**California State University, Fresno   
Lyles College of Engineering   
Electrical and Computer Engineering Department**   
**TECHNICAL REPORT**   
   
Assignment: Number 4  
Experiment Title: HW/SW Co-Design of an Embedded System on FPGA  
Course Title: ECE 178 (Embedded Systems)   
Instructor: Dr. Reza Raeisi   
   
 Prepared by: Anthony Herrick   
 Student ID #300144976  
Date Submitted: 10/09/2022

**INSTRUCTOR SECTION**   
   
Comments: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_   
 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_   
 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_   
 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_   
 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_   
 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_   
 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_   
   
Final Grade: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

TABLE OF CONTENTS

[Objective 4](#_Toc116242898)

[Hardware Requirements 4](#_Toc116242899)

[Software Requirements 4](#_Toc116242900)

[Background 4](#_Toc116242901)

[Project Overview 7](#_Toc116242902)

[Project Procedure 8](#_Toc116242903)

[Conclusion 30](#_Toc116242904)

[Appendix A (Part 1 Code) 31](#_Toc116242905)

[Appendix B (Part 2 Code) 33](#_Toc116242906)

[Appendix C (Part 3 Code) 36](#_Toc116242907)

[Appendix D (Part 4 Code) 37](#_Toc116242908)

[Appendix E (Part 5 Code) 40](#_Toc116242909)

LIST OF FIGURES

[1. Soft Core Embedded System Design 5](#_Toc116242871)

[2. 7-Segment Display Address Spaces 6](#_Toc116242872)

[3. 7-Segment Display Breakdown 7](#_Toc116242873)

[4. Register Values Before Call 8](#_Toc116242874)

[5. After Entering the Divide Subroutine 9](#_Toc116242875)

[6. Values After the First CONT loop 9](#_Toc116242876)

[7. Part 1 Final Values 10](#_Toc116242877)

[8. Value of Switches being loaded into r4 11](#_Toc116242878)

[9. Loading the Byte Sized Value for the 7 Segment 11](#_Toc116242879)

[10. Board After Value is Loaded into 7-Segment 12](#_Toc116242880)

[11. Squared Value on the LEDs 13](#_Toc116242881)

[12. After SRLI 14](#_Toc116242882)

[13. After And 15](#_Toc116242883)

[14. After r10 is iterated 16](#_Toc116242884)

[15. Starting Values Entering the Fourth Loop 17](#_Toc116242885)

[16. Fourth Loop After srli and and 18](#_Toc116242886)

[17. End of Part Three Program Registers 19](#_Toc116242887)

[18. Part 3 Test Case Results 20](#_Toc116242888)

[19. Registers before entering Ones Loop 21](#_Toc116242889)

[20. Registers after exiting Ones Subroutine 22](#_Toc116242890)

[21. Before Entering Ones subroutine Second Time 23](#_Toc116242891)

[22. After Exiting Ones Subroutine 24](#_Toc116242892)

[23. Final Values Of Program 25](#_Toc116242893)

[24. 7 Segment Part 4 26](#_Toc116242894)

[25. Starting values after Initialization 27](#_Toc116242895)

[26. Registers after Returning from Count 28](#_Toc116242896)

[27. Part 5 LEDs 29](#_Toc116242897)

# Objective

Upon the completion of this lab, one will have co-designed an embedded soft-core system and application development in an FPGA design environment. We will be using Intel’s EDA design tool Qsys, in conjunction with the Quartus software, and Intel FPGA monitor.

# Hardware Requirements

* Computer with Intel FPGA Monitor program 16.1
* Computer with Quartus Prime 16.1.
* DE2-115 Board
* A-B USB Cable

# Software Requirements

* Intel FPGA monitor program 16.1 or greater
* Quartus Prime with Qsys, version 16.1.

# Background

For this lab, we will be using the soft-core embedded system that was designed in the previous lab. The block diagram for this embedded system can be seen in the figure below.

Diagram

Description automatically generated

## Soft Core Embedded System Design

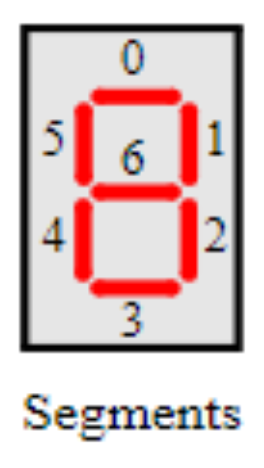
In order to write to a 7-segment display, we have two address spaces, one is for the four most significant bits and the second for the four least significant bits. This is because the DE2-115 board has eight 7-Segment displays. From the design in the last lab, the address spaces for these were, 0x2020 and 0x2030. The typical address spaces for the DE2-115 board can be seen below.

Diagram

Description automatically generated

## 7-Segment Display Address Spaces

Now, from this, we also need to know how to manipulate the 7-Segments in the way that we want. So, being that the display is an active low device, in order to light up a certain display we must pass a 1 into the bit that we want to light up. Now, the segments are controlled individually, so, passing a 7 bit value of 0b1111000 will turn on the bits corresponding to bit zero, one, and two. How these bits correspond to the actual 7 segment display can be seen below.



## 7-Segment Display Breakdown

With this knowledge we can begin this project.

# Project Overview

For this lab, we will be using the soft-core embedded system that was developed in the previous project in order to perform a variety of tasks on the DE2-115 board. The first of these tasks are converting a binary number to its decimal equivalent. Then, we will be creating a loop that will read the DIP switches, output the value on the 7-segment display, create a delay, and display the square on the LEDs. Next, we will be using a program to find the longest string of ones in a given word value. Then using this to work for a list of word values. And lastly creating a counter that works in the bits 10-15 on a given value.

.

# Project Procedure

**Part 1:**

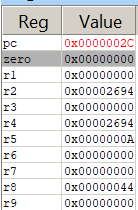
For the first part of this project, we are given a piece of code that converts a binary number to two decimal digits. We modify this code to be used with binary numbers of any size. Below we will walk through how this works, using the value of N = 9876.

Table

Description automatically generated

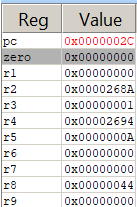
## Register Values Before Call

As is evident above, before we call the Divide Subroutine, R4 is loaded with the value 0x2694. Which is the hexadecimal equivalent of the decimal 9876. This value gets passed into the divide subroutine. Upon entering the divide subroutine, initial values for registers are set. This can be seen in the figure below.



## After Entering the Divide Subroutine

Here we can see that r2 which is our remainder is loaded with the input value. r5 is our divisor, so that is loaded with 10, and r3 is our quotient, so to begin that is loaded with the value 0. Now, after the first loop of the subroutine CONT we have the values seen in the figure below.



## Values After the First CONT loop

As we can see from above, the quotient, r3 has been incremented by 1 and the remainder been decremented by 10. This program will continue to perform this action until the value in our remainder is less than the value of our divisor. As, at that point, it is no longer divisible. The value will be stored in the remainder register, and the quotient will be accurate. After running through this we can see our final values below.

Table

Description automatically generated with medium confidence

## Part 1 Final Values

As we can see here our result is 3DB with a remainder of 6. Which in decimal is 987. When dividing 9876 by 10 we can expect this result. For part two, we will be reading the 4 bit DIP switches on the board and displaying this value of the 7 segment, then after a delay, display the square of this on the LEDs.

**Part 2:**

The code for this section can be found in appendix B. For the first part, reading the 4 bit DIP switches SW0-SW3, was pretty simple. We have the address of the DIP switches from creating the system in lab 3. We load the byte from the dip switches into a register, in this case r4, in which we and with 0b00001111 in case the dip switches after SW3 are turned on. We can see the state changes in the figure below.

Table

Description automatically generated

## Value of Switches being loaded into r4

Upon loading this value, we have a look up table for all the bytes that correlate to the hexadecimal value. Since they are in order, we add the first address value and simply add the hexadecimal value, to get the address of the byte sized binary value in which to load to the 7 Segment Display.

Table

Description automatically generated with medium confidence

## Loading the Byte Sized Value for the 7 Segment

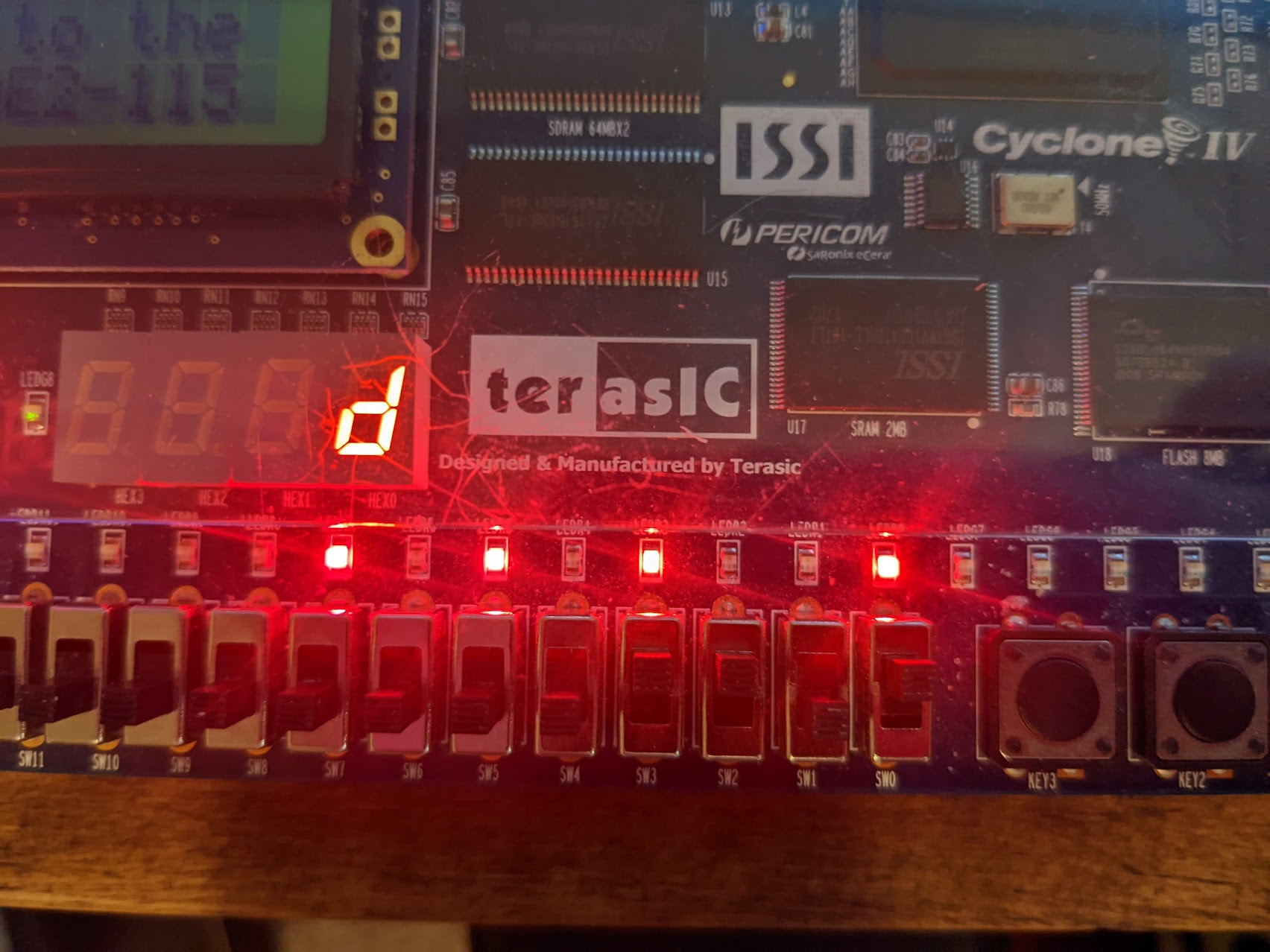
We can see here that r4 changes to the address 50D. The value in memory at this location is then loaded into r7. We can see here that that is 0xFFFFFFA1 which is the 7-segment display code for d. This value is then stored in the display, and the board looks like the following.

A screenshot of a video game

Description automatically generated

## Board After Value is Loaded into 7-Segment

Now that this has been loaded into the 7-Segment display, our next task is to have a delay and display the square of this value on the LEDs. This task is completed by having a delay loop, in which we iterate from 0x04C4B40 to 0. When this reaches 0, we return. This creates a delay of about half a second. However, this method is not as accurate as the board interrupt timers. After the delay, we enter the second loop, which takes the value of the Switches, multiplies them together, and outputs that value onto the LEDs. The result can be seen in the figure below.

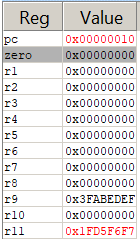


## Squared Value on the LEDs

Since our input value is 13, we are expecting the LEDs to output the binary equivalent of the decimal value 169. Our binary value on the LEDs is 10101001. Which does correlate to 169 in decimal. So, this works as intended. Next, part three is to count the longest string of 1’s in a hexadecimal number.

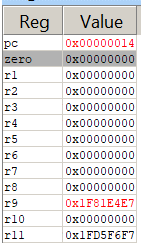
**Part 3:**

We begin part three by loading the program shown in appendix C. This program will count the longest string of 1’s in a word of data. Stepping through the program gives the following register values.



## After SRLI

After SRLI we can see that the 3FABEDEF changes to the 1FD5F6F7 stored in r11, In binary this is shown as the following: r9 -> r11  
0011 1111 1010 1011 1110 1101 1110 1111 -> 0001 1111 1101 0101 1111 0110 1111 0111   
As we can see here, the value gets shifted to the right once. Now, r9 and r11 are anded together and the following registers are created.



## After And

After the and occurs we see that r9 changes to 1F81E4E7 Using the R11 in binary from before this in binary looks like the following:

0001 1111 1000 0001 1110 0100 1110 0111

As we can see the longest string of 1’s the first string of 1’s has shrunk by one, which is by design. Since we shifted to the next 1, and anded it with 11, any strings that were preciously just a single bit long, are now gone. And the greatest string length is over 1 bit long. Next we count r10 by 1 and loop. Seen below.

A picture containing table

Description automatically generated

## After r10 is iterated

Now that we understand this loop, we will fast forward a bit into where the first 0 is. This is seen in the figure below.

Table

Description automatically generated

## Starting Values Entering the Fourth Loop

For reference, we can see here that the R9 is 0x7806021, and our r11 is 0x07C07031. In binary this computes to:

r9: 111 1000 0000 0110 0000 0010 0001 r11: 111 1100 0000 0111 0000 0011 0001

Now, after the logical shift right, and the and, our r9 and r11 values look as they do below.

Table

Description automatically generated with medium confidence

## Fourth Loop After srli and and

Now, our r9 has become 0x3802000 and r11 has become 0x03C0310 converting these values to binary allows us to see it more clearly.

r9: 11 1000 0000 0010 0000 0000 0000 r11: 11 1100 0000 0011 0001 0000

We can see, all that is left is the two largest strings, the one that started at length 7 and the one that started at length 5. This program will continue until the r9 has completely been shifted and anded with its own shift to be 0. And will continue to count until our largest string length is stored in r10, seen below.

Table

Description automatically generated

## End of Part Three Program Registers

We can see here that r10 becomes 7 which from the start, was our largest string of 1s. We can test this program again using an r9 starting value of 0x103fe00f, here after doing the binary conversion we have:

0001 0000 0011 1111 1110 0000 0000 1111

So, we can see the highlighted section is our longest string of 1’s. This has a length of 9, so after running we should loop 9 times, and have an r10 value of 9. This can be seen in the registers below.

Table

Description automatically generated with medium confidence

## Part 3 Test Case Results

We can see in using this test case that our r10 value is as expected at a value of 9. Next, for part 4, we will be taking this code, adding more words, finding the longest string in any of these words, and outputting the result into the 7-Segment display.

**Part 4:**

To start this part, we take the code from part 3 and make it into a subroutine using r4 to receive the input data and r2 to return the result. We accomplished this using the code in Appendix D. Here we can see that we added 10 numbers to run through, and a few new subroutines. Start initializes values, and main is our main loop. If r2 is greater than r6 which holds our largest string of ones, we will put the value of r2 into r6. If it doesn’t it will skip that part, load the next value from the list, and continue the program. When the input value is 0 it will exit the loop. Lastly, we needed to display the value on the 7-Segment Display. This was accomplished in similar fashion as in part 2. We will now walk through this code below.

Table

Description automatically generated with medium confidence

## Registers before entering Ones Loop

As we can see above, the first value is loaded into R4, 3FABEDEF. This value has a string of ones, that is 7 bits long. So, upon exiting the Ones subroutine, we should see r2 as 7. Below is the register values after exiting the subroutine.

Table

Description automatically generated with medium confidence

## Registers after exiting Ones Subroutine

Here we can see that the r2 value is now 7. This is in line with what we expected from this value. before entering the ones subroutine again, the main program will run this can be seen in the figure below.

A picture containing table

Description automatically generated

## Before Entering Ones subroutine Second Time

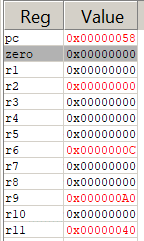
After the main program runs, before the call of the subroutine, we can see that the next value that was loaded into memory is now in r4, r6 now has the value of 7 which is now the longest string of ones, and r2 has been reset to 0. Now, 0xFF12384F has a string of 8 ones, so we should expect here that r2 will be 8 after exiting the subroutine. The registers at this point can be seen in the figure below.

A picture containing table

Description automatically generated

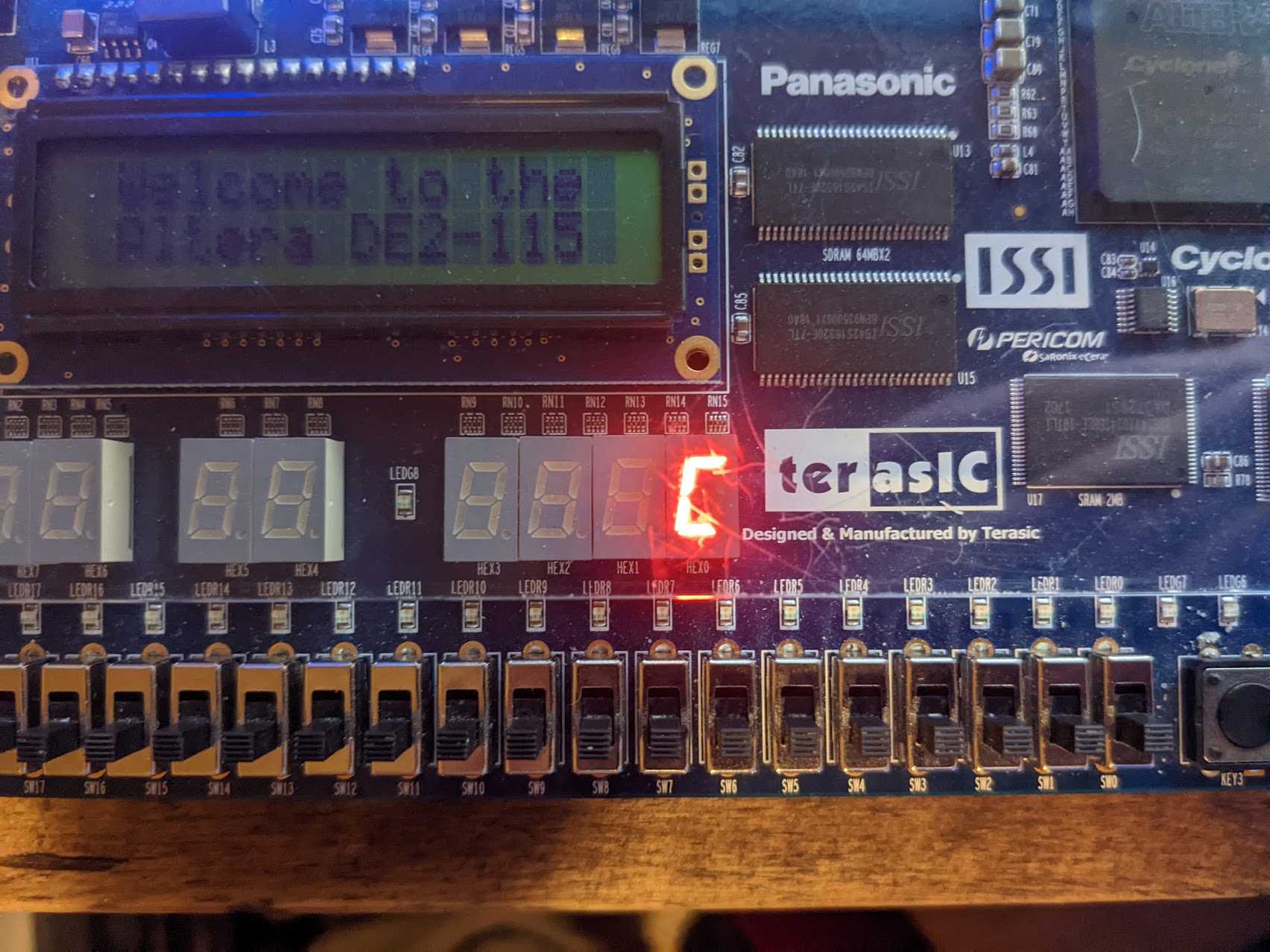
## After Exiting Ones Subroutine

As this program continues, we will encounter the 0xFFF00000 value which has 12 1s. So the value of r6 should be loaded with C. We can see this in the figure below.



## Final Values Of Program

As we can see above, the r6 value has acquired the value of C which is 12, which is the largest string of ones that is in our list. So, before the program ends we use a similar process to part 2 in order to store this value in the 7-segment display which is seen in the figure below.



## 7 Segment Part 4

We can see that the C from the program running above, was correctly stored in the 7-Segment Display. Lastly, for part 5, we will be using a word sized PIO device to make a 6-bit counter using bits 10-15.

**Part 5:**

To begin part 5, we first assume a 32 bit PIO device in which we form a 6-bit counter starting from 0, using bits 10-15. This code can be seen in Appendix E. So, starting with a value from memory, we are to not change anything other than bits 10-15. The starting registers after initialization can be seen below.

Table

Description automatically generated

## Starting values after Initialization

Here we can start to analyze the code, r2 and r3 are masks of bits 10-15 and r6 is the mask for everything other than bits 10-15. So, in r2 and r3 bits 10-15 are the only 1s, and r6 is the opposite, everything other than 10-15 are 1s. r1 is loaded with our value from memory. We then and r1, with r2, and store that value in r2, this is our starting count value. From which we then call the count function. In count, we add 0x400 to our bits 10-15, unless the value is equal to r3 which is all ones. Which it will be reset to 0 and return. So, at the end of count, the value has been incremented by one, then stored in the LEDs. We can see the value incremented in the figure below.

Table

Description automatically generated

## Registers after Returning from Count

After returning from the count subroutine, we can see that R2 has become 0xF000 which means it has become 111100. This value was then incremented to 0xF400 which was 111101 in bits 10-15. Then promptly displayed on the LEDs which can be seen in the figure below.

A picture containing graphical user interface

Description automatically generated

## Part 5 LEDs

Now, from above, we can see that the LEDs 10-15 are properly lit with the value of 0xF4, or 111101. From the aforementioned section. This completed part 5, and the lab as a whole.

# Conclusion

This lab accomplished its goal of using the softcore embedded system formed in the last exercise, and Intel’s FPGA design enviroment in order to complete certain tasks. We performed binary to decimal conversions, displayed an input value from the switches on the 7-Segment Display, and the square of that value on the LEDs, counted the longest string of ones from a list, and created a counter in certain memory bits. This further bolstered the knowledge of PIO devices and how they work in a NIOS II coding enviroment.

# Appendix A (Part 1 Code)

/\* Program that converts a binary number to decimal, you can modify the program \*/

.text

.global \_start

\_start:

movia r4, N

addi r8, r4, 4 # r8 points to storage location

ldw r4, (r4) #r4 loads N

call DIVIDE #parameter for DIVIDE is in r4

stb r3, 1(r8)

stb r2, (r8)

END: br END

#Subroutine can be modified if needed\*/

# Subroutine to perform the integer division r4 / 10 \*/

# Returns: quotient in r3, and remainder in r2 \*/

DIVIDE:

mov r2, r4 # will be the remainder

movi r5, 10 #divisor

movi r3, 0 #r3 will be the quotient

CONT:

blt r2, r5, DIV\_END

sub r2, r2, r5

addi r3, r3, 1

br CONT

DIV\_END:

ret

N: .word 9876 #the decimal number to be converted

Digits: .space 4 #space for storage location

.end

# Appendix B (Part 2 Code)

.data

LUT:

.byte 0b11000000#0

.byte 0b11111001#1

.byte 0b10100100#2

.byte 0b10110000#3

.byte 0b10011001#4

.byte 0b10010010#5

.byte 0b10000010#6

.byte 0b11111000#7

.byte 0b10000000#8

.byte 0b10010000#9

.text

.equ RLEDs, 0x2040

.equ SWITCHES, 0x2010

.equ GLEDs, 0x2030

.equ HexLSB, 0x2020

.equ HexMSB, 0x2000

.global \_start

\_start:

movia r2, RLEDs #LED Add

movia r3, SWITCHES #Switch Add

#movia r5, GLEDs #Green LEDs

movia r6, HexLSB #LSB Hex Bits

movia r5, 0x500 #LUT

movia r13, 0x1

br LOOP

DELAY:

movia r11, 0x0 #Initialize R11, our iterator

movia r10, 0x04C4B40 #what we have to iterate to

Delay2:

bgt r11, r10, LOOP2 #Compares iteration 1 instructino

add r11 ,r11, r13 #1 #Adds one to the iteration, 2 instructions

movia r12, 0x0 # 3 instructions (Filler code)

add r12, r12, r12 # 4 instructions (FIller Code)

br Delay2

LOOP:

ldbio r4, (r3) #Switch State into R4

add r4, r4, r5

ldbio r7, (r4)

stbio r7, (r6)

br DELAY

LOOP2:

ldbio r4, (r3)

mul r4, r4, r4

stwio r4, (r2)

br LOOP

.end

# Appendix C (Part 3 Code)

.text

.global \_start

\_start:

ldw r9, TEST\_NUM(r0)

mov r10, r0

LOOP: beq r9, r0, END

srli r11, r9, 0x01

and r9, r9, r11

addi r10, r10, 0x01

br LOOP

END:

br END

TEST\_NUM:

.word 0x3fabedef

.end

# Appendix D (Part 4 Code)

.text

.global \_start

\_start:

movia r9, TEST\_NUM

mov r10, r0

movia r6, 0

movia r2, 0

main:

ldw r4, (r9)

bgt r6, r2, skip

mov r6, r2

skip:

movia r2, 0

beq r4, r0, end

call ONES

addi r9, r9, 4

br main

ONES:

beq r4, r0, ENDONES

srli r11, r4, 0x01

and r4, r4, r11

addi r2, r2, 0x01

br ONES

ENDONES:

ret

end:

movia r1, 0x10000020

movia r7, LUT

add r7, r7, r6

ldb r6, (r7)

stb r6, (r1)

ENDLOOP:

br ENDLOOP

TEST\_NUM:

.word 0x3fabedef

.word 0xFF12384F

.word 0x99FCAB94

.word 0x10219341

.word 0xFCABEFCA

.word 0x12345678

.word 0xFFF00000

.word 0xF1231112

.word 0x00FEABCA

.word 0xF1234F87

.word 0x00000000

LUT:

.byte 0b11000000#0

.byte 0b11111001#1

.byte 0b10100100#2

.byte 0b10110000#3

.byte 0b10011001#4

.byte 0b10010010#5

.byte 0b10000010#6

.byte 0b11111000#7

.byte 0b10000000#8

.byte 0b10010000#9

.byte 0b10001000#A

.byte 0b10000011#b

.byte 0b11000110#C

.byte 0b10100001#d

.byte 0b10000110#E

.byte 0b10001110#F

.end

# Appendix E (Part 5 Code)

.text

.equ RLEDs, 0x2040

.global \_start

\_start:

movia r3, 0b1111110000000000

movia r6, 0b11111111111111110000001111111111

movia r4, RLEDs

movia r1, 0x3fabedef

movia r2, 0b0001111110000000000

Loop:

and r2, r2, r1

call count

and r1, r1, r6

or r1, r1, r2

br Loop

count:

beq r2, r3, reset

addi r2, r2, 0x400

stw r2, (r4)

ret

reset:

movia r2, 0x0

stw r2, (r4)

ret

END:

br END

.end