Alexander White
s1813126
30/11/20

Engineering Software 3 – Assessment 2

Scientific Calculator on Board

Contents

Introduction

The objective of Assessment 2 is to design a Scientific Calculator that can operate and run on the BASYS-3 Artix-7 FPGA board. The project builds upon the framework developed in Assessment 1, for the 8-bit calculator, with added utility and a reimagined UI and code structure.

This report will explain how all the Scientific Calculator specifications were met, in addition to the extra features built on top of the basic requirements. The extra features include a startup display prompt, the option to switch from Cartesian to Polar unit display, and a user-friendly display that uses keywords to indicate which values are being output.

User Guide

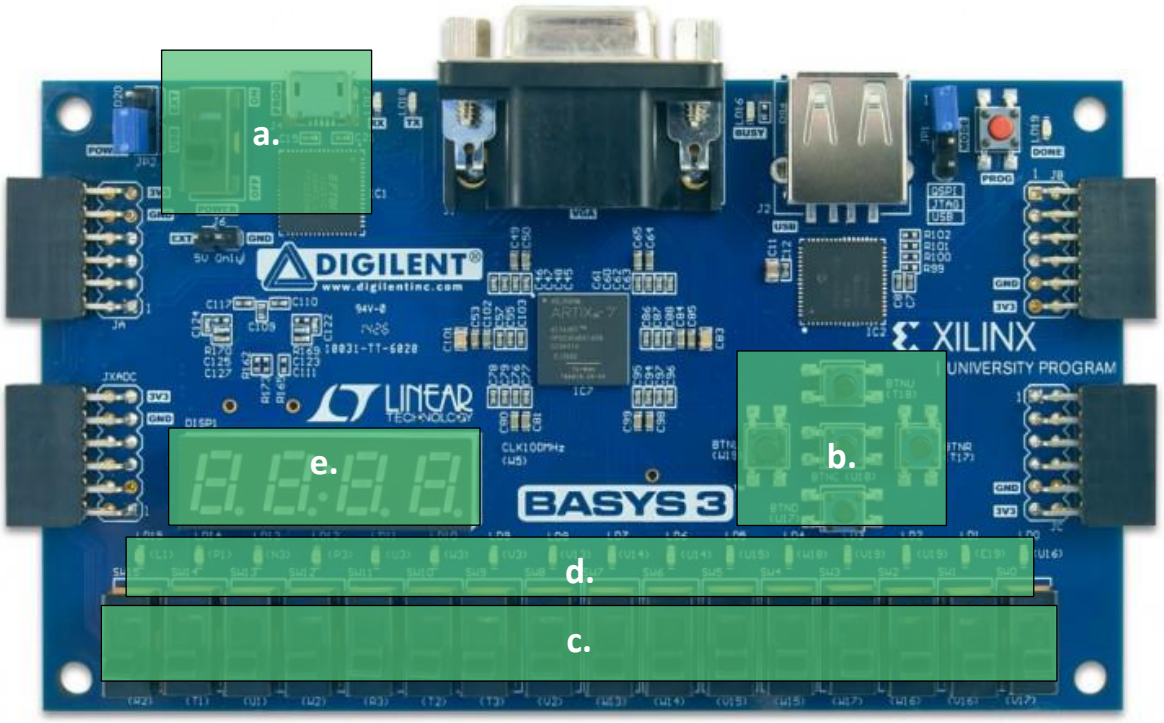

Figure 1 - Basys 3 Artix-7 FPGA

This user guide will first explain how to get the Scientific Calculator up and running on your Basys3 Artix-7 FPGA board, and then describe proper usage of the calculator step by step. Figure one will be referenced throughout the instructions, with reference to the lettered regions indicated.

Setting Up the Board

To setup the FPGA, direct your attention to **section a** of the board. Connect your board to a laptop or PC via the microUSB in the section, and flip the power switch up to the on position. The board should indicate that it's powered on by turning on some LEDs and cycling integer outputs from the seven-segment display.

After plugging in your FPGA, please extract the *Assignment2* folder submitted, open the *es3_hw_assmnt_2* folder, and run the Vivado Project File within it of the same name. If you haven't already, make sure to download Xilinx Vivado Design Suite and SDK to your PC. Ensure that you are running the 2015.2 archived version. A link with a more detailed description how to download the correct version of Vivado can be found in Appendix A.

Once Vivado 2015.2 has opened to the project, open the File tab from the top, and choose Export from the options. Proceed to Export Hardware, and make sure you select Include Bitstream before continuing. Once that is done, you should be able to open the File tab again, and Launch SDK. Once in the SDK, open the Xilinx Tools tab at the top, and select Program FPGA. The seven-segment display should turn off, but LEDs should stay on. It is then ready to receive code, and the *green arrow* to "Run Scientific Calc" can be pressed to upload the Scientific Calculator Program to the FPGA. If the words "Entr 1st num" cycle through the display, the calculator is ready for use.

Using the Calculator

This section will explain how to use the inputs and read the outputs of the calculator.

Section b in Figure 1 highlights the push buttons on the board.

> The **Right** button *Operates*, reading in an operation type from the slide switches, and performing it between the saved value output on the seven-segment display and the input value. In certain cases, such as trigonometric functions, the operator is only implemented on the saved value, disregarding the slide switch input value.
> The **Left** button *Sets*, meaning that the slide switches are read, disregarding the operation type switches, and the saved value output is *set* to the value on the switches.
> The **Down** button *Switches Output Type*, and by pressing it, the output will switch from Cartesian form to Polar form, or vice versa.

Section c covers the slide switches. The leftmost four switches are used to set the operation type. The next 6 switches are used for real number input, and the final six are used for imaginary number input.

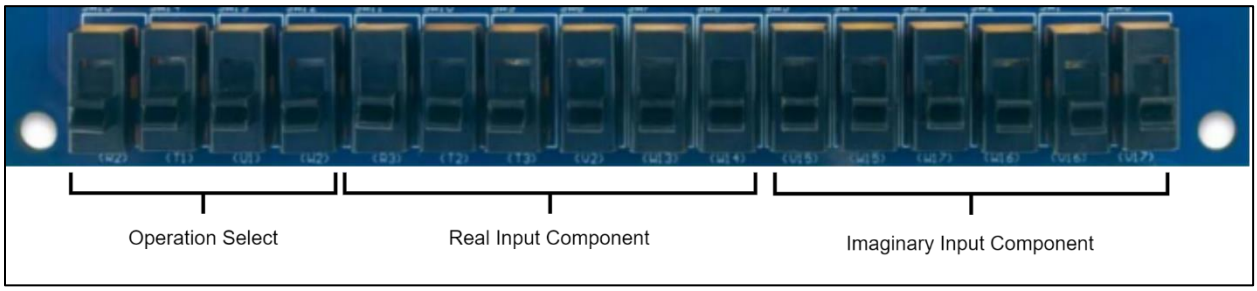

Figure 2 - Slide Switch Inputs

The MSB of each real and imaginary input controls the sign of the input, with binary 1 switching sign to negative, and binary 0 leaving the number positive. All other switches are read in binary. The possible operations are shown below, with extra operations outside of the project scope bolded.

Binary Input	Operation Output	Binary Input	Operation Output
0001	(Stored Value) + (Input Value)	0010	(Stored Value) – (Input Value)
0011	(Stored Value) * (Input Value)	0100	(Stored Value) / (Input Value)
0101	(Stored Value) ^ (Input Value)	0110	Sqrt(Stored Value)
0111	e ^ (Stored Value)	1000	sin(Stored Value)
1001	cos(Stored Value)	1010	tan(Stored Value)
1011	$\log_{10}$(Stored Value)	1100	ln(Stored Value)
1101	**(Stored Value) ^ (1/Input Value)**	**1110**	**ℝ (Stored Value) –I(Stored Value)**
1111	(Stored Value)	0000	(Stored Value)

Section d covers the LEDs. The LEDs light up when switches are turned on, allowing troubleshooting.

Section e covers the seven-segment display. When displaying numbers in cartesian form, the output will follow the pattern “(real value), Plus, (imaginary value), J, (blank)”. When displaying in polar form, the output will follow the pattern “(radius), r, (blank), (angle), Phi”. The seven-segment display output range goes from -999 to 9999, and always displays as many decimal places as possible depending on the output, ie. values such as -9.99 and 9.999 can be displayed as well.

Programmer's Guide

Overview

The state design of the Scientific Calculator was planned out roughly in a two state format - a *Start* state and an *Operative* state. As the calculator is a Mealy machine, both states operate according to their current state, switch inputs, and button press inputs.

The state graph to the right illustrates the general dataflow of the calculator. State handling is all handled in the main loop, with each state sitting within a while loop. The state code contents are mostly outsourced to functions outside of the *int main()* loop.

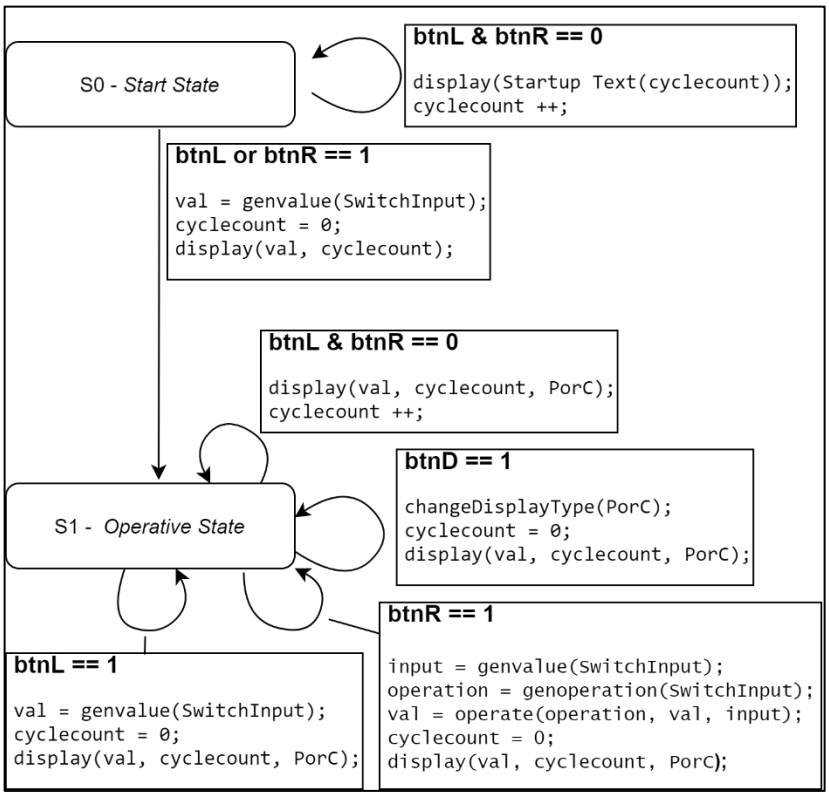

Figure 3 - State Graph of the Scientific Calculator

File Contents

The Scientific Calculator is created upon the framework developed in Assessment 1, the 8-bit calculator. This programmers guide is meant to be read in parallel with the mentioned files in the program for full understanding. The **gpio_init** files, **seg7_display** files, **platform** files, and **timer interrupt** files were all drawn from the 8-bit calculator, and changes will be discussed in this section. A **starttext** file was also created, modelled from the **instructiontext** file created for the previous assessment. This guide will not contain the C code, which can be found in the project file, but rather will explain the reasoning and design choices behind the code implemented.

main.c

The **main.c** file runs the state and variable management in the program, as well as GPIO initialization and input data processing. Ultimately, all variables in **main.c** were created as global variables, mainly to allow large chunks of code to be outsourced to individual functions that could report back critical values to main without worrying much about whether the references and pointers were properly managed. In a larger group project, this technique would not be used, and more care would be taken to declare functions with their necessary variables in the code, but in this project it was unnecessary. Six functions within the main file were created to carry out the above processes, described below.

Within *int main(),* the platform and GPIOs are initialized first. Following that, pointer variables are set to point to their appropriate locations. Then, the *S0* state is entered. The state is represented by a while loop tracking button presses to know when to exit the loop. With each cycle completed, a *cyclecnt* variable is incremented, and the **starttext.c** file function *displayStart* is called with the *cyclecnt* input. The result is that, until a button is pressed, a startup display sequence is shown on the seven-segment display. Once a button is pressed and released, the loop is exited, and the slide switches are read into the first value using

the *readinswitches* function, discussed later on. The **seg7_display.c** file is called with the function *displayComplex*, displaying the switch input to the display , and *cyclecnt* is set back to 0.

Then, *S1* is entered. The state is represented by an infinite while loop. The left, right, and down buttons are read at the beginning of the loop, and if pressed, respectively initialize the *set(), increment(),* or *switchstyles()* functions. The null state of the while loop increments *cyclecnt,* calls the *displayComplex()* function, and lights the LEDs if their respective switches are on.

The *cyclecnt* variable is set to 0 whenever it passes a specific threshold to keep the variable in an acceptable range for the functions utilizing it to accept.

The **double** complex function *readinswitches()* is called when input switch values need to be saved for operation or setting. The function returns a **double** complex value that results from combining the values shown right depending on the sign bits.

```
double complex value = 0 + 0*I;
u16 inR = (slideSwitchIn & 0x07C0) >> 6;
u16 signR = (slideSwitchIn & 0x0800) >> 11;
u16 inI = (slideSwitchIn & 0x001F);
u16 signI = (slideSwitchIn & 0x0020) >> 5;
```

Figure 4 - Reading Switches into Complex Number Value

The **void** *cartesiantopolar()* takes a complex cartesian value and generates polar coordinates from it for display if the *switchstyles()* function is called. The polar coordinate values are stored in separate variables from the cartesian variables as a safety to avoid dangerous truncation from converting back and forth if requested repeatedly. Pointers are used in to be able to modify multiple variables within a **void** function.

void *set()* is called when the left button is pressed. The function waits until the button is released, following the process shown right, to keep up LED and seven-segment display output while the button is held down, and then sets the current value equal to the slide switch input using the *readinswitches()* function discussed above. *cyclecnt* is also set to 0 so that future readouts from the display begin at their appropriate points.

```
while(pushBtnLeftIn == 1)//holds you here until button is released
{
    displayComplex(val, cyclecnt, cartorpolar, length, theta);
    slideSwitchIn = XGpio_DiscreteRead(&SLIDE_SWITCHES, 1);
    XGpio_DiscreteWrite(&LED_OUT, 1, slideSwitchIn);
    pushBtnLeftIn = XGpio_DiscreteRead(&P_BTN_LEFT, 1);

    cyclecnt++;
    if(cyclecnt>350)
    {
        cyclecnt = 0;
    }
}
```

Figure 5 - Button Hold Process

The **void** *increment()* function is called when the right button is pressed, and acts much like the *set()* function until the button is released. Then, an *input* variable is set by *readinswitches(),* and an *operation* variable is generated from switch inputs. The saved value is determined by the *operate()* function in **arithoperations.c** source file, taking in the variables *operation, val,* and *input*. Thus, an iterative *val* is generated using previous *val* and *input* variables, operated on with the *operation* specified. Following that, the *cartesiantopolar()* function is called to keep the polar coordinates up to date with the new *val.* Finally, as with *set()*, *cyclecnt* is set to 0.

void *switchstyles()* is called when the down button is pressed. It follows the same procedure as *set()* and *increment()* until the button is released. Then, a variable *cartorpolar* is switched from 0 to 1, or vice versa, thus switching the style of output. As before, *cyclecnt* is set to 0.

The last **void** function in **main.c** is the function *introloop()*, which is called in *S0.* It is purely utilized once a button is pressed, and follows the same procedure as the previous functions, but instead of

displayComplex(), the *startDisplay()* function is called to continue displaying the startup text. After the button is released, a variable *start* is set to 1, notifying the program to exit the *S0* loop.

gpio_init.c

References to three useable pushbuttons, the slide switches, the LEDs, and the seven-segment display are found in the ***xparameters.h*** header file and initialized in the ***gpio_init.c*** source file. The **gpio_init.h** header file is also updated with the new references.

starttext.c

This source file was written to contain a function *startDisplay()* in order to keep *main()* a bit more orderly. The function takes in the *cyclecnt* variable from main and chooses a word to display to the seven segment display, calling the word using **seg7_display.c** function *displayChar().* The gist of the code and message are shown right. The message is followed by a blank for 50 cycles.

```
if(cyclecnt < 50){
    displayChar(E, n, t, r);
}
else if(cyclecnt<100){
    displayChar(NUMBER_BLANK, 1, S, t);
}
else if(cyclecnt<150){
    displayChar(n, u, Ml, Mr);
}
```

Figure 6 - Startup Text

complex.h

The **complex.h** header file bears mentioning in this programmer's guide, as it was used extensively in the code. The header file allows for the implementation of complex variables, which can be written as [variable type] complex [variable name]. The header massively streamlines management of complex variables as well as arithmetic on complex variables, with many arithmetic and trigonometric functions built into the header. Due to its utility, the **complex.h** header file was included in **main.c**, **arithoperations.c**, and **seg7_display.c**. A link to the website used to learn about the header can be found in Appendix A.

seg7_display.h

This header file builds from the archetype developed in Assignment 1. It defines macros for each letter displayed, both defining their seven-segment output profile, as well as defining them to an integer that can be called in a switch case when displaying a selected letter. It also declares each function within the source file.

seg7_display.c

This seven-segment display source file is the second most involved source file following **main.c** due to the number and complexity of functions within it. Several variables are created as global variables within the file, including an array *digits[4]* which holds the digits to display, and several variables used to manage which digit from the array to display, as well as management of decimal spots and implementation.

```
u8 digitwithdec;
if((decimalspot == 0x2) && (digitNumber == 3))
{
    shouldidec = 0b01111111;
}
else if((decimalspot == 0x4) && (digitNumber == 2))
{
    shouldidec = 0b01111111;
}
else if((decimalspot == 0x8) && (digitNumber == 1))
{
    shouldidec = 0b01111111;
}
else {
    shouldidec = 0b11111111;
}

switch (digitToDisplay)
{
    case NUMBER_BLANK :
        XGpio_DiscreteWrite(&SEG7_HEX_OUT, 1, DIGIT_BLANK);
        break;
    case 0 :
        digitwithdec = shouldidec & DIGIT_ZERO;
        XGpio_DiscreteWrite(&SEG7_HEX_OUT, 1, digitwithdec);
        break;
    case 1 :
        digitwithdec = shouldidec & DIGIT_ONE;
        XGpio_DiscreteWrite(&SEG7_HEX_OUT, 1, digitwithdec);
```

Figure 7 - Decimal Management in displayDigit()

The first function discussed is **void** *displayDigit()*. First, the function decides where and if to place a decimal point in the output and creates a variable to represent the seven-segment representation of any integer and a decimal. A snippet of the code is shown in figure 7. You can see how the original switch case is altered to change the GPIO output depending on whether or not the decimal should be placed at the end of the digit. It is also of note that the switch case list is greatly expanded from the original capacity to only output integers. As stated before, new macros were added in the header file, an example of which can be seen in figure 8. Then, depending on the digit number to currently display, the digit is written to the proper seven-segment display unit.

```
case J :
    XGpio_DiscreteWrite(&SEG7_HEX_OUT, 1, DIGIT_J);
    break;
case H :
    XGpio_DiscreteWrite(&SEG7_HEX_OUT, 1, DIGIT_H);
    break;
```

Figure 8 - Expanded Switch Cases Include Letter Outputs

The **void** *displayChar()* will now be examined. It takes in four integers, representing the four inputs to display to the board. As before explained, macros for used letters are added using the header file, so such inputs as *displayChar(E, n, t, r)* can be easily interpreted to display the word "Entr" to the display. The timer ISR is watched in this function in order to keep outputs in time with the internal clock. An example is shown in figure 9.

```
case 4 :
    digitToDisplay = char4;
    digitNumber = count;
    while(digitDisplayed == FALSE);
    digitDisplayed = FALSE;
    break;
```

Figure 9 - Using the ISR to Display Characters

The **void** *displayNumber()* is somewhat similar to *displayChar()*. The difference is that the *displayNumber()* function takes in a *double* variable and handles processing internally, rather than having the pieces individually entered like in *displayChar()*. Immediately, the input number is altered to allow integer truncation to occur without sacrificing the accuracy of the displayed figure. The *calculateDigits()* function is called in order to fill the *digits[4]* array described before, and then the *digitToDisplay* variable is set to the correct place of *digits[4]*, and similarly to figure 9, the timerISR is waited for to display the digit. A case for numbers outside of the output range is also written into this funciton, displaying all dashes if an impossible number or error occurs.

void *calculateDigits()* is sent an altered *double* value from the *displayNumber()* function. *calculateDigits()* is called in order to populate the *digits[4]* array with appropriate values to send to the display, and also handles the process of deciding where to place the decimal point. Every place in the display will be utilized, meaning that a maximum of three decimal places may be displayed on the board. Depending on the magnitude and sign of the number, a different process will be carried out. Two examples are shown in figure 10, illustrating the difference in handling different magnitudes of input

```
if (number >= 1000)
{

    fourthDigit  = (int) number % 10;
    thirdDigit = (int) (number / 10) % 10;
    secondDigit  = (int) (number / 100) % 10;
    firstDigit = number / 1000;
    decimalspot = 0;
}
// Check if number is three-digits long
else if (number >= 100)
{

    fourthDigit  = (int) (number*10) % 10;
    thirdDigit = (int) number % 10;
    secondDigit  = (int) (number / 10) % 10;
    firstDigit = (int) (number/100)%10;
    decimalspot = 0x2;///we're gonna try something funky...
}
```

Figure 10 - Handling Various Magnitude Ranges

values. The double number needs to be typecast into *int* type in order to carry out the modulus operation, and the *decimalspot* variable can be observed being set into the second position of the display units, as the output would hold form XXX.X. The other case contents can be extrapolated from these examples or observed directly from opening the code.

The final function written in the **seg7_display.c** source file is **void** *displayComplex()*. This function takes in several variables in order to operate. It is called from main, and the variables *val, cyclecnt, cartorpol, length,* and *angle* are all input. This allows the function to first distinguish whether the value to display is meant to be in Cartesian or Polar form, and then display the proper output to the display depending on *cyclecnt*. In that respect, the code vaguely resembles *startDisplay(),* with the small change that some of the outputs are numbers instead of words. Figure 11 shows the case if the chosen display type is in Cartesian coordinates. A similar but slightly simpler output is used for Polar coordinates, displaying in the form ([length], "r", [blank], [angle], "Phi", [blank]).

```
if(cartorpol == 0){
if(cyclecnt < 100)
{
    displayNumber(creal(number));
}
else if(cyclecnt < 150)
{
    displayChar(P, L, u, S);
}
else if(cyclecnt < 250)
{
    displayNumber(cimag(number));
}
else if(cyclecnt < 300)
{
    displayChar(J, NUMBER_BLANK, NUMBER_BLANK, NUMBER_BLANK);
}
else
{
    displayChar(NUMBER_BLANK, NUMBER_BLANK, NUMBER_BLANK, NUMBER_BLANK);
}
}
```

Figure 11 - Cartesian displayComplex() Output

arithoperations.c

This source file was written to handle the arithmetic operations on complex numbers. A full list of the operations available can be found in a table previously in the report, under *Using the Calculator*. The singular function within the file doesn't bear much explaining, **double** complex *operate().* The function returns a **double** complex value into main for the new main, and takes in three variables, an **int** *operation*, and **double** complex values *val* and *input.* The function *switches* on the *operation* input, performing the operation specified in the table in *Using the Calculator*. It also heavily utilizes the functions included in the **complex.h**.

```
case 5:
    newval = cpow(val, in);
    break;
case 6:
    newval = csqrt(val);
    break;
case 7:
    newval = cexp(val);
    break;
case 8:
    newval = csin((val*180/3.14159));
    break;
```

Figure 12 - Operation Cases

xinterruptES3.c

This source file, combined with the short code in **timer_interrupt_func.c**, implements a system where information is sent to the display at specific time intervals. Individual digits are written to with a refresh period of 4ms, meaning that the whole display has a refresh period of 16ms. Since the *displayDigit()* function is only called by the timer code, every output follows this refresh period, with a rate of 62.5Hz.

Appendix

Appendix A – Useful Links and Sources

To download Vivado 2015.2, follow the links in the Course Information section of the Engineering Software 3 Learn page.

*The source used to implement **complex.h** header is cited below*

Avardhan, Avsadity. " Header File in C with Examples." GeeksforGeeks, 7 Apr. 2020, www.geeksforgeeks.org/complex-h-header-file-in-c-with-examples/.