



NUMERICAL DATA PROGRAM

ALEXANDRU C.

Table of Contents

Introduction	1
1.Circuit model	1
2. Choice of data structures	8
3.Program structure and description	9
4.Algorithm	11
5.Testing	16
5.1. Result value check	16
5.2. Overflow flag test	19
5.3. Invalid input	20
6.Reflection	22
References	23
Appendices	24
Appendix A: Timed run source code	24
Appendix B: Unsigned addition value test source code	25
Appendix C: Signed addition value test source code	25

Introduction

Bit adder circuits are crucial to every processor as they allow processing of information in meaningful ways. The scope of this project is to model such a circuit and then simulate its behaviour in software. The result is to gain deep insight into how these circuits operate.

1.Circuit model

The bit adder circuit model was built to be able to handle both signed and unsigned numbers. Overflow will be handled slightly differently, similar to an adder-subtractor circuit. Further in this document, the model will be presented in steps starting with a simple bit adder then arriving at the final model.

A simple bit adder has 7 main components, namely: upper bits (the first number), lower bits (the second number), result bits, half adder (HA), full adder (FA), the overflow flag, and carry bits (for each adder/ half adder).

Before going into what is a full adder and a half adder, it is useful to look at how everything is wired and works in ensemble.

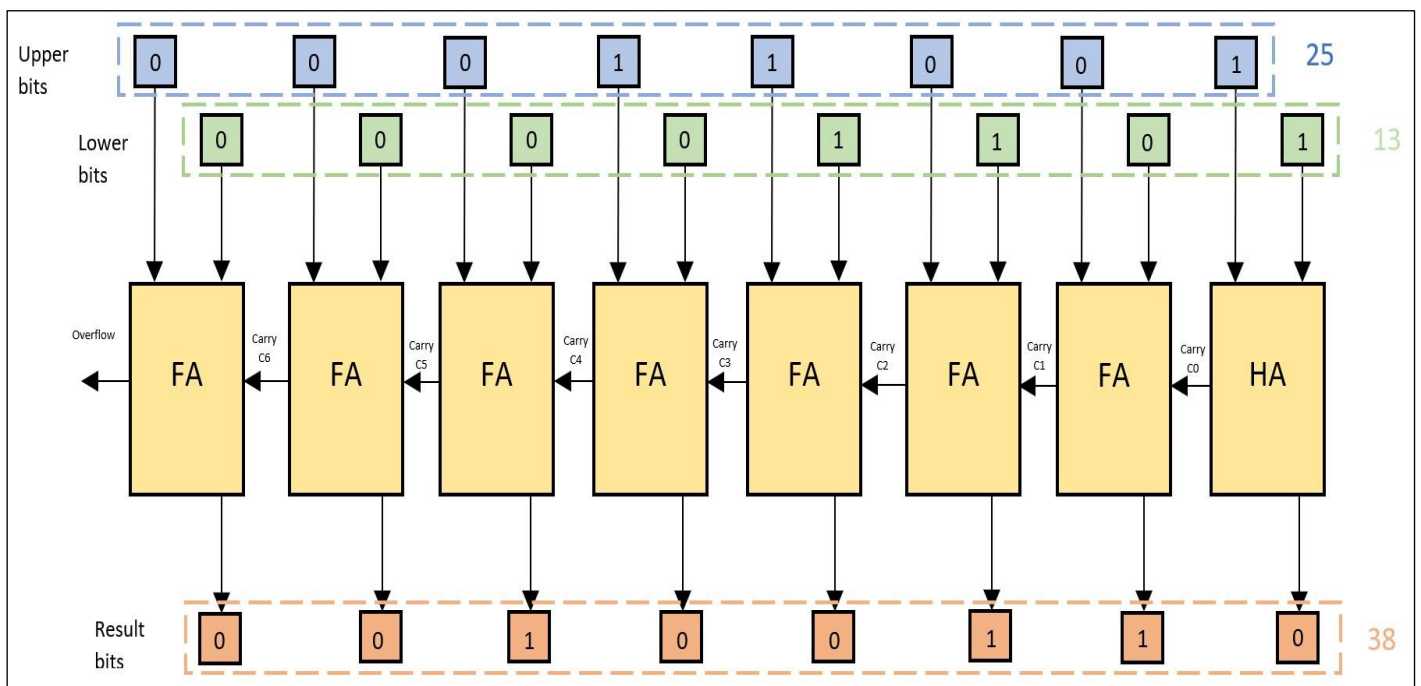


Figure 1. Unsigned bit adder circuit

The upper and lower bits represent the operands of the addition. Each pair of upper bit and lower bit that have the same position in the sequence are input into their respective full adders. An

exception being the first pair of bits, these are input into the half adder. The half adder takes the input and adds them together while the full adder also takes into consideration the carry flag from previous adder.

Overflow flag represents whether the result obtained is correct or not. For unsigned numbers if the following operation is carried out, $250 + 15$, the result obtained will be 9 and the carry flag will be set to a value of 1. This means the result value exceeds the range that can be represented with the given number of bits (8 bits in this case) and the result is incorrect.

Since the first pair of bits have no previous carry bit to take into consideration, there is no need for the logic gates that deal with the carry in. In essence, this is what the half adder is. As illustrated below, the half adder is a simpler version of the full adder.

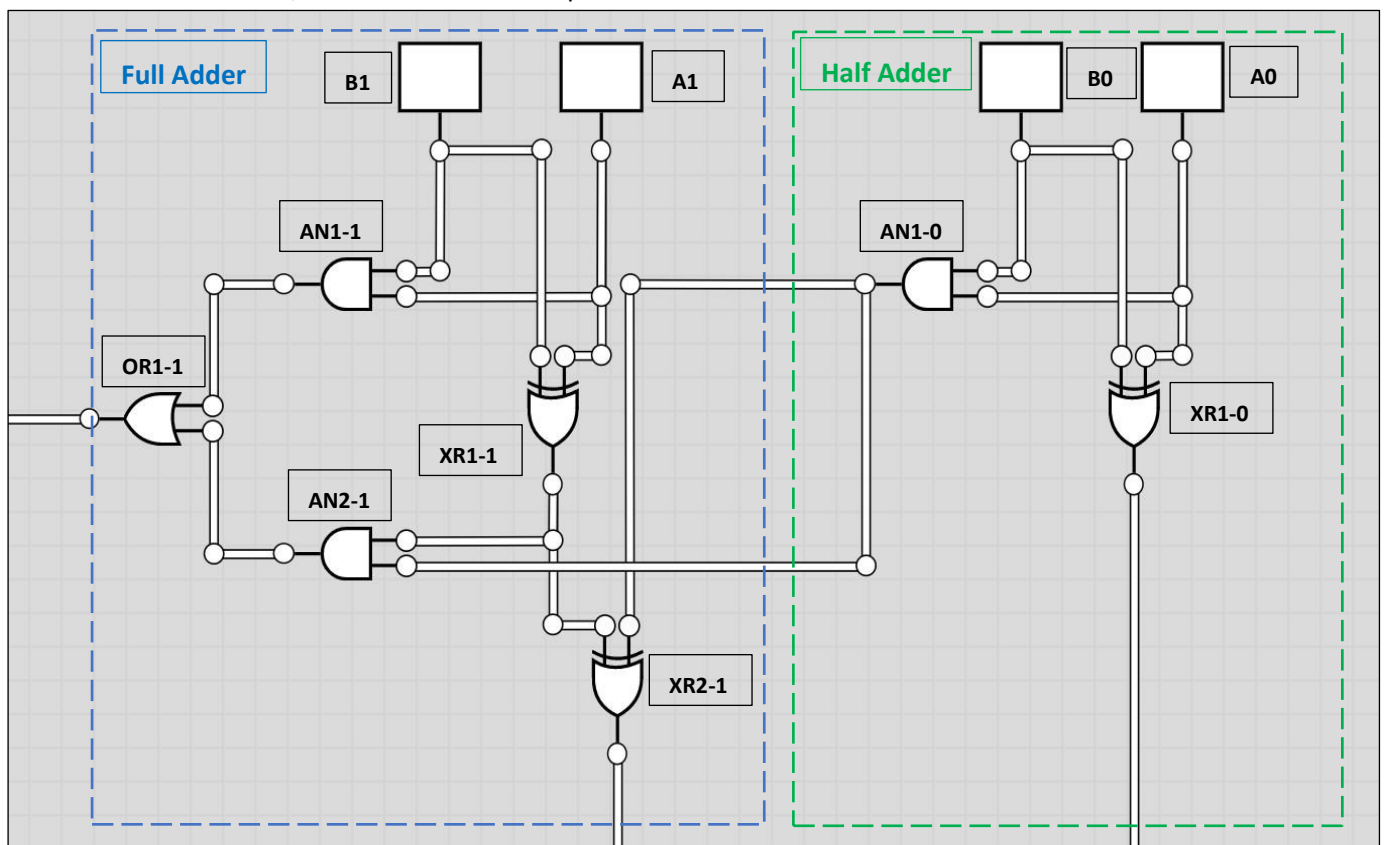


Figure 2. Full adder and Half adder

The logic gate components above are named following this rule: “<component prefix><order in current adder unit>-<order of the current adder unit in circuit>”. Input bits are an exception as they only need the prefix for the input and the position of the bit in their respective sequence.

The component prefixes are:

- AN – AND gate
- XR – XOR gate
- OR – OR gate
- A – first number
- B – second number

Below are the truth tables for the full and half bit adder in **Figure 2**.

A0	B0	A1-0	XR1-0
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Table 1. *Half adder truth table*

A1	B1	Carry in	XR1-1	XR2-1	AN1-1	AN2-1	OR1-1
0	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0
0	1	0	1	1	0	0	0
0	1	1	1	0	0	1	1
1	0	0	1	1	0	0	0
1	0	1	1	0	0	1	1
1	1	0	0	0	1	0	1
1	1	1	0	1	1	0	1

Table 2. *Full adder truth table*

In **Table 2**, “carry in” is the output from AN1-0 in the half adder. For the rest of the adders, carry in is the OR1 gate from the previous adder unit. Overflow flag is set by OR1-7 (the full adder unit in sequence).

What follows next is a scaled down version of the model in different key cases. It uses 3 full adders and 1 half adder (4 bits). It has a bulb attached to the carry out of the last full adder representing overflow and the sum bits are connected to a screen that displays the result of the sum. The sum is represented as a hexadecimal (base 16) digit because of the constraints of the circuit simulation software used. In place of the input bits (lower and upper) are switches that can be turned ON or OFF.

The reason for scaling down the circuit is to be able to fit an easy to view image of the whole circuit in action in this document. The behaviour is preserved as the carry out from one full adder is the carry in for the next full adder. The exception to this is the range of the values that can be input and output. While the 8 bit circuit has 256 possible values, the 4 bit circuit only has 16 possible values (0 to 15 for unsigned integers and -8 to 7 for signed integers).

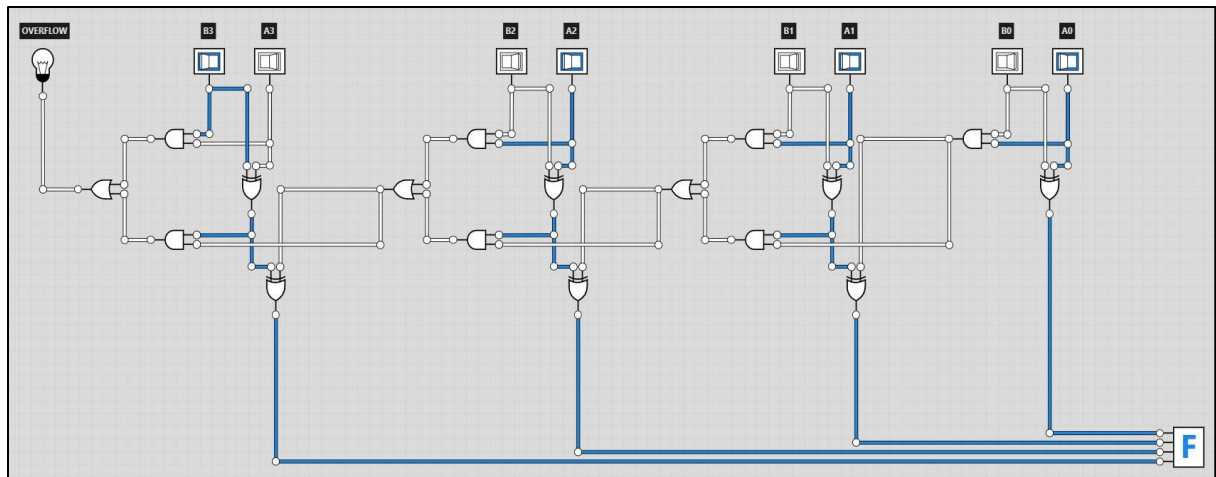


Figure 3a. Circuit adding 8 and 7

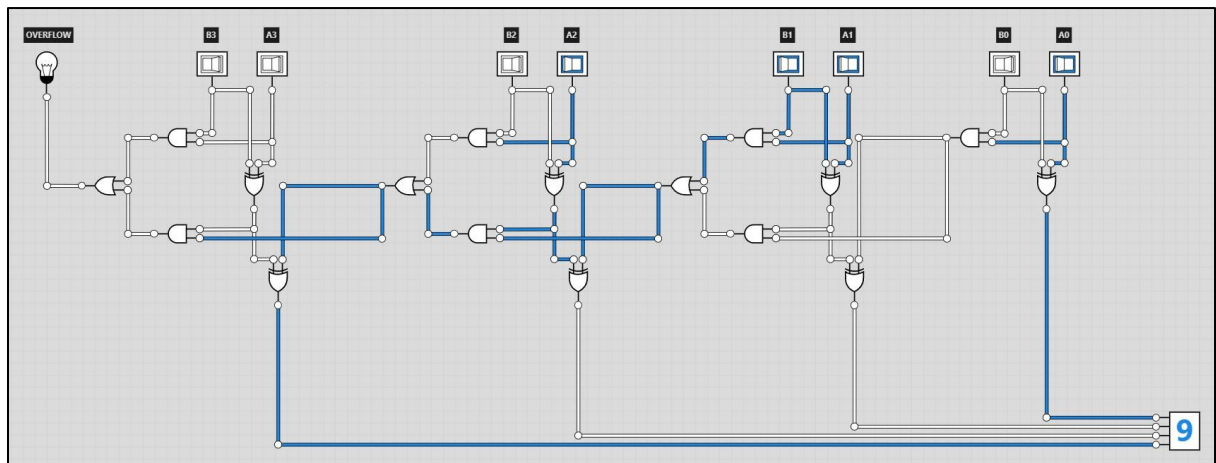


Figure 3b. Circuit adding 2 and 7

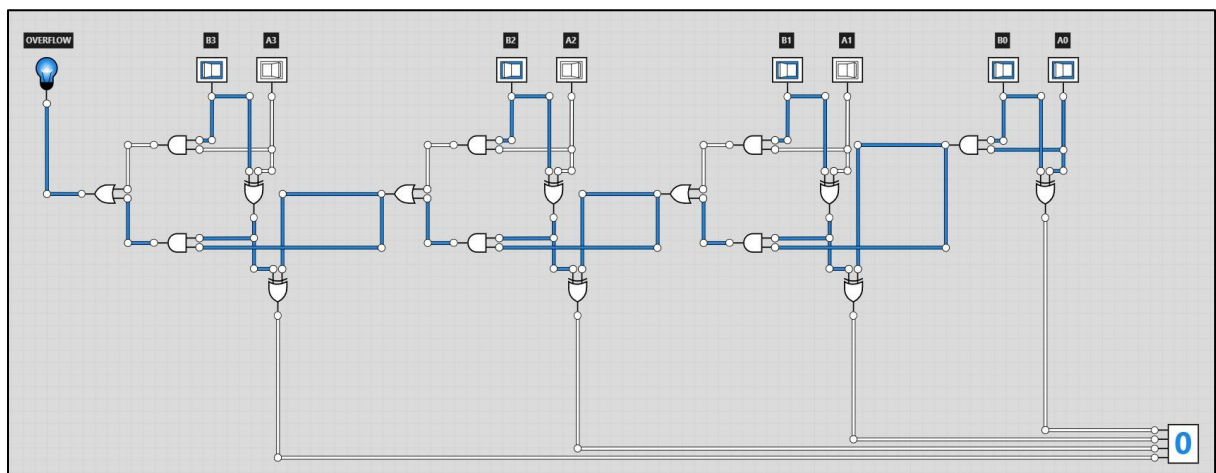


Figure 3c. Circuit adding 15 and 1 (overflow)

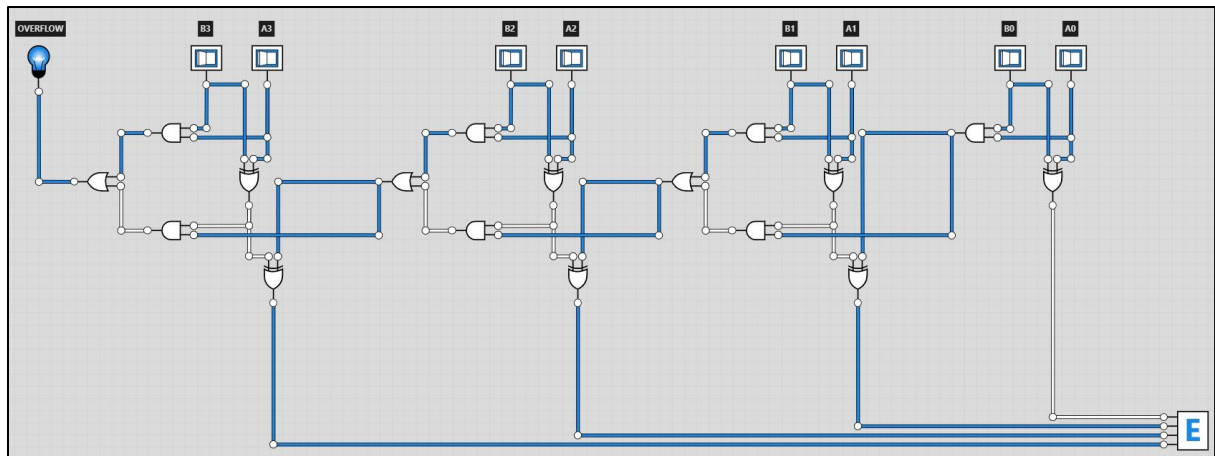


Figure 3d. Circuit adding 15 and 15 (overflow)

In the case presented in **Figure 6**, the result would be “1 1110” (30 in decimal) if the input numbers were to be added manually by a person. Because the scaled down circuit has 4 bits it can only use 4-bit numbers and represent sums that have only 4 bits so from “1 1110” only “1110” remains which corresponds to E (or 14 in decimal). That extra “1” will trigger the overflow flag, indicating the result is incorrect.

So far, the circuit works with no issues when it comes to unsigned integers. However, when introducing signed integers, the overflow value can become misleading. To fix this, the circuit was slightly changed so that 2 overflow flags are now present. Overflow flag C for unsigned addition and overflow flag V for signed addition. Only the relevant overflow flag will be taken into consideration according to their use case.

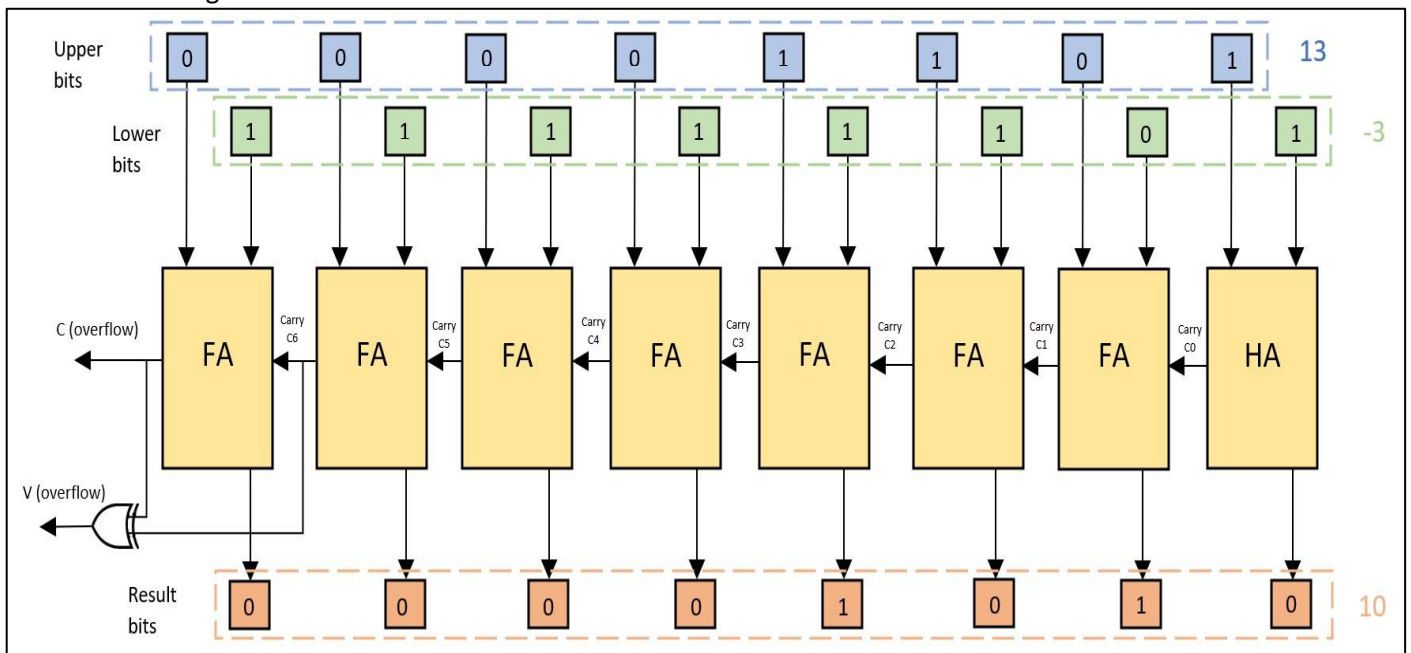


Figure 4. Complete bit adder circuit (the finished model)

As with the previous version of the circuit, below will be a few key cases of addition (or subtraction when adding a positive number to a negative one). The left-most, lowest lightbulb in the figures below, represents signed overflow used to determine overflow for signed integer addition. The remaining bulb represents unsigned addition and is not taken into consideration since the operations in the following examples are all signed integer operations.

It is important to note that the screen in this example is not capable of displaying negative numbers (simulation software limitation). However, if the signed values are mapped to their corresponding signed integer value it becomes evident that the result is correct.

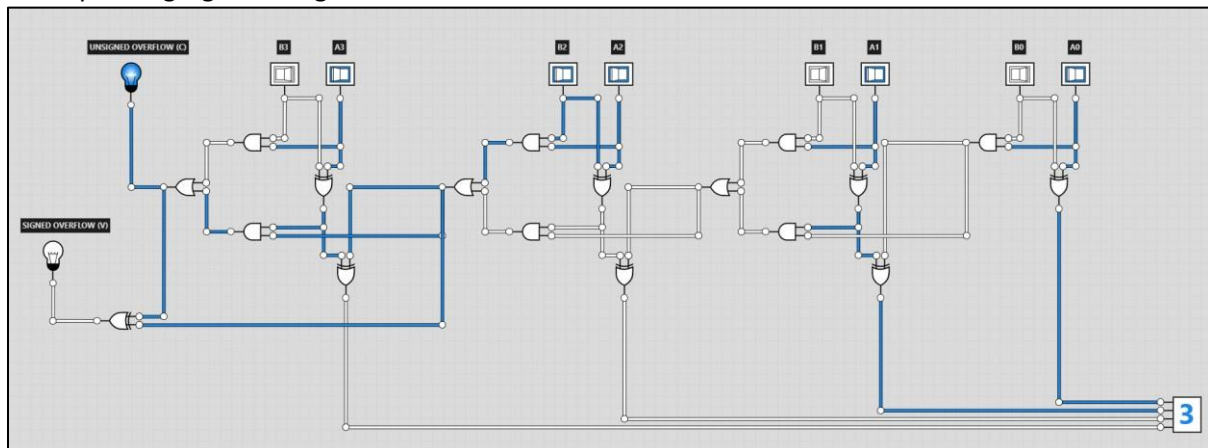


Figure 5a. Circuit adding 4 and -1

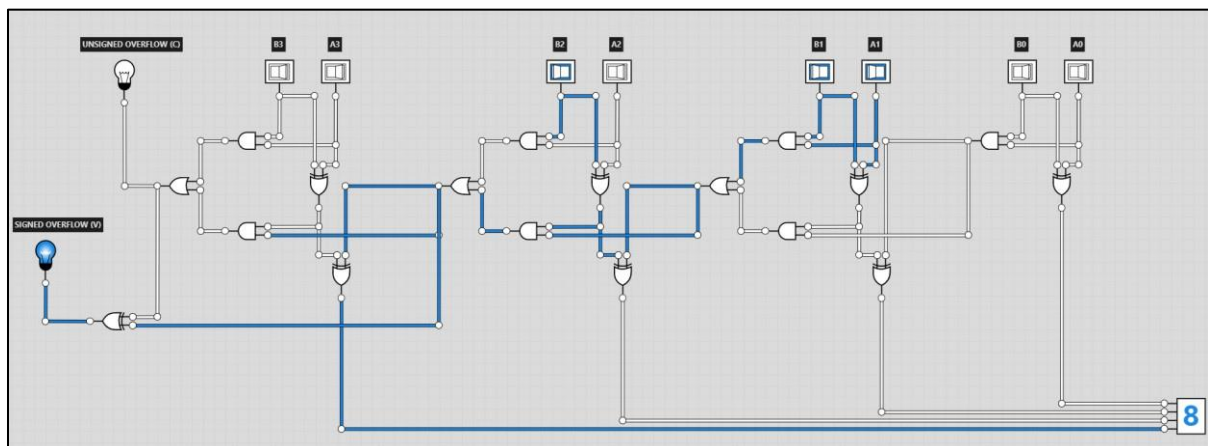


Figure 5b. Circuit adding 6 and 2 (overflow)

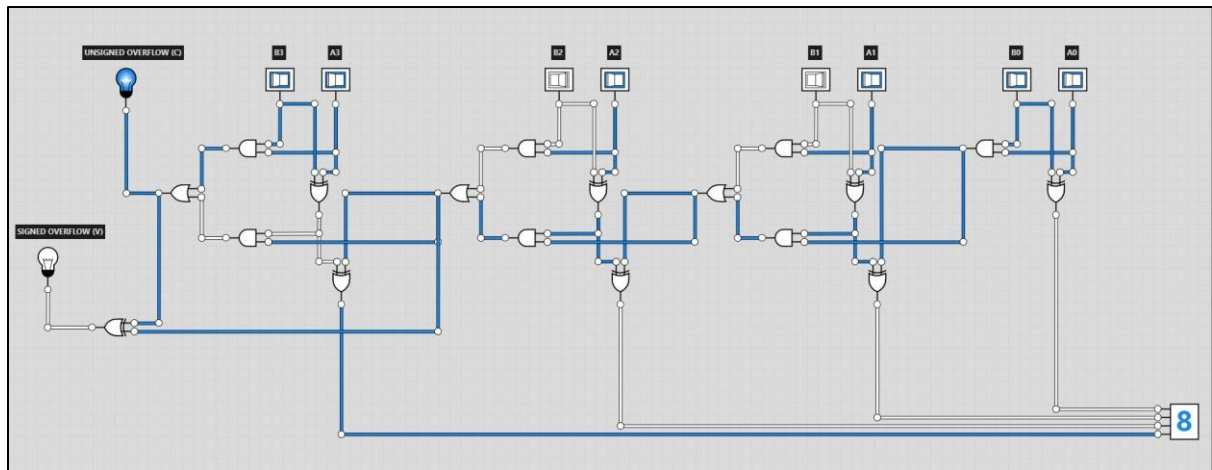


Figure 5c. Circuit adding -7 and -1

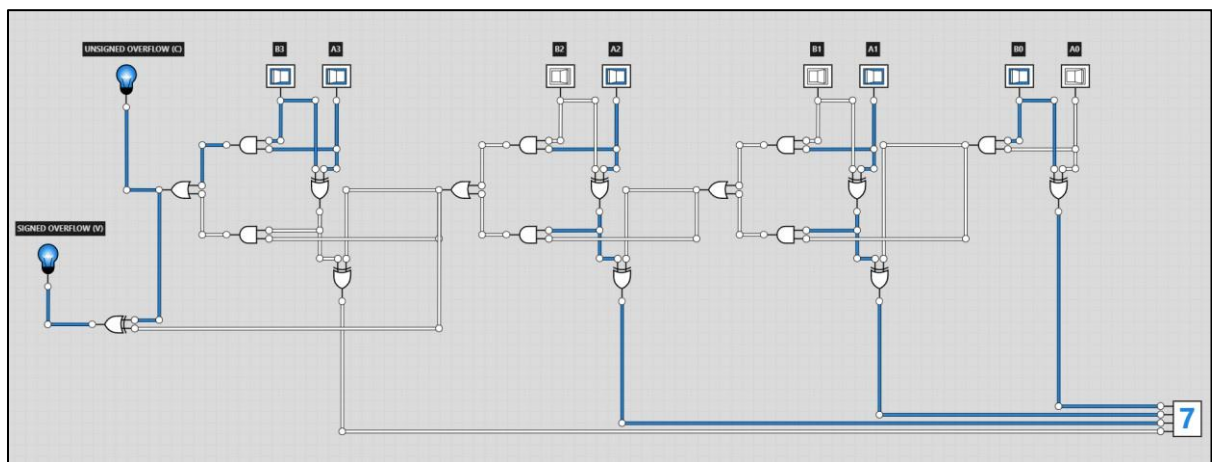


Figure 5d. Circuit adding -7 and -2 (overflow)

2. Choice of data structures

Choosing a data structure was crucial in making of the program. This is mainly because it changes fundamentally how the program is coded. For this program it was more advantageous to use regular python integers and bitwise operations rather than the more intuitive option of using arrays or lists.

At first, two versions of the program were coded. The first was done using arrays that held the bits of each input number. The second had an integer that corresponded to the value entered and relied on bitwise operations to produce a result. Both programs were written to use as few operations as possible. Using python's "timeit" module, the programs were measured in terms of how long it takes to add together all possible combinations of 8-bit numbers. They compute for both signed and unsigned numbers. The timing starts when the function that performs the addition begins and ends when the function finishes (this counts as 1 case). Timing it this way ensures that only the code that performs the addition is measured. This is repeated for all 65536 cases for both signed and then unsigned. For each type of addition, all the resulted case times are added together and averaged. The source code for this test can be found in **Appendix A** of this document.

The Python version used was 3.7.3 and measurements were made both through an online instance (server) and locally on a personal computer. The tests ran only the part of the code that does the addition meaning user interaction such as text prompts were not included (headless program). Below are tables presenting the data obtained.

ADDITION TYPE	BITWISE (TIME)	ARRAYS (TIME)
UNSIGNED	4.098147473996505e-05	5.035821959609166e-05
SIGNED	3.39989856001921e-05	3.302147160866298e-05

Table 3. *Server test of the headless programs*

ADDITION TYPE	BITWISE (TIME)	ARRAYS (TIME)
UNSIGNED	7.43103009881e-06	9.15525379241e-06
SIGNED	6.85119448463e-06	6.9275010901e-06

Table 4. *Local test of the headless programs*

In general, the arrays version of the program was slightly slower. In the case of an array, each element is stored at a different address and they need to be accessed first before being processed. Also, the arrays take more space in memory since each bit is stored as a different integer. Python integers (and primitives in general) have 24 bits. This number of bits per variable might be optimised (lowered) automatically by python's interpreter in the case of large arrays and lists.

Bit operations are much quicker for the processor since they are simpler instructions. In some cases a compiler (or in this situation the python interpreter) can automatically optimise some parts of the code to run faster if a combination of these operations are used.

3.Program structure and description

Modularising a program implies breaking the main problem that the program is concerned with into smaller problems that can be solved individually by different functions and procedures. By designing a program in a modular approach, it becomes much easier to read, maintain, improve on and sometimes old code can be reused to solve a completely new problem.

In order to understand the structure of the program better its structure is presented in the figure below.

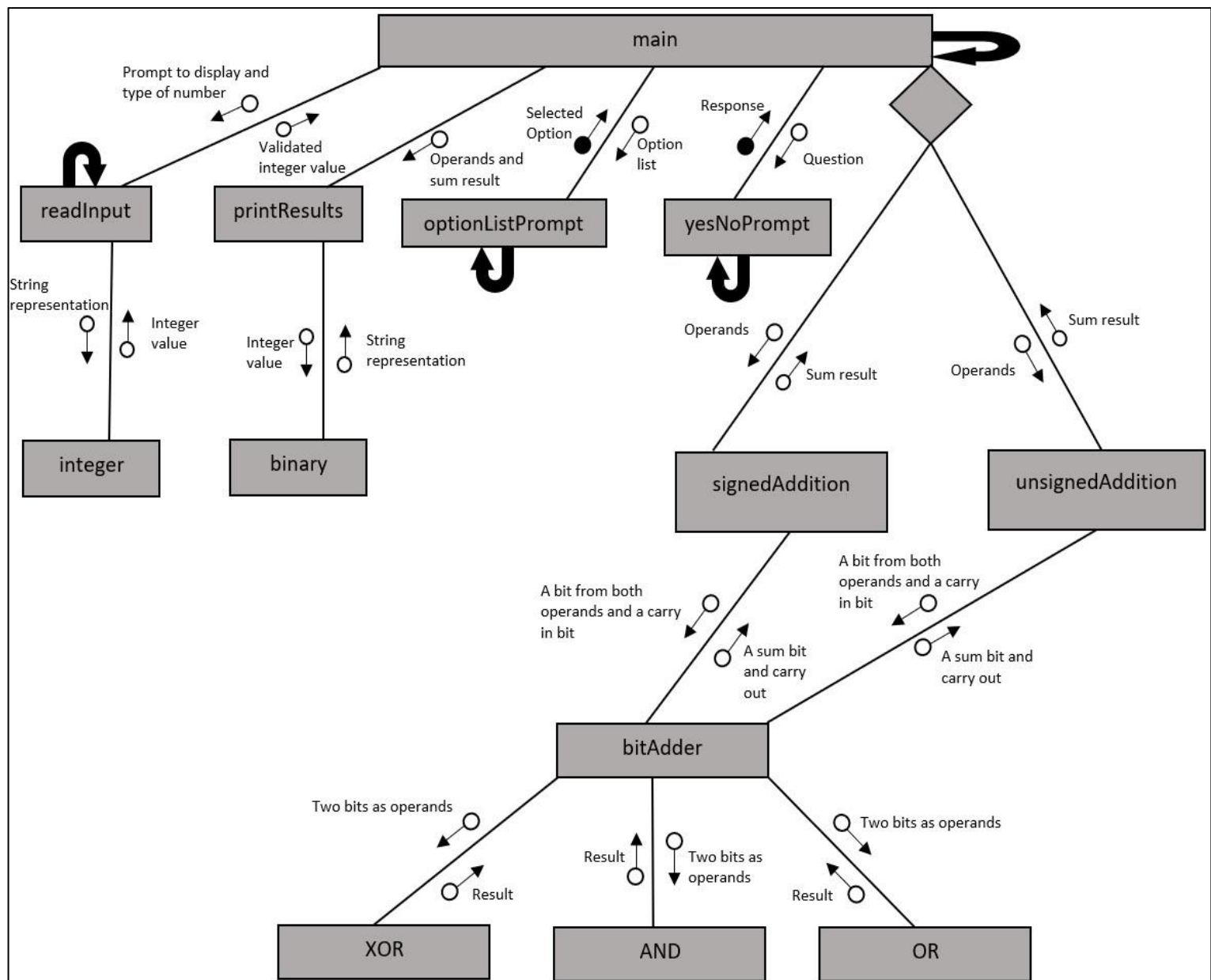


Figure 6. Structural chart of the program

When the program is run it enters the main function where it enters the main loop of the program. Here it begins by calling a function that displays a header text and below it a numbered list of options. The user can choose between performing a signed numbers addition or an unsigned numbers addition. The option is selected by entering the number of the option. If the input is not a valid option or something random then the program will tell the user to choose a valid option and displays the options again. The loop is infinite and only way to exit this function is to enter a correct option which will cause the function to return a value thus exiting the loop. Options are listed to the user starting from 1 but they are numbered starting from 0 inside the program.

After memorising the option selected from the option selected, two numbers A and B are read from the user. Before proceeding to reading the input, a text informing the user on how to enter numbers as a binary string is displayed along with an example. After, both A and B are set to the result of a the “readInput” function. This function takes in as parameters a text prompt to be displayed and a boolean representing whether the number is signed or unsigned. As implied earlier, the numbers can be input as a binary string as well. The function reads the number as a string and if the first character in the string is a “b” then the following characters are the binary representation of the number. If the number entered is not in binary representation then the number is simply converted to an integer using Python’s native function. Otherwise the string will be passed to the “integer” function. Here the string will have all white spaces removed from the string and its length checked to make sure that exactly 8 bits have been entered. A Python integer has 24 bits but the program uses only 8 bits therefore if a signed number is being processed the remaining 16 bits are set to match the sign bit of the 8-bit number. This is done to keep everything consistent with python interpreter as well.

The next step after reading both operands is to call either “unsignedAddition” or “signedAddition” functions with numbers A and B as parameters. Which one is called is based on the option selected by the user. The only similarity these two addition functions have is that they both use the “bitAdder” function. Adding the bits together works the same for both signed and unsigned numbers. The difference comes when interpreting overflow. In the case of unsigned addition, if the last carry flag is set to 1 it means there is overflow. The resulted number represents by how much the result exceeded 255. In the case of signed addition, overflow is determined by performing an exclusive or operation with the last carry flag (8th carry flag) and the 7th carry flag. In signed addition the correct sign is also enforced by setting the remaining bits of the Python integer to the most significant bit of the 8-bit number. When overflow happens, a text indicating so will be displayed on the screen. For signed addition, carry flags 8 and 7 are also displayed along with their actual values.

At the end of the addition function, they return an integer representing the sum of the input numbers. Together with the first and second number, the sum is passed to the “printResults” function. In here, 4 lines are printed to the screen. The first line contains a binary representation of the first number (as a string) which is obtained by passing the number to the “binary” function. Next to it is still the first number but in decimal form. It goes the same for the 2nd (second number) line and the 4th (the sum). The 3rd line is just a line drawn with minus characters.

The “binary” function mentioned in the previous paragraph takes in an integer and using bit operations it extracts each bit in turn and appends it to a string. The result is a representation of the first 8 bits of the integer passed. Because it starts with the least significant bit and end with the most significant bit, it must be reversed before returned for a correct representation. This function was implemented and used over Python’s native function to get a desired result much more easily.

Finally, the function “yesNoPrompt” is called with a prompt message that asks the user if they wish to continue. As input, the user must type either “Y” for YES or “N” for NO. This is case insensitive as the input is converted to uppercase and compared to uppercase values. If the user types “Y” the function will return “1” which will allow the program to keep looping. When the user types “N” the function returns “0” which will cause the program to exit the main loop and end. This function works in an infinite loop so if for any reason the input is not valid the question will be repeated until the answer is valid. Each time an invalid input is entered the user is prompted so.

4.Algorithm

With a clear picture of the program’s structure from the previous section, the algorithm’s functions will be presented bottom-up as presented in **Figure 6**. This way, it will be clear what each function when the main function’s steps are presented. Functions names were preserved in the pseudocode as they appear in the actual code.

Firstly, there are some atomic functions that are used in the “bitAdder” functions. These were implemented as follows:

```
FUNCTION AND(a AS INTEGER , b AS INTEGER )
    RETURN the result of 'and' bitwise operation between a and b

FUNCTION OR(a AS INTEGER , b AS INTEGER )
    RETURN the result of 'or' bitwise operation between a and b

FUNCTION XOR(a AS INTEGER , b AS INTEGER )
    RETURN the result of 'exclusive or' bitwise operation between a and b
```

Figure 7. Bit operations pseudocode

Each of the functions in **Figure 7** must be passed in a single bit. In turn the function returns the result of the operations for the input bits.

For the bit adder adders in the model, the function “bitAdder” was implemented to simulate each logic gate in turn. Since the half adder is similar to a full adder, the same function was used for it as well. In the case of the half adder the function “bitAdder” is always called with a carry in value of 0. This is because the half adder is the very first in the sequence and does not have a carry in.

```
FUNCTION bitAdder(Ax AS INTEGER , Bx AS INTEGER , carryIn AS INTEGER )
    SET xor1 TO FUNC XOR WITH PARAM <Ax>,<Bx>
    SET and1 TO FUNC AND WITH PARAM <Ax>,<Bx>
    SET and2 TO FUNC AND WITH PARAM <xor1>,<carryIn>

    SET bitSum TO FUNC XOR WITH PARAM <xor1>,<carryIn>
    SET carryOut TO FUNC OR WITH PARAM <and1>,<and2>

    RETURN bitSum,carryOut
```

Figure 8. Bit adder pseudocode

The bit adder function is used by 2 other functions: “unsignedAddition” and “signedAddition”. Each of these functions add the numbers the same way but handle overflow differently. Handling overflow correctly is important in telling whether the result is correct or incorrect (out of bit range).

```

FUNCTION unsignedAddition(A AS INTEGER , B AS INTEGER )
    SET S,carry, bit TO 0

    FOR i in closed range [0,7]
        SET bit,carry TO FUNC bitAdder WITH PARAM <bit on position i in A>,<bit on position i in B>
        Set the bit on position i in S to bit's value
    ENDFOR

    IF carry is not 0 THEN
        DISPLAY "Overflow"
    ENDIF

    RETURN S

```

Figure 9. Unsigned addition pseudocode

```

FUNCTION signedAddition(A AS INTEGER , B AS INTEGER )
    SET S,carry, bit, c6 TO 0

    FOR i in closed range [0,7]
        SET bit,carry TO FUNC bitAdder WITH PARAM <bit on position i in A> , <bit on position i in B>
        Set the bit on position i in S to bit's value

        IF i = 6 THEN
            SET c6 TO carry
        ENDIF
    ENDFOR

    Set the remaining bits of S to the correct sign according to the 8th bit of S

    SET overflow TO FUNC XOR WITH PARAM <carry>,<c6>

    IF overflow is not 0 THEN
        DISPLAY "Overflow!"
        DISPLAY "Carry bit 7: " + carry + " Carry bit 6: " + c6
    ENDIF

    RETURN S

```

Figure 10. Signed addition pseudocode

The addition type is determined by the selection that the user make from a list of numbered options. These are displayed by the “optionListPrompt” function. It also returns the option selected.

```
FUNCTION optionListPrompt(msg AS STRING , arrOptions AS ARRAY OF STRING )
    SET listSize TO length of arrOptions

    WHILE True
        DISPLAY msg

        FOR every option in arrOptions
            DISPLAY option number starting from 1 + option string from arrOptions
        ENDFOR

        SET response TO 0

        TRY
            READ response AS INTEGER
        EXCEPT
            DISPLAY "You must enter a number that corresponds to an option listed."
            SET response TO 0
        ENDTRY

        IF response > 0 and response <= listSize THEN
            RETURN response-1
        ENDIF
    ENDWHILE
```

Figure 11. List of options prompt pseudocode

The numbers that are passed to either signed or unsigned addition functions are read by the “readInput” function.

```

FUNCTION readInput(prompt AS STRING , signed AS BOOLEAN )
    WHILE True
        DISPLAY prompt
        READ inp AS STRING
        SET x = 0

        TRY
            IF first character of inp is "b"
                SET x TO FUNC integer WITH PARAM <inp without its first character>, <signed>
            ELSE
                SET x TO inp string representation converted to integer value
            ENDIF
        EXCEPT
            DISPLAY "Invalid input!"
            DISPLAY error message
            Skip the current iteration of the loop
        ENDTRY

        IF signed = True and x is in range [-128,127] THEN
            RETURN x
        ELSE-IF signed = False and x is in range [0,255] THEN
            RETURN x
        ELSE
            DISPLAY x+ " is not a valid number. (Number is out of "+ the corresponding range + " range)"
        ENDIF
    ENDWHILE

```

Figure 12. Read input function pseudocode

Numbers can be entered either in decimal form or as a binary string by adding the prefix “b” to the bit sequence. When numbers are entered as a bit sequence, the input is passed to a function called “integer” that converts it to an integer value that is appropriate for the context of the program.

```

FUNCTION integer(s AS STRING , signed AS BOOLEAN )
    Remove white spaces from s

    IF s does not contain exactly 8 characters THEN
        Raise an exception with a message about the format being invalid
    ENDIF

    SET x TO the integer value that corresponds to the binary representation of s

    IF signed = True and first character of s = "1" THEN
        Set all bits from outside the bit range of a byte to 1
    ENDIF

    RETURN x

```

Figure 13. Custom, string representation to integer value conversion

When printing the result the input numbers together with the result are passed to “printResult” procedure. This displays everything in a visually aesthetic way to the user.

```
PROCEDURE printResult(op1 AS INTEGER , op2 AS INTEGER , res AS INTEGER )
    DISPLAY empty line

    DISPLAY binary then decimal representation of op1 on the same line
    DISPLAY binary then decimal representation of op2 on the same line
    DISPLAY "-----"
    DISPLAY binary then decimal representation of res on the same line

    DISPLAY empty line
```

Figure 14. Pseudocode for printing the result

The binary representation of the numbers in “printResult” procedure is done by calling the “binary” function which is custom function. It’s different from Python’s “bin” function in the sense that it returns a string representing only 8 bits and gets rid of the “0b” prefix.

```
FUNCTION binary(x AS INTEGER )
    Extract first 8 bits from x by applying a bit mask of value 255

    SET s TO binary representation of x

    Fill in empty spaces in s with '0' to get an 8 characters bit string representation

    RETURN s
```

Figure 15. Custom, integer to string representation conversion

Whether the program keeps looping or not determined by the answer to a yes or no question that is prompted to the user. The pseudocode to handle this is as follows:

```
FUNCTION yesNoPrompt(msg AS STRING )
    WHILE True
        DISPLAY msg + "(Y/N) "

        READ response AS STRING

        Turn all letters in response to uppercase

        IF response is "Y" THEN
            RETURN True
        ELSE-IF response is "N" THEN
            RETURN False
        ELSE
            DISPLAY "Response invalid! Type 'Y' for YES or 'N' for NO."
        ENDIF
    ENDWHILE
```

Figure 16. Yes or No prompt pseudocode

Finally, the pseudocode for the main function of this program is as follows:

```

MAIN PROCEDURE main()
  SET A,B TO 0
  SET isSigned,stop TO False

  WHILE stop is False
    SET carry TO 0

    SET additionType TO FUNC optionPrompt WITH PARAM <option list header>, <options describing addition types>
    IF additionType is not 0 THEN
      SET isSigned TO False
    ELSE
      SET isSigned TO True
    ENDIF

    DISPLAY empty line

    DISPLAY "If you wish to enter a number in binary, add the prefix 'b' to it. Example 'b0000 0101' (5)"
    SET A TO FUNC readInput WITH PARAM <"A = ">, <isSigned>
    SET B TO FUNC readInput WITH PARAM <"B = ">, <isSigned>

    IF isSigned = True THEN
      SET S TO FUNC signedAddition WITH PARAM <A>,<B>
    ELSE
      SET S TO FUNC unsignedAddition WITH PARAM <A>,<B>
    ENDIF

    CALL printResult WITH PARAM <A>,<B>,<S>

    SET stop TO negation of FUNC yesNoPrompt WITH PARAM <"Do you wish to continue?">

    SET S TO 0
  ENDWHILE

```

Figure 17. Pseudocode of the main function

5. Testing

5.1. Result value check

Testing of the addition was done on all possible values to check if all of the results are correct. The testing was divided into 2 cases, signed addition and unsigned addition.

In the case of unsigned addition, when the value exceeds 255 the result loops back to 0. For example, if 250 and 10 are added the result is 4. The mathematical formula for this is $X = (A + B) \bmod 256$, where A and B are integers from the closed range [0,255]. Below is the Python code for this test case.

```

#Test for unsigned addition
def run_test_case1():
    failed = 0

    for A in range(0,256):
        for B in range(0,256):
            S = unsignedAddition(A,B)
            check = (A+B)%256

            if(S != check):
                failed = failed + 1
                print("Case failed!\nA: %d\nB: %d\nResult got: %d, correct result %d\n\n" % (A,B,S,check))

    if(not failed):
        print("All test cases passed.")
    else:
        print("%d cases failed" % failed)

```

Figure 18. Test case for unsigned addition

Testing for signed addition is a somewhat harder because the values loop around -128 and 127. In practice, this means that if -128 and -1 are added the result will give 127. For 127 and 1 it will give -128. To achieve the same result mathematically, the closed range [-128,127] can be mapped to the closed range [0,255].

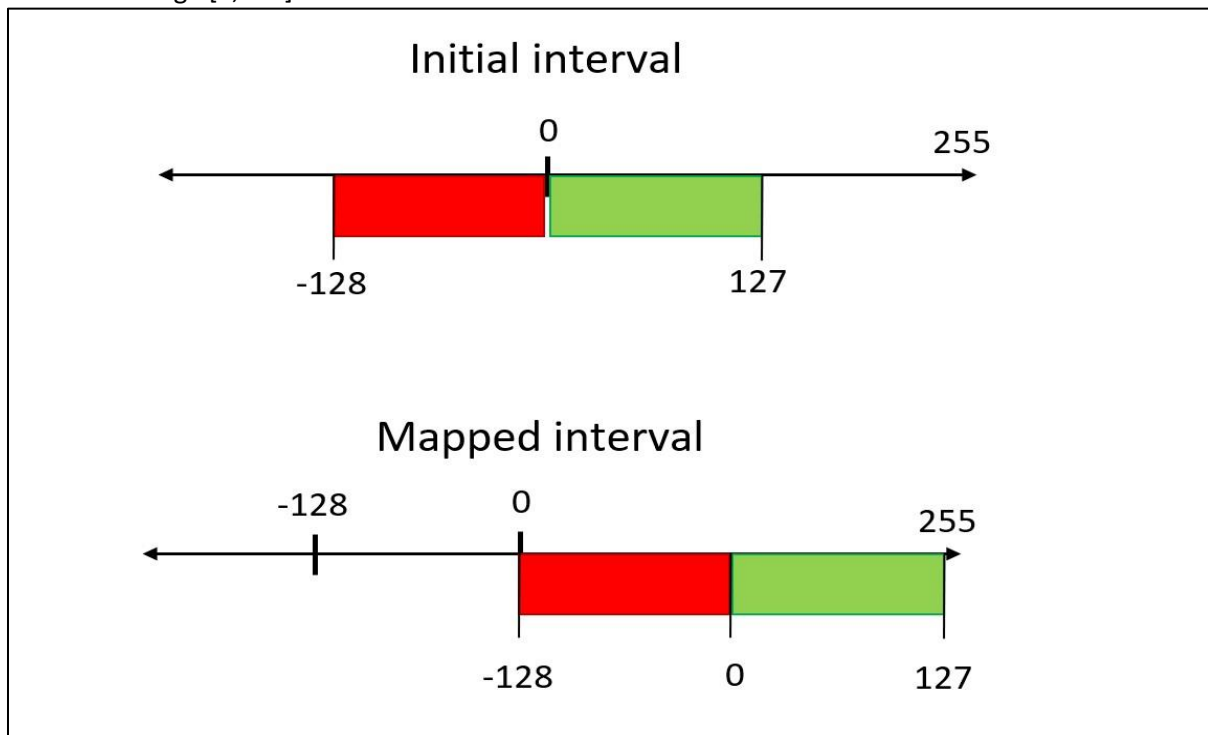


Figure 19. Initial and mapped interval of signed values

With this, the modulus operator can be used to keep the values in range. After the modulus operator is used, the values must be mapped back to their correct range which is $[-128, 127]$. In this way, the correct results will be obtained. The mathematical formula for this is $X = ((A+B+128) \bmod 256) - 128$.

```
#Test for signed addition
def run_test_case2():
    print("SIGNED ADDITION TEST")
    failed = 0
    check_range = range(-128,128)

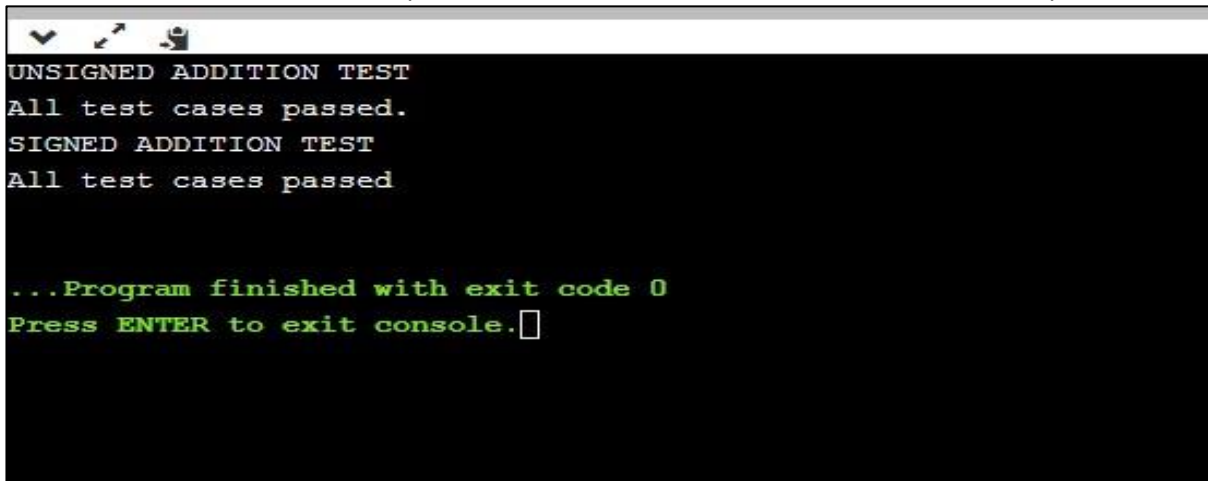
    for A in check_range:
        for B in check_range:
            S = signedAddition(A,B)
            X = ((A+B+128)%256)-128

            if(S != X):
                failed = failed + 1
                print("Case failed!\nA: %d\nB: %d\nResult got: %d, correct result %d\n\n" % (A,B,S,X))

    if(not failed):
        print("All test cases passed")
    else:
        print("Failed %d cases." % failed)
```

Figure 19. Test case for signed addition

Both tests were run on an online compiler and there were no failed cases for both. Aside from the test functions, all other print statements were commented out for clearer output.



```
UNSIGNED ADDITION TEST
All test cases passed.
SIGNED ADDITION TEST
All test cases passed

...Program finished with exit code 0
Press ENTER to exit console.
```

Figure 20. Results of test 1 and 2

5.2. Overflow flag test

With the values of the result checked in all cases only the correct triggering of overflow remains. For this the program was run for a few key values for signed addition then unsigned addition.

```
===== RESTART: C:\Users\Alex\Desktop\computer fundamentals CWK1.py =====  
  
Overflow!  
Carry bit 7: 1  Carry bit 6: 0  
  
10000000+      (-128)  
11111111      (-1)  
-----  
01111111      (127)  
  
Overflow!  
Carry bit 7: 0  Carry bit 6: 1  
  
01111111+      (127)  
00000001      (1)  
-----  
10000000      (-128)  
  
00011110+      (30)  
11011000      (-40)  
-----  
11110110      (-10)  
  
01010000+      (80)  
11001110      (-50)  
-----  
00011110      (30)  
  
11101100+      (-20)  
10111111      (-65)  
-----  
10101011      (-85)  
  
>>> |
```

Figure 21. Signed addition overflow test

As shown in **Figure 21**, the overflow flag for signed addition only triggers when the value is out of range but not when the last carry flag value is 1. Overflow for signed addition triggers correctly.

```
===== RESTART: C:\Users\Alex\Desktop\computer fundamentals CWK1.py =====  
  
Overflow  
  
11111010+      (250)  
00001010      (10)  
-----  
00000100      (4)  
  
01100100+      (100)  
00011011      (27)  
-----  
01111111      (127)  
  
Overflow  
  
11011100+      (220)  
11001000      (200)  
-----  
10100100      (164)  
  
>>>
```

Figure 22. Unsigned addition overflow test

As expected, the overflow triggers when the result value exceeds 255 which is equivalent to having the last carry flag set to 1 for unsigned addition. In **Figure 22**, the word “Overflow” is displayed before the actual operation is displayed so only the first and last case triggered the overflow.

5.3. Invalid input

Invalid input is subject to 3 cases: the input is of the wrong type (letters instead of digits), out of accepted range of values, invalid format (in the case of inputting a binary string). Except for the accepted range case, the invalid input shares the same behaviour for both signed and unsigned addition.

```
===== RESTART: C:\Users\Alex\Desktop\computer fundamentals CWK1.py =====
Select one of the options below
(1)Add 2 signed numbers    (positive and negative numbers)
(2)Add 2 unsigned numbers  (positive numbers only)
one
You must enter the number that corresponds to the option you wish to select.
Select one of the options below
(1)Add 2 signed numbers    (positive and negative numbers)
(2)Add 2 unsigned numbers  (positive numbers only)
1

If you wish to enter a number in binary, add the prefix 'b' to it. Example 'b0000 0101' (5)
A = qwerty!

Invalid input!
invalid literal for int() with base 10: 'qwerty!'

A = 97
B = !@£

Invalid input!
invalid literal for int() with base 10: '!@\xa3'

B = 3

01100001+      (97)
00000011      (3)
-----
01100100      (100)

Do you wish to continue?(Y/N)
NoOOo
Response invalid! Type 'Y' for YES or 'N' for NO.
Do you wish to continue?(Y/N)
1
Response invalid! Type 'Y' for YES or 'N' for NO.
Do you wish to continue?(Y/N)
n
>>>
```

Figure 23. Wrong input type test

When the user enters an input of the wrong type, the user is prompted so as expected.

An input can be in an invalid format in the case of inputting the numbers as a binary string. The function that converts the string representation to integer value expects a string that contains 8 of either '1' or '0' characters. The exception being white spaces that are trimmed out of the string before the conversion begins. White spaces are allowed so that the user can input a binary number in an easy to read format.

```

===== RESTART: C:\Users\Alex\Desktop\computer fundamentals CWK1.py =====
Select one of the options below
(1)Add 2 signed numbers (positive and negative numbers)
(2)Add 2 unsigned numbers (positive numbers only)
1

If you wish to enter a number in binary, add the prefix 'b' to it. Example 'b0000 0101' (5)
A = b
Invalid input!
Invalid format! Expected exactly 8 bits, got 0

A = b0000 00!0
Invalid input!
invalid literal for int() with base 2: '000000!0'

A = b0000 1010
B = 0000 0001

Invalid input!
invalid literal for int() with base 10: '0000 0001'

B = 00000001

00001010+      (10)
00000001      (1)
-----
00001011      (11)

Do you wish to continue?(Y/N)
n
>>> |

```

Figure 24. Invalid input format test

Without the “b” prefix, the input will be converted to integer value in base 10.

The input value is checked if it falls in the correct range depending on the type of operation selected. For signed numbers this range is [-128,127] and for unsigned numbers it is [0,255].

```

===== RESTART: C:\Users\Alex\Desktop\computer fundamentals CWK1.py =====
Select one of the options below
(1)Add 2 signed numbers (positive and negative numbers)
(2)Add 2 unsigned numbers (positive numbers only)
1

If you wish to enter a number in binary, add the prefix 'b' to it. Example 'b0000 0101' (5)
A = -200
-200 is not a valid number.(Number is out of [-128,127] range)
A = 5000
5000 is not a valid number.(Number is out of [-128,127] range)
A = 20
B = 44

00010100+      (20)
00101100      (44)
-----
01000000      (64)

Do you wish to continue?(Y/N)
n
>>> |

```

Figure 25. Signed value out of range test

```

===== RESTART: C:\Users\Alex\Desktop\computer fundamentals CWK1.py =====
Select one of the options below
(1)Add 2 signed numbers    (positive and negative numbers)
(2)Add 2 unsigned numbers  (positive numbers only)
2

If you wish to enter a number in binary, add the prefix 'b' to it. Example 'b0000 0101' (5)
A = -10
-10    is not a valid number. (Number is out of [0,255] range)
A = 2048
2048   is not a valid number. (Number is out of [0,255] range)
A = 200
B = 5

11001000+      (200)
00000101       (5)
-----
11001101       (205)

Do you wish to continue?(Y/N)
n
>>> |

```

Figure 26. *Unsigned value range test*

6.Reflection

Step by step I've managed to implement a program that simulates hardware unsigned integer addition. It got slightly complicated when signed integers and signed addition were introduced because of how the values are mapped and how the addition behaves. For example, adding a positive number to a negative one is in fact the positive number subtracted by the other one. This is easy to deduce when working with regular decimal numbers as a person, but for computers in binary it is immediately evident.

Because negative numbers were written in "two's complement" it was more difficult to check results. Noticing how the values wrap around their respective range when the results exceed the range for both in case of signed and unsigned addition helped a lot in writing test cases.

Writing code was not only about translating the model into code but also analysing performance and testing so that the program runs optimally and as intended. It was tedious but, in the end, very satisfying.

The topic seemed simple on the surface but the further I progressed I realised much more there is to learn beyond what I've already managed to implement at any given point in the project.

In conclusion, this project gave me deep insight into how computer add numbers and because of it I appreciate much more the complexity of computer architectures. I understand better how the bit adder circuit works and gained new skills in developing robust and efficient code.

References

1. GDB Online (2016 - 2021) "Online Python Compiler – online editor". Available at: https://www.onlinegdb.com/online_python_compiler (Accessed: 11 May 2021)
2. EnggClasses (2013) "Unsigned and Signed Binary Addition". Available at: https://www.youtube.com/watch?v=pr5gsJYOs_o&ab_channel=EnggClasses (Accessed: 14 March 2021)
3. Wolfram System Modeler, 8 bit adder diagram (no name). Available at: <https://www.wolfram.com/system-modeler/examples/more/electrical-engineering/8-bitadder> (Accessed: 19 May 2021)
4. Google Images "Four-Bit Adder-Subtractor". Available at: https://www.google.com/search?q=bit+adder&source=lnms&tbn=isch&sa=X&ved=2ahUKEwiGnJjVzd_wAhWPDxQKHYOMaisQ_AUoAXoECAEQAw&biw=1920&bih=969&dpr=1#imgsrc=fHgE3ldbDQAJ_M (Accessed 20 May 2021)

Appendices

Appendix A: Timed run source code

Requires Python's "timeit" module to be imported.

```
def timedRun():
    segmentStart = []
    segmentEnd = []

    #unsigned addition
    for a in range(0,256):
    for b in range(0,256):
        start = time.time()
        unsignedAddition(a,b)
        end = time.time()

        segmentStart.append(start)
        segmentEnd.append(end)

    casesSize =len(segmentStart)
    result = 0
    for i in range(0,casesSize):
        result = result + (segmentEnd[i] - segmentStart[i])

    result = float(result)/float(casesSize)

    print("UNSIGNED ADDITION")
    print("%d cases"% casesSize)
    print("Time: "+str(result))

    del segmentStart[:]
    del segmentEnd[:]

    #signed addition
    for a in range(-128,128):
    for b in range(-128,128):
        start = time.time()
        signedAddition(a,b)
        end = time.time()

        segmentStart.append(start)
        segmentEnd.append(end)

    casesSize =
len(segmentStart)    result = 0
    for i in range(0,casesSize):
        result = result + (segmentEnd[i] - segmentStart[i])

    result = float(result)/float(casesSize)

    print("\nSIGNED ADDITION")
    print("%d cases"% casesSize)
    print("Time: "+str(result))
```

Appendix B: Unsigned addition value test source code

```
#Test for unsigned addition
def run_test_case1():
    print("UNSIGNED ADDITION TEST")
    failed = 0

    for A in range(0,256):
        for B in range(0,256):
            S = unsignedAddition(A,B)
            check = (A+B)%256

            if(S != check):
                failed = failed + 1
                print("Case failed!\nA: %d\nB: %d\nResult got: %d, correct result %d\n\n" % (A,B,S,check))

    if(not failed):
        print("All test cases passed.")
    else:
        print("%d cases failed" % failed)
```

Appendix C: Signed addition value test source code

```
#Test for signed addition
def run_test_case2():
    print("SIGNED ADDITION TEST")
    failed = 0
    check_range = range(-128,128)

    for A in check_range:
        for B in check_range:
            S = signedAddition(A,B)
            X = ((A+B+128)%256)-128
            if(S != X):
                failed = failed + 1
                print("Case failed!\nA: %d\nB: %d\nResult got: %d, correct result %d\n\n" % (A,B,S,X))

    if(not failed):
        print("All test cases passed")
    else:
        print("Failed %d cases." % failed)
```