Design and Implementation of a Peak detector using UART

Ziyang Deng

ts20814@bristol.ac.uk

Oliver Bath

Rm21809@bristol.ac.uk

Tia Ma

ki19146@bristol.ac.uk

Yunji Nam

oq20050@bristol.ac.uk

Qian Chen

bh21669@bristol.ac.uk

This report contains the relevant information on our project plan and describes the architecture of our entire system, and of the data processor and command processor.

Contents:

Summary:

Team A (Ziyang Deng & Oliver Bath) completed all the allocated tasks for data processor: the FSM, Block-Level sketch, and VHDL code contains data path and control path in a single file named dataConsume.vhd ， and VHDL code that describes the array which stores bytes in another single file named dataArray.vhd. In addition, team A has successfully run both stand-alone simulations and full system simulation with provided code for the command processor, which shows ideal signals' wave during simulation. Furthermore, combined with provided codes, the codes for data processor are successfully run for stand-alone synthesis and implementation(routing) without error and serious warning. Eventually, the top.bit file has been generated and been downloaded into A7 board, through using Putty and PC keyboard's commands, the bit file shows the ideal functionality for any required commands, which convinces the team that has achieved the full functionality for the data processor.

Team B（Tia Ma & Yunji Nam & Qian Chen) completed partial allocated tasks for command processor: the FSM, detailed FSMs and VHDL code. However, the Block-level sketch is not provided. The team successfully ran both simulation and synthesis with generated bit file without error message. Unfortunately, it didn't achieve full functionality and output incorrect results in Putty. In addition, the waveform from simulation seems not to match the goal.

Integration didn't start due to sub-component failures.

Chapter 1： Group description

Group number :14 Signed implementation

Group composition

Data Processor (Team A）：

Ziyang Deng ts20814 (Team leader)
Oliver Bath rm21809

Command Processor (Team B）：

Tia Ma ki19146 (Team leader)
Yunji Nam oq20050
Qian Chen bh21669

Task division（who does what）：

TEAM A:

Ziyang Deng: Made the block-level sketch of the main logic components for the data processor. Designed and wrote code for the data array and control path in the data processor. Communicating between Team A and Team B. Made timeline for the whole group for week16-21 in TB2.

Oliver Bath: Designed and wrote code for the data path in the data processor. Made group timelines (Gantt chart) for the whole group for the entire project time.

Common: Simulation and synthesis experience. FSM ideas generated by both team members. We compared and discussed our FSMs and combined them into the final single FSM for the module design. Summary of team's work in report. Both checking simulations results and experiencing synthesis and implementation. Built up the structure and framework for the report. Records milestones for Team A.

TEAM B:

Tia Ma: 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.

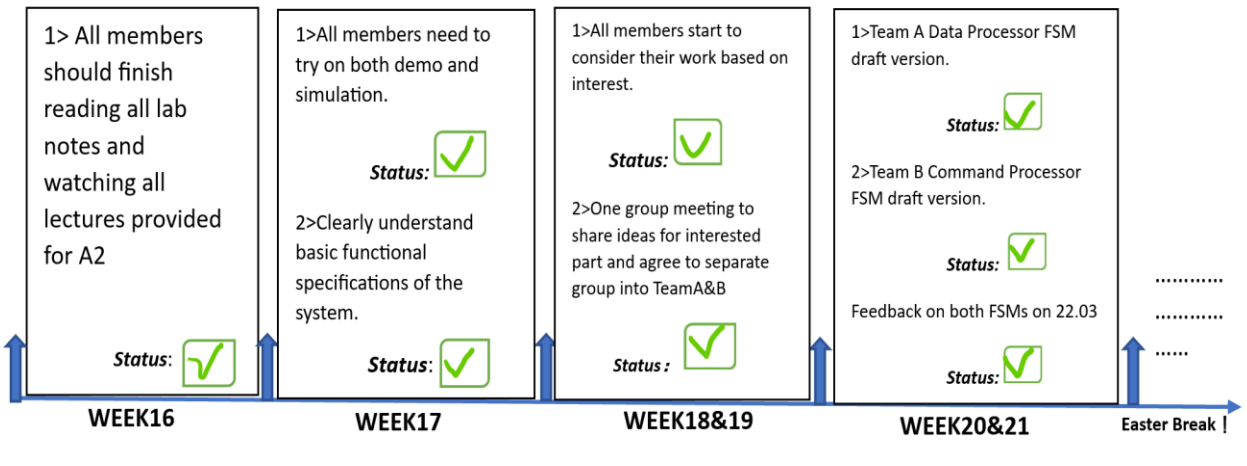
Qian Chen: Worked on the command processor. He completed part of the task, provided a detailed report that included the FSM chart and functional block explanations. Qian Chen and Ziyang Deng tested the code, which was approved to meet the goal.

Common： FSM ideas generated and finished by Tia Ma, Yunji Nam and Qian Chen. We compared and discussed our daft FSM by Tia Ma and new draft FSM by Yunji Nam and Qian Chen. 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.

Chapter 2：

Project Plan (shows the timeline for task, deliverables and milestones, as well as group meetings)

Timeline（Week16-21 TB2）



Milestones:

green for data processor, orange for command processor

3.20-- Completed draft version of data processor in terms of FSM and Block-level sketch of main logic components. (Ziyang Deng & Oliver Bath)

3.20-- Completed draft version of command processor in terms of FSM. (Tia Ma)

3.21-- Completed draft version of VHDL code for data array ， data path and control path in data processor. (Ziyang Deng & Oliver Bath)

3.22-- First simulation for data processor. The signals 'wave' is close to the goal compared with the demo simulation. (Ziyang Deng & Oliver Bath)

3.23-- First synthesis and implementation of data processor (showing the results of imperfect functionality in Putty.) (Ziyang Deng & Oliver Bath)

3.27-- Redraw the draft version of command processor in terms of FSM. (Yunji Nam &Tia Ma)

3.31-- Completed the draft version of ANNN processing part in command processor and write the draft of ANNN processing part. (Tia Ma)

4.1-- Completed the draft version of Decoder (ASCII) part in command processor and wrote the draft report of Decoder (ASCII) part and LP command part. (Tia Ma)

4.6-- Completed draft version of VHDL code for command processor, however it's un-synthesizable and waiting for further optimization. (Yunji Nam & Qian Chen)

4.10-- Synthesize the draft version of VHDL code for command processor of Qian Chen's version, however it's un-synthesizable and waiting for further optimization. Qian Chen's version has been proved could run properly by transferring old 2008 VHDL data type to VHDL data type. (Ziyang Deng & Qian Chen)

4.20-- Optimized version of synthesizable data processor code came out, showing full functionality and bug-free results in Putty. (Ziyang Deng)

4.21-- Final check of data processor and it is **ready for integration,** waiting for Team B's command processor's optimization. (Ziyang Deng)

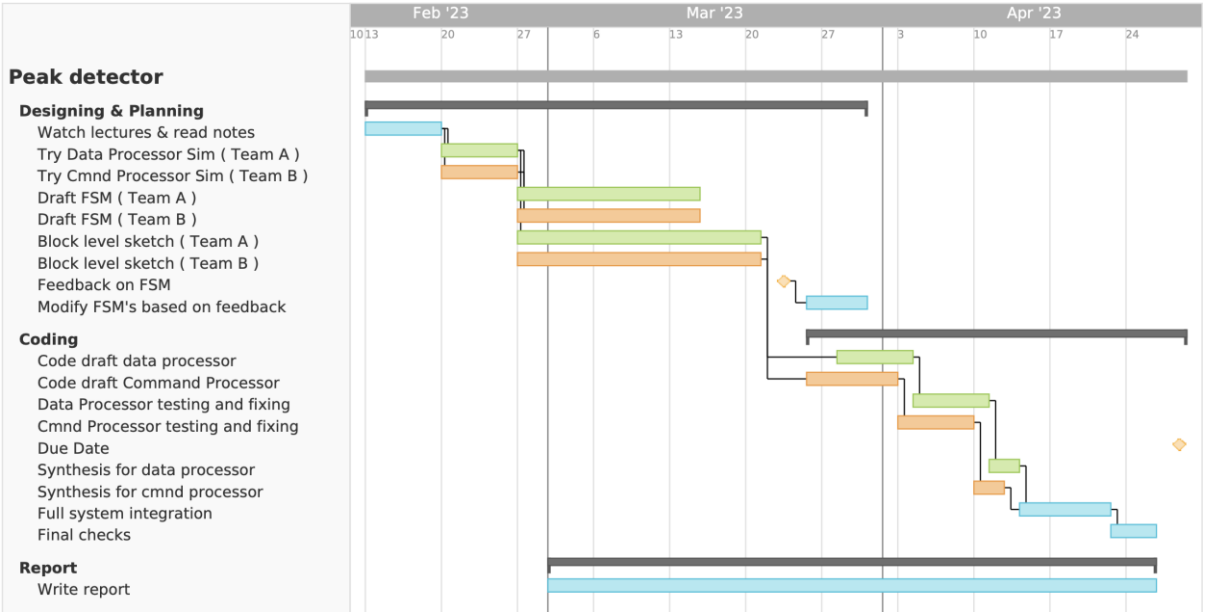
4.23-- First synthesis and implementation of command processor with incorrect results on Putty. (Yunji Nam & Tia Ma)

4.24-- The Gantt chart shows the project's lifetime completed. (Oliver Bath)

Group meeting arrangements:

1. Monday and Wednesday 's lab in TB2.
2. 10/03/23 10:00 – 10:30 a.m. in person meeting in MVB (Merchant Venturers Building) for separating works and team division.
3. 07/04/23 14:00 – 14:30 p.m. Teams online meeting during Easter to check each teams' progress.
4. Monday and Wednesday 's lab in TB4.
5. 26/04/23 13:00 – 14:30 p.m. in person meeting in group room Beacon House University of Bristol to finish report.

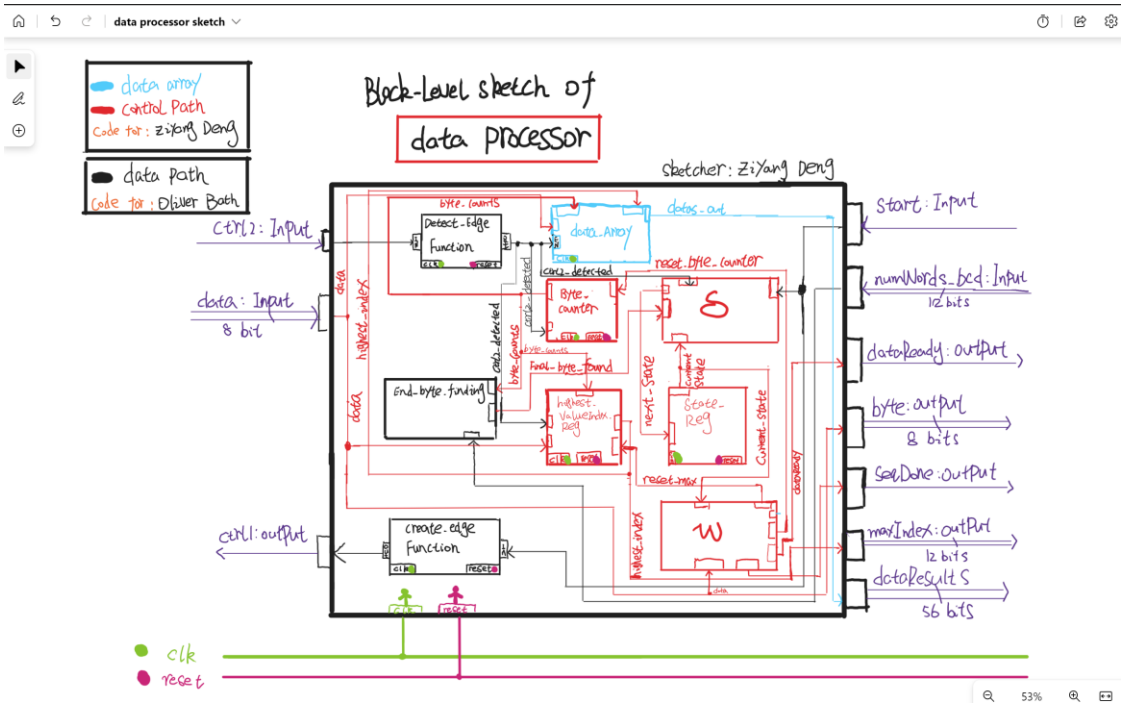
Gantt chart:



Here is a Gantt chart created of predicted timeline of project. Tasks that require the attention of the whole group are shown in blue, team A shown in green, and team B shown in orange.

Chapter 3：

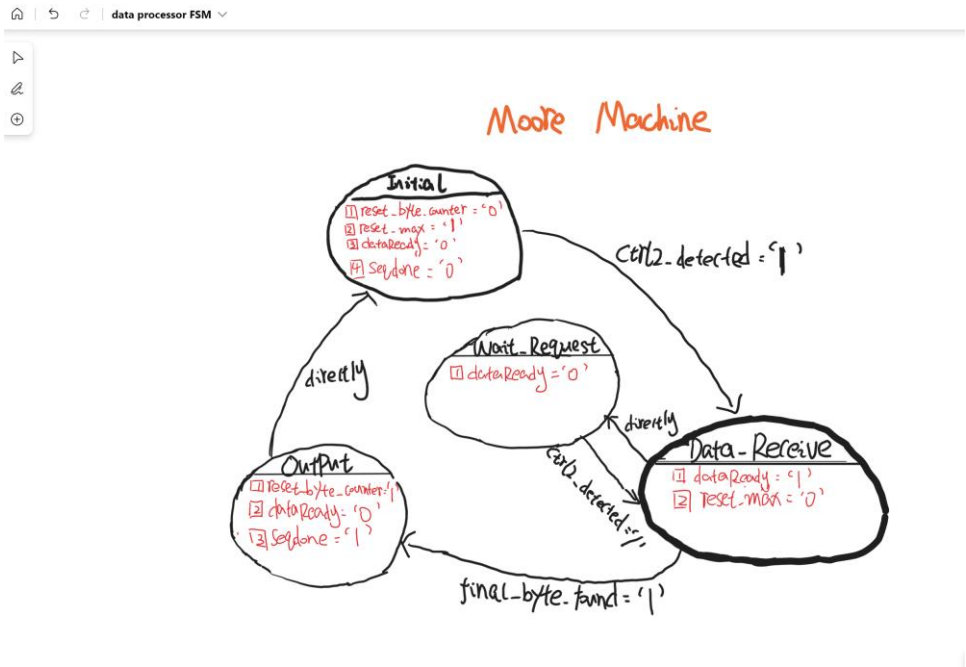
Team A Data Processor (Description of the architecture of the Data Processing module including a block-level sketch of the main logic components and a state diagram for the FSM) Ziyang Deng & Oliver Bath



[Figure 1] Block-level components sketch (Ziyang Deng)

- The data processor consists of three main modules: **data path**, **control path** and **data array**.

Data path is sketched in black color above (**Figure 1**). Detect_edge function is a process used to detect output signal transition from data generator, it contains a register to store previous ctrl2 input signal, through XOR this previous signal to the current same signal, once an event occurs on ctrl2 signal, it will output the ctrl2_detected signal to high for one clock to inform that a new byte has been supplied by the data generator. The ctrl2_detected signal is widely used within the data processor as well, it can be used as condition signal to update data array, byte counter, highest value and index register and nextState logic（delta) within the control path. In addition, there is a similar process named create_edge function to create an event on output signal ctrl1 when the start signal is high. Furthermore, the end_byte_found process is used to set its output signal final_byte_found to high to inform that the required number of bytes have been processed. ---- Oliver Bath



[Figure 2] Finite-State-Machine Moore model

Control path（FSM controlled path） is highlighted in red in the block-level components sketch (**Figure 1**). The byte counter is used to count the numbers of byte that have been processed. It has been updated every time the ctrl2_detected signal is getting high. There are two usages of its output signal byte_counts. Firstly, it can be used to check whether the final byte has been supplied by the data generator. In addition, it's used for updating the current highest value's index, as every time we find a new highest value, its index comes along with the number of bytes countered. The second component is highest_valueIndex_register, which stores the current highest data byte and its index. For signed implementation of our group, the highest data byte is reset to ''10000000' which is –128(the min value for signed 8-bit binary number) instead of ''00000000'. It is output signal highest_index is not only used for connecting to the maxIndex output port to command processor but could be used as the middle index to the data array for resulting the three bytes before and after and with the highest value on the output dataline dataResults to the command processor. The remaining components are used for

Finite State Machines (**Figure 2)**. In our design, we applied Moore model which outputs only based on the current states. We can observe 4 states used to achieve the desired functionality. When in the initial state, the system initializes relevant signals to low and set the reset_max signal to high. Once the system detects the high ctrl2_detected signal, which means the first byte has been supplied, it transfers to data_receive state during the next clock. In this state, it will set dataReady signal to high for only one clock to inform command processor that the new data has been supplied, also it will set the reset_max signal to low and now it's ready for that module to work. One clock after jumping to this state, it will then go to wait-request state without any condition, and during this state it will reset the dataReady signal to low, then waiting for the next high of ctrl2_detected to jump back to data-receive state. If the final_byte_found signal is high during the data-receive state, which means that all required bytes have been supplied, it will transfer to output state instead of wait-request state. Finally, in the output state, it will event seqDone to high to inform command processor that all the bytes are processed, and the

required data sequence is ready on the dataResults output line for printing. Despite that, it will reset the dataReady signal that is high in the previous state to '0' to avoid ambiguity. In addition, it will set the reset_byte_counter signal to high in order to set relevant values into initial during the next clock which is the initial state. After one clock in this state, it returns to Initial state directly and waits for the next task. -------Ziyang Deng

Data array （sketched in blue） is used to store bytes supplied by the data generator, which can be used to provide the required format of data sequence later. This module is described in another file called dataArray.vhd. It's initializing with 0s in each element at start, and checking the input signal ctrl2_detected connecting to its port enable_write in every rising edge clock, if the signal is high then it means the data generator has provided a new byte and at the same time the data array can store this new byte in the position depends on the input signal byte_counts derived from the byte counter to the byte_counts input port of data array.

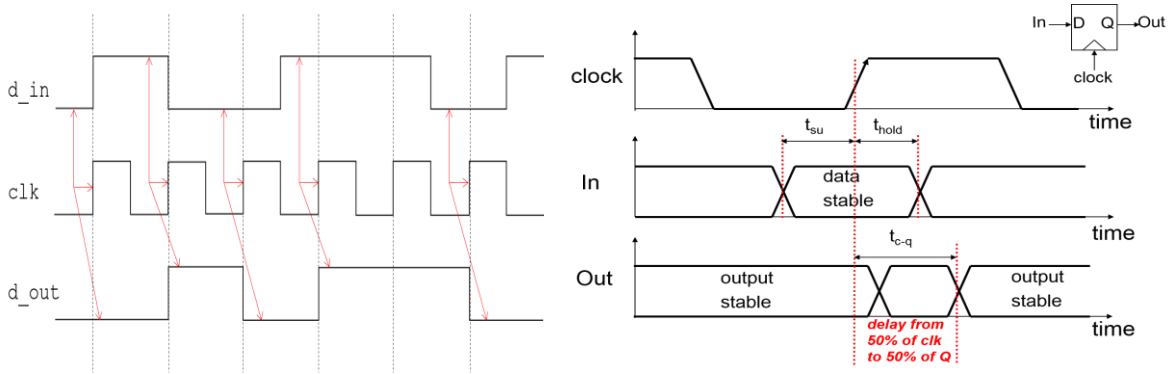


Figure 3 Timing waveform for registers （ **[1]** Dr.Hassani & Dr. Yan, **p.6.1 A1**）

Based on the 'Principles of Timing in Synchronous Systems '(**Figure 3**), as both the byte counter and data array are updated and synchronized with the clock and ctrl2_detected signal, thus in our VHDL code the index for the array doesn't need to be the number of bytes minus 1(**Figure 4**), because the index used for updating the data array is the previous clock period's byte counts number, which is identical to the index for the next element in the array. (e.g., at the first updating of array stage, when the clock is high, it will take the previous value of byte counter, which is 0, as index and it is used to update the first element of the array).

```
36  byte_array:process(clk)
37  variable index:integer ;
38    begin
39    index := byte_counts  ;
40
41     if rising_edge(clk) then
42         if enable_write = '1' then
43                 if index < 999 then
44                    data_array(index) <= data_in ;
45                 end if ;
46         end if ;
47     end if ;
48     end process ;
```

This bit is not byte_counts - 1 instead.

Figure 4 VHDL code portion for data array

Another thing worth mentioning is: The data array was designed to have a reset signal come out from FSM in the initial state, but due to our observation that it will then be time-consuming for both simulation and synthesis. We guess the reason for that is every time we come back to the initial state, the system will enter a loop to iterate through every element in the array (999 elements！) to reset each one to 0s. Thus, it was going to be time consuming, and we deleted that feature in our final design. "If the peak index is such that there aren't enough bytes to populate 3 bytes either side of the peak, only the bytes corresponding to those valid indices are updated."([2]Dr.Hassani & Dr. Yan, p2.4 A2) In this case, it would be unnecessary to reset the array back to 0s, as if we know the boundary of index for the data sequence, we would then know where the data become meaningless, thus the data out of index range though printed, is to be disregarded as there isn't an index after that.

Finally, the data array will output 7 bytes in an ordered sequence depending on the case of highest_index value. **(Figure 5）.** For the case where highest_index might cause corresponding indexes which are out of array's range, it going to just set the data on those ambiguous indexes to 0s. -----Ziyang Deng

```
outputData:process(highest_index)
begin
            case highest_index is
            WHEN 0 =>
                datas_out(6) <= (others => '0')                  ;
                   datas_out(5) <= (others => '0') ;
                   datas_out(4) <= (others => '0') ;
                   datas_out(3) <= data_array(highest_index)   ; --
                   datas_out(2) <= data_array(highest_index+1) ;
                   datas_out(1) <= data_array(highest_index+2) ;
                   datas_out(0) <= data_array(highest_index+3) ;

            WHEN 1 =>
                    datas_out(6) <= (others => '0')      ;
                   datas_out(5) <= (others => '0') ;
                   datas_out(4) <= data_array(highest_index-1) ;
                   datas_out(3) <= data_array(highest_index)   ; --
                   datas_out(2) <= data_array(highest_index+1) ;
                   datas_out(1) <= data_array(highest_index+2) ;
                   datas_out(0) <= data_array(highest_index+3) ;
```

Figure 5 Data Array portion outputting process

[**Hint**: In the block elements sketch (**Figure 1**), the clock signal connected is abstracted in green dot, the reset signal connected is abstracted in purple dot]

Chapter 4：

Team B Command Processor (Description of the architecture of the command processing module including a block-level sketch of the main logic components and a state diagram for the FSM) Tia Ma & Yunji Nam & Qian Chen

1. Receive 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_inact’ 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_inact’ state to look for the second digit. If the received data is an 'A' or 'a', then transitions back to the ‘valid_A_inact’ 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_inact’ state, to start a new process.

- NNN: The code of detecting NNN starts from the ‘valid_1_inact’ 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_inact’ state to look for the third digit. If it's 'A' or 'a', then transitions back to the ‘valid_A_inact’ 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_inact’ state.

In the ‘valid_2_inact’ 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 and Qian Chen)

In this implementation, the putty part is an important part for command processor. Putty states have been in the whole system 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 equal's 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 equal's 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]:

Hex	Binary	Hex	Binary
0	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011
4	0100	C	1100
5	0101	D	1101
6	0110	E	1110
7	0111	F	1111

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_inact' 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 detail, 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]:

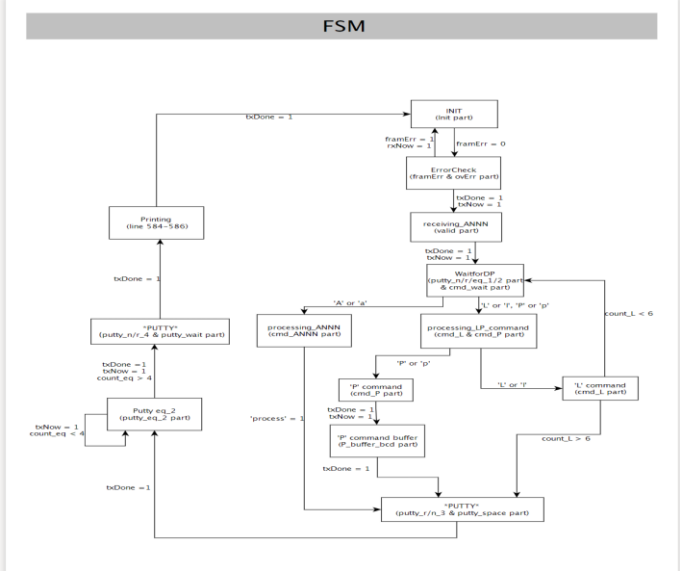
ASCII Code: Character to Binary

0	0011 0000	O	0100 1111	m	0110 1101
1	0011 0001	P	0101 0000	n	0110 1110
2	0011 0010	Q	0101 0001	o	0110 1111
3	0011 0011	R	0101 0010	p	0111 0000
4	0011 0100	S	0101 0011	q	0111 0001
5	0011 0101	T	0101 0100	r	0111 0010
6	0011 0110	U	0101 0101	s	0111 0011
7	0011 0111	V	0101 0110	t	0111 0100
8	0011 1000	W	0101 0111	u	0111 0101
9	0011 1001	X	0101 1000	v	0111 0110
A	0100 0001	Y	0101 1001	w	0111 0111
B	0100 0010	Z	0101 1010	x	0111 1000
C	0100 0011	a	0110 0001	y	0111 1001
D	0100 0100	b	0110 0010	z	0111 1010
E	0100 0101	c	0110 0011	.	0010 1110
F	0100 0110	d	0110 0100	,	0010 0111
G	0100 0111	e	0110 0101	(	0011 1010
H	0100 1000	f	0110 0110	;	0011 1011
I	0100 1001	g	0110 0111	?	0011 1111
J	0100 1010	h	0110 1000	:	0010 0001
K	0100 1011	I	0110 1001	'	0010 1100
L	0100 1100	j	0110 1010	"	0010 0010
M	0100 1101	k	0110 1011	(	0010 1000
N	0100 1110	l	0110 1100	)	0010 1001
				space	0010 0000

Figure 1: ASCII Conversion Table

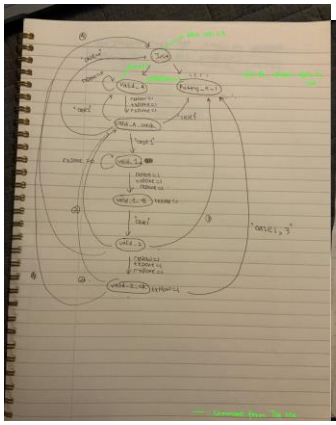
Moreover, for ‘P’ command, to design a communication protocol between two devices, where data is transmitted using UART (Universal Asynchronous Receiver Transmitter) protocol. A state machine for BCD has been created (lines 383-476), which can be seen in the draft state machine drawn by Yunji Nam. This 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.

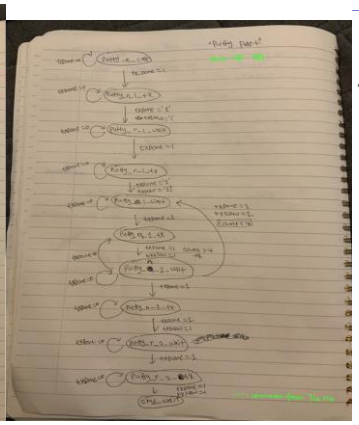
Command processor overall FSM:

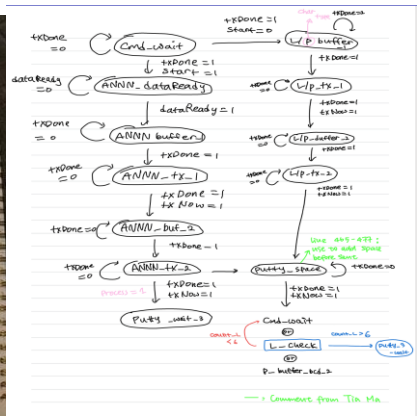


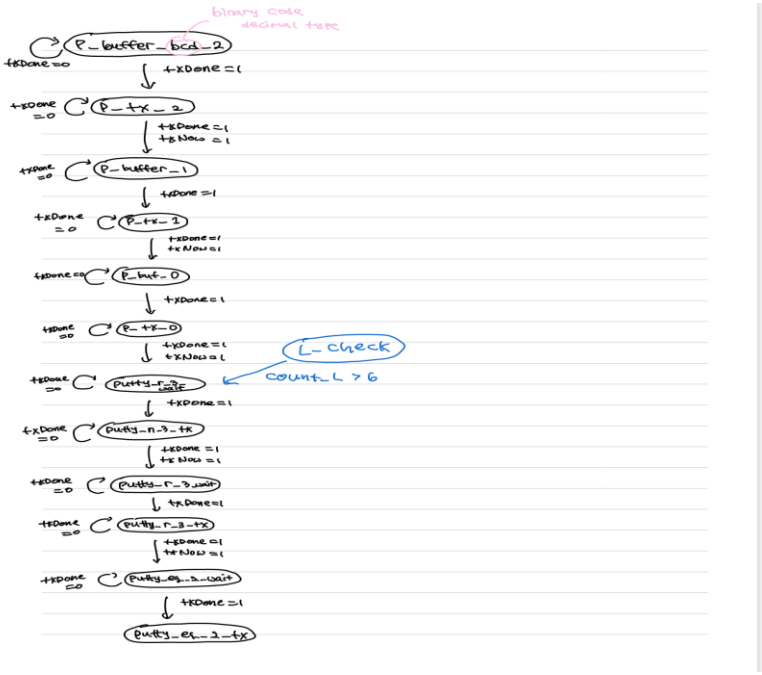
:

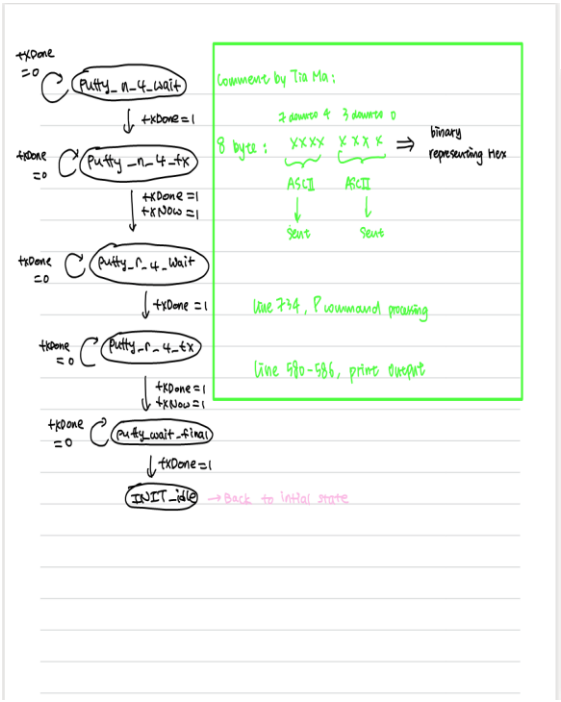
Command processor detailed FSM BY HAND:（6 PAGES IN TOTAL）

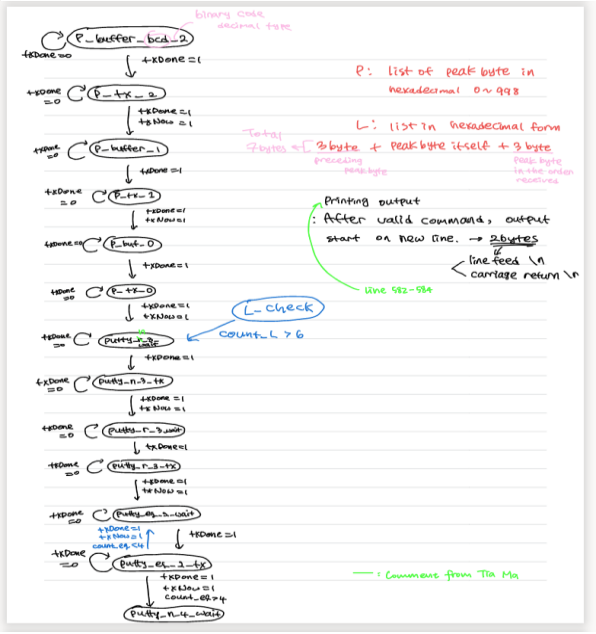







binary code
decimal type
P_buffer_bcd_2
txDone=0
txDone=1
P_tx_2
txDone=1
txNow=1
P_buffer_1
txDone=1
P_tx_1
txDone=1
txNow=1
P_buf_0
txDone=1
P_tx_0
txDone=1
txNow=1
L_check
count_L > 6
putty_n_3_wait
txDone=1
putty_n_3_tx
txDone=1
txNow=1
putty_n_3_wait
txDone=1
putty_n_3_tx
txDone=1
txNow=1
putty_eq_3_wait
txDone=1
putty_eq_1_tx

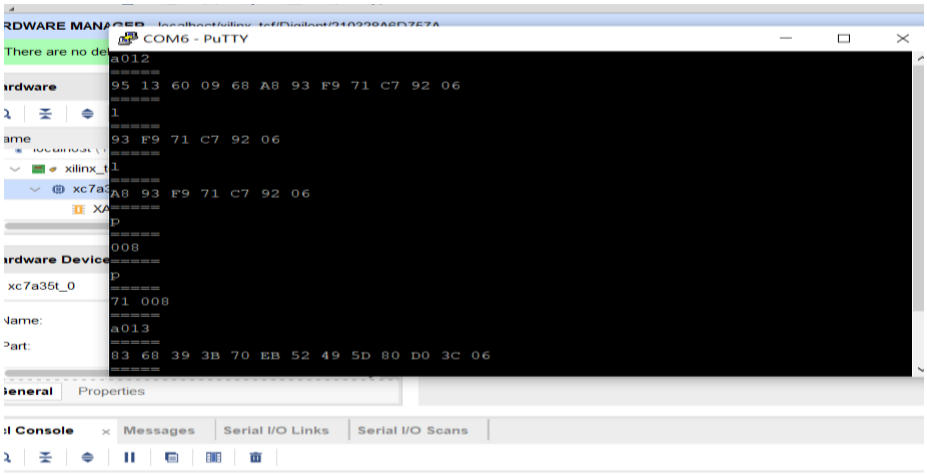

txDone=0
putty_n_4_wait
txDone=1
putty_n_4_tx
txDone=1
txNow=1
putty_n_4_wait
txDone=1
putty_n_4_tx
txDone=1
txNow=1
putty_wait_final
txDone=1
INIT_idle
→Back to initial state
Comment by Tia Ma:
2 source 4 3 source 0
8 byte: XXXX XXXX ⇒ binary representing Hex
ASCII
ASCII
Sent
Sent
line 734, P command proceing
line 580-586, print onepet


binary code
decimal type
P_buffer_bcd_2
P: list of peak byte in hexadecimal 0~998
L: list in hexadecimal form
Total 7 bytes = 3 byte + peak byte itself + 3 byte
preceding peak byte
Peak byte in the order received
Printing output
: After valid command, output start on new line. → 2 bytes
line feed \n
carriage return \r
line 582-584
L_check
count_L > 6
: Comments from Tia Ma

Chapter 5：

Integration and Systems Evaluation:

Data processor individual system evaluation:



(Figure 1 Putty results for data processor)

(Figure 2 Simulation results for data processor)

Team A successfully run both simulation and synthesis process for data processor. In figure 1, which shows the results from PC keyboard command ‘a012’,‘l’ and ‘p’ in Putty. It prints corresponding results with 12 values which are identical to the first twelve data in test_bench and the first twelve data from data generator’s data array, the correct sequence with ‘71’ in the middle, and the highest byte with its index from the sequence, which is ‘71 008’. Figure 2 shows the simulation results for when numWords_bcd signal is ‘0,1,2’, this result is highly similar to the demo simulation results with codes provided for both processors.

We did a slightly different thing compared with the demo version code provided. Firstly, we didn’t reset the dataResults to 0s in some particular period as we observed that it will be time consuming for our version code to reset that（see chapter 3 for data array）. Secondly, we set the data which sit in the impossible （meaningless） index to 0s in the dataResults sequence, instead of leaving it to some value like what the demo version code did. This might be better to not confuse the audience from our opinions.

Command processor individual system evaluation:

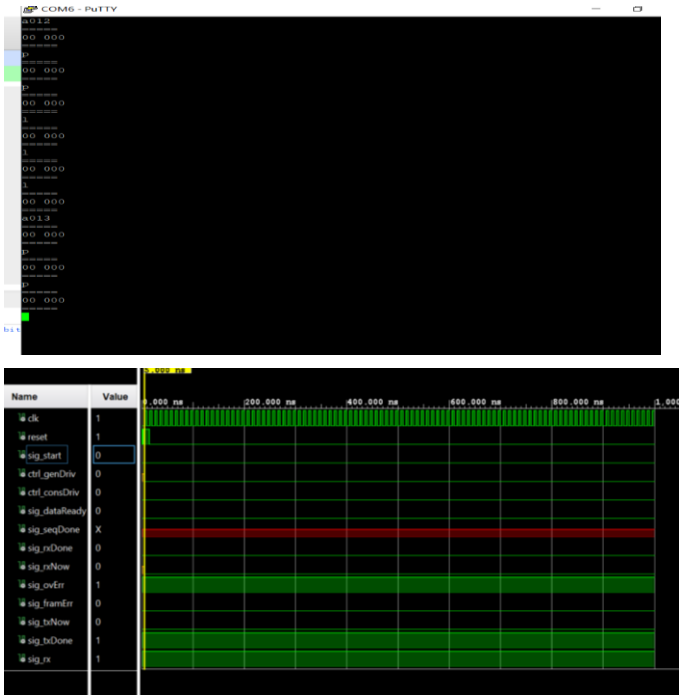


Figure 3 Command processor simulation and Putty results.

Team B can run both simulation and synthesis without error. Unfortunately, it didn't provide correct functionality. Shown in figure 3, the Putty can accept command from keyboard in the required format i.e.'a' or 'A' plus three digital numbers. It does achieve the partial functionality for the communication between the PC and command processor for requesting process. However, it seems not achieve the functionality for communicating between command processor and data processor（provided version）. Observed from figure 3, the simulation shows red error wave for seqdone signal, as well as the all-time low dataReady siganl output from data processor, which made command processor fail to store any byte transfered from data processor. That's the key reason we think that we failed the functionality for communication between data processor and command processor. We will try to fix it later!

[Hint: We used provided code for data processor for doing this simulation and synthesis. Not Team A 's one]

Integration system (Peak detector) evaluation:

Unfortunately, the command processor for Team B fails the individual stand-alone synthesis test on A7 Board, with incorrect functionality and outputs. Thus, the process for integrating both team's systems didn't start.

Peer Assessments:

Observer	Peer Comments (summary)
Ziyang Deng	**Ziyang** has good communication between both team A and team B. Also, very good leadership for team A to progress. Excellent contribution to FSM, codes, and reports. Fantastic contribution to block-level architecture sketch. Fully engagement throughout the whole teaching block.
Tia Ma	**Tia** has very good communication between team A and team B, very good leadership to Team B, very good engagement throughout the teaching block, and very good contribution to code. Fantastic contribution to command processor's FSM.
Oliver Bath	**Oliver** has very good communication within the team, very good contribution to the FSM and codes, excellent contribution to the reports, excellent engagement throughout the teaching block.
Yunji Nam	**Yunji** has good contribution to the command processor's FSM, very good communication between both team A and team B, excellent engagement throughout the teaching block, excellent contribution to the code.
Qian Chen	**Qian** has good contribution to the command processor's FSM, good communication within the team, good engagement throughout the teaching block, excellent contribution to the code.

Group Member	1	2	3	4	5	Total / spec (12.5 in total)
Name Username (e.g., ab12345)	Ziyang Deng ts20814	Oliver Bath rm21809	Tia Ma ki19146	Yunji Nam oq20050	Qian Chen bh21669	
Leadership	2.5	2.5	2.5	2.5	2.5	12.5
Team Engagement	2.5	2.5	2.5	2.5	2.5	12.5
Carrying out technical work	2.5	2.5	2.5	2.5	2.5	12.5
Contributing to the report	2.5	2.5	2.5	2.5	2.5	12.5
Total / member	10	10	10	10	10	50

Reference:

[1] Dr.Hassani & Dr. Yan. （2022）Assignment1 Digital Synchronous System Design, Digital Design, Group Project EENG28010, University of Bristol. Available at: https://seis.bristol.ac.uk/~sy13201/digital_design/ECAD/ (Accessed: April 25,2023）

[2] Dr.Hassani & Dr. Yan. (2022) Assignment2 Functional specification, Digital Design, Group Project EENG28010, University of Bristol. Available at: https://seis.bristol.ac.uk/~sy13201/digital_design/ECAD/ (Accessed: April 25,2023）

[3] Louis Azzopardi （ 2008 ） Computing & ICT Toolbox Available at: https://sites.google.com/site/computingicttoolbox/Home/form-3/ascii-and-ebcdic .

(Accessed: April 26, 2023)

[4] Copyright study-ccna.com (2023), IPv6 Address Format, Available at: https://study-ccna.com/ipv6-address-format/ (Accessed: April 26, 2023)