# Introduction

The aim of the project is to implement a scientific calculator using the Nios II processor along with the Altera DE2 development board. The calculator must be able to read commands from the standard input and display them on the LCD of the DE2 board. Furthermore, the calculator must be able to perform a finite set of instructions; the instructions are as follows:

- Addition
- Subtraction
- Memory store and memory clear
- Sine
- Cosine
- Tangent
- Logarithm
- Exponents
- Use of the memory unit in all of the above operations

The following sections outline the details of which design goals were met (and their corresponding implementations).

# Implementation and Results

## Hardware

The steps taken in the hardware implementation primarily involved the use of the SOPC (System On a Programmable Chip) builder tool found in Quartus II. The system consisted of the Nios II processor, its on-chip memory, an SDRAM controller, and LCD driver (see figure 1).

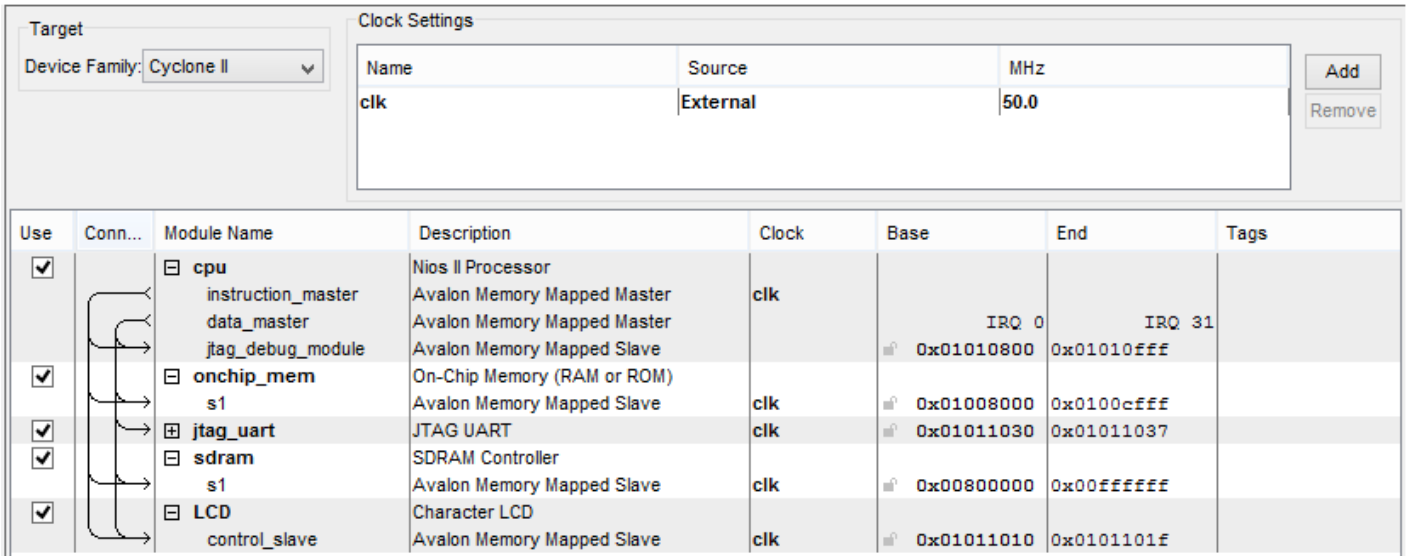

**Figure 1: The Nios II system setup inside the SOPC builder tool**

Unlike previous exercises where the on-chip memory of the Nios II processor sufficed, the program for the scientific calculator needed to be stored inside a larger memory. The solution was to utilize the 8 Mb SDRAM of the DE2 board. This was done by first adding an SDRAM controller component to the Nios II system via the SOPC builder (see figure 1) and connecting its ports. The ports were connected in the top-level entity of the design (named *Calculator.vhd*) by including the Nios II system as a component and port mapping the necessary ports in the architecture section of the code. Figures 2 and 3 show how this was performed.

```
COMPONENT nios_system
PORT (
              -- 1) global signals:
                 signal clk : IN STD_LOGIC;
                 signal reset_n : IN STD_LOGIC;

              -- the_LCD
                 signal LCD_E_from_the_LCD : OUT STD_LOGIC;
                 signal LCD_RS_from_the_LCD : OUT STD_LOGIC;
                 signal LCD_RW_from_the_LCD : OUT STD_LOGIC;
                 signal LCD_data_to_and_from_the_LCD : INOUT STD_LOGIC_VECTOR (7 DOWNTO 0);

              -- the_sdram
                 signal zs_addr_from_the_sdram : OUT STD_LOGIC_VECTOR (11 DOWNTO 0);
                 signal zs_ba_from_the_sdram : OUT STD_LOGIC_VECTOR (1 DOWNTO 0);
                 signal zs_cas_n_from_the_sdram : OUT STD_LOGIC;
                 signal zs_cke_from_the_sdram : OUT STD_LOGIC;
                 signal zs_cs_n_from_the_sdram : OUT STD_LOGIC;
                 signal zs_dq_to_and_from_the_sdram : INOUT STD_LOGIC_VECTOR (15 DOWNTO 0);
                 signal zs_dqm_from_the_sdram : OUT STD_LOGIC_VECTOR (1 DOWNTO 0);
                 signal zs_ras_n_from_the_sdram : OUT STD_LOGIC;
                 signal zs_we_n_from_the_sdram : OUT STD_LOGIC
      );

END COMPONENT;
```

**Figure 2: Component declaration of the Nios II system**

```
-- Instantiate the Nios II system entity generated by the SOPC Builder.
NiosII: nios_system
PORT MAP
(
--Global mappings
clk => CLOCK_50,
reset_n => KEY(0),
--LCD mappings
LCD_E_from_the_LCD => LCD_EN,
LCD_RS_from_the_LCD => LCD_RS,
LCD_RW_from_the_LCD => LCD_RW,
LCD_data_to_and_from_the_LCD => LCD_DATA,
--SDRAM mappings
zs_addr_from_the_sdram => DRAM_ADDR,
zs_ba_from_the_sdram => BA,
zs_cas_n_from_the_sdram => DRAM_CAS_N,
zs_cke_from_the_sdram => DRAM_CKE,
zs_cs_n_from_the_sdram => DRAM_CS_N,
zs_dq_to_and_from_the_sdram => DRAM_DQ,
zs_dqm_from_the_sdram => DQM,
zs_ras_n_from_the_sdram => DRAM_RAS_N,
zs_we_n_from_the_sdram =>DRAM_WE_N
);
```

**Figure 3: Mapping the ports of the Nios II system to the correct DE2 pins**

For correct operation of the SDRAM module on the DE2 board, it is necessary that its memory clock (named *DRAM_CLK* above) lead the 50 MHz system clock by 3 nanoseconds. To ensure that this happens, a PLL was used (see figure 4).

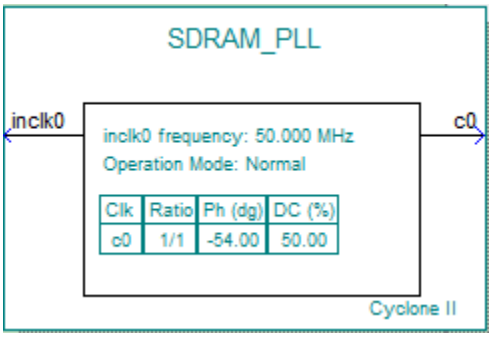


**Figure 4: PLL used to properly synchronize the SDRAM with the system**

The PLL was included as a component in the top-level entity, *Calculator.vhd*, as follows:

```
COMPONENT sdram_pll
PORT (
        inclk0  : IN STD_LOGIC;
        c0      : OUT STD_LOGIC );
END COMPONENT;
```

**Figure 5: Declaration of the PLL in the top-level entity**

It was mapped to the correct DE2 board pins with the following code:

```
-- Instantiate the entity sdram_pll (inclk0, c0).
neg_3ns: sdram_pll
PORT MAP
(
inclk0 => CLOCK_50,
c0 => DRAM_CLK
);
```

**Figure 6: Port mapping the PLL to the correct DE2 pins**

The next step in the implementation of the hardware was to add the LCD components to the system. Figure 1 shows the LCD driver connected to the Nios II system. Figures 2 and 3 show how the system was included in the top-level entity and its pins mapped accordingly. The overall system with all of the connections made is shown in figure 7 as a block diagram.

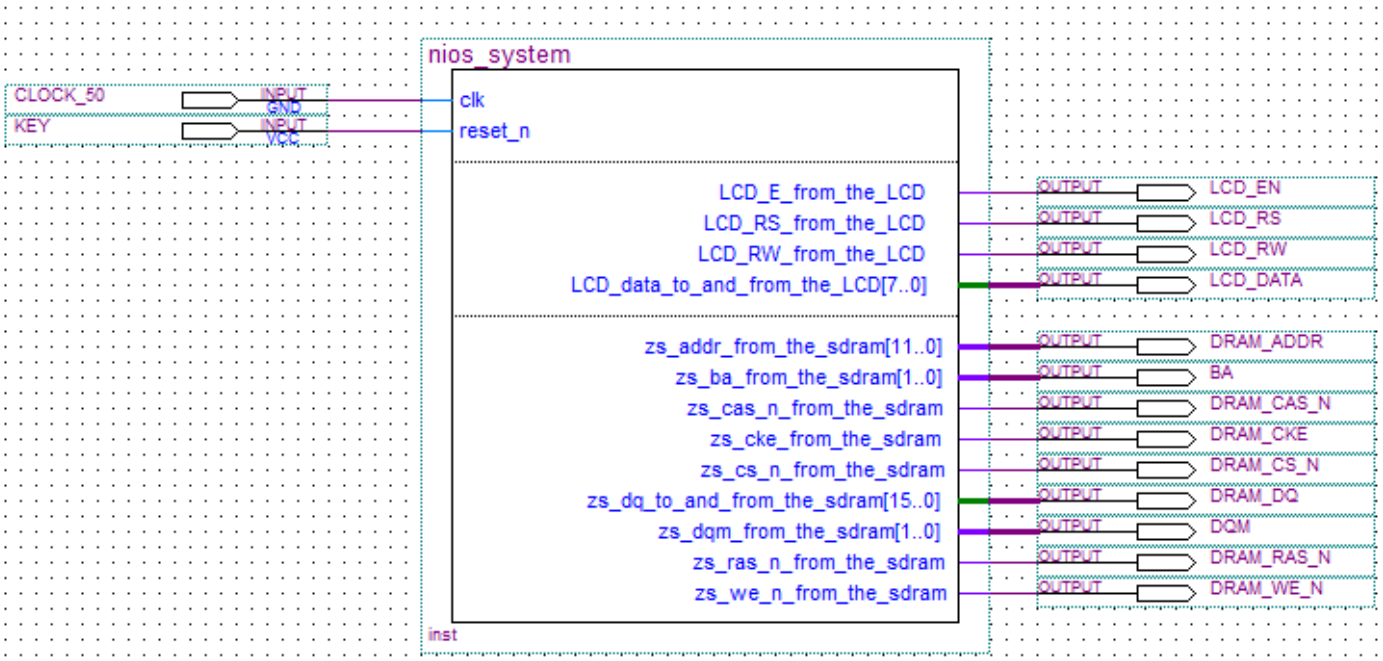


**Figure 7: The overall hardware implementation**

## Software

After running the Nios II IDE, the first step performed in the software implementation was to set up the program to be loaded into the SDRAM in the project's properties. Once the actual coding began, a stream to the LCD was first created with the following:

```
//Create LCD stream via fopen()
FILE *lcd = fopen(LCD_NAME, "w");
```

*LCD_NAME* is a string that contains the path to the LCD, located in the *system.h* header.

Once the stream was created, the program's interface was implemented. The user is first greeted with a menu-like interface that contains all of the functionality available:

```
//Give user a selection of operators
        char *reference = "Operator Reference:\n\n +    = Addition\n -    = Subtraction\n"
                          " *    = Multiplication\n /    = Division\n"
                          " MS   = Memory Store\n MC   = Memory clear\n"
                          " sin  = Sine\n cos  = Cosine\n tan  = Tangent\n"
                          " ln   = Natural Logarithm\n log  = Logarithm (base 10)\n"
                          " log2 = Logarithm (base 2)\n ^    = Exponent\n"
                          " !    = Factorial";

        printf("%s\n\n", references);
```

Like many embedded systems, the scientific calculator program is made to loop infinitely:

```
while(1)
{
    op1[0] = op2[0] = '\0';
    mem_used = 0;
    printf("Enter equation (or command): ");

    if(fgets(equation,sizeof(equation), stdin))
    { ...
```

Two character arrays, *op1* and *op2* are used as temporary storage for the detection of the operands (explained later); their first elements are set to the string terminating character, ‘ *\0* ’, to effectively “clear” the strings contained in each. This is done so that the old operands contained in the previously entered equation don’t interfere with the ones to be newly created. The *mem_used* integer is used as a flag to indicate whether the memory unit is part of the equation entered by the user. See the Appendix for the full code.

The loop executes in the following order, indefinitely:

- The user enters their equation
- The equation or command entered is parsed
- If an equation is entered, the operands and operators are put into appropriate containers
- The result is calculated or the command is carried out
- The result is sent to the LCD via a fprintf() function call

Based on the basic required functionality of the calculator, it can be deduced that there are two equation types and one command type that a user can enter:

i) [operand1] [operator] [operand2]
ii) [operator] [operand]
iii) [command]

The first equation type caters to the addition, subtraction, multiplication, division, and exponential operations. The second caters to the sine, cosine, tangent, and logarithm operations. The third equation type is for the memory store and memory clear operations. It should be noted that a fourth equation type technically exists: the factorial equation type. This type was not listed above, since it only pertains to a single operation that is experimental (with respect to the required functionality).

For equation type one, the equation string is parsed into the format above and its required terms (two operands and an operator) must then be used in calculating the result. The first step is to determine if the entered equation contains one of the five aforementioned operators. To do so, a call to the strstr() function (which is part of the *string.h* library) is made, in order to find whether the equation contains a substring consisting of each operator:

```
if(strstr(equation,"+")!= NULL || (strstr(equation,"-")!=NULL) || (strstr(equation,"*")!=NULL)
|| (strstr(equation,"/")!=NULL) || (strstr(equation,"^")!=NULL))
```

**Note:** The above code fits onto a single line in the code editor of the Nios II IDE. Five substrings were checked so that duplicate code for operand detection could be avoided.

If the strstr() function does not return *NULL*, then the equation entered does contain an operator that caters to equation type one. The next step in dealing with this equation type is to detect the first operand, *operand1*:

```
//Search for digits in the equation string or decimal places
//and assign them to to the operands

  while(isdigit(equation[i]) || equation[i] == '.'|| equation[i] == 'm')
  {
      if(equation[i] == 'm')
      {
          mem_used = 1;
          break;
      }
      else
      {
          strcat(op1,(char *)&equation[i]);
          i++;
      }
  }

              //Check if the memory unit was used
  if(mem_used == 1)
  {
      operand1 = memory;
  }
  else
  {
      operand1 = atof(op1);
  }
```

The isdigit() function (part of the *ctype.h* library) is used in conjunction with a while loop that checks each element of the *equation* character array (which contains the equation entered) to see if it is a digit. If the value in the current index of the *equation* array is a digit or decimal place, it is concatenated with the earlier mentioned *op1* character array (using the strcat() function) to form the first operand as a string. However, if the operand entered is an 'm', it means that the user would like to use the memory unit; in such a case, the loop is broken out of and the *mem_used* variable is set to 1. If the *mem_used* variable has a value of 1, the first operand is assigned to the currently stored value in the memory unit. If it is not, the first operand is assigned the value returned by the atof() function, which turns a string into a floating point value.

The next step in parsing an equation of type one is to ignore any spaces or operators in the equation string. This is done using the following:

```
while (!isdigit(equation[i])) //Skip spaces and operator
{
        i++;
}
```

As long as the current character in the *equation* array contains a non-digit value, its corresponding index will be skipped.

After the operator (a character, in this case) or any spaces have been skipped, the second operand must be detected. This is done in a similar fashion to the detection of the first operand:

```
while(isdigit(equation[i]) || equation[i] == '.'|| equation[i] == 'm')
{
    if(equation[i] == 'm')
    {
        mem_used = 1;
        break;
    }
    else
    {
        strcat(op2,(char *)&equation[i]);
        i++;
    }
}

if(mem_used == 1)
{
    operand2 = memory;
}
else
{
    operand2 = atof(op2);
}
```

This code allows a user to enter an equation such as *m + 4* or *m + m* when performing addition (or any other operation).

After the two operands have been detected and placed in their corresponding floating point variables, the specific operator entered by the user is checked. In effort to avoid tedious repetition, only the division operation will be looked at. The code is as follows:

```
//-- Division operation
else if(strstr(equation,"/") != NULL)
{
                    //Check if user is trying to divide by zero
    if(operand2 == 0.0)
    {
        printf("\nUndefined answer\n");
    }
    else
    {
        result = operand1 / operand2;
        if(mem_used == 1)
        {
            memory = result;
        }

        printf("\n%.2f / %.2f = %.2f\n\n",operand1, operand2, result);
        fprintf(lcd,"%.2f / %.2f \n= %.3f",operand1, operand2, result);
    }
```

```
}
```

The string contained in *equation* is checked using strstr() to see if it contains the division operator. If the second operand (the denominator, in this case) is zero, the user is notified that the answer cannot be computed and is undefined. If the denominator is not zero, the result is calculated; if the memory unit was used in the calculation, the result of the division is stored inside it (as per the required specifications of the calculator). Finally, the result is printed to the console, as well as the LCD (via an fprintf() function call).

The operators pertaining to equation type two are the trigonometric functions and logarithms. Again, only one from each of the two groups will be explored (the implementation of the other operators can be found in the Appendix): the sine and logarithm (with base 2) operations. Beginning with the trigonometric functions, they are first detected in the *equation* character array with the following code:

```
//-- Equation type 2 operations (sin, cos, tan)
else if ((strstr(equation,"sin")!=NULL)||(strstr(equation,"cos")!=NULL)
||(strstr(equation,"tan")!=NULL))
```

Similar to checking for equation type one operators, the conditions inside the if statement fit on one line in the code editor. Furthermore, a call to the strstr() function is used to detect the substrings containing each trigonometric operator.

Unlike equations of type one, type two equations have the operator first. Therefore, the indices containing non-digit characters or spaces must be skipped first, using the following code

```
i = 0;
while(!isdigit(equation[i])) //Ignore the operator and any spaces
{
    if(equation[i] == 'm')
    {
        mem_used = 1;
        break;
    }
    else
    {
        i++;
    }
}
```

If the non-digit value is an ‘m’, then the user wants to make use of the memory unit in the trigonometric calculation. This causes the *mem_used* flag to be set to 1, and the loop is broken out of. Otherwise, the index is incremented and the next character is checked.

Another significant difference between the two equation types is that equations of type two only contain a single operand, which is stored in the *operand1* floating point variable. The detection of the operand is identical to that of *operand1* and *operand2* in the equation type one operations. Therefore, the code will not be repeated. The next step is to detect each of the trigonometric operations individually (much like detecting each operator in the equation type one string).

The sine operation is detected and calculated as follows:

```
//-- Sine operation
if(strstr(equation,"sin") != NULL)
{
    result = sin(operand1);
    if(mem_used == 1)
    {
        memory = result;
    }

    printf("\nsin(%.2f) = %.2f\n\n",operand1, result);
    fprintf(lcd,"sin(%.2f) \n= %.3f",operand1, result);
}
```

Similar to the previous operations, the strstr() function call is made to detect whether the equation string inside the *equation* array contains the sine operator. The result is then calculated with the sin() function, found inside the *math.h* library. Like the previous operation, if the memory unit was used (indicated by a *mem_used* value of 1), the value of the result is stored inside it. The result is then printed to the console output, as well as the LCD on the DE2 board.

The logarithmic operators are similarly detected using the strstr() function. The code for the check is as follows:

```
//-- Remaining equation type 2 operations (ln, log, log2)
else if((strstr(equation,"ln") != NULL)||(strstr(equation,"log") != NULL)
      ||(strstr(equation,"log2") != NULL))
```

Once again, the above condition check fits onto a single line of the code editor. The operand is then detected in a similar fashion to the previous operations: `i = 0;`

```
//Ignore the operator, spaces, parantheses, etc.
while(!isdigit(equation[i])||((isdigit(equation[i])) && equation[i-1] == 'g'))
{
    if(equation[i] == 'm')
    {
        mem_used = 1;
        break;
    }
    else
    {
        i++;
    }
}
```

The major difference is in the third condition of the while loop statement. This condition accounts for the “log2” operator, which indeed contains a digit. If it was not there, the “log2” operator would always be detected as a “log” operator and the first digit of the operand would always be ‘2’. The spaces and operator and ignored identically to previous operations, otherwise.

Operand detection for all logarithms is identical to that of the trigonometric operators, and will not be shown. After the operand is entered, it is checked for a zero value as follows:

```
//Check if the user is trying to take the logarithm of zero
if(operand1 == 0.0)
{
    printf("\nUndefined answer\n");
}
else
{…
```

If the operand is zero, the user is notified that the answer is undefined, otherwise the logarithm type is detected and calculated. The logarithm (with base 2) is detected and calculated as follows:

```
//-- Logarithm operation with base = 2
else if (strstr(equation,"log2") != NULL)
{
    result = log10(operand1)/log10(2.0);
    if(mem_used == 1)
    {
        memory = result;
    }

    printf("\nlog2(%.2f) = %.2f\n\n",operand1, result);
    fprintf(lcd,"log2(%.2f) \n= %.3f",operand1, result);
}
```

For calculating the logarithm with base 2, there exists a function named log2() in the C99 standard (but not the C90 standard). Therefore, the calculation of “log2” is implemented using two calls to the log10() function (inside the *math.h* library) as shown above. This could have also been done with two calls to the log() function. As with all operations, the result is stored in memory if the memory unit was used as part of the calculation, and the result is printed to the console and LCD of the DE2 board.

As an additional feature to the calculator, the factorial operation was added; this was mainly done for experimental purposes. Its operand is first detected identically to the above operations. The operator, ‘!’, is then expected to follow the operand, so no code was used to explicitly ignore it. The calculation and display of the factorial is as follows:

```
for(c = 1; c <= (int)operand1; c++)
{
    result *= c;
}

if(mem_used == 1)
    {
        memory = result;
    }
printf("\n%.0f! = %.0f\n\n", operand1, result);
fprintf(lcd,"(%.2f)! \n= %.3f",operand1, result);
```

Much like a real scientific calculator, simply entering a number and pressing '=' will return the entered value as the result. This was implemented similarly (the only difference being the use of the return key on the keyboard). If the user does not enter an equation or single value, they are notified that an invalid equation was entered (see Appendix for the full code).

The commands available to the user are to store the result inside a memory unit, or clear the memory unit. The memory store operation is performed with the following code:

```
//-- Memory store
if (strstr(equation,"MS") != NULL)
{
    memory = result;
    printf("\n-- Memory stored --\n(Value = %f)\n\n",memory);
}
```

The result of the previous calculation is stored inside the memory unit. The value currently inside the memory unit is then displayed on the console. The memory clear operation is performed as follows:

```
//-- Memory clear
else
{
    memory = result;
     printf("\n-- Memory cleared --\n\n",memory);
}
```

The memory unit is set to zero, effectively "clearing" it (since the default value of a float is zero). A message is then sent to notify the user that the memory unit has been cleared.

# Conclusion

The final hardware and software implementation of the scientific calculator was successful. While the system met all of the required specifications with a few added features, improvements in the parsing of the equation string could have been made. This includes performing algorithms for an equation tree that performs BEDMAS on an equation with an unlimited number of terms. Overall, the project has served as a good way of exploring the Nios II environment, learning how hardware and software behave cohesively, as well as an exercise to maintain general programming skills and understanding.

# Appendix

### Calculator.vhd

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.std_logic_unsigned.all;
ENTITY Calculator IS

PORT
(
CLOCK_50 : IN STD_LOGIC;
KEY : IN STD_LOGIC_VECTOR(0 DOWNTO 0);
DRAM_CLK, DRAM_CKE : OUT STD_LOGIC;
DRAM_ADDR : OUT STD_LOGIC_VECTOR(11 DOWNTO 0);
DRAM_BA_1, DRAM_BA_0 : BUFFER STD_LOGIC;
DRAM_CS_N, DRAM_CAS_N, DRAM_RAS_N, DRAM_WE_N : OUT STD_LOGIC;
DRAM_DQ : INOUT STD_LOGIC_VECTOR(15 DOWNTO 0);
DRAM_UDQM, DRAM_LDQM : BUFFER STD_LOGIC;
LCD_EN : OUT STD_LOGIC;
LCD_RW : OUT STD_LOGIC;
LCD_RS : OUT STD_LOGIC;
LCD_ON : OUT STD_LOGIC;
LCD_DATA : INOUT STD_LOGIC_VECTOR(7 DOWNTO 0)
);
END Calculator;

ARCHITECTURE Structure OF Calculator IS
COMPONENT nios_system
PORT (
              -- 1) global signals:
                 signal clk : IN STD_LOGIC;
                 signal reset_n : IN STD_LOGIC;

              -- the_LCD
                 signal LCD_E_from_the_LCD : OUT STD_LOGIC;
                 signal LCD_RS_from_the_LCD : OUT STD_LOGIC;
                 signal LCD_RW_from_the_LCD : OUT STD_LOGIC;
                 signal LCD_data_to_and_from_the_LCD : INOUT STD_LOGIC_VECTOR (7 DOWNTO 0);

              -- the_LEDs
                 signal out_port_from_the_LEDs : OUT STD_LOGIC_VECTOR (7 DOWNTO 0);

              -- the_sdram
                 signal zs_addr_from_the_sdram : OUT STD_LOGIC_VECTOR (11 DOWNTO 0);
                 signal zs_ba_from_the_sdram : OUT STD_LOGIC_VECTOR (1 DOWNTO 0);
                 signal zs_cas_n_from_the_sdram : OUT STD_LOGIC;
                 signal zs_cke_from_the_sdram : OUT STD_LOGIC;
                 signal zs_cs_n_from_the_sdram : OUT STD_LOGIC;
                 signal zs_dq_to_and_from_the_sdram : INOUT STD_LOGIC_VECTOR (15 DOWNTO 0);
                 signal zs_dqm_from_the_sdram : OUT STD_LOGIC_VECTOR (1 DOWNTO 0);
                 signal zs_ras_n_from_the_sdram : OUT STD_LOGIC;
                 signal zs_we_n_from_the_sdram : OUT STD_LOGIC
      );

END COMPONENT;
```

```
COMPONENT sdram_pll
PORT (
              inclk0        : IN STD_LOGIC;
              c0            : OUT STD_LOGIC );
END COMPONENT;

SIGNAL BA : STD_LOGIC_VECTOR(1 DOWNTO 0);
SIGNAL DQM : STD_LOGIC_VECTOR(1 DOWNTO 0);

BEGIN

-- Instantiate the Nios II system entity generated by the SOPC Builder.
NiosII: nios_system
PORT MAP
(
--Global mappings
clk => CLOCK_50,
reset_n => KEY(0),
--LCD mappings
LCD_E_from_the_LCD => LCD_EN,
LCD_RS_from_the_LCD => LCD_RS,
LCD_RW_from_the_LCD => LCD_RW,
LCD_data_to_and_from_the_LCD => LCD_DATA,
--SDRAM mappings
zs_addr_from_the_sdram => DRAM_ADDR,
zs_ba_from_the_sdram => BA,
zs_cas_n_from_the_sdram => DRAM_CAS_N,
zs_cke_from_the_sdram => DRAM_CKE,
zs_cs_n_from_the_sdram => DRAM_CS_N,
zs_dq_to_and_from_the_sdram => DRAM_DQ,
zs_dqm_from_the_sdram => DQM,
zs_ras_n_from_the_sdram => DRAM_RAS_N,
zs_we_n_from_the_sdram =>DRAM_WE_N
);

-- Instantiate the entity sdram_pll (inclk0, c0).
neg_3ns: sdram_pll
PORT MAP
(
inclk0 => CLOCK_50,
c0 => DRAM_CLK
);

LCD_ON <= '1';

DRAM_BA_1 <= BA(1);
DRAM_BA_0 <= BA(0);

DRAM_UDQM <= DQM(1);
DRAM_LDQM <= DQM(0);

END Structure;
```

### Calculator.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <ctype.h>
#include "system.h"

float operand1, operand2, result, memory;
char equation[50]="", op1[30], op2[30];
int i, c, mem_used;
FILE *lcd;

int main() {

    //Give user a selection of operators
    char *reference = "Operator Reference:\n\n +    = Addition\n -    = Subtraction\n"
                      " *    = Multiplication\n /    = Division\n"
                      " MS   = Memory Store\n MC   = Memory clear\n"
                      " sin  = Sine\n cos  = Cosine\n tan  = Tangent\n"
                      " ln   = Natural Logarithm\n log  = Logarithm (base 10)\n"
                      " log2 = Logarithm (base 2)\n ^    = Exponent\n"
                      " !    = Factorial";

    printf("%s\n\n", reference);
       lcd = fopen(LCD_NAME, "w");

    while(1)
    {
        op1[0] = op2[0] = '\0';
        mem_used = 0;
        printf("Enter equation (or command): ");

        if(fgets(equation,sizeof(equation), stdin))
        {

             //-- Equation type 1 operations (+, -, *, /, and ^)
            //NOTE: This if statement should be placed on a single line
            if(strstr(equation,"+")!= NULL||(strstr(equation,"-")!= NULL)
              ||(strstr(equation,"*")!=NULL)||(strstr(equation,"/")!=NULL)
              ||(strstr(equation,"^")!=NULL))
            {
                i = 0;

              //Search for digits in the equation string or decimal places
              //and assign them to to the operands

                while(isdigit(equation[i]) || equation[i] == '.'|| equation[i] == 'm')
                {
                    if(equation[i] == 'm')
                    {
                        mem_used = 1;
                        break;
                    }
```

```
    else
    {
        strcat(op1,(char *)&equation[i]);
        i++;
    }
}

//Check if the memory unit was used
if(mem_used == 1)
{
    operand1 = memory;
}
else
{
    operand1 = atof(op1);
}

while (!isdigit(equation[i])) //Skip spaces and operator
{
        i++;
}

while(isdigit(equation[i]) || equation[i] == '.'|| equation[i] == 'm')
{
    if(equation[i] == 'm')
    {
        mem_used = 1;
        break;
    }
    else
    {
        strcat(op2,(char *)&equation[i]);
        i++;
    }
}

            if(mem_used == 1)
{
    operand2 = memory;
}
else
{
    operand2 = atof(op2);
}

            //-- Addition operation
if(strstr(equation,"+") != NULL)
{
    result = operand1 + operand2;
    if(mem_used == 1)
    {
        memory = result;
    }

    printf("\n%.2f + %.2f = %.2f\n\n",operand1, operand2, result);
    fprintf(lcd,"%.2f + %.2f \n= %.3f",operand1, operand2, result);
}
```

```
    //-- Subtraction operation
    else if(strstr(equation,"-") != NULL)
    {

        result = operand1 - operand2;
        if(mem_used == 1)
        {
            memory = result;
        }

        printf("\n%.2f - %.2f = %.2f\n\n",operand1, operand2, result);
        fprintf(lcd,"%.2f - %.2f \n= %.3f",operand1, operand2, result);
    }

    //-- Multiplication operation
    else if(strstr(equation,"*") != NULL)
    {
        result = operand1 * operand2;
        if(mem_used == 1)
        {
            memory = result;
        }

        printf("\n%.2f * %.2f = %.2f\n\n",operand1, operand2, result);
        fprintf(lcd,"%.2f * %.2f \n= %.3f",operand1, operand2, result);
    }

    //-- Division operation
    else if(strstr(equation,"/") != NULL)
    {
        //Check if user is trying to divide by zero
        if(operand2 == 0.0)
        {
            printf("\nUndefined answer\n");
        }
        else
        {
            result = operand1 / operand2;
            if(mem_used == 1)
            {
                memory = result;
            }

            printf("\n%.2f / %.2f = %.2f\n\n",operand1, operand2, result);
            fprintf(lcd,"%.2f / %.2f \n= %.3f",operand1, operand2, result);
        }
    }

    //-- Exponential operation
    else
    {
        result = pow(operand1, operand2);
        if(mem_used == 1)
        {
            memory = result;
        }

        printf("\n%.2f^%.2f = %.2f\n\n",operand1, operand2, result);
        fprintf(lcd,"%.2f ^ %.2f \n= %.3f",operand1, operand2, result);
    }
}
```

```
//-- Equation type 2 operations (sin, cos, tan)
//NOTE: This if statement should be placed on a single line
else if((strstr(equation,"sin") != NULL)||(strstr(equation,"cos") != NULL)
         ||(strstr(equation,"tan") != NULL))
{
    i = 0;
    while(!isdigit(equation[i])) //Ignore the operator and any spaces
    {
        if(equation[i] == 'm')
        {
            mem_used = 1;
            break;
        }
        else
        {
            i++;
        }
    }

    //Only one operand needs to be detected
    while(isdigit(equation[i]) || equation[i] == '.' || equation[i] == 'm')
    {
        if(equation[i] == 'm')
        {
            mem_used = 1;
            break;
        }
        else
        {
            strcat(op1,(char *)&equation[i]);
            i++;
        }
    }

    if(mem_used == 1)
    {
        operand1 = memory;
    }
    else
    {
    operand1 = atof(op1);
    }

    //-- Sine operation
    if(strstr(equation,"sin") != eNULL)
    {
        result = sin(operand1);
        if(mem_used == 1)
        {
            memory = result;
        }

        printf("\nsin(%.2f) = %.2f\n\n",operand1, result);
        fprintf(lcd,"sin(%.2f) \n= %.3f",operand1, result);
    }
```

```
    //-- Cosine operation
    else if (strstr(equation,"cos") != NULL)
    {
        result = cos(operand1);
        if(mem_used == 1)
        {
            memory = result;
        }

        printf("\ncos(%.2f) = %.2f\n\n",operand1, result);
        fprintf(lcd,"cos(%.2f) \n= %.3f",operand1, result);
    }

    //-- Tangent operation
    else
    {
        result = tan(operand1);
        if(mem_used == 1)
        {
            memory = result;
        }

        printf("\ntan(%.2f) = %.2f\n\n",operand1, result);
        fprintf(lcd,"tan(%.2f) \n= %.3f",operand1, result);
    }
}

//-- Remaining equation type 2 operations (ln, log, log2)
//NOTE: This if statement should be placed on a single line

else if((strstr(equation,"ln") != NULL)||(strstr(equation,"log") != NULL)
        ||(strstr(equation,"log2") != NULL))
{
    i = 0;
    //Ignore the operator, spaces, parantheses, etc.
    while(!isdigit(equation[i])||((isdigit(equation[i])) && equation[i-1] == 'g'))
    {
        if(equation[i] == 'm')
        {
            mem_used = 1;
            break;
        }
        else
        {
            i++;
        }
    }
    while(isdigit(equation[i]))
    {
         //Check if the memory unit was used
        if(equation[i] == 'm')
        {
            mem_used = 1;
            break;
        }
        else
        {
            strcat(op1,(char *)&equation[i]);
            i++;
        }
    }
```

```
    if(mem_used == 1)
    {
        operand1 = memory;
    }
    else
    {
    operand1 = atof(op1);
    }

    //Check if the user is trying to take the logarithm of zero
    if(operand1 == 0.0)
    {
        printf("\nUndefined answer\n");
    }
    else
    {
         //-- Natural logarithm operation
        if(strstr(equation,"ln") != NULL)
        {
            result = log(operand1);
            if(mem_used == 1)
            {
                memory = result;
            }

            printf("\nln(%.2f) = %.2f\n\n",operand1, result);
            fprintf(lcd,"ln(%.2f) \n= %.3f",operand1, result);
        }

         //-- Logarithm operation with base = 2
        else if (strstr(equation,"log2") != NULL)
        {
            result = log10(operand1)/log10(2.0);
            if(mem_used == 1)
            {
                memory = result;
            }

            printf("\nlog2(%.2f) = %.2f\n\n",operand1, result);
            fprintf(lcd,"log2(%.2f) \n= %.3f",operand1, result);
        }

         //-- Logarithm operation with base = 10
        else
        {
            result = log10(operand1);
            if(mem_used == 1)
            {
                memory = result;
            }

            printf("\nlog(%.2f) = %.2f\n\n",operand1, result);
            fprintf(lcd,"log(%.2f) \n= %.3f",operand1, result);
        }
    }
}
```

```
//-- Memory operations
else if ((strstr(equation,"MS") != NULL)||(strstr(equation,"MC") != NULL))
{
    //-- Memory store
    if (strstr(equation,"MS") != NULL)
    {
        memory = result;
        printf("\n-- Memory stored --\n(Value = %f)\n\n",memory);
    }

    //-- Memory clear
    else
    {
        memory = result;
         printf("\n-- Memory cleared --\n\n",memory);
    }

}

//-- Factorial operation (added as a bonus)
else if ((strstr(equation,"!") != NULL))
{
    i = 0;
    result = 1;

    while(isdigit(equation[i]) || equation[i] == 'm')
    {
        if(equation[i] == 'm')
        {
            mem_used = 1;
            break;
        }
        else
        {
            strcat(op1,(char *)&equation[i]);
            i++;
        }
    }

    if(mem_used == 1)
    {
        operand1 = memory;
    }
    else
    {
    operand1 = atof(op1);
    }

    for(c = 1; c <= (int)operand1; c++)
    {
        result *= c;
    }

    if(mem_used == 1)
        {
            memory = result;
        }
    printf("\n%.0f! = %.0f\n\n", operand1, result);
    fprintf(lcd,"(%.2f)! \n= %.3f",operand1, result);
}
```

```
            else
            {
                i = 0;
                //Check if the user entered a single digit
                if(isdigit(equation[i]))
                {
                    while(isdigit(equation[i]))
                    {
                        strcat(op1,(char *)&equation[i]);
                        i++;
                    }
                    result = atof(op1);
                    printf("\n%.3f\n\n", result);
                }
                //Else, let them know that the equation is invalid
                else
                {
                    printf("\nInvalid Equation or Command.\n\n");
                }
            }
        }

        fclose(lcd);
        remove(lcd);

    }
}
```