Pratical 5
Calculator (Part 2)

Approximate duration: 4 hrs.

For this practical, the **`GetInput`** subroutine is put at your disposal. It allows you to get a string of characters keyed in by a user. **`GetInput`** has the following inputs:

Inputs : **A0.L** points to a 60-byte buffer where the user string will be stored.
D1.B holds the column number where the user string will be displayed.
D2.B holds the line number where the user string will be displayed.
D3.L holds the time delay index before the first repetition.
D4.L holds the time delay index after the first repetition.

The buffer should be reserved by the `DS.B` assembly directive.

- Type the source code below and save it under the name "`GetInputTest.asm`".
- Copy the "`GetInput.bin`" file into the same folder.

```
                    ; ==============================
                    ; Vector Initialization
                    ; ==============================

                    org     $0

vector_000          dc.l    $ffb500
vector_001          dc.l    Main

                    ; ==============================
                    ; Main Program
                    ; ==============================

                    org     $500

Main                movea.l #sBuffer,a0
                    clr.b   d1
                    clr.b   d2
                    move.l  #60000,d3
                    move.l  #8000,d4
                    jsr     GetInput

                    illegal

                    ; ==============================
                    ; Subroutines
                    ; ==============================

GetInput            incbin  "GetInput.bin"

                    ; ==============================
                    ; Data
                    ; ==============================

sBuffer             ds.b    60
```

- Run this code and display the video output window. Enter a string of characters and press the **[Enter]** key. Have a look at the contents of memory location `buffer`. Repeat the process until you grasp how the **GetInput** subroutine works. Be careful, you are not asked to execute **GetInput** step by step; you just have to understand how to use it.

Note:
The **D3** and **D4** parameters should be adjusted according to the performance of your computer. If the repetition is too fast, you should increase these values.

Step 1

Write the **NextOp** subroutine that returns the memory location of either the first operator in a string or the null character if no operators are found. The string can contain any types of characters (letters, punctuation, digits, operators, etc.).

Input : **A0.L** points to a string.

Output : **A0.L** returns the address of the first operator in the given string or the address of the null character if no operators are found.

Ex.: Before:

	A0 ↓									
	'1'	'0'	'4'	'+'	'9'	'*'	'2'	'-'	'3'	0

After:

				A0 ↓						
	'1'	'0'	'4'	'+'	'9'	'*'	'2'	'-'	'3'	0

Step 2

Write down the **GetNum** subroutine that determines the integer value of a number in a string with error handling.

Input : **A0.L** points to a string.

Outputs : **Z** returns false (0) if a conversion error occurs.
Z returns true (1) if the conversion is right.
If **Z** returns false, then **D0.L** and **A0.L** do not change.
If **Z** returns true, then:

- **D0.L** returns the integer value of the number.
- **A0.L** returns the address of the character that follows the converted number.

Ex.: Before:

	A0 ↓									
	'1'	'0'	'4'	'+'	'9'	'*'	'2'	'-'	'3'	0

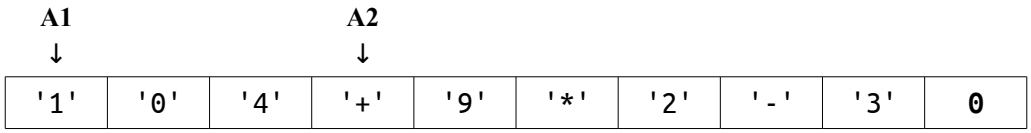
After:

				A0 ↓						
	'1'	'0'	'4'	'+'	'9'	'*'	'2'	'-'	'3'	0

With **Z** = **1** and **D0** = **104**

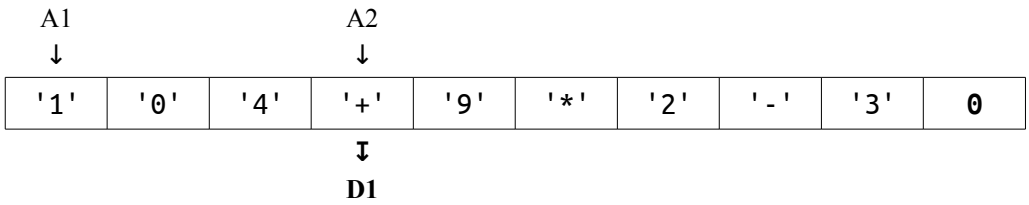
Tips:

- Use **A1** and **A2** to flank a number:

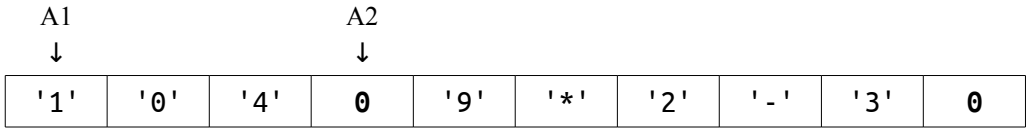


  Each number is separated by an operator (except for the last one, which is followed by the null character). Use the **`NextOp`** subroutine, which returns the memory location of either the next operator or the null character, to flank the number.

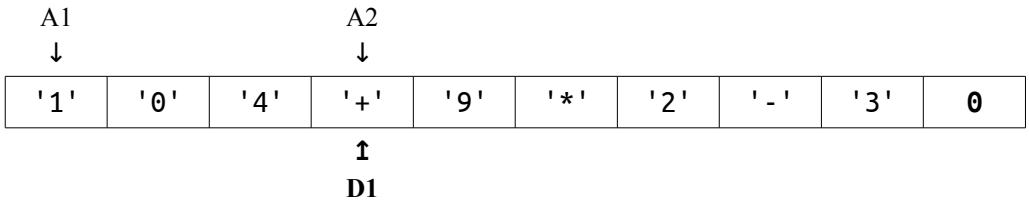
- Use **D1** to store the character that follows the number:



- Isolate the number from the string by replacing the character that follows it by the null character:



- Use the **`Convert`** subroutine to convert the isolated number.
- Replace the character saved in **D1** in its initial memory location.



- Be careful about output values and error cases.

Step 3

Write the **`GetExpr`** subroutine that calculates the integer value of an expression in a string of characters with error handling.

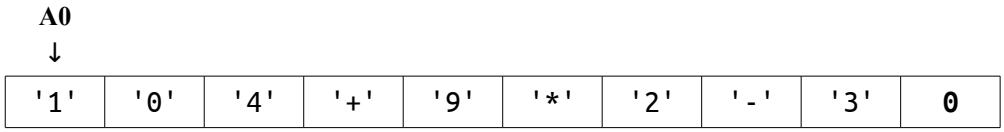
Input : **A0.L** points to a string of characters.

Outputs : **Z** returns false (0) if a conversion error occurs.
Z returns true (1) if the conversion is right.
If **Z** returns false (0), then **D0.L** is lost.
If **Z** returns true (1), then **D0.L** returns the integer value of the expression.

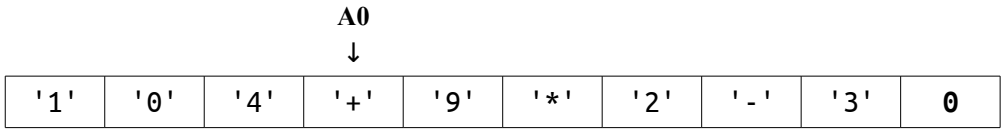
Tips:

- You should use the registers as follows:
  - ➔ **A0** should be used to move along the string.
  - ➔ **D0** should be used to convert each number of the string.
  - ➔ **D1** should be used to accumulate the final result.
  - ➔ **D2** should be used to hold the operator (or the null character).

- For instance, let us consider the following string:



- Use **`GetNum`** to calculate the integer value of the first number:



  - ➔ **A0** will point to the next operator.
  - ➔ **D0** will contain the integer value `104`.
  - ➔ **Z** will be `1`.

- Return false if a conversion error occurs.

- Initialize **D1** (final result) to the value of the first number.

- You should now use a loop that performs as follows:
  - ➔ Copy the operator (or the null character) into **D2**.
  - ➔ Make **A0** point to the next number.
  - ➔ If **D2** is null, return true (no error).
  - ➔ Otherwise, convert the next number (and return false if an error occurs).
  - ➔ According to the operator held in **D2**, perform the operation. That is to say, subtract, multiply or divide the contents of **D0** and **D1**.

- Do not forget the division by zero. This case should return false.

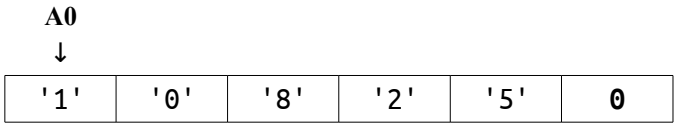
- Pay careful attention to the division instruction (see datasheet) that returns the quotient in the lower word and the remainder in the upper word of the destination operand. Therefore, a sign extension may be useful.

Step 4

Write the **`Uitoa`** subroutine that converts a 16-bit unsigned integer into a string of characters.

Inputs : **A0.L** points to a buffer where the string will be stored after conversion.
D0.W holds the 16-bit unsigned value to convert.

For instance, if **D0.W** = `10825`, the following string should be placed at the address held in **A0** :



Tips:

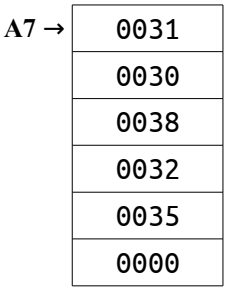
- Divide successively the number to convert by 10 and collect the remainders in order to convert them into characters.

  For instance, if **D0.W** = `10825`, the following five divisions should be performed:

Division	Quotient	Remainder
10825/10	1082	5
1082/10	108	2
108/10	10	8
10/10	1	0
1/10	0	1

Just collect each remainder and convert it into a character (use the ASCII code).

- But the problem we are faced with is that the first remainder matches the last character of the string. In fact, the remainders are in the opposite order as expected.
- A remainder converted into a character cannot be directly moved into the string. First, you should push it onto the stack.
- The first character to push will be the null character (do it before the first division).
- Then, push the `'5'`, `'2'`, `'8'`, `'0'` and `'1'` characters.
- When the quotient is null, stop the division.
- Here is what the stack should look like (the hexadecimal representation is used):



Note:
A byte cannot be pushed onto the stack. Only words wider than 16 bits can be.

Reminder: `'0'` = `$30`

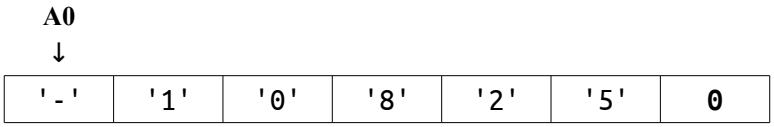
- Now, just pop all the characters off the stack (up to the null character) and move them into the string.
- Be careful, you cannot copy a character directly from the stack to the string. A character is 16 bits wide in the stack and 8 bits wide in the string. In other words, the source is 16 bits wide and the destination 8 bits wide. Therefore, you should move a character from the stack to the string through a register. That is to say, copy the character from the stack to a register (16-bit operation), then copy the character from the register to the string (8-bit operation).
- Pay careful attention to the division instruction (see datasheet) that returns the quotient in the lower word and the remainder in the upper word of the destination operand. You may use the `SWAP` instruction (see datasheet) that exchanges the 16-bit words of a data register.
- Pay also careful attention to the size of the operands of the `DIVU` instruction. In our case, only the 16 LSBs of the dividend (destination operand) are relevant. But the `DIVU` instruction takes the 32 bits into account. Therefore you should use a mask that sets the 16 MSBs of the dividend to zero just before performing the division.

Step 5

Write the **`Itoa`** subroutine that converts a 16-bit signed integer into a string of characters.

Inputs : **A0.L** points to a buffer where the string will be stored after conversion.
D0.W holds the 16-bit signed value to convert.

For instance, if **D0.W** = `-10825`, the following string should be placed at the address held in **A0** :



Tips:

- If the number is positive, just call **`Uitoa`**.
- If the number is negative, put the `'-'` character in the string and call **`Uitoa`** with the additive inverse of the number.

Step 6

Write the **`Main`** program of the calculator that complies with the following example:

```
Enter an expression:
50*4+2
Result:
202
```