COMP 40 Laboratory: Testing the Bitpack interface

Introduction

In this lab, you’ll test your implementation of Bitpack using three technologies:

- Exhaustive testing
- Random testing
- Algebraic laws

You’ll carry out your activity in these steps:

A. Write a function `check_laws` that takes parameters and checks the algebraic laws in the homework.

B. Write a loop that calls `check_laws` repeatedly, checking all values of some parameters and randomly chosen values of other parameters.

C. Call the whole thing from a `main()` function that opens file `/dev/urandom` as a source of randomness.

D. Add more laws to `check_laws`.

Step D is the most critical step, but when you have the infrastructure in place, it is easy.

Fields

To apply these technologies to Bitpack, we introduce a critical idea: a field of a word. A field is simply a sequence (possibly empty) of contiguous bits within a word. Each field is characterized by two numbers:

- The number of bits in the field is the width $w$.
- The position of the field is given by the position of the least significant bit, or more precisely, by the number of bits in the word that are less significant than the least significant bit of the field. We abbreviate this position $lsb$.

Any field can be characterized by a pair $(w, lsb)$, subject to these constraints:

\[
0 \leq w \leq \text{wordsize} \\
0 \leq lsb \\
lsb + w \leq \text{wordsize}.
\]

For example, the most significant byte of a 32-bit word would have width 8 and $lsb = 24$.

The use of fields of width zero may be surprising, but it eliminates a lot of case analysis—and case analysis is your enemy. We’ll define a field of width zero such that it may appear in any position, but it always stores the value 0.

With fields of width zero, given any field $f = (w, lsb)$, we can define two more fields $f_h$ and $f_l$ which are above and below field $f$, such that the three fields together make up an entire word. For example, if $f = (9, 23)$ and the word size is 64, then $f_h = (32, 32)$ and $f_l = (23, 0)$.

1. Given a general field $f = (w, lsb)$ and a word size of 64, give mathematical formulas for $f_h$ and $f_l$. 
Your homework assignment already contains two algebraic laws, near the bottom of page 12 of the assignment, near the phrase “satisfy the mathematical laws you would expect.” There are two different “get-new” laws. For this lab, you will write those and other laws as a C function you can use for testing.

2. Write a C function called check_laws which takes as parameters word, w, lsb, val, w2, and lsb2 as used in the laws of your handout. The function should assert that the laws hold.

   The body of the function should assert the laws on page 12 of the assignment, but you will have to guard against exceptions! The laws do not hold if val does not fit in w unsigned bits. Use #include <except.h> and

   TRY
   assert(...);
   ELSE
   fprintf(stderr, "Exception raised during testing\n");
   END_TRY;

Exhaustive and random testing

With what arguments will you call the function you define in step 2? Since you know the equations 0 ≤ w ≤ 64 and 0 ≤ lsb ≤ 64 − w, you can actually afford to test all combinations of (w, lsb) exhaustively—there are only about 2,000 combinations. But if you also test (w2, lsb2) exhaustively, that comes to 4 million combinations. If testing a law takes 100 nanoseconds (I’m guessing here), it might take half a second to try all combinations—and that’s with just one set of values for word and val. So you cannot afford completely exhaustive testing. I recommend that you test (w, lsb) exhaustively but use random testing for other values. In other words, your code should be structured like this:

   for (unsigned w = 0; w <= 64; w++)
      for (unsigned lsb = 0; lsb + w <= 64; lsb++)
         /* 1000 random trials */
         for (unsigned trial = 0; trial < 1000; trial++) {
            ... set values of other parameters randomly
            check_laws(...);
         }

As your source of random bits, I recommend that you use the special Linux file /dev/urandom.

Generating random values

Your main function can contain the lines

   FILE *randfp = fopen("/dev/urandom", "rb");
   assert(randfp != NULL);

At this point, you can set an integral value to all random bits as follows:

   uint64_t word;
   size_t rc = fread(&word, sizeof(word), 1, randfp);
   assert(rc == 1);

A random word is always safe, but to generate a sensible val, you will have to make sure that, for example val fits in w signed bits.

3. Given a completely random 64-bit val, how would you compute a val that fits in 5 unsigned bits?

   Hint: it can be done in just two instructions.
So that you can test both the signed and the unsigned functions in your Bitpack interface, you will want to have both signed and unsigned random vals.

4. Put all the pieces together and create a test program you can run.

5. Run it and see what works.

More algebraic laws

The real power of this testing method lies in the algebraic laws. Please add the following laws to your function check_laws:

6. Add a “new-new” law that says if you insert a new field \((w, lsb)\) and another new field \((w', lsb')\), and if the two fields have no bits in common, then the result is the same as first inserting \((w', lsb')\) and then \((w, lsb)\). The key is defining a concise test (formula) for determining whether two fields have any bits in common. You will find that the formulation using width and position leads to an elegant formula.

7. Add a “new-new” law that says if you insert a new field \((w, lsb)\) and another new field \((w', lsb')\), and if the field \((w, lsb)\) is entirely contained within the field \((w', lsb')\), then the result is the same as if you had inserted only \((w', lsb')\). Again, search for an elegant formula.

8. Add a “new-get” law that says if you insert a field \(f\), the bits \(f_h\) that are above (more significant than) the inserted field are unchanged. For this law, fields of width zero must work. And you must use one of your formulas from step 1 on page 1.

9. Add a “new-get” law that says if you insert a field \(f\), the bits \(f_l\) that are below (less significant than) the inserted field are unchanged. For this law, fields of width zero must work. And you must use the other one of your formulas from step 1 on page 1.

10. Add some interesting laws regarding the “fits” predicates. For example, it is true or false that

\[
\text{Bitpack\_fitsu}(n, w) \equiv \text{Bitpack\_fitsu}(n \ll 2, w + 2)\\
\text{Bitpack\_fitsu}(n, w) \equiv \text{Bitpack\_fitsu}(n \gg 2, w - 2)
\]

11. To test your code thoroughly, you will want duplicate signed and unsigned versions of all your laws. You can do this after lab.

Remember, in the C programming language, the result of a shift by 64 bits is not defined. You must treat this case specially in your Bitpack module and in any testing code.