Proven Delights

Contents

Other formats:

- HTML Version
- Source Code

1 Proven Delights

This project aims to include templated C++ implementations of many of the algorithms from Hacker's Delight, alongside CBMC proofs of the implementations. While the implementations are typically trivial, that they work at all is often nonobvious. Proofs can show that they work for all input values.

The fully built documentation of a recent version can always be found online: http://jepler.github.io/ProvenDelights/

1.1 Structure

1.1.1 The proofs/ directory

Each file in proofs/ is an asciidoc document. It must contain a code section lead off by ".Implementation" which implements the function as a C++ template or inline; and a code section lead off by ".Proof" which implements a CBMC proof of the function's properties as given in Hacker's Delight.

Where an implementation or proof needs to use the implementation of another function, it can #include the definition. For example, to use the implementation of `turn_off_rightmost_ones`, simply

```
#include "turn_off_rightmost_ones.h"
```

1.1.2 The structure/ directory

The asciidoc files in this directory create a structure which is intended to mirror the chapter and section numbering of Hacker's Delight (second edition). To this end, it will initially contain many empty and placeholder sections.

Where appropriate, chapter and section titles the same as those in Hacker's Delight are used for the purpose of identifying the correspondence between the book and this body of proofs.

1.1.3 The include/ directory

This directory holds utility code to be used in proofs, such as `assume`, `assert`, and `proof_popcnt`.

1.1.4 The gen-include/ directory

This directory holds the generated header files for each function implemented.

1.1.5 The docs/ directory

This directory holds the generated documentation, "proofs.html" and "proofs.pdf"

1.2 Compatibility and software versions

At this time, the oldest supported environment is Debian Buster. Generally, the oldest supported environment will be Debian stable or oldstable.

This means that the minimum versions of relevant software are:

- asciidoc 8.6.10
- cbmc 5.10
- python 3.7.3

1.3 Building

The software is built by invoking `make` in the top-level directory. By default, all functions are proven and all forms of documentation are built. Parallelism can be used safely (`make -j4` for a 4-thread system, for instance)

Other targets include `docs`, `pdf`, and `html` to build just (one form of) the documentation; `proofs` to just build the proofs, and `prove-`**`foo`** to just prove **foo**.

In the case of a successful proof, the output of CBMC is left in `.o/`**`foo`**`_proof.txt`. In the case of a failed proof, it is left in `.o/`**`foo`**`_proof.txt.err`.

1.4 Contributing

Submit contributions with github pull requests to https://github.com/jepler/ProvenDelights. Your contributions are subject to the GitHub Terms of Service and must therefore be offered under the Repository License, stated below.

Respect the license of the Hacker's Delight book. This means that code may be incorporated from the book, but prose may not.

1.5 Style

For prose, use professional English writing.

For code, use basic C++; make functions templates where appropriate, and inlines where inappropriate. Use the general coding style of Hacker's Delight:

- 4-space indentation
- no literal tab characters
- open curly braces don't get their own line
- include whitespace in expressions where it improves readability

1.6 License

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

Additionally, all blocks of code marked as ".Implementation" are also covered under the following license (commonly known as the "zlib license"):

This software is provided *as-is*, without any express or implied warranty. In no event will the authors be held liable for any damages arising from the use of this software.

Permission is granted to anyone to use this software for any purpose, including commercial applications, and to alter it and redistribute it freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not claim that you wrote the original software. If you use this software in a product, an acknowledgement in the product documentation would be appreciated but is not required.

2. Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.

3. This notice may not be removed or altered from any source distribution.

2 Basics

2.1 Manipulating Rightmost Bits

2.1.1 Turn off rightmost one

Implementation

```
template<class T> T turn_off_rightmost_one(T x) { return x & (x - 1); }
```

Proof

```
int main() {
    unsigned x = nondet_unsigned() | 1;
    unsigned y = nondet_unsigned() & 31;
    unsigned z = x << y;
    assert(turn_off_rightmost_one(z) == (z ^ (1u<<y)));
}
```

2.1.2 Next with same popcnt

Note that `next_with_same_popcnt` may not be called with 0 (this leads to division by zero).

When it is called with the largest number of a given popcnt, I don't understand the interpretation of the return value, so I don't assert anything about that case.

Implementation

```
template<class T> T next_with_same_popcnt(T x) {
    T s = x & -x;
    T r = s + x;
    return r | (((x ^ r) >> 2) / s);
}
```

Proof

```
int main() {
    unsigned u = nondet_unsigned();
    assume(u != 0);
    unsigned v = next_with_same_popcnt(u);
    unsigned w = nondet_unsigned();

    if(v > u) {
        assume(w > v && w < u);
        assert(proof_popcnt(u) == proof_popcnt(v));
        assert(proof_popcnt(u) != proof_popcnt(w));
    }
}
```

2.2 (no proofs in this section)

2.3 (no proofs in this section)

2.4 (no proofs in this section)

2.5 Average of two integers

2.5.1 Average, rounded down

Implementation

```
template<class T> T average_round_down(T x, T y) {
    return (x & y) + ((x ^ y) >> 1);
}
```

Proof

```
int main() {
    unsigned x = nondet_unsigned();
    unsigned y = nondet_unsigned();
    unsigned z = average_round_down(x, y);
    unsigned w = (uint64_t(x) + y) / 2;
    assert(z == w);
    return 0;
}
```

2.5.2 Average, rounded up

Implementation

```
template<class T> T average_round_up(T x, T y) {
    return (x | y) - ((x ^ y) >> 1);
}
```

Proof

```
int main() {
    unsigned x = nondet_unsigned();
    unsigned y = nondet_unsigned();
    unsigned z = average_round_up(x, y);
    unsigned w = (uint64_t(x) + y + 1) / 2;
    assert(z == w);
    return 0;
}
```

3 Power-of-2 Boundaries

3.1 (No proofs in this section)

3.2 Rounding Up/Down to the Next Power of 2

3.2.1 Greatest power of 2 <= x

Implementation

```
#include <nlz.h>
template<class T>
inline T flp2(T x) {
    return T(1) << (BITS(T) - nlz(x) - 1);
}
```

Proof

```
template<class T>
void prove() {
    T x = nondet<T>();
    assume(x);
    T y = flp2(x);
    assert(y <= x);
    assert(proof_popcnt(y) == 1);
    T z = nondet_unsigned();
    assume(z <= x && y < z);
    assert(proof_popcnt(z) != 1);
}

int main() {
    prove<unsigned>();
    prove<uint64_t>();
}
```

3.2.2 Least power of 2 >= x

This implementation of `clp2` does not return the right result for 0 or for values with no leading zero.

Implementation

```
#include <nlz.h>
template<class T>
inline T clp2(T x) {
    return T(1) << (BITS(T) - nlz(x-1));
}
```

Proof

```
template<class T>
void prove() {
    T x = nondet<T>();
    assume(x && nlz(x));
    T y = clp2(x);
    assert(y >= x);
    assert(proof_popcnt(y) == 1);
    T z = nondet_unsigned();
    assume(z >= x && y > z);
    assert(proof_popcnt(z) != 1);
}

int main() {
    prove<unsigned>();
    prove<uint64_t>();
}
```

4 (no proofs in this chapter)

5 Counting Bits

5.1 Counting 1-bits

5.1.1 Counting 1-bits in a word

Implementation

```
int pop(unsigned x) {
   x = x - ((x >> 1) & 0x55555555);
   x = (x & 0x33333333) + ((x >> 2) & 0x33333333);
   x = (x + (x >> 4)) & 0x0F0F0F0F;
   x = x + (x >> 8);
   x = x + (x >> 16);
   return x & 0x0000003F;
}
```

Proof

```
int main() {
    unsigned u = nondet_unsigned();
    int i = pop(u);
    assert(i == proof_popcnt(u));
}
```

5.2 Parity

5.2.1 Compute Parity

Implementation

```
inline int parity(uint8_t x) {
    x ^= (x >> 4);
    return (0x6996 >> (x & 0xf)) & 1;
}

inline int parity(uint16_t x) {
    x ^= (x >> 8);
    x ^= (x >> 4);
    return (0x6996 >> (x & 0xf)) & 1;
}

inline int parity(uint32_t x) {
    x ^= (x >> 16);
    return parity(uint16_t(x));
}

inline int parity(uint64_t x) {
    x ^= (x >> 32);
    return parity(uint32_t(x));
}
```

The proof for 64-bit inputs takes an infeasibly long time and consequently is disabled.

Proof

```
template<class T> void prove() {
    T x = nondet<T>();
    int px = parity(x);
    int pc = proof_popcnt(x) & 1;
    assert(pc == px);
}

int main() {
    prove<uint8_t>();
    prove<uint16_t>();
    prove<uint32_t>();
//    prove<uint64_t>();
}
```

5.2.2 Add Even Parity in MSB

Implementation

```
inline uint8_t pareven(uint8_t x) {
    return (((unsigned)x * 0x10204081u) & 0x888888ffu) % 1920;
}
```

Proof

```
#include <parity.h>

int main() {
    uint8_t x = nondet<uint8_t>() & 0x7f;
    uint8_t y = pareven(x);
    assert((y & 0x7f) == x);
    assert(!parity(y));
}
```

5.2.3 Add Odd Parity in MSB

Implementation

```
inline uint8_t parodd(uint8_t x) {
    return (((unsigned)x * 0x204081u) | 0x3db6db00u) % 1152;
}
```

Proof

```
#include <parity.h>

int main() {
    uint8_t x = nondet<uint8_t>() & 0x7f;
    uint8_t y = parodd(x);
```

```
    assert((y & 0x7f) == x);
    assert(parity(y));
}
```

5.3 Counting Leading 0s

5.3.1 Number of leading zeroes

Implementation

```
inline int nlz(unsigned x) {
    static char table[64] =
    {32,31,-1,16,-1,30, 3,-1,  15,-1,-1,-1,29,10, 2,-1,
     -1,-1,12,14,21,-1,19,-1,  -1,28,-1,25,-1, 9, 1,-1,
     17,-1, 4,-1,-1,-1,11,-1,  13,22,20,-1,26,-1,-1,18,
      5,-1,-1,23,-1,27,-1, 6,  -1,24, 7,-1, 8,-1, 0,-1};
    x = x | (x >> 1);     // Propagate leftmost
    x = x | (x >> 2);     // 1-bit to the right.
    x = x | (x >> 4);
    x = x | (x >> 8);
    x = x | (x >>16);
    x = x*0x06EB14F9;     // Multiplier is 7*255**3.
    return table[x >> 26];
}

inline int nlz(uint64_t x) {
    unsigned i = x >> 32;
    if(i) return nlz(i);
    return 32+nlz(unsigned(x));
}
```

Proof

```
#include <pop.h>
template<class T>
int nlz_simple(T x) {
    x = x | (x >> 1);
    x = x | (x >> 2);
    x = x | (x >> 4);
    x = x | (x >> 8);
    if(sizeof(T) > 2) {
        x = x | (x >>16);
    }
    if(sizeof(T) > 4) {
        x = x | (x >>32);
    }
    return proof_popcnt(~x);
}

template<class T> void prove(T x) {
    assert(nlz(x) == nlz_simple(x));
}

int main() {
    prove(nondet_unsigned());
    prove(nondet_u64());
}
```

6 Searching Words

6.1 Find First 0-Byte

6.1.1 Find leftmost 0-byte

Implementation

```
#include "nlz.h"
inline int zbytel(unsigned x) {
    unsigned y;
    int n;
    y = (x & 0x7f7f7f7f) + 0x7f7f7f7f;
    y = ~(y | x | 0x7f7f7f7f);
    n = nlz(y) >> 3;
    return n;
}

inline int zbytel_nonlz(unsigned x) {
    unsigned y;
    int n;
    y = (x & 0x7f7f7f7f) + 0x7f7f7f7f;
    y = ~(y | x | 0x7f7f7f7f);
    if(y == 0) return 4;
    if(y > 0xffff) return (y >> 31) ^ 1;
    return (y >> 15) ^ 3;
}
```

Proof

```
int zbytel_ref(unsigned x) {
    if((x >> 24) == 0) return 0;
    if((x & 0x00ff0000) == 0) return 1;
    if((x & 0x0000ff00) == 0) return 2;
    if((x & 0x000000ff) == 0) return 3;
    return 4;
}

int main() {
    unsigned x = nondet_unsigned();
    assert(zbytel(x) == zbytel_ref(x));
    assert(zbytel_nonlz(x) == zbytel_ref(x));
}
```

7 (no proofs in this chapter)

8 (no proofs in this chapter)

9 (no proofs in this chapter)

10 Integer Division By Constants

10.1 (no proofs in this section)

10.2 (no proofs in this section)

10.3 (no proofs in this section)

10.4 (no proofs in this section)

10.5 (no proofs in this section)

10.6 (no proofs in this section)

10.7 (no proofs in this section)

10.8 Unsigned Division by 3

10.8.1 Unsigned division by 3

Implementation

```
inline unsigned udiv3(uint32_t dividend, unsigned *remainder) {
    uint32_t q = (dividend * (uint64_t)0xAAAAAAABu) >> 33;
    if(remainder) *remainder = dividend - q * 3;
    return q;
}
```

Proof

```
int main() {
    unsigned n = nondet_unsigned();
    unsigned r;
    unsigned q = udiv3(n, &r);

    assert(q * 3 + r == n);
    // cbmc is not able to prove this, or anything else I could come up with to
    // fully prove division-by-3, in a reasonable period of time
    //assert(r < 3);
}
```

11 Some Elementary Functions

11.1 (no proofs in this section)

11.2 (no proofs in this section)

11.3 (no proofs in this section)

11.4 Integer Logarithm

11.4.1 Integer log 10

Implementation

```
#include "nlz.h"

// "one table lookup, branch free" (figure 11-12)
inline int ilog10(unsigned x) {
    int y;
    static unsigned table[11] = {0, 9, 99, 999, 9999,
        99999, 999999, 9999999, 99999999, 999999999, 0xFFFFFFFF};

    y = (19*(31 - nlz(x))) >> 6;
    return y + ((table[y+1]-x) >> 31);
}
```

Proof

```
// "simple table search" (figure 11-7)
int ilog10_simple(unsigned x) {
    int i;
    static unsigned table[11] = {0, 9, 99, 999, 9999,
        99999, 999999, 9999999, 99999999, 999999999, 0xFFFFFFFF};
    for(i = -1; i < 11; i++)
        if(x <= table[i+1]) return i;
    assert(0);
}

int main() {
    unsigned u = nondet_unsigned();
    assume(u);
    assert(ilog10_simple(u) == ilog10(u));
}
```

12 (no proofs in this chapter)

13 (no proofs in this chapter)

14 (no proofs in this chapter)

15 Error correcting codes

15.1 (no proofs in this section)

15.2 (no proofs in this section)

15.3 Software for SEC-DED on 32 Information Bits

Implementation The test hamming_exhaustive_012.c in the Hackers Delight code distribution, which exhaustively tests the ~3354×10^9 combinations of 0, 1 and 2 bits is noted to take 24 hours on "linux61". This proof takes about 3s on a core i5 CPU. An improvement of over 25,000x is nothing to sneeze at.

```
#include <parity.h>

inline unsigned checkbits(unsigned u) {
    /* Computes the six parity check bits for the
    "information" bits given in the 32-bit word u. The
    check bits are p[5:0]. On sending, an overall parity
    bit will be prepended to p (by another process).
    Bit Checks these bits of u
    p[0] 0, 1, 3, 5, ..., 31 (0 and the odd positions).
    p[1] 0, 2-3, 6-7, ..., 30-31 (0 and positions xxx1x).
    p[2] 0, 4-7, 12-15, 20-23, 28-31 (0 and posns xx1xx).
    p[3] 0, 8-15, 24-31 (0 and positions x1xxx).
    p[4] 0, 16-31 (0 and positions 1xxxx).
    p[5] 1-31 */
    unsigned p0, p1, p2, p3, p4, p5, p6, p;
    unsigned t1, t2, t3;
    // First calculate p[5:0] ignoring u[0].
    p0 = u ^ (u >> 2);
    p0 = p0 ^ (p0 >> 4);
    p0 = p0 ^ (p0 >> 8);
    p0 = p0 ^ (p0 >> 16);               // p0 is in posn 1.

    t1 = u ^ (u >> 1);
    p1 = t1 ^ (t1 >> 4);
    p1 = p1 ^ (p1 >> 8);
    p1 = p1 ^ (p1 >> 16);               // p1 is in posn 2.

    t2 = t1 ^ (t1 >> 2);
    p2 = t2 ^ (t2 >> 8);
    p2 = p2 ^ (p2 >> 16);               // p2 is in posn 4.

    t3 = t2 ^ (t2 >> 4);
    p3 = t3 ^ (t3 >> 16);               // p3 is in posn 8.
    p4 = t3 ^ (t3 >> 8);                // p4 is in posn 16.
    p5 = p4 ^ (p4 >> 16);               // p5 is in posn 0.

    p = ((p0>>1) & 1)  | ((p1>>1) & 2) | ((p2>>2) & 4) |
       ((p3>>5) & 8)  | ((p4>>12) & 16) | ((p5 & 1) << 5);
    p = p ^ (-(u & 1) & 0x3F);         // Now account for u[0].
    return p;
}
```

```
inline unsigned checkbits_and_parity_bit(unsigned u) {
    unsigned p = checkbits(u);
    p = p | (parity(u ^ p) << 6);
    return p;
}

inline int correct(unsigned pr, unsigned *ur) {
    /* This function looks at the received seven check
       bits and 32 information bits (pr and ur) and
       determines how many errors occurred (under the
       presumption that it must be 0, 1, or 2). It returns
       with 0, 1, or 2, meaning that no errors, one error, or
       two errors occurred. It corrects the information word
       received (ur) if there was one error in it. */
    unsigned po, p, syn, b;
    po = parity(pr ^ *ur);          // Compute overall parity
                                    // of the received data.
    p = checkbits(*ur);             // Calculate check bits
                                    // for the received info.
    syn = p ^ (pr & 0x3F);          // Syndrome (exclusive of
                                    // overall parity bit).
    if (po == 0) {
       if (syn == 0) return 0;      // If no errors, return 0.
       else return 2;               // Two errors, return 2.
    }
                                    // One error occurred.
    if (((syn - 1) & syn) == 0)     // If syn has zero or one
       return 1;                    // bits set, then the
                                    // error is in the check
                                    // bits or the overall
                                    // parity bit (no
                                    // correction required).

    // One error, and syn bits 5:0 tell where it is in ur.
    b = syn - 31 - (syn >> 5);    // Map syn to range 0 to 31.
    *ur = *ur ^ (1u << b);        // Correct the bit.
    return 1;
}
```

Proof

```
void zero_errors() {
    unsigned u = nondet_unsigned();
    unsigned p = checkbits_and_parity_bit(u);
    unsigned v = u;
    unsigned po = parity(p ^ u);
    assert(po == 0);
    assert(correct(p, &v) == 0);
    assert(u == v);
}

void introduce_error(unsigned *p, unsigned nbits) {
    unsigned b = nondet_unsigned();
    assume(b < nbits);
    *p = *p ^ (1u << b);
}

void introduce_errors(unsigned *p, unsigned nbits) {
    unsigned b = nondet_unsigned();
```

```
    unsigned c = nondet_unsigned();
    assume(b < nbits && c < nbits && b != c);
    *p = *p ^ (1u << b) ^ (1u << c);
}

void one_error_p() {
    unsigned u = nondet_unsigned();
    unsigned p = checkbits_and_parity_bit(u);
    unsigned v = u;
    introduce_error(&p, 7);
    assert(correct(p, &v) == 1);
    assert(u == v);
}

void one_error_u() {
    unsigned u = nondet_unsigned();
    unsigned p = checkbits_and_parity_bit(u);
    unsigned v = u;
    introduce_error(&v, 32);
    assert(correct(p, &v) == 1);
    assert(u == v);
}


void two_errors_p() {
    unsigned u = nondet_unsigned();
    unsigned p = checkbits_and_parity_bit(u);
    unsigned v = u;
    introduce_errors(&p, 7);
    assert(correct(p, &v) == 2);
}

void two_errors_u() {
    unsigned u = nondet_unsigned();
    unsigned p = checkbits_and_parity_bit(u);
    unsigned v = u;
    introduce_errors(&v, 32);
    assert(correct(p, &v) == 2);
}

void two_errors_pu() {
    unsigned u = nondet_unsigned();
    unsigned p = checkbits_and_parity_bit(u);
    unsigned v = u;
    introduce_error(&p, 7);
    introduce_error(&v, 32);
    assert(correct(p, &v) == 2);
}

int main() {
    zero_errors();
    one_error_p();
    one_error_u();
    two_errors_p();
    two_errors_u();
    two_errors_pu();
}
```