Introduction

The 6502 is an 8-bit little-Indian processor with 5 registers that address at most 64Kb of memory via a 16-bit address bus [1]. In the context of the project, only 4 addressing modes have been used: immediate, absolute, relative, and absolute indexed addressing. The objective of the project is to write a program that sums up numbers and prints the sum.

Summary of Program

The program:

- accepts values to be used as input from the keyboard and writes the calculated sum to the screen upon inputting "=".
- accepts infinitely 4-digit numbers if the sum is less than 10,000.
- features error-handling and prints hello-instructions and input examples.
- The code is structured in 3 main parts: main() method, print() methods, and arithmetic() methods, each part is visibly separated. Every line of code is annotated so that someone with no prior knowledge can understand what the program is trying to do.

Registers

There is a total of five registers, most of which have been used:

- Accumulator(A) which handles logic and arithmetic.
- X, Y registers for temporarily saving a value.
- S: a stack pointer that is used as an offset when a JSR (jump to subroutine) is executed. In other words, it tells the CPU where to return when the subroutine ends.
- P: a processor status that stores a 1 or a 0 for each flag in each one of its bits. Flags are negative, overflow, BRK command, decimal mode, IRQ disable, zero, carry. If a flag is triggered, it can allow for branching.
  e.g.: BEQ print_char -> goto print_char when the zero-flag is triggered.

Addressing Modes

i. **Immediate addressing**: The given value is a number that is used immediately by the instruction.
   e.g.: LDA #$00 loads the value $0 into the accumulator.
ii. **Absolute addressing**: The value given is the address of a memory location that will be used.
   e.g.: STA char -> which is equivalent to STA $2A22 because that is the address of variable char.
iii. **Relative addressing**: branch instructions require relative addressing. The next byte is a signed offset from the current address, and the net sum is the address of the next instruction executed. This offset is a 1 byte signed value, therefore branches can only span from -128 to +127 bytes relative to the next instruction address [1]. If the calculated offset is out of the above range, errors are thrown, and compilation is aborted (discussed more in "Error Handling")
   e.g.: BEQ print_overflow -> BEQ branches on zero flag in P register
iv. **Absolute Indexed Addressing**: The final address is the sum of the given address (known as base) and the value of X register (knows as offset)
   e.g.: LDA char,X -> where X contains 1 -> Accumulator is loaded with: $2A22 + 1 = $2A23

Arithmetic & IO (input/output)

Both addition and subtraction implement the carry flag to track carries. However, in the case of subtraction, it is necessary to set the carry flag as it is the opposite of the carry that is subtracted.

Since we are required to use the input and output subroutines, FFD2 and FFE4 respectively, the scanner needs to point to the numbers that are stored between $0030 a $0040 (in hex). So, if we simply stick to the default addition, an input of 1 + 1 which corresponds to #$0031 + #$0031 will output the value stored at #$0062 which does not equate to 2 (which is originally stored at #$0032). To fix this issue, every input is subtracted by 30 and then is stored, then the sum is added to 30 and is printed. Thus, an input of 1 + 1 = #$0001 + #$0001 = #$0002 = 2 and #$0032 is printed (which corresponds to 2).

Addition and Variable Handling

Users can input infinitely 4-digit numbers if the sum is less than 10,000. Let us assume that the user inputs 777+91+9+123= (result is 1000). The program executes these steps:

1. Typing “777+” triggers the program to execute this addition in the background:
   sum = sum + 777 = 000 + 777 = 777
2. Afterwards, typing “91+” triggers the program to execute this addition in the background:
   sum = sum + 91 = 777 + 91 = 868
3. Afterwards, typing “9+” triggers the program to execute this addition in the background:
   sum = sum + 9 = 868 + 9 = 877
4. Afterwards, typing “3=” triggers the program to execute this addition in the background:
   sum = sum + 123 = 877 + 123 = 1000

N.B.: The addition above occurs only when the user inputs “+” or “=”. The input digits are stored instantly. Let us take 123 as an example. This is how the number is stored:

1. When 1 is inputted
   1.1. char = 1
   1.2. digit1 = char = 1
2. When 2 is inputted
   2.1. char = 2
   2.2. digit2 = digit1 = 1
   2.3. digit1 = char = 2
3. When 3 is inputted
   3.1. char = 3
   3.2. digit3 = digit2 = 1
   3.3. digit2 = digit1 = 2
   3.4. digit1 = char = 3

Let us look at the 4th addition in depth (877+123):

1. Before typing “=”:
   1.1. sum1 = 7
   1.2. sum2 = 7
   1.3. sum3 = 8
   1.4. sum4 = 0
   1.5. digit1 = 3
   1.6. digit2 = 2
   1.7. digit3 = 1
   1.8. digit4 = 0
2. after typing “=”:
   2.1. sum1 = sum1 + digit1 = 7 + 3 = 10 -> if sum1>=10: carry1++ & sum1 = sum1 – 10 = 10 – 10 = 0
   2.2. sum2 = sum2 + digit2 + carry1 = 7 + 2 + 1 = 10 -> carry2++ & sum2 = sum2 – 10 = 10 – 10 = 0
   2.3. sum3 = sum3 + digit3 + carry2 = 8 + 1 + 1 = 10 -> carry3++ & sum3 = sum3 – 10 = 10 – 10 = 0
   2.4. sum4 = sum4 + digit4 + carry3 = 0 + 0 + 1 = 1
   2.5. carry4 ≠ 0 therefore print sum4, sum3, sum2, sum1 (without overflow). 1000 is printed.

Error Handling:

The program tries to replicate the behavior of a calculator without compromising the quality of use:

1. Input scanner only accepts: 0-9, "=", and "+".
2. The program accepts 4 digits at most.
3. Given the program accepts 1-to-4-digit numbers, if the sum is >= 10000, the statement “overflow!” is printed and the program is halted.
4. Upon extensive testing, the commodore simulator misses this input “+” once a dozen times. Therefore, the user is to be aware of this situation in the printed instruction upon starting the program.
5. Out-of-range branching error has been handled using a workaround [2] found in an online article. “Since branching uses relative addressing, the operand of branch instructions in 6502 is an offset from the address of the next instruction. The assembler will calculate the corresponding offset. If the calculated offset is out of the above range, errors are thrown, and compilation is aborted” [2]. The workaround proposed by the article is to invert the branch condition and make it skip a "JMP" instruction, as this uses a full address as an operand and can jump anywhere in memory.

References:

[1] https://dwheeler.com/6502/oneelkruns/asm1step.html

[2] https://atariage.com/forums/topic/265756-need-some-help-with-labels-and-branching/