Visual Graphic Array Floating Point Calculator

Kelley Harris, Benjamin Hayes, Benjamin Jackson, William Strand
Electrical and Computer Engineering Department
School of Engineering and Computer Science
Oakland University, Rochester, MI
e-mails: kjharri2@oakland.edu, bshayes@oakland.edu, bwjacks2@oakland.edu, wwstrand@oakland.edu

***Abstract*— Using Vivado, a Floating-Point Calculator was implemented using the Video Graphics Array (VGA) controller on the Nexys4 DDR Field Programmable Gate Array (FPGA) board to display the inputs and outputs generated by a floating point arithmetic unit. Images uploaded from MATLAB are used and downloaded onto the VGA controller.**

I. Introduction

The need for more accurate and precise scientific instruments is ever present in the fields of technology and research. Developing an understanding some of the instruments used today could lead to their refinement or even the creation of new instruments. This project attempts to break down the process of manipulating data in the form of a floating point calculator, as well as display the information in a meaningful way. The calculator will involve addition, subtraction, multiplication, and division, as well as a control for the number of integer bits and precision bits. The control for the calculator will come from the Nexys' switches and push buttons. The information generated from this will be routed to a VGA circuit, which will interpret the data and display it on a monitor screen.

II. Methodology

A. Input Method

The input system is done using the sixteen switches on the Nexus4 board. In total, two 32-bit floating point numbers and one 2-bit operation code need to be taken in. With the limited amount of switches, this needs to be done in 5 stages and controlled by a finite state machine (See Figure 01). The 16 most significant bits of the floating point number are put on the switches and then loaded into the program and simultaneously displayed on the 7-segment display (in hex) on a button push. This is repeated for the 16 LSB's, the operation code and finally the 16 MSBs and LSBs of the second floating point number. These 3 signals are then used as the inputs to the calculator unit. The state machine waits in state 6 indefinitely until reset is pressed.

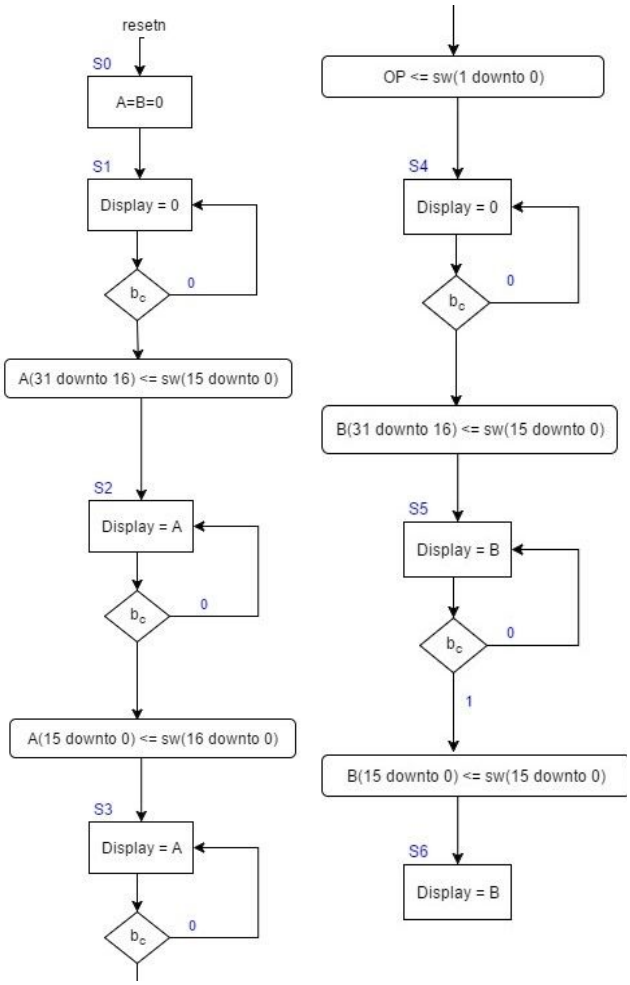


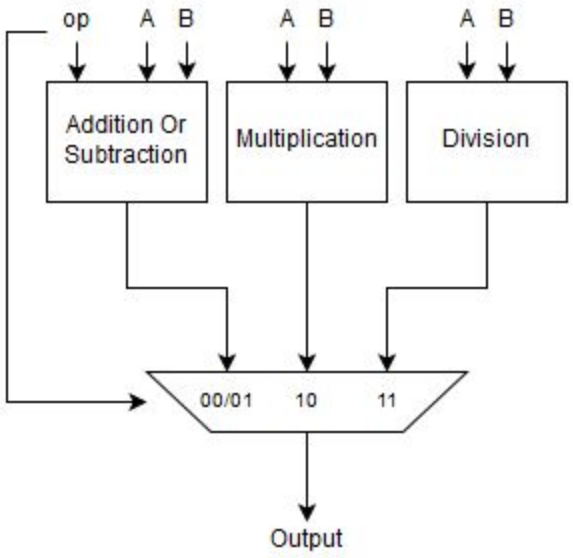
Figure 01: Input FSM

B. Calculator Architecture

An arithmetic module takes in the two 32-bit floating point numbers and a 2-bit operation signal (add, subtract, multiply or divide) from the input system, interpreted as single representation under IEEE-754 standard [1]. There are three circuits inside the arithmetic module, detailed in Figure 02 below: multiplication and division are separate circuits but addition and subtraction are integrated together and the appropriate operation is chosen via a multiplexor in

the arithmetic module top level circuit. All three circuits run in parallel and the desired result chosen afterward, this works but may be a source of slowdown in the project and as a future optimization could be simplified.

Figure 02: Arithmetic Module



The multiplexed output signal needs to be sent to the VGA to be displayed but at this point it is a 32-bit binary number and not in a very human-friendly format. To correct this it is sent through an intermediary step to be converted from binary to a readable decimal number. Detailed diagrams of the individual arithmetic units are included at the end of the report [Figures 06 & 07].

C. BCD Conversion

While the binary input and output floating point numbers could have been chosen to be displayed on the VGA, it is not easy for the user to read. The floating point numbers have three components that we need to convert to BCD: the sign bit, the 8-bit biased-exponent and the 23-bit mantissa. Converting the sign bit was as simple as using that bit as a selector in a 2-to-1 MUX and having the outputs be the appropriate plus and minus BCD codes. Converting the 8-bit exponent is a straight 1-to-1 conversion done using a simple, well-known conversion algorithm called the shift-add-three algorithm (also called the double-dabble algorithm) [2]. Before the 8-bit number was input into the BCD converter, the number needed to be converted into a signed number, the bias removed, the sign bit extracted and converted the same as the sign bit for the entire number, and the number returned to an unsigned format to be put into the converter. Dealing with the mantissa is quite a bit more involved. The mantissa is interpreted by the hardware as a 23 bit unsigned number, but in reality it must be interpreted as a fixed point number of the format [24 23], or 0.xxx… That is, if the unaltered mantissa was fed into the BCD, the output would not represent the integer values of the number. To solve this problem, the number would be multiplied by 10^N and then multiplied by 2^-N, where N is the size of the entire number. In theory, N would be equal to 23 since there are 23 numbers (it would not be 24 because the integer part, or left of the radix, is zero). However, if these operations would be carried out with N equal to 23, the result of the first multiplication would be a number larger than (2^31) - 1, the largest integer number that Vivado can handle. So, the 7 MSB's had to be extracted from the mantissa and be multiplied by 10^7 in order to not violate Vivado's rules. After this operation was completed, the resulting number can be fed into the BCD circuit. However, this solution results in a number that is slightly different than the actual mantissa, creating some error.
The results of each BCD stage (sign bit, e plus bias, and mantissa) were fed directly into the VGA controller.

D. VGA Controller

The VGA controller takes in the truncated digits of the calculator inputs and output after their conversion to BCD. We will display the first 4 significant digits of the output and its three digit exponent on the monitor, in the format of ± 1.xxx E ± xxx. To avoid further conversion complexity, the desired final number to be displayed on the VGA was chosen to be simply the decimal conversion of the significand and the decimal conversion of the exponent, this means the first digit of the decimal significand will always be one (as support for denormal numbers is not fully implemented) and means this "1" can be combined with the decimal point as a single character to save slightly on code complexity. Further, the exponent "E" can be combined with the exponent sign in the text file to reduce the number of characters to be displayed.

This means there are nine characters to be displayed on the VGA, one constant and 8 variable, with a total range of 15 possible characters. Covered below, matlab was used to convert these fifteen 64*64 pixel characters as images into a usable text file to be uploaded into 15 RAM modules inside the VGA controller.

The output data of these RAM modules are fed into a 15-to-1 bus-multiplexor. The selector of this multiplexer is the heart of the entire VGA controller. It is the result of a series of If statements that take in the current horizontal and vertical count (HC & VC) along with the BCD codes for the nine characters to be displayed. For example, for the first image it needs to be displayed while HC is between 0 and 63 AND while VC is between 0 and 63 as well. So the first If statement will be of the format (simplified for readability):

```
If 0≤HC≤63 AND 0≤VC≤63 then
    selBusMux <= sign;
```

```
End If;
```

Where selBusMux is the 4-bit signal that selects between the 15-to-1 mux, and sign is the 4-bit BCD code that we defined to be for either '+' or '-' and corresponds to the RAM multiplexer input for the '+' or '-' text file. So for an output number that is positive in magnitude, the BCD converter will give out a "1110" for character 1, where "1110" is the chosen code for '+' and selects the output of the 15th multiplexer input to be sent to the display (The full multiplexer code is available at the end of this report along with the relevant section of the VGA block diagram, see Figure 08 & 09). This process continues for the remaining nine characters: the conditions for VC never change as all the images are the same height while the conditions for HC increment the lower and upper bounds of the greater-than and less-than statements by 64 each. So the code for the second image is:

```
If 64≤HC≤127 AND 0≤VC≤63 then
    selBusMux <= "1101";
End If;
```

There are a few remaining pieces to the VGA, first is the selection between the multiplexer output, a 0xFFF code for a white background, and a 0x000 code for the porch areas. The background selection is done with a simple 2-to-1 bus mux that look only at HC and VC to determine if they're inside the drawing area (`HC≤575 and VC≤63`), outside of which 0xFFF replaces the RAM multiplexer output signal. The porch selection is decided by a video_on signal that is generated inside the provided VGA code.

The last remaining piece to the VGA controller is the address selection for the RAM inputs. Each RAM module has an input of [log base-2 of the image size] bits, in our case of a 64*64 image size this comes out to 12-bits wide. This input signal for each of the 15 RAMs is the concatenation of the six LBS of both the VC and HC signals respectively.

Since our image locations are A) unmoving, B) squares of 2^6 size and C) starting at the top left corner (0,0) we can use the cyclical nature of this address signal to our advantage to significantly simplify the address selection code for all RAM cases to be:

```
inRAM_add <= VC(5 downto 0) & HC(6 downto 0);
```

What the general goal of the address selector is is to load out the first RGB value when VC & HC are at the top left corner of where the image is to be displayed. So at HC=VC=0 the ram address is 0 and selects the first value in the text file. At HC=1, VC=0, the ram address is 1 and outputs the second value in the text file. At HC=VC=63, the ram address is "111111"&"111111" corresponding to the final value in the text file. Now, HC and VC can both take numbers larger than 63, so as stated before the ram address only takes the bottom 6 bits of each. This works perfectly to our advantage as we can demonstrate by looking at the address values for the second image:

The top left of the second image starts at HC=64, VC=0, this corresponds to an HC value of "1000000" where very critically the lowest 6 bits are "000000", **exactly** as they were in the top left corner of the first image and thus the entire address is 0 and will correctly select the first value of the corresponding text file. It can then be shown by the same logic, that the correct address will be selected for each pixel of not only the second image but also any of the nine images.

E. MATLAB

To begin, simple images needed for the VGA display are created in paint and then read into the MATLAB file. The images are resized to fit into the designated pixel ranges and converted into a 12-bit RGB image which will allow the image to be easily downloaded onto the FPGA board. The MATLAB code used for these images is taken from the class website [3] and slightly edited to read the new image files while eliminating the default android image that was used for the class website's example. The MATLAB code (Figure 03) that was used and the images converted (Figure 04) are shown below.

```
clear all; close all; clc;

I = imread ('negative.png'); % RGB image
figure; imshow(I);

% Resizing the image to 256x256:
IP = imresize(I,[64 64]);
figure; imshow (IP);

% 24-bit RGB image: we will convert it to a 12-bit RGB image:
for i = 1:3
    IN(:,:,i) = IP(:,:,i)/16; % every plane converted to 4 bits. right shift
end

figure; imshow(IN*16); % This is just so that 'imshow' can display the image properly

% ------------------------------------------------------------------------
% Converting to text file. Format: 0|R|G|B in hexadecimal
q = quantizer ('ufixed', 'round', 'saturate', [4 0]);
textfile = 'negative.txt';
fid = fopen (textfile, 'wt'); % generates text file in write mode

for i = 1:64
    for j = 1:64
        R = IN(i,j,1); G = IN(i,j,2); B = IN(i,j,3);
        Rh = num2hex(q, double(R)); Gh = num2hex(q, double(G)); Bh = num2hex(q, double(B)
        fprintf(fid, '0%s%s%s\n',Rh, Gh, Bh);
    end
end
```

Figure 03: MATLAB code

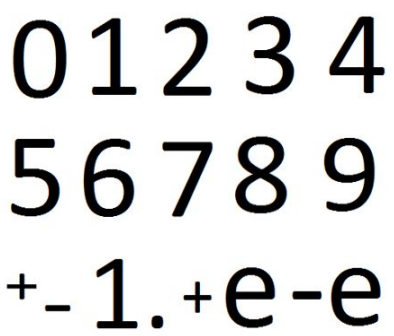


Figure 04: Images used

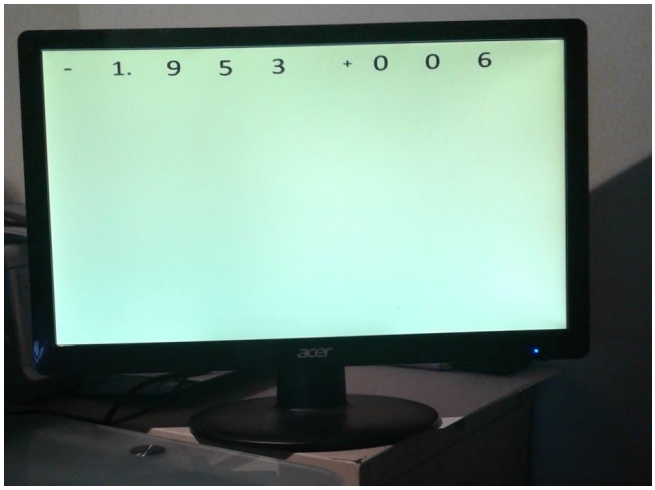
III. Experimental Setup

The physical project consisted of only the Nexys 4DDR board and an external VGA-capable display. All of the inputs and processing are done on the Nexys board and the only output is the VGA signal which displays the calculator/BCD converter output on the screen if all is functioning correctly.

IV. Results

The system successfully displays the correctly converted calculator output on the screen as can be seen below:

Figure 05: VGA Monitor Output



The results given as the final result are not completely accurate in all cases. Firstly, there is inherent error introduced by representing a fractional decimal number in floating point, for example some numbers cannot be represented exactly in binary, “1.2” for example as a single floating point format comes out to be 0x3f99999a which, converted back to decimal is 1.2000000476837158 * 2^0. The arithmetic unit also seems to give slightly inaccurate results, the reason behind this is unknown as A) time was not available to investigate and B) there was a much larger error induced by the BCD conversion process. That BCD conversion error is the result of limitations in the maximum size of numbers usable in the FPGA IDE. In order to convert the entire significand to BCD it would need to be multiplied by 10^23 in binary, it turns out that this number is many orders of magnitude larger than the largest integer supported natively by Vivado. The best we could do was to obtain the BCD conversion of the top 7 fractional bits of the mantissa. For some numbers and operations this causes the lowest displayed digit on the VGA to be off by one or two. The final project presented had a mistake wherein the “E-” and “E+” text files were accidentally replaced with just “-” and “+” respectively.

Conclusions

The project proved to be much tougher than originally planned. Interestingly, the arithmetic unit was probably the easiest piece to implement while the supporting structures like the BCD converter and VGA controller proved to be very difficult. It was a difficult learning process to get these to work correctly. Possible improvements include:

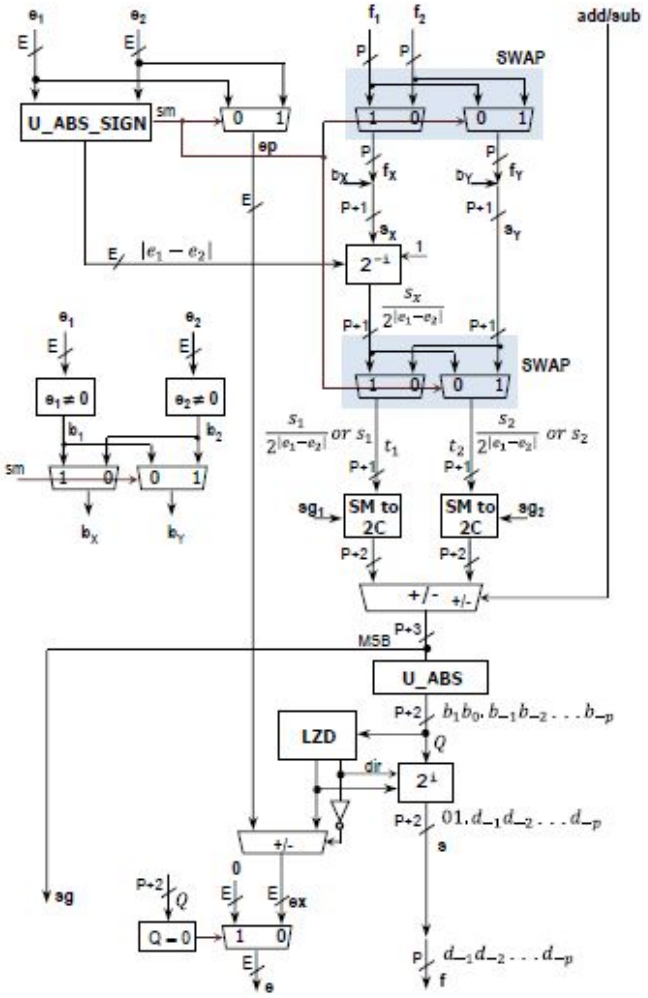
-Display of the inputs and the operation on the VGA
-Using a keypad/keyboard for input
-Increased accuracy of the output
-Full support for denormal numbers

References

[1] http://ieeexplore.ieee.org/document/4610935/

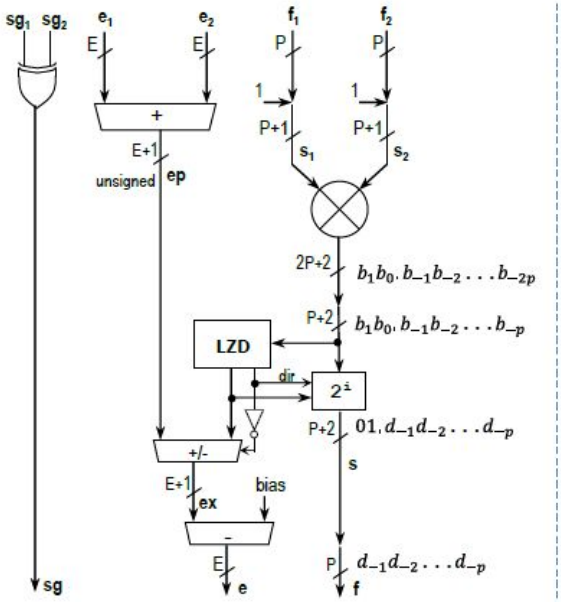
[2]https://www.embeddedrelated.com/showthread/comp.arch.embedded/18060-1.php

[3]]http://www.secs.oakland.edu/~llamocca/Tutorials/VHDLFPGA/ISE/Unit_7/img2txt.m

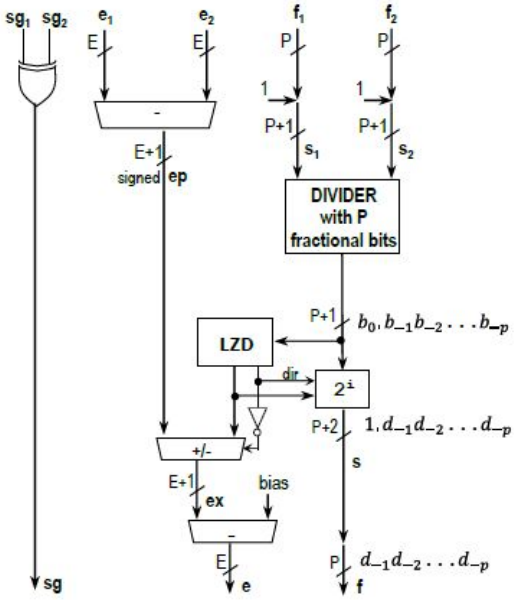


Left: Figure 06: Adder/Subtractor Block Diagram

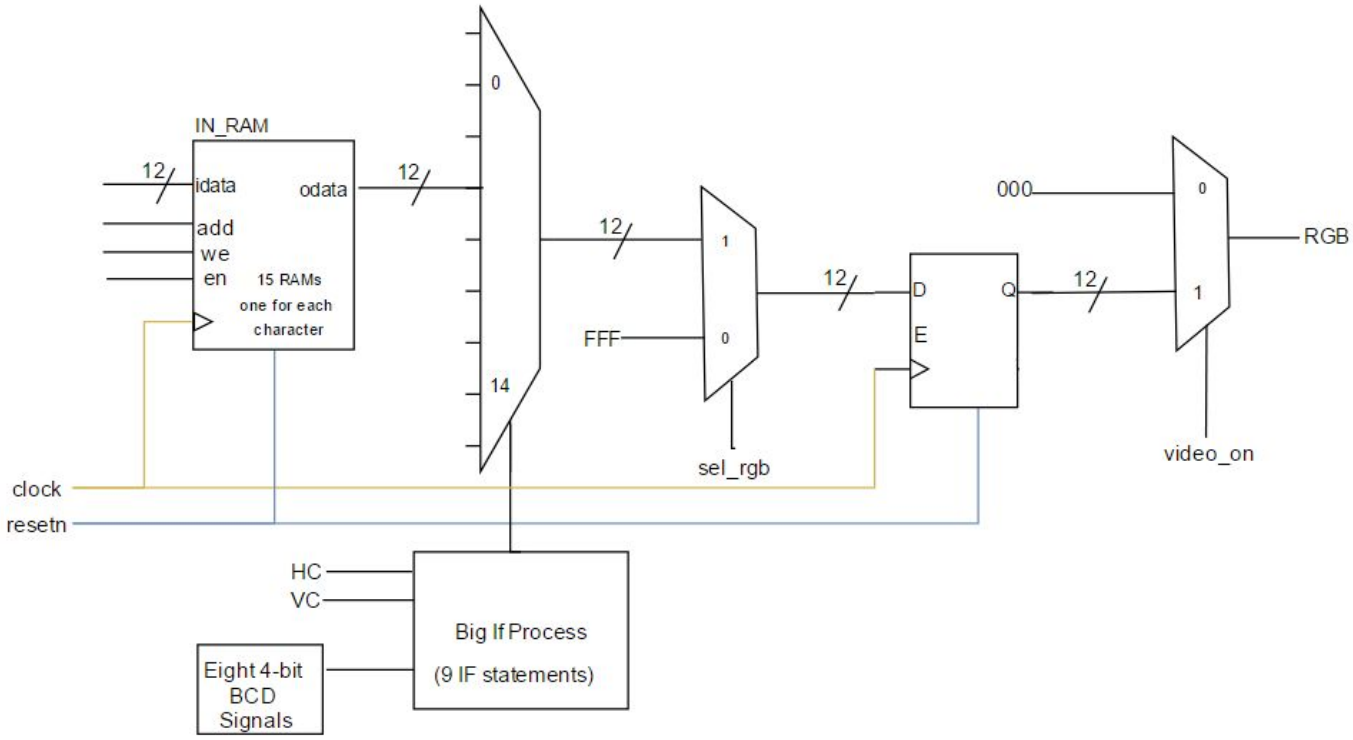
Below: Figure 07: Multiplier and Divider Block Diagram



FP MULTIPLIER



FP DIVIDER



Above Figure 08: VGA controller simplified block diagram
Below Figure 09: VGA 15-to-1 Bus Mux code

```
with selBmux select
       inRAM_odata <= inRAM_odata0 when "0000",            --this is the text file for '0'
                      inRAM_odata1 when "0001",        --this is the text file for '1'
                      inRAM_odata2 when "0010",        --this is the text file for '2'
                      inRAM_odata3 when "0011",        --this is the text file for '3'
                      inRAM_odata4 when "0100",        --this is the text file for '4'
                      inRAM_odata5 when "0101",        --this is the text file for '5'
                      inRAM_odata6 when "0110",        --this is the text file for '6'
                      inRAM_odata7 when "0111",        --this is the text file for '7'
                      inRAM_odata8 when "1000",        --this is the text file for '8'
                      inRAM_odata9 when "1001",        --this is the text file for '9'
                      inram_odataNege when "1010",     --this is the text file for 'e-'
                      inram_odataPose when "1011",     --this is the text file for 'e+'
                      inRAM_odataNeg when "1100",      --this is the text file for '-'
                      inRAM_odataOneDot when "1101",   --this is the text file for '1.'
                      inRAM_odataPos when "1110",      --this is the text file for '+'
                      "0000000000000000" when others;  --blank case, shouldnt ever happen
```