Unsigned peak detector

Yanbo Chen : xt20786@brsitol.ac.uk

Fan Yang : kz20614@brsitol.ac.uk

Jingrong Yang : hr20631@brsitol.ac.uk

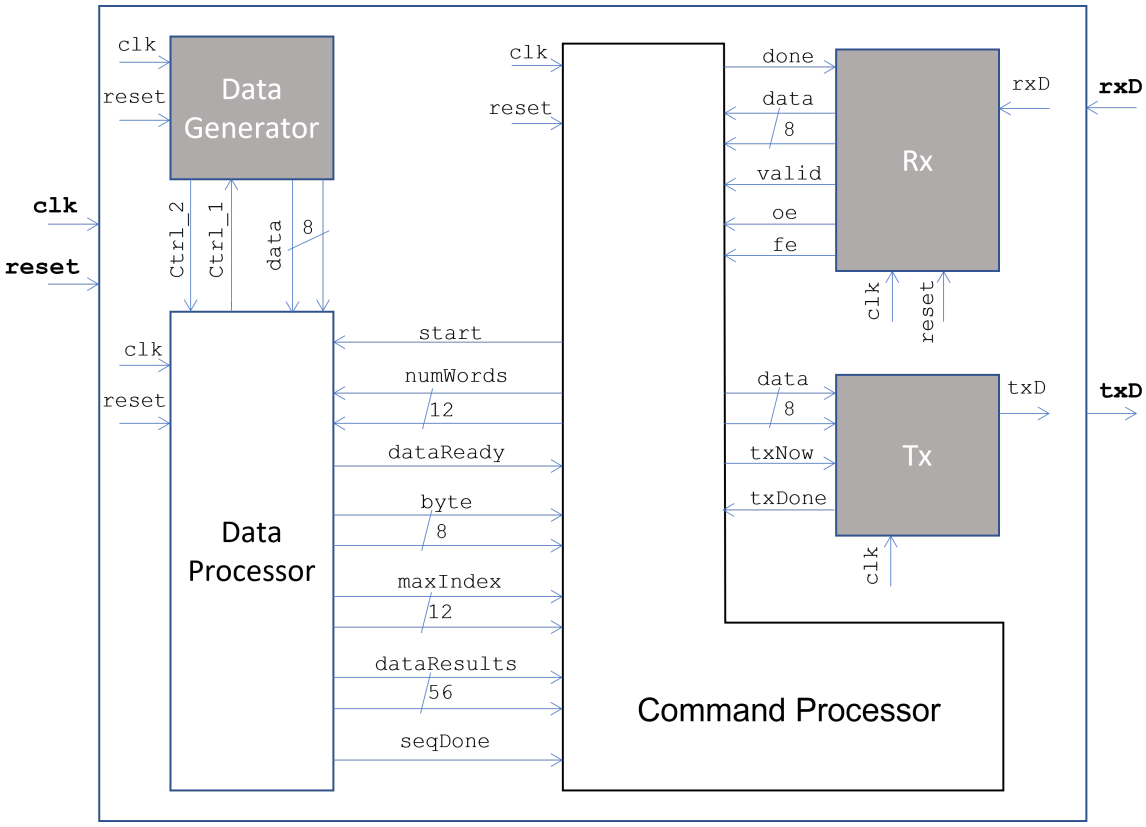
Xinyi Sun : lm20313@brsitol.ac.uk

Kaichen Bai : yf20844@brsitol.ac.uk

Introduction

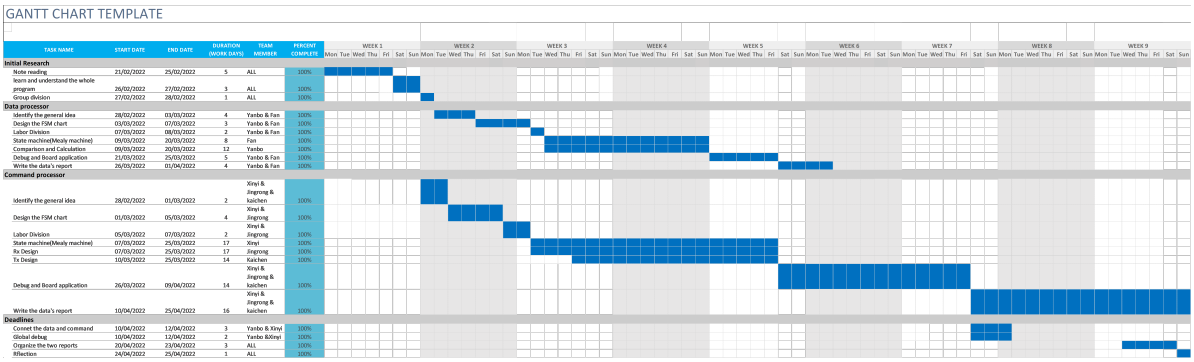
This report is about how to realize the function of a peak detector using VHDL and the principles of the implementation. Since our group is group 3, so we use the unsigned data.

The picture below[1] shown the basic structure of the project. The user enters ANNN to the computer to determine the number of data to be processed, and the RX receives it and sends it to the Command processor for transcoding (ASCII to BCD) and transfers it to the Data processor. Then the data processor will detect the highest value the data according to the number required, and packages the result with the previous and the next three numbers to the Command processor, and also the index of the peak value will be sent to command processor. The command processor then gives a different output depending on the L or P command issued by the user, which is sent by the TX.



The division of work in our group is as follows:

- Data processor: Fan Yang, Yanbo Chen
- Command processor: Xinyi Sun, Jingrong Yang, and Kaichen Bai



As shown in the Gantt Chart, our work flow is as follows:

1. Reading the document together to get a general idea.
2. After obtaining a basic concept about exactly what we needed to achieve we started grouping.
3. Each group completes their own part separately.
4. Combining the completed two groups into a whole and debugging it.

Contribution

Fan Yang

Fan Yang's contribution is as follows.

- Read the whole document, get a general idea and do the group division with my other group members
- Design the initial and internal operating logic of data processor with Yanbo Chen.
- Design different states and the usage of theirs
- Choose the machine type which is mealy machine
- Write the State machine part i.e. combi_nextState, combi_curState and delay_CtrlIn in the DataConsume file of part A
- Debug the overall data processor with Yanbo Chen.

Yanbo Chen

Yanbo Chen's main responsibilities included as the following:

- Thinking with the group about the implementation of the peak detector
- Completing the comparison of data values in the Data processor, and the conversion of input and output formats.
- Debugging the whole data processor with Fan Yang.
- Integration of data processor and instruction processor with group members
- Debugging of the final completed peek dector

Xinyi Sun

Xinyi Sun's main responsibilities included the following:

- To construct a command control module, which controls the overall peak detector system, as receiving ANNN command, receiving L and P command. Command**s details would be illustrated later.
- To establish data interaction with the data processor.

- To build registers to store essential data transported from the data processor module in suitable formats needed in UART transmitter (TX) port.
- To plot whole ASM draft of command processor.

Jingrong Yang

Jingrong Yang's main responsibilities included as the following:

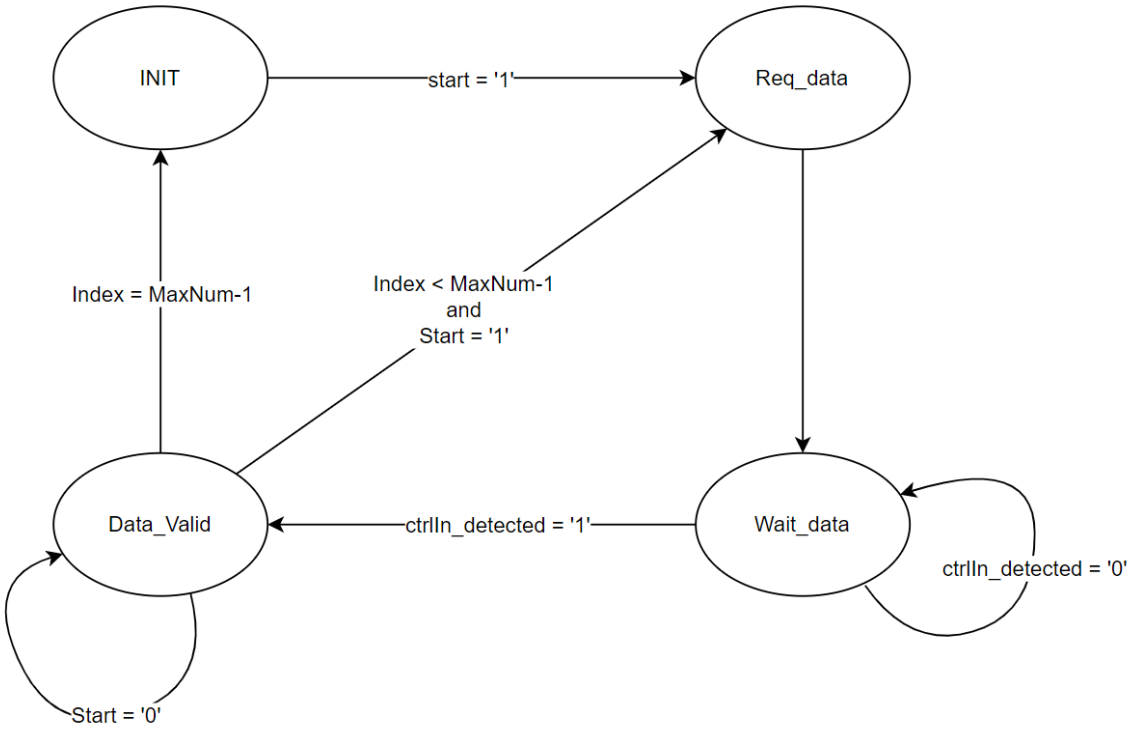
- To build the signal transfer mechanics between the UART receiver (RX) module and the command processor model.
- To build registers to store essential RX input data in suitable formats.
- To detect frame error (*fe*) and overrun error (*oe*) from UART RX.
- To plot the final version of FSM chart of command processor based on Xinyi Sun**s ASM draft.

Kaichen Bai

Kaichen Bai's main responsibilities included as the following:

- To construct different data interaction between command processor and UART transmitter (TX) in ANNN command, L command, and P command. Command**s details would be illustrated later.
- kaichen mainly focused on interaction of UART transmitter module data in the command processor.
- Depends on L command and P command, print correct result.
- This part will receive the data from the data processor, UART receiver (RX) module and UART transmitter (TX) module. And then print it out no matter if this signal is correct or wrong, this part needs to print this signal out and finish this cooperation.

Data Processing module



For the data processor, we need to recognize the **numword** and output an array with the highest byte and three bytes in front and behind it as a total of 7 bytes, and its index in the form of BCD code.

To briefly summarize the process, we first need to get a **start** signal of 1 high from the **command processor**, then we need to read a 3-byte BCD array name as **numword**, convert it to an integer and initialize the counter. Then when the counter value is less than the amount of **data** we will eventually need to process, we need to request new **data** from the data generator, i.e., to invert the original output. Then we wait for the input of the data generator to be inverted, or when the input of the **data generator** and its delay a **clk** cycle with an XOR result of 1, which can be interpreted as a ready signal, telling us that the data is ready, then we can read the data and compare it with the highest value we have stored before. When the new **data** is by far the highest, we need to replace the first four bytes in the **result register**, and then deposit the next three data into the **result register** in turn. There are two registers used, one is a scrolling record of the current and previous three bytes, which can store a total of 4 data bytes, and then the **result register** which can store a total of 7 data bytes, note that the registers here are stored in **char_array_type**, char array-type is an array type, which stores 8-byte binary vector, i.e., 2-byte hexadecimal data. Also, when we reach the peak, we need to save the current counter value which is the index and then convert it to BCD output as maxIndex.

We design our state machine in the same way as a mealy machine, with a total of 4 states, **INIT**, **REQ_DATA**, **WAIT_DATA** and **DATA_VALID**.

At the **INIT** state, if start is asserted, it begins to request data, i.e., it moves to the **REQ_DATA** state. Then after requesting data, it moves directly to the next state: **WAIT_DATA**. and when it receives a ready signal, it moves to the **data_valid** state, waiting for a new start to be asserted. In this process, the computation part is done in the process. When the number of processed data is equal to the amount of data to be processed, it enters the state of **INIT** and waits for a new start and numword, otherwise, it moves on to **REQ_DATA** and continues to request data.

Functions and principles of processes used are shown below:

Current Register:

The register is used to hold the current and previous three bytes of data is a total of four bytes.

When the rising edge of a **clk** cycle and **ctrlIn_detected** are asserted, we need to shift the last three bytes of the total 4 bytes of data already stored forward and leave the last byte empty to store the new data, so that we can also transfer the 4 bytes to the result register directly when the peak is detected.

Comparator

This is the process that compares the data with the value of the detected peak.

When the reset is asserted, the peak index is set to 0 and the current **data** is stored in the third byte of the **result register** (where the peak is stored), which is used to initialize the process.

Since there is also **data** in the sensitive list, whenever a data is inputted , it is compared with the third byte of the **result register**. If the data is greater than the previous peak, the **current register** is saved and the first four bytes of the **result register** are replaced with the **current register**, and the next three bytes are saved in the **result register** process.

Result register

When the difference between **Peak index** and the current **index** is 0, 1, 2 or 3, the following operations are performed respectively

0: Bytes 4, 5 and 6 are cleared to **'00000000'**

1: The fourth byte is stored with current **data**

2: The fifth byte is stored with current **data**

3: The sixth byte is stored with current **data**

maxIndex register

This process is used to convert from integer to a binary BCD_arrat_type.

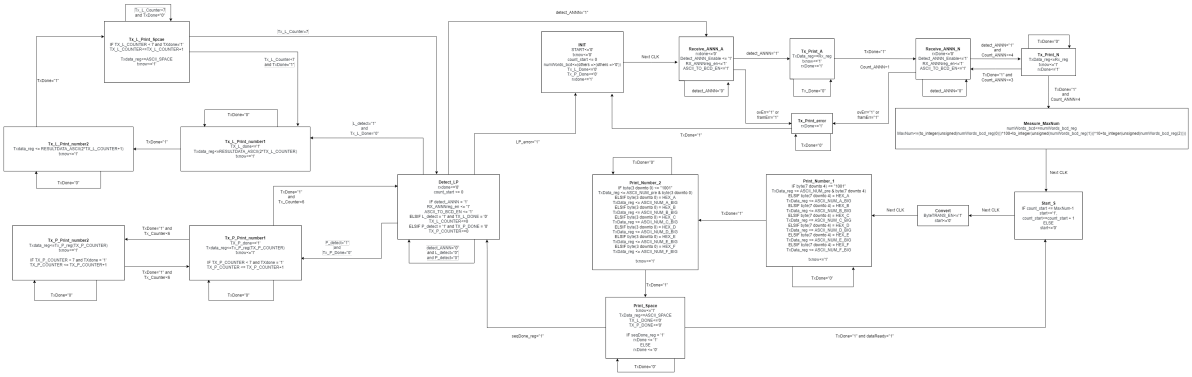
For the hundreds digit, we simply divide the number by 100 to obtain the remainder of 10 and convert the result to bytewise binary in the form of BCD code 8421

For the tens digit, divide by 10 to get the remainder of 10.

For unit digits, the remainder is taken as 10.

Command Processing module

FSM state diagram



The main function of command processor is to receive the signals input from each port, process different kinds of signals and operate signal transmission internally to control the entire peak detector system.

To understand how a command processor works, it is necessary to draw an FSM diagram, and set up different states and registers to process the signals from Rx, Tx and data processor. We used Mealy machine to represent its logic.

Figure below [1] illustrates a description of each command that the command processor needed to operate. Moreover, the mechanism of each command will be explained later.

Command	Description
ANNN or aNNN where NNN are 3 decimal digits.	Start command to process NNN bytes of data. Print each byte processed in the terminal window in hexadecimal format, separated by spaces. For example bytes "11111111" and "10101001" in sequence are to be printed as "FF A9".
L or l	Print the 3 bytes preceding the peak in the order they were received, the peak byte itself and the 3 bytes following the peak in the order received. They should be printed in hexadecimal format and separated by spaces. For example, the output of typing this command after processing a data sequence may be "09 FA A0 FD BC 10 DE".
P or p	Print the peak byte itself, followed by a space and the peak index in decimal format. For example, the output of typing this command after processing a data sequence may be "FD 197".

Rx is one of the ports connected with command processor, it transmits a string of 8-byte data, which might be **ANNN**, **L** or **P**. The command processor needs to determine what kind of signal it inputs and then send the corresponding command to the central part of command processor to make it proceed with further operations. When it receives the signal **valid=1**, which means that the signal transmission of Rx is completed, command processor will send the signal **done** to **Rx**, telling it to start preparing for the next round of signal input. However, if it receives the signals **oe** and **fe** about the error of signal transmission, which shows that there is a problem with the signal from **Rx** during the transmission process and cannot be used for subsequent operations. At this time, command processor will report the problem to command processor and terminate the operation, aiming to return the state to the initial status (although some signals still need to be sent to Tx for print, such as ANNN). In addition, there are two signals, **clk** and **reset**, which might occur in any state. No matter under what circumstances, the signal **reset=1** is received, all programs are terminated, the value of next state changes to initial state and all signals return to their initial values immediately.

Firstly, some registers for internal signals storage need to be initialized in the initial state value. The **START** signal is sent to the data processor. When it receives this signal, data processor will send each signal in data generator to command processor in sequence through the **byte** signal. In order to enable Tx to print these signals, **txnow** signal needs to be transmitted to Tx to trigger it. **numWords_bcd** signal is used for storage of signals in BCD format. It can be converted to ASCII, decimal or hexadecimal format in future operations. **Tx_L_DONE** and **Tx_P_DONE** signals are about the detection of **L** and **P** signals.When the detection results match the assigned value, they will become high for reminding system that the input of signals **L** and **P** is completed.

After a Clock, it will automatically enter the **RECEIVE_ANNN_A** state from initial state. This state is used to detect **A** in the ANNN command from Rx port. The **rxdone** signal is initialized here and it will go high only when the signal transmission from Rx is successfully received. Three enable signals(**Detect_ANNN_Enable**, **RX_ANNNreg_EN**, **ASCII_TO_BCD_EN**) will be triggered in this state. Their role is to enable the process to detect ANNN, store **rxdata** signal and

numwords_BCD signal into a register, respectively. If **detect_ANNN** signal is equal to 1, which means that it successfully received the **A** in ANNN command, it will pass to next state **Tx_Print_A**. Otherwise, it will stay in this state. The detection method is setting the constants **A** and **a** in ASCII format in advance and comparing them with **rxdata** signal. If the value of **rxdata** is equal to any, the detection is considered successful. However, there are two kinds of errors in signal transmission: overrun error and the error where start and stop bytes are not detected in sequence. If one of the above errors is detected, it will enter the **Tx_Print_error** state, which means ANNN command detection fails.

Although the transmission of **ANNN** fails, it is still necessary to print the wrong instruction in **Tx**. The **Tx_Print_error** state aims to print the wrong command and then go back to the initial state. It sends **rxdone** to Rx to tell it all data has been read and clear register to wait for the subsequent signal transmission. After printing **A**, command processor continues to receive **N** in ANNN. The rest of processing is the same as **A**. Because the number of **N** is three, counter **Count_ANNN** needs to be used for counting. Each time a letter from ANNN is received, the counter is increased by 1. If the counter does not exceed four, the state will jump between **Reveive_ANNN_N** and **Tx_Print_N**. When the counter counts to 4, proving that all four letters of **ANNN** have been input and printed, the system will transfer to **Measure_MaxNum** state. The role of this state is to convert the **P** result (the index of the signal with maximum value) in BCD format to decimal format.

The next operation is to print all the signals in data generator. Enter the state **Start_S** after a clock. It aims to send a **start** signal to data processer to tell it to send all signals from data generator. Counter **count_start** counts how many starts are input into the data processor to prevent the output number from exceeding the maximum number of signals in the data generator. It resets to zero after completing one turn. The **Convert** state that command processor enters after a clock is used to convert the signal in the data generator from hexadecimal BCD format to ASCII format. From the next clock, it will change to **Print_Number** state to print all signal values. It is divided into two states, **Print_Number_1** and **Print_Number_2**, which print the first four bytes and the last four bytes of the 8-byte signal. The **byte** signal from the data processor is compared with the hexadecimal constants set in advance. If it matches the corresponding number or letter, the constant set in ASCII format is input into the signal, which will be sent to Tx. Every time the conversion of a 4-byte signal is completed, a **TxDone=1** signal will be sent in order to tell Tx to be ready to receive new data. When the printing of bytes from 0 to 3 is completed, the system will enter the **Print_Space** state and input space in ASCII format to **TxData** signal. After printing the space, the command processor will determine whether the **seqDone** signal from the data generator is 1. If **seqDone** =0, it shows that signals have not been fully transmitted, and it needs to return to the **Start_S** state for a new round of operations. If **seqDone** =1, it will go to the next step: detecting **L** and **P** signals.

After the commander processor received the *ANNN* command and execution was completed well, the signal **maxIndex** and **dataResults** would be stored in the command processor module registers, waiting for uses UART TX. Meanwhile, the command processor would be in a state to detect the following command provided by the UART RX port. If the command provided is ***P*** or ***L***, the command processor would turn to the next state to operate and transfer the necessary bytes as a txData signal.

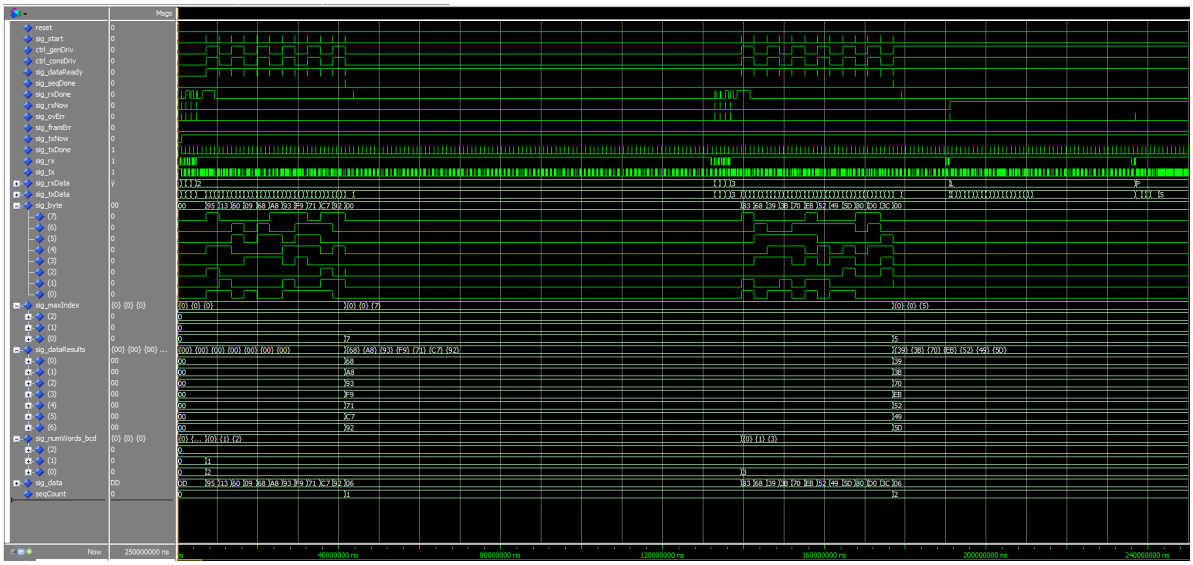
The main purpose of **L** (List) command was to print a list from **dataResults** signal composed of 7 sets of 2-bytes hexadecimal numbers by UART **TX** module, which included 3 sets of hexadecimal numbers preceding the peak hexadecimal number, the peak hexadecimal number, and the three sets following the peak. Moreover, every set of numbers was composed of one byte of binary codes, and every 4 bytes presented one hexadecimal figure. However, the UART TX module could only acquire one byte of ASCII code presenting one figure. We transferred **dataResults** signal (7 x 8 byte) into 14 bytes of ASCII code, and each byte presented one figure. Besides, for users to have a convenient reading experience, we inserted spaces between numbers. Every two figures were printed out in the UART **TX** module; the command processor would transmit one byte of space ASCII codes into the TX module to prevent numbers from sticking together until the final numbers were printed out.

The main aim of the **P**(peak) command was to print the peak value with the following index of it. The middle from the **data results** signal captured the peak value, which is also the fourth byte of the 2-bytes hexadecimal numbers in it. It would be transferred into 2 bytes of ASCII codes as previously mentioned in the explanation of the *L* command. Furthermore, the index of peak value was generated from the **maxIndex** signal composed of 3 sets of 4-character binary-coded decimal (BCD). Every set of the BCD data presents one figure from index numbers, and it also needs to be transferred into one byte of ASCII data format. Besides, we also insert space between the value and the index to ensure information is presented clearly.

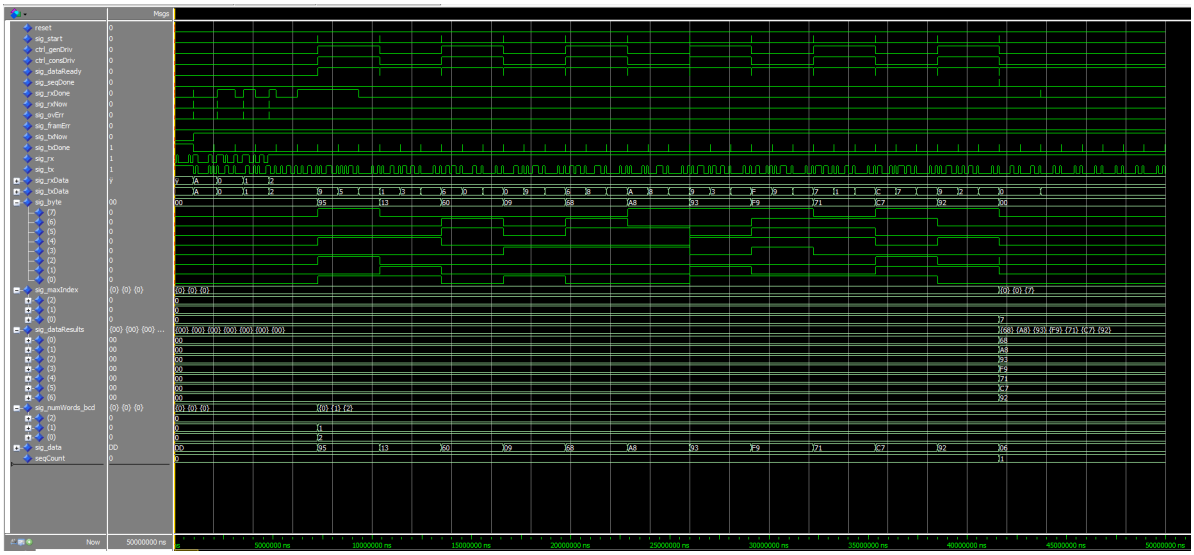
After one command of *L or P* was executed well, the command processor would turn to a state detecting another *L or P* command. Suppose there was no correct command from the UART RX module accepted by the command processor, depending on the signal we received. In that case, the state turns to the INIT state to initialize all settings or the ANNN detection state to proceed subsequent peak detection.

Results Analysis

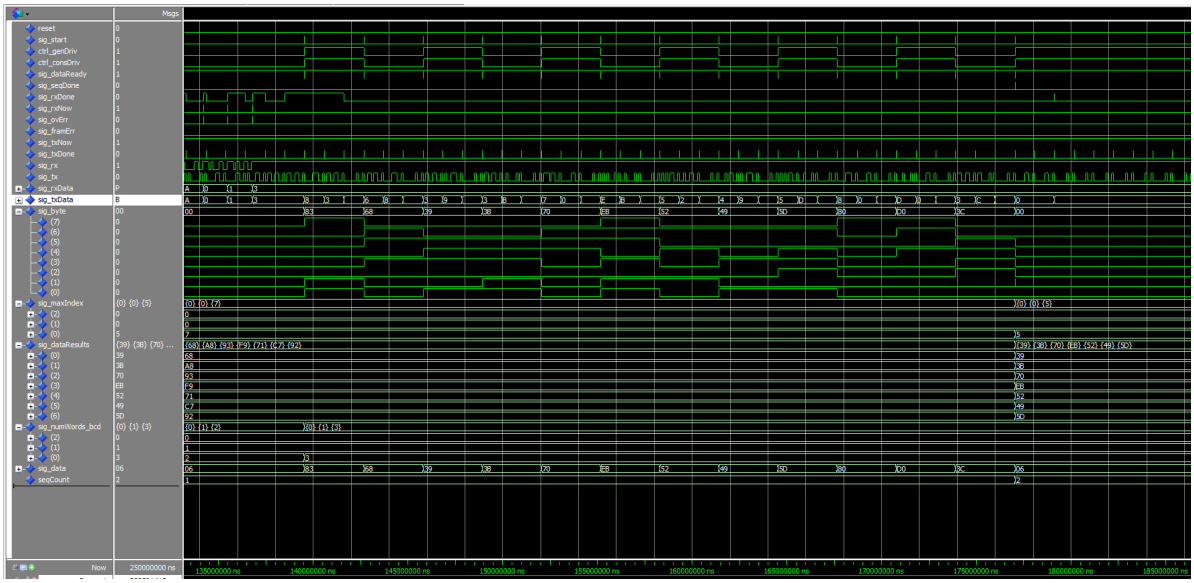
The following screenshot is the whole result of our project, include data comsume and command processer, the total running duration is 260 Ms. The duration can be seen as two parts: the first part is from 0 ns to about 130 Ms and the second part is from 260 Ms. The first part is the process that command processer received the first ANNN value and didn't receive the final L or P, and move directly to the INIT state, while the second part is the process that the command processer received the second ANNN value and then receive the L and P.



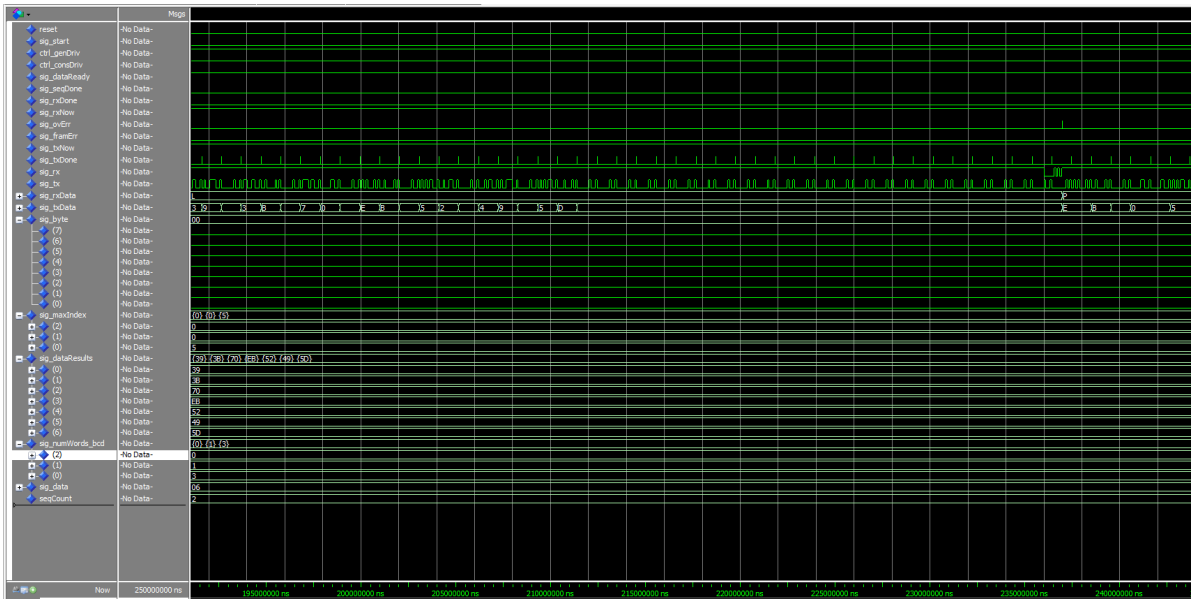
For the first 50ms section, RX accepts a full ANNN at 5,000,000ns, which is A012, and transmits it to **numWord** around 5500000ns, it gets 0 1 2. When the data processor receives a high start signal, it processes a data, transfers the processed data to the byte and sets the data ready to high, the command processor receives the new byte and gives it a **clk** high start signal again. Around 42,000,000ns, all data has been processed, and the **dataResult** outputted the final result, and the **maxIndex** signal also stored the final result.



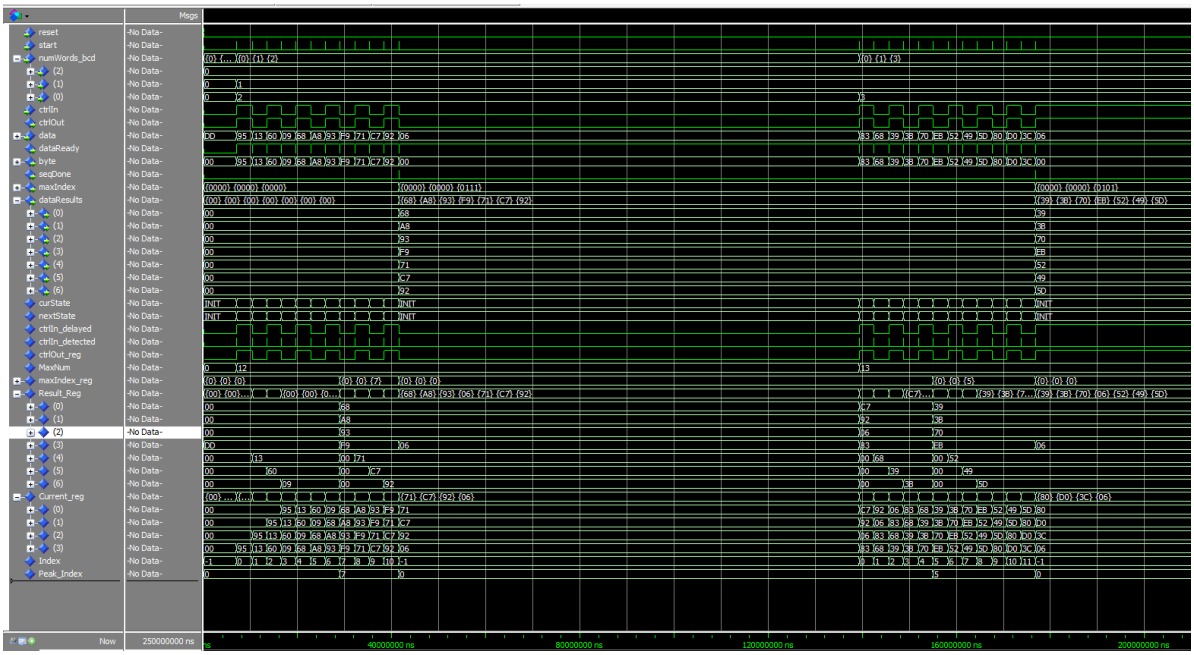
The second 50ms, basically the same as the first 50ms, differs only in the **numword** received.



Finally there is the part about the detection of **L** and **P**. We can see that the first half of the picture is the **L** part started from around 187,000,000ns and the second half is the **P** part started from around 237,000,000ns. For the **L** part, tx outputs the previous and next three bytes of the peck in order, separated by spaces. And for the **P** part, **tx** outputted the peak byte and its index , separated by spaces too.



The last picture shows the view of the data processor in the first 50ms and second 50ms. We can see the process of the data processor to process the data, and how to save the result in the register and output them to the command processor.



Conclusion

In our group's lab, we used the Mealy machine as our main idea, which meant we did not need to set up more states and facilitated our code writing. Finally, the system presents a peak detector communicating with a data source that delivers unsigned bytes on request, identifying and storing the byte with the highest value and its adjacent bytes. Our group had done the FSM or ASM drawing before coding, which meant we could follow the ASM in programming the code. However, this also caused some problems, especially when we found issues in the state diagram during debugging that caused some signals to be early or delayed, which had to be fixed after modifying the state diagram, mainly because of the initial drawing of the state diagram was not thorough enough. We also tried to use more processes instead of the functions in the state program to prevent the state diagram from becoming too unwieldy.

Besides, there are also other limitations. Firstly, the signals from the test bench provided by the University of Bristol did not include the **oe** (overrun error) and **fe** (frame error) signals, which means our simulation could not consider all the situations we may face. Furthermore, we only think about making sure the commands work correctly and give the user the correct feedback, but we did not put the results into practice using UART hardware. But even so, we have already completed the core of the simulation and interaction instructions, and we can continue to improve the code based on this later.

Reference

[1] Digital Design Group Project Contents (bristol.ac.uk)](https://seis.bristol.ac.uk/~sy13201/digital_design/ECAD/A2_index.htm)