UCD School of Electrical and Electronic Engineering

EEEN30190 Digital System Design

Calculator Design Report

Name: Daniel McManus **Student Number: 16337336**

Working with: Oluwademilade Oke **Team: 10**

I certify that ALL of the following are true:

1. I have read the *UCD Plagiarism Policy* and the *College of Engineering and Architecture Plagiarism Protocol*. (These documents are available on Blackboard, under Assessment.)
2. I understand fully the definition of plagiarism and the consequences of plagiarism as discussed in the documents above.
3. I recognise that any work that has been plagiarised (in whole or in part) may be subject to penalties, as outlined in the documents above.
4. I have not previously submitted this work, or any version of it, for assessment in any other module in this University, or any other institution.
5. I have not plagiarised any part of this report. The work described was done by the team, but this report is all my own original work, except where otherwise acknowledged in the report.

Signed: Daniel McManus **Date:** 3 December 2018

Introduction

The following report details the design process and verification for the hardware of a simple calculator. The type of calculator designed is an Algebraic calculator. It displays numbers using 2's complement. It uses LED's to indicate when overflow occurs or when an integer being displayed is negative. The calculator is able to add, subtract, multiply and determine the square root of a number.

Calculator Functionality

The functionality we chose to use was that of an Algebraic Calculator. We decided to use this approach because we were much more familiar with the ins and outs of the Algebraic calculator design in terms of how the user interacts with it and felt that this would benefit us more to design in the lab.

The calculator designed has a 4-digit display and contains an Overflow LED to indicate if calculations spill over the 4 integers we are able to display. The calculator design also has an LED to indicate whether a number being displayed is negative or not, in order not to alter a Display Interface we designed in previous lab sessions.

The calculator is capable of computing several types of operations.

The types of operations the calculator can compute are as follows:

1. Addition
2. Subtraction
3. Multiplication
4. Squaring
5. Change of Sign

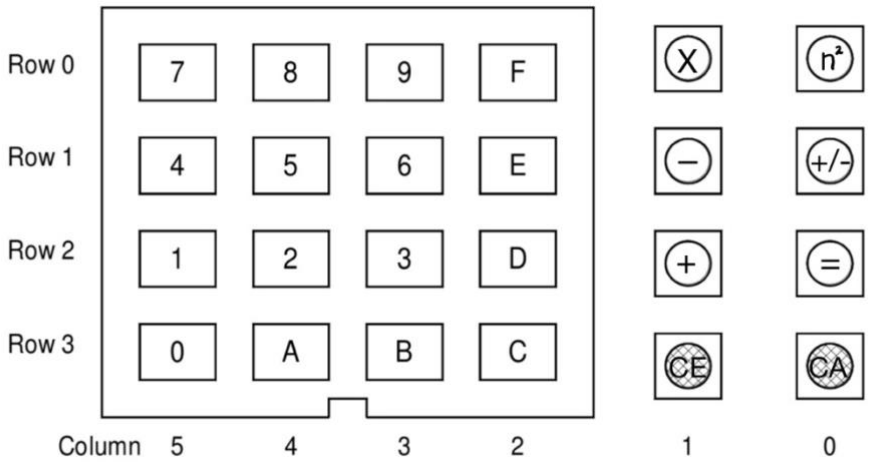
To enter a negative number into the calculator, the user must first enter their digit and then press the Change Sign button. If the user changes his mind whether he wants the number to be negative or not he can simply press the button again and continue with use.

In order to do new calculations on this calculator, the user must press either Clear key in order to clear the X register and the number being displayed. However, if the user wishes to work with the answer of the previous calculation, they can continue to do so by pressing the next operation key needed. This design does not allow for direct chain calculations, i.e. you are unable to do a calculation by entering $a + b + c =$. You must do this type of calculation by using the Equals Key. $a + b = + c =$.

Implemented into this design is a Clear Entry and a Clear All button. These buttons do exactly what it says on the tin, Clear Entry clears what is currently displayed on the screen while Clear All clears everything on the memory in the calculator.

Again as mentioned previously, in this design we must press the Clear All or Clear Entry key in order to compute a new calculation, unrelated to the last one computed.

The following figure shows the button mapping used in this design.

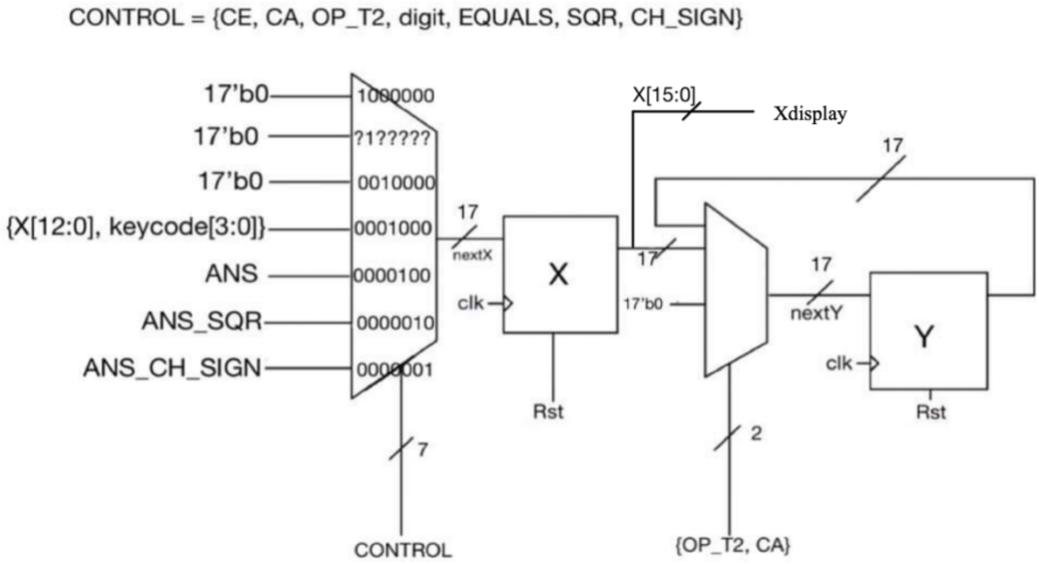


The LEDs used to display the Overflow Warning and Negative Integer on the Nexys 4 board are LED0 and LED1 respectively. l

Calculator Hardware

The design of this calculator is very complex to be able to draw out and represent in one large RTL diagram, therefore each aspect of the design will be discussed individually alongside small RTL diagrams.

The X and Y Registers



First to be discussed was one of the more challenging parts we found in the design at the beginning, appears as the simple registers and multiplexers that control the X and Y registers.

The X and Y register contains 17 Flip-Flops each, and react on the positive edge of the clock. The input of the X register is nextX, which is controlled by a multiplexer.

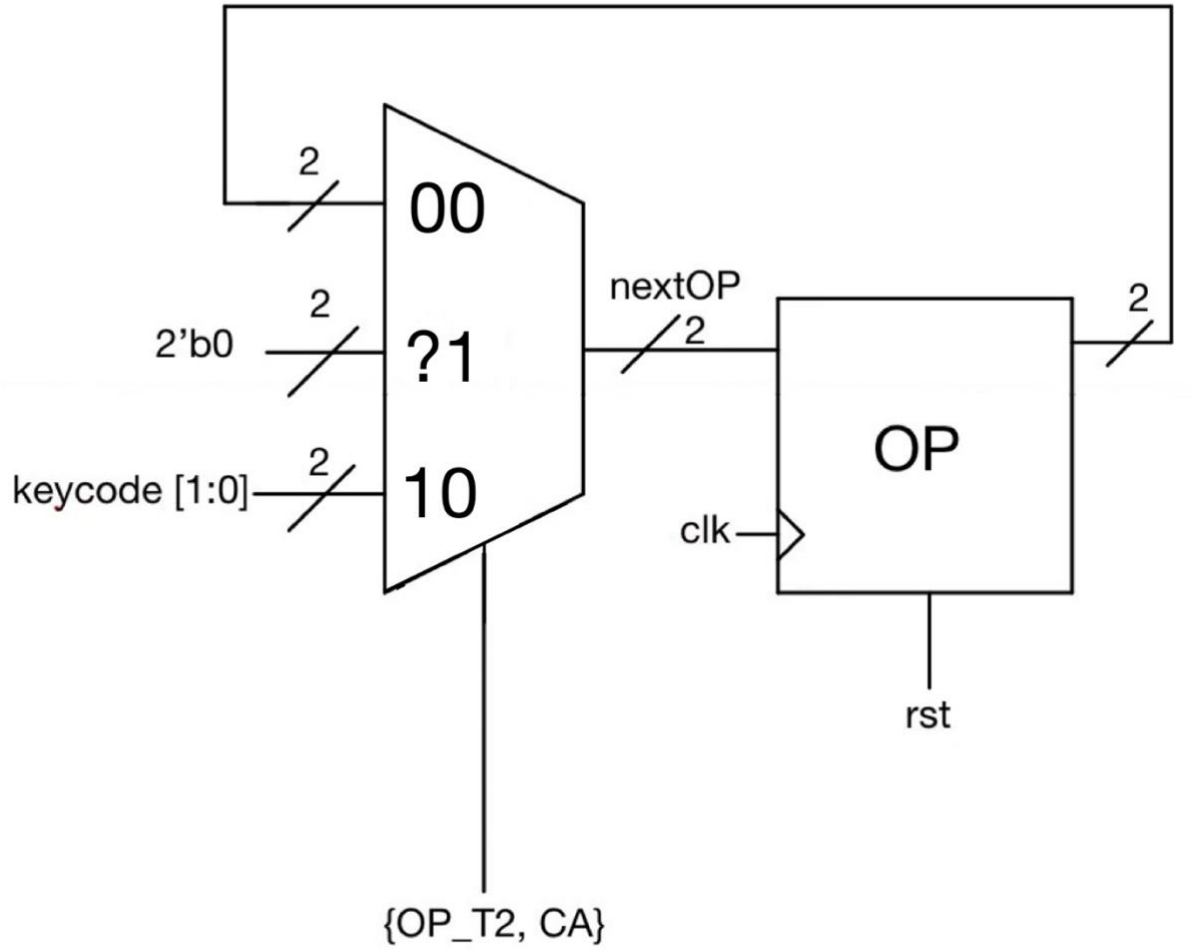
- When the Clear Entry, Clear All or when a Binary Operation is pressed (+,- or x), 17 0's are inputted into the X register. 16 of these bits are the 4 Hexadecimal digits and the 17th is to mark a negative number.
- When a digit is being entered, the multiplexer will output: the 12 right-most bits, shifted over 4 bit places and then in place of the 4 right-most bit places available, the 4 right-most bits of the keycode corresponding to a hexadecimal number.
- If the equals key is pressed, the answer of the calculation being computed is inputted into the X register
- If the Square key is pressed, then the square of what is in the X register is inputted into the X register.
- And similarly, if the Change Sign button is pressed, then the negative number of that contained into the X register is outputted of the multiplexer, and then inputted into the X register.

We then have the Y register, which changes based on a multiplexer which is controlled by when a Binary Operation is pressed (+,- or x) and when the Clear All button is pressed.

- When the Clear All key is pressed, all bits in the Y register are set to 0.

- When a Two Tier Operand signal is generated, and the Clear All key isn't being pressed, the X register will be inputted into the Y register.

The OP Register

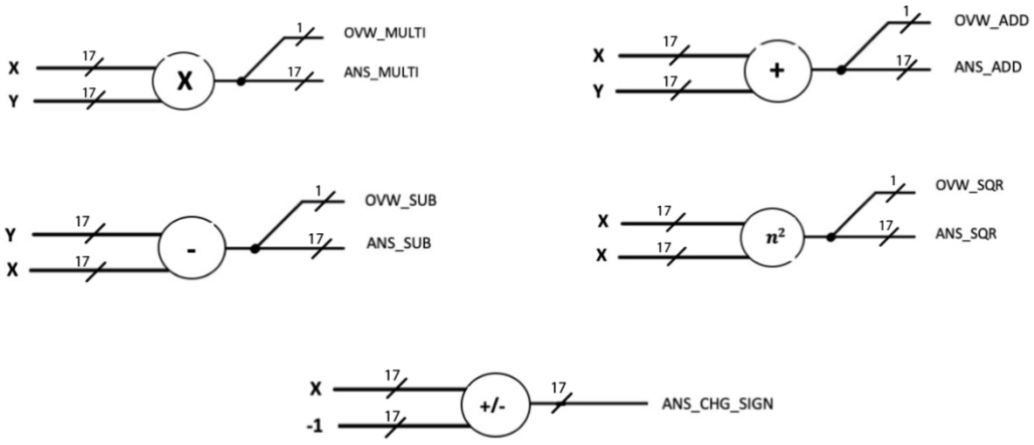


The OP Register keeps track of what Operation is currently occurring, once it is a Binary Operation. It is a two-bit register.

The input of the OP Register is the two bit output of a multiplexer, called nextOP. The multiplexer that outputs the input for OP is controlled by two signals, OP_T2, and CA. OP_T2 is a signal that is high when a Binary Operation key is pressed. CA is a signal that goes high when the Clear All button is pressed.

- When neither of the signals are high, OP is the input for the OP register.
- When the Clear All button is pressed, 0 is inputted into the OP register.
- If the OP_T2 signal is high and the CA signal is low, then the last two bits of the keycode of the Operation key being pressed is inputted into the OP register.

Mathematical Operations



There are 5 different types of operations that the calculator designed can compute. It can add, substract, multiply, determine the square of a number and also change the signage of a number.

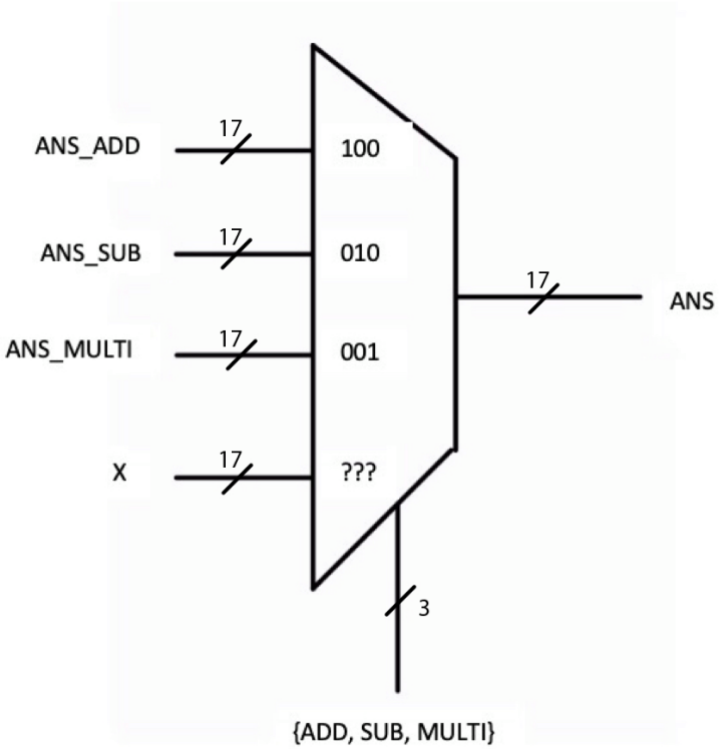
We can see included is the RTL for such tasks. All of these options are computed when we input operations into our calculator, we just choose to show the desired result, which we'll go into more detail later.

Doing multiplication is a simple matter of taking in the 16 bits of the X and the Y register, multiplying them together and storing them as a 17-bit result. 16 of these bits are the ANS_MULTI signal, and the other one bit is the rightmost bit, which is to be saved as the OVW_MULTI signal. This method allows us to easily identify occuring overflow in calculations.

For addition and subtraction, we have similar operations. We take in the bits from the X and Y registers, send the answers to a signal to store there respective result, and then split it between an ANS signal and an OVW signal, in order to determine if the calculation has overflown.

We can also see how for squaring an integer, we only need one register. This is the same approach taken for changing the sign of an integer. We use the X register alone for these calculations and will for squaring, multiply the X register by itself in order to obtain the result, in the exact same approach as we do for multiplying. We will do this in a similar way for Changing the Sign of a number, but in this case the X register is multiplied by a constant, -1.

The Answer Multiplexer



We previously mentioned how we input a signal ANS into the X-Register, and how we also calculate multiple operations, store all the results, and will only show on the display the result the user requests. The Answer Multiplexer is what helps us achieve this.

We see above a multiplexer that has 17-bit signal inputs, ANS_ADD, ANS_SUB, and ANS_MULTI. It also includes a 17-bit signal X. X is a default for the multiplexer, incase of some error in the calculator that it will still display what is in the X register.

The other 3 signals entering the multiplexer are output signals from the hardware that does addition, subtraction and multiplication. The control signal for this multiplexer in a concatenation of the control signals ADD, SUB and MULTI. These control signals become active due to the button press of their respective operation keys.

From the press of one of these keys, the desired signal will be outputted as the ANS signal to be used in the multiplexer for nextX.

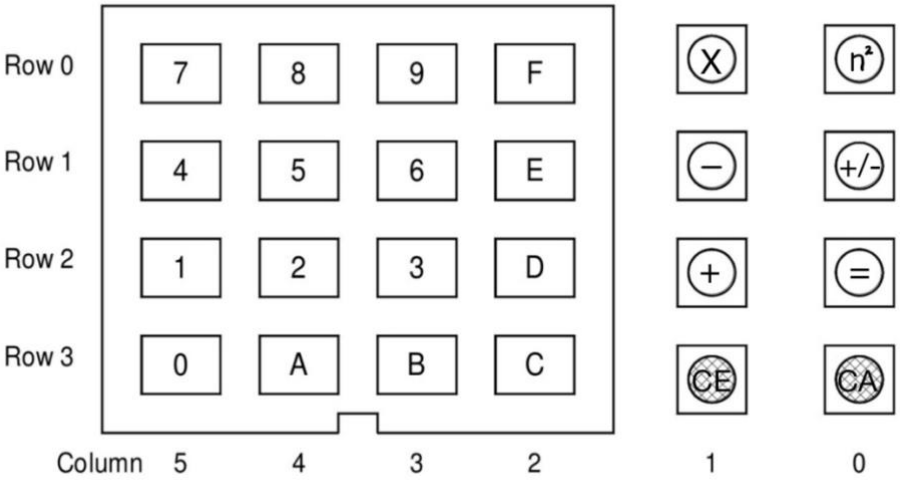
If the operation result required isn't in this register, it goes straight to the nextX multiplexer.

Mapping Keys

For simplicity and for the purpose of making the Verilog easier to write, we created a local parameter with names of all the keys corresponding to their binary values being read in from the keyboard. This made the Verilog a lot simpler to write and it also benefits of making it easier to understand when reading the source code.

```
localparam[4:0] KEY_CE = 5'b01100,
                KEY_CA = 5'b00100,
                KEY_MULTI = 5'b01001,
                KEY_SUB = 5'b01010,
                KEY_ADD = 5'b01011,
                KEY_SQR = 5'b00001,
                KEY_CH_SIGN = 5'b00010,
                KEY_EQUALS = 5'b00011;
```

The source code above from the Verilog file shows the local parameter created to identify all of the buttons that are not integers. This mapping corresponds to the diagram included earlier of what operations were assigned to individual keys.



Control Signals

```
wire ADD = (OP == KEY_ADD [1:0]);
wire SUB = (OP == KEY_SUB [1:0]);
wire MULTI = (OP == KEY_MULTI [1:0]);
wire OP_T2 = (newkey && (keycode[4:2] == 3'b010) );
wire CA = (newkey && (keycode == KEY_CA) );
wire CE = (newkey && (keycode == KEY_CE) );
wire digit = (newkey && (keycode[4] == 1'b1) );
wire EQUALS = (newkey && (keycode == KEY_EQUALS) );
wire SQR = (newkey && (keycode == KEY_SQR) );
wire CH_SIGN = (newkey && (keycode == KEY_CH_SIGN) );
wire [6:0] CONTROL = {CE, CA, OP_T2, digit, EQUALS, SQR, CH_SIGN};
```

We used multiple control signals throughout this design to simplify the whole process. Outlined above is an extract from the Verilog used to create these signals. You can also see from this extract the conditions in which the control signals are active high. There is a control signal for each type of process, such as ADD, SUB, and MULTI being for when we wish are doing addition, subtraction and multiplication, respectively. We can also see that there is a signal OP_T2, which identifies when an operation will require the X and Y register, or just the X register alone.
A significant control signal used in this design was CONTROL, which is a simple concatenation of 7 individual control signals in order to simplify the control of the nextX multiplexer.

Verification

Verification Plan

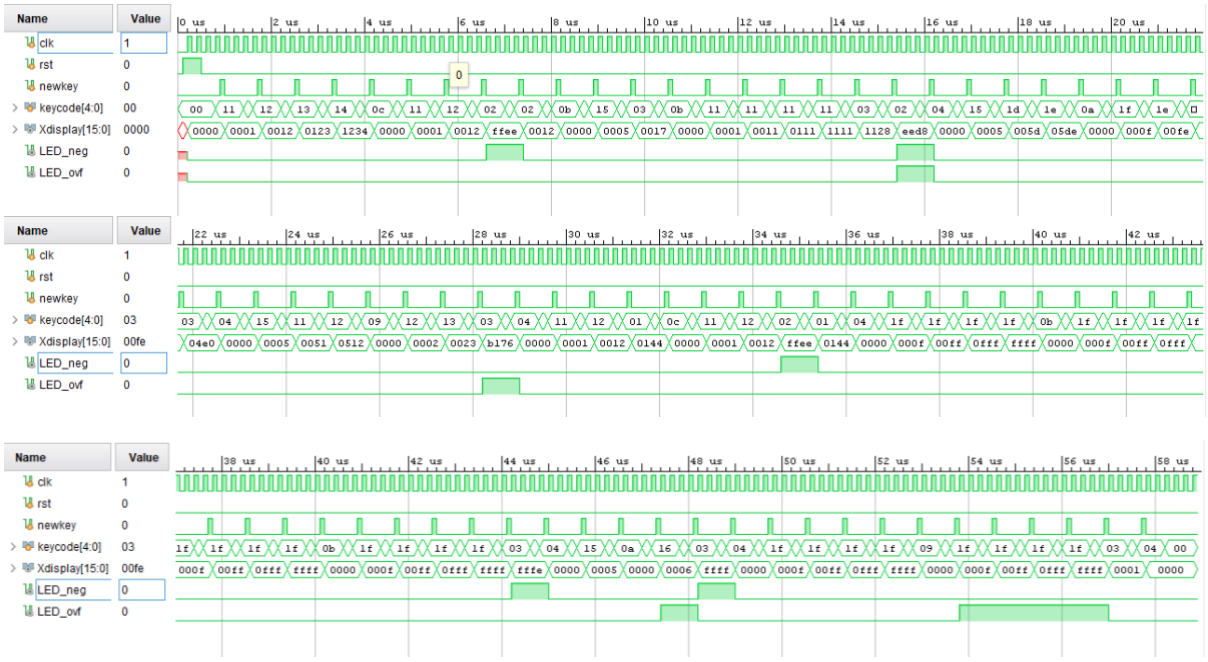
A plan was designed that would outline the specification of this assignment, and to also test the other features that we included along the way. The general plan is outlined as follows:

1. First thing for all testbenches is we must reset the hardware.
2. We show that we can input a number
3. We clear the X Register, and thus the display by pressing Clear Entry
4. We input another number and demonstrate the Change Sign feature
5. Now to demonstrate addition
6. Test chain calculations are possible
7. See if we can change the sign of an answer
8. Test Clear All works correctly
9. Test multiplication
10. Test Subtraction
11. Test negative integers
12. Test squaring
13. Test the square of a negative integer
14. Test overflow in addition, subtraction and multiplication

Testbench

After deciding on our plan for our verification, we set out by writing down the actual figures we were going to use in the steps of our verification, and the results that we would expect with those figures. When we did this, we began by writing it into our testbench. We were provided with a sample testbench by our lecturer and with the aid of this implemented our strategy. The testbench implemented was very long so we decided to shrink the timing of signals down small enough in order to be able to show our full simulation on one page.

Below are the results of our Testbench when we ran our Behavioural Simulation in Vivado.



We could see in this testbench that our display was working out quite well, however there was a few things we wanted to change but ran out of time in lab the sessions. We can see that the display will

display just 0 when we press a Binary Operator. We believed we fixed this in an earlier session in a lab using a simpler testbench at the beginning, but when we began to work with a properly structured testbench soon realised our attempt at solving this wasn't enough and we had to revert to what we have shown before you.

It was necessary for this assignment to implement our design on hardware and present it working. In Vivado we ran a Quick Flow implementation of our design and then downloaded the bit stream onto the Nexys board. We followed a similar approach to our Testbench verification and tested all aspects of the design such as Chain calculations, Addition, Subtraction, etc.

It was noted in the testbench and implementation that the Negative and Overflow LED's didn't work as required in this design. They would work in some cases but in others they would simply fail. Unfortunately, we had ran out of time to return back to the design stage and work out a new approach in which they would work. It was also noted with deep regret that part of the specification was entirely missed, the display of negative integers. In our design, we display 2's complement values, but as part of the specification it is required to display the magnitude of negative values for an easier user interface.

Synthesis and Implementation

Synthesis Warnings

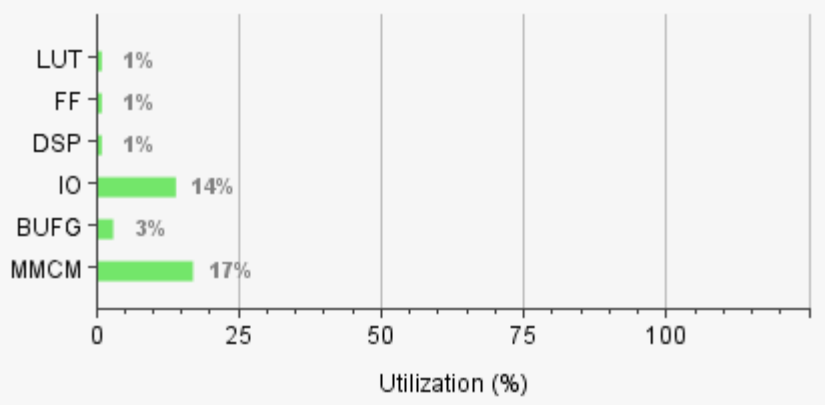
During Synthesis of the hardware we receive 4 warnings relating to our Verilog.

```
[Synth 8-3917] design calculator_top has port digit[7] driven by constant 1
[Synth 8-3917] design calculator_top has port digit[6] driven by constant 1
[Synth 8-3917] design calculator_top has port digit[5] driven by constant 1
[Synth 8-3917] design calculator_top has port digit[4] driven by constant 1
```

These errors are in relation to the Display Interface instantiated in the top-level module. Digit[7:4] are set to a constant 1 to keep the four left-most 7-segment displays off. This warning may be ignored as we know this was intended and part of our design, and thus won't affect the results of our hardware.

Hardware Implementation

The implementation ran in Vivado was a Quick Flow implementation. This means that it is essentially a "rough draft" of how the hardware could be ran on the FPGA, essentially just for verification purposes. When implementing a "final draft" of the hardware, one would run an implementation of an optimised design. This could be either less hardware used or a faster system.



For this assignment we stuck with only running a Quick Flow. In implementation 224 Look Up Tables (LUTs) were required as logic. Vivado warns us that this is indeed a large final LUT and a full implementation tends to be lower. However, it may be noted that this is also significantly less than 0.5% of the LUTs provided on the chip (values are rounded in the chart) so it doesn't pose a huge problem in this case, and we must keep in mind as this isn't a full optimisation of the design that the total LUTs required will be less than what we have now.

Vivado reports that we have used 2 DSPs. This is possibly due to the hardware used in our Verilog to multiply the X and Y registers and to compute the square of the X register.

Vivado reported that this assignment acquired 86 flip-flop registers which accounts for 0.07% of which we have at our disposal using the Artix-7 FPGA chip. These 36 of these are used for the design of our calculator logic, 17 for each the X and Y registers, and 2 for the OP register.

Hardware also used in this design is 30 Bonded IOBs. These are the inputs and outputs required to connect the FPGA with the Nexy's board. We have also used a BUFG and MMCM. These are required for the Clock Module provided to us that supplied us with a 5MHz signal.

Timing Analysis

Our calculator uses a 5Mhz clock which gives us clock periods of 200ns. The Worst Negative Slack computed by Vivado is 191.36 ns. The Worst Hold Slack is 0.018ns and the Worst Pulse Width Slack is calculated at 3 ns. While we do have a high Worst Negative Slack, it does meet our requirements and our hardware has plenty of time to be ready for the next clock cycle. It also may be noted that due to the speed of the hardware, a user wouldn't notice any slow hardware at this stage as if it required

perhaps 1, 2 or even 3 clock cycles to display an answer, it would still be too quick for the user to realise.

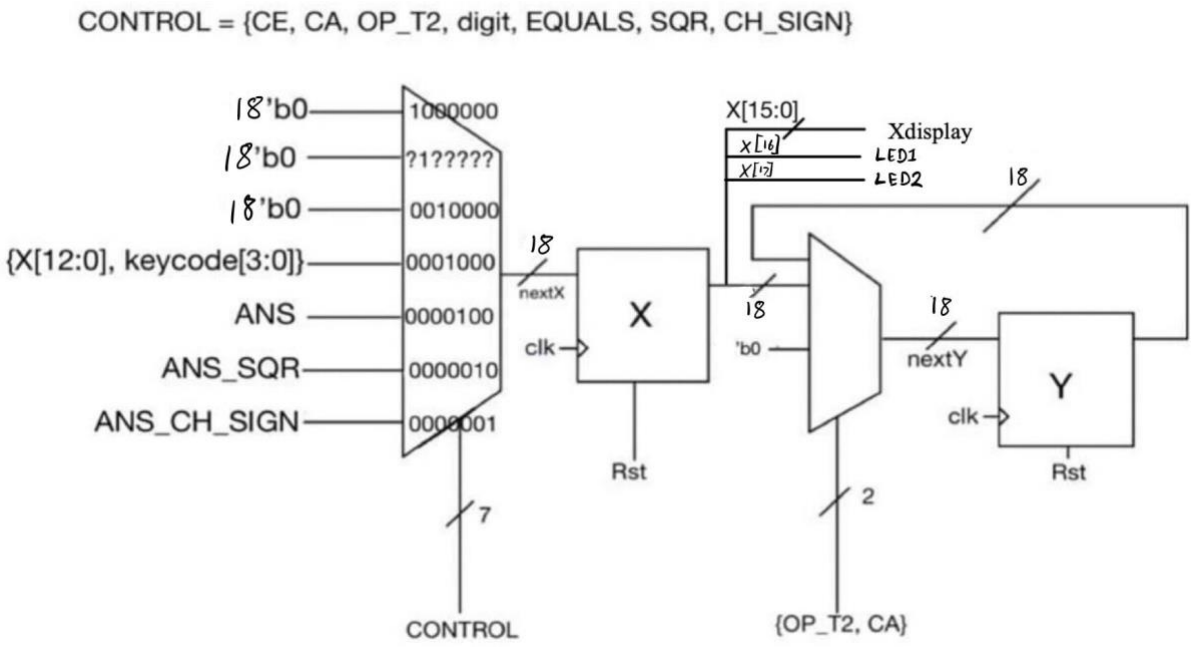
Conclusion

The design we have provided was able to compute various mathematical functions, such as computing addition, subtraction, multiplication and the square of an integer. Our design is able to compute all of these correctly. However, we did run into some issue with how it displays this. While we do use 2's complement in this design, it is hard to tell whether a number is displayed in its negative or positive form. It would be found a lot easier if we displayed values in magnitude, but considering we still have an LED light to indicate when a number is a negative number it isn't impossible. Unfortunately in our circumstance, the Negative Number LED doesn't work so it doesn't benefit the user. In light of this, we were able to come up with an effective solution to combat this issue.

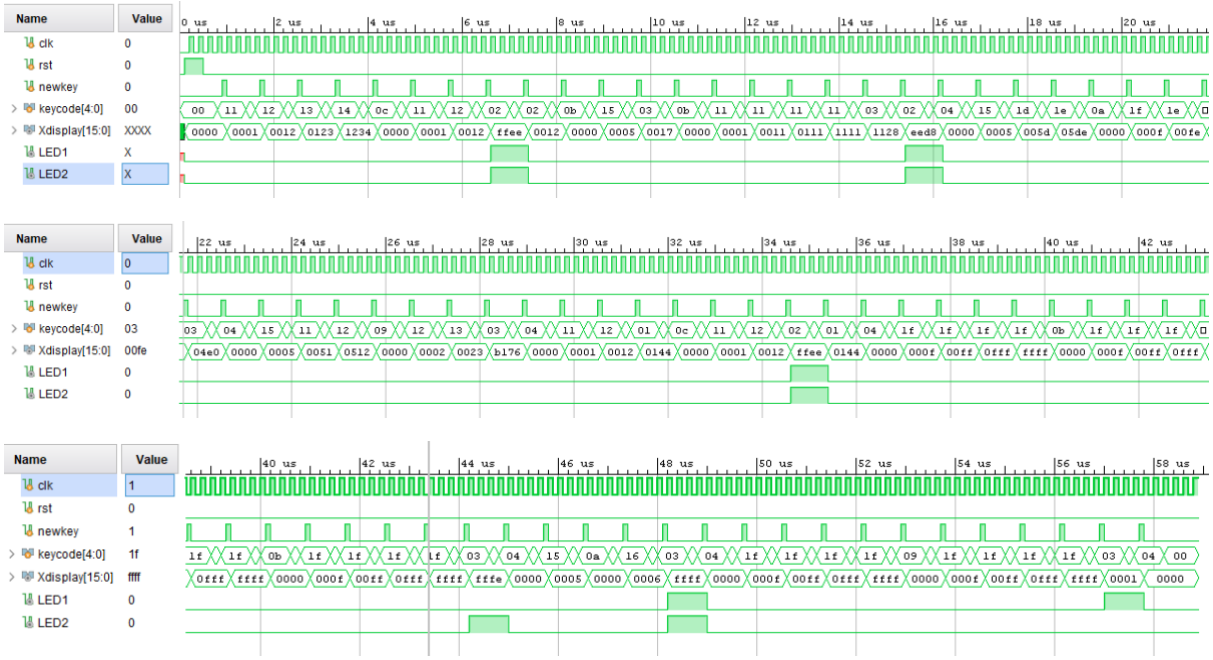
In our design we used 17-bit registers, with the 17th bit being used as an indicator for when an integer is negative. We can see how this solution is pretty crude, as it can also become active when we receive overflow.

We determined a small alteration we could make to our design in order to conclude this report. The change would be to use 18-bit registers instead of 17-bit registers. The 17th and 18th bits in the X-Register will control 2 LED's. There will be 2 active LEDs for when there is a negative integer displayed, and only 1 LED active when there is overflow occuring.

The RTL diagram would be as follows:



The changes shown in this RTL diagram give us the following results when running the Testbench we created earlier.



In comparison to the results we obtained previosuly, the alterations in our design suit the task perfectly. When 1 LED is active, we have detected an overflow, while when 2 LED's are active, we are displaying a negative integer.