expression `(n & 1)` can also be represented as `(n % 2)`, but if I understand correctly how `%` is implemented, then doing `(n & 1)` should be faster.

Mystery Function 5

We are provided with the declaration:

```
unsigned int MysteryFunction5(unsigned int A, unsigned int B);
```

By convention, `edi` must contain the value of `A` (dword, 4 bytes), and `esi` must contain the value of `B` (dword, 4 bytes). Because the return type is an `unsigned int` (dword, 4 bytes), the return value must be in `eax`.

Execution Steps

The first step done by the code is to perform XOR of `A` and `B`, storing it in `edi`. This technically overwrites `A`, but since we didn't need it anymore anyway, there's no harm done in that. In addition, since the result of this XOR operation will later be manipulated, it must be its own variable, which I will call `C`: `int C = A ^ B;`. Because we later do `SAR` on `C`, we know that it must be a signed `int` and not an `unsigned int`.

This is also followed by the storing of `0` in `eax`, which will later be manipulated and ultimately returned; it too must be a variable, which I'll name `ones`. Therefore, this is another declaration: `unsigned int ones = 0;`.

Following this, we then jump to `.L19` and check to see if `C == 0`. If it is zero, then we return the current value of `ones` (zero by default); otherwise we jump back up to `.L20`, and from there, we'll eventually trickle back down to test again if `C == 0`, breaking if true. This indicates a `while` loop: `while (C != 0) { ... }`.

In the body of the loop (from `.L20`, we grab just the first bit of `C` and add it to `ones`: `ones += (C & 1);`. After that, we right-shift `C` by a single position: `C = C >> 1;`. If this were a *logical* right-shift (`SHR`), then this would work perfectly. However, there is a bug in this code due to the use of an *arithmetic* shift (`SAR`) here: If `A` XOR `B` turns out to have a `1` in the leftmost (most significant) binary digit, then eventually `sarl %edi` will forever yield `edi = 0xFFFFFFFF` (where `edi` holds `C`), and so `C != 0` will always be true, meaning that the loop will never break.

Setting aside this bug, however, after the shift we move down to `.L19` again, where we perform the test `C != 0` again. If we break the loop, then we return

ones: return ones;.

"Decompiled" C Code & Interpretation

This function's purpose is to *calculate the number of* `1`*s in* `A` XOR `B`, which quantifies the number of bits that are different between `A` and `B`.

```
unsigned int MysteryFunction5(unsigned int A, unsigned int B) {
     int C = A ^ B;
     unsigned int ones = 0;
     while (C != 0) {
        ones += (C & 1);
        C = C >> 1;
     }
     return ones;
}
```