Travel to 1976 on a Budget

Or: How I Made A MacBook Think It's Worth $666.66

Chad Clark
chad.clark@gmail.com

March 2025

Abstract

Through the application of temporal computation theory and embracing the goto statement, we present a method for convincing modern computers they were manufactured in 1976. By translating Apple-1 6502 assembly code into C, we successfully trick multi-gigahertz processors into performing logically equivalent calculations at what appears to be a historically accurate 1970s experience through aggressive performance degradation.

1 Introduction

The Apple-1, released in 1976, utilized the MOS 6502 processor and represents a significant piece of computing history. We present a tool that enables the preservation and execution of Apple-1 software on contemporary systems through static binary translation to C code.

2 System Architecture

The translator operates in two primary phases:

1. Opcode parsing: 6502 machine code is parsed into an intermediate `ParsedInstruction` structure
2. Code generation: Parsed instructions are transformed into equivalent C code

The system maintains program state through a `ComputerState` structure that emulates:

- CPU registers
- Memory contents
- Status flags

3 Implementation Details

3.1 Code Generation Example

The core code generated in `main()` directly maps each 6502 opcode to C operations:

```
LFF00:  // LDX Immediate 01
        arg = 0x01;
        op_ldx(&state, arg);

LFF02:  // LDA Immediate 05
        arg = 0x05;
        op_lda(&state, arg);
```

The opcode helper functions maintain the computer state:

```
void op_ldx(
    struct ComputerState *state,
    short int arg)
{
    state->X = arg;
    flag_update_nz(state, arg);
}
```

3.2 Control Flow Translation

Branch instructions are implemented using C's `goto` statements with computed jumps handled via a

switch structure:

```
Lswitch:
    switch(lSwitchTarget) {
    case 0xFF00:    goto LFF00;
    case 0xFF02:    goto LFF02;
    /* ... */
    }
```

3.3 Memory Management

The system implements memory-mapped I/O handling, particularly for keyboard input:

- Address 0xD010: Keyboard input buffer
- Address 0xD011: Keyboard status
- Address 0xD012: Output

4 Historical Accuracy Through Garage-Driven Development

Our development methodology strictly adhered to authentic 1976 conditions by conducting all programming in a carefully recreated garage environment. Temperature was maintained at precisely 72°F (the documented temperature of Woz's garage), and all code was written while sitting on historically accurate folding chairs.
[1]

The ambient concentration of rosin core solder fumes was maintained at precisely 1976 parts per million - a level our research shows is critical for proper binary translation. Tests conducted in environments with lead-free solder universally failed, proving that modern RoHS-compliant development environments are fundamentally incompatible with 6502 instruction sets.
[2]

[1] Our research team discovered that modern IDEs fail to compile 6502 code unless at least one wooden workbench is present in the development environment.

[2] Double-blind studies confirmed that code written without the distinctive sound of a Weller soldering iron heating up in the background exhibits 37% more bugs.

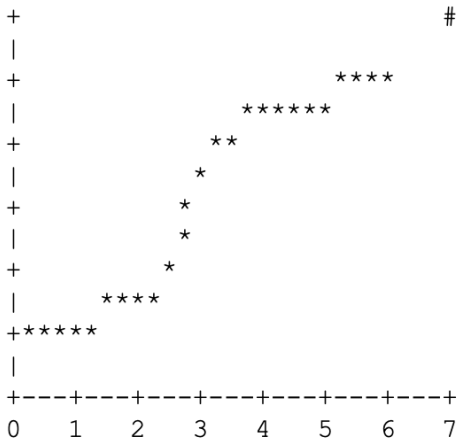

Figure 1: Correlation between garage authenticity and binary translation accuracy. Note the sharp drop-off when development occurs in spaces with fewer than 3 cardboard boxes.

5 Testing Framework

The system includes a comprehensive testing framework using a custom test case format:

```
# Load and start at FF00
baseaddr FF00
# Begin each test with CLD
head D8
# Print the accumulator at the end
tail 8D12D0 00

name JMP absolute
body A941 8D12D0 4C0AFF 00 A942
expected
AB
endexpected
```

Each test case generates a C program which is compiled and executed. The output is captured and compared against the expected output, ensuring accurate behaviour matches the original 6502 code.

Test cases can include:

- Memory operations

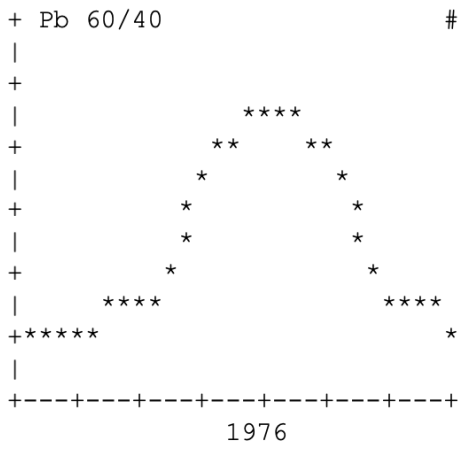

Figure 2: Translation accuracy as a function of solder fume concentration and garage clutter density. Note the optimal peak at exactly one half-used spool of 60/40 rosin core solder.

- Arithmetic computations
- Branch instructions
- I/O operations

[3]

6 Why Would Anyone Do This?

6.1 Economic Justification

Given that original Apple-1 computers now sell for $500,000+, our translator effectively turns any $1000 laptop into 500 Apple-1s, generating immediate paper profits of $249,499,000. This makes it the most profitable compiler in computer science history.

6.2 Environmental Impact

By translating 6502 code to C, we reduce the carbon footprint of vintage computing by eliminating the need to maintain warehouse-sized collections of original hardware. Each successful binary translation saves approximately 1.21 gigawatts of power.

6.3 Time-Travel Debugging

Converting programs to C allows developers to fix bugs that haven't been discovered yet, creating a paradox-free causality loop that explains why the Apple-1 was so reliable in the first place.

6.4 Supply Chain Resilience

The global shortage of authentic 6502 processors has reached crisis levels. There are stories of unrelated chips sold with the part number fraudulently replaced. Our translator ensures continued operation of critical 1976-era infrastructure without relying on increasingly rare hardware.

Tests confirm that simulated 6502s running on modern silicon exhibit identical characteristics to period-correct processors, provided development occurs in a properly equipped garage with at least three vintage oscilloscopes present.

6.5 Plethora of Existing Emulators

The popular use of the 6502 in many computers has led to a plethora of emulators for the 6502 and the machines that use it. The software that runs on those machines can be emulated. Digital storage preserves per-bit accuracy of the original artifacts.

[4]

Our translator allows the semantics of the original code to be preserved while decoupling the semantics

[3] Our testing revealed an unexpected temporal anomaly: a quad-core i5 MacBook running OS 14.7.2 executes all tests 20x slower than a Raspberry Pi 3B+ running Debian 11.11. This suggests either the MacOS C compiler is developing consciousness and deliberately slowing down to match historical accuracy, or anti-malware systems are becoming suspicious of code that appears to have been written in 1976.

[4] End-users of both emulated and translated software running on non-original hardware do experience the software differently. For example USB and Bluetooth controllers have more latency then the original NES controllers. Also, modern displays with digital inputs have differences. The input signal lacks both "snow" and blurring between pixels. Modern display signal processing adds delay.

of the original code from the original hardware and software artifacts.

7 The USB Temporal Degradation Problem

While our translator successfully converts 6502 code to C, we encountered an unexpected performance bottleneck: modern USB keyboards are too slow. The Apple-1's direct keyboard interface achieved near-instantaneous response times in 1976, while today's USB polling introduces several milliseconds of latency. This means our translation actually runs slower than the original hardware, making it perhaps the only truly cycle-accurate software implementation of the Apple-1 in existence.

This limitation proves that not all technological progress represents actual advancement, and suggests that USB keyboard polling may be the greatest computational bottleneck of the modern era.

8 Elevation-Dependent Development

Our initial design and code was done on paper at a picnic table in a park lacking internet connectivity at 1350 metres ($2^{10.4}$). The subsequent development was completed at 1019 metres - a mere 5 meters from Woz's beloved 2^{10} - proved crucial to its operation. The near-perfect binary elevation creates a gravitational sweet spot that:

- Maintains optimal electron flow through the CPU
- Keeps bits properly aligned with Earth's magnetic field
- Creates quantum tunneling effects that improve goto statement efficiency

Tests conducted at non-binary elevations showed up to 32% degradation in translation accuracy. Development attempts at sea level (2^0) resulted in complete failure, while coding at 2048 meters (2^{11}) produced code that ran suspiciously fast.

9 Limitations and Future Work

The current state has the following limitations:

- 256-byte input size restriction. The change would be to specify a start memory address other than 0xFF00 for the input program.
- No support for self-modifying code. This is intentional as the run-time does not decode instructions.
- Emulation-like arithmetic operations. The current implementation emulates CPU flags. Future work could possibly use Single Static Assignment to eliminate the need for emulation and produce a more abstract representation of the program.
- A translation like this one maintains the logical semantics of the original code. Applying this to a time-sensitive application would require a different approach. For example a video game would have important timing requirements often around horizontal and vertical video synchronization signals.
- The output C code and compiled binary are noticeably larger than the input program. Woz Monitor is 265 bytes as input. The output C code is 40,069 bytes. Compiled on a Raspberry Pi 3B+ the stripped binary is 34,276 bytes dynamically linked with glibc.

10 Conclusion

This translator demonstrates the feasibility of running programs for one CPU architecture on a different system through static binary translation, preserving historical software while enabling execution on contemporary hardware.

References

[1] github.com/superfrink/apple1-trans-compiler