Formal Proofs of Bit Hacks in Machine Code

Humam Alhusaini
Humam.Alhusaini@UTDallas.edu
University of Texas at Arlington
University of Texas at Dallas
United States of America

ABSTRACT

Mission-critical systems depend on bit arithmetic algorithms for specific and efficient calculations. Providing formal guarantees for these algorithms is difficult because it requires reasoning about obscure facts of binary arithmetic. These obscure properties often build off one another, suggesting that a library of solved algorithms would provide the necessary logic to prove the correctness of more complex algorithms. This paper presents proofs of correctness for 14 bit arithmetic algorithms at the raw (stripped) machine code level and a supporting theory library covering subtle properties of binary arithmetic within the Rocq interactive theorem-proving environment.

1 PROBLEM AND MOTIVATION

Bugs in mission critical systems often have catastrophic consequences that are dangerous to people and institutions. The most robust way to prevent bugs is to provide guarantees about a system's behavior through formal verification. Providing these guarantees without assumptions often requires verifying a system from the top down to its leaf functions. These leaf functions frequently use *bit hacks*, or techniques that directly manipulate bits for optimized and specialized behavior.

An example of a crucial leaf function that uses a bit hack is i386 strlen. It uses the following bit hack, which will return a non-zero number if v contains a non-zero byte.

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

The above algorithm relies on subtle bit-arithmetic properties that are difficult to prove. This is why an existing proof for the correctness of i386 strlen in *Picinæ* dedicates 120 LOC, or more than half the proof, to establishing these obscure bit-arithmetic properties [2].

The proof for i386 strlen is an example in a larger trend where significant effort is dedicated to reasoning about "tricky properties of bit-arithmetic" [12]. This motivates our development of a proven library of bit hacks verified at the machine code level using Picinæ, which reduces the effort of verifying larger programs in two ways. First, we are enabling automated reasoning about similar code patterns across projects. Second, these cases provide theorems necessary for the verification of more complex functions and hacks. By verifying these algorithms at the machine code level, we create a solid foundation for the bottom-up verification of mission-critical systems.

1.1 Bottom-up Formal Methods

While similar correctness properties established in our library can be proven at the source level, working with machine code allows for analysis of programs without a source and code that is already low level.

Some mission critical systems don't have their source available for analysis, such as closed source COTS software and legacy code whose source has been lost. Firmware is a particularly important case, as it is present on virtually every computing system, operates at the highest privilege level, and is almost always closed source. The only way to establish formal guarantees for this code is through machine code analysis.

Unlike high level languages like C, assembly is nearly one-to-one with machine code. This means that the effort required to provide correctness properties of machine code is essentially identical to what would be needed to do the same for assembly. Targeting a machine code IR like BAP [7] would also allow a formal analysis tool to support multiple ISAs, while a tool that analyzed assembly at the source would be limited to that ISA with marginal benefit.

2 BACKGROUND AND RELATED WORK

2.1 Picinæ

Picinæ is a framework within the Rocq interactive theorem prover (ITP) for representing and reasoning over arbitrary machine code [12]. Due to supporting linear temporal logic, Picinæ is able to prove any properties that are parameterized by a history of CPU states, such as correctness, timing [3], and fault tolerance [4]. Picinæ is able to do this by symbolically executing Picinæ IL, which is a low level, ISA generic intermediate language for modeling machine code semantics that is formalized in Rocq. It is similar to Ghidra P-code[13], so Picinæ can verify any binary that Ghidra supports.

To analyze a program for partial correctness, the user specifies a precondition, exit points, and invariants. The precondition will usually bind *proof metavariables* to represent the values of CPU elements when the program enters the function. Exit points are addresses where Picinæ will stop symbolically executing the program. An invariant is placed on an address to make an assertion about the state of the program when the symbolic executer encounters the address. An invariant that lies on the same address of an exit point is called the post-condition, and one that lies on the loop guard is called the loop invariant.

Once these preparations are made, Picinæ can begin symbolically executing the program instruction by instruction. When the interpreter reaches a conditional instruction, it branches into two separate cases: one in which the condition holds true and another in which it false. When the interpreter encounters an invariant, its execution will halt, requiring the user to manually check whether the invariant is satisfied. Once all goals have been verified and exit points reached, Picinæ will stop symbolically executing the program, concluding the proof.

2.2 Bit Hacks

Bit hacks are techniques that directly manipulate bits for optimized and specialized behavior. They cover behaviors ranging from basic algebra, like log_{10}, to complex bit manipulations, like Morton numbers [1]. Thus, they appear across many domains, including cryptography [19], compilers [17], machine learning [22], and games [14].

2.2.1 Techniques. Bit hacks often present several functionally equivalent approaches for achieving the same behavior [1], the most notable of which is the lookup table.

Lookup tables are arrays that store precomputed results which are retrieved based on the input. This avoids recalculating values and is one of the primary ways bit hacks achieve constant time. To avoid high memory usage, their length tends to be limited to 256, which corresponds to all possible values in a byte.

Bit hacks that don't take advantage of lookup tables may take advantage of *magic numbers*, which are values with a special property that helps calculate new values. This property may be related to the number's binary representation, as demonstrated in 1, or derived algebraically, as in "Find integer log base 10 of an integer" in this popular article [1].

Bit hacks also sparingly use control-flow statements because branching instructions hinder instruction fetching and parallel execution [23]. Despite this, bit hacks that use control flow statements still appear in important projects [20]. Of the bit hacks we verified, three used a while loop.

2.3 Related Work

Prior work has applied Picinæ to verify crucial utility functions and the bit manipulation algorithms they employ, including ARM strlen and ARM memset [12].

2.3.1 Other Correctness Proofs of Bit Hacks. To the best of our knowledge, no published work specifically targets bit hacks as objects of formal verification. The closest related work includes a widely cited survey of hacks which validated its examples through brute force and the UCLID verification system [18], an analysis of the underlying algorithms using the Z3 SMT solver [9], and a proof of equivalence between C implementations and their naive counterparts using CBMC [21]. None of these works establish formal guarantees at the binary level.

2.3.2 Correctness Proofs of Machine Code. Significant parts of AWS lib-crypto and AWS s2n have been verified at the machine code level [10]. Their verification used Cryptol [6], a DSL for cryptography, to specify mathematical properties that were checked against the machine code generated by the hand written assembly using SMT solvers and Rocq[8, 16].

Large parts of AWS s2n-bignum were verified with a framework similar to Picinæ [15]. However, it differs in that it proved correctness by demonstrating equivalence between optimized and verification-friendly implementations of s2n-bignum routines.

3 CASE STUDIES

We implemented each bit hack as a C function, compiled them to AArch64 using GCC with the `-O3` optimization flag, analyzed the artifact with Ghidra, and lifted the Ghidra P-code to Picinæ IL using a Ghidra script. All bit hacks were verified using Picinæ to guarantee that their post-condition holds over all possible inputs; no proofs were left admitted.

3.1 Determine if an integer is a power of 2

Listing 1 returns 1 if an input is a power of 2; otherwise 0.

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

Listing 1: Power-of-two check using a bit trick

```
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
```

Listing 2: AArch64 assembly generated for Listing 1

We proved the below post-condition, where w_0 and w_0' are proof variables bound to the initial and final value of register `w0` respectively.

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

We focus on proving the above post-condition holds for positive numbers; 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), an operation used in several bit hacks. In the theorem below, definition $\text{lsbi}(x)$ finds the index of the LSB in number x and $\oplus$ is bit-wise XOR.

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

This would be simple to solve if the whole theorem only contained bit-wise operations, as their effects are localized to individual bits. However, the inclusion of $x - 1$ means we must define what decrement by one means at the bit level. The theorems below define this behavior; $\text{bit}(x, n)$ returns true if bit n of x is set, otherwise false.

$$\forall x \in \mathbb{N}^+,\ \text{bit}\big(x - 1,\ \text{lsbi}(x)\big) = false$$
$$\forall\, x \in \mathbb{N}^+,\ \forall\, i \in \mathbb{N},\ i < \text{lsbi}(x) \implies \text{bit}\big(x - 1,\ i\big) = true$$
$$\forall\, x \in \mathbb{N}^+,\ \forall\, i \in \mathbb{N},\ i > \text{lsbi}(x) \implies \text{bit}\big(x - 1,\ i\big) = \text{bit}\big(x,\ i\big)$$

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 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 relevant lemmas, it is trivial to solve the post-condition. That concludes the formal verification of the bit hack "Determine if a number is a power of 2". While the proof was only around 100 LOC, 300 LOC was dedicated to theorems concerning the new definition $lsbi(x)$.

3.2 Computing parity of a word

Listing 3 returns 1 if the input has an odd number of bits set; otherwise 0.

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

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

    return p;
}
```

Listing 3: Parity computation using bit clearing

```
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
```

Listing 4: AArch64 assembly generated for Listing 3

We verified the implementation against the following post-condition, where popcnt(x) denotes the number of set bits in x, w_0 is the input, and w_0' is the output:

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

The case where the input is 0 skips the loop and is trivial, so we focus on proving the post-condition holds for non-zero input. 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:

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

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. This concludes the proof for "Computing parity of a word". Despite being more complicated, this proof only required 150 LOC because of how it used the theorems developed in the previous case study. This demonstrates that theorems from past proofs are often vital in the verification of more complex functions.

4 RESULTS AND CONTRIBUTIONS

We proved 14 distinct bit hacks, and they are 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 by 1 « s without a division operator
(13) Compute log base 2
(14) Compute log base 10, naïve way

The article describes some bit hacks as naïve, in the sense that they are an obvious or less efficient version. Inefficient hacks still appear in crucial projects [20] and verifying more optimized versions falls under future work.

In writing these proofs, we created theorems that filled in gaps within Rocq's standard library and Picinæ's library. These theorems describe fundamental properties of base-2 numbers, including arithmetic shifts, merging according to a mask, clearing the least significant set bit, population count, and negative numbers.

4.1 Future Work

SMT solvers would be particularly effective at solving the theorems proven in this paper, as they can use bit-blasting to reason about obscure bit-arithmetic properties. Integrating Picinæ with SMT-Coq [11] would allow the user to discharge the burden of proof to a compatible SMT solver, speeding up the creation of new proofs. This requires modifying SMT-Coq to translate Picinæ into smt-lib [5] in a way that SMT solvers will be able to reason about.

REFERENCES

[1] Sean Eron Anderson. 2005. Bit twiddling hacks. https://graphics.stanford.edu/~seander/bithacks.html

[2] Charles Averill. 2019. Picinæ i386 strlen Verification. https://github.com/CharlesAverill/Picinae/blob/master/Examples/i386_strlen.v

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

[4] Charles Averill, Ilan Buzzetti, Alex Bellon, and Kevin Hamlen. 2026. LAPSE: Automatic, Formal Fault-Tolerant Correctness Proofs for Native Code. *NDSS BAR* (Feb 2026). https://www.charles.systems/publications/bar2026-camera-ready.pdf

[5] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. 2010. The smt-lib standard: Version 2.0. In *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, UK)*, Vol. 13. 14.

[6] Sally Browning. 2010. Cryptol, a DSL for cryptographic algorithms. In *ACM SIGPLAN Commercial Users of Functional Programming* (Baltimore, Maryland) *(CUFP '10)*. Association for Computing Machinery, New York, NY, USA, Article 9, 1 pages. https://doi.org/10.1145/1900160.1900171

[7] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J Schwartz. 2011. BAP: A binary analysis platform. In *International Conference on Computer Aided Verification*. Springer, 463–469.

[8] Andrey Chudnov, Nathan Collins, Byron Cook, Joey Dodds, Brian Huffman, Colm MacCárthaigh, Stephen Magill, Eric Mertens, Eric Mullen, Serdar Tasiran, et al. 2018. Continuous formal verification of Amazon s2n. In *International Conference on Computer Aided Verification*. Springer, 430–446.

[9] Martin Di Paula. [n.d.]. Verifying some bithacks. https://book-of-gehn.github.io/articles/2021/05/17/Verifying-Some-Bithacks.html

[10] Mike Dodds. 2023. The Impact of Provable Security: AWS and Supranational. https://www.galois.com/articles/the-impact-of-provable-security-aws-and-supranational

[11] Burak Ekici, Alain Mebsout, Cesare Tinelli, Chantal Keller, Guy Katz, Andrew Reynolds, and Clark Barrett. 2017. SMTCoq: A Plug-In for Integrating SMT Solvers into Coq. In *Computer Aided Verification*, Rupak Majumdar and Viktor Kunčak (Eds.). Springer International Publishing, Cham, 126–133.

[12] Kevin W. Hamlen, Dakota Fisher, and Gilmore R. Lundquist. 2019. Source-free Machine-checked Validation of Native Code in Coq. In *Proceedings of the 3rd ACM Workshop on Forming an Ecosystem Around Software Transformation*. ACM, London United Kingdom, 25–30. https://doi.org/10.1145/3338502.3359759

[13] Brian Knighton and Chris Delikat. [n.d.]. Ghidra - Journey from Classified NSA Tool to Open Source.

[14] Eric Lengyel. 2011. Bit Hacks for Games. In *Game Engine Gems 2*. CRC Press, 385.

[15] Denis Mazzucato, Abdalrhman Mohamed, Juneyoung Lee, Clark Barrett, Jim Grundy, John Harrison, and Corina S. Păsăreanu. 2025. Relational Hoare Logic for Realistically Modelled Machine Code. In *Computer Aided Verification*, Ruzica Piskac and Zvonimir Rakamarić (Eds.). Springer Nature Switzerland, Cham, 389–413.

[16] Yan Peng. 2024. Formal Verification of AWS-LibCrypto. https://sos-vo.org/system/files/2024-05/Formal%20Verification%20of%20AWS-LibCrypto_1.pdf

[17] Alex Rønne Petersen. 2024. Zig Language Library: lib/libcxx/libc/src/__support/FPUtil/FPBits.h. https://github.com/ziglang/zig/ Line 207, commit 738d2be.

[18] Sanjit A Seshia and Pramod Subramanyan. 2018. UCLID5: integrating modeling, verification, synthesis and learning. In *2018 16th ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*. IEEE, 1–10.

[19] Brian Smith. 2024. AWS-LC: montgomery_inv.c. https://github.com/aws/aws-lc/ Line 72 - 150, Commit 6144be8.

[20] Linus Torvalds and contributors. 2024. Linux Kernel: fsi-master-i2cr.c. https://github.com/torvalds/linux/ Line 39 - 49, Commit 0257f64.

[21] ts00ey. 2023. verifying the abs value function. https://ts00ey.bearblog.dev/verifiying-abs/

[22] Thomas Walther. 2024. TensorFlow: irfft2d.cc. https://github.com/tensorflow/tensorflow/ Lines 53, Commit b8dba19.

[23] Henry S. Warren. 2013. *Hacker's Delight* (2nd ed ed.). Addison-Wesley, Upper Saddle River, NJ. See p. xv for cons of branching instructions.