5-bit signed calculator

Mohammed Abdul Rafay, Mohammed Abdul Wasay, Sinan Ghareeb
Electrical and Computer Engineering Department
School of Engineering and Computer Science
Oakland University, Rochester, MI
e-mails: mohammedabdulra@oakland.edu, mohammedabdulwa@oakland.edu, sinanghareeb@oakland.edu

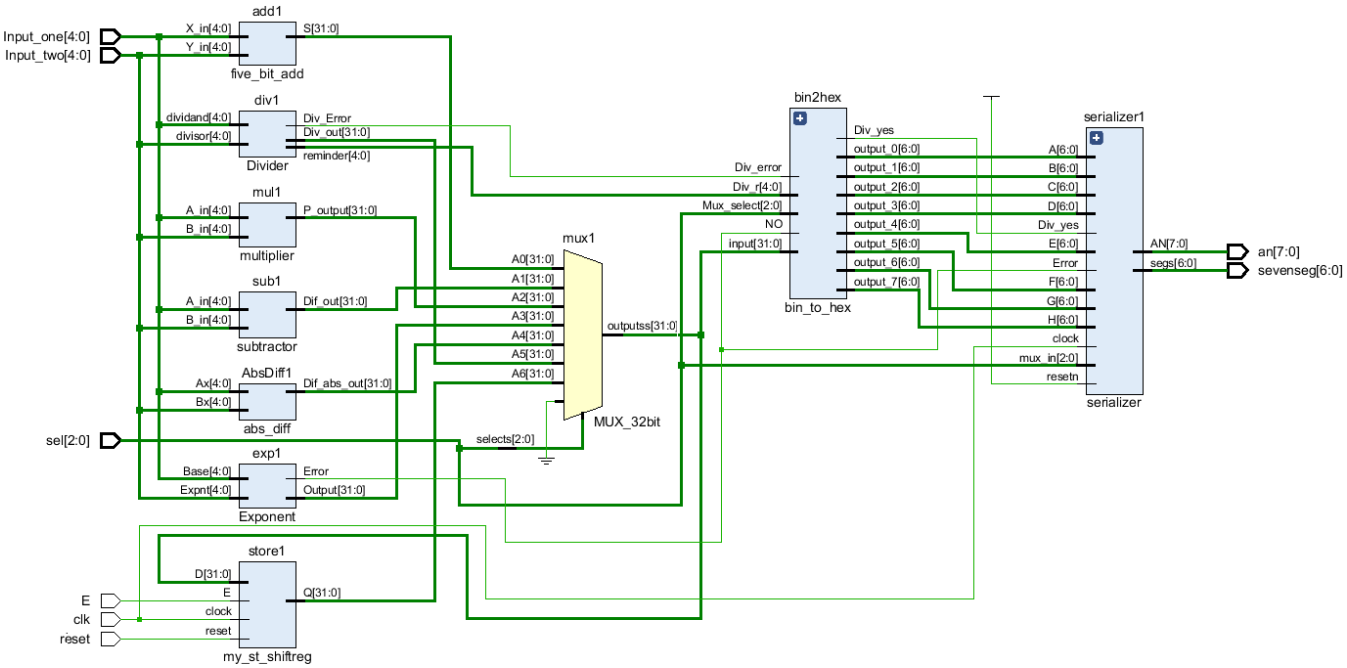


Figure 1: A schematic of the top-file design of the calculator.

***Abstract*—This project aimed to create a signed calculator with the ability to perform addition, subtraction, multiplication, and taking exponents, among many other functions. Converting into integer then calculating was very helpful in many components, as well as the use of conditional statements. The serializer and binary to hexadecimal converter had many custom changes to accommodate an "error" display on the board. Many changes could be done to improve this project to make it better suited for more advanced research or educational uses.**

I. Introduction

This project's aim was to create a general calculator with binary inputs using switches, hexadecimal outputs from eight seven segment displays, a few new functions in order to set it apart from other calculators. In addition to the basic functions of addition, subtraction, and multiplication, a division function displaying the remainder alongside the quotient was made. Also, an exponent calculator and store function were also created. To simplify coding, in some components, the inputs were converted into integers and then calculated. For the division and exponent though, more modifications were required to take into account inputs that were unable to be computed either due to a lack of displays or due to undefined outputs. Modifications accommodating special cases as those described were also done on the binary-to-hexadecimal converter and the serializer. These modifications mostly involved the use of numerous "if" statements. Testing revealed some important mistakes in these modifications that were later resolved.

This project has the potential to be applied in many areas in real life. This calculator could be adapted for use as a simple scientific calculator, by adding a function for scientific notation and providing the output in decimal format. Another application is to solve simple mathematics problems, like those in an educational setting or in accounting.

II. Methodology

This is the body of your report. Here you explain how you designed your project.

A. Overview

The goal of this project is to visualize the calculator on the Nexus-A7 board. In order to that, essential task of designing the code needs to be completed. This task determines whether

the circuit is feasible. The main steps for designing this specific signed calculator successfully is as follows:

- A schematic detailing the top-level design should be created.
- The inputs and outputs of each component should be determined.
- The design of each component will be determined based on the task that should be performed.

Based on these steps, first, a diagram like Figure 1 was created. While the schematic was being drawn, the inputs and outputs of each component were determined. For example, the addition function should have two inputs for the two numbers being added and one output for the result. The order the components was also being determined. The inputs were decided to first pass through each of the components simultaneously, and then a multiplexor would determine which operation is being carried out. Then, the output would be modified by the binary-to-hexadecimal converter and then the serializer for the 7-segment displays. Next, each component was designed.

B. Addition

The Addition component was made using Dr. Llamocca's Generic 2's complement Adder/Subtractor Unit [1]. To avoid any scenario when there is an overflow with 5-bit inputs, the inputs were sign extended to 7-bits and then placed into a 7-bit adder. This adder had 7 full adders port-mapped in series. The output was then sign-extended to 32 bits for the multiplexor.

C. Subtraction

Signed subtraction was implemented using the library IEEE.NUMERIC_STD.ALL. This library allows for the implementation of general basic mathematical operations like addition, multiplication, division, and subtraction. Subtraction can simply be implemented using the (-) operator, but this operation can only be implemented on integer data types. This project has inputs provided in the std_logic_vector format; therefore, the input needs to be converted into integer, used for subtraction, and then converted back to the original format. This library helps by providing the to_signed and the to_integer functions, among many others. For example, to implement the subtractor, the command below is used :

std_logic_vector(to_signed((to_integer(signed(A_in)) - to_integer(signed(B_in))),32))

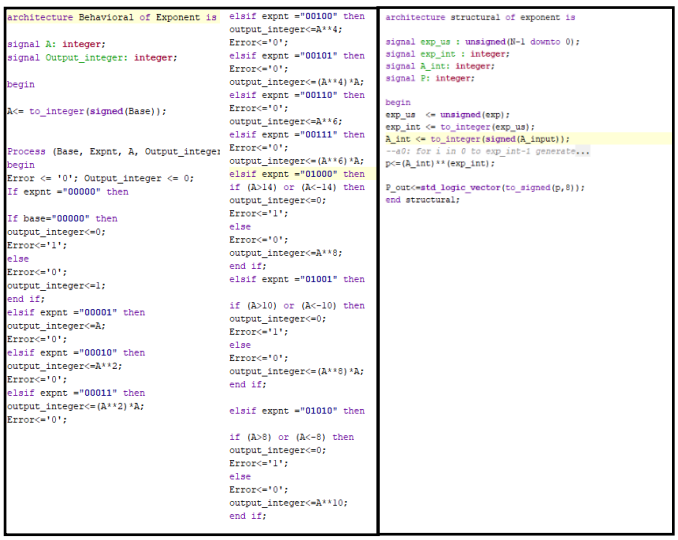
In this command A_in and B_in are of data type std_logic_vector which is converted to integer and then after applying the operator output is then converted back to STD__LOGIC_VECTOR. One benefit using this library is that the answer does not have to be sign-extended, as that is taken care of by the function used when converting back to std_logic_vector.

D. Multiplication

This component was made by adapting Dr. Llamocca's Unsigned Integer Multiplier [1] such that the outputs would be 32 bits. This needed to be done because the output goes into a 32-bit multiplexor. The inputs were sign-extended to make them from 5 to 10 bits and were then made the input of the multiplier core. No modifications to the unsigned multiplier core needed to be done since the input is already in 2C. Therefore, the numbers can be multiplied the same way as unsigned integers. The output was once again sign-extended to be 32 bits.

E. Exponential

The Exponential function also utilized the IEEE.NUMERIC_STD.ALL library. Using the functions described earlier, the inputs were converted to integer, the exponent calculation was performed, with Input 1 as the base and Input 2 as an unsigned, 5-bit exponent, and the result converted back into a signed 32-bit std_logic_vector. However, some of the outputs were bigger than 32-bits, and raising negative bases to odd-numbered exponents caused the calculator to fail. Outputs bigger than 32-bits cannot be displayed on the Nexus A7 in hexadecimal format. Additionally, it was assumed that Vivado had difficulties in raising negative bases in integer format to odd exponents. To remedy these issues, a large if statement checking whether the inputs lead to outputs bigger than 32-bits and an error output was created, and $(\text{Base})^{(\text{odd number})}$ was changed to $(\text{Base})^{(\text{odd number} - 1)} \times (\text{Base})$. These fixes resolved all issues.



```
architecture Behavioral of Exponent is

signal A: integer;
signal Output_integer: integer;

begin

A<= to_integer(signed(Base));


Process (Base, Expnt, A, Output_integer)
begin
Error <= '0'; Output_integer <= 0;
If expnt ="00000" then

If base="00000" then
output_integer<=0;
Error<='1';
else
Error<='0';
output_integer<=1;
end if;
elsif expnt ="00001" then
output_integer<=A;
Error<='0';
elsif expnt ="00010" then
output_integer<=A**2;
Error<='0';
elsif expnt ="00011" then
output_integer<=(A**2)*A;
Error<='0';
elsif expnt ="00100" then
output_integer<=A**4;
Error<='0';
elsif expnt ="00101" then
Error<='0';
output_integer<=(A**4)*A;
elsif expnt ="00110" then
Error<='0';
output_integer<=A**6;
elsif expnt ="00111" then
Error<='0';
output_integer<=(A**6)*A;
elsif expnt ="01000" then
if (A>14) or (A<-14) then
output_integer<=0;
Error<='1';
else
Error<='0';
output_integer<=A**8;
end if;
elsif expnt ="01001" then

if (A>10) or (A<-10) then
output_integer<=0;
Error<='1';
else
Error<='0';
output_integer<=(A**8)*A;
end if;

elsif expnt ="01010" then

if (A>8) or (A<-8) then
output_integer<=0;
Error<='1';
else
Error<='0';
output_integer<=A**10;
end if;
```

```
architecture structural of exponent is

signal exp_us : unsigned(N-1 downto 0);
signal exp_int : integer;
signal A_int: integer;
signal P: integer;

begin
exp_us  <= unsigned(exp);
exp_int <= to_integer(exp_us);
A_int <= to_integer(signed(A_input));
--a0: for i in 0 to exp_int-1 generate...
p<=(A_int)**(exp_int);

P_out<=std_logic_vector(to_signed(p,8));
end structural;
```

Figure 2: Two snippets of the new code taking into account the size limit of 32-bits (on the left) against the old code (on the right) for comparison.

F. Absolute Difference

Absolute difference between two signed numbers can be calculated using the formula $|A - B|$. This was implemented also using the library IEEE.NUMERIC_STD.ALL. A was subtracted from B, and the abs() function was performed on the result of the simple difference. The main command for the conversion to integer and then applying the operator is given below:

std_logic_vector(to_signed(abs((to_integer(signed(Ax)) - to_integer(signed(Bx)))),32)).

G. Division

To implement the division operation, the same library and datatype conversion functions mentioned above were utilized. In addition, two different functions were utilized. One of them was rem(), which return the remainder of the input data, and the other one was (/), which returned the quotient. Also, a Div_Error output was created in order to account for division by zero.

H. Store and Display Function

The store function was created using Dr. Llamocca's N-bit Parallel access (right/left) shift register with enable and synchronous clear - Structural version [1]. The register's function was reduced to only a Parallel access register by deleting all lines, inputs, and outputs pertaining to the shifting operation and the synchronous clear. When the store function is enabled, the output is 'loaded' into the component. When it is disabled, then the store function will maintain its output till the next time it is enabled. The store function and the counter share the same eight millisecond clock cycle because it was convenient to do so. Also, it is impossible for humans to flip the enable switch in that amount of time. The store function does not have the ability to store the remainder outputs of division, but only the quotient.

I. 32-bit Multiplexor

Multiplexors are generally used to forward a certain input to the single output at a time. In this project, a 32-bit multiplexor is used to select which operation is to be displayed on the display. A 32-bit 8-to-1 multiplexor has eight 32-bit input buses, a 3-bit select line input, and a single output of 32-bits. A case operator is used to implement the design of the multiplexor in this project. The basic function of a multiplexor is that on the basis of certain data at the select line, a certain input is selected at the output. It acts like a multi-pole single through. This calculator only had seven operations, so one of the buses of the multiplexor had a "0x00000000" input. The selection of '111' on the multiplexor is considered an error, and this would be handled in the binary-to-hexadecimal converter.

J. Modified Binary-to-Hexadecimal converter

Generally, the Binary-to-Hexadecimal converter converts values by grouping four binary values together, which would show as one nibble on the 7-segment display.

However, it was desired to display the signed remainder of division to display a more accurate number on the 7-segment display. Division by zero is also not supposed to give a result, so the word 'Error' needs to be displayed on the 7-segment display. In addition to the aforementioned issues, some results in the exponent component consisted of more nibbles than the eight 7-segment displays available, and since there were seven operations in this calculator using a multiplexor with a 3-bit select, there was one unwanted combination for the select.

To display the division result with the quotient and the remainder, a conditional statement was used using an input from the select of the multiplexor and another input from the remainder of the division. The decision to use a conditional statement was taken since the following operations would only take place when the division operation is selected. The quotient and the remainder of the division were directly converted to the form that is recognized by the seven-segment displays. Since division by integers would result in smaller numbers, two 7-segment displays were turned off. An 'r' was also coded between the result and the remainder so that they would be separate.

The other issues are in the same category; that the result cannot be displayed. Therefore, the word 'Error' would be displayed whenever any of the other scenarios occur, so they became part of the same conditional statement as the one in the previous paragraph. The output for all of these 'special cases' were set to be the values required for the 7-segment display to show 'Error.' The output of division and the other 'special cases' in this binary-to-hexadecimal converter were set to be the output without any other modifications.

During the design process, it was envisioned that there would be a separate 'number checker' that would check whether the size of the result is smaller than eight nibbles, or 32 bits, then there would be an output from the 'number checker' signaling whether the result has more than 32 bits or not. However, checking the number of bits would only be required for the result of the exponential, so the process of checking the number of bits was moved inside the exponent component. As a result, the 'number checker' became redundant and was scrapped.

K. 7-segment display serializer

The general format of the 7-segment display serializer is adapted from Dr. Llamocca's 7-segment serializer [1]. However, some modifications took place in the Hexadecimal-to-7-segment decoder. The division and Error cases would pass through the decoder without any changes to the input. However, the decoder would work normally under other circumstances. This was accomplished using conditional statements. The serializer was also modified so that it can handle eight 7-segment displays.

In simulation, when division was selected, the output of the serializer was showing a dash in some cases. Upon further inspection, it was found that the input that went into the decoder was having the correct value, but the output had the unexpected result. This was because a single bit input that was used to determine whether division was occurring (labelled 'Check for Division operation' in Figure 4) was not being used in the conditional statement for division.

III. Experimental Setup

The solution was devised and was designed successfully in the implementation phase. The dip switches were used to input the data and all the relevant information for the task. That includes two 5-bit numbers and a 3-bit select line for choosing the desired operation of ALU. Select lines are the input of MUX which selects which input will be given on the output. The data was processed by the operators as their instances were called in the top model and data of all the operations were available on the mux input side. The select line would decide which function was selected and that data will go on the output. The seven segments on the board were

attached to the output by means of the decoder to show the actual result.

IV. Results

The 5-bit signed calculator worked as expected after all the issues mentioned in the previous sections were resolved. The user would enter two 5-bit signed binary numbers that are input 1 and input 2. This would be done by turning the switch on for the bit to be given a value of 'one,' and turning the switch off for the value to be a 'zero.' Then, the user has to select the desired math operation using the 3-bit select. '000' would select addition, '001' would select subtraction, '010 would select multiplication, '011' would select exponent, '100' would select absolute difference, '101' would select division, and '110' would display a stored result. If the user selects '111', the word 'Error' would be displayed on the seven-segments since it is not supposed to be used and is not programmed to do an operation.

The result would be displayed on the seven-segment displays in hexadecimal. If the user wants to store a result that they would want to display later, the enable switch would need to be turned on, and then off. If the user wants to clear the stored result, they would need to turn the reset switch on and then off. The order of the switches on the board and the bit placement of the inputs on the board, as well as a link to the demonstration of the calculator functioning is in the captions for Figure 3.

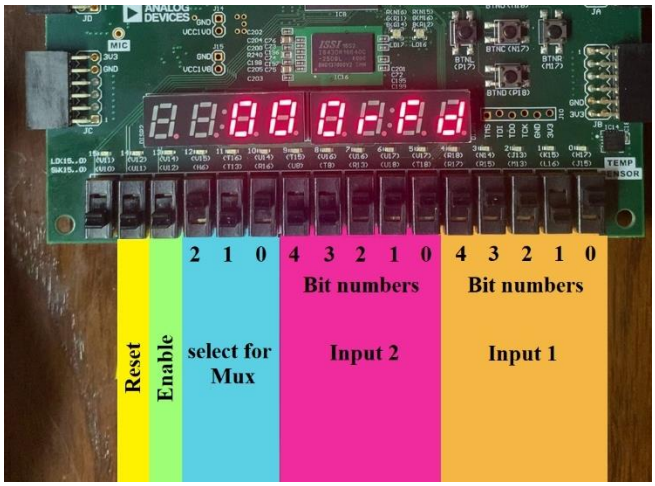


Figure 3: The bit arrangement of the inputs used. For a demonstration of the calculator, please visit this link: https://youtu.be/WcpUoIuIUdQ.

One unexpected result on the board was that the division by zero was not showing the word 'Error' on the seven-segment display. Initially, some inferred latches were found but fixing those did not solved the problem. It was discovered that an output that signaled to the serializer that division was selected by the user was not set to the correct value when dividing by zero and this was fixed. Another unexpected result was that the seven-segment on the board displays showed dash as the sign of the remainder when it was positive instead of a zero. When reading the binary to hexadecimal converter code, it was found that the segments in the display that were supposed to '1' were '0' and those that were '0' were '1.' Since the 7-segments are active low, the value '1111110' would show a dash on the 7-segment display. This was rectified to '0000001' so that a zero would appear.

Conclusions

The assigned task was successfully implemented on the FPGA Nexys-A7 board. The output was shown on the seven-segment available on the board. The input data is controlled by the dip switches and operations are also selected by the dip switches. In implementing the task hands-on learning is done to use the dip switches, display, and coding style. This can help in implementing the different tasks on the board.

Despite making a calculator that is unique in many ways, there are plenty of improvements that could be done. One of these could be adding support for negative exponents or taking roots, as that will allow the calculator to be used in more advanced settings, like scientific research and higher-level mathematics courses. Another suggestion is adding the ability to use constants like π and e in calculations, as this will allow modelling exponential populational growth or decline.

References

[1] D. Llamocca, VHDL Coding for FPGAs. [Online]. Available: http://www.secs.oakland.edu/~llamocca/VHDLforFPGAs.html [Accessed 6 December 2020].

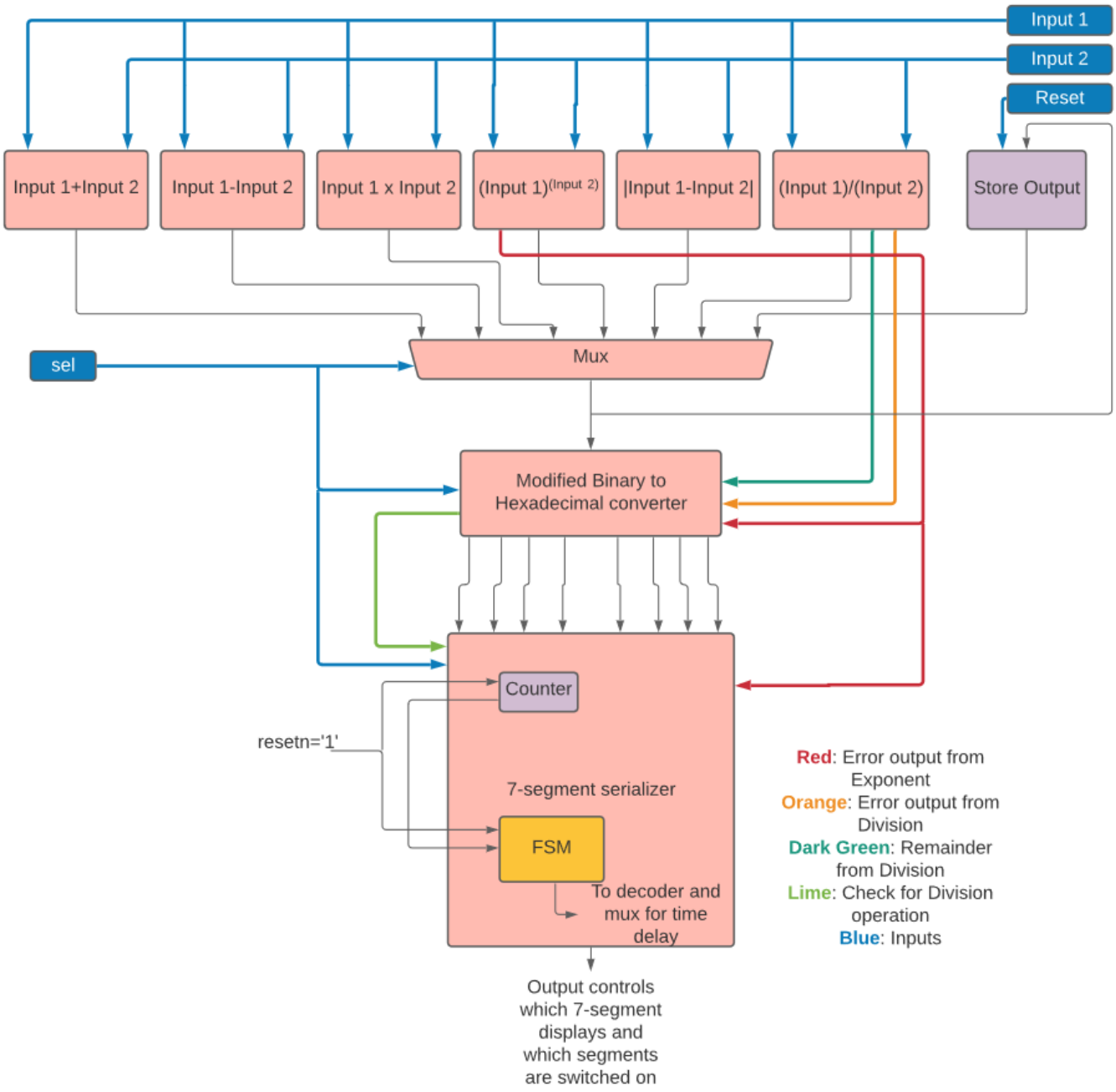


Figure 4: A block-diagram of the calculator, including the Control (in gold), and the Datapath (in various other colors). The counter and the store function share the same clock input (not pictured).