Digital Design -- Peak Detector Report

University of Bristol -- Digital Design Project

hi20791@bristol.ac.uk

Wenbo Deng

INTRODUCTION

This report is the coursework on Digital Design. The object of this coursework is to design a peak detector, which can print NNN number(based on command) data on the screen, print the peek value of this NNN data and print the neighbours of the peak value. And this peak detector has two parts-the command processor and the data processor.

*A. **Command Processor Overview:***

The command processor in UART consists of four primary blocks: data transmission, data reception, status, and control. These interconnected blocks work together to control and operate the data transmission and reception process.

*1) **Data Transmission**:* The data transmission block is responsible for transmitting the data to the external device. When the data is transmitted, it is sent bit-by-bit at a specific baud rate. The transmitter generates a start bit, followed by the data bits, parity, and stop bits. After the data is transmitted, the transmitter waits for the next data set.

*2) **Data Reception**:* The data reception block is responsible for receiving the data from the external device. When the data is received, it is stored in a register or sent to print. If there are no errors, the data is transferred to the next part to do the operation for further processing.

*3) **Status**:* The status block provides information about the status of the UART device. It indicates whether the device is ready to transmit or receive data and whether needs to wait. The status block also provides information about every used state's combination and work function.

*B. **Data Processor Overview***

The data consumer has three primary design sections: peak comparing and shifting, state machine, and data retrieval via the two-phase handshaking protocol.

*1) **Two Phase**:* The two-phase handshaking protocol focuses on the detection of changes in the control line signal. The two processors are ctrlIn_delayed and ctrlIn_detected. When the value of ctrlIn_detected equals 1, the start value is checked and the value of ctrlout_reg is set to "NOT ctrlOut delayed".

*2) **Peak Detector**:* Dataprocess compares data to find the maximum value, using two signals: max reg and maxIndex dec. When data is greater than max_reg, it is assigned to max_reg and the count corresponding to the value generated is assigned to maxIndex_dec.

COMMAND PROCESSOR

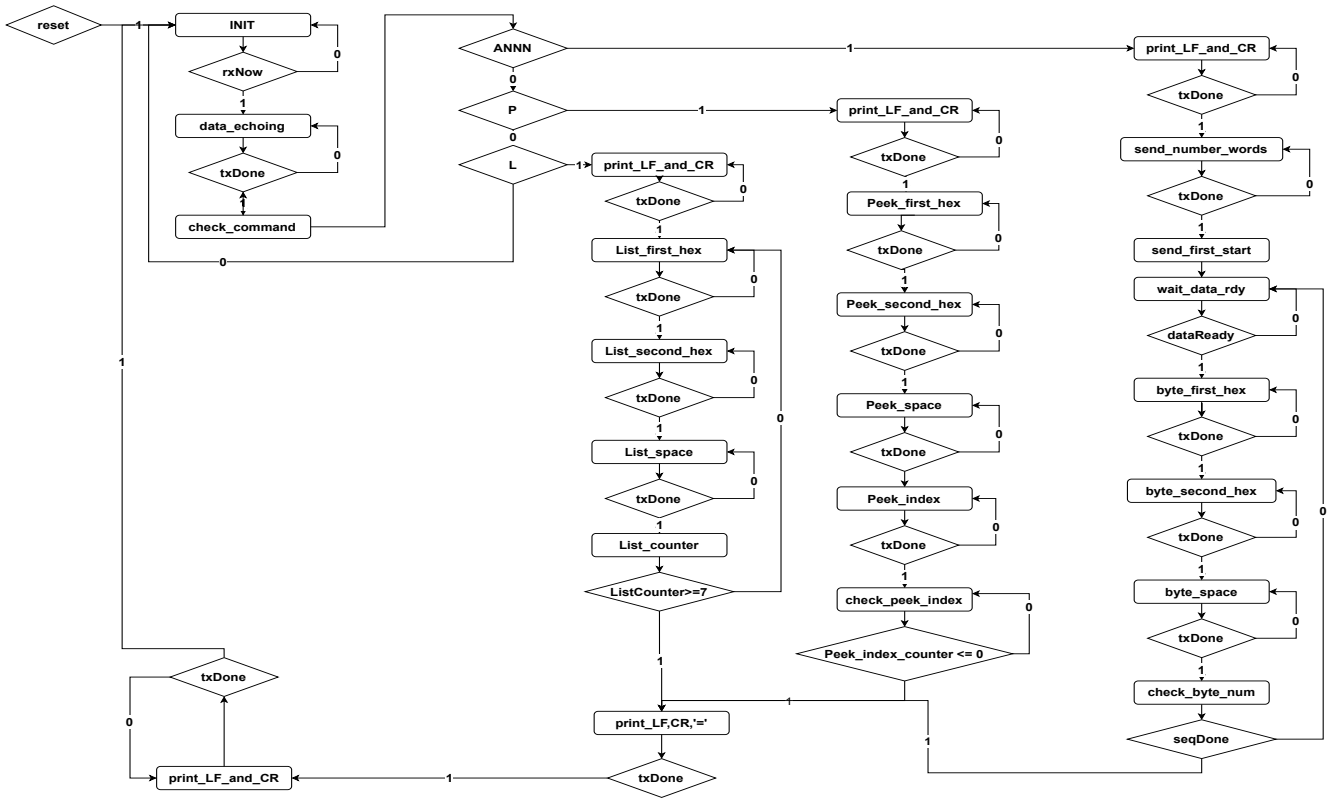

Fig. 1: The whole command processer ASM chart

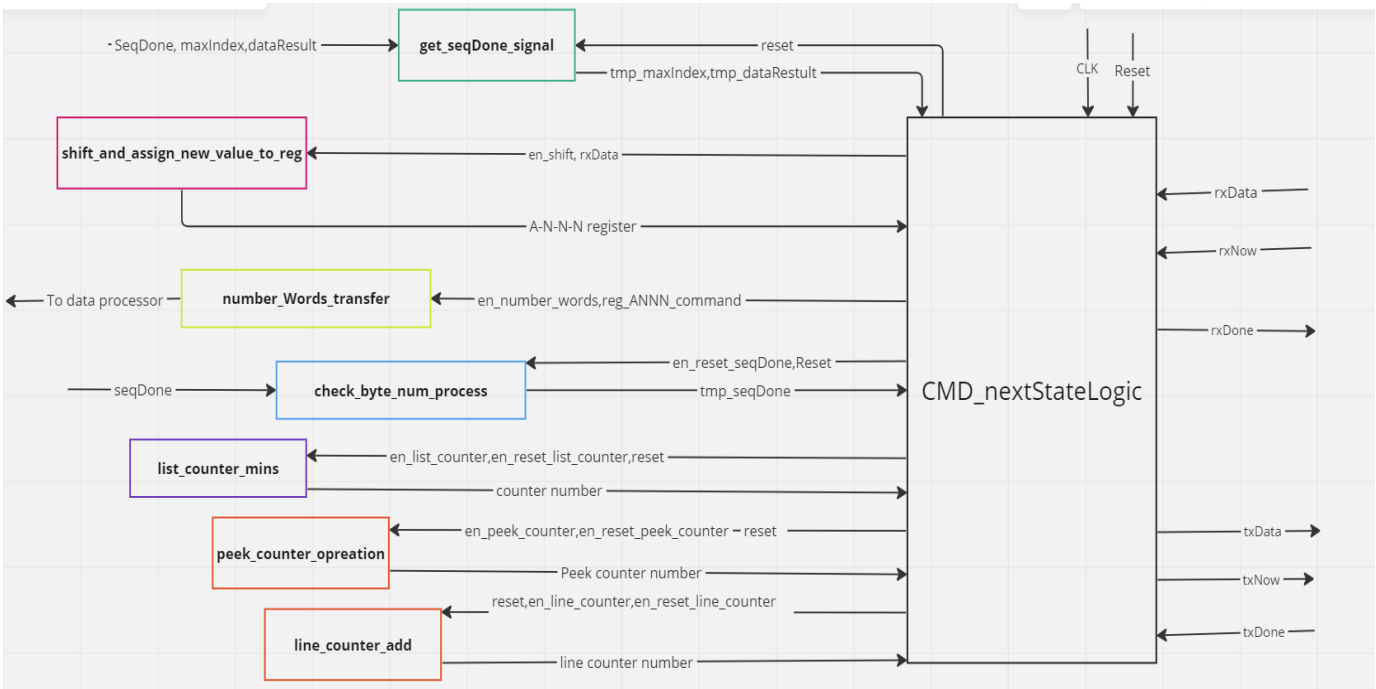

Fig. 2: The processes used in command part

Figure 1 is our entire command processor, consisting of several important parts: data echoing, ANNN register, and PL pattern. Each of these parts plays a crucial role in the processing of commands. And Figure 2 shows the process (except the

next-state logic process), which we write in our command processor part.

***Defined Function**:* The command processor has many data type conversions. To facilitate the data type conversion we have defined a function of our own. And the above figure 3 shows the function that can convert the BCD data type to ASCII data type, which command processor processes data from the data processor in preparation for transmission to the TX module.

```
function BCD_to_ascii(BCD_num :std_logic_vector(3 downto 0)) return std_logic_vector is
   --tansfer ascii to BCD
    variable result : std_logic_vector(7 downto 0);
  begin
    case BCD_num is
      when "0000" => result := "00110000"; --0
      when "0001" => result := "00110001"; --1
      when "0010" => result := "00110010"; --2
      when "0011" => result := "00110011"; --3
      when "0100" => result := "00110100"; --4
      when "0101" => result := "00110101"; --5
      when "0110" => result := "00110110"; --6
      when "0111" => result := "00110111"; --7
      when "1000" => result := "00111000"; --8
      when "1001" => result := "00111001"; --9
      when "1010" => result := "01000001"; --A
      when "1011" => result := "01000010"; --B
      when "1100" => result := "01000011"; --C
      when "1101" => result := "01000100"; --D
      when "1110" => result := "01000101"; --E
      when "1111" => result := "01000110"; --F
      when others => result := (others => 'X');
    end case;
    return result;
  end function;
```

Fig. 3: Defined Function–BCDToASCII

***Data echoing**:* Whenever anything is typed into the putty terminal, it is displayed on the screen first. This function is carried out by the data echoing component. The received data is then stored in a register for further processing.

***ANNN register**:* As data is typed, it is simultaneously stored in a 4-byte shifting register, essential for linking the command and data processors. Once 4 bytes, such as A-N-N-N, have been received, the numWords_BCD is sent to the data processor.

***NNN Byte Print**:* When the received ANNN or aNNN command, the command processor will send the numWords_BCD and start to the data processor to get the NNN numbers byte. After receiving data, the command processor needs to transfer these data into ASCII code type. Then send them to the TX module.

***Peak and List Command**:* The PL module receives P or L commands from the RX and returns the corresponding output to the TX. The P command returns the peak value and peak index generated by the data processor. The L command will list seven values generated by the data processor, including the peak value in the middle.

Print LF, CR and "=": In our design, there are some states to print the Line Feed("00001010"), Carriage Return("00001101") and "=" symbol("00111101"), which can make the print clearer on the screen.

Data-echoing part

The below ASM chart(see Figure 4) is our data echoing part, which starts from the INIT part. This ASM chart is our design, letting the data we received show on the screen individually. Then check the value of rxNow, which is from the RX. If it is high, the command processor must handle it to show it on the screen. Next, the command processer will need to set the txNow is high to tell TX that there has data coming and transmit the data from RX to TX. When these operations are finished, we can see on the putty that what we typed on the keyboard will show on the screen. And there have a signal 'en_shift', which will start a process-"shift_and_assign_new_valu_to_reg"(see Fig 2) to shift bit 2 to bit 0 to bit 3 to bit 1 and assign the new rxData to bit 0.

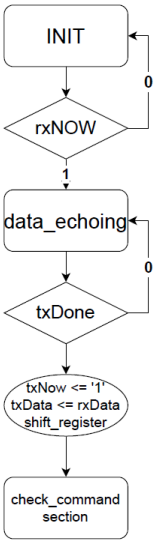

Fig. 4: Data-echoing

ANNN register

Our ANNN register(see Figure 5) shifts and receives data from RX. We define it as an ASCII array type. In our 4-byte shifting register, the positions 0-1-2-3 represent the array's byte positions from right to left. Before receiving any data, we shift all four bytes to their left positions, with the fourth position (3) being shifted to null. We don't want it to remain in the register, so once we finish shifting, we place the rxData into position 0. This creates a loop where all the bytes are shifted left and a new byte is added to the right.

Due to our design, it only cares about if the value that the 4-byte register stored is (A-N-N-N); if last any one byte is not correct for the command, it will just pass the incorrect byte to null and continue receiving new byte until it could be the A-N-N-N type we want.

This is a helpful and efficient part as it only uses a few lines of process parallel with the main process to do the work; if we don't have it, our next state part will be so long.

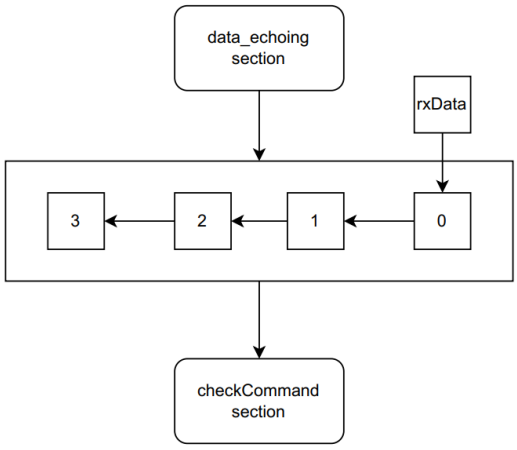

Fig. 5: ANNN Register

_ *Check Command*

The check_command component(see Figure 6) matches the register. If there are bytes in the register, it first checks if they can be used for the A-N-N-N part. If not, it waits and checks if they can be used for the P/L part.

If the third bit of the A-N-N-N register is A(”01000001”) or a(”01100001”), and the other three positions are number-0-9(”00110000” to ”00111001”), the next state will jump to ANNN part(there is a special case is A000 or a000, this will be excluded. Because these two commands should not print any data.). P and L are specific conditions that are easy to detect. Once a P or L command is received, it will first enter the zero-bit position of the A-N-N-N register. The check command state will monitor the zero-bit of the A-N-N-N register. If its value is P(”01010000”) or p(”01110000”), the next state will jump to the peak states. If its value is L(”01001100”) or l(”01101100”), the next state will jump to the list state.

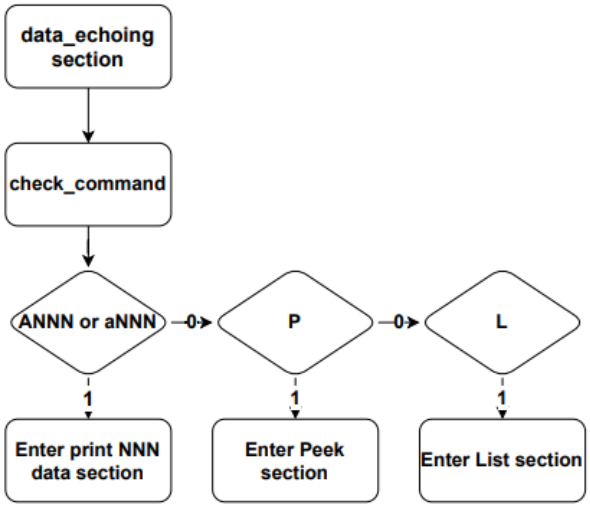

Fig. 6: Check Command

NNN Byte Print Part

Once data is received as A-N-N-N or a-N-N-N(see Figure 7), this is a correct command that could indicate the data processor to find numbers and send them to the command processor. To send the numberWords_bcd to the data processor, we used a process named ”number_Words_transfer”(see Figure 2), and the corresponding state will enable it. After that, the command process needs to send start, wait for data ready and send this byte to the tx module when the byte has been received.

Due to the showing principle, printing data is a hexadecimal value of 8-bit bytes(such as 95, C7). However, it needs to be shown as two 4-bit bytes. When the last data is finished(txDone is ok), assign txNow to high and send the 7 down to 4 bits(first byte) to txData. Finished this, the second byte (3 down to 0 bits) repeats the operation. After that, space should also be printed in the two data to distinguish them. So, assign txNow to high and give the space for the ASCII code(”00100000”) to txData.

While the above part is for one value printing, all value depends on the Rx module receiving. For example, the command is A-0-1-2, which will print 12 values from the data processor on-screen.

Also, to check that all values are printed over, there is a state to check if the command processor receives the seqDone signal from the data processor. To check the seqDone signal, we used a process named ”check_byte_num_process” to assign high to one tmp_seqDone signal. Because the seqDone signal only high one clock, we need to store in tmp_s and keep it high until the state returns to the INIT state. When the command processor detects the tmp_seqDone signal is high, the state machine will go to the print LF, CR and ’=’ section to print the separator. After that, the state will return to the INIT state to wait for the next valid command. In addition, a process named ”get_seqDone_signal”(see Figure 2) records the data result and max index of the peak value. For this process, it will record the data Result and max index and stores them to register to prepare for printing the p and l commands.

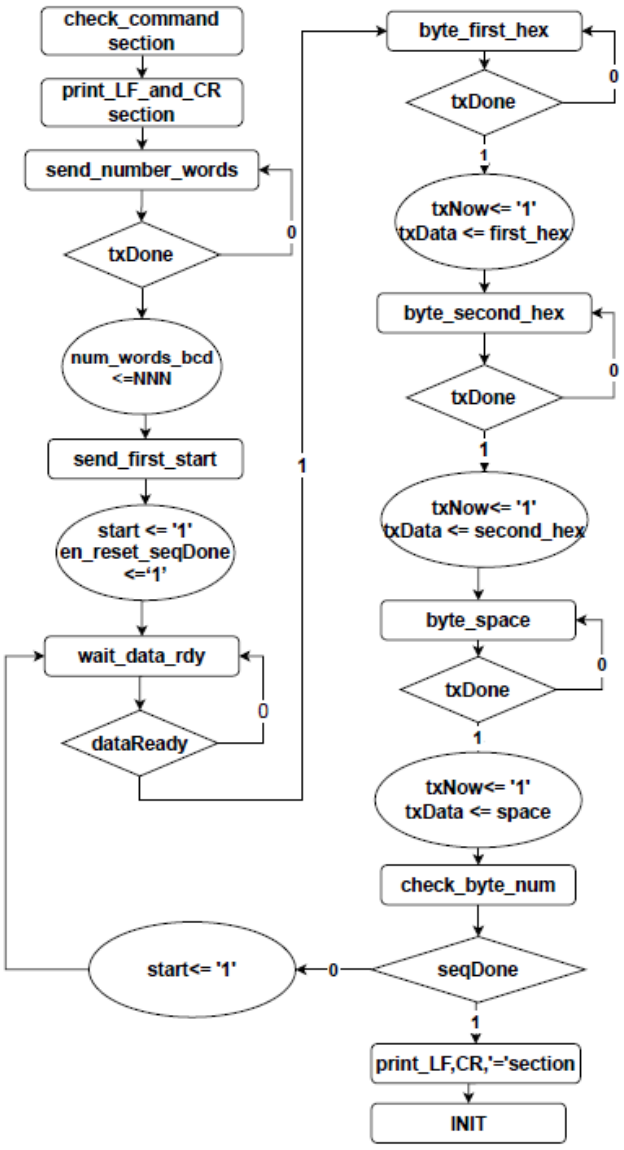

Fig. 7: Print NNN data ASM Chart

Peak Part

The Peak command(see Figure 8) receives the peak value and the maximum index from the data processor and sends them to the TX module for output. The default value will be printed if the peak value and maximum index are not received. The peak value(the third position in the data reasult) received from the data processor is in 8-bit BCD format. To output it as a 2-digit hexadecimal in the command line, we need to split the 8-bit BCD into two 4-bit BCDs and convert each into an 8-bit ASCII code for output. To output two digits in the command line, two states are implemented. In each state for outputting digit, it waits for the txDone to go high, indicating that the TX is ready to receive data. Then the state will place the data on txData and set txNow to high to send the data. After outputting all the peak values, a state is implemented to output space(”00100000”). Then the state machine will then implement a sequential output of the 3 digit peak indexes, which uses the same previous operation. And we designed a peek index counter process(see Figure 2) to update the peek index number. At the start of the List command, a counter is initialized to 3. After outputting each digit index, the en_peek_counter signal will start peek index counter process. Then the counter is decreased by 1 until all 3 digits indexes are outputted. When the 3 digits index has been outputted, the state machine will go to the print LF, CR and ’=’ section to print the separator. After that, the state will return to the INIT state to wait for the next valid command and send an en_reset_peek_counter signal to reset the peak index counter.

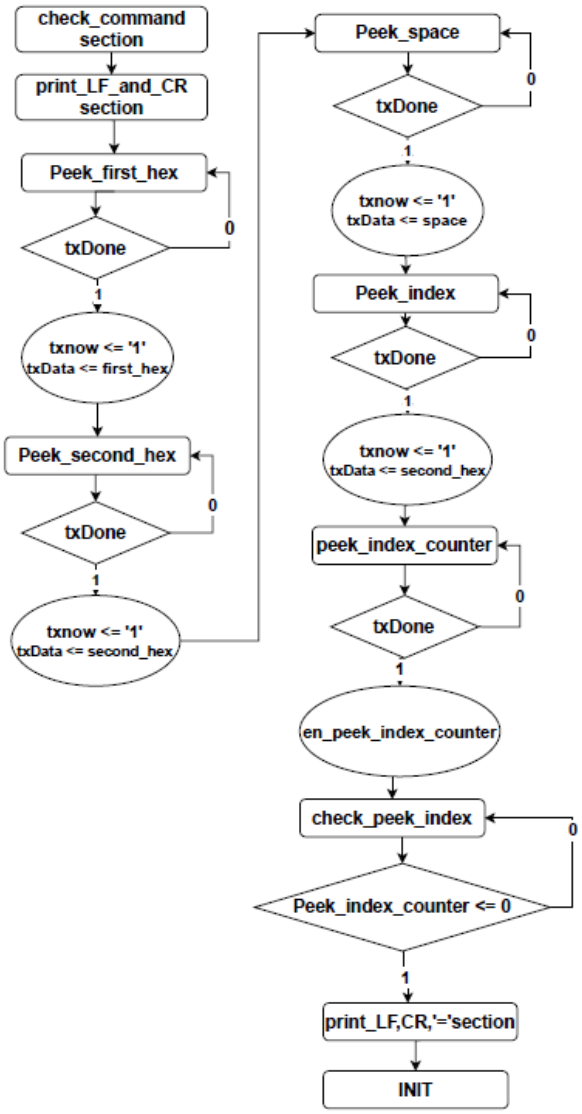

Fig. 8: Peak ASM Chart

List Part

The List command(see Figure 9) outputs 7 bytes received from the data processor. If the bytes are not received, the default bytes will be outputted. Each byte will be outputted as a 2-digit ASCII code in the command line. Different with the Peek command, the List command needs to print all data in dataResult. So, there is a counter to record the number of data that has been printed. We designed a process named ”list_counter_mins”(see Figure 2) to update the counter number. And at the start of the List command, a counter is initialized to 7. After each byte is outputted, the en_list_counter signal will send. Then, the counter is decreased by 1 until all 7 bytes are outputted. Two states are implemented for the 2-digit ASCII code for each byte, followed by a state to output space(”00100000”). And in each state for outputting digit, it waits for the txDone to go high, indicating that the TX is ready to receive data. Then the state will place the data on txData and set txNow to high to send the data. When the 7 bytes list data has been outputted, the state machine will go to the print LF, CR and ’=’ section to print the separator. After that, the state will return to the INIT state to wait for the next valid command and send an en_reset_list_counter signal to reset the list counter.

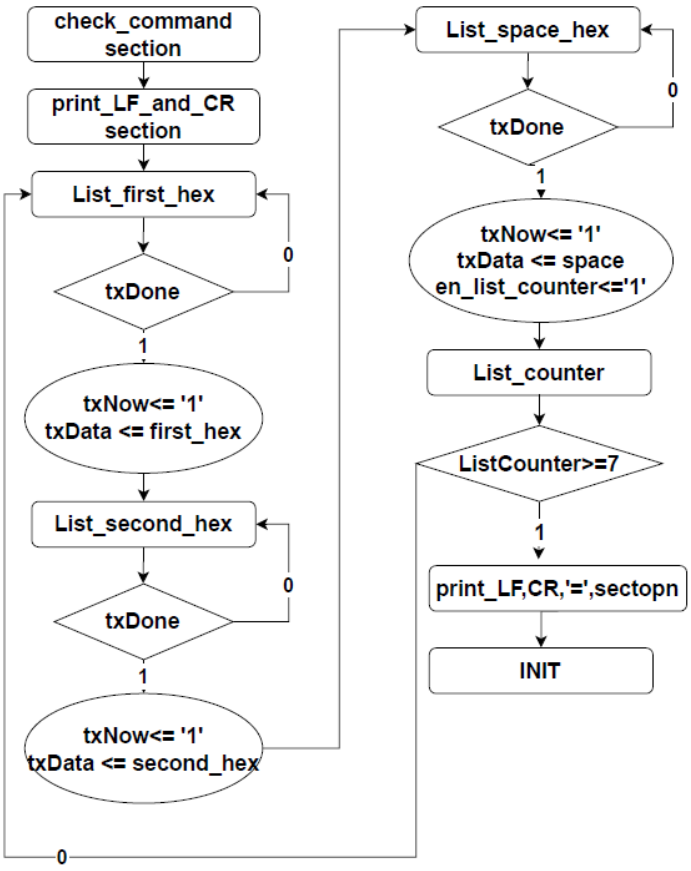

Fig. 9: List ASM Part

Command Processor TestBench Result

After designing, coding and debugging, the command processor can be complied with and doesn't report any error. And the below screenshot(see Figure 10) shows the Command processor Simulation result with its command processor test bench(given to us to test), which runs 250ms.

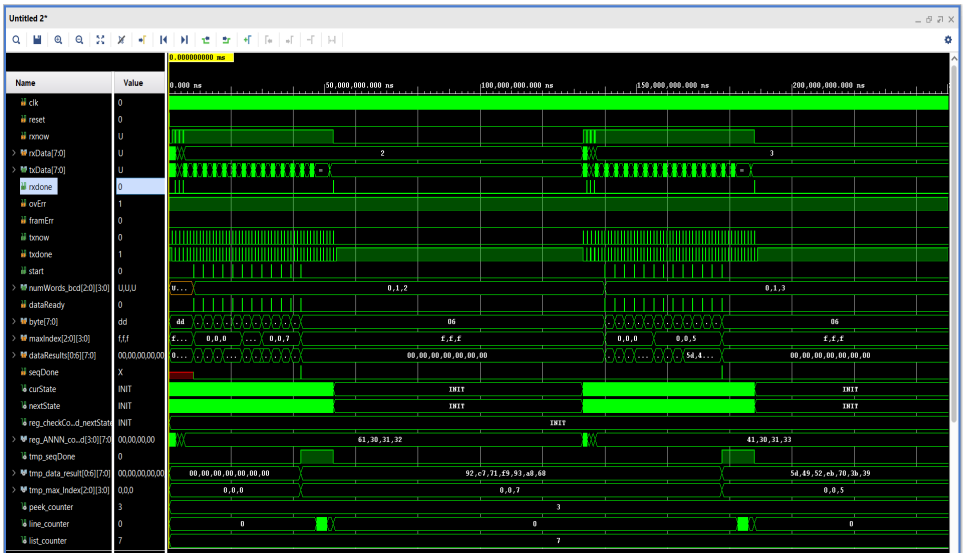

Fig. 10: Command Processor Test Bench Result

DATA PROCESSOR

Two phases

The theory of the two-phase handshaking protocol concentrates on detecting changes in the control line signal rather than its absolute value. The two processors in the source code are ctrlIn_delayed and ctrlIn_detected. Every clock cycle updates the values of ctrlIn and ctrlOut_reg (ctrlOut), which are stored by the ctrl_delayed process. The XOR operation between ctrlIn and ctrl_delayed produces the output of ctrlIn_detected. When the value of ctrlIn_detected equals 1, it means that the value of ctrlIn differs from the value of ctrl_delayed and that new data is available on the data line. The start value is checked at this moment. The value of ctrlout_reg (ctrlOut) will be set to ”NOT ctrlOut_delayed” if the start equals 1. This ensures that the new value is different rather than giving ctrlout_reg (ctrlOut) a particular value. This ensures that the new value differs from its current value (ctrlOut_delayed) rather than assigning a specific value to ctrlout_reg (ctrlOut). It should be noticed that the output port (ctrlOut) cannot be read, hence ctrlout_reg is used instead. For ctrlOut's value to be read by ctrlOut_delayed, it must first be registered (ctrlout_reg). Due to the name of the port difference between the data generator and dataconsume, it is important to use ctrl_genDrive and ctrl_consDrive signals to reduce confusion. As the port map shows, ctrl_genDrive connects with ctrlIn in the code.

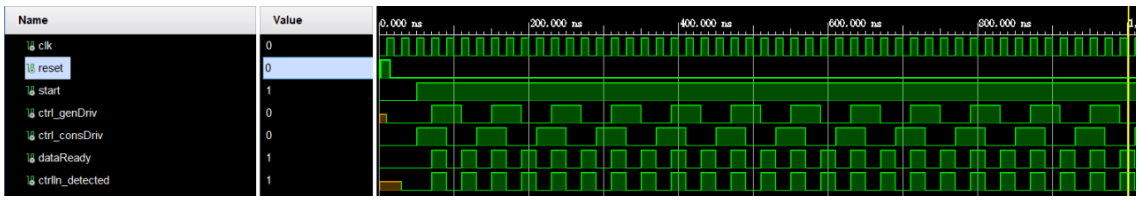

Fig. 11: 1000 ns simulation of two-phase protocol

Counter

To find the index of peak value and count the byte, a counter is used to count when a byte is sent from the data generator to data_consume. There is a register called max_reg store with the peak byte. At each count, if the new byte is greater than the peak byte, this index would be saved in maxIndex_dec. The maxIndex signal would update each rising clock when enCount is true, converting from the maxIndex_dec.

State machine

There are four states in state machine: Initial, Data_recieve, Loop, Finish. To avoid latches in case of missing conditional statements, all signals used in this process are assigned to a default value at the start. At the Initial state, if start equals 1, the Command Processor requests data from the Data Consumer, and ctrlOut_reg (ctrlOut) is set to NOT ctrlOut_Delayed, transitioning to the FIRST state. This triggers a change in the data request line between the Data Consumer and Data Generator, updating new data on the data line.

In the data_recieve state, ctrlIn_detected is monitored, and when it equals 1, new data is available on the data line. This sets dataReady high, notifying the Command Processor that the new byte has been sent to it, and transitioning to the Loop state. Additionally, the counter is enabled.

In the Loop state, count is compared to numWords_integer, and if it is greater than or equal to numWords_integer, the number of bytes sent to the Command Processor is sufficient, transitioning to the finish state. However, if count is less than numWords_integer, the number of bytes sent is not enough, and if start equals 1, the next state returns to data_recieve, and ctrlOut_reg (ctrlOut) is set to NOT ctrlOut_Delayed. This triggers the two-phase handshaking protocol to request new data from the Data Generator.

The finish state is the finishing state, only entered when all required bytes have been sent to the Command Processor. If seqDone is TRUE, the next state will be Initial. Please see the ASM chart (see Figure 12) and simulation output (see Figure 13) below.

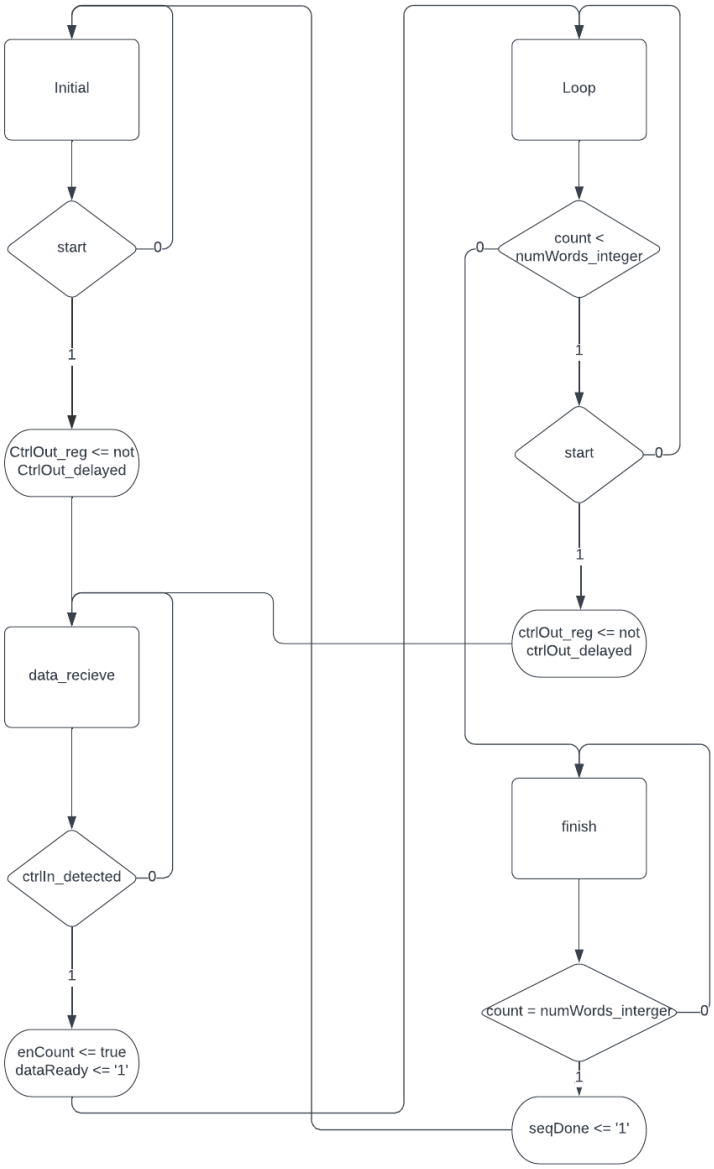

Fig. 12: The whole Data processer ASM chart

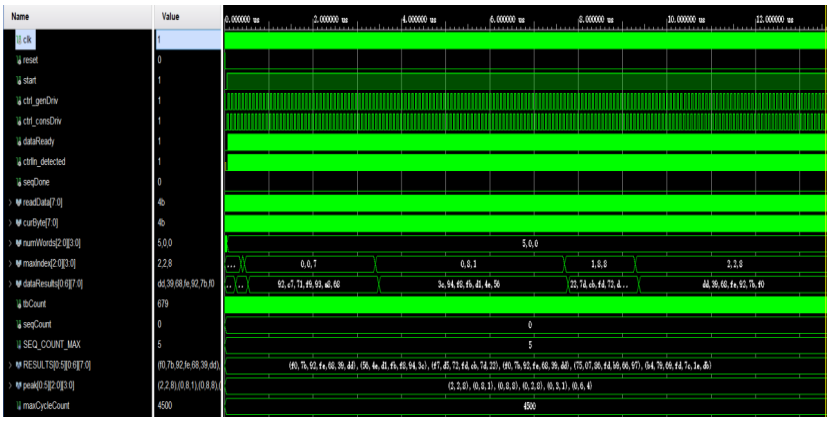

Fig. 13: behavior simulation output

Storage

When we request data from the data generator, we need a signal to store the data and compare its size to find the peak (dataResults can only be an Out signal and cannot do any processing other than assigning values). To do this, we use a 7-bit signal data_reg of type CHAR_ARRY_TYPE to store the most recent data generated. In S1, the newly generated data is stored in the last bit of data_reg, data_reg ¡= data, and the generated data is assigned to a byte and sent to the command process: byte ¡= data.

Shifting

To ensure that the data flows properly, we designed shifting. data_reg (6) ¡= data_reg(5); data_reg (5) ¡= data_reg(4); data_reg (4) ¡= data_reg(3); data_reg (3) ¡= data_reg(2); data_reg (2) ¡= data_reg(1); data_reg (1) ¡= data_reg(0); This ensures that data_reg stores the latest 7 data.

Comparison

The main function of dataprocess is to compare the data and find the maximum value. We need to store the Peakvalue in the middle position of dataResults and transmit it along with the index value of Peakvalue in the generated byte to the Commandprocess for processing. The most important thing is to find the Peakvalue. We use a simple algorithm and introduce two signals, max_reg, which is used to store the value of Peakvalue, and maxIndex_dec, which is used to store the index value of peakValue. When each new data is received, it is compared with max_reg. If data ¿ max_reg, the newly received data will be assigned to max_reg as the new Peakvalue, and the count corresponding to the value generated will be assigned to maxIndex_dec as the index value of Peakvalue. When the new value moves to the third position of data_reg, i.e. data_reg (3) = max_reg, and count = maxIndex_dec + 4, the 7 numbers stored in data_reg are Peakvalue, i.e. the data before and after the first three digits. At this point, assigning data_reg to dataResults produces the desired result.

Coding

Since the maximum value we found earlier is a decimal integer, and what we transmit should be 3 digits of BCD type data, we also need to encode maxIndex to convert it to BCD type. First, separate the 3 digits of the integer. Store the remainder of maxIndex_dec/100 (i.e. the hundreds digit of the integer) in the signal maxIndex_reg_array2, maxIndex_reg_array2 ¡= maxIndex_dec/100; store the tens digit and the ones digit in maxIndex_reg_array1 and maxIndex_reg_array0 respectively; then convert the three digits into binary numbers, such as maxIndex(1) ¡= std_logic_vector(to_unsigned(maxIndex_reg_array1, 4)); store them in maxIndex in order. This completes the encoding process, converting the decimal integer into BCD type.

Special case

It is not difficult to find that dataResults will be produced 3 data later than Peakvalue. In most cases, this is not a problem, but when Peakvalue is in the last three digits of all numbers or the instruction count is less than 3 times and Peakvalue is in the first three digits, dataResults is not updated and the program ends. The transmitted dataRsults is incorrect. This is a vulnerability that needs to be addressed. To solve this vulnerability, we first need to determine whether the two situations mentioned above occur.

We use maxIndex_dec to determine this. When maxIndex_dec ¡ 3, a Boolean signal spcase1 ¡= true; is used to indicate that special case 1 occurs when Peakvalue is in the first three digits. When maxIndex_dec ¿ 4, spcase2 ¡= true; indicates that special case 2 occurs. Due to the problem of the storage position of the data, it is difficult to uniformly handle all situations in the problem. Therefore, we used an enumeration method.

Case 1

When maxIndex_dec = 0, Peakvalue is in data_reg's 6th, 5th, and 4th positions respectively for count = 1, 2, and 3. Therefore, Peakvalue is stored in the third position, and the rest are input in order. Since this is the instruction at the beginning of the program, there is no need to assign a value of 0 to the remaining bytes. When maxIndex_dec = 1, Peakvalue is in data_reg's

6th and 5th positions respectively for count = 2 and 3. When maxIndex_dec = 2, count = 3, and Peakvalue is in data_reg's 6th position.

The result is shown in the figure(see Figure 14) below: That's the result for maxIndex_dec = 0;

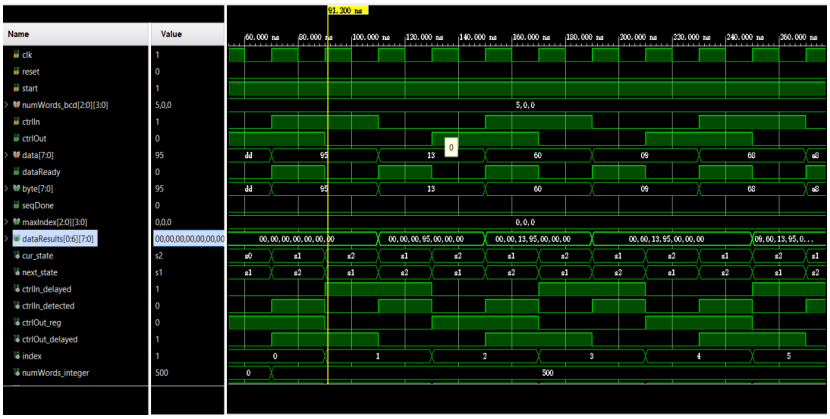

Fig. 14: case 1

Case 2

When the Peak value appears in the last 3 digits, the processing method is different. In addition to storing the corresponding Peakvalue, the extra bytes need to be set to 0. When maxIndex_dec = numwords_integer - 3 (the third-last digit), dataResults' 6th position is assigned X'00'; Peakvalue is in data_reg's 4th position. When maxIndex_dec = numwords_integer - 2 (the second-last digit), dataResults' 6th and 5th positions are assigned X'00'; Peakvalue is in data_reg's 5th position. When maxIndex_dec = numwords_integer - 1 (the last digit), Peakvalue is in data_reg's 6th position, and dataResults' 6th, 5th, and 4th positions are assigned X'00'.

The result is shown in the figure(see Figure 15) below: that's the situation that max_dec = numwords_integer – 2;

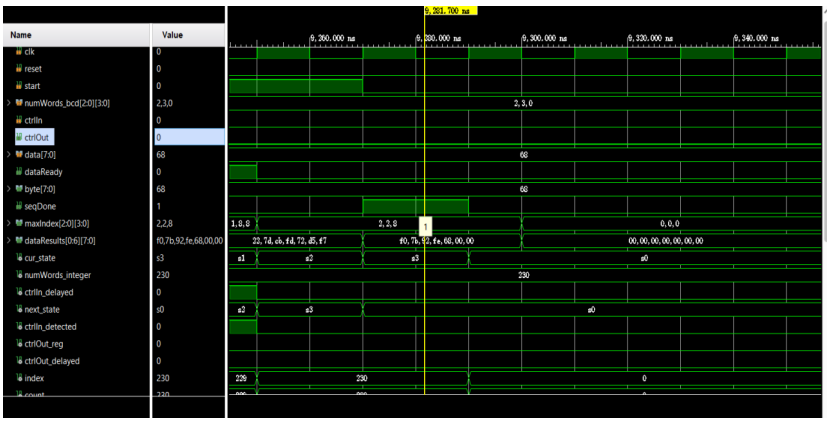

Fig. 15: The whole Data processer ASM chart

FACING PROBLEMS AND SOLUTIONS

Command Processor:

At the initial ASM chart design, we are confused about how to drive the state to reach: if A-N-N-N, then goes next state, stay or turn back to the initial state. Also P/L part is hard to add in these states efficiently; it's not good to add in every check state. Finally, with the professor's reminder and help, we considered registering and connecting with the shifter and using it as our 'A-N-N-N' register.

Initially, we used the seqDone signal to detect the state turn condition when outputting NNN data. And we found that seqDone raised a clock before the last byte of TX output was completed, so we decided to use a counter to record whether the output ANNN command was finished.

Data Processor

When converting the maxIndex from integer to BCD type for the first time, we try to split each bit initially and convert each bit to BCD type using the following code below.

```
maxIndex_reg_array2 <= maxIndex_dec /100;
maxIndex_reg_array1 <= (maxIndex_dec - (maxIndex_reg_array2 * 100))/10;
maxIndex_reg_array0 <= maxIndex_dec - (maxIndex_reg_array2 * 100) - (maxIndex_reg_array1 * 10);
```

Fig. 16: MaxIndex Before

These three lines of code are in a single processor, leading to an issue that maxIndex_reg_array would not update in the current cycle. This would cause some errors in the output of maxIndex. We then correct our code by eliminating maxIndex_reg_array when finding maxIndex_reg_array. This perfectly solves the problem which in each cycle there's no update on each maxIndex_reg_array.

```
maxIndex_reg_array2 <= maxIndex_dec /100;
maxIndex_reg_array1 <= (maxIndex_dec mod 100)/10;
maxIndex_reg_array0 <= maxIndex_dec mod 10;
```

Fig. 17: MaxIndex Update

TEST ON BOARD

Command Processor:

When first tested on the board, whenever we typed any key on the keyboard, the putty would print infinitely what we typed. To fix this, we found a problem with the txNow assignment, which was not just one clock higher, but many clocks higher, because we had initially assigned 0 to txNow in the state machine. To fix this, we gave txNow the default value, which was set to 0 after the state transition.

After solving the problem with txNow, we found that our code in putty could only print on screen what was typed on the keyboard and did not recognise the commands ANNN, P and L. Passing a test message to txData in each of the possible error states revealed that the register we had designed did not work. To solve this problem, we replaced the judgment condition rxNow of the process handling this register with another signal, 'en_shift'(see Figure 2).

Data Processor:

During testing on FPGA, we discovered that after all bytes had been outputted correctly, the Peak value, maxIndex, and dataResults results were all "00" when using the 'p' and 'l' commands. This indicated that the data processing function was not functioning correctly.

After communicating with our team members, we found that the processing logic for maxIndex and dataResults in the command processor function was to receive and process the last group of dataResults and maxIndex after receiving the seqDone signal. Upon inspection, we discovered that when the number of instructions specified by numWords was reached in our data processing logic, the seqDone signal was assigned a value of '1', and all array signals were cleared. This meant that the two commands were completed at the same time. The simulation showed that dataResults and maxIndex had already been cleared on the rising edge of the next clock after seqDone was assigned a value of 1, i.e., the array signals were cleared after seqDone and before the command process received them.

To solve this issue, we processed the array and seqDone signals asynchronously. We set the "clearCount" signal to zero in the initial state to clear it. Compared to the seqDone signal's high-level signal, which cleared one clock later, we obtained the correct results. On the FPGA, dataResults and maxIndex were outputted correctly.

CONCLUSION

In conclusion, we successfully designed and implemented both the command and the data processor. And we also successfully combined these two processors to test on FPGA cpg236-1 board. Figure 18 shows the simulation result of the Command processor with the data processor, which runs 250ms. And figure 19 displays that the system runs on the FPGA board, types the valid command on Putty, and prints the corresponding data. But we still found that one special situation didn't solve after we debugged our code on board. Our code couldn't find the peek value when the peak value appeared in the last position, which had the same behaviour in the demo. So, we guess that the FPGA has a clocking problem that causes transmission problems when the peek value appears in the last bit for comparison. And during the project, we found that we spent much time testing on the board because Putty will not report any error and uses our logic to print information. And we also don't have much experience with debugging on board, as we had only simulated testing the validity of the code through the test bench before. So, we hope that we can get more support on the approach of the testing board, which can help us save much time. For example, when the system stalls at a certain stage, the error on Vivado hardly reflects the real problem if it is due to a signal reception problem. We want an error message similar to the one in the simulation, such as when the process is stalled because the system is not receiving the expected signal. The signal displayed on the putty will pop up as "error, The expected signal seqDone not received". This helps us locate the code problem and make changes quickly. During the doing the whole project, our teamwork also faced some problems. Someone couldn't attend the lab and didn't do some effective work initially. But thankfully, the team members became more efficient after a few reminders in face-to-face meetings. We also completed the peer assignment form(see Figure 20) based on the performance of each individual in this project. And next time and in the next project, we will improve these problems, including the code and teamwork.

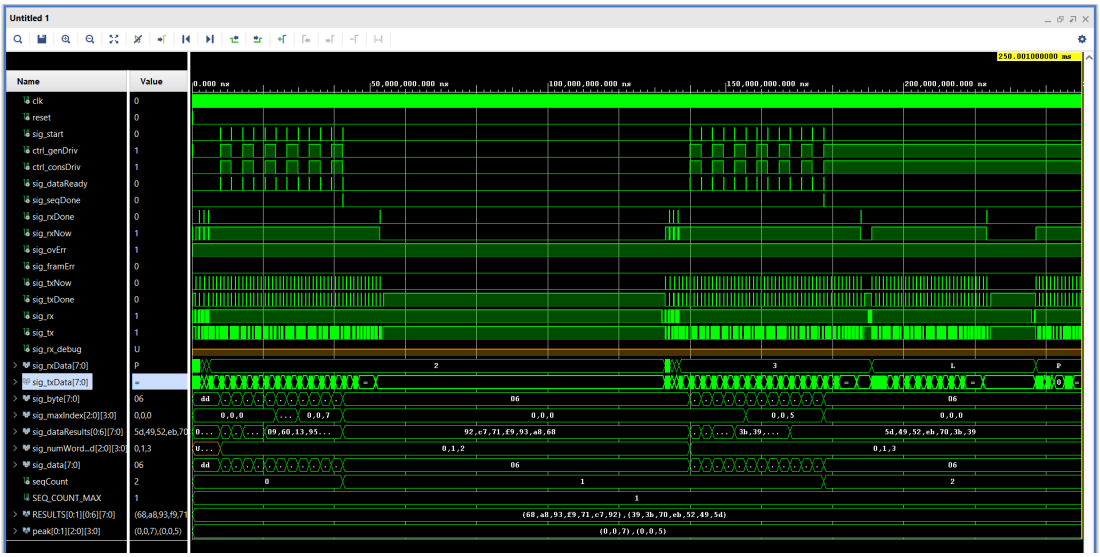

Fig. 18: Whole System in Simulation

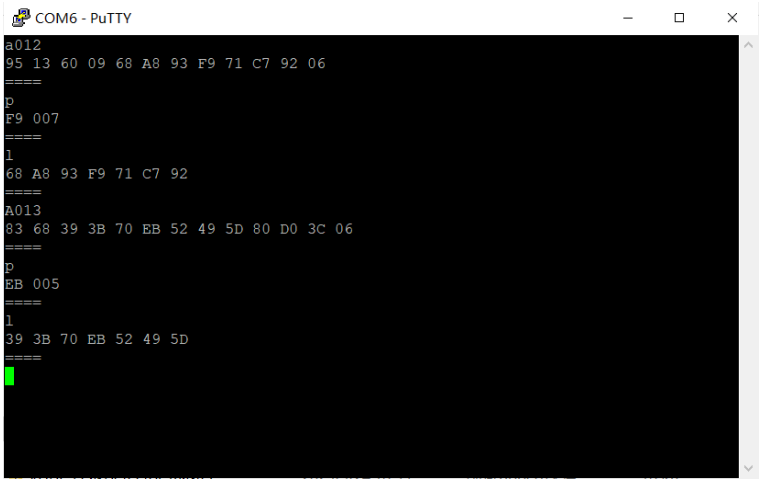

Fig. 19: Test on Board(command a012 A013)