Contributions:

Tianyu Ma was responsible for explaining the numbering system to team B and creating the original FSM chart. She played a role in the receiving ANNN and all decoder (ASCII) codes, and also assisted Yunji Nam with debugging, and finally completed the formal version of FSM chart.

Yunji Nam redraw the draft FSM chart and completed the remaining codes, especially for putty part. She optimized the entire codebase to achieve the desired goal and made significant contributions to debugging.

Qian Chen worked on the command processor in VHDL 2008 version due to system issues. He completed the task independently, provided a detailed report that included the FSM chart and functional block explanations. Qian Chen and Ziyang Deng tested the code, which was approved to meet the goal provided the data types used in VHDL version were transferable to match the data type used in VHDL 2008 version.

Common: FSM ideas generated and finished by Tia Ma and Yunji Nam. We compared and discussed our daft FSM by Tia Ma and new draft FSM by Yunji Nam. Summary of team’s work in report. All checking simulations results and experiencing synthesis and implementation. Built up the structure and framework for the report.  Simulation and synthesis experience.

1 Receiving ANNN: (Tia Ma)

This processing is used to process the data input from Receiver module and detect the ANNN, which A is referring to the ‘A’ or ‘a’ in ASCII format and the ‘NNN’ is refers to the three ASCII number, referring to line 128.

* A: The code for detecting character A begins in the ‘valid\_A\_idle’ state, then waits for two conditions to be meet: the ‘rxNow’ signal is high, indicating that data has been received, and the ‘txDone’ signal is high, indicating that the previous transmission has been completed and ready to send another byte. Once both conditions are met, the ‘v\_rxDone’ signal is set to high and jump to the ‘valid\_A\_check’ state.

In the ‘valid\_A\_check’ state, the ‘‘v\_txNow’’ signal is set to high, to trigger a sent. The received data is then checked to determine if it matches any valid inputs. If the received data is a number between 0-9, then transitions to the ‘valid\_1\_idle’ state to look for the second digit. If the received data is an 'A' or 'a', then transitions back to the ‘valid\_A\_idle’ state to wait for a new input. If received 'l' or 'L', 'p' or 'P', and the ‘processed’ signal is high, then transitions to the ‘putty\_n\_1\_wait’ state to wait for ‘L’ command & ‘P’ command processing module. If the received data does not match any of the above conditions, then transitions to the ‘INIT\_idle’ state, to start a new process.

* NNN: The code of detecting NNN starts from the ‘valid\_1\_idle’ state and waits for two conditions: the next input to be received, indicated by ‘rxNow’ being high, and for the previous transmission to be completed, indicated by ‘txDone’ being high. Once these two conditions are meet, ‘v\_rxDone’ is set to high, indicated that the input has been received, and transitions to the ‘valid\_1\_check’ state. In the ‘valid\_1\_check’ state, ‘‘v\_txNow’’ is set to high, indicating ready for transmit. Then, it checks the received data whether match any of the valid inputs that’s looking for, which is the same as in the ‘valid\_A\_check’ state. If it matches a number between 0-9, then transitions to the ‘valid\_2\_idle’ state to look for the third digit. If it's 'A' or 'a', then transitions back to the ‘valid\_A\_idle’ state. If it's an 'l' or 'L', 'p' or 'P', and ‘processed’ signal is high, then transitions to the ‘putty\_n\_1\_wait’ state to wait for ‘L’ command & ‘P’ command processing module. If it's none of the above, then jump to the ‘INIT\_idle’ state.

In the ‘valid\_2\_idle’ state, it repeats the implementation of processing second digit. The only change is, after matched the number 0-9, it then jumps to the ‘putty\_n\_1’ waiting for the processing of transmitting ANNN to data processor.

2 Putty Part: (Yunji Nam)

In this implementation, putty part is an important part for command processor. Putty sates have been in the whole system for waiting for the transmission of each character to be completed before moving onto the next one.

* For ‘putty\_n/r\_1/2\_wait/tx’：

1. ‘putty\_n\_2\_wait’ state waits for ‘txDone’ to go high before transmitting the second character of the command. If ‘txDone’ is already high, it moves to the next state (putty\_n\_2\_tx) and transmits the second character. If not, it stays in this state and continues to wait.
2. ‘putty\_n\_2\_tx’ state transmits the second character of the command and waits for ‘txDone’ to go high. If ‘txDone’ is high, it sets ‘v\_txNow’ to 1 and moves to the next state (putty\_r\_2\_wait), which waits for a response after transmitting the third character. If not, it stays in this state and continues to wait.
3. ‘putty\_r\_2\_wait’ state waits for a response after transmitting the third character of the command. If ‘txDone’ is high, it moves to the next state (putty\_r\_2\_tx), which transmits the fourth character and waits for ‘txDone’ to go high. If not, it stays in this state and continues to wait.
4. ‘putty\_r\_2\_tx’ state transmits the fourth character of the command and waits for ‘txDone’ to go high. If ‘txDone’ is high, it sets ‘v\_txNow’ to 1 and moves to the next state (cmd\_wait). If not, it stays in this state and continues to wait

* For ‘putty\_eq\_1\_wait/tx’:

1. ‘putty\_eq\_1\_wait’ state waits for the transmission done signal (‘txDone’) to go high. If it is high, it moves to the next state (putty\_eq\_1\_tx) and transmits the first character of a command. If not, it stays in this state and continues to wait for ‘txDone’ to go high.
2. ‘putty\_eq\_1\_tx’ state transmits the first character of a command and waits for ‘txDone’ to go high. If ‘txDone’ is high, it sets a variable ‘v\_txNow’ to 1 and checks the value of a counter (‘count\_eq’). If ‘count\_eq’ is greater than 4, it moves to the next state (‘putty\_n\_2\_wait’). Otherwise, it returns to the ‘putty\_eq\_1\_wait’ state to transmit the next character of the command.

* For ‘putty\_n/r\_3/4\_wait/tx’:

1. ‘putty\_n\_3\_wait’ state waits for a response after transmitting the third character of the command. If ‘txDone’ (transmission done) is high, it moves onto the next state ‘putty\_n\_3\_tx’ for transmitting the third character of the command and waits for transmitting finish while ‘txDone’ is high. Otherwise, it stays in this state until ‘txDone’ becomes high.
2. ‘putty\_n\_3\_tx’ state is for transmit ting the third character of the command and waits for ‘txDone’ to become high. If ‘txDone’ is high, it sets ‘‘v\_txNow’’, which means transmit now, to 1 and moves onto the next state ‘putty\_r\_3\_wait’, waiting for a response after transmitting the fourth character of the command. If ‘txDone’ is not high, it stays in this state until ‘txDone’ becomes high.
3. ‘putty\_r\_3\_wait’ state waits for a response after transmitting the fourth character of the command. If ‘txDone’ is high, then moves onto the next state ‘putty\_r\_3\_tx’, which transmits the fourth character of the command and waits for ‘txDone’ to become high. Otherwise, it stays in this state until ‘txDone’ becomes high.
4. ‘putty\_r\_3\_tx’ state transmits the fourth character of the command and waits for ‘txDone’ to become high. If ‘txDone’ is high, it sets ‘‘v\_txNow’’ to 1 and moves onto the next state (putty\_eq\_2\_wait), which waits for a response after transmitting the equal’s sign. If ‘txDone’ is not high, it stays in this state until ‘txDone’ becomes high. And so is putty\_r\_4\_wait and putty\_r\_4\_tx.

* For ’putty\_eq\_2\_wait/tx’:

1. ‘putty\_eq\_2\_wait’ state waits for a response after transmitting the equal’s sign. If ‘txDone’ is high, it moves onto the next state (putty\_eq\_2\_tx), which transmits the equals sign and waits for ‘txDone’ to become high. Otherwise, it stays in this state until ‘txDone’ becomes high. putty\_eq\_2\_tx: This state transmits the equals sign and waits for ‘txDone’ to become high. If ‘txDone’ is high, it sets ‘‘v\_txNow’’ to 1 and checks if ‘count\_eq’ (the count of equals signs transmitted so far) is greater than 4. If ‘count\_eq ‘is greater than 4, it moves onto the next state (putty\_n\_4\_wait), which waits for a response after transmitting the fifth character of the command. Otherwise, it moves back to putty\_eq\_2\_wait to transmit another equals sign. If ‘txDone’ is not high, it stays in this state until ‘txDone’ becomes high.
2. ‘putty\_eq\_2\_tx’ state transmits the equals sign and waits for ‘txDone’ to become high. If ‘txDone’ is high, it sets ‘v\_txNow’ to 1 and checks if ‘count\_eq ‘(the count of equals signs transmitted so far) is greater than 4. If ‘count\_eq ‘is greater than 4, it moves onto the next state (putty\_n\_4\_wait), which waits for a response after transmitting the fifth character of the command. Otherwise, it moves back to putty\_eq\_2\_wait to transmit another equals sign. If ‘txDone’ is not high, it stays in this state until ‘txDone’ becomes high.

3 Decoder: Processing byte: (Tia Ma)

This process is used to convert a byte signal into a string of ASCII characters and is sensitive to the rising edge of the ‘clk’ signal. Inside the process, there are two case statements that decode the ‘byte’ input into ASCII characters and are stored in ‘ANNN\_dataTx’. In this case, ‘byte’ signal is 8 bits wide and contain 2 pieces of information (4 bits each) in binary format referring to hex information, includes integers from 1 to 9 and letter from ‘A’ to ‘F’, which is shown as Figure.2 [2]:

Table

Description automatically generated

Figure 2: Binary to Hex table

The first case statement decodes the most significant nibble of ‘byte' (bits 7 downto 4) and stores the corresponding ASCII character in bits 15 downto 8 of ‘ANNN\_dataTx’. The second case statement decodes the least significant nibble of ‘byte’ (bits 3 downto 0) and stores the corresponding ASCII character in bits 7 downto 0 of ‘ANNN\_dataTx’.

If ‘byte’ does not match any of the specified cases, then the default case ‘others’ is used, which stores the null character in the corresponding bits of ‘ANNN\_dataTx’. After the ‘others’ case, the processor will jump to the ‘INIT\_idle’ state to reset the processor and waiting for moving on ‘valid\_A’, waiting for the new data input.

4 Decoder: ‘L’ command & ‘P’ command: (Tia Ma)

LP command is the important part of command processor and in this implement, this is achieved by ‘L’ command and ‘P’ command in FSM chart. L command is to list the peak byte and 3 bytes on both left and right side, referring to line 345 and the P command is to print the peak byte by following the peak byte index, referring to line 381.

‘L’ command and ‘P’ command will only be implemented if ‘seqDone’ signal is set to high and use a synchronous process triggered on a rising edge of the clock signal. In details, the process for the 'L' command takes an additional ‘count\_L’ signal as an input, which indicates the byte index to start decoding from. The process for the 'P' command only uses the ‘out\_dataResults’ signal as an input.

In implementation of ‘L’ command and ‘P’ command, both processes use a case statement to decode the input signal to suit the team numbers coding style, which could be optimised in the future. The input signal is checked for the most significant nibble (4 bits) and least significant nibble separately. Depending on the value of the input signal, the output signal ‘L\_dataTx’ or ‘P\_dataTx’ is assigned a value that represents an ASCII character.

The ASCII values for the nibbles are hardcoded into the case statements. For example, if the most significant nibble is ‘0000’, the value of ‘L\_dataTx’ or ‘P\_dataTx’ is set to ‘00110000’, which is the ASCII code for the digit 0. If the most significant nibble is ‘1010’, the value of ‘L\_dataTx’ or ‘P\_dataTx’ is set to ‘01000001’, which is the ASCII code for the letter 'A'.

If the input signal does not match any of the cases, the output signal is set to ‘00000000’, which is the ASCII code for the null character. Here is the reference table of ASCII table (Figure.1) [1]:

Table

Description automatically generated

Figure 1: ASCII Conversion Table

Moreover, for ‘P’ command, in order to design a communication protocol between two devices, where data is transmitted using UART (Universal Asynchronous Receiver Transmitter) protocol. A state machine for BCD has been created (lines 383-476), which could be seen in the draft state machine draw by Yunji Nam. This is based on the signal ‘txDone’ which indicates the current byte of data has been transmitted if ‘txDone’ is high. P buffers have been created to store the nth character to be sent and P transmit states have been created for transmitting the nth character and wait for ‘txDone’ to be high.

It is important to note that a space needs to be printed after each ‘s\_dataTx’ being sent, which is referring to the ‘putty\_space’ statement, indicated by line 464-476. A separate constant consisting of an ASCII space is stored in the constants as '00100000'. After each byte is sent in the L or P command, then jump to the ‘putty\_space’ state to create a space and sent, indicated by line 464.

Result of Command Processor:

Waiting for Ziyang to fill this part. Many thanks!

Discussion:

Despite the positive attitudes and contributions of every team member, there were some misunderstandings within team B. Initially, a time arrangement was made, and a timetable was set for the project. However, some team members had other work to attend to, and Qian Chen faced an emergency issue during the Easter break. As a result, the rest of the team members started working on the codes, assuming that Qian Chen would join them upon his return. However, they did not anticipate that he would complete the entire command processor by himself.

Upon checking Qian Chen's code, the team realized that he had used the old 2008 version of VHDL, which made it challenging to understand and synthesize his work. Qian Chen and Ziyang Deng took on the responsibility of reviewing the code. Eventually, both Tia Ma and Yunji Nam completed the command processor codes and Qian Chen completed the command processor codes in VHDL 2008 and handed them over to Ziyang Deng for integration.

For codes optimization, in receiving ANNN part, at ‘valid\_A’ state (referring to line 142), this could be simpler if we just detect the least 4 significant bits, because in 0-9, all of them have the same most 4 significant bits of 0011. For the whole codes, this could be optimized by adding more registers and buffers to decrease the states numbers.

Reference:

[1] https://sites.google.com/site/computingicttoolbox/Home/form-3/ascii-and-ebcdic

[2] https://study-ccna.com/ipv6-address-format/

(For Ziyang Deng:

1. Is chen qian’s version can be integrated properly? If so, plz add a line about his codes could also meet the goal and integrated with DP and a picture of his output. If not, just post the output of Version of Tia Ma and Yunji Nam (referring to cmdProc.vhd in Github), and add a line talk about Qian’s work also but without presenting output.
2. I will email you about the formal FSM chart and the detailed state machine, which should all be uploaded for understanding. The first one is generated by Tia Ma and second one is draw by Yunji Nam. Could u plz insert these two into the right place with the right format (not too big either not too small)? Thanks a lot!
3. Plz help to cite the reference properly