Formal Proofs of Bit Hacks in Machine Code

https://humam.systems/

Humam Alhusaini

Advisor: Dr. Kevin Hamlen

The University of Texas at Dallas

Humam.Alhusaini@UTDallas

Motivation and Background

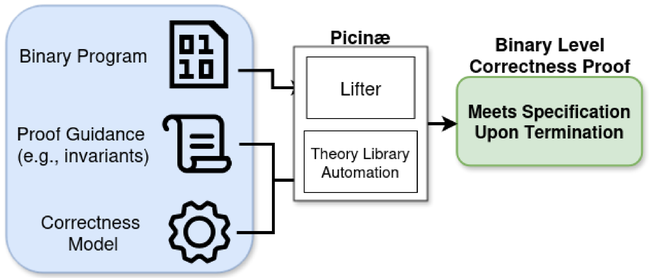
Proving the correctness of leaf functions is **often difficult because they commonly use *bit hacks***, requiring the user to reason about subtle properties of bit-arithmetic. This is why an existing proof of correctness for strlen in *Picinæ* dedicates **120 LOC, or more than half the proof**, to properties concerning the bit hack used in it, which is shown below:

$$((v + \texttt{0xFEFEFEFF}) \oplus v)\ \&\ \texttt{0x01010100} \tag{1}$$

This motivates our **proven library of bit hacks** verified at the machine code level using Picinæ, thus reducing the effort of verifying larger programs by providing **solved cases and proven theorems concerning complex bit-arithmetic**.

Background

Picinæ is a framework within the Rocq interactive theorem prover (ITP) for representing and reasoning over arbitrary machine code using **linear temporal logic and separation logic** [4]. It is able to prove any properties that are parameterized by a history of CPU states, such as **correctness, timing [2], and fault tolerance [3]**.



Results

We proved **14 distinct bit hacks**, listed below according to a popular bit hack article [1]:

1. Compute the sign of an integer: -1 if negative, 0 if not
2. Detect if two integers have opposite signs
3. Compute the integer absolute value (abs) without branching
4. Compute the minimum of two integers without branching
5. Compute maximum of two integers without branching
6. Determining if an integer is a power of 2
7. Conditionally set or clear bit without branching
8. Conditionally negate a value without branching
9. Merge bits from 2 values according to a mask
10. Counting bits set, naïve way
11. Compute parity of a word, naïve way
12. Compute modulus division 2^x without a division operator
13. Compute log base 2
14. Compute log base 10, naïve way

Case Study: Determine if an integer is a power of 2

This function returns true if the input is a power of 2, else false.

```
    cbz     w0, done        // if v == 0 return 0
    sub     w1, w0, #1      // w1 = v - 1
    tst     w1, w0          // test v & (v - 1)
    cset    w0, eq          // w0 = (result == 0)

    done:
    ret
.
```

Original C:

```
bool is_pow2(unsigned int v) {
    return v && !(v & (v - 1));
}
.
```

We proved the below post-condition:

$$(\exists\ i\ .\ 2^i \equiv w_0 \wedge w_0' \equiv 1) \vee$$
$$(\forall\ i\ .\ 2^i \not\equiv w_0 \wedge w_0' \equiv 0)$$

w_0 is modified in the following manner.

$$w_0' = \begin{cases} 1 & \text{if } w_0\ \&\ (w_0 - 1) = 0 \\ 0 & \text{otherwise} \end{cases}$$

In order to prove the post-condition in this case, we proved that

$$x\ \&\ (x - 1) \tag{2}$$

clears the least significant set bit (LSB). In the theorem below, $\text{lsbi}(x)$ finds the index of the LSB in number x and $\oplus$ is bit-wise XOR.

$$\forall x \in \mathbb{N}, \quad x\ \&\ (x-1) = x \oplus (1 \ll \text{lsbi}(x)) \tag{3}$$

The inclusion of $x - 1$ means **we must define what decrement by one means at the bit level**, $\text{bit}(x, n)$ returns true if bit n of x is set, otherwise false.

$$\forall x \in \mathbb{N}^+,\ \text{bit}\Big(x-1,\ \text{lsbi}(x)\Big) = false$$

$$\forall\, x \in \mathbb{N}^+,\ \forall\, i \in \mathbb{N},\ i < \text{lsbi}(x)\ \Rightarrow\ \text{bit}\Big(x-1,\ i\Big) = true$$

$$\forall\, x \in \mathbb{N}^+,\ \forall\, i \in \mathbb{N},\ i > \text{lsbi}(x)\ \Rightarrow\ \text{bit}\Big(x-1,\ i\Big) = \text{bit}\Big(x,\ i\Big)$$

We use the above theorems to solve theorem 3, concluding the first step to verifying the post-condition.

The second step requires that we prove the property that indicates whether a positive number is a power of 2 or not.

$$\forall x \in \mathbb{N}^+,\ x\ \&\ (x-1) = 0 \iff \exists i \in \mathbb{N},\ x = 2^i \tag{4}$$

$$\forall x \in \mathbb{N}^+, \quad x\ \&\ (x-1) \neq 0 \iff \forall i \in \mathbb{N},\ x \neq 2^i \tag{5}$$

These lemmas are true because **powers of 2 are the only positive numbers with only 1 set bit**. Expression (2) clears the only set bit in a power of 2, resulting in 0, and fails to clear all bits in a non-power of 2, resulting in a non-zero value.

Now that we have proved these relevant lemmas, it is trivial to solve the post-condition.

Case Study: Determine the parity of a word

This function returns true if the input has an odd number of set bits; else false.

```
    mov     w1, w0          // copy input v
    mov     w0, #0          // parity

    cbz     w1, done        // if v == 0 return 0

    loop:
    sub     w2, w1, #1      // v - 1
    eor     w0, w0, #1      // toggle parity
    ands    w1, w1, w2      // v = v & (v - 1)
    b.ne    loop            // continue while v != 0

    done:
    ret
.
```

Original C:

```
bool parity(int v) {
        bool p = false;

        while (v) {
            p = !p;
            v = v & (v - 1);   // clear lowest set bit
        }

        return p;
    }
.
```

We verified the implementation against the following post-condition:

$$w_0' = \text{popcnt}(w_0) \bmod 2.$$

The loop semantics can be expressed as follows, where M corresponds to the mutable value of the input register w_0 and w_0' stores the boolean result

$$\textbf{while } M \neq 0: \quad \begin{cases} w_0' \leftarrow \neg w_0' \mod 2 \\ M \leftarrow M\ \&\ (M-1) \end{cases}$$

We placed the following loop invariant on the loop guard:

$$\text{popcnt}(w_0) \bmod 2 = (\text{popcnt}(M) + w_0') \bmod 2$$

Solving this loop invariant **requires we prove expression 2 decrements the number of set bits by 1**. We reuse theorem 3 to prove the below theorem:

$$\forall x \in \mathbb{N}^+,\ \text{popcnt}(x\ \&\ (x-1)) = \text{popcnt}(x) - 1$$

This lemma makes the loop guard trivial, and the loop guard solves the post-condition.

References

[1] Sean Eron Anderson. Bit twiddling hacks, 2005.

[2] Charles Averill. Formally-Verified, Tight Timing Constraints for Machine Code. *PLDI SRC*, 2025.

[3] Charles Averill, Ilan Buzzetti, Alex Bellon, and Kevin Hamlen. LAPSE: Automatic, formal fault-tolerant correctness proofs for native code. *NDSS BAR*, Feb 2026.

[4] Kevin W. Hamlen, Dakota Fisher, and Gilmore R. Lundquist. Source-free Machine-checked Validation of Native Code in Coq. In *Proceedings of the 3rd ACM Workshop on Forming an Ecosystem Around Software Transformation*, pages 25–30, London United Kingdom, November 2019. ACM.