

GRAPHING CALCULATOR

University of Hertfordshire

Erol Gelbul

17062941

**Brief Description**

The project idea itself came out to be impossible to achieve at first glance, because of the amount of information that would have to go through the process with abundant instructions and pass onto the device. The first demo was planned to be made by a single LED display, that would draw the graph of an abscissa(x) & ordinate(y) equation.

Other than the problem of small memory of the processor, the potential time it would take the device to draw a graph was going to be significantly high. Although, the initial plan was to get it working in any way shape or form.

The LED panel is made of 32 bits, 1 bit representing each light switch; 0 is off, 1 is on and x (third state) to be NULL. All states would show a different colour; therefore, it was necessary to determine which state would represent “off”.

How could the user enter values? Should this transfer be made before running the clock or during the runtime? How to display the values entered by the user? Should they be displayed?

Is a single 32x32 LED enough to display a graph? How far could this be extended for more clarity? Should x’s 0 and y’s 0 be included, in other words, a 32-string left out for the x and y axis, lighting up if the x or y has a point on this part of the graph? If this case exists which part of the graph should a 32-string may be taken from? Should this result in 31x31 graph, 16 negative y values and 15 positive y values.

This is a solo-project, everything has been prepared by Erol Gelbul. It is not part of any pre-made project idea, it does not follow any procedural steps. Erol Gelbul has been granted permission by Prof. Alex Shafarenko to do this as an ATP.

**The Circuit (Appendix A)**

The initiation starts in the circuit, where the values are entered by the user. The “User Interface” section is responsible for this process [1]. After each choice is ready, the “Choice Passer” section is responsible for sending these values when the processor demands them [2]. This is accomplished by the help of the I/O bus, where in/out’ and I/Osel should be 1 at the same time.

After the processor has completed its job, with the help of a special instruction with register-specific instruction code, the data coming from the processor is fetched via the design located at the “Data Fetcher” region [3]. After the confirmation of the original instruction, when the I/Osel is up, which means there is data currently being initialized (stored) at memory location 0xf0. Additionally, this design servers another feature, counting the amount of times a data is being passed (new data coming in), this will be useful for advancing on the next 32-string on the LED display.

The 4 LED display is hooked up to tunnels each visioning a 32-bit string. [4] For the simplicity each LED is numbered. 1-2 regarding the whole set of y values at a given x, starting from -32 till -1. 3-4 regarding to the set of y values at a given x, starting from 1 till 32.

The translators [5] are placed to transform the data taken from the processor, in order to make it readable by the 32-bit LED display. This is accomplished by the constants placed for each piece of 32-bit LED input.

The triggers [6] are set up for the upcoming raising design. Allows the device to move on to the next data after each x value program cycle.

The last information passage is the deciding circuit [7]. This design focuses on which LED will have all 0’s and which LED will have the data input. For instance, if the value from the processor for x value: -12 is 30. The 20th LED will be passed onto the upper LED since it is positive (looking at the 7th bit of the value), therefore, the lower LED will get all 0’s. The data input for this design is the translated version of the processor value.

**The Assembly Program (Appendix B)**

The assembly code kick-offs with getting values from the device, and then pushing these values into the stack, which makes it easier to read and use these values.

Some of the stack operations have a gap in between them. This is for the practicality reasons, also allows space for the possible expansion of the program code.

For the operations concerning the calculator, there are 3 subroutine segments. One of them is for multiplication, and the other 2 are for division. The reason for having 2 division types is for dividing a positive number and a negative number. The structure of the division requires this makes it mandatory.

**Codesign Issues**

The registers in the circuit design would start to disagree with the I/Osel after a certain amount of cycles. Mainly register in the Data Fetching station were set up as Rising Edge, however this would cause a problem on the upcoming stages. High-Clock solution made it possible to alter the problem.

Previously the main idea was to treat all LED panels as one. This was possible for positive values given by the processor. However, when the graph reached to calculate the negative values, it would get corrupted because of the inverse positioning of the lower LEDs. An example for this can be the value +1 for y, being the 0th bit at the upper LED, and -1 value for y being the 31th bit at the lower LED. This would add an extreme level of complexity to the project. Therefore, it would be easier to treat each LED different then each other. Although, connection between 1-2 and 3-4 must be established, since a value coming from the processor should concern a value parameter starting from -32 to 32.

**Testing Strategies**

The entire cycle for a specific x value could take extremely time consuming to complete. In order to monitor this progress, in each division cycle the quotient (increasing as division progresses) is being pushed to the stack.

Other testing strategy used in the circuit design was the register abundance. If a value was being passed registers would hold these values until the next cycle of x value procedure.

Testing for the assembly program had a simple solution, pushing desired values into the stack at the very start of the program would benefit in testing.

The circuit design has procedural steps, overall. Starting from left to right, data is entered, altered and displayed. Therefore, unit testing was straightforward. The structure of the design is like a spanning tree. Ultimately, the data entering the unit is certain.

**Extra Features**

* Addition of 3 LED panels, total of 4 panels.
* Constant in the Equation
* Coefficients can reach up to 9

**Exploration and Optimisation**

The completion time for the first test of 4 LED panels was, on estimate, 4 times longer than the current calculation speed. The program code was significantly improved in terms of faster multiplication and division.

There were certain calculations in the program code that would increase the completion time of the calculator. They were removed and replaced in the circuit design. For instance, the index (counter) [3].

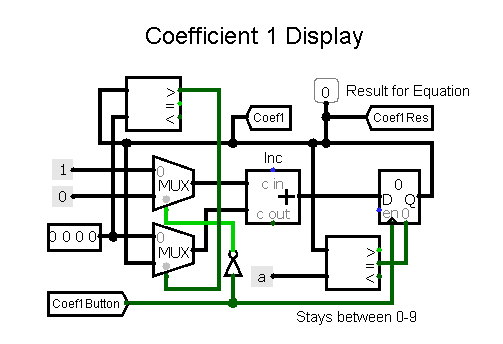
**Demo**

Before running the circuit, using the buttons, select the values for coefficient 1, coefficient 2 and constant. Then run the circuit.

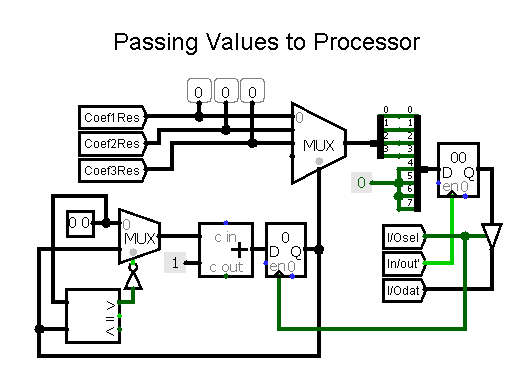
**Statement of Contributions**

Not included since this is a solo project.

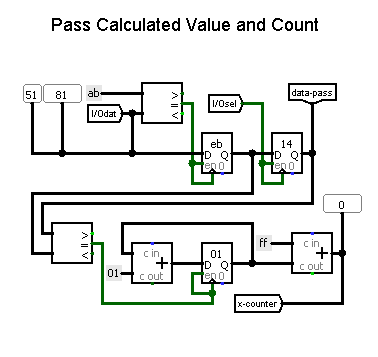
**Appendix A**

****

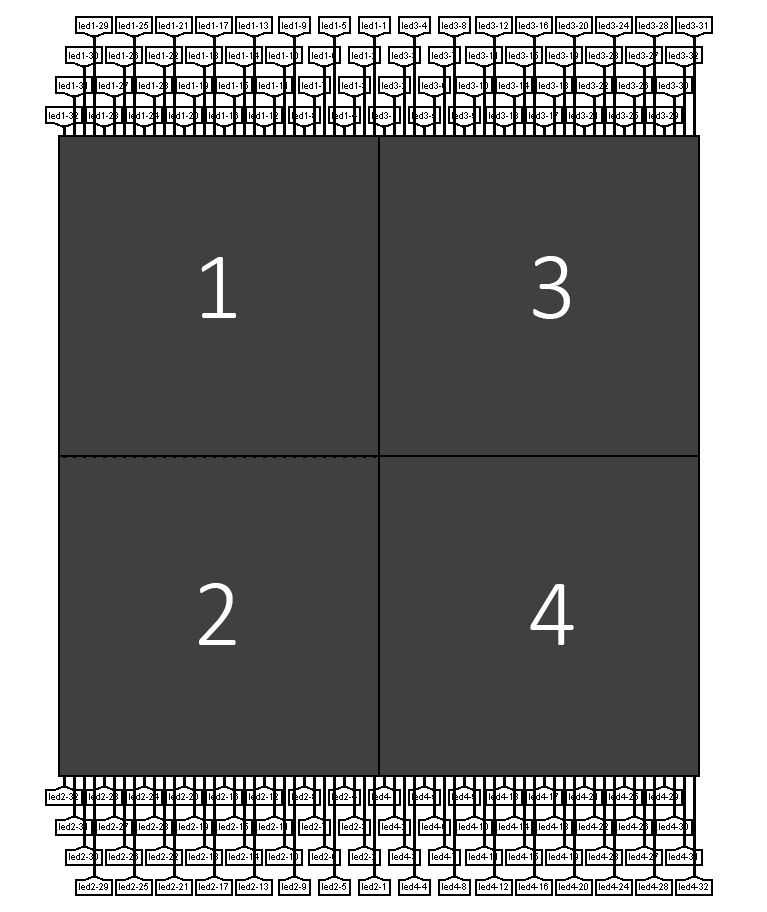
**[1]**



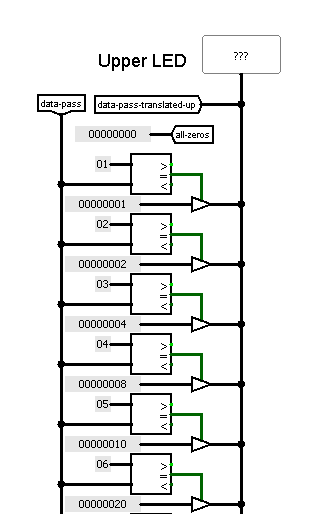
**[2]**



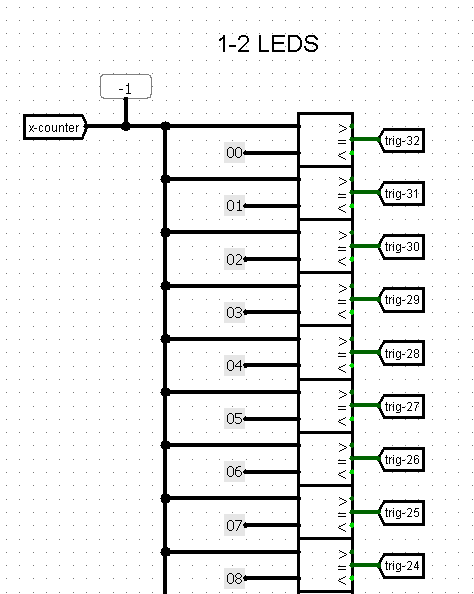
**[3]**

****

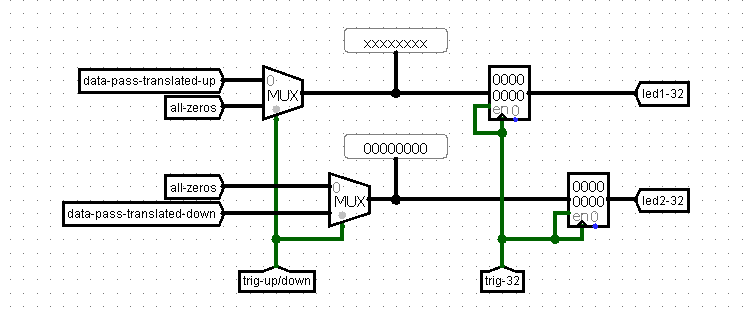
**[4]**

****

**[5]**

****

**[6]**

****

**[7]**

**Appendix B**

asect 0

setsp 0xf0 #set the stack pointer to 0xf0 for data flow

ldi r0, 0xf0 #load r0 with the adress 0xf0

ld r0, r1 #load the value passed from to device into r0, using r1

push r1 #push the value into the stack (coef1 value)

ld r0, r1

push r1 #coef2 value

ld r0, r1

push r1 #constant value

clr r1

clr r2

ldi r1, -32 #r1 starting at x value -32

ldi r2, 32 #should processed until 32

while

cmp r2, r1 #looping -32 to 32

stays ge

clr r2

addsp 2 #move stack pointer by 2

pop r2 #r2 now holds coef1

addsp -3 #move stack pointer by -3

jsr multiply #go to muliplication

move r3, r0 #moving r3 to r0 where it is saved in each subroutine (save ans from 1st mult)

pop r2 #pop constant

add r2, r0 #constant + (X x Coef1)

if #this if statement is to check if the constant + (X x Coef1) is 0, which in this case shouldnt be negated

tst r0

is z

#do nothing

else

neg r0 #move value to right, so negate

fi

pop r2 #pop coef2 into r2

addsp -4 #ready it for division sub by moving the stack pointer

if #depending on the value on the RHS, certain type of division will be executed

tst r0

is pl

jsr posdivision

else

jsr negdivision

fi

setsp 0xf0

ldi r2, 0xf0

st r2, r3 #passing the data to the device

ldi r2, 32 #reset the r2 for the bigger comparison while loop

inc r1 #inc -32 to -31 to -30 and so on

setsp 0xed #reset location for the stack pointer

wend

multiply: #need r1 to be X or Y # need r2 coef

save r1

#r0 is the count = set to 1

#r1 is the X or Y from device -32, -31...

#r2 is the coef multip

#r3 is product sum = set 0

clr r0

ldi r0, 1

clr r3

ldi r3, 0

while

cmp r0, r2

stays le

add r1, r3

inc r0

wend

restore r1

rts

posdivision:

save r0

save r1

#r3 count = set to 0

#r2 divider (Coef2) (bottom)

#r0 dividend(top)

clr r3

ldi r3, 0

while

cmp r0, r2

stays ge

neg r2

add r2, r0

neg r2

inc r3

ldi r1, 0xc0

st r1, r3

wend

restore r1

restore r0

rts

negdivision:

save r0

save r1

#r3 count = set to 0

#r2 divider (Coef2) (bottom)

#r0 dividend (top)

clr r3

ldi r3, 0

neg r0

while

cmp r0, r2

stays ge

neg r2

add r2, r0

neg r2

inc r3

ldi r1, 0xc0

st r1, r3

wend

neg r3

restore r1

restore r0

rts

end