8 Bit Computer Reverse Engineering

James Riley Dorough

Liberty University

March 14th, 2021

**Intro**

This paper covers relevant material to the 8-bit Logisim based functional computer included in the files of our CSIS 342 course. Specifically, this paper dissects the operation of the 8-bit computer to include its opcodes and each of their purposes. Additionally, this paper focuses on answering the question of Turing completeness against the collaboration of all the opcodes that the 8-bit computer currently has access to. As a total, this paper provides a strong introduction to the function of opcodes in computers and how they relate to the requirements of a Turing complete machine.

**Opcodes**

Opcodes act control systems based in a hexadecimally quantifiable format. This section of the paper outlines each of the twelve opcodes that the 8-bit computer accepts and can operate through. Each opcode is defined on its own. However, all pertinent information about the operation, functionality, and structure of the opcode is collected from and attributed to *The Western Design Center Inc.*’s 6502 datasheet (The Western Design Center Inc., 2018). This paper includes the lack of an opcode as well to ensure that all its operations are fully understood. Each opcode is visually represented or noticeable by the 12 busses output from the instruction decoder.

**00 – No Operation**

This code has no operation or functionality for the 8-bit computer. However, it does allow for a continuous transition through memory locations. Parsing through memory while intending the hexadecimal 00 to be a valid opcode will result in the data as an operand being treated as an opcode itself. This information is not quite beneficial to programing but can be beneficial to avoiding and understanding logical errors.

**01 – LDX – Load the X Register with Memory**

This was a relatively direct opcode to discover. As you parse through memory the instruction register picks up the op code one cycle after passing over its location in memory. It takes the data value in memory directly after the opcode. The data value to be stored in the X register is stored over the next two clock cycles. After this the value remains in the X register until another instruction manipulates the X register. This opcode is denoted by the hexadecimal 01.

**02 – LDY – Load the Y Register with Memory**

This was a relatively direct opcode to discover. As you parse through memory the instruction register picks up the op code one cycle after passing over its location in memory. It takes the data value in memory directly after the opcode. The data value to be stored in the Y register is stored over the next two clock cycles. After this the value remains in the Y register until another instruction manipulates the Y register. This opcode is denoted by the hexadecimal 02.

**03 – LDA – Load the Accumulator with Memory**

This was a relatively direct opcode to discover. As you parse through memory the instruction register picks up the op code one cycle after passing over its location in memory. It takes the data value in memory directly after the opcode. The data value to be stored in the accumulator is stored over the next two clock cycles. After this the value remains in the accumulator until another instruction manipulates the accumulator. This opcode is denoted by the hexadecimal 03.

**04 – ADC – Add Memory to the Accumulator with Carry**

Understanding this opcode was the hardest to understand as it requires multiple aspects before it can be used, it is a longer series of instruction and operand, and it does not work on this 8-bit computer. This opcode should require the user to specify a data value to be added to the accumulator by setting the data in memory right after the opcode. The 8-bit computer recognizes the opcode. However, after it picks up the data value to be added in the operand buffer, it simply dumps the value and does not actually add it. This opcode is hexadecimal 04.

**05 – SEC – Set the Carry Flag**

This opcode required a bit more experimentation to fully discover. It primarily sets the carry flag to a one after one memory location past the instruction call. What caused a drawback was that in certain situations such as a reset while it is set to one will cause the function to toggle instead of locking the flag to one. This can cause more issues with any other instructions and a disconnect in the functionality of the clear carry flag instruction. This opcode is denoted by the hexadecimal 05.

**06 – STA\* – Store the Accumulator’s Data in Memory (Degenerate)**

This opcode required an external reference to verify its operation. It acts similarly to the store the accumulator into memory opcode (0C). However, it does not take in an operand to specify which memory location to store the accumulator’s data. Instead, the opcode instructs the 8-bit computer to store the accumulator’s data exactly two memory locations from the opcode’s location. This opcode is denoted by the hexadecimal 06.

**07 – BCS – Branch on Carry Flag Set**

This opcode behaves very similarly to the unconditional jump. The main difference is that to jump to a new location the instruction first checks to make sure that the carry flag is set to one. If it is, the opcode takes the value from the next memory location and moves the program counter to that location. If the carry flag is not set, the value is ignored, and the computer continues parsing through memory. This opcode is denoted by the hexadecimal 07.

**08 – CLC – Clear the Carry Flag**

The clear carry flag opcode was only possible to discover after the set flag opcode was recorded. To properly use this opcode, you first must set the carry flag. To use both the clear and set carry opcodes you must include them in any instruction a memory location with an even value. Otherwise, the computer fails the read the opcode and becomes offset by a code or may pick up a data value as the next instruction set. This opcode is denoted by the hexadecimal 08.

**09 – TAX – Transfer the Accumulator to the X Register**

This opcode took the longest time of all the instructions to completely discover and understand. To use it you must first assign a data value from memory to the accumulator. Then you can use this opcode to transfer the value saved in the accumulator to the X register. Although, this opcode does not transfer the data, but instead copies it to the X register. The three hex numbers can all boarder each other in a LDA, data, TAX format. This opcode is denoted by the hexadecimal 09.

**0A – STX\* – Store the X Register’s Data in Memory (Degenerate)**

Just the same as the degenerate version of store the accumulator into memory, this opcode required reference to the external instructions. This computer does not have a non-degenerate version of this instruction. It does not take in an operand to specify which memory location to store the accumulator’s data. Instead, the opcode instructs the 8-bit computer to store the accumulator’s data exactly two memory locations from the opcode’s location. This opcode is denoted by the hexadecimal 0A.

**0B – JMP – Unconditional Jump**

The unconditional jump is a simple opcode to discover. Its operation requires only one operand to function properly. The unconditional jump is labeled as “JMP” and simply takes a single memory location as an operand and moves the program counter to that selected memory location. The opcode must be in an even memory location to be called, otherwise the computer treats it as a data value with no opcode to use it. This 8-bit computer implements the unconditional jump as opcode hexadecimal number 0B.

**0C – STA – Store the Accumulator’s Data in Memory**

This is a more complex version of the degenerate store the accumulator’s data. This opcode allows the user to specify a location in memory to store the data currently held in the accumulator. Instead of automatically storing the data two memory locations after the instruction call, the opcode takes the value directly after it as the memory location to store the accumulator’s data. This 8-bit computer implements the unconditional jump as opcode hexadecimal number 0C.

**Turing Completeness**

**Definition**

To answer the question of touring completeness against the given opcodes, we first must define and understand what we are trying to achieve by working towards a Turing complete machine. There is no better way to gain an understanding of Turing completeness than from the author of the theory and the proofs himself. In Allan Turing’s 1936 paper on the Entscheidungsproblem, he claimed that such a complete machine can theoretically exist. This can only be accomplished should enough memory to operate such a devices instruction also exist. His proof lies in the method of suppling instructions to the machine. Allan Turing defines the method of passing through instructions as passing over a paper tape, divided into an even number of squares, and marked with symbols to label the instructions to be performed. The machine is only able to perceive one instruction at a time and operates off the pervious instructions (Turing, 1937, p. 231-235). In a shorter more distinct description of Turing completeness and its application to modern computing machines; a Turing complete machine is fully able to simulate a machine which can solve or perform any algorithm supplied to it given enough time, memory, regardless of complexity, and regardless of data structures (Reprintsev 2018, p. 235-236). With this understanding of Turing complete machines, we can begin to dissect the 8-bit computer referenced in this paper and its Turing completeness.

**Relevance**

Does the 8-bit computer included in this lab and paper, as it currently exists, fulfill the requirements to be a Turing complete machine? The short answer is no. As it currently exists the 8-bit computer does not exist as a Turing complete machine. The 8-bit computer is very close to meeting the requirements. However, it is missing a couple key factors that allows it to be truly Turing complete. The 8-bit computer currently can not add values due to the perception of an error in logic or a bug. It also cannot operate in negative numbers. This means the computer can not subtract numbers either by means of adding negatives or subtracting positives. While complexly and in efficiently, the 8-bit computer can move values into a second “X” register to allow for the possibility of comparisons. The only way to store the data from the X register is to place it degenerately two memory locations away from the instruction call. If this were not the case, the accumulator would be the only section on the instruction “tape” that input or output to memory. Additionally, the computer can not perform any comparisons in its current state. There does exist a branch if carry flag is set instruction, but that is a direct command and is not comparing values. None of the registers are compared against each other or memory values either. This means that the computer must be predefined with all the required instructions in a specific order before beginning to complete an algorithm. This defeats the purpose of the algorithm as you have all but solved the last value output by the algorithm.

Should all these issues be solved, the 8-bit computer can be considered touring complete. At least, only if sufficient to infinite memory, power, and time be given. However, the caveat here is that there must be a finite number of “symbols” (instructions), a starting state, a halting state, a blank symbol, and a set of input symbols to a partial function (Klein, 2017, p. 1-2). This can be accomplished with the inclusion of each of the previously addressed missing aspects. There are, however, some proposed issue with Turing complete computing machines. One of which should be abundantly clear. Namely with a complex enough problem or algorithm, can humanity truly supply enough memory to reach a solution? Additionally, should a system operate on a Turing complete machine, it is bound to break down and degrade or possibly overheat given that theoretically Turing complete machines should be able to operate with infinite time. The access and capability that a true Turing complete machine has poses a critical security risk.

**Turing Complete Risk**

The work of Homescu, Stewart, Larsen, Brunthaler, and Franz outlines a cyber security risk that a Turing complete machine contains. Their claim is that if a machine is to be able to simulate any system and solve any problem, it must have access to the computing device’s memory. If an application is written for a Turing machine, it could have vulnerabilities that a bad actor could use to pry sensitive data from unsecured data storage or data manipulation systems. An example would be a malicious actor could manipulate the program to alter where memory is stored so that it is more easily accessed. The authors mention this regarding pushing and popping data to a stack, if data is pushed to the stack and then the return address is changed, as is the case in a return-into-libc attack, the bad actor could send sensitive data to a new location that is unsecured or fails to pass through proper encryption systems when being stored (Homescu, 2012, p. 1-2). Keeping this in mind, computing systems will only grow more complex and harder to fully secure. If a truly Turing complete machine for pseudo-infinitely large problems is created, there is a dangerously high probability that it can be exploited at some level. This 8-bit computer shows promise for meeting the baseline requirements of a Turing complete machine. However, proper system security should remain in the forefront of the developer’s mind.

**References**

Homescu, A., Stewart, M., Larsen, P., Larsen, S., & Franz, M. (2012). Microgadgets: Size Does Matter in Turing-Complete Return-Oriented Programming. *6th Workshop on Offensive Technologies 12*, 1–3. https://www.usenix.org/system/files/conference/woot12/woot12-final9.pdf

Klein, D., & Rendsvig, R. (2017). Turing Completeness of Finite, Epistemic Programs. *ArXiv:1706.06845*, *1*, 1–4. https://arxiv.org/pdf/1706.06845.pdf

Reprintsev, A. (2018). Turing Completeness. *Oracle SQL Revealed*, 235–242. https://doi.org/10.1007/978-1-4842-3372-6\_10

The Western Design Center Inc. (2018). 65xx W65C02S 8–bit Microprocessor. *The Western Design Center Inc.*, 21-22.

Turing, A. M. (1937). On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, *s2-42*(1), 230–265. https://doi.org/10.1112/plms/s2-42.1.230