A Behavioral Approach to 6502 Emulation *

M. J. Coppola, N. S. Shelby, L. E. Carleton
Computer Science 330
Siena College
Loudonville, NY, 12211

Abstract

An abstract goes here. We made a 6502 emulator that was partially implemented and easy to extend. there were some results and we're really proud. Or something along those lines.

1 Overview

The RP2A03 is the CPU found in the Nintendo Entertainment System (NES), launched October 18th, 1985 [1, 2]. This processor was based off the MOS Technology 6502, differences being a lack of a decimal mode and the inclusion of the Audio Processing Unit (APU) [1]. Our goal is to create a partial implementation of an emulator for this CPU designed to be easily extended to a simple NES emulator.

The emulator was designed with the NES in mind. That being said, we only emulate the behavior of the 6502, not precicely the actual hardware. This serves us a huge simplification in implementation. Official operations of the 6502 only need to be calculated, and not emulated. We opted in for calculating the output of the operations immediately, and then waiting the number of clock cycles that operation took to execute.

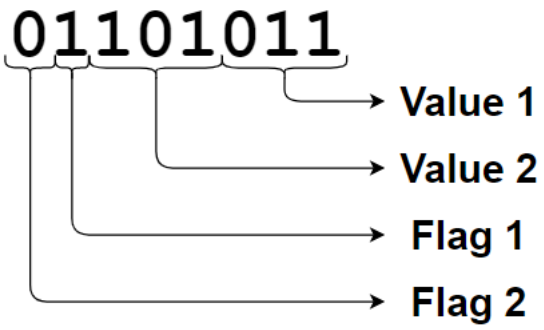
Overview to be continued...

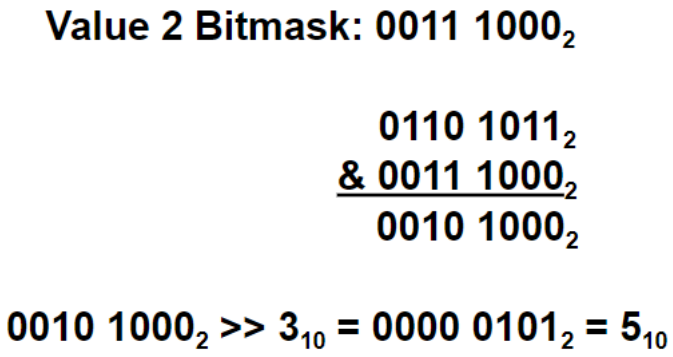
*This work was completed in partial fulfillment of the final project requirement for Computer Science 330 at Siena College, Fall 2020.

2 Bitwise Operations

Bitwise operations are major aspect of making an efficient emulator. Supose we have an 8-bit value. We typically express this value as a number, or possibly two hexidecimal digits. One way the 6502 uses 8-bit numbers is by defining parts of the binary expression as flags or smaller width values, shown in figure 1a. In emulation, it's important that we can extract information from this 8-bit number, as well as convert information we already have into information we can store into this number. To achive this, we use bitwise operations.

To extract information from a binary number, we would first need to isolate that information using a bitmask. We can create a bitmask by creating a new 8-bit number that has 1's placed over the important digits, and 0's on digits that are not. With this mask, we can use a bitwise AND operation (&) and have a result that only includes the digits defined by the bitmask shown in





(a) A single binary value representing multiple values. (b) Using a bitmask to extract values from a binary number.

figure 1b [3].

Inorder to make this value useful to us, we may have to then use a bitwise shift-left ($>>$) to push the important digits to the least signifigant order of the digit. By doing this, the entirety of the resulting 8-bit value numerically represents the value stored in the section of the original 8-bit value [3].

A similar process is used for setting bits as well. We create a bitmask for the bits we want to set and or it with our value. To unset, we can & with the inverse (˜) of our mask. We can toggle/flip bits with a mask that is XOR'd (ˆ) with our value.

References

[1] Cpu. wiki.nesdev.com/w/index.php/CPU, 2019. Accessed: 2020-11-17.

[2] Nintendo entertainment system. https://en.wikipedia.org/wiki/Nintendo_Entertainment_System, 2020. Accessed: 2020-11-17.

[3] O. Lawlor. Bits and bitwise operators. https://www.cs.uaf.edu/courses/cs301/2014-fall/notes/bits-bitwise/, 2014. Accessed: 2020-11-17.