Assembly calculator with UI

David Strikaitis   
*department of Computing*  
*South East Technological University Carlow*Carlow, Ireland  
C00283152@setu.ie

Veronika Syncakova  
*department of Computing*  
*South East Technological University Carlow*Carlow, Ireland  
C00288340@setu.ie

*Abstract*—Assembly calculator with user interface is simple but very useful project. It uses the fundamentals of logical mathematical operations that the computer does anyways. The added value of the user interface increases it’s complexity. It requires mouse IRQ, collision checks and displaying graphics. Even though the layout is on the simpler side, it provides a real-time feedback to the user by showing his selected option.

Keywords—Motorola 68000, user interface, calculations, subroutine, simple

The main idea of this project is to create a simple calculator. The added value is the user-friendly simple interface. Before the popularization of smartphones, we were oftentimes told that we would not have a calculator on us in all situations. However, this statement is mostly untrue nowadays as we carry a portable computer on us almost all the time. Certainly, a lot of downsides come with this fact and we should not depend only on these devices in our pockets, but they are very good helpers. One example is that we have the already mentioned calculator on us wherever since most smartphones come with pre-installed calculator apps as it is very useful tool. This is a reason why we chose this as our project idea - it’s functionality and usability in everyday lives.

# Description

## Motorola 68000

Our calculator is designed for the Motorola 68000 microprocessor. This processor was introduced in 1979 and has a 16-bit internal data bus and 32-bit instruction set. Since it provides enough data register memory for our simple calculator, we are able to store almost all of our data in the memory buffer registers for fast execution.

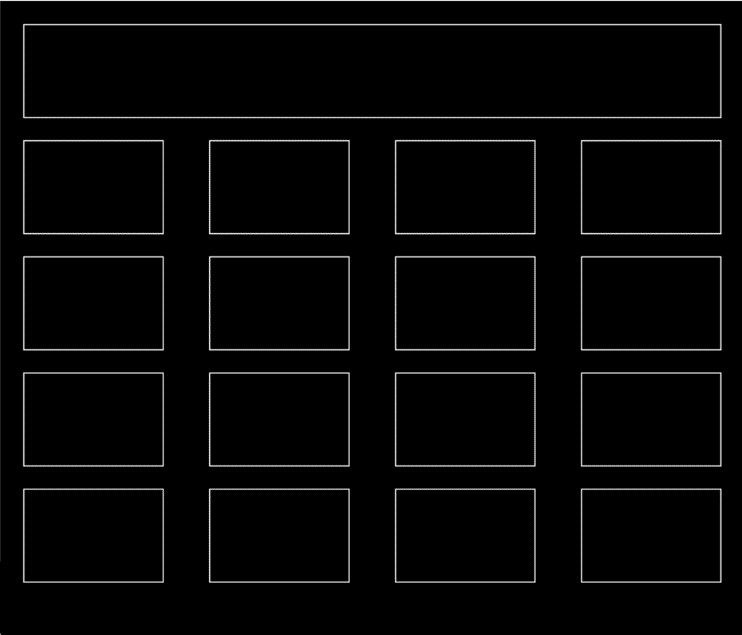
In order to be able to edit, assemble and run m68k programs on Windows, we used EASy68k Structured Assembly Language IDE.

## Ease of use

As our microprocessor was widely used in a, for that time, new generation of personal computers with graphical user interfaces, we decided to design a simple user interface to meet today’s standards slightly more.

The layout consists of rectangles in 4 columns with 4 rows each, making a total of 16 buttons. 10 buttons are for numbers and the rest 6 are for multiplication, division, addition, subtraction, equals, and clear all to simulate the physical calculators with buttons. The display window where you can see the selected numbers and the result of the chosen mathematical operation, is located above all these buttons.

Inspiration for this layout came from the physical calculators and other calculator apps with user interface. For the ease of use, we wanted to provide a dynamic feedback for the user. Highlighting while hovering and change of colour once the button has been pressed add our app more modern and smooth feel.



1. Calculator UI layout

## Functionality

As mentioned beforehand, this simple calculator can perform 4 mathematical operations – multiplication, addition, division, and subtraction. It uses the MULUS, DIVS, ADD, and SUB operations for this. As we are using the mouse as our input, we have to check the mouse position constantly and determine if a button has been pressed. We created a mouse IRQ for this. Once we register that the button has been pressed, we then check which button it was by the BNE commands.

You can also find the classical equals and clear all options there as well.

## Instruction set

First, your exceptions should be disabled. You can check this when you press execute and then open the Options menu and look for the Enable Exceptions line. It shouldn’t be ticked.

Now you can proceed to use our calculator. Simply choose the first number by clicking on it with the left mouse button. Then you choose the mathematical operation you wish to perform. You can do this by clicking on the box with it’s icon. Now, you can proceed to enter your second number the same way as you chose the first one. If you wish to calculate your equation, simply click on the equals sign to show you the result. If you wish to do more calculations, use the CE button to clear everything so you can start your new equation.

You can only calculate 2 numbers at a time. If you insert more numbers with more mathematical operations, only the last mathematical operation on the last two numbers will be performed. The result is not stored anywhere, if you want to use it as a value in your next calculation, you should manually choose the number. Otherwise, the last number you chose beforehand will be inserted instead of the result.

A screenshot of a computer

Description automatically generated

1. Disabled exceptions

# Code explenation

Finally, we can move onto the code explenation part. As this was our first real Assembly project, we learned a lot by the hard way of error and trial. The main challenges we faced were the drawing of the rectangles and changing the colour of the pen, getting the mouse x and y position and saving it, collision with the number boxes, displaying the text in the correct position, and finally the mathematical operations by itself. I will now elaborate on some of these topics and explain it’s code.

## Main game loop

Before creating the core game loop that will constantly check for the user input, we have to set up the window size and the mouse first.

To set the window size, we move #33 into the D0, then we can move #<screen width>\*<memory location>+<screen height> into D1. Now we can perform trap #15.

To set the mouse, we move #60 into D0 and do trap #15. An IRQ is created when a mouse button is pressed, released or moved in the output window.

After this, our programs enters our main game loop that consist of update and render. We branch to subroutine update, then to subroutine render and then we branch back to the start of the game loop, aka subroutine update.

## Getting the mouse position

This was surprisingly the hardest part. The main reason was that we had to perform a logical shift to the right and use logical AND operations to get the x and y coordinate separately.

Trap #15 and task #61 id D0 reads the mouse state. Then we can specify in the D1 what action we want to read. 00 to read the current state of the mouse, that’s what we chose as we are highlighting the boxes that you are hovering over.

Coordinates are stored as YYYY XXXX. To get the x coordinate, we have to do AND.L #$0000FFFF,D1. This will clear the first 4 numbers representing the y coordinates. However, here we run into the main issue where we couldn’t get the y coordinate back afterward. To solve this, we had to use logical shift right (LSR). This instruction will shift the bits of the destination operand to the right.  The number of times that the bits are shifted is decided by the source operand. In our case, we had to shift it by 16 bits LSR.L #8,D1 twice in a row. Beforehand, we obviously had to store the whole mouse position somewhere else. Therefore we performed the actions in this order:

* we got the mouse coordinates in D1
* stored position in D3 as well
* performed AND.L #$0000FFFF on D1, which got us the x coordinate and gets stored in A0
* we clear D1 and move the original position stored in D3 there
* perform logical shift right which will give us the y coordinate

Now we can return back to our update loop. The next thing we do is that we check if the mouse button was clicked. We store the mouse state in A1, we move it to D0 so we can perform CMP #1, D0. CMP or Compare command subtracts the source operand from the destination operand to get the CCR conditions.  The destination operand is not changed however. After this we can check if any button has been pressed. If the value in D0 is less than 1, no button has been pressed and we will branch to NO\_CLICK. For this we are using BLT. If button has been clicked, therefore the number in D0 is greater than 0, we branch to subroutine HANDLE\_CLICK by using BSR.

In our HANDLE\_CLICK we check which button has been pressed. The first chosen number is stored in A3, chosen operator is stored in A4 and the second number is stored in A5. The result is then stored in A6.

## Render

We enable the back buffer by trap #15 and task #94 in D0. Now we can move onto drawing the display rectangle which is in the subroutine DRAW\_OUTPUT\_FIELD. Here we set the pen color. We first define our colors as constants. We are using white, yellow and blue. This is how we defined white color:

*WHITE EQU $00FFFFFF*

The RGBA order starts at the right side because of the least significant bit. Therefore the order is ABGR.

We also have to define a pen type. First, we define then as constants, in our case we are using fill pen - FILL\_BOX EQU 81, and a normal pen - NORMAL\_PEN EQU 80. Their tasks (80 or 81) in D0 and Trap #15 will set the pen type. After this, we set the starting position and width and height by using constants that we defines as well and we draw the rectangle representing the display.

As we return back to our render loop, we now have to check for the collision of the mouse with the action buttons so we can provide the dynamic feedback in a form of highlighting. We now branch to CHECK\_MOUSE subroutine. In here, we at first move the predefined button position to the D1 and then the mouse position to D3 where we perform logical AND operation to get the x coordinate again. Now we can compare D3 and D1. If D1 (the button x position) is greater than the mouse x pos in D3, we return to the render loop because we are out of bounds. If it is not, we continue and we add predefined button width to the D1 and compare if that x button position is less than mouse position. If it is, it is out bounds and we return to render loop as our mouse cursor is out of bounds. If we are withing x bounds, we now perform same check but for the y coordinate. I we are withing bounds of a button, we change the highlight of the box. If we register that the button has been pressed, we change the color of the box to blue until we are pressing the button. All of these are stored in D5.

Since we know the state of mouse and its collision now, we can finally move onto the drawing of the buttons. This is being done in the DRAW\_BUTTON subroutine. We set the pen style and check for the highlight color by using the D5 and apply it. To draw the buttons, we used the pre-defined positions and size. We then repeat this for all of our buttons with adjusted positions.

To finish the rendering of buttons, All that needs to be done is to display the text at the right location. We can set the location of a string by task #95 in D0 and trap #15, then we can specify the x and y locations by adding them to D1 and D2. This terminates our render loop and returns back to the main game loop

## Mathematical operations

As mentioned beforehand, we focused mainly on the UI, therefore we kept this part more simple. We are checking for the selected operators in the HANDLE\_CLICK subroutine, where we insert a specific number into the A4 accordingly. Once you click the equal button, the program moves A4 into D0 and checks which operation has been selected by using the BNE command. According to the selected operation, it performs MULUS, ADD, SUB or DIVS functions.

# Results

The final product that came out of this project is a simple calculator with a focus on the graphical interface where we perform collision checks, IRQ for mouse input, and dynamic feedback for the user in the form of color changes of the buttons. Since the calculator is widely known and done by many in the past, it is almost impossible to come up with something jaw-dropping. Despite this fact, this project showed us what happens behind the calculations and that something so simple as mouse collision checking is in fact quite hard and we should be grateful for how easy it is for us to use in the high-level programming languages.

# Conclusion

All things considered, this project forced us to research a lot of new things on our own, such as more tasks that can be performed for getting the mouse position or changing the pen color. There are certainly a lot of possible improvements for the future. One of the firsts would be to work more on the actual calculations and give more options for the user. Possible options would be to be able to input equations that are longer than 2 numbers, determine the priority of the mathematical operations, and save the result so it is easy to reuse it in the next calculation.

1. Student contribution

|  |  |  |
| --- | --- | --- |
|  | Topic | % Contributions |
| Student 1 | David Strikaitis | 50 |
| Student 2 | Veronika Syncakova | 50 |