Digital Design Report
EENG 28010

Group members: Casper Wang, Mingzhang Deng, Otis Lee, Vincent Wong, Yichen Dong

Table of Contents

Introduction

This project aims to implement a system capable of detecting peaks and communicating with a mother computer. The system identifies the peak(s) from the data line and provides 6 adjacent bytes upon request (3 index above and below the peak). The system is hosted on a FPGA via a Universal Asynchronous Receiver and Transmitter (UART), where the sequence and result list will be communicated.

This task is split into 2 main parts; the Data Processor which consists of a peak detector module and a two-phase handshake protocol module, and the Command Processor which consists of Three command handling modules and 5 peripheral input/output handling modules.

Task Division

Team A (Data Processor):

- Casper Wang: Implemented peak detection algorithm for signed binary that uses minimal memory, and has the ability to find both the peak number and its surrounding as well as the index of the peak in bcd format.

- Otis Lee: Created two-phase handshake protocol architecture and main counter. It was used to communicate with the data generator and count the amount of data received using a 4-state state machine.

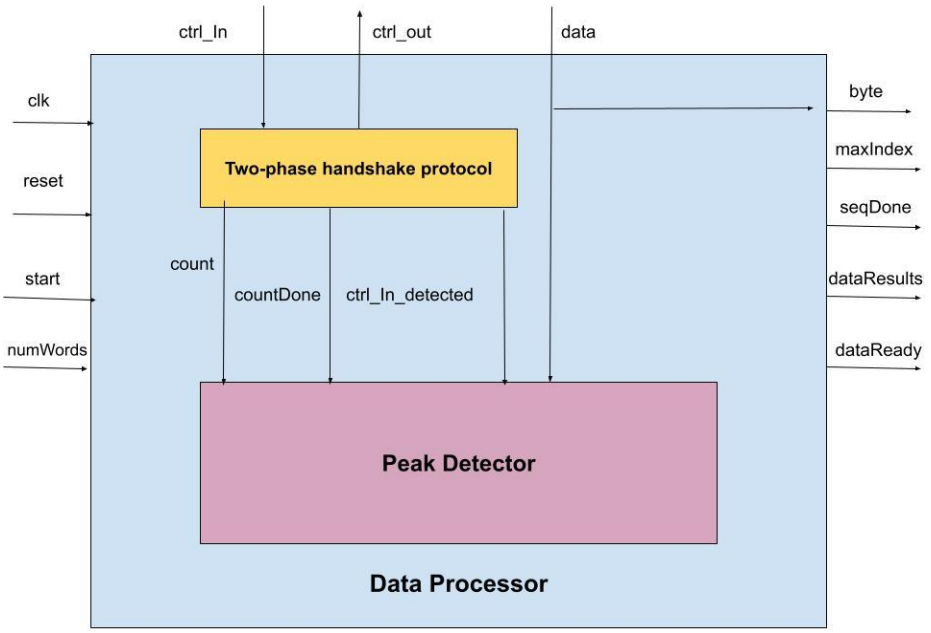
Team B (Command Processor):

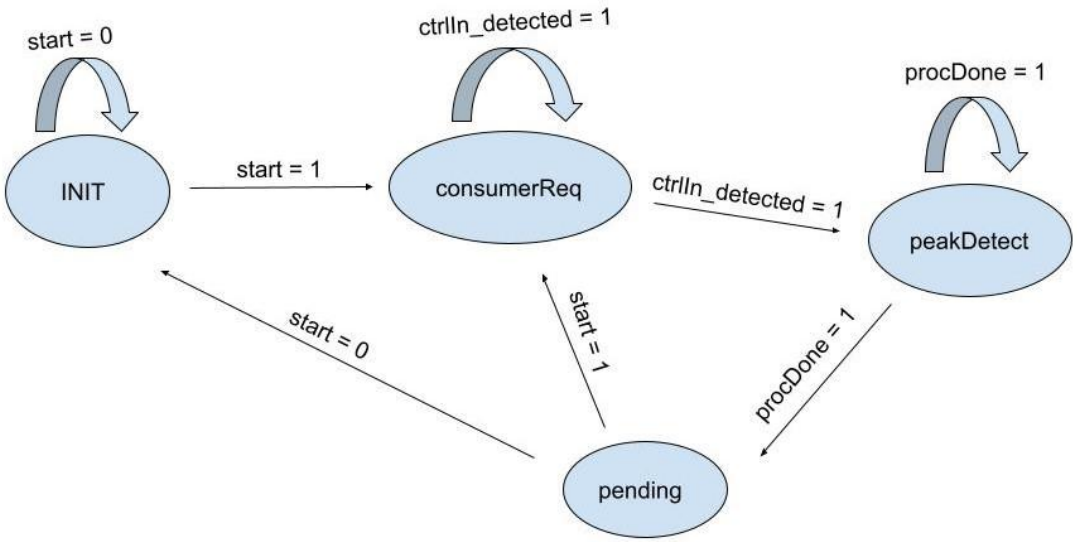
- Mingzhang Deng: Complete the Basic skeleton of CMD Processor, Design the CMD_Proc workflow.

- Vincent Wong: Designed L and P Process and testbenches for the processes.

- Yichen Dong: Completed the implementation of data communication between data processor and command processor. Which fetch and pass data from each parts by a 7-state machine.

Design and simulation of Data Processing module





Our Data Processing module consists of 4 states in total, that's the result of many optimizations we made. We started with Moore Machine, then gradually changed it into Mealy Machine.

In the consumerReq state, all the handshaking protocol and counting are done where some of the internal signal will be outputted for later use. In the peakDetect state, shifting the bits

and operations on the data will be done there. Index and current peak will also be recorded here. Then in the pending state, depending on the start signal, the program will either go into a loop that back to the step one of processing the new data or simply output the max index as well as the dataresults to the command processor. In this case, it indicates that the entire cycle is done and the program will be back to the INIT state waiting for the next cycle. This way, no redundant state is implemented. And we can still achieve the desirable result that we want.

Two-phase handshake protocol - by Otis Lee

I'm in charge of the main counter and the two-phase handshake protocol. The protocol consists of a transmission phase and a receiving phase, and both the data generator and the data processor have corresponding processes to facilitate this protocol.

```
-- ctrlIn delayed by 1 clock cycle (for transition detecting purpose)
signal ctrlIn_delayed: std_logic;

-- ctrl transition detected (used to signal dataReady)
signal ctrlIn_detected: std_logic;

-- Register ctrl Out
signal ctrlOut_reg: std_logic:= '0';
```

The former phase is responsible for creating a transition of the ***ctrlOut*** signal, which is received by the data generator. This is achieved by creating a ***ctrlOut*** signal register ***ctrlOut_reg***, the register's level is flipped every time the system goes to ***consumerReq***.

```
-- runs at consumerReq
phase_1_handshake: process (curState)
 begin
      if (curState = consumerReq) then
        ctrlOut_reg <= not ctrlOut_reg;
      end if;

end process phase_1_handshake;
ctrlOut <= ctrlOut_reg;
```

The latter phase is responsible for detecting a change in signal level (edge sensitive instead of level sensitive). The data generator detects that ***ctrlIn*** (corresponds to ***ctrlOut*** at data processor) has made a transition and transmits its own ***ctrlOut*** signal that indicates some data is available on the dataline. This transition is detected by the data processor and ***data*** is retrieved. It settles into a pending state, then repeats the identical cycle if ***start*** is detected, otherwise the machine falls back into ***INIT*** state.

```
phase_2_handshake: PROCESS(clk)
  BEGIN

    IF clk'event AND clk='1' THEN
       ctrlIn_delayed <= ctrlIn;

    END IF;

END PROCESS phase_2_handshake;
ctrlIn_detected <= ctrlIn_delayed xor ctrlIn;
```

Peak Detection - by Casper Wang

I am in charge of the processing of the data and finding the peak along with the peak index. This design also considered the extra complexity of regarding the data in a signed fashion.

The algorithm works like this:
Having two arrays for storing the peak value and its left 3 and its right 3 values (see below)

```
result_reg, cur_reg: CHAR_ARRAY_TYPE (6 downto 0)
```

Having three counting signals (see below)

```
SIGNAL count2: Inte

-- index for maximu
SIGNAL countResult:

-- current running
SIGNAL countCurr: I
```

The effect of **count2** will be explained later, but in general, result is used to store the current maximum peak and index, cur is used to store the current value and index.

When new data comes in, what happens first is that the **cur_reg** array will be shifted to the left by one bit, and new data would go into the middle which would be index 3. After this, the **cur_reg**[3] value will be compared with **result_reg**[3]. If it's bigger, the entire cur_reg array will be replaced on the result_reg array. If it's smaller, the **cur_reg**[3] will be appended to **result_reg**. That's where **count2** comes in, it tells the program where to append the data on.

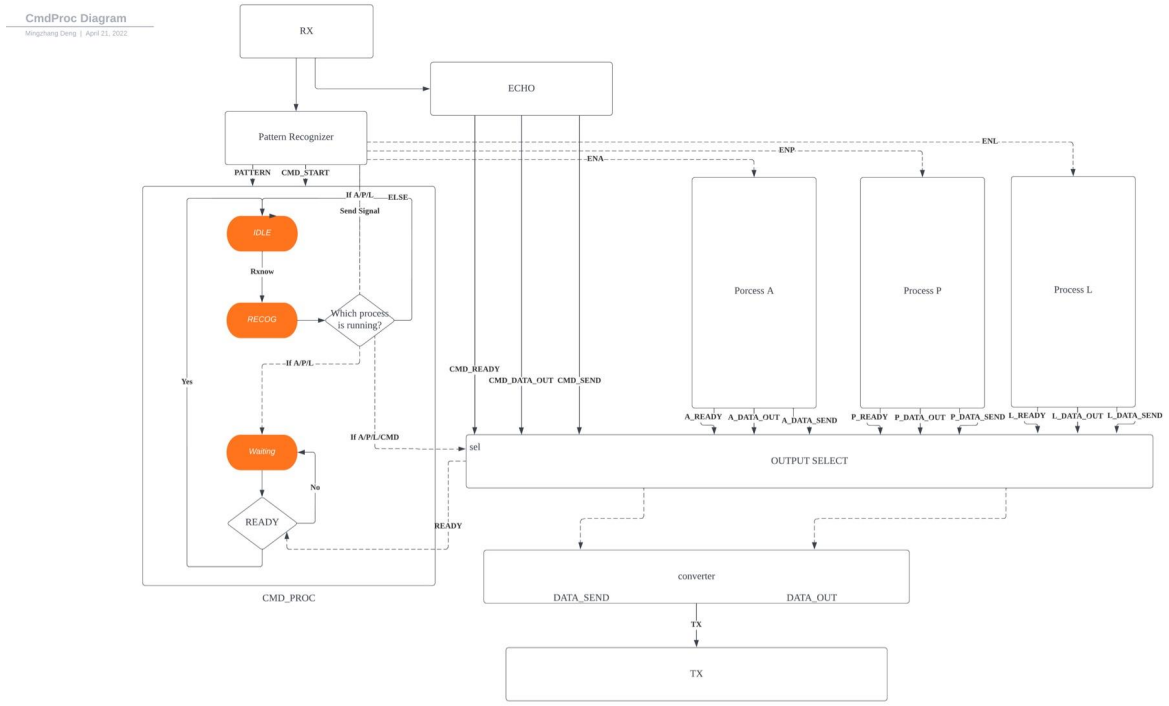
The idea behind is that, this way, the memory space would be saved. As otherwise, we would need to store the entire dataset (say 500 data) in an array and find peak that way. This is clearly not very memory-efficient. So by using what we called **shift-compare** operation. We can maintain the peak value and index without using too much memory.

After Otis sent me a signal indicating that all the data had been sent, and I finished processing the last data. I will output the **maxIndex**[1] and **dataResults** along with **seqDone** to the command processor.

[1] (8 bit Binary to BCD converter - Double Dabble algorithm, 2022)

Design and Simulation of Command Processing module

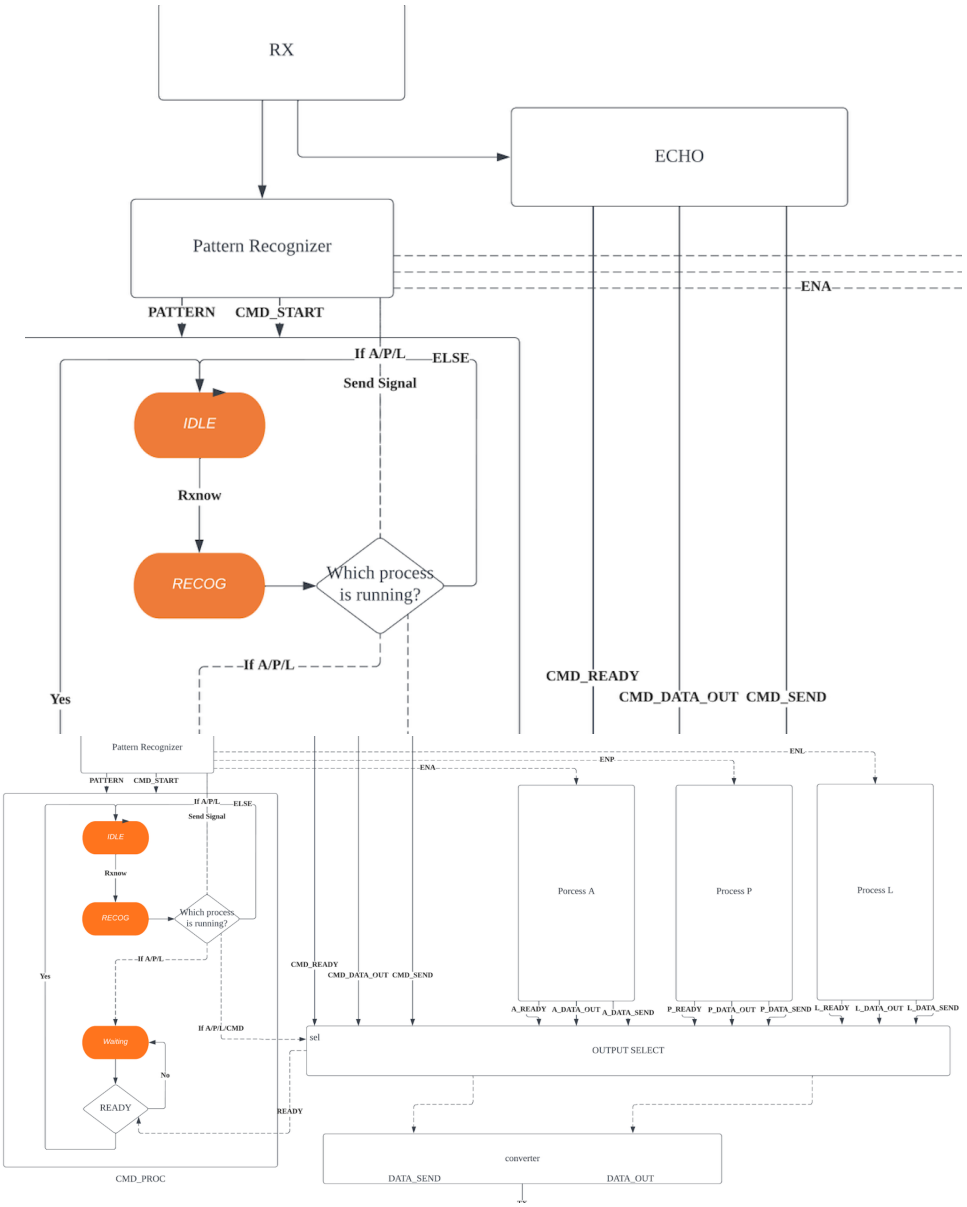
Command Processor Overall Design (Mingzhang Deng)



According to the system component chart above, Entire Command Processor is separated into subcomponents below:
1.Pattern Recognizer — Component which recognizes the input pattern.
2.Echo — Echo input from RX unit
3.CMD_PROC — Command Processor main process
4.A Proc —- Process dealing with command A
5.P Proc — Process dealing with command P
6.L Proc — Process dealing with command L
7.Output Selector — Select output value according to current state of Command Processor
8.Output Converter — Formatting the output sending to TX unit.

Typical behaviour of Command Processor is described below:
When commands come, the Echo component will Echo the input directly to TX Unit. At the same time, Pattern Recognizer will Recognize the input pattern. If the input pattern matches one of A/P/L commands, Pattern recognizer will send the start signal to ***CMD_Proc*** when it receives ***txdone***. (worth notice here that we assume pattern recognizer is way more faster than the tx unit, which means Pattern recognizer will finish its task before txdone goes high)

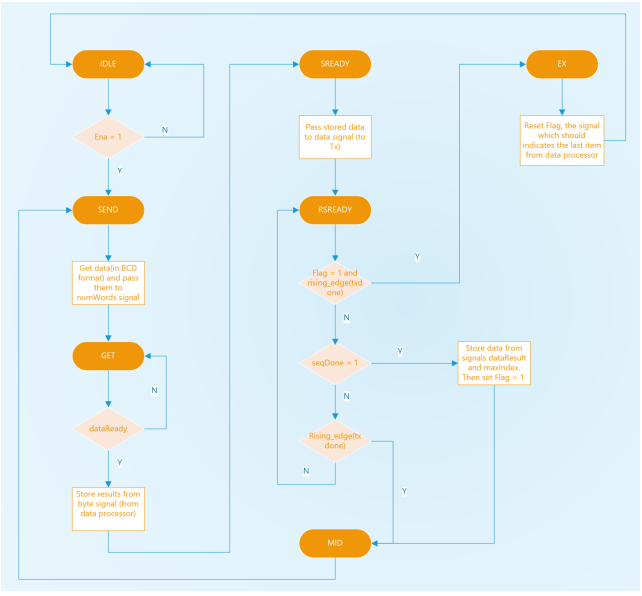
CMD_Proc receives the Command , and then sends signal ENA/P/L to the corresponding component. Component A/P/L starts to work. At the same time, the Output Selector changes to corresponding state, which allows the currently working component to utilise the TX



component to complete its task.

When Work is done, Component A/P/L will set the ready signal high for one clock cycle to notify ***CMD_Proc*** work has been done. Then ***CMD_Proc*** will go back to IDLE state waiting for the next command.
There are two output formats. One is to directly print messages from the component. The other one is to convert the data into Hex (Eg. 11110001 => F1) Therefore, we create a converter as a proxy to handle output formatting.

COMPONENT DETAIL



Process A (Yichen Dong)

Data communication between the command processor and the data processor is achieved by means of a finite state machine.

To achieve the functionality, the state machine is divided into the following parts：

IDLE: The initial state of the finite state machine, from which everything is processed when the A process is activated.

SEND: Command processor gets the data from the RX, it converts the corresponding data into the BCD
When the format required by the data processor and sends it to the A process, which is, into the ***GET*** state and forwards the data to the data processor via numWords. Then move on to the next state - ***GET***.

GET: This state is used to wait for and receive data from the data processor when it has finished processing. When the rising edge of the dataReady signal is detected, the data appearing on the inputByte signal is saved and sent to Tx in the next state.

SREADY: As the next state of the ***GET***, it is used to send the stored data (from the data processor) to the Tx for processing.

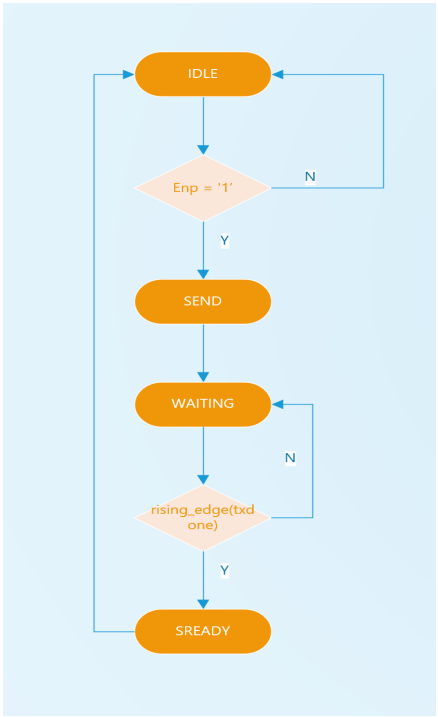
RSREADY: It is mainly used to detect the Txdone signal and to initiate the next round of processing after the Tx has finished processing the data (obtaining the data and sending it to the data processor, then sending the processed result to the Tx). After that, go to ***MID*** stage for launching new round. If the data processor has finished processing the last number at this point, seqDone is on the rising edge and saves the dataResult and maxIndex while sending the data result to Tx and then enters the ***EX*** state.

MID: Used as an auxiliary state to bridge two different rounds of data processing and to help synchronise clock cycles.

EX: Also used as an auxiliary signal for the final round of processing of this data sequence and for bridging back to the initial state.

Process P (Vincent Wong, Yichen Dong)

The ASM chart is shown below.



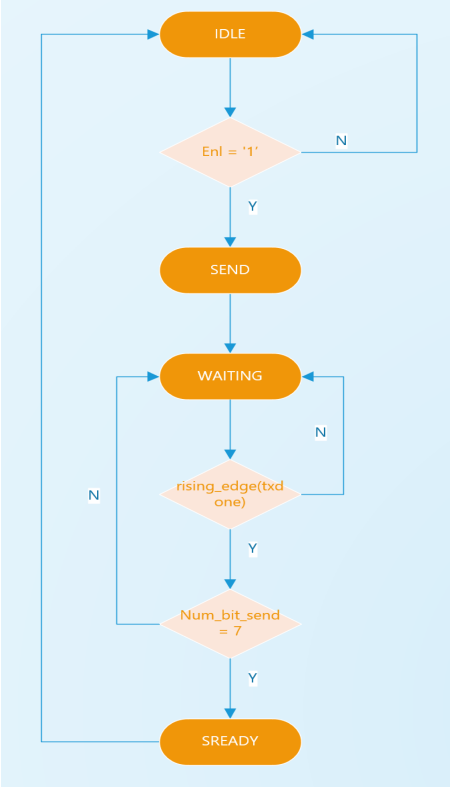
IDLE: Process P waits in IDLE for '***enp***' to go high before going to SEND.

SEND: In ***SEND*** state the '***send_proc***' process sends the peak byte to '***dataOut***'.

WAITING: In ***WAITING***, the process checks for the rising edge of '***txdone***' to make sure the data has been transmitted before going to the ***SREADY*** state.

SREADY*:** The ***SREADY state is the state the process enters once the command has been carried out so the process can reset.

send_proc: 'The ***send_proc***' process is used to send the peak byte out. Once the Process P is in the ***SEND*** state, the peak byte is sent through '***dataOut***' and '***dataSend***' goes high to communicate with the Output Selector. In the next clock cycle, when Process P is in the ***WAITING*** state, where '***dataSend***' is set back to low. In the clock cycle after, in the ***SREADY*** state Process P ends and '***ready***' is set high for the next process.



Process L (Vincent Wong, Yichen Dong)

Here is the ASM chart.

IDLE: Process L waits for '***enL***' to go high before L process is carried out.

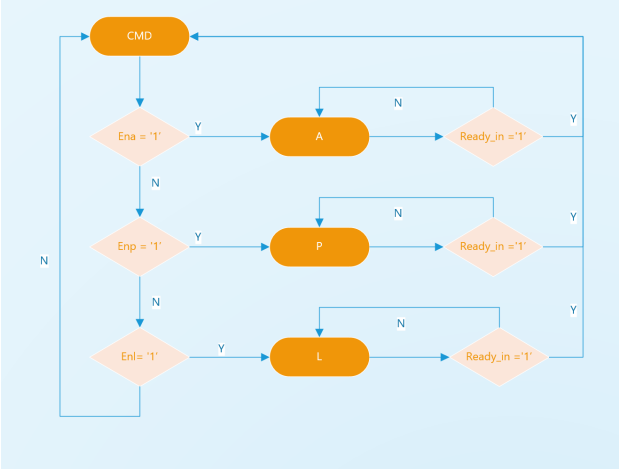
SEND: While in ***SEND***, ***send_proc*** is carried out which sends 1 byte every clock cycle from the dataResult given from the Data Processor to dataOut.

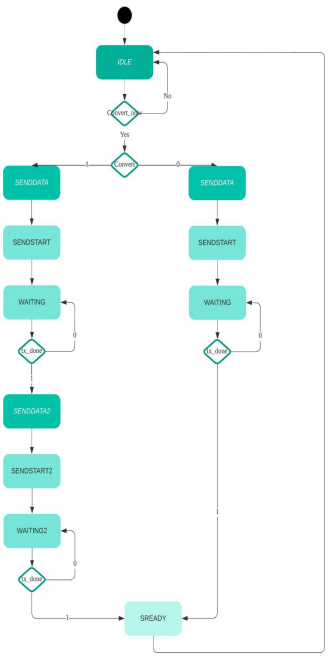
WAITING: After every ***SEND*** cycle, the state goes to WAITING which checks if all 7 bytes have been sent to the output, using signal '***num_bit_send***' which iterates for every byte sent. If not, '***nextState***' is set to ***SEND*** to send the next byte.

SREADY: The state Process L enters once all 7 bytes have been sent to the output.

send_proc: The process used to send data out. It waits for the component to be in the ***SEND*** state before any instructions are carried out. Once in ***SEND***, using the integer signal '***num_bit_send***', the process sends the current byte to '***dataOut***', sets 'dataSend' high to communicate with the Output Selector and then iterates upon '***num_bit_send***'. In the next clock cycle, if in the ***WAITING*** state, '***dataSend***' is set to low and for another byte to be sent in the clock cycle after. In the ***SREADY*** state, '***send_proc***' resets Process L and sets '***ready***' high for the next process.

Output Selector (Mingzhang Deng, Yichen Dong)





Converter (Mingzhang Deng):

Under this project, we have two possible output formats.

1. Directly output data to the TX component. (eg. data = 00110001, output '1')
2. Output data as hexadecimal characters. (eg. data = 00110001, output '3''1')

In order to relieve each process from dealing with formatting, Converter, as an output proxy, automatically converts the output into the intended format.

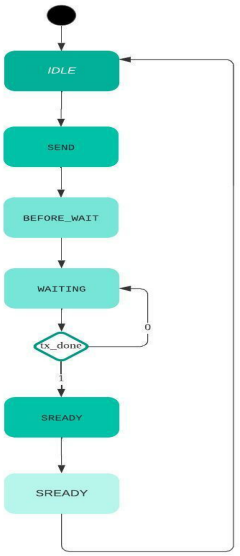
The workflow of the converter is shown below.

Converter, as a proxy to the TX component, can be ignored or just treated as TX component when interacting with this component.

Because of the existence of this component, other components need not to write formatting related states. Reduce the workload for other component designers.

ECHO (Mingzhang Deng):

Echo is the component simply echoing everything received from the RX component. The ASM Chart of this component shown on the right.



Pattern Recognizer (Mingzhang Deng):

Pattern recognizer is the component used to deal with input pattern recognition and receive the input parameters (three numbers after A character) .

According to the system component chart, the pattern recognizer and Echo component work together at the same time. For these two components, one assertion must be achieved in order to make sure they functioned correctly: the tx component must send the ***txNow*** signal later than the 2 clock cycle.

Simulation Results

Data Processor Wave Diagrams

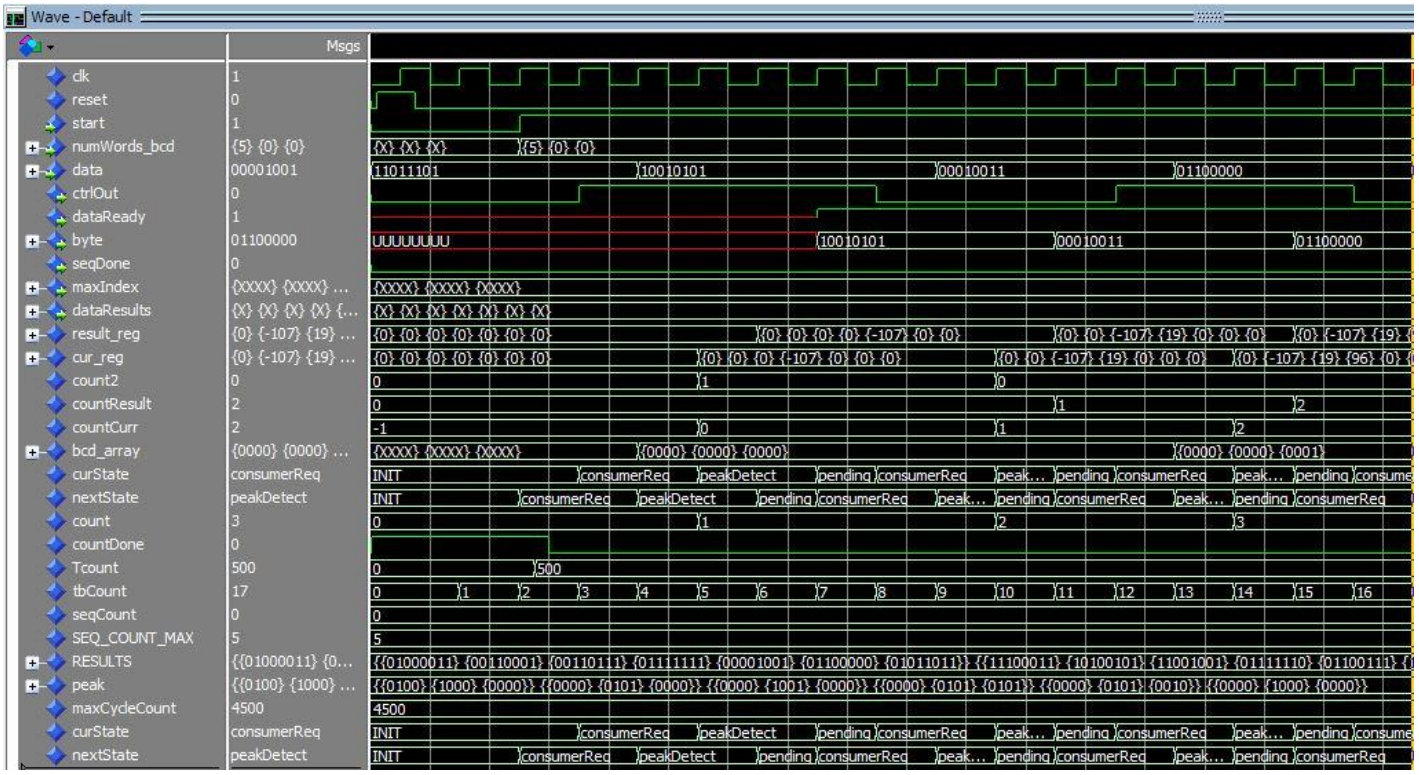


Fig 1: Initialising Data Processor module

The program starts with an asynchronous high reset, which initialises all signals, including *count*, ***cur_reg*** (which stores the current register), ***maxIndex***, and forces all states to ***INIT***. The *start* signal is asserted after 2.5 periods along with the numWords signal, the former is a signal which drives ***nextState*** to transition into ***consumeReq*** *(short for consumer request)*, the latter is type casted into an integer (***Tcount***) through an external module as the counter operates on integers. As the ***curState*** jumps into ***consumeReq***, an outward signal of ***ctrlOut*** makes a transition from high to low (or vice versa). The data generator module detects this transition and transmits some input on the data line while it makes a transition on the ***ctrlIn*** signal, where the two-phase handshake protocol recognizes it as a ***dataReady*** signal.

The post-process involves counting, shifting, replacing or appending. In the first clock cycle after detecting the (internal) ***dataReady*** signal, one count is incremented and the input data is shifted into the current register ***cur_reg***. During the next clock cycle, the middle bit of the ***cur_reg*** is compared with the ***result_reg***, if the former is bigger, the entire ***result_reg***

gets replaced, otherwise, the data is appended to the end. As this process finishes, a ***procDone*** signal is asserted high, which indicates ***nextState*** to loop back into ***consumerReq***, and the request-receive process repeats itself. If anything happens other than shifting or appending, or if the state machine falls into an unknown state, there's a catch-all statement which resides the next back to ***INIT***.

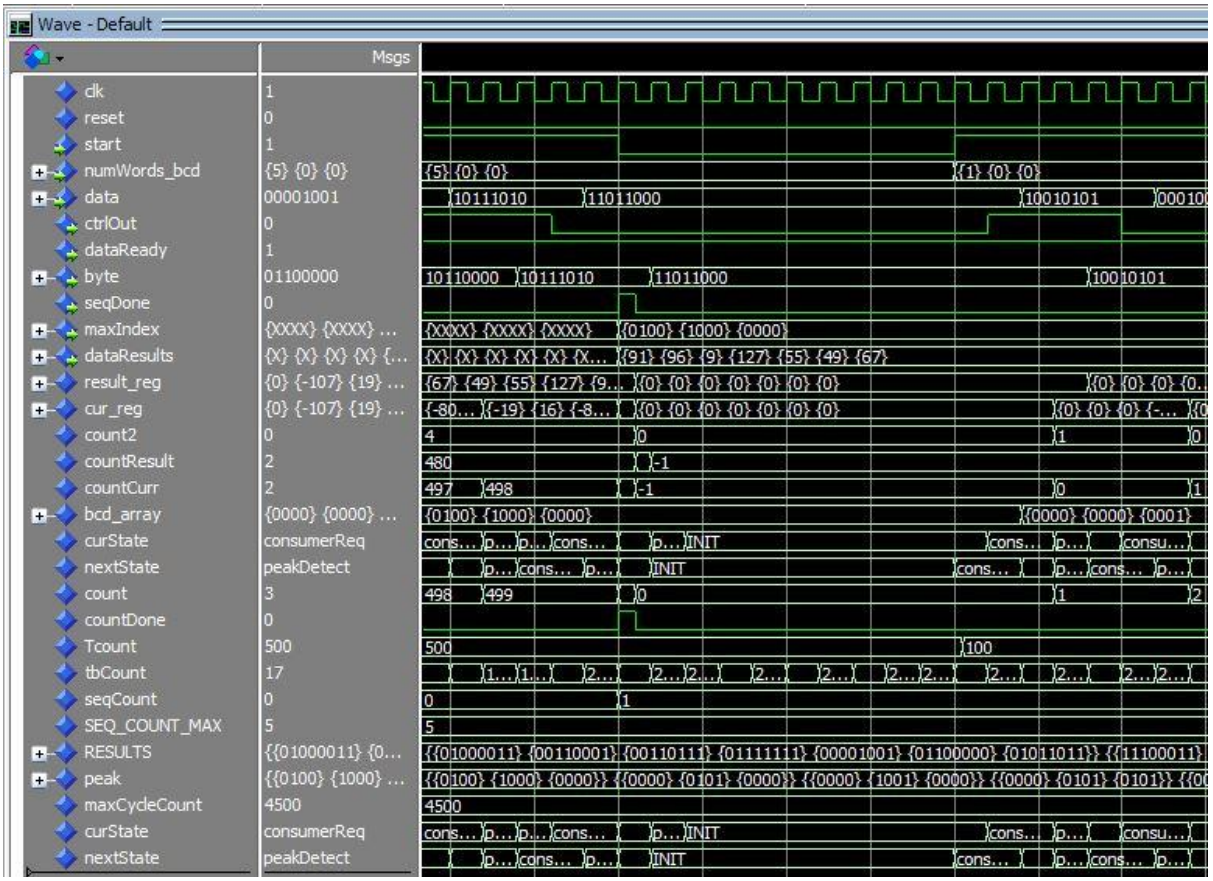


Fig 2: 500 data points looped through

At the end of the data collection / analysis process, if ***Tcount*** is reached, the ***countDone*** (internal) signal and the ***seqDone*** (external) signal are asserted along with ***localReset*** (an internal reset controlled by internal signals). The system waits for another ***start*** signal from the command processor module, and the above starts again, as illustrated in figure 2.

The next 5 cycles are identical, with a few exceptions. The first exception is that the number of words is shortened from 500 to 100. The second exception is that after each cycle there's no wait time. This sometimes causes errors in output because data points are uncounted due to timing problems. Therefore, when ***localReset*** is high, instead of resetting the counter to 0, it was set to -1 to accommodate for the missing count, as illustrated by figure 3.

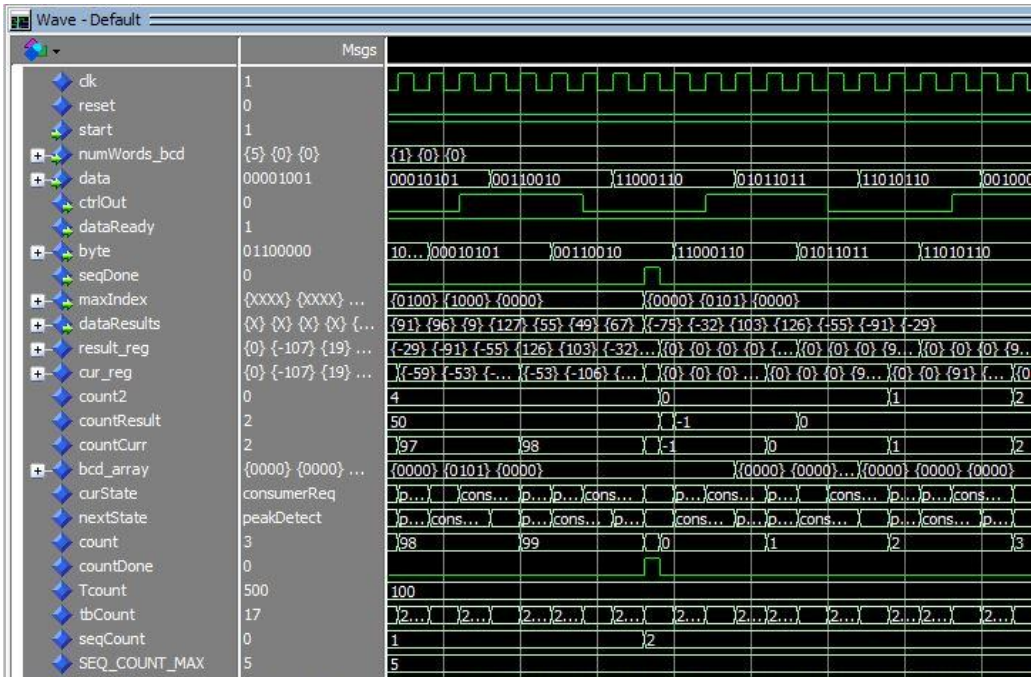


Fig 3: 100 words cycled through

Command Processor Wave Diagrams

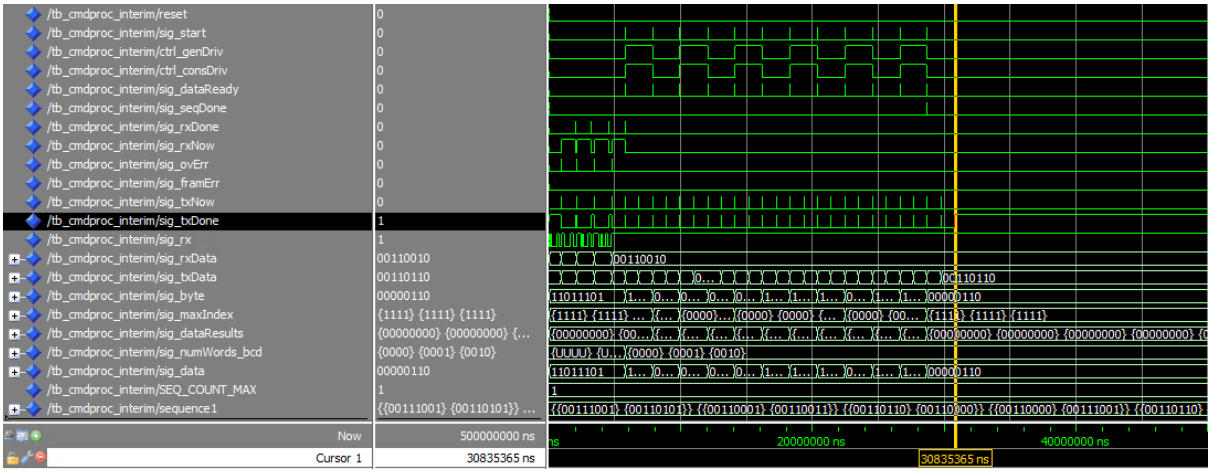
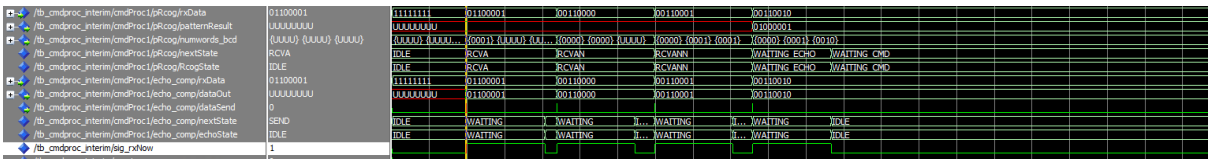


Fig. Command A

At the beginning of the program, reset signal sets every component to ***IDLE/CMD*** state(Initial state).

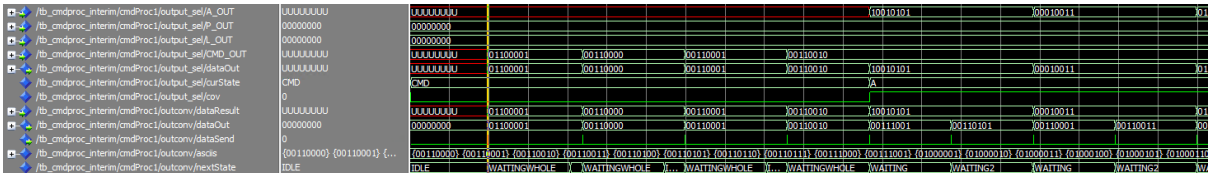
When we receive the first ***rxNow***(991505ns) , Echo and Pattern Recognizer(pRcog in graph) goes into ***RCVA*** state, and waits for the Tx unit to finish echoing the input result.



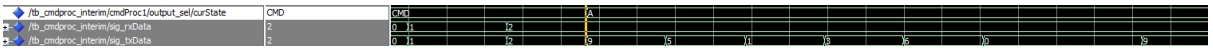
Once received ***rising_edge(txDone)*** from the TX unit, it sets the ***RxDone*** signal high for one clock cycle.

At the same time, Echo component sets ***dataSend*** to the received data and ***dataOut*** = 1 for 1 clock cycle then goes to ***WAITING*** State waiting for ***rising_edge(txDone)***.

dataSend and ***dataOut*** signals are not directly connected to the TX component. Instead, they are connected to the Output Selector. Let's see what happening on Output Selector.



Output selector(outpuit_sel in graph) is at ***CMD*** state, so it will map all signals from the ECHO component to the TX component.

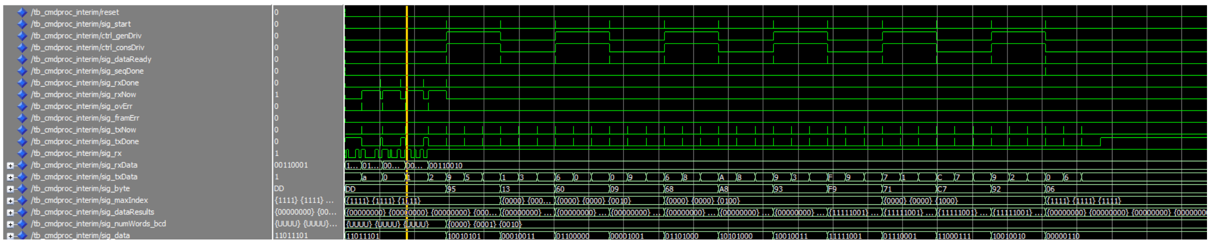


After Receiving all commands from the RX component, The command is sent to the CMD_Proc and CMD_Proc changes its state according to the input command. In this case, CMD_Proc changes to A state after receiving "A" . At this Point, CMD_Proc passes its control privilege to A_Proc and waits for signal ***ready*** = 1 from A_Proc(Running details above) .

When A_Proc's work is done , the ready signal will be set to high for one clock cycle, notifying CMD_Proc that all things are done. Then CMD_Proc deals back to the ***IDLE*** state waiting for the next command.

TestBench result :

```
VSIM 17> run
# ** Warning: There is an 'U'|'X'|'W'|'Z'|'-' in an arithmetic operand, the result will be 'X'(es).
#    Time: 0 ns  Iteration: 0  Protected: /tb_cmdproc_interim/dataConsume1/<protected>
# ** Warning: There is an 'U'|'X'|'W'|'Z'|'-' in an arithmetic operand, the result will be 'X'(es).
#    Time: 0 ns  Iteration: 0  Protected: /tb_cmdproc_interim/dataConsume1/<protected>
# ** Warning: There is an 'U'|'X'|'W'|'Z'|'-' in an arithmetic operand, the result will be 'X'(es).
#    Time: 0 ns  Iteration: 0  Protected: /tb_cmdproc_interim/dataConsume1/<protected>
# ** Warning: There is an 'U'|'X'|'W'|'Z'|'-' in an arithmetic operand, the result will be 'X'(es).
#    Time: 0 ns  Iteration: 0  Protected: /tb_cmdproc_interim/dataConsume1/<protected>
# ** Warning: There is an 'U'|'X'|'W'|'Z'|'-' in an arithmetic operand, the result will be 'X'(es).
#    Time: 0 ns  Iteration: 1  Protected: /tb_cmdproc_interim/dataConsume1/<protected>
# ** Warning: There is an 'U'|'X'|'W'|'Z'|'-' in an arithmetic operand, the result will be 'X'(es).
#    Time: 0 ns  Iteration: 1  Protected: /tb_cmdproc_interim/dataConsume1/<protected>
# ** Note: =====Process of Sequence No.1 starts.
#    Time: 5832335 ns  Iteration: 1  Instance: /tb_cmdproc_interim
# ** Note: =====Process of Sequence No.1 is done.
#    Time: 40211015 ns  Iteration: 3  Instance: /tb_cmdproc_interim
# ** Note: =====A total of 12 bytes has been processed;
#    Time: 40211015 ns  Iteration: 3  Instance: /tb_cmdproc_interim
# ** Note: =====Process of Sequence No.2 starts.
#    Time: 137832335 ns  Iteration: 1  Instance: /tb_cmdproc_interim
# ** Note: =====Process of Sequence No.2 is done.
#    Time: 175336345 ns  Iteration: 3  Instance: /tb_cmdproc_interim
# ** Note: =====A total of 13 bytes has been processed;
#    Time: 175336345 ns  Iteration: 3  Instance: /tb_cmdproc_interim
```



References

1. Vhdlguru.blogspot.com. 2022. *8 bit Binary to BCD converter - Double Dabble algorithm*. [online] Available at: <https://vhdlguru.blogspot.com/2010/04/8-bit-binary-to-bcd-converter-double.html> [Accessed 27 April 2022].