Project Gummi

16-bit Microprocessor

Kazumi Malhan, Chris Petros, Justen Beffa, Marc, Nahed
Electrical and Computer Engineering Department
School of Engineering and Computer Science
Oakland University, Rochester, MI
E-mails: kmalhan@oakland.edu, ctpetro2@oakland.edu, jbbeffa@oakland.edu, mmnahed@oakland.edu

The purpose of this project is to show the process, methodology, and design choices to create a 16-bit microprocessor that performs basic arithmetic, logic, and bit wise functions. The unique contribution that this project offers is the user can monopolize the benefits of serial communication and use an intuitive user interface to manipulate data without having to interact directly with the microprocessor.

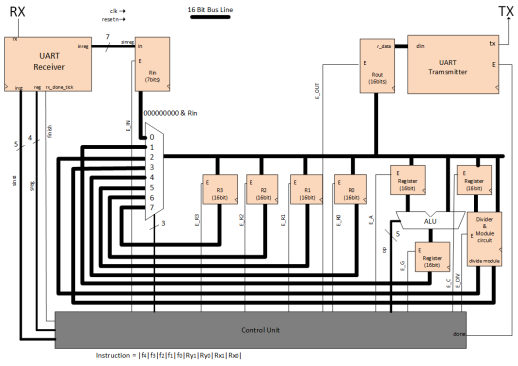
I. Introduction

The goal of this design project was to create a functioning 16-bit microprocessor which can store data into one of four user selected registers and give the user the ability to manipulate that data within the confines of 1 of 27 user selected functions.

A Universal Asynchronous Receiver/Transmitter (UART) handles the communication between the NEXYS 4 and the Visual Basic user interface. The user interface sends an Instruction Code to the microprocessor through 3 cycles of receiving data. The user interface reads data in the form of ASCII, which requires the UART to send 17 cycles of data from the transmitter back to the user interface. The result is then interpreted by the Visual Basic interface and displays the requested data in binary on the interface.

The motivation behind this particular project was to show a deep understanding how to design and implement a state machine and data path, which can effectively manipulate data in a meaningful way. The desire to create a microprocessor, which performs select arithmetic and logic, was born of a desire to create a user-friendly device that does not require the user interact with the NEXYS 4. The design choices made were done so that the user could easily manipulate a point and click interface, which can efficiently manipulate the data and produce desired results.

Figure 1: Top Level View of the Microprocessor



II. Methodology

A. Universal Asynchronous Receiver/Transmitter (UART)

To accomplish the goal of not interacting with the NEXYS 4 board during the operation, the UART plays a critical role. In this project, communication speed is set to be 9600 bps with 8 data bits per cycle. Therefore, the length of one bit becomes 1/9600 = 10.416 microseconds. The challenges we had were conversion of data between binary and ASCII, changing bit length, and repeating the process several times.

i. UART Receiver

The UART receiver has a modulo-651counter, which generates a pulse 16 times per each bit length. By dividing each bit 16 times, it can minimize the reading error. Once the receiver starts to get "0" from the UART, it checks 7th segment sample to confirm whether it is start bit or not. After receiving start bit, then the receiver takes the 15th segment sample of bit data and reads its value. A modulo-8 counter keeps track of how many data bits the system receives. Finally, it waits for a stop bit to finish the cycle of communication.

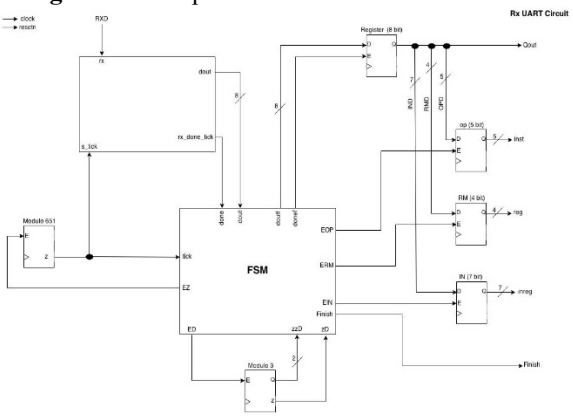
The instruction set for this microprocessor consists of a 5-bit operation instruction, a 4-bit register selection, and a 7-bit binary input. Since the UART can send only 8 bits per cycle, it is required to repeat the communication three times. In this project, the 1st cycle sends the operation instruction, the 2nd cycle sends the register selection, and the final cycle sends input binary numbers.

To send an instruction from computer, we convert each instruction to appropriate ASCII character, and then send that value to the NEXYS 4 board. For example, when the operation instruction is "01110" (copy), then the ASCII character containing "01110" in its least five bits is selected, transmits to board. In this case, N = "010**01110**" is sent to board. The board strips the least 5 bits and keeps the operation instruction in a register. For register selection, the system finds an ASCII character containing 4-bit selection code in least 4 bits. The board, it stripes least 4 bits and stores to register selection register. For 7-bit binary input, at the interface side, it converts binary number to decimal number and then sends the ASCII character that has the same decimal representation. For instance, if we sends "1001101"

= 77, the system sends Chr(77) = "M". At the board, it strips the least 7 bits and stores to input register. When all communication is completed, the system generates a finish signal to microprocessor control unit, and sends each instruction. These conversions are necessary because the NEXYS 4 board treats everything in binary ("1" or '0").

One challenge we faced was incorporating the receiver block from the "FPGA Prototyping by VHDL Examples" book [1]. The code is treated as a black box and only concerning the inputs and outputs to the block. By placing another Finite State Machine (FSM) shown in figure 2, we were able to repeat the communication and receive specific bit combination.

Figure 2: Data path for UART Receiver



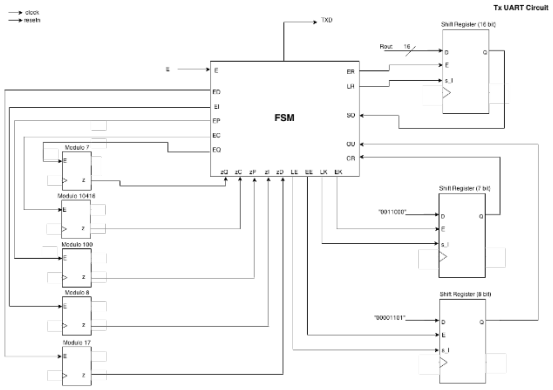
ii. UART Transmitter

When the UART transmitter receives an enable signal from microprocessor control unit, it begins to send the result back to the computer. The result from microprocessor is 16-bit binary number which is stored in 16-bit parallel access shit register, however, we cannot send just these 16 bits back to computer as the computer treats the data in ASCII character format. Therefore, it is required for transmitter to convert the 16-bit binary number to 16 ASCII representation of "1" and "0".

The point here is that ASCII character representation of "0" is "00110000" and "1" is "00110001". As first seven bits will be same for both numbers, the system contains a 7-bit parallel access shit register. For each bit, we concatenate common the 7-bit portion and the one bit from output shift register. That signal is then sent back to the computer. The output shift register is loaded only at the beginning of communication, and the common 7-bit shift register loads before sending each data bit. The system repeats it 16 times to send 16-bit binary number.

At the end of 16th cycle, the transmitter sends ASCII character for "carriage return" to the shift the cursor to the next line on serial interface result textbox. The data path of this system is shown in figure 3. To keep track of length of each bit, number of data bit sent, number of cycle transmitted, length between each transit, the system contains five different counters.

Figure 3: Data path for UART Transmitter

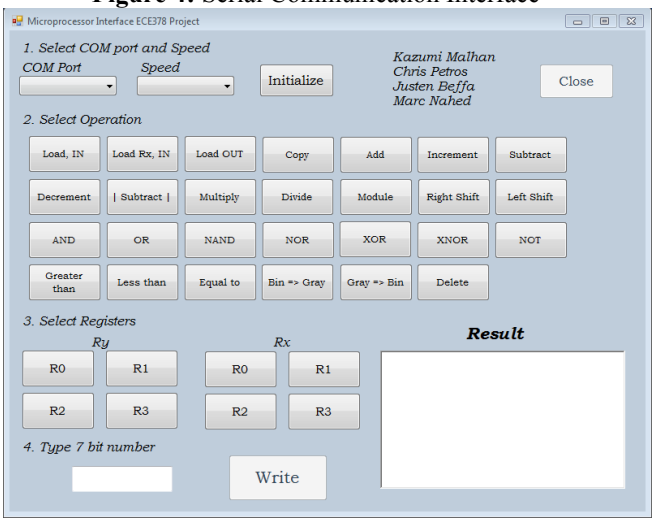


The challenge we faced on transmitting the data was the transmitting order of ASCII character. At the beginning, the serial interface was receiving different ASCII characters because the system sent the common 7-bit, then 1-bit from output shift register. We performed a functional simulation by reducing the most of the counters to 2. It is found that ASCII character should be send from least significant bit, and by sending 1 bit output first, then common 7-bit, we successfully solved the issue.

B. Serial Communication Interface on Computer

When it comes to UART communication, most projects use *"Putty"* as serial communication interface on computer side. However, it is difficult to use for first time users. To accomplish the goal the user does not need to read a manual to operate the processor, we have created our own serial communication interface for this project using visual basic shown in figure 4.

Figure 4: Serial Communication Interface



When the interface loads, it scans all COM ports on the computer, and displays the only available ports. After selecting the COM port and speed, the user can establish the connection. Establishing the connection enables the write button. The interface has an operation section, register selection section, and an input binary number section. Users

can simple select an operation, register, input bits. When the user hovers over the operation instruction, the help box appears and tells the user, which registers the result will be placed on.

There are different restrictions that are implemented to prevent the interface from crashing. The first restriction is that users can only select one button from each category. When the user selects another button in same category, the previous selection operation will be deactivated. The second restriction is that user cannot type anything in result textbox. The third restriction is that user can only type "1" or "0", up to 7 bits to input textbox. Input is converted to 7 bits by appending "0" even if user types less than 7 bits.

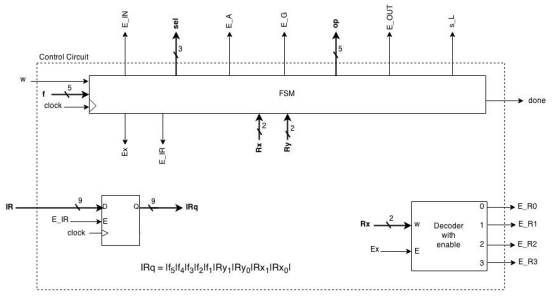
When the serial communication interface receives data from NEXYS 4 board, it goes through simple test before it appears on result textbox. Since only ASCII character coming from board is ether "1" or "0" or "Carriage Return", the system compares each ASCII character with these three possible character. If it does not match, system output "ERROR" instead of result.

C. *Microprocessor Control Unit*

The control unit performs several key functions for the operation of the microprocessor. The control unit manages the flow of data from bus to register, sends the operation code to the ALU, keeps the correct timing of every operation and finally tells the TX unit when to transmit data to the user interface.

Figure 5 shows the design of the control unit. The major components of the control units include a Finite State Machine (FSM), a two to four decoder and a 9-bit register.

Figure 5: Control Circuit Schematic



Initially the control unit waits for the finish signal of the RX unit and loads the instruction to the IR register of the control unit. This occurs during state S1.

State S2 is where the control unit, based on the instruction code, prepares to perform the desired operation. To achieve this the appropriate value for the multiplexer selector is sent out and the desired memory location is enabled. For some instructions like load to Rx and copy from Ry to Rx, this is enough and then the control unit transitions to state S27. At state S27, the control unit outputs a high "done" signal and waits for its enable signal (w in the coding) to return to '0'. When this occurs, the control unit's "done" signal tells the TX unit to begin transmission of data back to the user interface. An interesting fact is that this occurs at the end of every operation, so unless the load out operation is selected, the value displayed on the user interface is not a final value that the user may want.

For the vast majority of operations, the transition from S2 to S27 requires two additional states to process a single operation. For example, to implement a typical operation, the FSM concatenates '1' & Rx and sends the correct select signal to allow the value of Rx to be loaded onto register A of the ALU. From this point the Finite State Machine transitions to a state SQa. In state SQa a concatenation of '1' & Ry selects which value to load onto the main 16-bit bus and therefore is used as the B value within the ALU. (This is the method used to select Rx in the previous state.) Additionally, the control unit sends a specific operation code to the ALU and enables the G register. For all but the division operations SQa only requires one clock cycle. The result of the operation is typically loaded onto register G. From SQa the FSM transitions to SQb. Universally SQb selects for the value stored on register G to be loaded onto Rx. After one clock cycle the FSM transitions from SQb to state S27.

As mentioned before in S27, the control unit sends out a "done" signal, which acts as the initialization signal for transmission to the user interface to begin, and then transitions back to S1 waiting for a new operation to be given and processed.

D. *Operations*

The operations for the 16-bit microprocessor composed of a total of 27 operations, which consisted of 21 operations in the ALU, plus an additional 6 operations outside the ALU. The operations within the ALU were split into two categories: arithmetic and logic. The arithmetic portion consisted of increment (A+1), decrement (A-1), addition, subtraction, absolute subtraction, multiplication, left shift, and right shift. The logic portion consisted of 1's compliment, AND, OR, NAND, NOR, XOR, XNOR, greater than (outputs greater input), less than (outputs smaller input), equals to, binary to gray, gray to binary, and a reset function. The multiplier was the only operation out of the 27 operations that took in the least significant 8-bits out of the 16-bit input. The reason why the inputs were trimmed was to insure that the output from the ALU did not exceed 16-bits. All the operations in the ALU took a total of one clock-cycle to compute.

In addition to the 21 operations in the ALU, 4 of the 6 operations, consisting of load in, load in to a register, load out, copy, are computed by the 4 registers (R0, R1, R2, R3). The rest of the 6 operations, consisting of division and modulus, are computed in a separate component. The purpose of having the division and modulus functions outside the ALU is because these two functions require a

total of 16 clock cycles to compute. Therefore, if the user wants to carry out an operation from the ALU, there would be no reason to wait 16 clock cycles to compute the output when it could be computed in 1 clock cycle. The input for the two functions, as well as all of the functions from the ALU, comes from a register (A) and from the Bus (B).

III. Experimental Setup

During development, the project was divided into the following four components: UART receiver, UART transmitter, ALU, and control unit.

In order to verify the functionality of each operation of the ALU, a functional simulation was used to confirm each signal value. Then a timing simulation was used to check the timing. After these simulations, the ALU was combined with the control unit for further testing. Again, both functional simulation and timing simulation was performed to verify the basic functionality.

The control unit and the ALU were then implemented to the NEXYS 4 board. It took a 5-bit operational instruction, a 4-bit register selection, and a 7-bit input controlled by 16 switches, and outputting the result to 16 LEDs on the board. The testing was successful and was able to confirm that all functions were working properly within the scope of control unit.

Checking the functionality of the UART receiver was challenging, as input data came from outside the board. Since NEXYS 4 board uses a UART to program the board, it was not possible to loop back the output back to board for simulation. In order to check, the receiver goes through all of the states in the FSM, decreased the counter to 2, and performed the functional simulation. At this stage, we were able to discover that the receiver block from "FPGA Prototyping by VHDL Examples" book [1] resets the block when reset is "1". The problem was solved by flipping the reset signal going to the block. Then the code was implemented to the board, taking the input from the UART, and outputting the received instruction set to the LEDs. After a few modifications, we confirmed that the instruction sent from the computer was correctly displayed on the LEDs.

A similar process was used for the testing of the UART transmitter. An initial functional simulation was performed to check that the states in the FSM were moving at the expected timing. The transmitting order of a bit was also checked during this process by comparing the order with a working single ASCII character transmitter. Then the code was implemented to the NEXYS 4 board, taking inputs from the 16 switches, enable signal from a button, and outputting to the UART. A few modifications were done to a counter that counts the timing in between the transmitting cycle to minimize an extra waiting time.

After confirming the functionality of each component, everything was put together for the final testing. At this point, since we are using the input from the UART, we cannot do a functional or timing simulation. The code was implemented to the NEXSYS 4 board, taking inputs from the UART, and then outputting to the UART. The instruction code was displayed on the LED to confirm that the board received the correct instruction. By using a custom serial communication interface, every possible operational instruction and register combination was tested. After confirming all of the results, we concluded that our microprocessor system was functioning correctly.

IV. Results

Testing of the Microprocessor was done in several different stages. After the ALU was completed, a behavioral simulation was devised to test the 21 different operations that were contained inside the component. Once it proved to be successful, a top file, consisting of every component except the UART receiver and transmitter was created and tested. To insure that all bugs were accounted for, a behavior simulation, seen in Figure 6 (reference appendix), and a hardware implementation were used to test every operation. As seen in the behavioral simulation in Figure 6, 4 unique binary numbers were inputted into separate registers and multiple operations, which include left-shift, right-shift, addition, absolute subtraction, and multiplication respectively, were checked for correctness. Once both variations of testing were complete and successful, a final top file was created, consisting of the previous top file and the UART components. A third and final test was completed, testing all the operations using the Visual Basics graphical user interface to input the instructions as well as the inputs. With the help of the interface, the implementation as well as the testing proved to be much more efficient and user friendly than using the NEXYS 4 board. Overall, the Microprocessor proved to be successful and all bugs found were accounted for.

Conclusions

Throughout the designing stage, constructing stage, and the testing stage of the 16-bit Microprocessor, the group learned how to work together effectively. Upon completion of the project, a 16-bit microprocessor was successfully operational. Data was able to be loaded onto 1 of 4 registers and 1 of 27 operations could be used to manipulate the data. Users were able to enter data via the Visual Basics interface. Operations were clearly defined, along with clues dealing with which registers would be used during the selected operation. Once the user entered a 7-bit binary number, the data was transmitted using the UART, which handled the communication between the NEXYS 4 and the Visual Basic user interface. Once the microprocessor completed the operation, data was transmitted back to the Visual Basics interface and the result was displayed, thus the user never touched the NEXSYS 4 board.

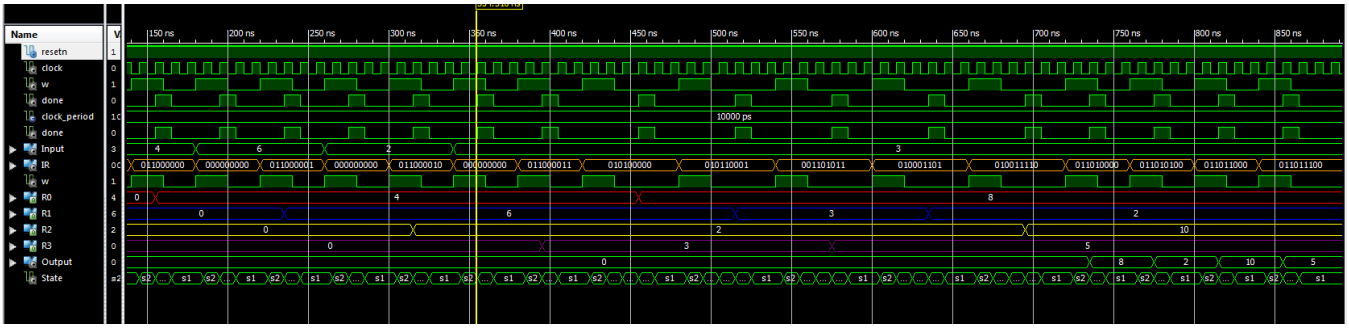
Overall, this project helped deepened the group's understanding of designing a microprocessor, constructing it, and finally testing it. It was satisfying to see how individual pieces of knowledge throughout the semester were combined and used to complete one project.

References

[1] P. P. Chu, "UART" in FPGA Prototyping by VHDL Examples, Hoboken, New Jersey: Wiley-Interscience, 2008, pp. 562–1

APPENDIX

Figure 6: ALU and Control Unit Testbench



Input and instructions for testbench in Figure 6 are as follows:

```
Input <= "0000100"; IR <= "000000000";    -- Load in 4
IR <= "011000000";                        -- Save in register R0
Input <= "0000110"; IR <= "000000000";    -- Load in 6
IR <= "011000001";                        -- Save in register R1
Input <= "0000010"; IR <= "000000000";    -- Load in 2
IR <= "011000010";                        -- Save in register R2
Input <= "0000011"; IR <= "000000000";    -- Load in 3
IR <= "011000011";                        -- Save in register R3
IR <= "010100000";                        -- Left-shift register R0, new value: 8
IR <= "010110001";                        -- Right-shift register R1, new value: 3
IR <= "001101011";                        -- Add register R2 with R3, new value in register R3: 5
IR <= "010001101";                        -- Absolute subtract register R1 from R3, new value in register R1: 2
IR <= "010011110";                        -- Multiply register R2 and R3, new value in register R2: 10
IR <= "011010000";                        -- Load Output R0
IR <= "011010100";                        -- Load Output R1
IR <= "011011000";                        -- Load Output R2
IR <= "011011100";                        -- Load Output R3
```