Universal Translator

Sacha El Masry

April 2, 2016

Contents

1 Introduction

Here follows an interesting and random collection of technical problems, algorithms and techniques. I have come across them in books, tutorials, journals, articles and innumerable other locations. I consider this nothing more than my *sandbox*, and after identifying important or interesting problems, I'll try to solve them in my favourite languages.

I'm not attaching any particular value or purpose to this project, it'll merely force me to think in different paradigms, to discover and overcome obstacles in a range of languages. In doing that, I hope to free my mind from a single way of thinking, to start discovering new solutions to problems, new approaches, and what each language is best suited for.

My inspiration for computer science and many of the problems in this collection, comes from the inexhaustible work of Donald Knuth, a giant in our field, George Pólya, John McCarthy, Robert Sedgewick and many more.

> To learn an algorithm well, one must implement it. Accordingly, the best strategy for understanding...is to implement and test them, experiment with variants, and try them out on real problems. – Robert Sedgewick[1]

[1]Taken from *Algorithms* (1983), Addison Wesley (p.3)

2 Methodology

Initially, I want to flex my muscle in each language; I want to concentrate on *core* functionality, not on external libraries, for as long as that makes sense. I fully expect that there are several to a dozen plausible solutions to each problem here, but will go for the most idiomatic and straight-forward implementation in each language before I explore others, if I even do that at all. I also expect to pay a penalty for doing so, in possibly coming up with the most complex or resource-hungry implementation; if and when I have time, I will come back to improve upon some of these, through additional implementations.

While I have looked at Rosetta Code from time to time, I've *never* done so to help with the solution, only to compare; it's uncanny how some solutions are exactly alike.

The languages I'll be using to write my solutions range in syntax, influence and cover at least five paradigms between them:

Language	Paradigm
Assembly	imperative
Clojure	functional
Common Lisp	multi-paradigm: procedural, functional, OO, meta, reflective
JavaScript	multi-paradigm: scripting, prototype OO, imperative, functional
Lua	multi-paradigm: scripting, imperative, procedural, prototype OO, functional
OCaml	functional
Prolog	logic programming, declarative

2.1 Notes

The Prolog targetted is SWI Prolog[2]. Even though it's not entirely ISO-compatible—it covers part 1 of the ISO standard—it is the de-facto Edinburgh Prolog standard, unlike the other open-source, and fully ISO-compliant GNU Prolog, it is fully maintained.

In all the following OCaml examples, I'm using the Core library provided by Jane Street[3]. Jane Street's core is a better and more capable core library, with more of the libraries a developer needs, and it is extremely well maintained.

2.1.1 Common Lisp

The Common Lisp implementation in use is SBCL[4], a particularly fast open-source Lisp, 64-bit version 1.2.x onwards. It is compiled and distributed with the ASDF 3 system definition facility, with QuickLisp as the package manager. SBCL is quite portable across platforms, but its binaries are not always compiled with certain features, such as threading support on BSDs. To enable these features, an existing CL implementation is needed to compile SBCL from source, with the relevant build flags (`–fancy`) switched on.

2.1.2 Clojure

2.1.3 Lua

2.1.4 JavaScript

2.1.5 OCaml

2.1.6 Prolog

2.1.7 Assembly

Where I attempt assembly code, it's targeted at the 64-bit 80x86 (x86, or x86-64) platform. Being the lowest-level language there is besides raw machine language, assembler is non-portable and closely linked to the target platform it was coded for. It is possible to introduce a certain level of application portability by using *macro* assemblers. Macro

[2] SWI Prolog (`http://www.swi-prolog.org/`)
[3] Open Source @ Jane Street (`https://janestreet.github.io/`)
[4] Steel Bank Common Lisp, SBCL (`http://sbcl.org/`)

assemblers—of which Microsoft's MASM and Borland's TASM are the most famous—are a more complex, multi-pass, type of assembler providing pre-processor directives and other macro-like (not in the Lisp sense) facilities to shape the code at compile-time. For example, by using `%ifdef` and similar pre-processor directives, it's perfectly possible to include only definitions that fit the context: by defining system calls by name, and linking the definitions and values to the target operating system, it's possible to target a wider range of operating systems with a single source code.

This is the technique employed in all assembly examples here. I'm using the Netwide Assembler (NASM)[5], an x86 and x86-64 assembler portable across many OSs, including DOS, Windows, Mac OS X, Linux and all the BSDs.

In the first instance, my development is targeted at OpenBSD (amd64) and Linux, but should work on Darwin (Mac OS X) and other Unices as well. I am not building a wider compatibility layer, nor testing on other platforms, so the code may not compile and run outside of these two platforms.

NASM uses Intel-format instructions: `opcode destination source`, as opposed to the AT & T style, `opcode source destination`.

Below follows a code snippet which will form the prologue of any ***main*** assembly function; this code sets up the target OSs program segment prefix (PSP), as well as a thin layer of system call indirection. This code block is labelled `asm-prologue`, and will be called by including a «`asm-prologue`» directive within other code.

```
%ifdef  NetBSD
section .note.netbsd.ident
      dd      7,4,1
      db      "NetBSD",0,0
      dd      200000000
%endif

%ifdef  OpenBSD
section .note.openbsd.ident
      align   2
      dd      8,4,1
      db      "OpenBSD",0
      dd      0
      align   2
%endif

      section .text

%ifidn __OUTPUT_FORMAT__, macho64       ; MacOS X
      %define SYS_exit        0x2000001
      %define SYS_write       0x2000004

      global  start
      start:
%elifidn __OUTPUT_FORMAT__, elf64
      %ifdef  UNIX            ; Solaris/OI/FreeBSD/NetBSD/OpenBSD/DragonFly
              %define SYS_exit        1
              %define SYS_write       4
      %else                   ; Linux
              %define SYS_exit        60
              %define SYS_write       1
      %endif

      global  _start
      _start:
```

[5] Netwide Assembler, NASM (`http://www.nasm.us/`)

```
%else
      %error  "Unsupported platform"
%endif
```

This whole document is written in a Literate programming style[6], and relies on `noweb` *tangling*[7], which calls predefined, named blocks of code using `«block-name»` include style. As assembly code is very verbose, it's easier for my planning and thinking, and for future readability, if I break the code into smaller and more manageable blocks. So, where I implement an assembly solution, I will start out by listing a *skeleton* of named code blocks listing the functionality I'd like to have, after which I can write the code. At the end of the process, I merely have to *tangle* the code to get the output file, ready for compiling.

Finally, to compile assembly examples to native binaries, two steps are needed: assembling and linking. On Linux, run `nasm -f elf64 < source-filename >` to assemble, and `ld -s -static < source-filename >.o` to link. For debugging in GDB, add the following flags to the assembly directive `-F dwarf -g`, to export symbols for the debugger to read and display.

OpenBSD used the `a.out` format, but has now switched to `elf`, like Linux. This implementation has issues in relocating symbols and requires the `-nopie` (create position independent executable) argument at the linking step: `ld -nopie -static -s -o < binary-filename > < object-filename >`.

3 General Problems

3.1 FizzBuzz

FizzBuzz is a group word game, teaching children about division and screening inept computer programmers[8]. While this is a trivial problem, I want to see how easy it is to produce something slightly more complex, and useful, than "Hello, world" in a range of languages.

The goal is to create a list of 100 numbers starting from 1, replacing each number divisible by 3 by the string "Fizz", each number divisible by 5 with "Buzz". Any numbers divisible by both three and five will be replaced by "FizzBuzz".

There are many far more involved variations of this challenge, here I'm working with the simplest variation only.

3.1.1 Common Lisp

For the sake of purity, though not ease, my solution here relies on Lisp primitives `do` and `princ` instead of the easier and more powerful `loop` and `format` functions.

```
(do ((i 1 (1+ i)))
    ((> i 100))
  (princ
   (cond ((zerop (mod i 15)) "FizzBuzz")
         ((zerop (mod i 5)) "Buzz")
         ((zerop (mod i 3)) "Fizz")
         (t i)))
  (princ ", "))
```

3.1.2 Clojure

The easiest solution is imperative:

```
(for [i (range 1 101)]
  (print (cond (zero? (mod i 15)) "FizzBuzz"
               (zero? (mod i 5)) "Buzz"
               (zero? (mod i 3)) "Fizz"
```

[6] Literate Programming, D. E. Knuth (1984) (http://www.literateprogramming.com/)
[7] Noweb (https://www.cs.tufts.edu/~nr/noweb/)
[8] Wikipedia: Fizz buzz (https://en.wikipedia.org/wiki/Fizz_buzz)

```
            :else i)
      """))
```

though, it's slower than the functional, mapping variation, one that doesn't rely on any printing:

```
(map (fn [i]
       (cond (zero? (mod i 15)) "FizzBuzz"
             (zero? (mod i 5)) "Buzz"
             (zero? (mod i 3)) "Fizz"
             :else i))
     (range 1 101))
```

3.1.3 OCaml

Imperative code is always easy to write, generally quick to evaluate though less easy to understand; this is effectively using the functional OCaml to quickly solve the problem at hand—it is *not* idiomatic code. It is a waste of time using OCaml to program like this.

```
open Core.Std

let fizz_buzz_imp =
  for i = 1 to 100 do
    if (i mod 15) = 0 then printf "FizzBuzz"
    else if (i mod 5) = 0 then printf "Buzz"
    else if (i mod 3) = 0 then printf "Fizz"
    else printf "%d" i;
    printf ", "
  done;;
```

The following code makes far better use of OCaml: it builds and traverses the list *functionally*, using OCaml's killer feature—pattern matching—to return our "Fizz" and our "Buzz".

```
open Core.Std

let fizz_buzz_fun i =
  match i mod 3, i mod 5 with
  | 0, 0 -> "FizzBuzz"
  | _, 0 -> "Buzz"
  | 0, _ -> "Fizz"
  | _, _ -> string_of_int i

let lst =
  List.map ~f:fizz_buzz (List.range 1 101)
```

It's important to note that this runs two `mod` divisions for every member of the list, while the above imperative code runs anywhere from one to four of these operations. As 3 is the lowest divisor here, it stands to reason it will run more often than the other two, with the code dropping to the `else` clause most often, having previously carried out all three division operations. Intuitively, the `mod 3` operation will be run 33 times in a list of 100 numbers, which implies—any compiler optimisation aside—that the other two divisions are carried out 33 times, too. By the same logic, the `mod 5` operation will run 20 times, and the `mod 15` will only run 6 times. Execution will fall through all three operations for 53 numbers:

$\frac{100}{3} + \frac{100}{5} - \frac{100}{15}$

which indicates that we expect three sets of division operations to be carried out a full 53 times, with a single `mod 15` division operation carried out only six times. If the compiler cannot be expected to optimise away much of this repetition, the second algorithm is only slower than the first for 6 numbers out of 100, but is quicker for a further

53! The number of divisions employed in the second algorithm is a constant of $2n$, or $2 \times 100 = 200$, whereas the first algorithm needs to repeat it $(6 \times 1) + ((20 - 6) \times 2) + ((33 - 6) \times 3) + (53 \times 3) = 274$. Thus, all else being equal, the imperative algorithm should be approximately 37% slower than OCaml's idiomatic style.

3.1.4 JavaScript

```
for (var i=1; i<=100; i++){
    if ((i%15) == 0) console.log("FizzBuzz");
    else if ((i%5) == 0) console.log("Buzz");
    else if ((i%3) == 0) console.log("Fizz");
    else console.log(i);
}
```

3.1.5 Lua

Lua makes this really easy, though in the typical and verbose C-like syntax.

```
for i=1,100 do
  if (i % 15) == 0 then
    print("FizzBuzz")
  elseif (i % 5) == 0 then
    print("Buzz")
  elseif (i % 3) == 0 then
    print("Fizz")
  else
    print(i)
  end
end
```

3.1.6 Prolog

Prolog twists the mind in beautiful ways, in finding the solution. It tries very hard to make imperative programming difficult and unwieldy. While functional programming is an option in many other languages, and a recursive number generator is merely a possibility, in Prolog it's almost a necessity.

```
recursive_count :-
  recursive_count(10).
recursive_count(X) :-
  recursive_count(0, X, 1).
recursive_count(X, Y) :-
  recursive_count(X, Y, 1).
recursive_count(X, Y, Z) :-
  X < Y,
  X_mod_5 is X mod 5,
  X_mod_3 is X mod 3,
  fizz_buzz(X_mod_5, X_mod_3, X),
  NX is X + Z,
  recursive_count(NX, Y, Z).

fizz_buzz(_, _, 0) :-
  !,
  write('0 ').
fizz_buzz(0, 0, _) :-
  !,
  write('FizzBuzz ').
```

```
fizz_buzz(0, _, _) :-
  !,
  write('Fizz ').
fizz_buzz(_, 0, _) :-
  !,
  write('Buzz ').
fizz_buzz(_, _, Z) :-
  format('~d ', Z).
```

In Prolog, it's easier to build rules to match every possible outcome, as in OCaml, once this concept is grasped, it makes for conceptually much cleaner code than conditionals. I can't fully reason about the performance of this matching, nor of recursion, except to say that it's likely to result in comparatively slow code. Since the function is called recursively for each number, we are paying the price of a few dozen clock cycles N times.

Why have I chosen to repeat the rule so many times? Defensive coding; to check all possible invocations of either function as well as to provide an element of conditionality and variadic argument dispatch, I have the rules matching on varying numbers of provided arguments. Unlike other languages, this seems very repetitive but results in much cleaner and smaller rules and functions, and makes for code that is conceptually much easier to understand.

The following example still relies on the above `fizz_buzz` rules, but provides a non-recursive way of generating numbers, relying instead on the built-in `between/3` function. Is this likely to be faster? Perhaps only marginally, as in place of an expensive function call for each number, it now rebinds a variable instead, a memory operation which still costs a few clock cycles.

```
non_recursive_count :-
  non_recursive_count(10).
non_recursive_count(X) :-
  non_recursive_count(0, X, _).
non_recursive_count(X, Y) :-
  non_recursive_count(X, Y, _).
non_recursive_count(X, Y, Z) :-
  between(X, Y, Z),
  Z_mod_5 is Z mod 5,
  Z_mod_3 is Z mod 3,
  fizz_buzz(Z_mod_5, Z_mod_3, Z),
  Z = Y.
```

Using the built-in `time/1` function, the recursive function (for numbers 0 to 101), makes 712 inferences, takes 99% cpu and uses 4404251 Lips, while the non-recursive one makes 616 inferences using 93% cpu and only 2194209 Lips, making it more efficient of the two.

3.1.7 Assembly

For the initial, naive, version of this, I'd like to bring my functional and modular mind to the problem; I want to create reusable functions and to use mathematical thinking, instead of thinking from the point of view of the machine.

I will build this example up in three parts: printing data (strings), a way to iterate and generate the list of numbers, then finally a function to map the range of input values (list of numbers), converting them to our target strings.

Here is the complete example, followed by a breakdown of each section.

```
%ifdef  NetBSD
section .note.netbsd.ident
      dd      7,4,1
      db      "NetBSD",0,0
      dd      200000000
%endif

%ifdef  OpenBSD
```

```
  section .note.openbsd.ident
        align   2
        dd      8,4,1
        db      "OpenBSD",0
        dd      0
        align   2
  %endif

        section .text

  %ifidn __OUTPUT_FORMAT__, macho64       ; MacOS X
        %define SYS_exit        0x2000001
        %define SYS_write       0x2000004

        global  start
        start:
  %elifidn __OUTPUT_FORMAT__, elf64
        %ifdef  UNIX            ; Solaris/OI/FreeBSD/NetBSD/OpenBSD/DragonFly
                %define SYS_exit        1
                %define SYS_write       4
        %else                   ; Linux
                %define SYS_exit        60
                %define SYS_write       1
        %endif

        global  _start
        _start:
  %else
        %error  "Unsupported platform"
  %endif

        mov     rcx, 1
for:
        cmp     rcx, 10
        jl      forcode
        jmp     end

forcode:
        push    rcx
        call    _fizzbuzz
        pop     rcx
        inc     rcx
        jmp     for

end:
        call    _exit

        global _print
_print:
        push    rbp
        mov     rbp, rsp

        mov     rax, SYS_write
```

```
        mov     rdi, 1
        syscall

        leave
        ret
        global _exit
_exit:
        mov     rax, SYS_exit
        xor     rdi, rdi
        syscall
        global  _fizzbuzz
_fizzbuzz:
        push    rbp
        mov     rbp, rsp
        mov     rcx, [rbp+16]

if_div_by_15:
        xor     rax, rax
        xor     rdx, rdx
        mov     eax, ecx
        mov     r9, 15
        div     r9
        cmp     rdx, 0
        jz      div_by_15

if_div_by_5:
        xor     rax, rax
        xor     rdx, rdx
        mov     eax, ecx
        mov     r9, 5
        div     r9
        cmp     rdx, 0
        jz      div_by_5

if_div_by_3:
        xor     rax, rax
        xor     rdx, rdx
        mov     eax, ecx
        mov     r9, 3
        div     r9
        cmp     rdx, 0
        jz      div_by_3

else:
        lea     r8, [ascii_numbers + rcx]
        mov     rsi, r8
        mov     rdx, 1
        call    _print
        mov     rsi, space
        mov     rdx, 1
        call    _print

        jmp     end_if_div
```

```
end_if_div:
        leave
        ret

div_by_15:
        mov     rsi, fizz_buzz
        mov     rdx, 8
        call    _print
        mov     rsi, space
        mov     rdx, 1
        call    _print
        jmp     end_if_div

div_by_5:
        mov     rsi, buzz
        mov     rdx, 4
        call    _print
        mov     rsi, space
        mov     rdx, 1
        call    _print
        jmp     end_if_div

div_by_3:
        lea     rsi, [fizz]
        lea     rdi, [output_string]
        cld
        mov     rcx, 4
        rep     movsb

        mov     rsi, fizz
        mov     rdx, 4
        call    _print
        mov     rsi, space
        mov     rdx, 1
        call    _print
        jmp     end_if_div

        section .data

        ascii_numbers
                db  0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37, 0x38, 0x39
        fizz    db  ”Fizz”
        buzz    db  ”Buzz”
        fizz_buzz
                db  ”FizzBuzz”
        space   db  ” ”
        comma   db  ”,”
        section .bss

        align 4
        output_string resb 400
```

Below is a simple print function. It expects that the pointer to the string has already been placed in the `rsi` register, with the string length in `rdx`. As with most well-behaved functions, there is a stack-pointer saving function prologue, followed by its restoration at function's end.

```
        global _print
_print:
        push    rbp
        mov     rbp, rsp

        mov     rax, SYS_write
        mov     rdi, 1
        syscall

        leave
        ret
```

As we construct the output string, in order to print it at some point, we need to inform the `_print`

Finally, our program has to end, and here is an `exit` function for it; no return is expected from this function, so there is no redundant function entrance prologue saving the stack pointer.

```
        global _exit
_exit:
        mov     rax, SYS_exit
        xor     rdi, rdi
        syscall
```

The following code sets `rcx`, the traditional count register to 1, from where we will iterate by one in a `for` style loop. As this loop forms the basis of the program, it sets itself up, comparing `rcx` to n, where n is the inclusive limit of our count, calling the `_fizzbuzz` function each time.

Once it's iterated through the range of numbers, it will call `_exit` to terminate the entire program.

```
        mov     rcx, 1
for:
        cmp     rcx, 10
        jl      forcode
        jmp     end

forcode:
        push    rcx
        call    _fizzbuzz
        pop     rcx
        inc     rcx
        jmp     for

end:
        call    _exit
```

The `_fizzbuzz` function will be called to inspect the number (the value of our iterator, `rcx`), performing the three desired divisions and checks: $nmod15$, $nmod5$ and $nmod3$, in Lisp's `cond` style. If the code falls through all three tests, it will merely print the number.

Both the condition set-up and testing code, as well as the actual *then* statement code is duplicated over and over, instead of being refactored into its own function; while the function will be cleaner, *unrolling* the code here will make it execute faster, without the need to go through the expensive set-up, function call and jump—with the associated loss of code in the cache pipeline—followed by teardown.

```
        global  _fizzbuzz
_fizzbuzz:
        push    rbp
        mov     rbp, rsp
        mov     rcx, [rbp+16]

if_div_by_15:
        xor     rax, rax
        xor     rdx, rdx
        mov     eax, ecx
        mov     r9, 15
        div     r9
        cmp     rdx, 0
        jz      div_by_15

if_div_by_5:
        xor     rax, rax
        xor     rdx, rdx
        mov     eax, ecx
        mov     r9, 5
        div     r9
        cmp     rdx, 0
        jz      div_by_5

if_div_by_3:
        xor     rax, rax
        xor     rdx, rdx
        mov     eax, ecx
        mov     r9, 3
        div     r9
        cmp     rdx, 0
        jz      div_by_3

else:
        lea     r8, [ascii_numbers + rcx]
        mov     rsi, r8
        mov     rdx, 1
        call    _print
        mov     rsi, space
        mov     rdx, 1
        call    _print

        jmp     end_if_div

end_if_div:
        leave
        ret

div_by_15:
        mov     rsi, fizz_buzz
        mov     rdx, 8
        call    _print
        mov     rsi, space
```

```
        mov     rdx, 1
        call    _print
        jmp     end_if_div

div_by_5:
        mov     rsi, buzz
        mov     rdx, 4
        call    _print
        mov     rsi, space
        mov     rdx, 1
        call    _print
        jmp     end_if_div

div_by_3:
        lea     rsi, [fizz]
        lea     rdi, [output_string]
        cld
        mov     rcx, 4
        rep     movsb

        mov     rsi, fizz
        mov     rdx, 4
        call    _print
        mov     rsi, space
        mov     rdx, 1
        call    _print
        jmp     end_if_div
```

At the very end of the program, here is a definition of all the required data:

```
section .data

ascii_numbers
        db  0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37, 0x38, 0x39
fizz    db  "Fizz"
buzz    db  "Buzz"
fizz_buzz
        db  "FizzBuzz"
space   db  " "
comma   db  ","
```

And an uninitialised data area, where we can build up our final string, initialised to 400 bytes, enough space to hold 100 units of numbers, with the occasional four-byte “Fizz” or “Buzz”, and the very rare “FizzBuzz”. At the moment, this isn’t used as every iteration prints its own result immediately, but in the long run, it’s better and more efficient to construct the final string, printing it only once.

```
section .bss

align 4
output_string resb 400
```

This initial version tries hard to be easy to read and follow, to be modular, and to work without employing any low-level *tricks*. Further, I’m implicitly expecting to be able to convert a number to ASCII representation, printing it. In truth this holds only for values 0–9, which is why the iterative loop is limited to only the first nine numbers.

Beyond that, we enter hexadecimal territory, which needs an—as yet unwritten—hex to ASCII decimal conversion function: $n_{16} -> n_{10}$.

Assembly can be used in any paradigm as desired, but the most *idiomatic* way is to save as many clock cycles as possible, otherwise one may as well program in a high-level language. This mentality looks for the most performant solution to the problem, at the cost of readability and comprehension. In this FizzBuzz example, that means not converting numbers from hex to ASCII, as that is recursive and relies on multiplication, both of which are slow at the level of the machine. Instead, using binary-coded decimal (BCD) numbers is preferable, and the processor has been built with exactly this optimisation in mind.

Note: the binary-coded decimal processor instructions have been deprecated in 64-bit mode, thus I will use the more complex floating point unit (FPU) for BCD conversion in the processor. As this requires explicit loading from memory and popping to memory, not accepting loading from registers or immediate values, the following example makes many more references to memory than would have been needed in the past.

Secondly, division is one of the most expensive operations a CPU provides; in place of running a `div` operation to check the remainder, the idiomatic way to perform this check is through the use of counters. By keeping separate counters for the divisible by three, by five, and by fifteen checks we need to carry out, decrementing each with every iteration, the program will quickly be able to choose between the four conditions. A further optimisation at this stage is not to use yet another counter for five, as only binary-coded decimals with values `0x30` or `0x35` are divisible by five. Two comparison operations are an acceptable price to pay to save on two memory accesses—one to read the counter, a second incrementing it—or the loss of one register to hold the counter.

The final method to speed up the running of this simple program, and to reduce code needed is to not modularise it in any way; by using subroutines or jump locations, the code will be terse and short. Obviously, all of these performance modifications come at the cost of the developer's time and comprehension, as well as a *perfect* understanding of the problem. If any variable of the problem changes, the program will fail remorselessly, unless it is rethought and rebuilt from the ground up. Thus, this type of heavily optimised solution is brittle, difficult to comprehend and is, ultimately, unmaintainable.

Though I would never want this solution, except in a complex inner loop as a bottleneck optimisation, here is an attempt at 1980s style assembly (when computing resources were scarce).

```
%ifdef  NetBSD
section .note.netbsd.ident
      dd      7,4,1
      db      "NetBSD",0,0
      dd      200000000
%endif

%ifdef  OpenBSD
section .note.openbsd.ident
      align   2
      dd      8,4,1
      db      "OpenBSD",0
      dd      0
      align   2
%endif

      section .text

%ifidn __OUTPUT_FORMAT__, macho64       ; MacOS X
      %define SYS_exit        0x2000001
      %define SYS_write       0x2000004

      global  start
      start:
%elifidn __OUTPUT_FORMAT__, elf64
```

```
        %ifdef  UNIX            ; Solaris/OI/FreeBSD/NetBSD/OpenBSD/DragonFly
                %define SYS_exit        1
                %define SYS_write       4
        %else                   ; Linux
                %define SYS_exit        60
                %define SYS_write       1
        %endif

        global  _start
        _start:
  %else
        %error  "Unsupported platform"
  %endif

        mov     rcx, 100
        mov     r8, 3
        mov     r9, 5
        mov     r10, 15
        finit
        fbld    [increment]

for_loop:
        fbld    [temp_bcd]
        fadd    st0,st1
        fbstp   [temp_bcd]
        dec     r8
        dec     r9
        dec     r10

check_for_fifteen:
        cmp     r10, 0
        je      write_fizzbuzz

check_for_five:
        cmp     r9, 0
        je      write_buzz

check_for_three:
        cmp     r8, 0
        je      write_fizz

else:
        lea     esi, [temp_bcd]
        lea     edi, [bcd_num]
        xor     eax, eax
        mov     al, [esi]
        mov     rbx, rax

ho_digit:
        shr     al, 4
        cmp     rax, 0
        je      lo_digit
        add     ax, 0x30
```

```
lo_digit:
        and     bl, 0xf
        add     bx, 0x30
        shl     bx, 8
        add     rax, rbx
        stosw

        push    rcx
        mov     rsi, bcd_num
        mov     rdx, 2
        call    _print

        mov     rsi, space
        mov     rdx, 1
        call    _print
        pop     rcx

continue_iteration:
        dec     rcx
        cmp     rcx, 0
        jbe     exit
        jmp     for_loop

exit:
        mov     rsi, newline
        mov     rdx, 2
        call    _print

        mov     rax, SYS_exit
        xor     rbx, rbx
        syscall

write_fizz:
        push    rcx
        mov     rsi, fizz
        mov     rdx, 5
        call    _print
        pop     rcx
        mov     r8, 3
        jmp     continue_iteration

write_buzz:
        push    rcx
        mov     rsi, buzz
        mov     rdx, 5
        call    _print
        pop     rcx
        mov     r9, 5
        jmp     continue_iteration

write_fizzbuzz:
        push    rcx
```

```
        mov     rsi, fizzbuzz
        mov     rdx, 9
        call    _print
        pop     rcx
        mov     r8, 3
        mov     r9, 5
        mov     r10, 15
        jmp     continue_iteration

  global  _print
_print:
        mov     rax, SYS_write
        mov     rdi, 1
        syscall
        ret

section .data
        space   db  " "
        newline db  10,13
        fizz    db  "Fizz "
        buzz    db  "Buzz "
        fizzbuzz  db "FizzBuzz "
        increment db 0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
        temp_bcd  db 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
        bcd_num   db 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
```

And, that's it: a complete x86-64 assembly language FizzBuzz example, effectively in only 105 lines of code which can still be ruthlessly pared down, weighing 856 bytes when compiled.

3.2 Reversing a string

Take a string and reverse it, so that "asdf" becomes "fdsa"[9].

For the purposes of the exercise, in the languages that support it, I will use the built-in *length* function. While this may be slow, depending on the implementation, it will simplify the code by not requiring me to implement an imperative or recursive length calculator. However, I will not take the next obvious step of using the provided string reverse function, to build my own.

3.2.1 Common Lisp

The imperative solution is terse and easy to roll out:

```
(defun reverse-string-imperatively (string)
  "Takes a string, returning it reversed."
  (coerce
   (loop for i from (1- (length *source-string*))
         downto 0
         collect (elt *source-string* i)) 'string))
```

while its functional equivalent is slightly more involved:

```
(defun reverse-string-recursively (string)
  "Takes a string, returning a reversed string."
  (labels ((rec-reverse (list accumulator)
             (cond ((car list) (rec-reverse
```

[9]Reverse a string, Rosetta Code (http://rosettacode.org/wiki/Reverse_a_string)

```
                                (cdr list)
                                (cons (car list) accumulator)))
                 (t accumulator))))
    (coerce
     (rec-reverse (coerce string 'list) NIL)
     'string)))
```

3.3 Convert RGB to hexadecimal

3.4 Concordance

4 Low-level Manipulation

4.1 Test the high-order bit

4.2 Count all the bits in an integer value

5 Sorting

5.1 Quicksort

6 Mathematical Algorithms

6.1 Fibonacci sequence

The Fibonacci sequence is a sequence F_n of natural numbers defined recursively[10]:

$F_0 = 0$
$F_1 = 1$
$F_n = F_{n-1} + F_{n-2}, ifn > 1$

Write a function to generate the nth Fibonacci number, iteratively or recursively, though recursive solutions are generally considered too slow.

The formula above is already in recursive format; it's a little more difficult to work out an iterative algorithm, though. We need four variables to do this: n, $f(n-1)$, $f(n-2)$ and $f(n)$.

n	i	0	1	2	3	4	5	6	7	8	9	10
$f(n)$	j	0	1	1	2	3	5	8	13	21	34	55
$f(n-1)$	k	0	0	1	1	2	3	5	8	13	21	34
$f(n-2)$		0	0	0	1	1	2	3	5	8	13	21

With n starting with $n=2$—we already know what $f(n)$ is for $n = 0$ and $n=1$—for every iteration of n, we add $f(n-1)$ (the current value of j) and $f(n-2)$ (the current value of k), assigning the sum to j. In order to minimise needed space and variable use, the algorithm is already hard to understand, even before implementation. This is the true cost of imperative programming, in the necessary quest for maximum performance. It's easy to primitively benchmark each solution below by calculating F_{39}, which is $63, 245, 986$.

6.1.1 Common Lisp

Recursive solution:

```
(defun fibonacci-number-recursive (n)
  "Returns the fibonacci number given an input n."
  (when (not (minusp n))
    (cond ((zerop n) 0)
```

[10] Fibonacci sequence, Rosetta code (http://rosettacode.org/wiki/Fibonacci_sequence); Fibonacci number, Wikipedia (https://en.wikipedia.org/wiki/Fibonacci_number)

```
        ((= n 1) 1)
        (t (+ (fibonacci-number (- n 1))
              (fibonacci-number (- n 2)))))))
```

just for fun, a tail-recursive solution:

```
(defun fibonacci-number-trecursive (n &optional (m 1) (l 0))
  "Returns the fibonacci number given an input n, using tail-optimised recursion."
  (if (zerop n)
      l
      (fibonacci-number-trecursive (1- n)
                                   (+ m l)
                                   m)))
```

While this is truly a tail-recursive solution, in reality it has none of the mathematical beauty of its predecessor; it is now completely iterative.

Finally, a completely iterative solution:

```
(defun fibonacci-number-iterative (n)
  "Returns the fibonacci number given an input n."
  (do ((i 0 (1+ i))
       (j 0 k)
       (k 1 (+ j k)))
      ((= i n) j)))
```

In the end, even though the tail-recursive algorithm doesn't force the stack to grow, it's still more resource-hungry than the iterative one. In my tests, the iterative F_{39} takes 1,503 processor cycles, the tail-recursive needs 40% longer, at 2,113 cycles. The recursive algorithm takes over 500,000,000% longer, at 8,634,779,396 cycles! Its running time, at only 3.7 seconds doesn't seem bad, but the other two algorithms both report 0 seconds. It's only possible to exercise the iterative and tail-recursive algorithms at very high values of n; `fibonacci-number-trecursive` run with $n = 1,000,000$ takes 28.9 seconds to run, and up to 70 ***billion*** processor cycles, while `fibonacci-number-iterative` takes 28.7 seconds, at very slightly less CPU peak usage and 1 billion fewer processor cycles. At this magnitude of n, though, a substantial amount of time is needed to sum, store and print the number which is now a gargantuan 208,987 digits long!

6.2 Euclidean Algorithm

$gcd(m, n) = (m > n \rightarrow gcd(n, m), rem(n, m) = 0 \rightarrow m, T \rightarrow gcd(rem(n, m), m))$

```
(defun gcd (m n)
  (cond ((> m n) (gcd n m))
        ((zerop (mod n m)) m)
        (t (gcd (mod n m) m))))
```

6.3 Newtonian Algorithm