VLSI Practical 3 – Multi-Processor SoC

Harry O'Toole	***B00128907***
Violet Concordia	***B00125142***
Bernardo Borile	***B00127982***
Kacper Czajka	***B00123855***

Department of Engineering
School of Informatics & Engineering
Technological University Dublin, Blanchardstown
Dublin 15.

VLSI System Design
5th of May, 2023

Table of Contents

Table of Figures

Table of Figures

Introduction

Objectives

- The aim of this practical is to introduce the learner to the design and construction of a VHDL model of a Multi-processor System on a Chip (MpSoC)
- This practical is designed to be worked on in the Electronics laboratory during the time-tabled practical time.

Design Requirements

The MpSoC should be able to:

- Add/Subtract numbers to a maximum of 9999 (4 digits, each from 0-9),
- Accept user inputs of numbers and operators via a keypad or a keyboard,
- Transmit the result to another groups 7-segment display,
- Receive and display results from another groups calculator.

The system was implemented on a Digilent Basys 3

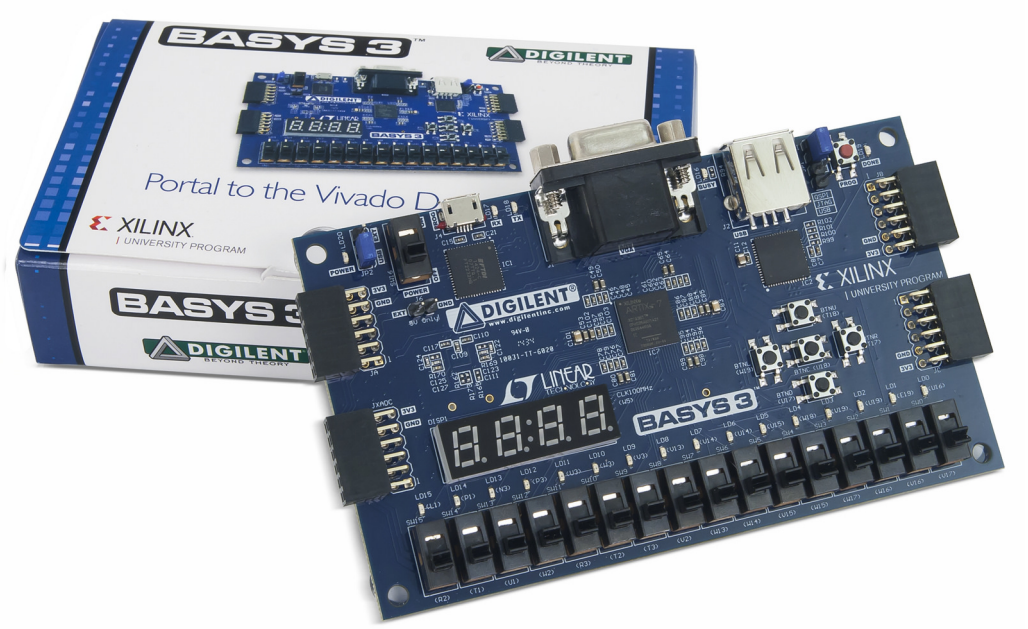

Figure 1: Xilinx Basys 3 Artix-7 FPGA Board [3]

System Block Diagram

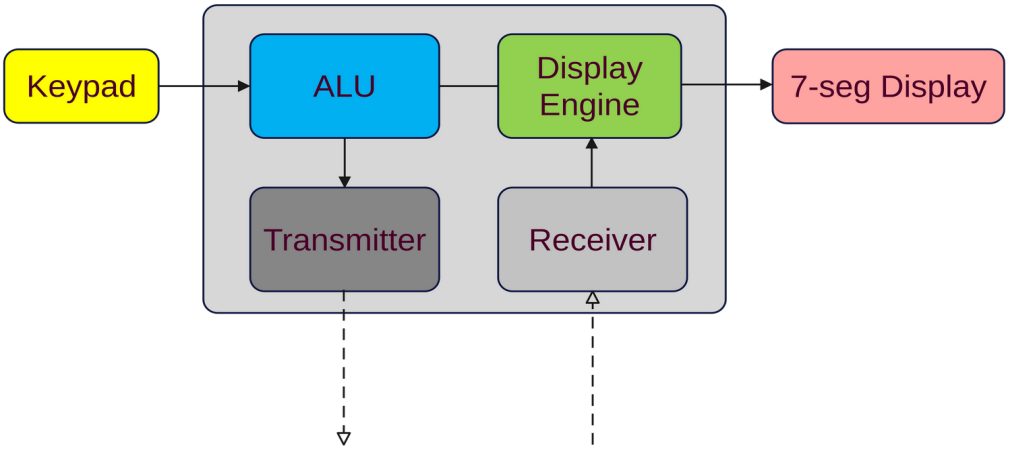

Figure 2: System Block Diagram

Testing procedures

The testing approach for each component of this assignment will be carried out using Vivado's built-in simulation software, which analyzes the bit levels of each input and output connected to the system. By comparing the results shown on screen with the ones derived from the design phase of this project, it will then be possible to determine if the system is functioning as intended or not. This procedure will then be repeated until the desired outcome is achieved for each module specified previously on the system block diagram.

In addition to testing each component of the assignment using Vivado's built-in simulation software, it is also important to conduct functional testing of the complete system. This involves testing the integrated system as a whole and verifying that all the individual modules are working together seamlessly. Functional testing can be done through a combination of simulation and physical testing using test equipment.

For example:

- One of the main test procedures in this project involved connecting a USB keyboard to the USB port of the FPGA board to test the functionality of the input module. This test was important to verify that the keyboard interface was working correctly and that the FPGA board was able to receive input from an external device.
- During the test, various keys on the keyboard were pressed and the corresponding hex codes were displayed on the 7-segment display. The purpose of this test was to ensure that the FPGA was properly decoding the input signal and converting it into the correct output format.

Key components

This project can be broken down into 4 main components:

- An input processor to receive the corresponding digits and operators,
- Display driver to process the output of the result onto a 7-segment display,
- An 'ALU' (Arithmetic Logic Unit) to calculate the operation,
- A transmit/receive module to handle the results of the other groups calculator.

Project Design

Input Processor

The system will take a input from a keyboard. The user will use the NumPad on the keyboard to insert digits and operations into the system.A USB-HID library is used to decode the inputs from the keyboard to the input processor.

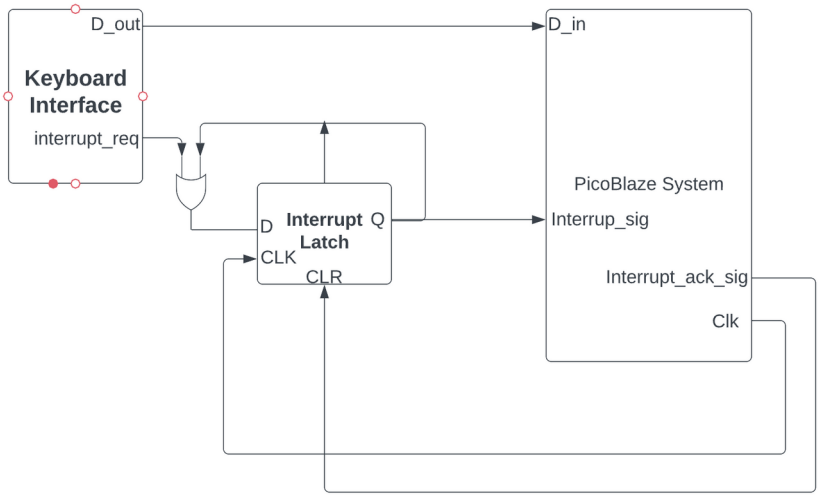

Figure 3: Project design: Input Processor

Display Driver

- The display driver receives a binary input
- It then converts it to decimal by splitting the input into thousands, hundreds and tens, while keeping track of the count for each one

The proposed design of the input system can be seen in the figure above. The input processor consists of 3 components:

- The keyboard interface which will send a 8-bit number in binary to the Pico blaze processor. It is important to note that each key on the keyboard has a corresponding HEX value. The said value must be converted to binary and sent to the Pico Blaze. After each press of a key the HEX value E0 is sent to the interface to tell the system that the press has ended. This can be used to trigger an interrupt which will wake the system and listen for upcoming digits.
- The interrupt latch is a d-flip flop. It is responsible of keeping track of the incoming interrupts in the system.
- The Pico blaze takes in the 8 bit number and stores it in a register. When all the numbers and the operation is entered, the Pico blaze will perform the calculation and then send it to the display driver.

Binary to Decimal representation coding

The assembly code takes a binary number in 16 bits of data and converts it into a decimal number with 4 digits. Here's how it works:

1. The input is stored in register s0.
2. Registers s1 to s5 are initialized to 0.
3. The code counts the number of thousands, hundreds, tens, and units by subtracting the appropriate value from the input and incrementing the corresponding register until the remaining value in the input is less than the next place value.
4. The values in s4, s3, s2, and s1 are then shifted left and concatenated into a single 16-bit value in s5.
5. The resulting value in s5 represents the 4-digit decimal equivalent of the binary input, with the leftmost 4 bits being the thousands place and the rightmost 4 bits being the units place.

See commented code on the next page:

```
start_binary_to_decimal:
   input   s0, 01  ;input

   load    s1, 0   ;units
   load    s2, 0   ;tens
   load    s3, 0   ;hundreds
   load    s4, 0   ;thousands
   load    s5, 0   ;output

;s9 is used for the compare command, because compare destroys the value it compares

   load    s1, s0
count_thousands:
   load    s9, s1  ;load s1 to s9
   compare s9, 3E8 ;compare s9 to 1000
   jump    nz, count_hundreds  ;jump if less than
   add     s4, 01  ;increment thousands count
   sub     s1, 3E8 ;subtract 1000 from s1
   jump count_thousands

count_hundreds:
   load    s9, s1 ;load s1 to s9
   compare s9, 64 ;compare s9 to 1000
   jump    nz, count_tens  ;jump if less than
   add     s3, 01 ;increment hundreds count
   sub     s1, 64 ;subtract 100 from s1
   jump count_hundreds

count_tens:
   load    s9, s1 ;load s1 to s9
   compare s9, A  ;compare s9 to 1000
   jump    nz, finish_count  ;jump if less than
   add     s2, 01 ;increment tens count
   sub     s1, A  ;subtract 10 from s1
   jump count_tens

finish_count:
   SL0     s4, 12  ;shift left 12 times, put 0 for the least significant
   SL0     s3, 8   ;shift left 8 times, put 0 for the least significant
   SL0     s2, 4   ;shift left 4 times, put 0 for the least significant
   ;example: if s4, s3, s2, and s1 are 1111
   ;s4 becomes 1111000000000000
   ;s3 becomes 0000111100000000
   ;s2 becomes 0000000011110000
   ;s1 remains 0000000000001111

   add     s5, s4
   add     s5, s3
   add     s5, s2
   add     s5, s1

   ;example continuation:
   ;s5 becomes 1111111111111111
   ;aka 4 bits of thousands, 4 bits of hundreds, 4 bits of tens, and 4 bits of units
   ;Result: Representation of a 4 digit decimal number in 16 bits of binary data
   ;[thousands] [hundreds] [tens] [units]
```

Drawing 1: PicoBlaze [2] assembly binary that converts binary numbers to a 4 digit decimal representation of units, tens, hundreds, thousands.

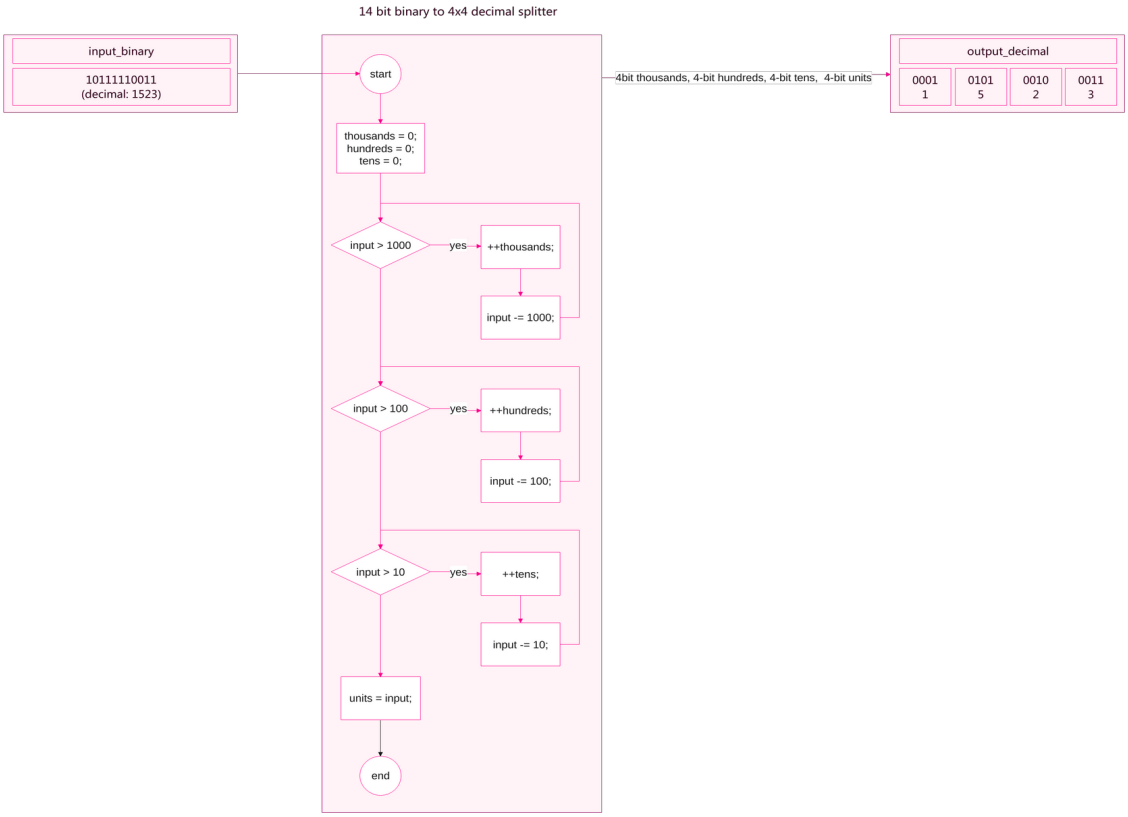

Figure 4: 16 bit binary to 4x4 decimal representation decoder. Made using EdrawMax

The decoder operates by decrementing the highest 10^n value until none remains, and then moving onto 10^n-1, until n == 0. The value n is the number of digits minus one that the decimal number represents. Because we know we are only dealing with 4 decimal digits, we start at thousands, or 10^3, until we reach 10^0. This method allows us to

For example: For the number 1523,

1. Is the number 1523 more than 10^3, or 1000? Yes. Increment thousands count by 1, decrement number by 1000. Is the number 523 more than 10^3, or 1000? No. Continue to hundreds.
2. Is the number 523 more than 10^2, or 100? Yes. Increment hundreds count by 1, decrement number by 100. Repeat until no hundreds remain, until the number is 23. Notice how this is the exact same step as step 1, but with hundreds.
3. Repeat the same step 1 again, but for tens: Is the number 23 higher than 10? And then finally, the unit count remains.
4. We can now reassemble this information into the following format:
4 bit Thousands count 4 bit Hundreds count 4 bit Tens count 4 bit Unit count, to a total of 16 bits to represent the 4 decimal digit number.

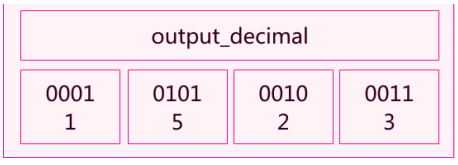

Figure 5: 1523 stored in 4x4 decimal representing bits

5. Now for each of the 4 7-segment display units, we decode the according number using a simple switch statement.

Binary to 7-segment LCD decoding

```
process(LED_BCD)
begin
    case LED_BCD is
    when "0000" => LED_out <= "0000001"; -- "0"
    when "0001" => LED_out <= "1001111"; -- "1"
    when "0010" => LED_out <= "0010010"; -- "2"
    when "0011" => LED_out <= "0000110"; -- "3"
    when "0100" => LED_out <= "1001100"; -- "4"
    when "0101" => LED_out <= "0100100"; -- "5"
    when "0110" => LED_out <= "0100000"; -- "6"
    when "0111" => LED_out <= "0001111"; -- "7"
    when "1000" => LED_out <= "0000000"; -- "8"
    when "1001" => LED_out <= "0000100"; -- "9"
    when "1010" => LED_out <= "0000010"; -- a

    when "1011" => LED_out <= "1100000"; -- b
    when "1100" => LED_out <= "0110001"; -- C
    when "1101" => LED_out <= "1000010"; -- d
    when "1110" => LED_out <= "0110000"; -- E
    when "1111" => LED_out <= "0111000"; -- F
    end case;
end process;
```

Figure 6: The 4 bit Binary to 7-segment (7 bit) display decoder.

- The input `LED_BCD` is a 4 bit number.
- The output is a 7 bit coded value, each bit representing if an LED segment should be on or off on the output LED display. Those segments form shapes that represent the corresponding number/letter.

Arithmetic Logic Unit

This module is responsible for handling the actual add and subtract operations

- For dealing with the operations, it was simply a matter of using the ADD and SUB functions that are built in assembly for the scope of this project.
- The next step required multiple Comparisons to be made to check the inputs of the user and match these inputs the correct operators.

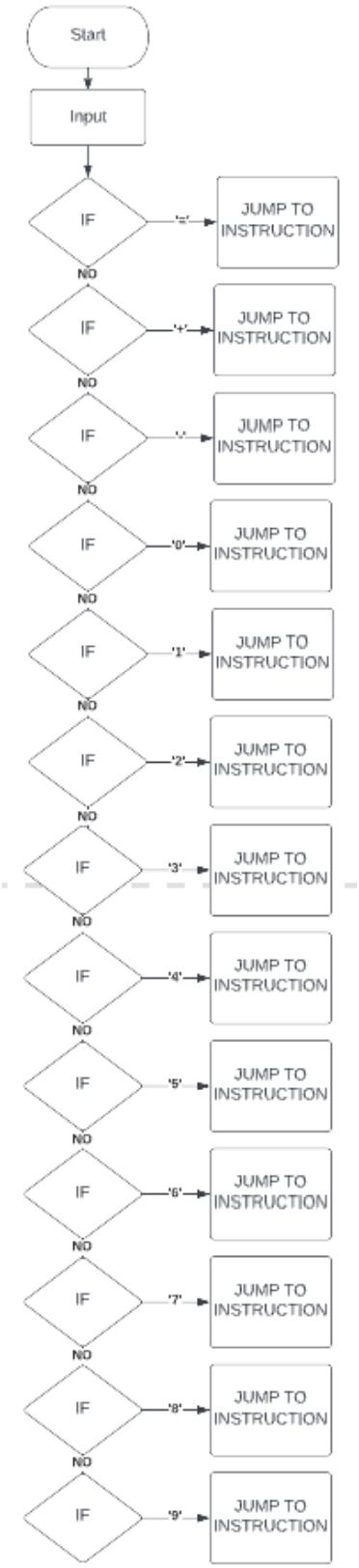

Figure 7: Project design: Instruction decoding from key input

Receive/transmit module

Discussion

Unfortunately, the full practical specification was not achieved due to several hurdles along the way:

- The IDE used, Vivado [1] throws errors that are incredibly technical, and difficult to investigate as students that were first introduced to VLSI in only the previous semester.
    - While this is a challenge on its own, a particular issue has thrown off the workflow significantly more than an error that can actually be tracked down and correct, one that did not produce an error at all, but rather – cause the Synthesis process to never end.

The group was separated into two interchanging parts that allowed for asynchronous progression on several modules of the project simultaneously.

In the end, several parts of the project were achieved successfully, such as:

- Basic interrupt handling.
- Key input decoding.
- 4x4 bit decimal coding and display on the 7-segments display.
- Communication between VHDL and PicoBlaze through registers.

Conclusion

While the implementation part of the project was only partially successful, the challenge provided was a valuable learning experience, which illustrated the importance of both: planning, and the difference between theory and actually implementing the design on the system.

As a resource of experience and anexercise of problem solving in a group, this project was successful.

References

[1] “Vivado,” *Xilinx*. https://www.xilinx.com/support/download.html (accessed May 05, 2023).
[2] “PicoBlaze 8-bit Microcontroller,” *Xilinx*. https://www.xilinx.com/products/intellectual-property/picoblaze.html (accessed May 05, 2023).
[3] “Basys 3 Artix-7 FPGA Board,” *Xilinx*. https://www.xilinx.com/products/boards-and-kits/1-54wqge.html (accessed May 05, 2023).