



Satakunnan ammattikorkeakoulu
Satakunta University of Applied Sciences

JUSSI-PEKKA AALTONEN

The software emulation of the MOS 6502 microprocessor

DEGREE PROGRAMME IN ELECTRICAL AND
AUTOMATION ENGINEERING
2023

ABSTRACT

Aaltonen, Jussi-Pekka: The software emulation of MOS 6502 microprocessor
Bachelor's thesis

Electrical and Automation engineering

March 2023

Number of pages: 75

While processor technology has advanced greatly during the past 40 decades, the basic working principle of processors has not changed drastically during this time. However, the more efficient processors have made it possible to build computers and other devices that that principle has become more and more obscure for the end user or even a programmer.

The aim of this thesis was to create a learning material for people new or unaccustomed to processor technology and how processors work in general by examining and explaining the functionality of a simple 8-bit processor from the end of the 1970s. The thesis is divided in two parts – the theory part, which consists of a written learning material and the practical part consisting of an application that allows one to examine how and when the contents of registers in the processor and the memory connected to it change when the processor is given machine language instruction.

The written part is based on information from various literary and online sources, and application was written in C++ using a variety of source material describing the functionality of the processor. Once finished, the application was tested with a well known and widely used test suite created for the processor. As a result the reader can use the knowledge gained from the written part to create custom programs to run on the emulator.

Keywords: Computer science, Processor, Emulator, Assembly language, Machine language, 6502, Binary, Hexadecimal

CONTENTS

1 INTRODUCTION	5
2 NUMERAL SYSTEMS AND ARITHMETIC	7
2.1 Notation	7
2.2 Binary numbers	7
2.3 Hexadecimal numbers	10
2.4 Bitwise operations and Boolean logic	10
2.5 Representing negative numbers	13
2.6 Arithmetic operations	14
3 THE STRUCTURE AND FUNCTIONALITY OF MOS 6502	16
3.1 Processor clock	17
3.2 Buses	18
3.3 Registers	19
3.4 Accumulator	19
3.5 Index registers	19
3.6 Status register	20
3.6.1 Carry flag	20
3.6.2 Zero flag	20
3.6.3 Interrupt flag	21
3.6.4 Decimal Flag	21
3.6.5 Break Flag	22
3.6.6 Overflow Flag	22
3.6.7 Negative flag	23
3.7 Program counter	23
3.8 Stack pointer	23
3.9 Interrupts	23
3.10 Memory and memory handling	26
3.10.1 Types of memory	28
3.10.2 Addressable memory	30
3.10.3 Storing data	31
3.10.4 Stack	32
4 PROGRAMMING THE MOS 6502	36
4.1 Programming languages	36
4.1.1 Interpreted languages	36
4.1.2 Compiled languages	37
4.1.3 Assembly and machine language	38

4.2 Addressing modes.....	40
4.2.1 Accumulator and implied mode	40
4.2.2 Absolute and zero page mode	40
4.2.3 Absolute indexed and zero page indexed.....	41
4.2.4 Immediate mode	42
4.2.5 Indirect mode	42
4.2.6 X indirect mode.....	43
4.2.7 Indirect Y mode.....	43
4.2.8 Relative mode.....	43
4.3 6502 instruction set	44
4.3.1 Variable assignment	44
4.3.2 Arithmetic operations	45
4.3.3 Logical operations.....	45
4.3.4 Comparison operations.....	46
4.3.5 Data transfer operations	46
4.3.6 Flag operations	47
4.3.7 Control flow operations	47
4.3.8 Stack operations	49
4.3.9 Transfer operations.....	49
4.3.10 NOP.....	50
4.3.11 Undocumented opcodes	50
5 THE DEVELOPMENT PROCESS OF THE EMULATOR	52
5.1 Tools used.....	52
5.2 Development	53
5.2.1 Variables.....	53
5.2.2 Methods.....	55
5.3 Design	57
5.4 Testing.....	61
6 USING THE EMULATOR.....	63
6.1 Controls and commands.....	64
6.2 Loading custom programs	65
7 CONCLUSION	67
APPENDIX 1: THE 6502 OPCODE MATRIX.....	68

1 INTRODUCTION

The aim of this thesis is to provide a learning material on processor technology by the means of emulating the 6502 microprocessor, which is a well-known and widely used 8-bit processor originally designed by MOS technologies in the year 1975, and which is still manufactured almost 50 years later. The emulator application offers an insight to the inner parts of the processor by visualizing the state and contents of its registers, buses, and memory, and in addition the written part of the thesis handles the subjects needed to understand the functionality of a processor and the emulator, such as machine language, binary and hexadecimal numbers, interrupts, memory, and registers. As the thesis is mainly aimed at an audience with an interest in technology and to keep the scope of the thesis manageable, these topics are not handled extensively but an attempt has been made to provide at least a precursory knowledge on these topics to those who are not familiar with them.

As the technology has moved from the 8-bit to 64-bit processors, the hardware side of computer science has been so abstracted from the end user, that even many programmers don't necessarily know how processors work anymore. While the processor in question is nearly 50 years old, the underlying technology has changed surprisingly little. The modern processors are magnitudes faster, can address more memory and have a larger set of registers than the 6502, but the principle is still pretty much the same.

While knowledge of the low level operations of hardware devices or processors is not by any means necessary to become a proficient programmer these days, a more in-depth knowledge is useful in embedded software or device driver development, and an argument can be made that understanding the functionality of one processor carries over when learning of other processors.

Due to its simple and easy to grasp design, an 8-bit processor such as the 6502 makes it an ideal learning tool for a beginner.

2 NUMERAL SYSTEMS AND ARITHMETIC

2.1 Notation

As throughout this thesis different numeral systems are used, a distinct notation for each is needed to avoid confusion. As there is no definite standard for expressing hexadecimal or binary numbers, the most common one to appear in 6502 assembly language was chosen. Thus, an assumption can be made that the figures in this text prefixed with a '\$' are hexadecimal, the ones prefixed with a '%' are binary numbers, and that the decimal numbers do not have a prefix. This same applies to assembly code examples, although these may have an additional prefix of a '#'. This is only to make a distinction between an *address* and a *literal value*. I.e., a value of \$FF in assembly source code signifies the address \$00FF, and a value of #FF signifies a hexadecimal value of \$FF, or -1 or 255 in decimal, depending on the situation – see section 2.5.

This notation is also used in the emulator application with one exception – the memory contents shown on the left hand of the screen will *always* be shown in hexadecimal, but without any prefixes. This is a common practice in applications that show data in hexadecimal form, and no reason was found to stray from this.

To indicate a range, the standard notation of a closed interval is used. A closed interval includes both its limit points, and it is defined by using square brackets. For example, [0,255] would indicate a range of values from 0 to 255, with the lowest value of 0 and highest value of 255. To keep things simple, no other intervals will be used in this thesis. (Wikipedia, 2022d)

2.2 Binary numbers

Binary (or base-2) system is a way of presenting numbers and figures with only two numerical values – 1 and 0, which are known as binary digits or *bits*. Thus, an 8-bit binary number consists of 8 consecutive digits each of which are either

1 or 0, for example %1100 0101. Binary digits are usually referred to as n-bit numbers depending on how many bits they have. For example, a binary number that consists of 8 bits would be referred to as an 8-bit number.

Although perhaps not as intuitive or easy to grasp, in essence using binary numbers differs little from the traditional way of presenting them in decimal (or base-10) system, which operates with numerical values within the range [0, 9]. For example, representing the figure 237 in decimal system can also be written as a simple equation:

$$2 \times 10^2 + 3 \times 10^1 + 7 \times 10^0$$

Here each digit is multiplied by a power of ten and then added together to make up the final number. Binary numbers use the same logic – for example, a binary number such as %1011 can be expressed with an equation

$$1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$$

which in decimal would translate to

$$8 + 2 + 1 = 11$$

(Zaks, R., 1983, p. 12)

Additions work in a similar way to decimal numbers as well, where the calculation is performed bit by bit starting from the least significant bits (LSB, or the rightmost bit), and with a carryover of 1 to the next bit to the left each time adding the bits together would result in a value larger than 1. For example,

$$\begin{aligned} & \%0011\ 1101\ (61) \\ & + \%0000\ 0011\ (3) \\ & = \%0100\ 0000\ (64) \end{aligned}$$

Subtractions also work by using the same logic as with decimal numbers, although usually it is more convenient to do an addition with a *negative* number, which are covered in section 2.5. (Zaks, R., 1983, p. 14)

Although not necessary in real life, in computer science binary numbers are represented with a fixed number of bits that is usually a power of two, for example 4, 8, 16, 32 or 64 bits, even if the number itself would not need as many bits. Although this approach might seem counterintuitive, it is necessary when taken into account the way computers store data (in one or more 8-bit

bytes) or how negative numbers are represented. If an arithmetic operation results in a value that is larger than the number of bits it contains would allow, the overflowing bits are culled. For example, consider the following addition of two 8-bit values, where the result is stored in a third 8-bit value:

$$\begin{array}{r} \%1111\ 1111 \\ +\%0000\ 0001 \\ =\%(1)\ 0000\ 0000 \end{array}$$

As the result in this case can only store 8-bit values, the ninth bit is automatically discarded leaving a value of `%0000 0000`. In the case of a processor, this bit is stored in the carry bit of its status register. (Zaks, R., 1983, p. 20)

In the context of the 6502, only the first three bit sizes are relevant, as the processor can handle only 8-bits worth of data (known as *bytes*) at a time, and its address bus can only handle 16-bit wide addresses (known as *words*). 4-bit numbers (known as *nibbles*) are needed when running the processor in the decimal mode (see section 3.6.4) or when manually converting binary to hexadecimal numbers. It is also quite common to divide binary numbers into nibbles thus reducing errors and making them easier to read. For example, a 16-bit number `%1111101111111111` is a lot harder for a human eye to process, whereas dividing it into nibbles produces a significantly more comprehensible result: `%1111 1011 1111 1111`.

Understanding binary numbers is essential when operating on a level as low as the processor, as the values of a bit have a direct relation to an electric charge – on or off, or 1 or 0. As an electronic component itself, the operation of a processor is manipulated by changing either the state of the electric charge found in its parts or the route an electric current takes in its connection lines. Binary and hexadecimal numbers are used as a human friendly representation of these and feeding them to the processor changes the behaviour of it or the device that makes use of the processor in different ways. This is known as programming the processor and will be discussed more in depth later. (Zaks, R., 1983, p. 10)

2.3 Hexadecimal numbers

While processors can only think in terms of bits, representing larger values in binary results in longer string of digits. As was demonstrated in section 2.2, this increases the likelihood of an error and makes them more difficult read. A common, and a more compact way of representing numbers in computer science is known as *hexadecimal*. Hexadecimal (colloquially known as hex) is a base-16 system that extends the range of numbers to alphabets. Numbers in range [0, 9] are represented with numbers, and the numbers in range [10, 15] are represented with the letters [A, F] in their alphabetical order. I.e., 10 is represented with 'A', 11 with 'B' and so on to 15, which is represented with 'F'. With this numeral system 16-bit numbers can be represented with only four digits in the range [\$0, \$FFFF] instead of 16 needed by the binary system, or five needed by the decimal system. Although the letters in hexadecimal numbers will be capitalized in this thesis, it is by no means necessary, as the casing of the letters is irrelevant. (Lampton, C., 1985, p. 13)

Also, as both binary and hexadecimal have a base number that is a power of two, converting between these two systems is easier than converting between binary or hexadecimal and decimal numbers. Each hexadecimal digit [\$0, \$F] represents a four bit binary value, so to convert from binary to hexadecimal, one must divide the binary value to four bit sections starting from the least significant, or the rightmost bit of the binary value. A conversion to decimal may be occasionally needed, but as the range is only [0, 15], this is only a minor task once one has become more acquainted with binary and hexadecimal numbers. (Lampton, C., 1985, p. 14)

2.4 Bitwise operations and Boolean logic

Boolean logic is a form of algebra that deals with the truth values of statements, meaning that it regards every statement to be true or false, or more commonly 1 and 0. There are only three basic logical operations, AND, OR and NOT, and a further four secondary operations, of which only Exclusive OR (EOR) is significant in terms of processor technology and programming. Of these

operations AND results in a value of 1 if and only if all the statements in the equation are 1, OR results in 1 if at least one of the statements is 1, and EOR results in 1 if and only if *only one* of the statements is 1. NOT negates the value of a statement, meaning that a negated statement having a value of 1 becomes a 0 and vice versa. A common practice to determine the result of a Boolean equation is to use a truth table, where each combination and its result is laid out. So, assuming statements A and B have a truth value, the operations above can be expressed in the way outlined in Figure 1. (Wikipedia, 2023b)

A	B	A AND B
1	1	1
1	0	0
0	1	0
0	0	0

A	B	A OR B
1	1	1
1	0	1
0	1	1
0	0	0

A	B	A EOR B
1	1	0
1	0	1
0	1	1
0	0	0

A	NOT A
1	0
0	1

Figure 1. Truth tables for logical AND, OR, Exclusive OR and NOT

Bitwise operations expand this concept a bit further. The principle is the same, but instead of performing the operation on statements, it is performed on numerical values, and each bit is considered its own statement. For example, if A would be 10, and B would be 6, their respective binary values would be %1010 and %0110. Now, if performing a bitwise AND between A and B, the operation is performed between each *bit* starting from the rightmost or the least significant bit (LSB) of both values, therefore resulting in %0010 (or 2 in decimal) as the following truth table demonstrates:

A	B	A AND B
0	0	0
1	1	1
0	1	0
1	0	0

Figure 2. A truth table of bitwise AND between A and B

Although there is no specific instruction for the bitwise negation in the 6502 instruction set, one can easily be done by performing a bitwise EOR between a value, and a value that has all its bits set, for example:

A	\$F	A EOR \$F
0	1	1
1	1	0
0	1	1
1	1	0

Figure 3. A truth table of bitwise EOR between A and \$F

In addition to bitwise logical operations, there are four other operations that alter the bits of a given value, known as shifts and rotations. Shifts move the bits of the value either to the left or to the right, essentially making it either a multiplication or division operation. A bitwise shift to the left moves all the bits to the left while inserting a 0 to the least significant bit of the value and moving the old most significant bit (MSB, the leftmost bit) to the carry bit in the status register. Conversely, a bitwise shift to the right moves the old LSB to the carry bit in the status register and inserts a 0 to the MSB. For example, if operating on an 8-bit value of %0001 0101 (which is 21 in decimal) and carry bit in the status register being set (i.e., its value is 1), performing a shift to the left will result in %0010 1010 (42 in decimal), effectively multiplying the value by two and setting the carry bit to 0. Performing a shift to the right results in %0000 1010 (10 in decimal) and a carry bit being set to 1 in the status register, which is equal to an integer division by 2. Bitwise rotations work in a similar manner – the bits are shifted and the old LSB or MSB is shifted into the carry bit as before, depending on the shift direction. However, the new bit that gets inserted into the LSB or MSB now comes from the carry bit of the status register instead of always being a zero. For example, consider previous example of shift to the right, in which the resulting value was %0000 1010 and the carry bit being 1. If this value would now be rotated to the right, the result will be %1000 0101, and carry bit being 0. Rotating the value to the left instead of right results in %00010101 which is the original value, but the carry bit is now set to 0, which was the MSB before the rotation. (Wikipedia, 2023c)

Using bitwise operations is a convenient way to quickly multiply or divide numbers with powers of two, or to mask out or set or clear certain bits of a value. Considering the limits the processor imposes on the user, it may sometimes be more prudent to use a single variable to hold multiple values, providing their combined number of bits fit into an 8-bit variable. For example, assuming the program needs to track eight on/off states within its execution, only a single 8-bit variable is needed, as each bit can be set to track whether a certain state is set or not, much like the status register. Bit n can be turned on by performing a bitwise OR with the variable and a value of 2^n , its state can be checked by performing a bitwise AND with the value of 2^n , and it can be cleared by performing a bitwise AND with the negation of the value 2^n . Although packing data this way is possible, it is also possible that the extraction process is more trouble than it's worth, sometimes even completely negating the advantages gained from the packing. As such, it should be approached with reasonable caution.

2.5 Representing negative numbers

Representing negative numbers in binary is not as straightforward as with decimal numbers, as one cannot just add a negative sign in front of it. For this reason the most significant bit (MSB, or the rightmost bit) is used as a signed bit, and the negative number is formed with a technique called *two's complement*, which rearranges the range the binary number represents from [0,255] to [-127,128]. The two's complement is formed by reversing or negating the values of each bit in the absolute value of the number and adding a 1. For example, the value of -6 would be converted in the following way:

$$\%0000\ 0110 \rightarrow \%1111\ 1001 + 1 \rightarrow 1111\ 1010$$

To get the absolute value of a negative number, the same process can be used:

$$\%1111\ 1010 \rightarrow \%0000\ 0101 + 1 \rightarrow \%0000\ 0110$$

The necessity of using a fixed number of bits to represent negative numbers should now be clear – if the numbers used only as much bits as was necessary, it would be impossible to know *when* the number in question would

be negative. Like previously mentioned, a common way of performing subtractions on binary numbers is to perform an addition with a negative number. This makes sense since $a - b = a + (-b)$. For example:

$$\begin{aligned} & \%0000\ 1010\ (10) \\ + & \%1111\ 1010\ (-6) \\ = & \%1\ 0000\ 0100\ (4) \end{aligned}$$

As the two operands of the equation are 8-bit numbers, the result is the same size, and since the most significant bit is the ninth bit, it is automatically ignored. (Thakur, A., 2019)

2.6 Arithmetic operations

The 6502 surprisingly has only two *actual* arithmetic operations – addition and subtraction. While bitwise shifts do perform multiplication and division operations, they work only on powers of two and even then can be performed only one bit at a time, and as such are too limited to be considered as true arithmetic operations. The lack of multiplication and division operations means that these have to be manually implemented. Fortunately, multiplication is easily done with an addition operation, as

$$n \times x = x_1 + x_2 + x_3 + \dots + x_n$$

Division operation is a bit more complicated, but can also be done with the help of subtraction and addition, as the following pseudocode algorithm demonstrates:

```
Quotient=0
Remainder=0
While Dividend > Divisor:
    Dividend-Divisor
    Quotient+1
Remainder=Dividend
```

Although more efficient algorithms exist, even these basic methods show that to solve more complicated operations all one really needs is addition and subtraction. The rest is just a question of implementing proper algorithms to achieve the result. Trigonometric functions, for example, are just division

operations, and several algorithms for square root exist that utilize all four basic arithmetic operations of addition, subtraction, multiplication and division, one of which is the Newton-Raphson method (Regmi, S., 2020).

3 THE STRUCTURE AND FUNCTIONALITY OF MOS 6502

The sole purpose of the processor is, as the name implies, to process the data within the device – i.e., it transfers the data from one place to another via load and store instructions and manipulates them by performing arithmetic or logical operations on them. What the data is used for, however, is of no concern to the processor – that task is left for the auxiliary interface chips that handle the audio, video and I/O interfaces, for example. As covering the functionality of these various auxiliary chips would expand the scope too much, this thesis will only concentrate on the processor.

The processor itself is housed in a 40-pin Dual In-line Package (DIP), which is a rectangular shaped plastic casing and has 20 connection pins on both sides. As the electronic aspect of the operation of the processor is beyond the scope of this thesis, only some of these pins and their purpose will be covered in their separate sections.

6502 PINOUTS

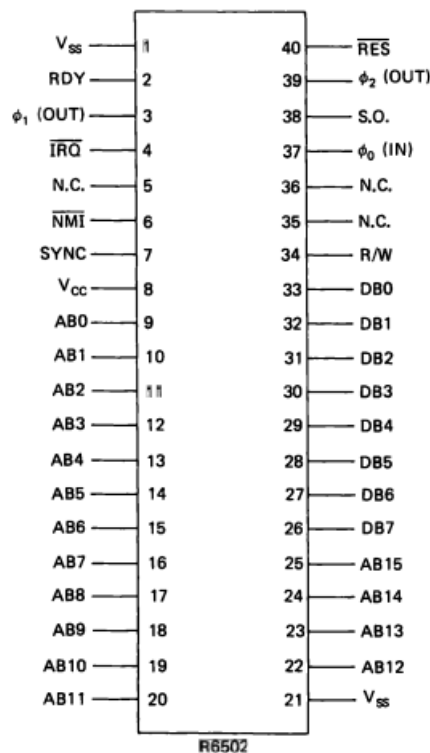


Figure 4. The pin configuration of the 6502 (Carr, J., 1984, p. 30)

3.1 Processor clock

Although not usually a part of the processor per se, the clock generator is arguably the most important part of it. A clock signal created by the generator (also known as crystal) is needed to synchronize the operation of a processor or a circuit (Wikipedia, 2022c).

The generator oscillates with a fixed rate, which in the case of the 6502 can be from 1MHz to 4MHz depending on the version of the chip – for example, a 1MHz clock rate means that the generator produces one million clock cycles per second. One clock cycle is defined as the time between the pulses from the clock generator. It consists of four phases – low phase, rising edge, high phase, and descending edge, where the low and high phase refer to the voltage level of clock signal, and the rising and descending edges refer to the transition from one phase to another. During each clock cycle the processor either fetches a new instruction or executes a part of the instruction currently in execution. Some sources say that older processor can only execute one instruction per clock cycle, but this is either incorrect or bad wording. While it is true that older processors can only perform execution of one instruction per cycle, the cycles needed to complete the execution often take more cycles than just one. E.g., the amount of clock cycles needed by the instruction can vary from 2 to 7 on the 6502. (Computer Hope, 2017) (Doe, J., n.d.) (Wikipedia, 2023a) (Synertek, 1976a, p. C-2)

The pin 37 on figure 4 (Φ_0) is the input pin for the clock signal coming from the generator. The 6502 generates two clock outputs (Φ_1 on pin 3 and Φ_2 on pin 39), in such a way that Φ_1 is 180° out of phase with the other two. This means that Φ_1 is high when both Φ_0 and Φ_2 are low, and vice versa. This two phased clock is used for both internal timing of the instruction execution and telling external devices when they can read or write to the buses. (Lateblt, 2012) (Synertek, n.d., p. 21)

3.2 Buses

The term 'bus' in computer science refers to the way the data is transferred within the device – in effect they can be thought of as transport lines that determine what data is being handled, and where it is loaded from or stored to. The 6502 has two buses – the data bus and the address bus.

The data bus consists of pins [26,33] of the processor (see figure 4). As each line represents one bit, a total of 8 pins means the data bus can only handle 8-bit numbers at a time, making the 6502 an 8-bit processor. The lines of the data bus hold the data that is to be handled either as an instruction, operand in an operation or to be stored in the memory. Whenever data comes from or goes into the memory or an external device it gets stored in the data bus before the execution phase of the instruction. When a new instruction cycle starts, the first thing to be loaded into the data bus is the numerical value of the instruction to be executed. (Synertek, 1976a, p. 3)

The address bus is made up of the pins [9,25] of the processor. The address bus is 16-bits or lines wide and can handle 65,536 different memory addresses. The address bus determines where the data is read from or written to, including the address of the current instruction in the same way as the data bus holds the numerical value of the instruction. (Synertek, 1976a, p. 51)

In close conjunction with the buses is the Read/Write (R/W) pin of the processor. This determines which way the data flows between the processor and the memory – when the voltage of the pin is high (i.e., 5 volts), the processor reads the data from the address defined by the address bus, and stores it into the data bus. When the voltage is low (0 volts), the data in the data bus is written to the address given by the address bus. (Synertek, n.d., p. 6)

3.3 Registers

The 6502 has six different registers, of which five are 8-bit sized and one is 16-bit. Three of these registers can be used to load and store data, and only one of them can perform operations beyond incrementing or decrementing its value by one. Each register can be thought of as a small amount of memory within the processor, as each can store data according to their size. As registers have a more direct connection to processor, accessing them is faster than accessing memory, and for this reason operations that involve handling data almost always have at least one register as either the source or the target of the operation, only exceptions being the instructions that read, modify, and write using the same location in memory (ASL, DEC, INC, LSR, ROL and ROR). Operations between two different memory locations are not supported. (Wikipedia, 2023a) (Wikipedia, 2022b)

3.4 The accumulator

The accumulator, or A register, is one of the five 8-bit registers of the 6502, and the only register than can perform arithmetic or logical operations. As such it is the target or source register of most of the instructions in the 6502 instruction set and can be considered as the main register of the processor. (Synertek, 1976a p. 4)

3.5 Index registers

X and Y are 8-bit registers that are used mainly as index registers, i.e., their contents are usually used to add an offset when fetching data from memory, or as a counter when performing a loop. Data can be loaded, stored, compared or transferred in the same way as with the A register, but otherwise their value can only be incremented or decremented by one. (Birnbaum, I., 1984, p. 59)

3.6 Status register

Status register stores the state of the processor by using 7 of its 8-bits as status flags, each indicating a different thing. Some of these flags are used as conditions on whether to take a branch in the program code, and some can be set or cleared by the programmer, while some are only affected by the processor. The 5th bit of the status register is unused (Synertek, 1976a, p. 24) (Figure 5)

Bit	b ₇	b ₆	b ₅	b ₄	b ₃	b ₂	b ₁	b ₀
Flag	N	V	U	B	D	I	Z	C

Figure 5. An illustration of the status register

3.6.1 Carry flag

The first flag is the Carry flag (or C) in bit 0. This indicates whether an addition resulted in a wrap over from \$FF to \$00, meaning the result was higher than 255, and carry needs to be added if performing operations on 16-bit numbers. It is also used in subtraction as a borrow bit, when its negation is deducted from A. Carry flag can be set by the programmer with SEC (SEt Carry), cleared with CLC (CLear Carry), and used for branching with BCC (Branch if Carry Clear) and BCS (Branch if Carry Set). (Synertek, 1976a, pp. 24-25)

3.6.2 Zero flag

The Zero flag (Z in bit 1) indicates whether the operation results in or data in data bus is zero. This is one of the flags that only the processor can affect, but it can be used for branching with BEQ (Branch if EQual to zero) and BNE (Branch if Not Equal to zero). Some sources state that the flag is set if the contents of the A register is set to zero by previous operation, but this is false. For example, comparison operations are done by performing a subtraction with between a register (A, X or Y) and memory without storing the results, and this sets or clears the flag accordingly. Furthermore, not *all* instructions affect the Z flag (although most of them do), which means that the branch operations

don't care about *how* or *when* the flag was set or cleared, they just check its state. For example

```
LDA #$00
PHP
STA $1234
BEQ branch
```

loads a value of zero to A register, setting the Z in the process. It then stores this value to memory location \$1234 and pushes the current contents of status register to stack. Neither of these operations affect *any* of the flags in the status register. Finally, BEQ checks the zero flag, and as it is still set, moves the execution of the program to location in memory that is labelled "branch".

(Synertek, 1976a, p. 25) (Wong, C., n.d.) (Zaks, R., 1983, p. 164) (Zaks, R., 1983, p. 178)

3.6.3 Interrupt flag

The Interrupt flag (I in bit 2) indicates whether a hardware or a software interrupt can take place. As the hardware interrupt request (or IRQ) is *level sensitive*, the interrupt flag, if set, blocks the incoming interrupt requests to avoid overlapping interrupts. This does not, however, block the non-maskable interrupts (or NMIs), which are *edge sensitive* and detect the changed state of the NMI pin of the processor. Interrupts will be covered more extensively in section 3.9. (Synertek, 1976a, pp. 25-26)

3.6.4 Decimal Flag

The Decimal flag (D in bit 3) indicates whether decimal mode is on. Decimal in this context refers to the numbering system (i.e., base 10), rather than to floating point numbers. When this mode is on, addition and subtraction instructions consider the data operand (or the data in A) to be in the range of 00-99. In this mode the number is treated on digit-by-digit basis, and the byte is divided into nibbles. While each of the bytes hold 4 bits and thus can represent a number in the range of [0, 15], only the range [0, 9] is acknowledged. This means that to get the right digit, 6 must be added to the

corresponding nibble which will then carry over to the next nibble or to carry flag, depending on which nibble the operation was done to. For example, if the digit is \$F, or 15 in decimal, its binary representation is %1111. Now, when 6 is added to it, the binary equation becomes

$$\%1111 + \%0110 = \%0001\ 1010$$

which is \$15 in hex and the correct result when using the decimal mode. Decimal mode is only used by ADC (ADd with Carry) and SBC (SuBtract with Carry) and can be set by SED (SEt Decimal flag) and cleared by CLD (CLear Decimal flag). section 3.9. (Synertek, 1976a, pp. 26-27) (Zaks, R., 1983, p. 64)

3.6.5 Break Flag

The Break flag (B in bit 4) indicates that a software interrupt has taken place. It has no other function and cannot be set or cleared. RTI (ReTurn from Interrupt) and PLP (Pull status register) ignore this flag when pulling the stored status register from the stack. No branch operation makes use of it, but it can be used to distinguish between software and hardware interrupts. (Synertek, 1976a, p. 27) (Synertek, 1976a, pp. B-22-B-23)

3.6.6 Overflow Flag

The V (oVerflow) flag (bit 6) indicates whether an overflow happened during an operation. This can only happen when adding or subtracting data to A and happens when the sign bit of the result is different than expected, like in the cases of adding two positive or negative values together, and subtracting a positive value from negative or vice versa (Zaks, R., 1983, p. 22). E.g., adding \$2F (47) to \$64 (100) would result in \$93 which is %1010 1110 in binary. As the most significant bit is set, the processor considers it a *negative* value instead of a positive, i.e., the value is -109 instead of 147 in decimal. Like the decimal flag, overflow flag is only used by ADC and SBC, and while it can be cleared, it cannot be set by the programmer. It can be used for branching with BVS (Branch on oVerflow Set) or BVC (Branch on oVerflow Cleared). . (Synertek, 1976a, pp. 27-29)

3.6.7 Negative flag

The N (Negative) flag (bit 7) indicates whether the data or result of the last operation was negative, i.e., its most significant bit is set, making it \$80 or larger. Like the break and zero flags, the programmer has no control over its state. BMI (Branch on Minus) takes a branch if it is set, BPL (Branch if Plus) branches if it is clear. (Synertek, 1976a, p. 29)

3.7 Program counter

Program counter (henceforth PC) is the only true 16-bit register in 6502. It holds the current execution location of the program, from which the instructions or their operands are fetched, and it is automatically incremented according to the number of operands (one or two bytes) for each instruction. The PC can be re-adjusted by branching operations, interrupts (both software and hardware) or by jumping to a new memory location or subroutine, from which it will eventually return to the location the PC held before making the subroutine jump. This latter also applies to interrupts. (Birnbaum, I., 1984, pp. 7-8) (Synertek, 1976b, p. 31) (Zaks, R., 1983, pp. 43-45)

3.8 Stack pointer

Stack pointer (henceforth SP) is similar to the PC in the way that it holds a location in memory. However, only the lower 8 bits (bits 0-7, or low byte) of the address are in use, and bits 8-15 (or high byte) are hardcoded to hold a value of \$1. Thus, the SP is an 8-bit register instead of 16-bit. Stack itself will be covered in section 3.10.4. (Zaks, R., 1983, pp. 48-49)

3.9 Interrupts

Interrupts are a way for the peripheral devices, such as storage media, printers, and keyboards to inform the processor that something has happened that requires its attention. As constantly checking for the status of these

devices would consume the resources of the processor needlessly, it makes sense to do that only when prompted to, and this is usually achieved via an additional I/O interface controller. There are three different hardware interrupt signals available for the 6502 – reset, maskable interrupt and non-maskable interrupt, each with their own dedicated signal line on the processor – as well as a software interrupt signal, which is usually used for debugging purposes when writing programs in assembly language. Each of the hardware signals are active low signals, meaning that they are considered to be active only when the voltage of the pin is low. (Wikipedia, 2022a) (Synertek, 1976b, p. 16)

Interrupts quite literally interrupt the current operation of the processor and transfer the control to a specified interrupt service routine (ISR), after the instruction currently in execution has finished. The address of these ISRs are stored in six different memory locations at the end of the addressable memory range, known as *vectors* – vector in this context referring to an entry point rather than the mathematical concept – and each vector uses two 8-bit locations to store a 16-bit address of the corresponding ISR. Locations \$FFFA and \$FFFB are reserved for the non-maskable interrupt requests (NMI), the reset vector is located in \$FFFC and \$FFFD and \$FFFE and \$FFFF, the last two bytes of the addressable range are both for the maskable interrupt requests (IRQ) and software interrupts (BRK). (Wilson, G., 2014)

IRQs (pin 4 in figure 4) differ from the NMIs in various ways; firstly, they are *level sensitive*. This means that every time the voltage level of IRQ pin is low, i.e., 0, the interrupt is triggered. Being level sensitive also means that interrupts would be constantly triggered whenever the voltage is low, filling up the stack. To prevent this IRQs also make use of the interrupt bit in the status register to block further interrupts – an IRQ can only be triggered when both the voltage level and the interrupt bit are 0. If the bit is 1, incoming IRQs are ignored. NMIs (pin 6 in figure 4), on the other hand, are *edge sensitive*. This means that an NMI is triggered every time the voltage level of the NMI pin changes from high, i.e., 5 volts, to low. Unlike IRQs, they don't use the interrupt bit in status register to mask the incoming interrupts, which means that they are able to interrupt a routine for an IRQ. NMIs also have a higher priority than IRQ, so if both happen

to trigger at the same time, the service routine for NMI is executed. It is usually used to indicate a situation where something drastic has happened with the hardware, and immediate attention is required

(Wilson, G., 2014) (Wikipedia, 2022e)

Reset (pin 40 in figure 4) and software interrupts are, in a way, non-maskable interrupts themselves due to their respective use cases – a software interrupt is intentionally placed within the program code to a specific to debug the program, and a system reset is usually performed intentionally, too. Any guesswork on whether or not they are triggered or not would defeat their purpose.

When an interrupt occurs, the current address of the PC is stored into the stack along with the status register. When the execution returns from the ISR after a dedicated instruction has been called, these values are retrieved from the stack in reverse order. As the programmer or the user cannot predict when an interrupt happens, the contents of A and the index registers *must* be stored to stack in the beginning of the ISR and retrieved before returning to the main program, if the ISR uses them. To prevent unwanted interrupts *during* the IRS, the interrupt bit of the status register is set automatically during the initial interrupt sequence after the status register has been pushed into the stack. Retrieving it when returning to the main program clears it automatically, although this does not apply to the reset sequence. While the reset signal initially triggers a similar sequence as any of the other interrupts, the machine code routine in the reset vector points to uses a jump instruction to relocate the PC, as the return address in the stack would most likely point to an incorrect location. This means that the interrupt bit must be cleared at the end of the reset routine. (Wilson, G., 2014) (Figure 6)

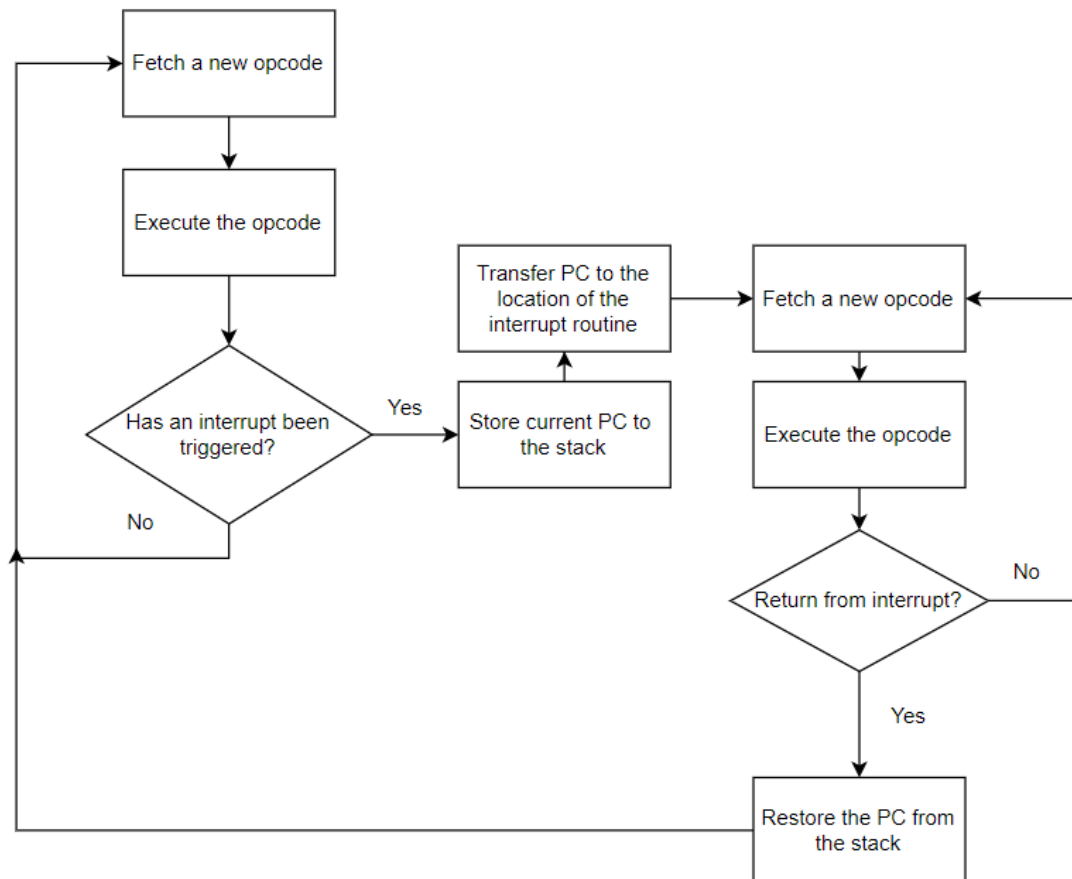


Figure 6. A flow chart illustration of the interrupt process

As interrupt and reset routines and especially their addresses are often vital in regard to the operation of the device, the default values and routines are usually stored in ROM so they are available even after a reboot of the device. However, as using only hardcoded addresses and routines would impose unnecessary limitations for developers, these can often be rerouted and rewritten to accommodate for the needs of the programmer. (Wilson, G., 2014)

3.10 Memory and memory handling

Although the memory itself strictly speaking is *not* part of the processor, it is an essential part of any functioning device making use of a processor of any kind. Memory is the main storage unit of a computer or other device. It will hold the program code and the data it needs during run time, and in most cases, data needed for the device's functionality, like I/O-routines.

Memory can be thought of as a set of numbered boxes that are all the same size and can hold only one thing at a time. Even though the memory in its lowest level consists only of memory cells made of transistors that each represent one bit, it makes more sense from the user's point of view to handle more than one bit at a time. Therefore the 6502, as well as modern devices, support accessing data in 8-bit chunks, each with its own location (Wikipedia, 2023d). A byte can hold 256 different values ranging from 0 to 255 (or [-128, 127]) making it large enough to be able to store ASCII-characters and represent numbers, and as from the device's point of memory is just a very long line of 1s or 0s, larger numbers or character sets can be stored by using multiple consecutive bytes.

The memory is organized in 256-byte sized sections known as *pages*, and the maximum number of pages for 16-bit address bus is 256. This means that in total the 6502 has 256*256 bytes (or "boxes", if you will) where data can be stored and which can be accessed by knowing its number or, rather, its *address*. The contents of a memory page are often illustrated by using a 16x16 matrix, as it allows the reader to see the whole layout of the page at once. (Figure 7)

Address	BYTE ₀	BYTE ₁	BYTE ₂	BYTE ₃	BYTE ₄	BYTE ₅	BYTE ₆	BYTE ₇	BYTE ₈	BYTE ₉	BYTE _A	BYTE _B	BYTE _C	BYTE _D	BYTE _E	BYTE _F
\$0000	B2	CB	37	E2	46	BE	B6	CD	8E	8D	B7	69	94	DF	64	4F
\$0010	2B	2D	AA	D3	2A	A6	6D	E0	44	96	8B	6B	C6	7D	A5	AC
\$0020	FE	BE	65	4D	71	AD	C8	F4	94	15	80	47	26	7B	50	D0
\$0030	8A	AA	A2	DC	BC	D3	C6	79	1C	F8	C2	18	29	6C	2C	B7
\$0040	D1	5B	27	5A	EA	A7	70	D3	BE	4B	97	4C	E1	B0	F4	75
\$0050	1E	13	3B	29	2B	9E	BA	6B	6B	4C	1B	81	FC	42	D8	1F
\$0060	ED	CF	5A	65	FB	53	67	23	37	F0	93	7E	CD	2D	BE	FB
\$0070	62	8F	26	1C	7A	A8	51	6E	FE	F8	74	C1	25	79	99	3C
\$0080	3C	95	1C	B2	1D	9E	B6	C3	D9	74	74	24	9B	23	7D	82
\$0090	FB	17	A1	70	39	CF	EA	24	2C	74	AF	F9	80	5B	72	58
\$00A0	6B	DC	77	EA	A6	C0	24	D7	92	9F	45	E3	3A	DF	7D	AE
\$00B0	9B	45	60	B4	57	53	13	43	D2	AF	58	E2	B7	5C	6E	5E
\$00C0	EA	36	33	34	B2	63	FF	E2	FE	3C	CF	5B	3A	B2	12	65
\$00D0	B6	63	2A	7F	E2	BB	42	A2	23	C1	88	3B	13	B7	DA	9A
\$00E0	25	88	69	2E	C6	52	AF	48	B5	7F	1D	E1	A6	AA	80	F0
\$00F0	6A	48	F1	7A	AA	4B	D5	87	94	B8	DD	11	F6	23	CA	82

Figure 7. An illustration of the memory layout of zero page

All the data in memory can be at least read and used, but not necessarily altered. Sometimes a term *memory-mapping* is used in conjunction with memory and addresses. This refers to a certain address or a range of addresses being used by an I/O-device or an auxiliary chip, meaning for

example that the data written to certain addresses is used by a video chip to determine the palettes for drawing graphics, or that a certain address is used by the sound chip to write the oscillator values to. (Wilson, G. 2022) (Nesdev, 2023a) (C64-Wiki, 2021)

Memory organization is significant, as accessing the range [\$0000, \$00FF] (the first 256 byte page known as the *zero page*) is usually faster and instructions take one byte less than when accessing any other location in memory. Additionally, in some cases executing an instruction may result in an extra clock cycle when crossing the page boundary (e.g., when a branch location is on the different page than the opcode following the branch operation). As the amount of clock cycles determines the execution time and speed of the program, it pays to be aware where extra cycles may occur or how to reduce them. For this reason, zero page is commonly used as a storage for often used variables instead of program code, although this is not always possible. Apple II, for example, reserves most of the zero page for system programs leaving only 24 bytes of guaranteed free and safe space. (Birnbaum, I., 1984, p. 14) (Zaks, R., 1983, p. 49) (Espinosa, C., 1979, pp. 74-75)

3.10.1 Types of memory

There are two types of memory – RAM and ROM. Random-access Memory, or RAM, is writable memory where the program is loaded and executed from, and where data that needs to be accessed during run time is often stored, especially if speedy access is required. Data in RAM exists only as long as power is supplied to the memory cells by the device, or by an additional battery. Some game cartridges on Nintendo Entertainment System (NES) utilize a battery to save game data on the cartridge, for example.

The name Random-access Memory refers to the way data can be accessed. Each byte in memory can be accessed with little or no difference in time regardless of its location, and there are no limitations in which order the data can be stored or read (Wikipedia, 2023). In contrast, accessing data on a

physical storage media such as a cassette tape (which was a common way to store data and programs in the 1980s) can only be done *sequentially*, i.e., the data must be read and stored in an ordered sequence, requiring feeding the tape through the drive's tape head (Wikipedia, 2022f). On the other hand, floppy disks allowed a certain amount of random access, although moving the physical read/write head and spinning the magnetic disk containing the data to a correct position meant that while faster than tapes, it was slower than accessing memory on device (Smith et al, 1994, p. 4/21).

Read-Only Memory, or ROM is memory that can only be read. ROM is similar to RAM in the sense that it allows random access, but unlike RAM, ROM will retain the data written in it even without a power supply. This is achieved by "burning" the data on the memory chip during its fabrication (Smith et al, (1994), p. 3/5) using a process known as photolithography, where the desired pattern is created by masking out unwanted parts of the chip and exposing only the needed parts to UV-light (Wikipedia, 2022g). A ROM chip created this way is called *masked ROM* and while cheaper to produce in bulk than reprogrammable ROMs (PROMs, EPROMs, EEPROMs), it means that the data it contains must be absolutely perfect before entering production phase, as it cannot be altered afterwards (Wikipedia, 2023f).

ROM is usually used to store data necessary for the device's intended usage, such as the boot sequence or I/O and file management functions of a device, or to store program or game files in cartridges. Depending on the device, ROM may be an integral part of the device's memory map, or it may be turned off to expose the RAM underneath. For example, on NES the address range of located in \$4020-\$FFFF is always reserved for the ROM cartridges (with the option of having 2 kB of battery backed save or work RAM in range \$6000-\$7FFF), while on the C-64 the ROM is an additional memory that is mapped on top of certain parts of the 65,536 bytes of RAM it has. If needed, these parts can be remapped to RAM by turning off one or all the three bits of the byte stored in address \$1. (Forster, J., n.d.) (Nesdev, 2023b)

3.10.2 Addressable memory

As previously mentioned in section 3.2, the 6502 has a 16-bit wide address bus. This means that it can address a total of 65536 bytes, with an address range of 0-65535 (\$0000-\$FFFF in hexadecimal). However, this does not mean that the device has or is limited to that amount of physical memory. Nintendo Entertainment System (NES), for example, only has 2 kB of internal RAM (located in \$0000-\$07FF), and additional 20 bytes reserved for I/O, audio, and video registers. As there is slightly less than 50 kB reserved for the cartridges that contain the games, around 14 kB of unaccounted for memory remains. This is a result something called *memory mirroring*, which occurs when one or more address line is ignored by the chip. In practice this means that the same data has multiple addresses it can be accessed at. For example, the 2 kB RAM located in \$0000-\$07FF can also be accessed at ranges \$0800-\$0FFF, \$1000-\$17FF and \$1800-\$1FFFF without copying it to those locations (Nesdev, 2022).

On the other hand, there is a possibility that the device has *more* memory than it can access via the address bus. When additional RAM (or ROM) chips are connected to the device, a technique called *bank switching* is used. This allows the user to decide which memory bank is used by the processor either by using a physical switch or setting up control bits in a specific register on the device. When in use, bank switching logic maps parts of the memory to different banks as per user's instructions, i.e., it assigns the addresses it can access to different memory chips, while hiding the contents of the rest of the banks from the processor. If the device architecture supports DMA, data can be moved between the memory bank and the main memory of the device relatively quickly. However, as program execution can only happen *within* the addressable range, and possible programs thus need to be transferred to device's memory before execution, these extension units can only be used as storage devices. While their data can be accessed much faster than the data on a physical storage media like cassettes or diskettes, the fact that they store data on RAM and as such need to be powered at all times, either by the device

or a battery, makes them volatile. (C64-Wiki, 2023) (C64-wiki, 2022) (Wikipedia, 2022h)

The only ranges the 6502 actually *requires* are [\$0100, \$01FF] for the stack, and [\$FFFA, \$FFFF] for the interrupt and reset vectors. These last bytes of the addressable range contain the addresses of interrupt and reset routines and are hardcoded into the processor. While their contents (i.e., the address of their respective routines) may be changed, their location is fixed, as is the address range of the stack. Furthermore, as previously mentioned, the speedier access of zero page makes it a valuable addition, but as the stack can be accessed like any other part of the memory this would not strictly speaking be necessary. (Wilson, G., 2020)

3.10.3 Storing data

As already established, from the perspective of the programmer, data is organised to 8-bit sized chunks within the memory rather than as individual bits. However, this does not mean that only 8-bit numbers in the range of [0,255] can be stored into the memory. At its lowest level, and from the perspective of the processor, memory is still just a string of consecutive bits each of which is either on or off. This means that to store 16-bit numbers, multiple 8-bit numbers can be stored to consecutive bytes. In fact, apart from the size limitations of memory or storage media, an arbitrary sized number can be stored this way.

The 6502 does not store data in memory in a way that is intuitive to humans, i.e., the most significant byte or number first. Instead, the *least* significant byte is stored first, and most significant byte last. E.g., instead of storing \$1234 to consecutive memory locations as \$12 and \$34, the data is stored as \$34 and \$12. This way of storing data is known as *little-endian* and is still used by modern processors today. Storing data the other way around (most significant byte first) is known as *big-endian* and is used by some processors as well as the TCP/IP protocol.

Endianness is not something the end user often needs to worry about or even encounters, as it usually becomes apparent only when writing cross-platform programs that utilize the same data, writing programs in assembly, or when examining the contents of the memory via a memory monitor application or the binary file of a program with the help of a hex editor.

Those with programming background may be familiar with the concept of *typed* variables in some programming languages, where each variable has its own type. These come in multiple forms, such as integers and floating point numbers, both of which come in different sizes, or characters, character strings, and Boolean values. However, from the processor's point of view there exists nothing but 0s and 1s, which means that it does not care what type the data is intended to be. Typing is only a convention for the benefit of the programmers, who can use it to better manage the memory usage by choosing a suitable variable type for their needs. As even characters can have a numerical representation, this means that what the memory or a storage device *really* consists of is just a lot of numbers.

It is also important to note that while the end user of a computer or software may delete files or encounter a situation where the data is not found, in reality data never really disappears, it only changes its form. After all, essentially data is just a collection of electric charges within the device. This means that as an electric charge has only two states (off and on, or 0 and 1), even the total absence of electric charge would represent a value of 0. Thus, when a user deletes something from memory or a storage device, it does not follow that nothing can be read from that location. The value may be zero, completely random or, depending on the deletion method, in some cases the original value may still be recoverable.

3.10.4 Stack

Stack is a part of memory that is used to store data using the Last In First Out (LIFO) principle. Consider a pile of books, for example. When a book is put on

that pile, the first book taken out of the pile will be the one that was last put on top of it. Stack memory works in a very similar fashion – data is put (or pushed) into the stack and is taken out (or pulled) from it usually in reverse order. Unlike on modern processors or even its contemporaries, the address of the stack on the 6502 is fixed to a certain range – [\$0100, \$01FF] to be precise. Sub-routine jumps use the stack to store the return address, while interrupts store the return address as well as the status register. In addition, a programmer can store contents of A or the status register to stack at any time. As the intended use of the stack differs a bit from the other parts of memory, it is usually illustrated as a linear table instead of a matrix. While this brings out the growing nature of the stack, it also means that showing the whole stack would require a lot more space, which is why only the most relevant parts are usually shown. (Zaks, R., 1983, pp. 48-49) (Zaks, R., 1983, pp. 92-43) (Figure 8)

Address	Value	
\$01FF	5E	
\$01FE	1F	
\$01FD	D4	
...		
\$0145	3E	
\$0144	A8	← <i>Stack Pointer</i>
\$0143	39	
\$0142	19	
...		
\$0101	E1	
\$0100	9F	

Figure 8. An illustration of the stack memory

The SP points to the location on memory that is the target of the next stack operation, and it grows *backwards* – i.e., when data gets pushed to the stack, the SP is automatically decremented by one after the data has been stored. Conversely, when data is pulled from the stack, the SP is incremented automatically before the data is retrieved. The pull operation does *not* affect the data itself; it remains in the memory as is. Only the SP is affected, pointing to a new location. (Zaks, R., 1983, pp. 48-49) (Zaks, R., 1983, pp. 92-43) (Figure 9)

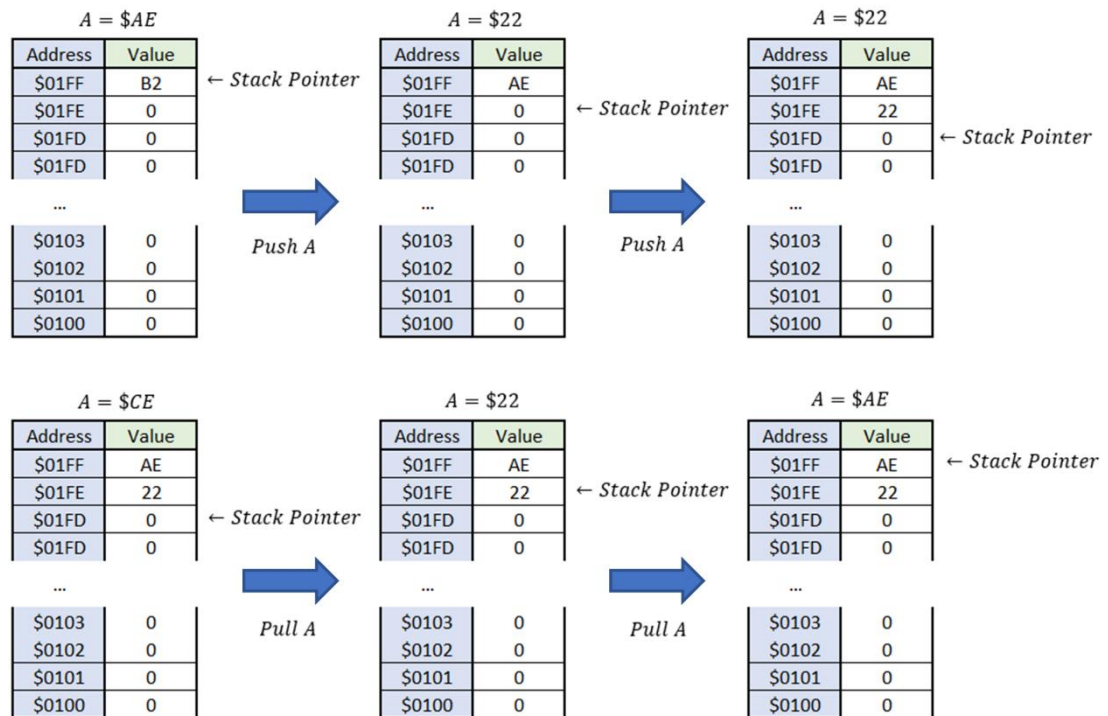


Figure 9. An illustration of the effects of push and pull operations on the stack

There seems to be no definitive explanation as to why exactly this backward growth approach was chosen, but one possible reason might be to protect the contents of the stack, as it quite often holds the return address stored by interrupts or sub-routines. If the SP started from $\$0100$ and grew towards $\$01FF$, there is the slight possibility of program overwriting the first bytes of page $\$01$.

Some sources say that the SP will be randomized after a reset (Tootill, A. & Barrow, D., 1984, p. 2), and according to Synertek's programming manual for 65XX family the start sequence is identical to interrupt sequence (Synertek, 1976a, p. 127) making three pushes to the stack. Thus, the SP does *not* start from the top of the stack after the reset, rather, the routine found in the address stored in the reset vector customarily sets it to $\$01FF$.

Although previously mentioned that the last thing that was put in the stack is usually the first to come out, this is not always the case. Stack does not differ from other memory locations and can be accessed by load and store

instruction in the same way. For example, if the SP is at \$01AA, one can fetch data to A register from \$01FF simply with

```
LDA $01FF
```

In addition, the location of the SP can be changed either by transferring the contents of X register to the SP with the instruction TXS. Adjusting the SP may also result in data corruption if the interrupts are not disabled and an interrupt occurs, or a jump to a sub-routine is made without re-adjusting the SP. Data in stack may also be lost if the SP *wraps around*. This happens when the SP is at \$0100, and data is pushed into the stack. As the high byte of the SP is always hardcoded to \$1, instead of the SP pointing to \$00FF it now points back to \$01FF after it is decremented. This is known as *stack overflow* (Wikipedia, 2022i). A similar case naturally occurs when the SP points to \$01FF and a pull is performed, although as the stack grows towards \$0100, this instance is less likely to replace critical data.

While stack can be accessed like any other parts of the memory, using its designated commands may have some benefits. For example, if the program needs to fetch several consecutive bytes using the absolute indexed addressing mode (where the data is fetched from a base address offset by an index register) and load them to A register, it will need 4 or 5 cycles to perform the load operation and additional two cycles to increment the index register, a total of 6 or 7 clock cycles and 4 bytes. However, storing that data to the stack and pulling them to A with PLA only takes 4 cycles and one byte without the need of an index register, which is a significant save.

4 PROGRAMMING THE MOS 6502

4.1 Programming languages

A programming language is how programmers communicate with their device, not unlike how humans communicate with each other. As with contemporary processors, there usually are several different languages a 6502 based device can be programmed with. These can be roughly divided in three categories – interpreted languages, compiled languages and assembly or machine language, with each category having a different abstraction level. Abstraction in terms of programming refers to the amount of required knowledge of the chosen hardware, and usually correlates with the readability of the code.

Although programming wise this thesis will concentrate on the machine and assembly language that was (and still is) usually the language of choice for the more seriously minded programmers on 6502, a few words should be said about each category to bring out their differences.

4.1.1 Interpreted languages

Interpreted languages are at the highest level of abstraction, with their instruction set often consisting mainly of instructions bearing a close resemblance to English language, for example. This makes the code much more readable and understandable even to a layman, but at the same they hide most of the functionality of the processor from the programmer. Consider the following example in BASIC (Beginner's All-purpose Symbolic Instruction Code):

```
10 PRINT "HELLO"  
Program 1.
```

This small program prints the word "HELLO". The function of the first line (or line 10 of the program code) is self-explanatory, and even without experience the function of the program can be easily deduced, due to its close resemblance of English.

The ease of use comes with a price: interpreted languages need a separate interpreter to function. Following the previously established analogy between programming and spoken languages, this resembles a situation where two people communicate to each other via an interpreter. To relay a sentence correctly, the interpreter must first hear it, if not in its entirety, then at least very close before translating it to the other person. The same holds true for interpreted programming languages – the interpreter must first read the instruction before translating it to machine language, the only language the processor knows. The interpretation is done during the execution of the program, slowing down the execution speed.

4.1.2 Compiled languages

Continuing with the analogy, compiled languages (like C or C++) can be thought of as a spoken language one *knows* but is not as fluent with as a native speaker. Unlike interpreted languages, compiled languages don't need an interpreter, but they *do* need a compiler. A compiler translates the program code into machine language *before* execution, and this needs to be done only once. The result is a binary file that can be loaded into memory and executed from there.

Compiled languages have lower abstraction level than interpreted language and enable writing more complex programs, but on the other hand their syntax is more complex and cryptic, as the following example shows:

```
#include <stdio.h>

int main()
{
    printf("HELLO\n");
    return 1;
}
```

Program 2.

Program 2 behaves in the same way as Program 1, but the number of instructions needed to achieve this makes it more obfuscated. However, as compiled languages often allow for more access to processor and memory,

and as the run time interpretation is not needed, programs are generally more efficient and faster than their interpreted counterparts.

4.1.3 Assembly and machine language

Although assembly and machine language are strictly speaking not the same thing, assembly bears a near 1:1 relation to machine language, and the terms are often used interchangeably. Such a close relation also means that programmer sometimes knows *exactly* the contents of the final binary file and determining the execution speed may be as simple as counting the clock cycles needed by each instruction.

Machine language is the only language the processor knows, and in the terms of our analogy of languages programming in assembly can be thought of as conversing with someone who shares the same native language as you, making it the fastest option. At its lowest level machine language is just a string of 1s and 0s as these are the only numbers the processor knows and cares about. Eventually every program, no matter what language it is written in, boils down to machine language that gets loaded into memory, where each instruction and its possible operands are stored into memory as 8-bit values. As remembering hexadecimal values of instructions (much less their binary version) would likely prove difficult if not impossible for most people, assembly language is used instead. Assembly language replaces the numerical representation of instructions with short combination of letters known as *mnemonics*. These mnemonics are generally abbreviations of their function (e.g., LDA – Load Accumulator, BNE – Branch if Not Equal to zero), making them easier to remember. If there can be multiple target registers for a certain operation, it is usually denoted by the last letter of the instruction (e.g., LDA targets A register, LDX targets X register and so on).

As a result, assembly code is usually very cryptic and hard to follow, as the following example for C-64 shows:

```
*=$2000  
LDX #0
```

```
LOOP:    LDA MSG, X
         JSR $FFD2
         INX
         CPX #5
         BNE LOOP
         RTS
MSG:     !SCR "HELLO"
```

Program 3.

Once again, Program 3 behaves the same as Program 1 and Program 2, but without the knowledge of assembly language or an explanation it is almost impossible to understand what happens. Furthermore, as each device family is unique in architecture design compared to those of the other or even the same manufacturers, and the programmer must use peripherals, I/O interfaces or subroutines designed exclusively for that device, programs written in assembly language often need to be rewritten to accommodate the architecture of the target device to achieve the same result. Languages with higher abstraction levels usually work between different platforms, although there may be additional instructions or slight differences in BASIC syntax, and compiled languages need to be recompiled on the target device.

As far as abstraction goes, this is the lowest level, but it also means that nearly everything needs to be done manually, and one must know what to do and how to do it. On the other hand, being able to work on a level as low as this means that truly efficient programs can be written as the programmer has near absolute control over memory, registers, and other parts of the device. For example, as both the instructions and operands for the 6502 are 8 bits in size, and the programmer can access any location in memory, data and code can be used interchangeably. This means that a programmer can read a value from an address that holds an instruction and use it in arithmetic calculations or as a part of an address for indirect addressing modes. Conversely, data stored in memory can be used as an instruction, by moving the PC to point to a suitable address, or an instruction can be changed to another just by writing a new value in its location. When the usable resources are limited, a situation may arise where data in memory must be used in all possible ways with the

least amount of clock cycles, which may require writing a program that is able to modify itself during run time.

4.2 Addressing modes

Addressing mode is the way an instruction accesses or addresses data in memory. All in all, there are 13 different addressing modes – accumulator, absolute, absolute indexed (indexing can be done with both the X and Y registers), implied, immediate, indirect, X indirect, indirect Y, relative, zero page and zero page indexed (again with both the X and Y registers). Some of these addressing modes are similar in function, and for this reason they will be dealt with at the same time in their respective sections. (Synertek, n.d., pp. 7-8)

4.2.1 Accumulator and implied mode

First such a group is the one consisting of accumulator and implied modes. With accumulator mode the instruction targets the data in A register with no additional operands needed (e.g., LSR A, or usually just LSR, performs logical shift right to A). This is similar to Implied addressing mode, where the target is implied by the instruction (e.g., CLI, which clears the interrupt bit or DEX, which decreases the value in X register by one). In fact, these two addressing modes are so identical, that some of the literature only include the implied mode when dealing with the addressing modes. (Zaks, R., 1983, pp. 194-195) (Synertek, 1976a, pp. 57-58)

4.2.2 Absolute and zero page mode

Another group consists of absolute and zero page modes. Opcodes using these modes use their operands as the address where they fetch the data with which they perform the actual operation. Zero page and absolute modes differ only in that absolute mode must fetch two bytes (both high and low bytes of the address), whereas zero page only requires the low byte. For example


```
LDA $0400
```

fetches the value from the absolute address of \$0400 and loads it into the A register. Likewise

```
LDA $10
```

performs a similar operation, only this time the data is fetched from address \$0010. As the high byte is zero there is no need to fetch it, which makes instructions using zero page addressing mode slightly shorter and faster than absolute addressing mode. However, it is often possible to use absolute mode with zero page addresses just by giving a two byte address as an operand (e.g., LDA \$0010), although this is counter-productive both in terms of memory and clock cycle usage. Furthermore, it is uncertain whether all assemblers handle such a case as an absolute mode instruction instead of automatically switching it into a zero page mode instruction. (Zaks, R., 1983, p. 195) (Synertek, 1976a, pp. 59-63)

4.2.3 Absolute indexed and zero page indexed

The third group is closely related to the previous one, as it contains the absolute indexed and zero page indexed modes. With these modes the opcode first fetches the *base* address designated by its operand or operands, after which it adds the contents of an index register (either X or Y) to it to get the final address for fetching data. Once again, zero page needs only one byte for the base address compared to the two used by the absolute mode, but there's also another difference – in indexed zero page mode the address is *confined* to zero page. For example,

```
LDA $FF, X
```

loads data to A register from the address \$00FF+X. Now, assuming X to have the value of 5, the result of \$00FF+\$5 is *not* \$0104, but rather \$0004. If the line of code in question would be

```
LDA $02FF, X
```

the instruction would use the absolute indexed mode, and assuming X would be \$5, the result of \$02FF+\$5 would be \$0304. As with the zero page and absolute modes without indexing, zero page mode uses only two bytes instead of three, but it is *not* necessarily faster. Both modes take at least four clock cycles to execute, but an instruction using the absolute mode may need an

extra cycle, if a page boundary is crossed – i.e., the indexing operation results in the final or *effective address* being on a different page than the base address.

The possibility of using a zero page address as an operand for an absolute mode applies here, too. However, with indexing such a possibility makes more sense, as a programmer may want or need to use the lower part of the stack as a storage. (Zaks, R., 1983, pp. 197-198) (Synertek, 1976a, pp. 79-83)

4.2.4 Immediate mode

Immediate mode uses a literal value, i.e., a value given by the programmer as its operand. For example

```
LDA #$FF
```

loads a hexadecimal value of \$FF to A register, with no additional access to memory. This operand can only be one byte in size. (Synertek, 1976a, p. 58)

4.2.5 Indirect mode

Indirect mode uses its two byte operand as a 16-bit address that contains the low byte of the actual address the data is fetched from. The high byte of the final address comes from the next memory address. It may be helpful for those familiar with C/C++ to think in terms of pointers (pointers being variables that point to the location in memory, i.e., that hold an address of data instead of data itself), as the functionality is very similar. For example, if the address \$1234 holds a value of \$0A, and \$1235 holds a value of \$11, then

```
JMP ($1234)
```

first fetches \$0A from the address \$1234 and uses it as a low byte of the final address. It then fetches \$11 from address \$1235 and uses that as the high byte. Finally, a jump is made to the address \$110A. JMP is the only instruction to make use of pure indirection. (Bjarne, S., 2014, pp. 588) (Bjarne, S., 2014, pp. 590) (Zaks, R., 1983, p. 198) (Synertek, 1976a, pp. 83-84)

4.2.6 X indirect mode

X indirect (or indexed indirect) mode is only possible to do with the X register. Its basic functionality is similar to indirect mode in that it fetches the final address containing the data from the address designated by the operand indexed by X, much like the zero page indexed and absolute indexed modes. For example, assuming X holds a value of \$5, and having a value of \$34 in address \$25 and a value of \$12 in address \$26, then

```
LDA ($20, X)
```

would first set the base address at \$0020, add \$5 to it and then fetch the low byte of the final address from address \$25. As before, the high byte of the 16-bit address comes from address \$26, making the final address of the data \$1234. Unlike the indirect mode, X indirect can only use values stored in zero page to point to the final location. For example, if the operand is \$FF and value of X is \$5, the low byte of the final address is fetched from \$04. (Zaks, R., 1983, pp. 198-199) (Synertek, 1976a, pp. 85-86)

4.2.7 Indirect Y mode

Indirect Y (or indirect indexed) mode can only be used with Y register, and it uses only one byte as an operand, meaning the base address will be on the zero page. However, this time it is the *final* address that is indexed, instead of the address pointing to it. Once again assuming Y to have a value of \$5 and addresses \$25 and \$26 holding \$34 and \$12 respectively, then

```
LDA ($25), Y
```

first fetches the low byte of the final address from address \$25 and the high byte from address \$26, resulting in the final base address being \$1234. It then adds the contents of Y to the base address, making \$1239 the final address for the data. (Zaks, R., 1983, p. 199) (Synertek, 1976a, pp. 87-88)

4.2.8 Relative mode

Finally, there's the relative addressing mode. This mode is only used when performing one of the branching operations, where the program counter is

moved to a different location relative to its current location. As branching operations have only one byte operand, and as the jump is relative rather than absolute, the operand is treated as a *signed* byte. This means that branch can only be taken within the range of [-128, 127] bytes. (Zaks, R., 1983, pp. 191) (Zaks, R., 1983, pp. 196)

4.3 6502 instruction set

The 6502 instruction set comprises 56 different instructions, of which many make use of multiple memory addressing modes that define how the instruction operates in each case. In total there are 151 variations of the 56 base instructions known as operation codes, or *opcodes* (Wikipedia, 2023a). Do note that this thesis will not cover each of these instructions in depth. Instead, a more general look will be taken, and instructions that hold the same purpose are dealt at the same time. A complete list of documented opcodes, their addressing modes, cycle and byte counts and the flags they set is provided in appendix 1.

4.3.1 Variable assignment

The location of the data to be stored or loaded can be given as a literal hexadecimal value, or more conveniently by using *labels*. Labels are names that can be assigned by the programmer for the memory location, which makes the code much more readable and easier to understand. For example,

```
VARIABLE = $A000  
LDA $0200, X  
STA VARIABLE
```

would assign the variable to the address \$A000, load a value from address \$0200 indexed with X, and then store it to the location of VARIABLE.

(Birnbaum, I., 1984, p. 12)

4.3.2 Arithmetic operations

As already mentioned in section 2.6, the 6502 has only two actual arithmetic operations – addition and subtraction. If we exclude the increment (INC, INX, INY) and decrement (DEC, DEX, DEY) operations, that either respectively increment or decrement the value in a memory location or in index registers, we are left only two – ADC (ADd with Carry) and SBC (SuBtract with Carry) that both operate on the value of the A register. ADC adds a value that can either be an immediate value or a value fetched from a memory location plus the carry bit to A. There is no addition instruction without carry, so it must be cleared if it is not needed. SBC subtracts the value (immediate or fetched from memory) and the *negation* of carry bit to A. Here the carry bit serves as a borrow bit, and if a subtraction without borrowing is to be made, the carry flag must be set. (Zaks, R., 1983, p. 113) (Zaks, R., 1983, pp. 145-148) (Zaks, R., 1983, pp. 139-142) (Zaks, R., 1983, p. 173)

4.3.3 Logical operations

Logical operations (AND, ORA and EOR) of the 6502 as well as other bitwise operations (left shift ASL, right shift LSR, left rotation ROL, right rotation ROR) have been discussed more in-depth in section 2.4. These operations can target either the A register, or a memory location and store the result to the same location. In addition to these, BIT is used to perform a bitwise logical AND with A fetched from the memory, the result of which is not stored. The N and V flags set to reflect the 7th and 6th bits of the value in memory, and Z is set to 1 if none of the bits match, and 0 if at least one match was made.

(Zaks, R., 1983, pp. 115-118) (Zaks, R., 1983, p. 122) (Zaks, R., 1983, pp. 143-144) (Zaks, R., 1983, pp. 158-159) (Zaks, R., 1983, pp. 162-163) (Zaks, R., 1983, pp. 167-170)

4.3.4 Comparison operations

Comparison operations (CMP for A, CPX and CPY for the index registers) all perform in a similar way – they subtract a value (an immediate one, or one fetched from memory) from the register without storing the result. The Z flag is set if the result is 0 and the C flag is set if the value in register is greater than or equal to the comparison value. Additionally, the N flag is set to the 7th bit of the *result* of the subtraction operation.

(Zaks, R., 1983, pp. 133-138)

4.3.5 Data transfer operations

Data transfer operations can be divided into two groups – load and store operations. Load operations (LDA, LDX and LDY) load data to either A, X or Y register. Data can be loaded to all registers by using either absolute, absolute indexed, immediate, zero page or zero page indexed modes. Instructions that target either of the index registers (LDX or LDY) can naturally do the indexing only with their counterpart. LDA, on the other hand can use both when using absolute indexed mode, as well as X indirect and indirect Y modes. However, this does not apply to zero page indexed mode, where indexing can only be done with X. This is presumably because the address mode is defined by three bits within the opcode, allowing only for eight different addressing modes.

(Zaks, R., 1983, pp. 152-157)

Store operations (STA, STX and STY), store the contents of a register to memory. STX and STY can only use absolute, zero page and zero page indexed (indexing with their counterparts) modes, and STA can use the same addressing modes as LDA, except for immediate mode.

(Zaks, R., 1983, pp. 178-181)

4.3.6 Flag operations

There are two kinds of flag operations with rather self-explanatory purposes – setting and clearing. A corresponding status flag is turned on with the set instruction and turned off with the clear instruction. Only C, D, I and V flags can be cleared, and only C, D and I can be set by the programmer. Instruction for setting a flag follows the format SE[flag] and instruction for clearing is in the format CL[flag] – for example, SEC sets the carry flag, CLC clears the carry flag. (Zaks, R., 1983, pp. 129-132) (Zaks, R., 1983, pp. 175-177)

4.3.7 Control flow operations

Control flow operations come in two flavours – absolute (JMP, JSR, RTS, BRK and RTI) and relative (BCC, BCS, BEQ, BMI, BNE, BPL, BVC and BVS). The branch or jump address can be given either as a hexadecimal value or by using labels. Similar to the variable assignment, the labels point to the location of the instruction that immediately follows it. For example,

```
VAR = $A000
LOOP:
LDA $0200, X
STA VAR
DEX
BNE LOOP
```

would assign the variable to the address \$A000, load a value from address \$0200 indexed with X, store it to location of VAR, decrement X register and branch back to the address of the LDA if the Z flag is not set. The dynamic nature of labels offers huge benefits compared to using literal and static addresses. Firstly, they allow for making additions and changes to the source code without the need to change the branching or jump addresses manually. Secondly, like with variable assignment, using descriptive names for labels make the source code more understandable.

(Birnbaum, I., 1984, p. 39)

Relative control flow operations are also known as branching operations and depend on the state of certain flags in status register, making them conditional branches. BCC and BCS check the state of carry flag (clear or set,

respectively, as with the following branching pairs), BNE and BEQ check the state of zero flag, BPL and BMI check the state of negative and BVC and BVS check the state of overflow flag. If the branch takes place, the PC is moved to a location relative to its current position, and as was established in section 4.2.8, this needs to be within the range of [-128,127] bytes. As machine languages in general do not contain control flow operations that are commonly found in higher level languages (e.g., if-else or for loops), conditional branching is used as all these operations and their kin rely on some sort of logical condition.

(Zaks, R., 1983, p. 109) (Zaks, R., 1983, pp. 119-121) (Zaks, R., 1983, pp. 123-125) (Zaks, R., 1983, pp. 127-128)

Absolute control flow operations, however, are unconditional and always take place, and can address the whole memory range. JMP jumps to a new location by using an absolute or indirect addressing modes, while JSR first pushes the current location to the stack (high byte first, followed by the low byte) before jumping to the location of a subroutine. RTS returns from that subroutine by pulling the return address from the stack in reverse order. RTI returns from the interrupt (either a software or hardware) by pulling the status register and return address in reverse order they were pushed in. The only difference to RTS is that it pulls the preserved status register from the stack. BRK is a software interrupt instruction intended for debugging that always fetches the interrupt routine address from the IRQ vector \$FFFE-\$FFFF. It behaves similarly to JSR, although as it is comparable to hardware interrupt it also pushes the status register into the stack after the return address. Although it assembles as a one byte instruction, the return address it pushes into the stack is one byte further than one would expect, and the programmer must take this into account by adjusting the return address stored in the stack manually, or by using the extra byte as a breakpoint identifier or an error message if multiple BRK instructions are used during the debug process.

(Zaks, R., 1983, p. 109) (Zaks, R., 1983, pp. 149-151) (Zaks, R., 1983, pp. 171-172) (Birnbaum, I., 1984, pp. 161-162) (Zaks, R., 1983, p. 126)

The original 6502 had a design flaw, in which an indirect JMP would fetch a wrong address if the low byte of the final effective address is fetched from the last byte of the page. In this case the high byte of the effective address is fetched from the first byte of the *same* page, instead of the next page. For example

```
JMP $02FF
```

would fetch the low byte from address \$02FF, and the high byte from \$0200. This behaviour was fixed in the 65c02, which was introduced in 1983. However, this bug is implemented in the emulator. (The Western Design Center, 2018, p. 30)

4.3.8 Stack operations

Although stack can be accessed like any part of the memory, there are four instructions that are dedicated for stack usage – PHA, PHP, PLA and PLP. PHA and PHP first push respectively either A or status register to the current SP location, and then decrement the SP. PLA and PLP first increment the SP, and then pull the value from its location to either A or status register.

(Zaks, R., 1983, pp. 163-166)

4.3.9 Transfer operations

Transfer operations (TAX, TAY, TSX, TXA, TXS and TYA), transfer data between registers and follow the format of T[source][destination]. TAX and TAY transfer the contents from A to X and Y respectively, TXA and TYA from X and Y to A. TSX and TXS transfer data between X and stack pointer. It is good to note that these operations target the stack pointer *register*, not the memory location it points to. For example, if the stack pointer points to \$01FD (as it does after a reset) and X is \$FF, then TSX would result in X having a value of \$FD. Conversely, TXS would set the stack pointer to \$01FF.

(Zaks, R., 1983, pp. 183-187)

4.3.10 NOP

NOP does not really fit into any of the aforementioned categories. Its mnemonic stands for NO oPeration, and the instruction does absolutely nothing for two cycles. As such it is only useful when used to get the timing of the program right or to remove unwanted instructions from the code, although this latter use case has become obsolete with modern cross-compilers. (Zaks, R., 1983, p. 160)

4.3.11 Undocumented opcodes

As each opcode is represented by an 8-bit value, and only 151 opcodes have been implemented, 105 possible combinations are left out. This, however, does not mean that there are only 151 usable opcodes. Instead of storing its instruction decoding logic as a ROM, 6502 uses a programmable logic array that saves memory, but this results in a side effect that was common during the era of its creation – undocumented behaviour. (Wikipedia, 2022j) (Wikipedia, 2023g)

To make a crude simplification, when a new instruction is fetched, it is sent to the PLA for decoding, which makes a comparison between the bits that need to be on, bits that need to be off and the current clock on each line of the PLA. If these match, the signal level of that line is turned on, and the processor performs a task of that clock cycle for that instruction according to which lines have been turned on. As the processor does not know whether the byte fetched as a new opcode is valid or not, it tries to decode them all. A signal of a line in the PLA may be turned on unintentionally, if the mask used to determine the correct bits on assumed opcode is similar enough. This may happen for example with load instructions, where LDX/LDY/LDA work partly identically, and their encoding varies only slightly. As there is no error handling to prune out these invalid opcodes, the 6502 essentially considers each byte sent to the PLA as a valid opcode and tries to execute it. (Steil, M., 2008)

As said, this results in undocumented behaviour in the form of additional opcodes, some of which can be beneficial, but most of which are either unstable, do nothing, or jam the processor. This behaviour was later fixed in 65c02 version, which converts all undocumented opcodes to NOPs with various sizes and clock cycle counts. These additional opcodes are not included in the emulator. (The Western Design Center, Inc., 2018, p. 30)

Using undocumented or illegal opcodes, as they are colloquially known, was not straightforward, as a programmer would have need to have used an assembler that would support them or modify the assembled binary file by hand or by writing a program that modifies itself.

5 THE DEVELOPMENT PROCESS OF THE EMULATOR

This section contains the report of the development process of the emulator application for those interested and as an addition to the main topic of this thesis. As such, an assumption is made that the reader has some preliminary knowledge of programming and is familiar with most of the terms and concepts.

5.1 Tools used

This software was created with Microsoft Visual Studio IDE using C++ programming language and a graphics framework by David Barr called the olcPixelGameEngine (olcPGE). This combination was chosen mainly for the ease of use, portability, and lesser need of other dependencies, such as .NET or Mono. OlcPGE is a simple to use OpenGL based cross-platform framework that has been implemented as a single header file. This allows for quick installation and easy portability to at least Linux platforms, providing they support OpenGL 1.0 and have all necessary graphics libraries installed, and the application does not use Windows specific libraries. The latest version of olcPGE, its optional extensions and documentation can be found at <https://github.com/OneLoneCoder/olcPixelGameEngine>.

Other software used during the development included Gimp as a graphics editor and C64 Studio and an online assembler and disassembler at <https://www.masswerk.at/6502/> for debugging and developing example and demonstration programs. An online simulator at <http://www.visual6502.org/> and Commodore 64 emulator called Vice were also used to compare their behaviour to the emulator being developed.

5.2 Development

The application consists mainly of two classes – one for the processor emulator, and another for the processor monitor. At a later stage a small utility class was added for tracking the time it took to complete the final test phase. The monitor class is mainly just a user interface implementation, and thus will be dealt with in the section 5.3.

A conscious decision was made not to encapsulate the variables used by both the emulator and the monitor. Although not a good programming practice, encapsulation would have meant that individual getters and possibly setters would be needed for each variable, or a struct consisting of all variables needed by the monitor would have to be created, passed to the monitor, and then be extracted as needed. This was deemed more trouble than it was worth, and these variables were left public, with the variables and methods needed only by the emulator being declared private.

5.2.1 Variables

Variables used in the application can be roughly divided into two groups – processor variables and visualization variables, both consisting of various types, such as Booleans, strings, or unsigned integers of different sizes. Visualization variables consist mainly of variables intended purely for providing information for the user, e.g., strings explaining what happens on each cycle, the purpose of current instruction, the number of cycles that have passed, etc. These variables have no effect on the functionality of the emulated processor.

Processor variables, on the other hand, are variables that have a direct effect on the emulation process. In some cases they are also used for visualization purposes, but their main function is to ensure the proper functionality of the emulation. Considering their task, they reflect as close as possible their real life counterparts, e.g., memory is a vector of 65 536 unsigned 8-bit integers, registers are individual unsigned 8-bit integers and the PC and the SP are both 16-bit unsigned integers. Even though the SP is an 8-bit register, it made more

sense to implement it as a 16-bit variable, as it, too, holds an address. A vector in this context is an array-like container that stores data contiguously in memory and has random access. Unlike an array, however, the size of a vector may change dynamically (Bjarne, 2014 pp. 117, 120).

In addition, a few enumerated lists were created to help with determining the addressing modes, distinguish between IRQ and NMI or to handle the individual bits in the status register. Although not strictly speaking necessary, as using normal constants or even variables would certainly have sufficed, enumerations make the code more readable and manageable. For example,

```
enum
{
    C = 1 << 0,
    Z = 1 << 1,
    I = 1 << 2,
    D = 1 << 3,
    B = 1 << 4,
    U = 1 << 5,
    V = 1 << 6,
    N = 1 << 7,
};
```

declares eight constants, each with a value of 1 shifted to the left with a certain number of bits, and the use of scope brackets intuitively sets them apart from other variables or constants, linking them together as a part of the same group. In this collection C equals %0000 0001, Z equals %0000 0010 and so on to N, which equals %1000 0000, giving us a set of named constants that each correspond to their respective bits in the status register. Now setting the zero flag in the status register is as simple as performing a bitwise or with a constant called Z instead of $1 \ll 1$ (or 2 in decimal), which reduces mistakes.

Finally, a few miscellaneous variables were used to determine the current state of the processor, the number of cycles an instruction would take, the opcode to be executed and its current execution phase, and the state (high or low) of clock signal.

5.2.2 Methods

The main bulk of the emulator code is taken by the opcode implementation. Each instruction was implemented as its own private method in the emulator class, and its behaviour reflects that of the single cycle execution summary outlined in appendix A of the Synertek's datasheet for the 6500-processor family (Synertek, n.d., pp. 25-37). To reduce the amount of code and to refrain from using similar code repeatedly, a variety of helper methods were implemented to be used where applicable. For example, the behaviour of the branch operations is identical, bar for the flag and its state used to determine whether branch is to be taken. The principle is the same for the single byte instructions with implied addressing modes, RMW instructions with multiple addressing modes and interrupt/reset sequence to name but a few. Thus, only the instruction specific parts of the execution needed to be implemented in their respective methods, leaving only a few instructions in the need of a full dedicated implementation. A vector of 256 function pointers was created to store the functions for a more convenient calling.

If an instruction has several address modes, the mode used by the currently executing opcode was determined by a separate function – again to reduce the size of the code and repetition. As some of the instructions follow a base pattern in their encoding, it is possible to mask out the bits that make up the addressing mode. To achieve this two enumerated lists of bitmasks were created. The other was based on the information and instruction groups given in appendix 1 of Ian Birnbaum's book (Birnbaum, I., 1984, p. 225), the other consisted of the base pattern of each instruction. As the instructions in group 1A and 1B use the same bits to make up the addressing mode but have slightly different addressing modes, and operations using X and Y registers can only use their counterpart in indexing modes, the other enumeration was needed to handle these edge cases. When determining the addressing mode of an instruction, each instruction passes the bitmask of the group it belongs to this method, which then masks out the corresponding bits and sets the addressing mode accordingly.

Though small and simple, the clock method is, in a way, the main method of the emulator. It is called every frame during the execution of the application, and during each call it advances the clock half a cycle. If the clock has advanced a full cycle, i.e., the state of the clock signal is high, the method enters its two phased execution routine. Firstly, a comparison is made between the current cycle of the opcode being executed and the total cycles needed for its completion. If the execution has been completed, the PC is incremented, and a new opcode is fetched. A new opcode is also fetched after the completion of the reset sequence and when the PC is altered by an interrupt or a control flow instruction, but in these cases the PC is not incremented. Once a new opcode is fetched, the vector containing the disassembly of the next 10 instructions is updated.

The second phase of the clock method is the instruction execution phase. This happens on each complete clock cycle independent on whether a new opcode was fetched or the previous one is still being executed. The clock method will call the method of the proper instruction by using a function pointer stored in the previously mentioned vector. Each method appears in the vector as many times as it has different addressing modes, and their place in the vector is determined by their respective opcode values. This means that calling the correct method is as simple using the value of the fetched opcode as the index for the vector. If a hardware interrupt or a reset was triggered by the user, a corresponding sequence will be run instead of executing the next opcode. The reset sequence will start on the next high signal of the clock cycle, whereas the interrupt sequences will run only after the opcode execution has been completed.

The emulator class also includes two methods for loading assembled binaries into memory – one for custom programs, and one for one the testing suites used to test the emulator. If the user wishes, custom programs may be created for example with the help of an online assembler found at <https://www.masswerk.at/6502/assembler.html>.

In addition to these, a few helper methods were used for triggering interrupts, converting decimal to hexadecimal, setting up flags of the status register and disassembling the code in emulators memory.

5.3 Design

As said, the second major class was mainly an implementation of the user interface by using the olcPGE. It utilizes the public values from the emulator class to provide visualizations of memory and register contents and showing the state of the data and address bus lines as well as part of the code to be executed and the action of the opcode being executed.

The layout of the monitor was kept as simple and intuitive as possible and was arranged so that it follows the natural flow of reading from left to right and top to bottom, with the least significant information being at the lower right corner of the window. (Figure 10)

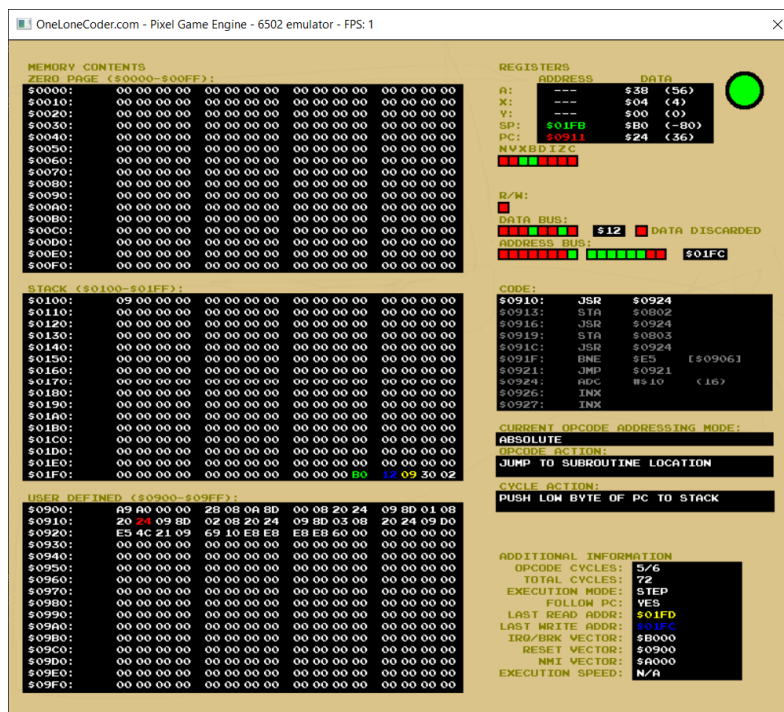


Figure 10. The user interface of the monitor application

The left hand of the screen is dedicated to memory contents. The top two windows show the contents of the zero page and stack respectively, and are

always visible, as these carry more significance than the rest of the memory. The page shown in the third window can be freely chosen by the user, although by default it shows the page the PC is located on. (Figure 11)

MEMORY CONTENTS				
ZERO PAGE (\$0000-\$00FF):				
\$0000:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
\$0010:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
\$0020:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
\$0030:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
\$0040:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
\$0050:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
\$0060:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
\$0070:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
\$0080:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
\$0090:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
\$00A0:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
\$00B0:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
\$00C0:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
\$00D0:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
\$00E0:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
\$00F0:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
STACK (\$0100-\$01FF):				
\$0100:	09 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
\$0110:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
\$0120:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
\$0130:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
\$0140:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
\$0150:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
\$0160:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
\$0170:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
\$0180:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
\$0190:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
\$01A0:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
\$01B0:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
\$01C0:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
\$01D0:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
\$01E0:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
\$01F0:	00 00 00 00	00 00 00 00	00 00 00 B0	12 09 30 02
USER DEFINED (\$0900-\$09FF):				
\$0900:	A9 A0 00 00	28 08 0A 8D	00 08 20 24	09 8D 01 08
\$0910:	20 24 09 8D	02 08 20 24	09 8D 03 08	20 24 09 D0
\$0920:	E5 4C 21 09	69 10 E8 E8	E8 E8 60 00	00 00 00 00
\$0930:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
\$0940:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
\$0950:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
\$0960:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
\$0970:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
\$0980:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
\$0990:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
\$09A0:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
\$09B0:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
\$09C0:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
\$09D0:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
\$09E0:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
\$09F0:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00

Figure 11. Memory section of the user interface

The upper part of the right hand side shows the contents of the registers and the state of the clock signal. Data in each register is given both in hexadecimal and decimal format, the latter being in brackets. The addresses of the SP and the PC are colour-coded so that they can be located easily from the memory. As each bit of the status register is its own value, it is represented with a bank of coloured squares under their respective flags, where set flags are coloured

green and cleared flags are coloured red. The same colour scheme applies to the clock signal as well as to buses— green means the signal is high, red means it is low. (Figure 12)



Figure 12. The contents of the registers and the state of the status register

Like the status register, the buses also have banks of coloured squares to visualize the current state of their lines, and the number of squares naturally reflects the width of each bus. In addition, the value of the buses is shown in hexadecimal next to it. The square under the label “R/W” denotes the state of the read/write pin of the processor – when it is high, the processor reads data from the address shown on the address bus, and it is moved to the data bus. Sometimes data that has been read to the data bus gets discarded, and this is denoted by the coloured square next to the label “DATA DISCARDED”. (Figure 13)

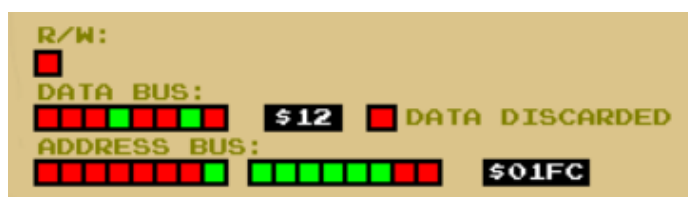


Figure 13. Contents of the buses and the state of the read/write pin

Beneath the buses is a window that shows the next 10 instructions to be executed counting from the current location of the PC, with the topmost one being the one most recently fetched. It will not consider possible branches or jumps. (Figure 14)

```

CODE:
$0910:    JSR    $0924
$0913:    STA    $0802
$0916:    JSR    $0924
$0919:    STA    $0803
$091C:    JSR    $0924
$091F:    BNE    $E5    [ $0906 ]
$0921:    JMP    $0921
$0924:    ADC    # $10    ( 16 )
$0926:    INX
$0927:    INX

CURRENT OPCODE ADDRESSING MODE:
ABSOLUTE
OPCODE ACTION:
JUMP TO SUBROUTINE LOCATION

CYCLE ACTION:
PUSH LOW BYTE OF PC TO STACK

```

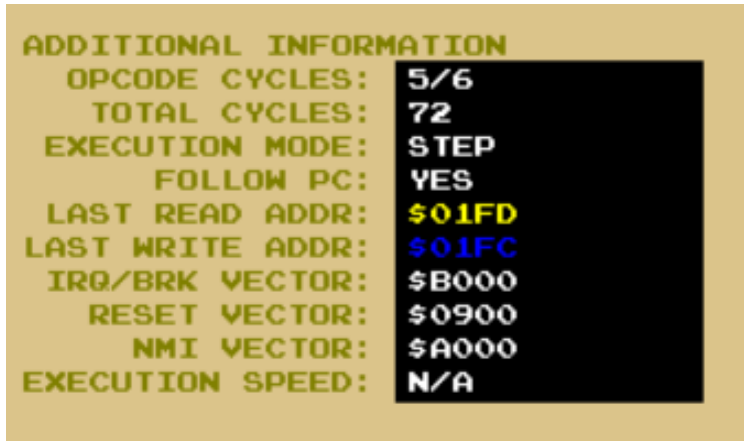
Figure 14. Listing of the next 10 instructions and the addressing mode and the actions of the instruction currently being executed

The code window is divided into four columns – the first column contains the address of the instruction, the second column contains the instruction itself, and the third column contains its operand, if any. The fourth column is only used on three occasions – when an immediate value is used as an operand, in which case its decimal value is given in brackets, and when the instruction is a branch or an indirect jump instruction, in which case the branch or final jump address is shown.

Below the code window the functionality of the opcode is explained. The first field tells the user which addressing mode the current opcode is using, the second tells what it eventually does, and the third field shows what (if anything) happens on each cycle of the execution. This last field mostly reflects the events laid out in the appendix A of the datasheet for the 65xx processor family (Synertek, n.d., p. 25).

Finally, some additional information is provided to the user. The first line shows how many cycles the opcode has executed out of the total needed; the second one shows the amount of clock cycles executed since reset; third one tells the user whether the emulator is in step or continuous execution mode and the

fourth tells if the third shown memory page follows the PC or not. The last read and write addresses are colour-coded for the same reason as SP and the PC, and below them the addresses contained in the three interrupt and reset vectors are shown. Finally, the execution speed of continuous mode is shown. (Figure 15)



```

ADDITIONAL INFORMATION
  OPCODE CYCLES: 5/6
  TOTAL CYCLES: 72
  EXECUTION MODE: STEP
  FOLLOW PC: YES
  LAST READ ADDR: $01FD
  LAST WRITE ADDR: $01FC
  IRQ/BRK VECTOR: $B000
  RESET VECTOR: $0900
  NMI VECTOR: $A000
  EXECUTION SPEED: N/A
  
```

Figure 15. Additional information provided for the user

5.4 Testing

Testing of the emulator was done in three phases and having passed the final test it was deemed completely functional. In the first phase the functionality of each instruction was test superficially as soon as it was implemented to eliminate larger mistakes. The second phase was performed with a test program designed for NES emulators by Kevin Horton, and its purpose was to carry out a preliminary check of functionality and visualization. Finally, the decisive test was conducted by using Klaus Dormann's comprehensive testing suite for 6502.

The reason for using two different test suites was twofold – firstly, the suite for NES emulators lacked the tests for 6502's decimal mode as the chip in NES does not implement it, and it also included the undocumented opcodes which would eventually jam the emulator – although not before testing the official opcodes. On the other hand, using Dormann's test suite proved to be *too* comprehensive to be run using the monitor. The visualizations slow the

framerate to around 150 frames per second on the test machine, and as the clock advances only half a cycle each frame, only about 75 full cycles can be completed each second. As the test takes more than 96 million cycles to complete, performing the complete test suite using the framework is in no way feasible. However, performing the tests only on the *core* of the emulator, i.e., without creating an instance of the monitor class, took only about a minute. As Horton's test suite was not as comprehensive (taking less than 20 thousand cycles before entering the undocumented opcode part), it was well suited for preliminary testing of the official opcodes and how the visualizations worked. Each test suite would signify a failed test by entering an infinite branch loop.

Checking the validity of Dormann's test without the monitor was done by examining the original source code and the disassembly of its binary, and by interrogating address \$0200 of the emulator that holds the running number of the most recent test passed. By reading the source code it was established that the last value to be stored in that location would be \$F0 (240 in decimal), after which the program would enter an infinite loop at a certain memory location. Further examination and comparison of the disassembly and the original source code revealed this location to be at \$3469.

A simple core test section was implemented to check these values, and it remains slightly altered as an option for those who wish to check the integrity of the emulator. As Dormann's test suite is published under GPL-3 license, and thus can be freely redistributed, its binary is included in the GitHub repository (<https://github.com/Jpaaltonen/6502Emulator>) along with the other resources (the emulator source code, windows binary, and a demonstration program). The user will be prompted at the start of the application and may choose to run either the monitor or the core test, which will exit the program after the tests are completed. The binary can also be loaded while using the monitor, although, as has already been mentioned, running it completely in this mode is not recommended.

6 USING THE EMULATOR

When the application starts, the user is prompted to choose between running the core test and starting the monitor. As previously mentioned, running the core test lacks all the visualizations, and will take about a minute to complete, after which the application will exit. Most of the time (around 55 seconds on the test machine) will be taken by test 41, which is the comprehensive addition/subtraction test, while most of the other tests take only fractions of a second.

The monitor application has two execution modes – continuous and step mode, between which the user may choose at will, although by default the monitor will run in step mode. The continuous mode will advance the clock continuously according to the set speed, which may be altered by the user. The step mode, however, will advance the clock half a cycle only when prompted to by the user, which gives the user more time to examine the contents of the registers and memory and the actions of the currently executing instruction. All available information is presented to the user, as was shown in section 5.3.

While the zero page and stack are always visible, the third page shown is left for the user to choose. The user may choose between following the PC, whereupon the page shown is always the one the PC is currently on, or the user may turn this feature off and move freely with either keyboard or console commands.

The user has also the ability to trigger a hardware interrupt (either maskable or non-maskable) and the reset sequence of the emulator. Note that the reset is a so called *warm reset* as opposed to a *cold reset*, meaning that the loss of power is not simulated, and the data present in memory and the registers at the moment of reset stays intact, except for the stack pushes the reset sequence performs. The reset sequence is also triggered each time a new program is loaded into memory.

6.1 Controls and commands

While olcPGE is simple and easy to use framework, what it really lacks is decent set of user interface building tools. Thus, the emulator only consists of keyboard and console commands. These have been kept to a minimum and as intuitive as possible. (Table 1) (Table 2)

	KEYBOARD COMMANDS
F1	Increase speed for continuous mode
F2	Decrease speed for continuous mode
ESC	Open/close console
SPACE	Advance clock in step mode
LEFT	Previous memory page (wraps around), not applicable if following the PC
RIGHT	Next memory page (wraps around), not applicable if following the PC
UP	Increase memory page by \$10 (wraps around), not applicable if following the PC
DOWN	Decrease memory page by \$10 (wraps around), not applicable if following the PC
C	Toggle between continuous and step modes
F	Toggle the PC following mode on or off
I	Trigger interrupt request
N	Trigger non-maskable interrupt request
R	Trigger reset sequence

Table 1. Keyboard commands of the emulator

	CONSOLE COMMANDS
cls	Clear the console
load [file]	Load a binary file. Both the command and the operand are case-sensitive
jump [address]	Move the PC to the address given by the user. The command is case sensitive, and the address is given in

	hexadecimal, either with the prefix '\$' or without it. Note that olcPGE uses UK keyboard layout, and the combination for '\$' is Shift+4.
--	--

Table 2. Console commands for the emulator

6.2 Loading custom programs

As said, the user may create custom programs for the emulator. When creating these it is important to note that the loader method of the emulator assumes the first two bytes of the binary file to contain the start address the binary is meant to be loaded to. Furthermore, the reset vector must be set to point to the location the program execution is meant to start. For example,

```
LOAD = $800
PRG_START = $900
```

```
* = LOAD
.WORD LOAD
;POSSIBLE PROGRAM CODE AND VARIABLES
```

```
* = START
;START LOCATION OF EXECUTION
;PROGRAM CODE
```

```
* = $FFFC
.WORD START
```

would load a program into memory starting from location \$800 with the two first bytes being \$00 and \$08 respectively, which is the assembly address. The reset vector contains values \$00 and \$09, which is the start location of the execution, and where the PC will be transferred to. The '*' signifies the address (which can be given either as a literal value or as a variable) the assembler will assemble the following instructions or data, and the ".WORD" tells the assembler to insert a 16-bit value to the current location in little endian format. These are called *directives* that tell the assembler how and where to assemble the source code and may differ between assemblers. The semi-colon is used as a comment marker, meaning that everything following it on that line are to be ignored by the assembler.

If interrupts are to be used, their vectors must be set accordingly, for example,

```
IRQ = $A000
```

```
NMI = $B000
```

```
* = $FFFA
```

```
.WORD NMI
```

```
* = $FFFE
```

```
.WORD IRQ
```

would set the IRQ vector to point to the location \$A000 and NMI vector to point to the location \$B000.

The templates above can be used for example with the aforementioned online assembler. As previously mentioned, the syntax of directives may vary between assemblers, meaning that if another assembler is used, this must be accounted for.

7 CONCLUSION

The whole process of writing this thesis and developing the emulator application has been an educational experience. Although a lot of the topics this work covers were based on personal interest and knowledge that had already been acquired some 30 years ago, a lot of it had already been forgotten and at that time the learning process consisted mostly of word of mouth and trial and error rather than research. Being forced to read through several books and datasheets to explain things even superficially instead of saying “it just works like this” has given new insights on both how processors (especially the 6502) work in general, and what actually happens when programming the 6502.

This thesis is by no means meant to be a definitive guide for the 6502 - a whole book could be dedicated solely to cover the technical aspects of the processor, and another one to cover how to program it. And indeed, many have already been written. But as the best possible attempt has been made to keep the most important parts and concepts in the text as clear and the emulator as intuitive and informative as possible, as an introductory package to the 6502 and understanding processors in general, it can be said that it achieves its goal while keeping the scope in decent limits.

REFERENCES

Zaks, R. (1983). Programming the 6502 (fourth edition). Sybex.

Lampton, C. (1985). 6502 Assembly-language programming for Apple, Commodore and Atari computers. Franklin Watts.

Carr, J. (1984). 6502 User's manual. Reston Publishing Company, Inc.

Synertek. (1976a). SY6500/MCS6500 Microcomputer family programming manual. Synertek.

Synertek. (n.d.). SY6500 8-bit Microprocessor Family. Synertek.

Birnbaum, I. (1984). Assembly Language Programming for the BBC Microcomputer (second edition). Macmillan Publishers Ltd

Synertek. (1976b). SY6500/MCS6500 Microcomputer family hardware manual. Synertek.

Tootill, A. & Barrow, D. (1984). 6502 Machine Code for Humans. Granada

Smith, E., Bacon, D., Barnes, N., Beards, C., Bevan, J., Blaen, R., Blazynski, T., Carvill, J., Clarkson, T., Compton, P., Coveney, V., Cullum, R., Davies, A., Easton, R., Eliades, P., Enright, D., Fraser, C., Goodger, E., Gregory, E., ... Wyatt, L. (1994). Mechanical engineer's reference book (twelfth edition). Butterworth-Heinemann Ltd

The Western Design Center, Inc. (2018). W65C02S 8-bit Microprocessor. The Western Design Center, Inc.

Espinosa, C. (1979). Apple II reference manual. Apple Computer Inc.

Bjarne, S. (2014). Programming: Principles and Practice using C++ (second edition). Addison-Wesley.

Thakur, A. (2019). Negative Binary Numbers. Tutorialspoint. <https://www.tutorialspoint.com/negative-binary-numbers>

Doe, J. (n.d.). Pulse Circuits – Signal. [An online education site concentrated on computer science and other engineering subjects] https://www.tutorialspoint.com/pulse_circuits/pulse_circuits_signal

Regmi, S. (2020, January 17). How to Calculate the Square Root of a Number? – Newton-Raphson Method. Medium. <https://surajregmi.medium.com/how-to-calculate-the-square-root-of-a-number-newton-raphson-method-f8007714f64>

Lateblt (2012, April 25) The 6502's two clock signals. Livejournal. <https://lateblt.livejournal.com/88105.html>

Wong, C., (n.d.) 6502 Assembly – 6502 Architecture – Register: status – Flags. Google Sites. <https://sites.google.com/site/6502assembly/6502-architecture/registerstatus/flags/flagzero>

Steil, M. (2008, July 29) How MOS 6502 Illegal Opcodes really work. <https://www.pagetable.com/?p=39>

Wilson, G. (2014, May 17). Investigating Interrupts. 6502.org. <http://6502.org/tutorials/interrupts.html>

Computer Hope. (2017, April 26). Clock cycle. <https://www.computerhope.com/jargon/c/clockcyc.htm>

Forster, J. (n.d.). Commodore 64 memory map <https://sta.c64.org/cbm64mem.html>

Wilson, G. (2022, December 19). Address Decoding. Retrieved March 3, 2023, from http://wilsonminesco.com/6502primer/addr_decoding.html

Wilson, G. (2020, December 13). Memory Map Requirements. Retrieved March 3, 2023, from <http://wilsonminesco.com/6502primer/MemMapReqs.html>

Nesdev. (2023a, February 16). PPU Palettes. Retrieved March 3, 2023, from https://www.nesdev.org/wiki/PPU_palettes

Nesdev. (2023b, January 11). CPU memory map. Retrieved March 3, 2023, from https://www.nesdev.org/wiki/CPU_memory_map

Nesdev. (2022, June 28). Memory Mirroring. Retrieved March 3, 2023, from <https://www.nesdev.org/wiki/Mirroring>

C64-Wiki. (2023, February 19). REU. Retrieved March 3, 2023, from <https://www.c64-wiki.com/wiki/REU>

C64-Wiki. (2022, January 4). Bank Switching. Retrieved March 3, 2023, from https://www.c64-wiki.com/wiki/Bank_Switching

C64-Wiki. (2021, February 1). SID. Retrieved March 3, 2023, from <https://www.c64-wiki.com/wiki/SID>

Wikipedia. (2022a, November 29). Interrupts in 65xx processors. Retrieved March 3, 2023, from https://en.wikipedia.org/wiki/Interrupts_in_65xx_processors

Wikipedia. (2023a, March 3). MOS Technology 6502. Retrieved March 3, 2023, from https://en.wikipedia.org/wiki/MOS_Technology_6502

Wikipedia. (2022b, October 24). Hardware register. Retrieved March 3, 2023, from https://en.wikipedia.org/wiki/Hardware_register

Wikipedia. (2022c, December 22). Clock generator. Retrieved March 3, 2023, from https://en.wikipedia.org/wiki/Clock_generator

Wikipedia. (2022d, December 20). Interval (mathematics). Retrieved March 3, 2023, from [https://en.wikipedia.org/wiki/Interval_\(mathematics\)](https://en.wikipedia.org/wiki/Interval_(mathematics))

Wikipedia. (2023b, February 20). Boolean algebra. Retrieved March 3, 2023, from https://en.wikipedia.org/wiki/Boolean_algebra

Wikipedia. (2023c, January 26). Bitwise operation. Retrieved March 3, 2023, from https://en.wikipedia.org/wiki/Bitwise_operation

Wikipedia. (2022e, August 11). Non-maskable interrupt. Retrieved March 3, 2023, from https://en.wikipedia.org/wiki/Non-maskable_interrupt

Wikipedia. (2023d, March 1). Byte addressing. Retrieved March 3, 2023, from https://en.wikipedia.org/wiki/Byte_addressing

Wikipedia. (2023e, January 16). Random-access memory. Retrieved March 3, 2023, from https://en.wikipedia.org/wiki/Random-access_memory

Wikipedia. (2022f, January 24). Sequential access. Retrieved March 3, 2023, from https://en.wikipedia.org/wiki/Sequential_access

Wikipedia. (2022g, December 28). Photolithography. Retrieved March 3, 2023, from <https://en.wikipedia.org/wiki/Photolithography>

Wikipedia. (2023f, February 27). Retrieved March 3, 2023, from https://en.wikipedia.org/wiki/Read-only_memory

Wikipedia. (2022h, June 9). Bank switching. Retrieved March 3, 2023, from https://en.wikipedia.org/wiki/Bank_switching

Wikipedia. (2022i, October 28). Stack overflow. Retrieved March 3, 2023, from https://en.wikipedia.org/wiki/Stack_overflow

Wikipedia. (2022j, December 26). Programmable logic array. March 3, 2023, from https://en.wikipedia.org/wiki/Programmable_logic_array

Wikipedia. (2023g, January 19). Illegal opcode. March 3, 2023, from https://en.wikipedia.org/wiki/Illegal_opcode

APPENDIX 1: THE 6502 OPCODE MATRIX

The matrix in figure 16 contains all the documented opcodes for the 6502 with their respective addressing modes and cycle and byte counts. The topmost row contains the low nibble of the opcode, and leftmost column contains the high nibble. For example, opcode \$10 is BPL, which has a relative addressing mode, uses two bytes (the opcode itself and a 1 byte operand) and execution takes 2-4 cycles to complete – two at the minimum, plus an additional cycle if a branch is taken, and another one if the page boundary is crossed. Table 3 holds a legend for this matrix.

	-0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-A	-B	-C	-D	-E	-F
0-	BRK imp cycles: 2 bytes: 1	ORA idx cycles: 6 bytes: 2	???	???	???	ORA zp cycles: 3 bytes: 2	ASL zp cycles: 5 bytes: 2	???	PHP imp cycles: 3 bytes: 1	ORA imm cycles: 2 bytes: 2	ASL acc cycles: 2 bytes: 1	???	???	ORA abs cycles: 4 bytes: 4	ASL abs cycles: 6 bytes: 4	???
1-	BPL rel cycles: 2* bytes: 1	ORA idy cycles: 6 bytes: 2	???	???	???	ORA zpx cycles: 4 bytes: 2	ASL zpx cycles: 6 bytes: 2	???	CLC imp cycles: 2 bytes: 1	ORA aby cycles: 4+ bytes: 3	???	???	???	ORA abx cycles: 4+ bytes: 3	ASL abx cycles: 7 bytes: 3	???
2-	JSR abs cycles: 6 bytes: 3	AND idx cycles: 6 bytes: 2	???	???	BIT zp cycles: 3 bytes: 2	AND zp cycles: 3 bytes: 2	ROL zp cycles: 5 bytes: 2	???	PLP imp cycles: 42 bytes: 1	AND imm cycles: 2 bytes: 2	ROL acc cycles: 2 bytes: 1	???	BIT abs cycles: 4 bytes: 3	AND cycles: 4 bytes: 3	ROL abs cycles: 6 bytes: 3	???
3-	BMI rel cycles: 6 bytes: 2	AND idy cycles: 5+ bytes: 2	???	???	???	AND zpx cycles: 4 bytes: 2	ROL zpx cycles: 6 bytes: 2	???	SEC imp cycles: 2 bytes: 1	AND aby cycles: 4+ bytes: 3	???	???	???	AND abx cycles: 4+ bytes: 3	ROL abx cycles: 7 bytes: 3	???
4-	RTI imp cycles: 6 bytes: 1	EOR idx cycles: 6 bytes: 2	???	???	???	EOR zp cycles: 3 bytes: 2	LSR zp cycles: 5 bytes: 2	???	PHA imp cycles: 3 bytes: 1	EOR imm cycles: 2 bytes: 2	LSR acc cycles: 2 bytes: 1	???	JMP abs cycles: 3 bytes: 3	EOR abs cycles: 4 bytes: 4	LSR abs cycles: 6 bytes: 3	???
5-	BVC rel cycles: 2* bytes: 2	EOR idy cycles: 5+ bytes: 2	???	???	???	EOR zpx cycles: 4 bytes: 2	LSR zpx cycles: 6 bytes: 2	???	CLI imp cycles: 2 bytes: 1	EOR aby cycles: 4+ bytes: 3	???	???	???	EOR abx cycles: 4+ bytes: 3	LSR abx cycles: 7 bytes: 3	???
6-	RTS imp cycles: 6 bytes: 1	ADC cycles: 6 bytes: 2	???	???	???	ADC zp cycles: 3 bytes: 2	ROR zp cycles: 5 bytes: 2	???	PLA imp cycles: 4 bytes: 1	ADC imm cycles: 2 bytes: 2	ROR acc cycles: 2 bytes: 1	???	JMP ind cycles: 5 bytes: 3	ADC abs cycles: 4 bytes: 3	ROR abs cycles: 6 bytes: 3	???
7-	BVS rel cycles: 2* bytes: 2	ADC idy cycles: 5 bytes: 2	???	???	???	ADC zpx cycles: 4 bytes: 2	ROR zpx cycles: 6 bytes: 2	???	SEI imp cycles: 2 bytes: 1	ADC aby cycles: 4+ bytes: 3	???	???	???	ADC abx cycles: 4+ bytes: 3	ROR abx cycles: 7 bytes: 3	???
8-	???	STA idx cycles: 6 bytes: 2	???	???	STY zp cycles: 3 bytes: 2	STA zp cycles: 3 bytes: 2	STX zp cycles: 3 bytes: 2	???	DEY imp cycles: 2 bytes: 1	???	TXA imp cycles: 2 bytes: 1	???	STY abs cycles: 4 bytes: 3	STA abs cycles: 4 bytes: 3	STX abs cycles: 4 bytes: 3	???
9-	BCC rel cycles: 2* bytes: 2	STA idy cycles: 6 bytes: 2	???	???	STY zpx cycles: 4 bytes: 2	STA zpx cycles: 4 bytes: 2	STX zpx cycles: 3 bytes: 2	???	TYA imp cycles: 2 bytes: 1	STA aby cycles: 5 bytes: 3	TXS imp cycles: 2 bytes: 1	???	???	STA abx cycles: 5 bytes: 3	???	???
A-	LDY imm cycles: 2 bytes: 2	LDA idx cycles: 6 bytes: 2	LDX imm cycles: 2 bytes: 2	???	LDY zp cycles: 3 bytes: 2	LDA zp cycles: 3 bytes: 2	LDX zp cycles: 3 bytes: 2	???	TAY imp cycles: 2 bytes: 1	LDA imm cycles: 2 bytes: 2	TAX cycles: 2 bytes: 1	???	LDY abs cycles: 4 bytes: 3	LDA abs cycles: 4 bytes: 3	LDX abs cycles: 4 bytes: 3	???
B-	BCS rel cycles: 2* bytes: 2	LDA idy cycles: 5+ bytes: 2	???	???	LDY zpx cycles: 4 bytes: 2	LDA zpx cycles: 4 bytes: 2	LDX zp cycles: 3 bytes: 2	???	CLV imp cycles: 2 bytes: 1	LDA aby cycles: 4+ bytes: 3	TSX imp cycles: 2 bytes: 1	???	LDY abx cycles: 4+ bytes: 3	LDA abx cycles: 4+ bytes: 3	LDX abx cycles: 4+ bytes: 3	???
C-	CPY imm cycles: 2 bytes: 2	CMP idx cycles: 6 bytes: 2	???	???	CPY zp cycles: 3 bytes: 2	CMP zp cycles: 3 bytes: 2	DEC zp cycles: 5 bytes: 2	???	INY imp cycles: 2 bytes: 1	CMP imm cycles: 2 bytes: 2	DEX imp cycles: 2 bytes: 1	???	CPY abs cycles: 4 bytes: 3	CMP abs cycles: 4 bytes: 3	DEC abs cycles: 6 bytes: 3	???
D-	BNE rel cycles: 2* bytes: 2	CMP idy cycles: 6 bytes: 2	???	???	???	CMP zpx cycles: 4 bytes: 2	DEC zpx cycles: 6 bytes: 2	???	CLD imp cycles: 2 bytes: 1	CMP aby cycles: 4+ bytes: 3	???	???	???	CMP abx cycles: 4+ bytes: 3	DEC abx cycles: 7 bytes: 3	???
E-	CPX imm cycles: 2 bytes: 2	SBC idx cycles: 2 bytes: 6	???	???	CPX zp cycles: 3 bytes: 2	SBC zp cycles: 3 bytes: 2	INC zp cycles: 5 bytes: 2	???	INX imp cycles: 2 bytes: 1	SBC imm cycles: 2 bytes: 2	NOP imp cycles: 2 bytes: 1	???	CPX abs cycles: 4 bytes: 3	SBC abs cycles: 4 bytes: 3	INC abs cycles: 6 bytes: 3	???
F-	BEQ rel cycles: 2* bytes: 2	SBZ idy cycles: 5* bytes: 2	???	???	???	SBC zpx cycles: 4 bytes: 2	INC zpx cycles: 6 bytes: 2	???	SED imp cycles: 2 bytes: 1	SBC aby cycles: 4+ bytes: 3	???	???	???	SBC abx cycles: 4+ bytes: 3	INC abx cycles: 7 bytes: 3	???

Figure 16. Opcode matrix for the 6502 instruction set

SYMBOL/ABBREVIATION	MEANING
BRK	Opcode
???	Undocumented opcode
Acc	Accumulator
Abs	Absolute
Abx	Absolute, X indexed
Aby	Absolute, Y indexed
Idx	X indirect
Idy	Indirect Y
Imm	Immediate
Imp	Implied
Ind	Indirect
Rel	relative
Zp	Zero page
Zpx	Zero page, X indexed
Zpy	Zero page, Y indexed
*	+1 if branch taken, +1 if page boundary crossed
+	+1 if page boundary crossed

Table 3. legend for the opcode matrix

The table in figure 17 contains all the official instructions in the 6502 instruction set, and indicates which flags in the status register, if any, their use may affect.

	N	V	D	I	Z	C
ADC	●	●			●	●
AND	●				●	
ASL	●				●	●
BCC						
BCS						
BEQ						
BIT	●	●			●	
BMI						
BNE						
BPL						
BRK				●		
BVC						
BVS						
CLC						●
CLD			●			
CLI				●		
CLV		●				
CMP	●				●	●
CPX	●				●	●
CPY	●				●	●
DEC	●				●	
DEX	●				●	
DEY	●				●	
EOR	●				●	
INC	●				●	
INX	●				●	
INY	●				●	
JMP						

	N	V	D	I	Z	C
JSR						
LDA	●				●	
LDX	●				●	
LDY	●				●	
LSR	●				●	●
NOP						
ORA	●				●	
PHA						
PHP						
PLA	●				●	
PLP	●	●	●	●	●	●
ROL	●				●	●
ROR	●				●	●
RTI	●	●	●	●	●	●
RTS						
SBC	●	●			●	●
SEC						●
SED			●			
SEI				●		
STA						
STX						
STY						
TAX	●				●	
TAY	●				●	
TSX	●				●	
TXA	●				●	
TXS						
TYA	●				●	

Figure 17. Flags affected by the instructions