COMP 40 Assignment: Integer and Logical Operations

(Part C, the challenge problem, will be assigned and due after the midterm.)

Contents

1 Purpose, overview, and instructions

The primary purpose of this assignment is to give you practice unpacking and repacking representations that put multiple integers (both signed and unsigned) into a word. You'll also learn to analyze two's-complement arithmetic so that you know how many bits are needed to store the results of calculations, and when not enough bits are available, you'll know how to adapt your code. You'll be exposed to some of the horrors of floating-point arithmetic. Finally, you'll get a post-midterm challenge problem that will test the modularity of your code.

As a minor side benefit, you'll also learn a little bit about how broadcast color TV works as well as the basic principle behind JPEG image compression.

Here's what you'll do:

- Write *and test* linear bijections: a discrete cosine transform and a bijection between RGB and component video ($Y/P_R/P_B$) color spaces.
- Write functions to put a small integer into a word or extract a small integer from a word. You'll work with both signed and unsigned integers.
- Write a lossy image compressor that takes a PPM image and compresses the image by transforming color spaces and discarding information that is not easily seen by the human eye.

There is a long story below about the representation of color and brightness and the use of techniques from linear algebra for image compression. The story is interesting and important, but the real reason you're doing this work is to give you a deep understanding of the capabilities and limitations of machine arithmetic. The amount of code you have to write is fairly small, certainly under 400 lines total. But to understand what code to write and how to put it together, you will have to analyze the problem.

Begin by running

```
git clone /comp/40/git/arith
```

to get files you will need for the assignment.

1.1 Problem-solving technique: stepwise refinement, analysis, and composition

In COMP 40, you practice *solving problems by writing programs*. You'll find problem-solving more difficult (and more satisfying) than simply writing a program someone has told you to write. To solve the problem of image compression, we recommend a technique called *stepwise refinement*.

When using stepwise refinement, one analyzes a problem by breaking the problem into parts, which in turn can be broken into subparts, and so on, until the individual sub-sub-parts are either already to be found in a library or are so easy as to be quickly solvable by simple code. Each individual subpart is solved by a function or by a collection of functions in a module. Each solution is written as another function, and so on, all the way up to the `main` function, which solves the whole problem. In other words, the solution to the main problem is composed of solutions to the individual parts. To design software systems successfully, you must master the techniques of analysis and composition.

Keep in mind these units of composition:

- The *function* should do one, simple job.
- The *interface to an abstract data type* packages an important abstraction in the world of ideas and makes it usable in a computer program. Such an interface *hides representation*, promoting reuse.
- Other *interfaces* can also promote reuse. Here are two useful design principles for interfaces:
  - *Package together* collections of functions that *operate in the same problem domain*. Examples might include statistical functions (mean, variance, covariance, and so on) or linear-algebra functions (inner and outer products, matrix multiply, matrix inversion, and so on).
  - *Package together* functions that *share a secret*. The idea is to hide the secret so you enable modular reasoning: the rest of the program doesn't know the secret, so it can depend only on the functions in the interface. A good example of this kind of interface is the `Pnm` interface, which hides the secret that each kind of PNM file has two different on-disk representations, as well as hiding the details of those representations.

In C, each interface is expressed in a `.h` file, and it normally is implemented by a single `.c` file. *We will evaluate your work according to how well you organize your solution into separate files.*

1.2 What we provide for you

All files we provide will be in `/comp/40/include` or `/comp/40/lib64`, or else you will acquire them using `git` (as discussed above). The files include

- The header file `bitpack.h`, which is not implemented, but which you can compile against with the compiler option `-I/comp/40/include`. You will implement a corresponding `bitpack.c`.
- The header file `compress40.h`, which is not implemented, but which you can compile against with the compiler option `-I/comp/40/include`. You will implement a corresponding `compress40.c`.

- The header file `arith40.h`, which you compile against with the `-I/comp/40/include` option, and whose implementation is in the library `libarith40.a`, which you link with the options `-L/comp/40/lib64 -larith40`[1]
- The file `40image.c`, which contains a `main` function that can handle the command line for you. You can acquire it by `git clone /comp/40/git/arith`.

1.3 What we expect from you

Your **design document**, to be submitted using `submit40-arith-design`, should *describe your overall design*, and it should also include *separate descriptions of each component*. Your sections on the `Bitpack` module and on parts of the `40image` program can be relatively short, since in these cases I have done some of the design work for you. But you should have a *detailed plan for testing* each of these components. Your design document may be a plain text document named `DESIGN` or a PDF named `design.pdf`.

Also, your `40image` program should not be implemented as a single component. Your design document should not only explain how `40image` is to be implemented by a combination of components, but should also present a separate design description of each component.

The following elements of your design document will be *critical*:

1. Separate documentation of the architecture of each major component as well as the overall architecture.
2. Architecture sections that identify modules, types, and functions *by name*. Choosing good names is valuable, so do it early. Formal *definitions* or *declarations* of your types and functions are not necessary at this stage; if you prefer not to write C interfaces, just sketch the types' definitions and functions' specifications in *concise*, informal English.
3. *You must have a plan for testing each individual component in isolation.* If you don't have a good test plan, your compressor likely won't work. Your best bet is to write down universal laws and write code to be sure that they hold on a variety of inputs. DON'T FORGET TO INCLUDE YOUR TESTING PLAN. In past years, many students have lost credit by failing to outline a sufficiently detailed test plan. The plan need not be complicated, but it must describe an effective approach to testing your implementation.

An additional element that should help guide you toward a good design is an answer to the following question:

4. How will your design enable you to do well on the challenge problem in Section 2.3 on page 13?

Finally, here is a question that is not critical but that I would like you to answer in your design document:

5. An image is compressed and then decompressed. Identify all the places where information could be lost. Then it's compressed and decompressed again. Could more information be lost? How?

Your **implementation**, to be submitted using `submit40-arith`, should include

6. File `bitpack.c`, which should implement the `Bitpack` interface.
7. All the `.c` and `.h` files you create to implement the `40image` program.
8. A `compile` file, which when run with `sh`, compiles all your source code and produces both a `40image` executable binary and also a `bitpack.o` relocatable object file. File `bitpack.o` should contain the entire implementation of the `Bitpack` interface (and nothing else). You should create your compile script by adapting the ones used for earlier assignments.
9. A `README` file which

[1] As detailed in Norman Ramsey's handout on writing compile scripts, the linker translates the option `-lfoo` into a search for a file named `libfoo.so` or `libfoo.a`. It searches all directories named in a `-L` option as well as some built-in directories. You can see all the directories by running `gcc` with the `-v` option.