ECE 4515: Digital Design 2

Project 1

Solving the Maclaurin Series of (e^x – 1) in a soft processor

Manav Bhavesh Shah
Bradley Department of Electrical and Computer Engineering
Virginia Polytechnic Institute and State University
Blacksburg, USA
manavbs28@vt.edu

"*I have neither given nor received unauthorized assistance on this assignment.*"

February 14, 2025

~Manav Shah

1. Introduction

The purpose of this project is to solve the Maclaurin series $e^x - 1$ on a softcore processor deployed over the De1 SOC FPGA board.

A **softcore processor** is a processor which is built using the resources available in the programmable logic fabric of the FPGA. It is completely implemented using logic synthesis.

Bare metal Programming is generally referred to as the programming done without the dependance of any operating system. This type of programming interacts directly with the hardware, giving the programmer complete control.

The Maclurin series is approximated in polynomial form as shown in the formula given below.

$$e^x - 1 = \sum_{n=1}^{\infty} \frac{x^n}{n!} = x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} \dots$$

For the purposes of this project, we are allowed to approximate the result to the first 4 terms of the expansion formula. This region of interest for x ranging from -1 to 1 is used in the test stimulus provided to the solver. In a previous assignment, we worked on making a softcore processor with platform designer in Quartus. The same processor is being repurposed for this project with modifications on the amount of On-Chip ram, to allow enough space to import things like <math.h>, <altera_avalon_pio_regs.h> and <system.h>.

2. RTL View of components in Quartus project.

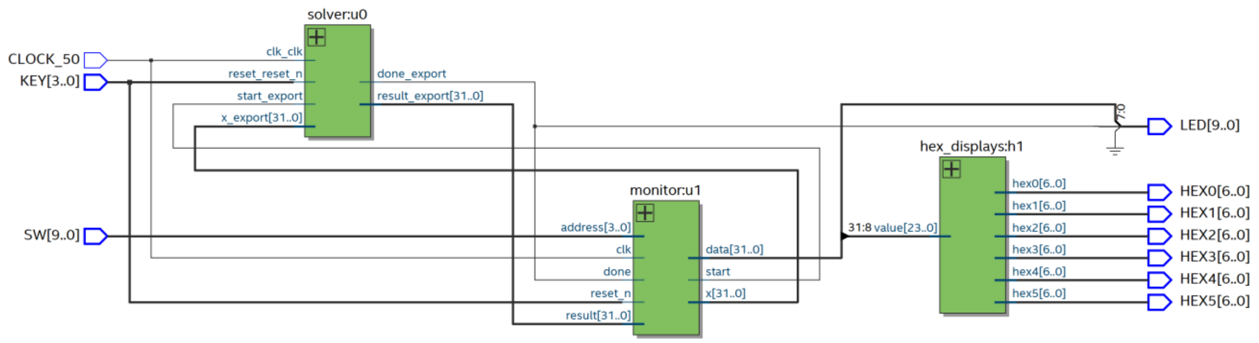


Fig. [1]. Quartus RTL view of top file

The solver module is the softcore processor which is instantiated in the top module of this Quartus project.

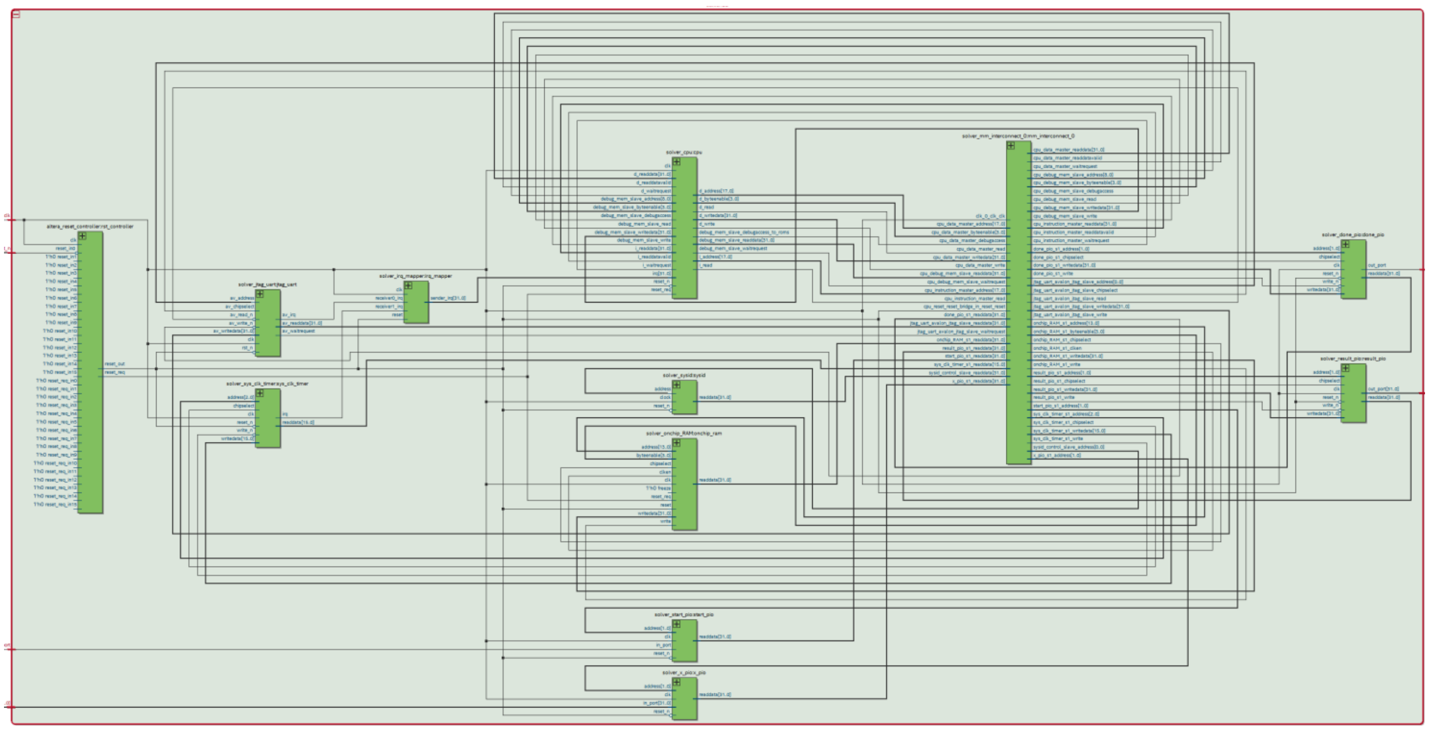


Fig. [2]. RTL view of Solver

‘Solver’ is the softcore CPU designed using platform designer.

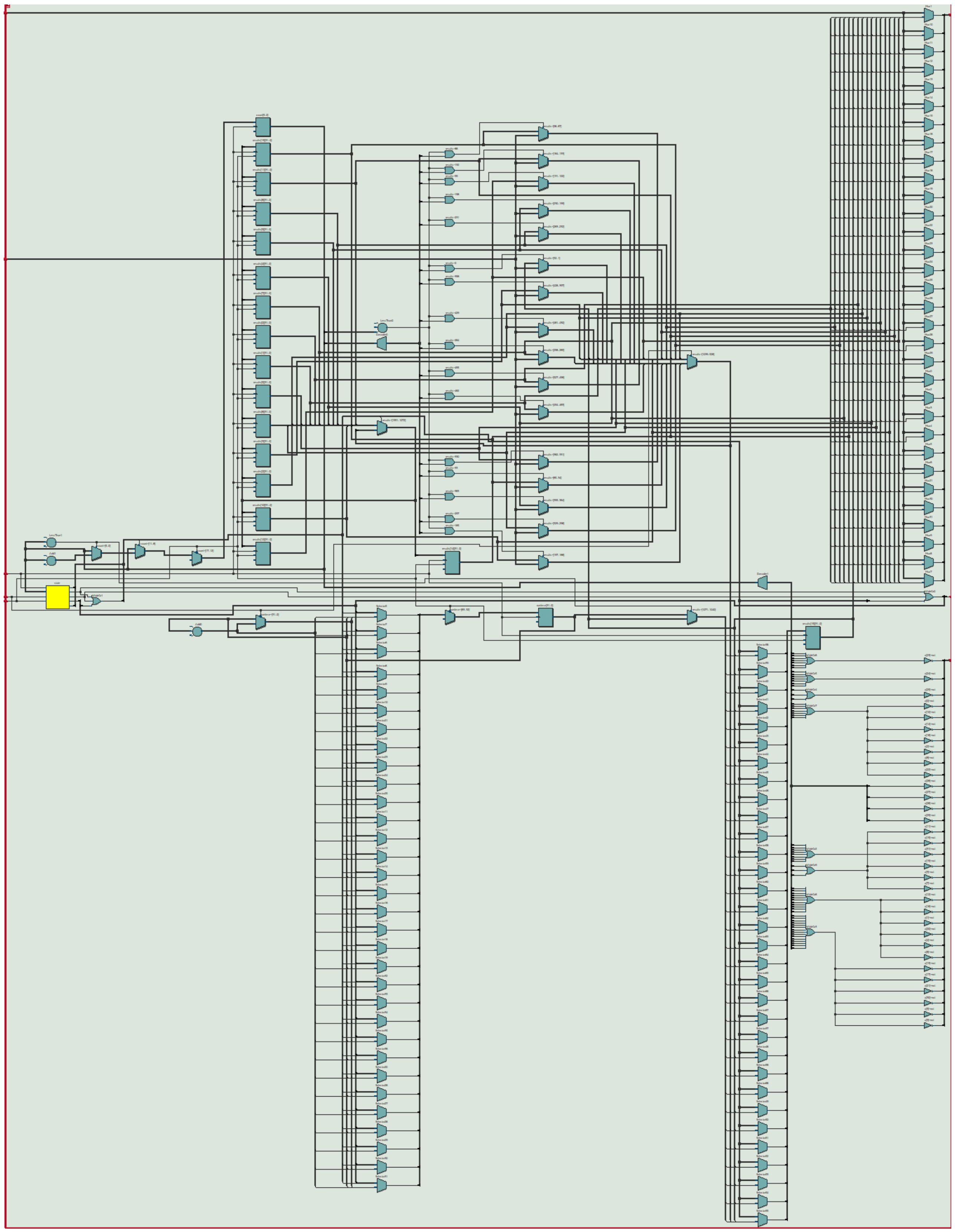


Fig. [3]. RTL view of Monitor

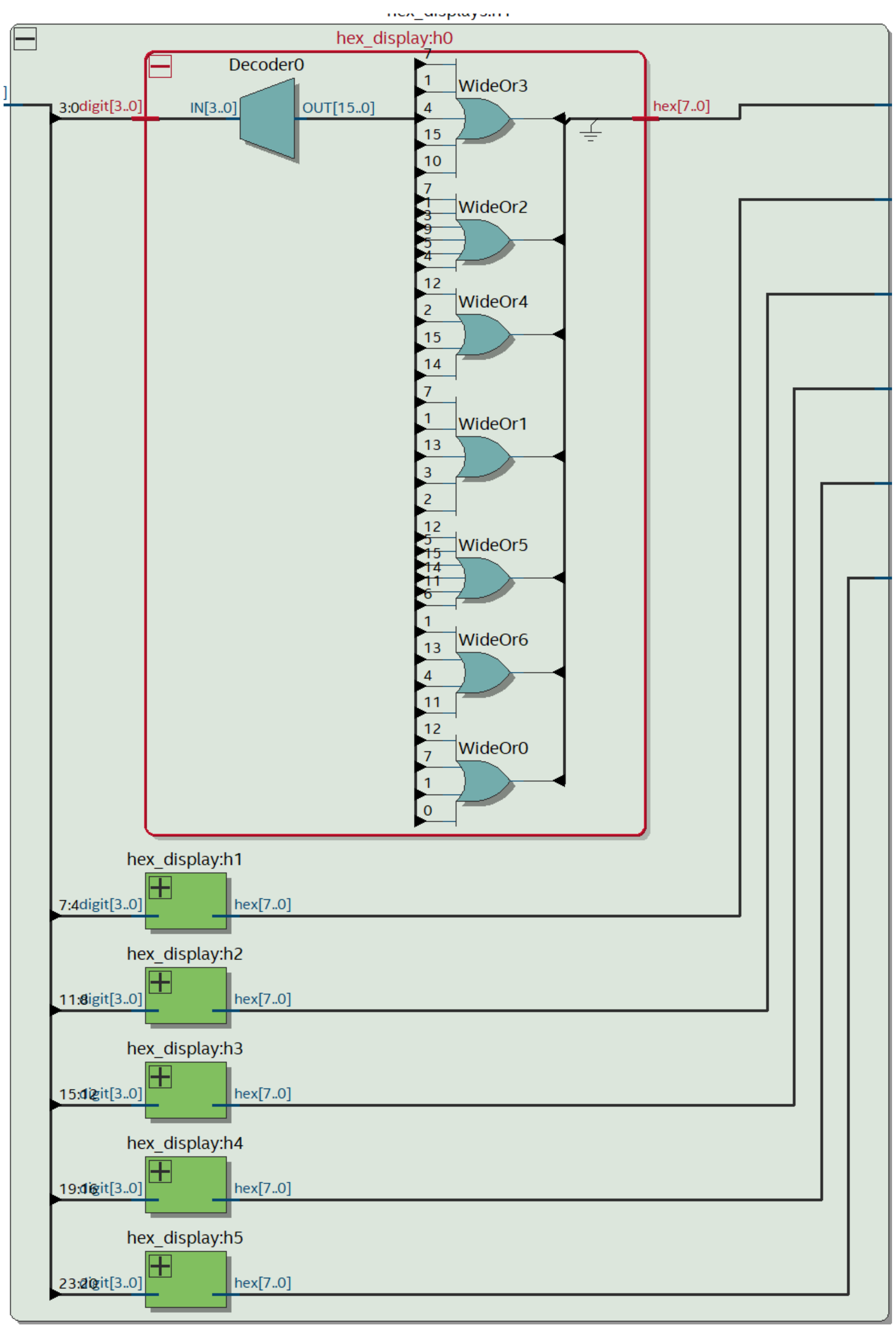
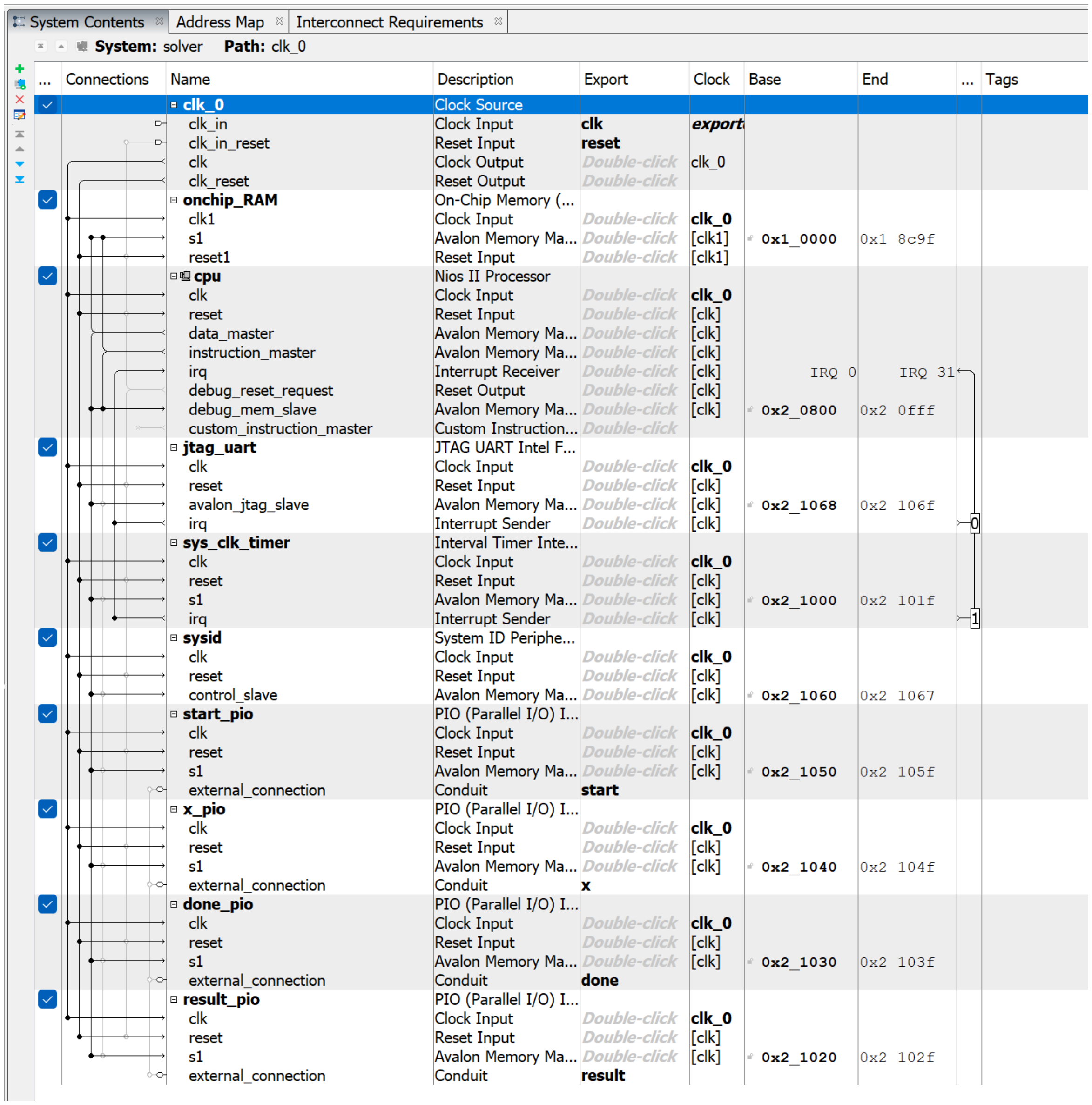


Fig. [4]. RTL view of Hex Display

3. Softcore Processor Design

Designing a softcore processor provides a practical way to offload some of the design workload onto an FPGA, enabling the instantiation of a functional CPU within the SoC fabric to process information and 'solve' complex mathematical functions such as the Maclaurin series which require exponent and factorial calculations. Shown below is the complete capture of the 'solver' I built for this project by following the guidelines in HW2 Hello World.



System Contents | Address Map | Interconnect Requirements

System: solver **Path:** clk_0

...	Connections	Name	Description	Export	Clock	Base	End	...	Tags
✓		**clk_0**	Clock Source						
		clk_in	Clock Input	**clk**	*export*				
		clk_in_reset	Reset Input	**reset**					
		clk	Clock Output	*Double-click*	clk_0				
		clk_reset	Reset Output	*Double-click*					
✓		**onchip_RAM**	On-Chip Memory (...						
		clk1	Clock Input	*Double-click*	**clk_0**				
		s1	Avalon Memory Ma...	*Double-click*	[clk1]	0x1_0000	0x1 8c9f		
		reset1	Reset Input	*Double-click*	[clk1]				
✓		**cpu**	Nios II Processor						
		clk	Clock Input	*Double-click*	**clk_0**				
		reset	Reset Input	*Double-click*	[clk]				
		data_master	Avalon Memory Ma...	*Double-click*	[clk]				
		instruction_master	Avalon Memory Ma...	*Double-click*	[clk]				
		irq	Interrupt Receiver	*Double-click*	[clk]	IRQ 0	IRQ 31		
		debug_reset_request	Reset Output	*Double-click*	[clk]				
		debug_mem_slave	Avalon Memory Ma...	*Double-click*	[clk]	0x2_0800	0x2 0fff		
		custom_instruction_master	Custom Instruction...	*Double-click*					
✓		**jtag_uart**	JTAG UART Intel F...						
		clk	Clock Input	*Double-click*	**clk_0**				
		reset	Reset Input	*Double-click*	[clk]				
		avalon_jtag_slave	Avalon Memory Ma...	*Double-click*	[clk]	0x2_1068	0x2 106f		
		irq	Interrupt Sender	*Double-click*	[clk]			0	
✓		**sys_clk_timer**	Interval Timer Inte...						
		clk	Clock Input	*Double-click*	**clk_0**				
		reset	Reset Input	*Double-click*	[clk]				
		s1	Avalon Memory Ma...	*Double-click*	[clk]	0x2_1000	0x2 101f		
		irq	Interrupt Sender	*Double-click*	[clk]			1	
✓		**sysid**	System ID Periphe...						
		clk	Clock Input	*Double-click*	**clk_0**				
		reset	Reset Input	*Double-click*	[clk]				
		control_slave	Avalon Memory Ma...	*Double-click*	[clk]	0x2_1060	0x2 1067		
✓		**start_pio**	PIO (Parallel I/O) I...						
		clk	Clock Input	*Double-click*	**clk_0**				
		reset	Reset Input	*Double-click*	[clk]				
		s1	Avalon Memory Ma...	*Double-click*	[clk]	0x2_1050	0x2 105f		
		external_connection	Conduit	**start**					
✓		**x_pio**	PIO (Parallel I/O) I...						
		clk	Clock Input	*Double-click*	**clk_0**				
		reset	Reset Input	*Double-click*	[clk]				
		s1	Avalon Memory Ma...	*Double-click*	[clk]	0x2_1040	0x2 104f		
		external_connection	Conduit	**x**					
✓		**done_pio**	PIO (Parallel I/O) I...						
		clk	Clock Input	*Double-click*	**clk_0**				
		reset	Reset Input	*Double-click*	[clk]				
		s1	Avalon Memory Ma...	*Double-click*	[clk]	0x2_1030	0x2 103f		
		external_connection	Conduit	**done**					
✓		**result_pio**	PIO (Parallel I/O) I...						
		clk	Clock Input	*Double-click*	**clk_0**				
		reset	Reset Input	*Double-click*	[clk]				
		s1	Avalon Memory Ma...	*Double-click*	[clk]	0x2_1020	0x2 102f		
		external_connection	Conduit	**result**					

Fig. [5]. Solver design in Platform Designer.

The On-Chip RAM is kept at 16kB

(optimizations in the C program were made to keep this number low).

4. C Program for solving Maclaurin Series.

without optimizations:

```c
#include <system.h>
#include <altera_avalon_pio_regs.h>
#include <math.h>

volatile unsigned long x;
  volatile unsigned long result;
  float x_float;
  float result_float;

  float calc_maclaurin(float f){
     float ans = f + (pow(f,2)/2) +(pow(f,3)/6) +(pow(f,4)/24);
     return ans;
  }

  int main()
  {
    while(1){

        // Set Done = 1
          IOWR_ALTERA_AVALON_PIO_DATA(DONE_PIO_BASE,1);

          // Check if Start signal is high
          if(IORD_ALTERA_AVALON_PIO_DATA(START_PIO_BASE)){

                // read the value of input x to variable x
                x = IORD_ALTERA_AVALON_PIO_DATA(X_PIO_BASE);

                // convert unsigned long x to float x
                x_float = *((float*)&x);

                // write 0 to Done signal
                IOWR_ALTERA_AVALON_PIO_DATA(DONE_PIO_BASE,0);

                // calculate result (in float)
                result_float = calc_maclaurin(x_float);

                // convert result to unsigned long
                result = *((unsigned long*)&result_float);

                // write the result. (base address: 0x21020)
                IOWR_ALTERA_AVALON_PIO_DATA(RESULT_PIO_BASE,result);

                // set done = 1
                IOWR_ALTERA_AVALON_PIO_DATA(DONE_PIO_BASE,1);
      }
    }
  }
```

Optimizations:

To reduce the number of cycles of operation, I changed some aspects of the code:

1) Removed print statements used in debugging and removed <stdio.h>
2) Hard coded the formula for the 1st 4 terms in Maclaurin expansion and removed function calc_maclaurin(). This allowed me to remove <math.h>
3) Restricted the code to use only unsigned int and float data types.

```c
#include <system.h>
#include <altera_avalon_pio_regs.h>

volatile unsigned int x;
volatile unsigned int result;
float x_float;
     float result_float;

     int main()
     {
           //printf("Hello From Manav's SOLVER Nios 2. \n");
           //printf("This code is to solve the Macluarin Series upto first 4 terms of the explansion formula. \n");

           while(1){

           // Set Done = 1
             IOWR_ALTERA_AVALON_PIO_DATA(DONE_PIO_BASE,1);

             // Check if Start signal is high
             if(IORD_ALTERA_AVALON_PIO_DATA(START_PIO_BASE)){

                   // read the value of input x to variable x
                   x = IORD_ALTERA_AVALON_PIO_DATA(X_PIO_BASE);
                   //printf("The value of x is: %u. \n", x);

                   // convert unsigned long x to float x
                   x_float = *((float*)&x);

                   // write 0 to Done signal
                   IOWR_ALTERA_AVALON_PIO_DATA(DONE_PIO_BASE,0);
                   //printf("Done signal set to 1 for performing calculations");

                   // calculate result (in float)
                   result_float = (x_float + ((x_float*x_float)/2) +((x_float*x_float*x_float)/6) +((x_float*x_float*x_float*x_float)/24));

                   // convert result to unsigned long
                   result = *((unsigned int*)&result_float);

                   // write the result. (base address: 0x21020)
                   IOWR_ALTERA_AVALON_PIO_DATA(RESULT_PIO_BASE,result);
                   //printf("Result written");

```

```
                    // set done = 1
                    IOWR_ALTERA_AVALON_PIO_DATA(DONE_PIO_BASE,1);
            }
        }
    }
```

4. Results

a) Functional Results

For ease of illustration, this section is condensed into a table which encapsulates all the values for every test provided in the monitor module. The way the design works is the values of x (ranging from -1 to 1) are provided by the monitor module to the solver (which is the NIOS softcore processor "solver" running the C program).

The 2nd column shows the input x value as specified in the test stimulus. Column 3 shows the expected value of the Maclaurin series.

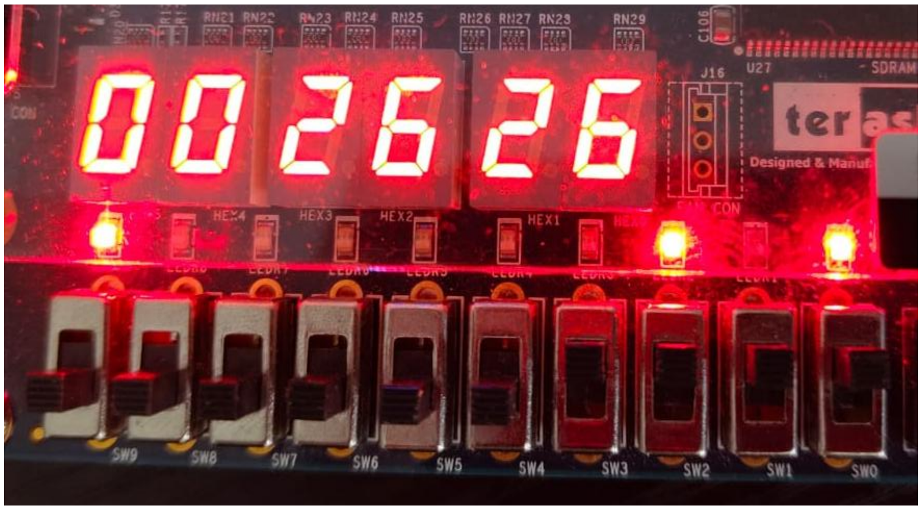
The Hex displays and LEDs show the result of every input 'x'. The switches SW0 to SW3 are used to toggle between these results and the values observed on the display are noted in the 4th column of the table given below. These HEX values need to be converted into decimal format which is done manually by using an online hex to decimal converter.

INDEX	Maclaurin Input (x)	Expected Values	Values on HEX display	Display Values in Float
0	-1	-0.625	bf200000	-0.625
1	-0.8	-0.548266667	bf0c5b64	-0.54826665
2	-0.6	-0.4506	bee6b506	-0.4506
3	-0.4	-0.3296	bea8c155	-0.3296
4	-0.3	-0.2591625	be84b0f2	-0.25916252
5	-0.2	-0.181266667	be399df8	-0.18126667
6	-0.1	-0.0951625	bdc2e48f	-0.0951625
7	0	0	0	0
8	0.1	0.105170833	3dd763cf	0.10517084
9	0.2	0.2214	3e62b6ae	0.22140001
10	0.3	0.3498375	3eb31de8	0.34983775
11	0.4	0.491733333	3efbc479	0.49173334
12	0.6	0.8214	3f524746	0.82139987
13	0.8	1.2224	3f9c779b	1.2224001
14	1	1.708333333	3fdaaaaa	1.7083334

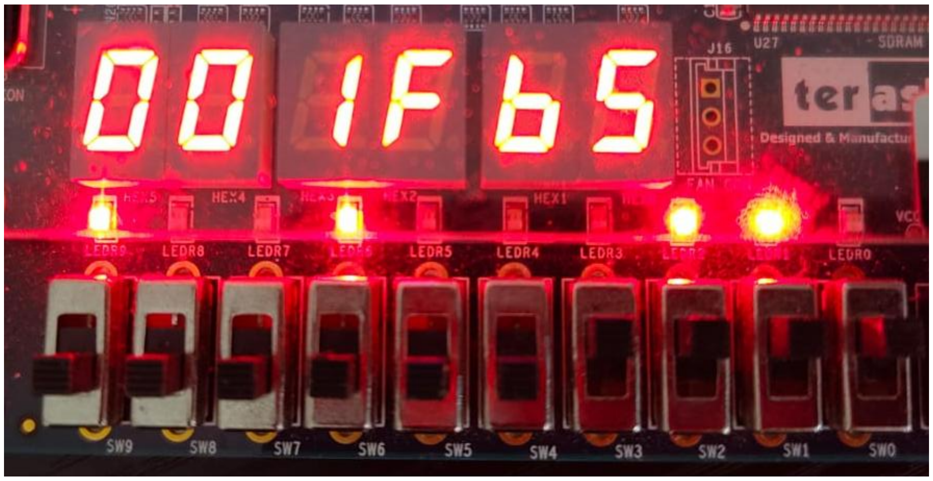
This table verifies the functional correctness of the work done in this project since the values obtained by using the solver to obtain the approximation of the Maclaurin series are the same as the expected solution.

b) Temporal Results

The monitor counts the number of clock cycles required to complete the fourteen calculations. This value is stored at address 0xF of the results.
Initially, I was using multiple print statements for the sake of debugging the C program. Due to this the number of clock cycles required was higher. (2,500,101 Cycles)

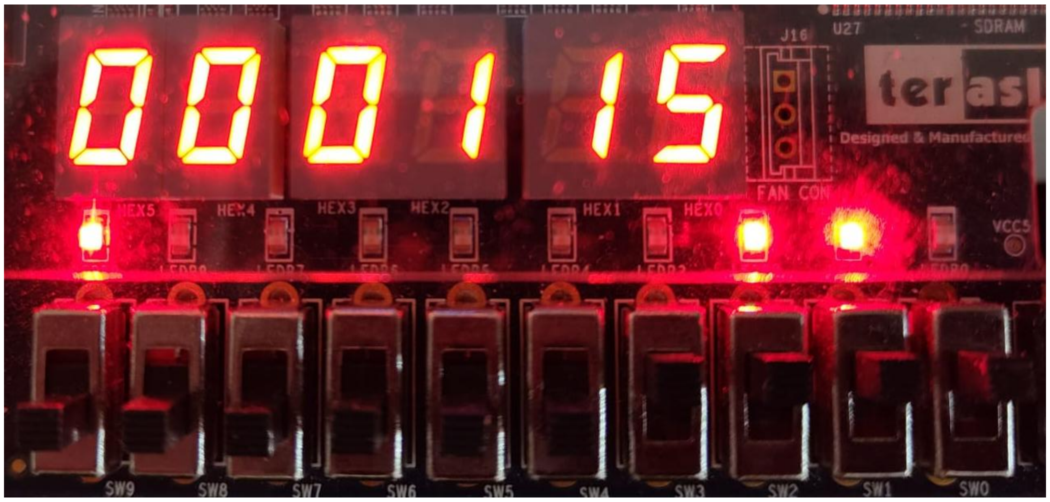


The next step was removing the print statements and running bare metal code which was tested for its functional correctness in the prior run with multiple debug statements. This brough the number of cycles down to 2,078,022 Cycles.



To further optimize the code and reduce cycles, I altered the C program in the following ways:

1) Removed the function named calc_maclaurin and the math.h library. This function used pow() which was included via <math.h>
2) The first 4 terms of the Maclaurin series were directly coded in the program as follows:
   *result_float = (x_float + ((x_float*x_float)/2) +((x_float*x_float*x_float)/6) +((x_float*x_float*x_float*x_float)/24));*
3) Only unsigned int and float data types were used in the program.

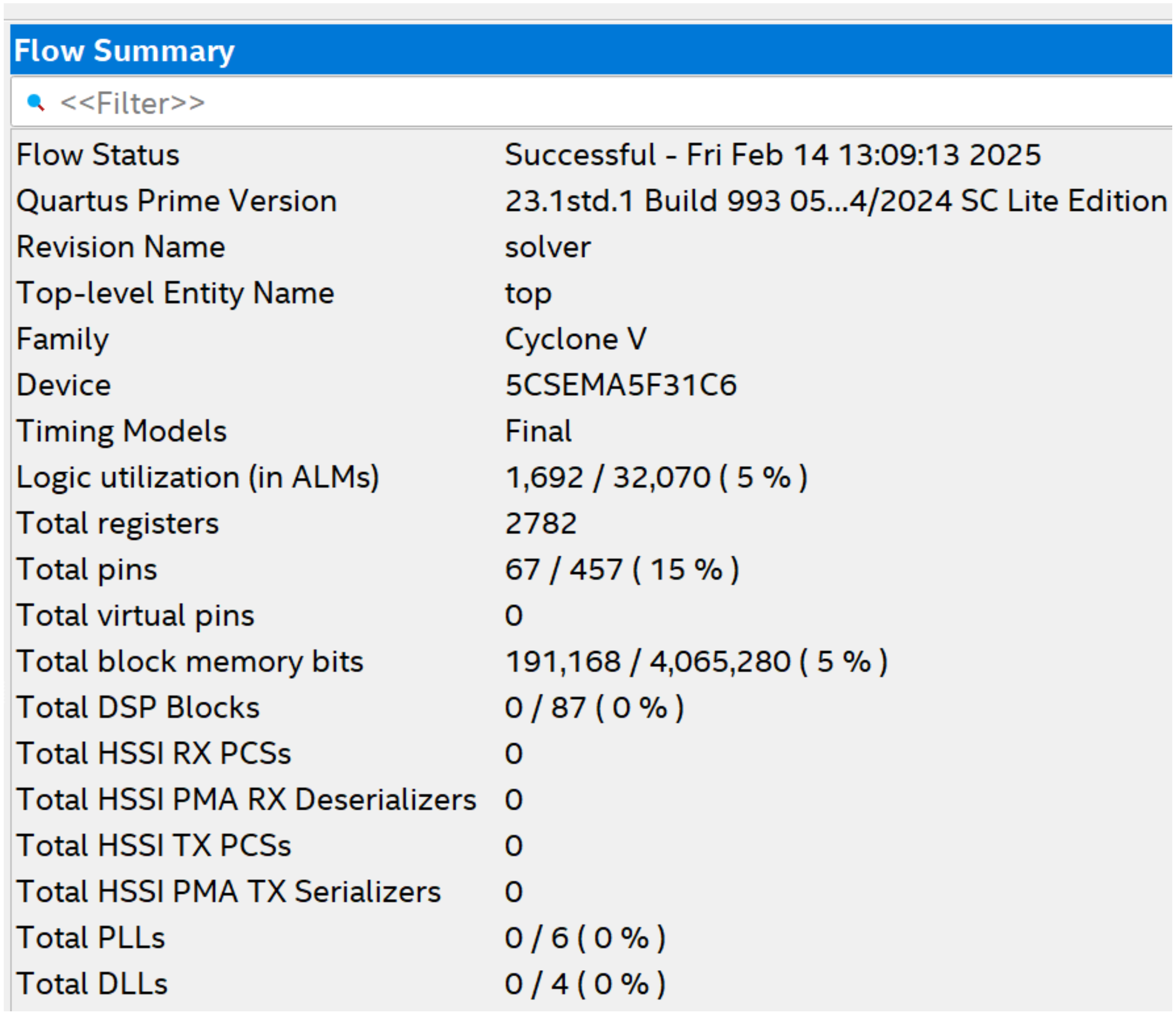


These changes reduced the number of cycles required to 70,918.

c) Spatial Results

The area calculation is based on the resource usage of your implementation. Use the following formula to calculate the area of your design:

$$
\begin{aligned}
\mathit{Area} &= (75 * \mathit{DSP}) + (\mathit{ALM}) + (0.001 * \mathit{BLOCK\_MEMORY\_BITS}) \\
&= (75 * 0) + (1692) + (0.001*191168) \\
&= 1883.168
\end{aligned}
$$



Flow Summary	
Flow Status	Successful - Fri Feb 14 13:09:13 2025
Quartus Prime Version	23.1std.1 Build 993 05...4/2024 SC Lite Edition
Revision Name	solver
Top-level Entity Name	top
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	1,692 / 32,070 (5 %)
Total registers	2782
Total pins	67 / 457 (15 %)
Total virtual pins	0
Total block memory bits	191,168 / 4,065,280 (5 %)
Total DSP Blocks	0 / 87 (0 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0 / 6 (0 %)
Total DLLs	0 / 4 (0 %)

5. Conclusion

This project helped me learn about the design and implementation of softcore processors in an FPGA. The bare metal programming optimizations were crucial in understanding the memory size and clock cycle requirements.