Digital Design

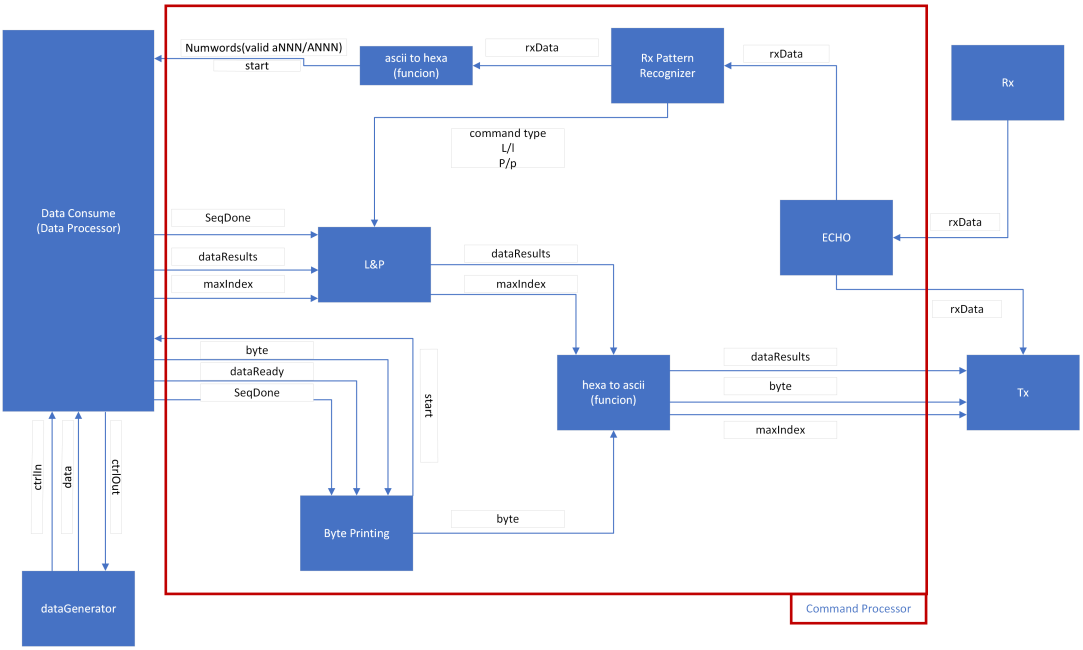
EENG28010

Jiaying Shen, Yumu Xie, Yiyao Wang, Jiaxin Fan.

Contents

1 Introduction

This project implements a system which is capable of data generation, peak detection and command implementation with host computer. The system identifies a start command $aNNN$ or $ANNN$, where NNN are three digits numbers specifying the number of bytes being generated. Then the system will output hexadecimal bytes and produce value of peak byte, its index and list the six bytes which are surrounding with the peak byte in hexadecimal format according to the user's command like L or P. Finally, these results will be displayed on terminal.



2 Task Division

This project is divided into two parts, Team A Command Processor and Team B Data Processor.

2.1 Team A Command Processor

- Yiyao Wang: Rx pattern recognizer and ECHO
- Yumu Xie: L, P and Byte printing

2.2 Team B Data Processor

- Jiaying Shen: Two-phase control
- Jiaying Shen, Jiaxin Fan: Data consume

3 Design and Simulation of Command Processor

3.1 Command Processor Overall Design

3.1.1 Rx Pattern Recognizer and ECHO-Yiyao Wang

Pattern recognizer is the component designed to receive the input, then detect input's pattern including $a/Annn$ and L/P pattern, registered the nnn value consecutively and finally pass $numWords$ array to data processor. It also works with echoing at the same time, passing all commands to echoing components. Data echoing component is responsible for printing out all commands including both valid and invalid commands to the terminal. It also prints out the byte from data processor with ASCII conversion once data echoing received *dataReady* and *byte* from data processor. To some extent, like $hexa\_to\_ascii$ function, it coordinates with L and P and help these components to print out on terminal with ASCII conversion.

$a/Annn$ recognition: When the pattern of a/A followed by three consecutive numbers is recognized, it is converted from ASCII into binary-coded-decimal(BCD) format through a function called $ascii\_to\_hexa$ and then registered in $reg\_numWords$ and finally passed the three bytes register to the $numWords$ array. It is worth to mention that $a000$ / $A000$ will be isolated and does not interact with dataConsume to avoid unexpected error.

L/P recognition:When L/l or P/p is recognized, the process will jump to a state L_LF or P_LF to prepare the L or P printing.

ECHO: when component RX receives command inputs from host computer, ECHO will directly coordinate with component TX and echo all command inputs to the terminal in order to display it.

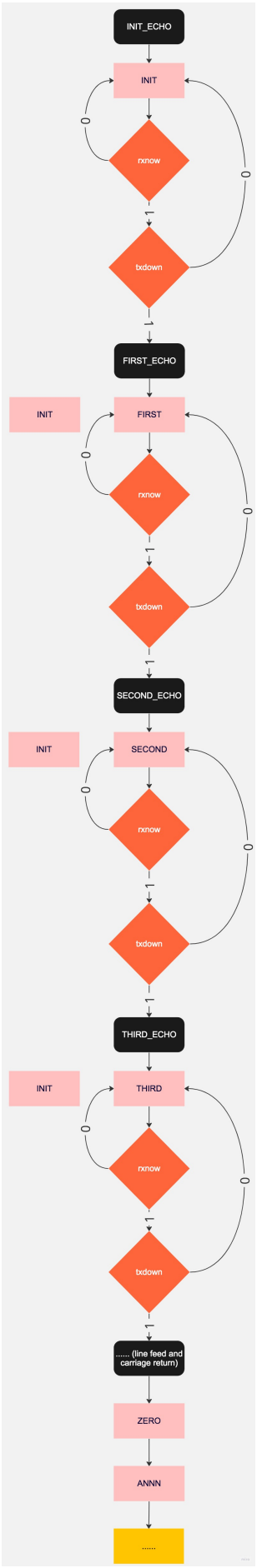


Fig. 1 Annn and Echo

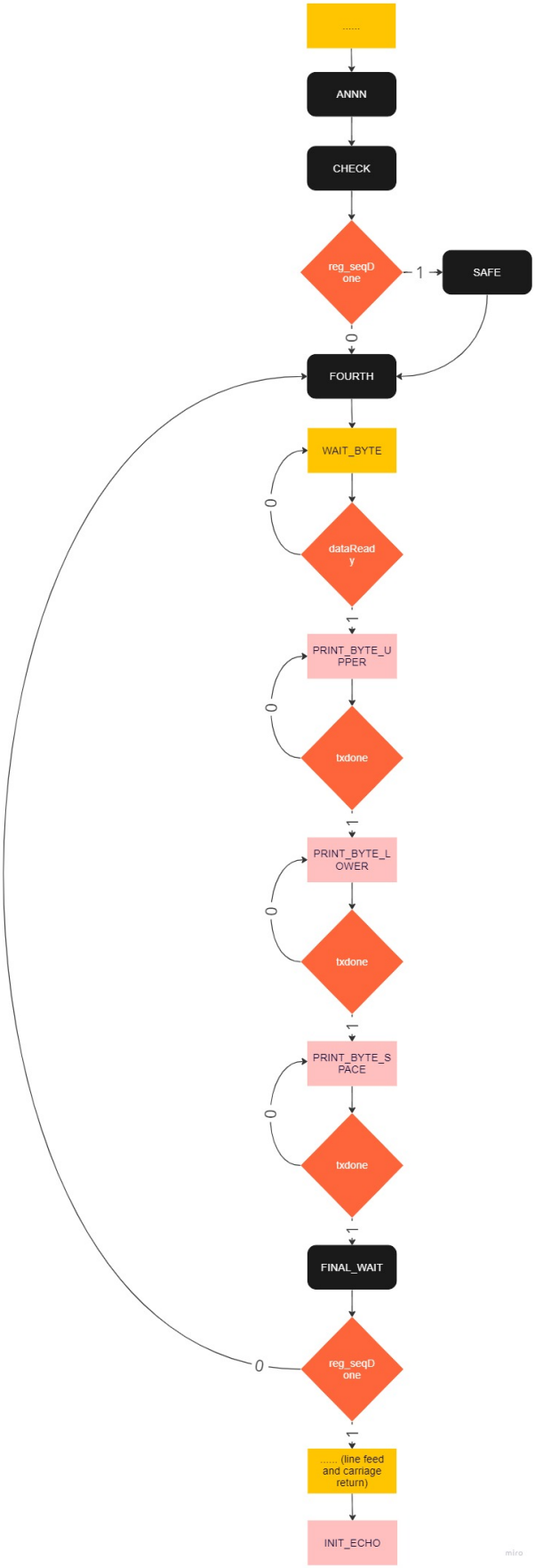


Fig. 2 byte printing

3.1.2 Byte Printing-Yumu Xie

Byte printing: when a valid $numWords$ (except $a000$ / $A000$, these commands have been isolated in $ZERO$ state) has been transferred from Rx Pattern recognizer in $ANNN$ state, the state will immediately get into $CHECK$ state. In $CHECK$ state, $reg\_seqDone$ will be checked because we found out that if we program the chip, which means chip has been initialised. An automatic $seqDone$ will be sent from dataConsume to cmdProc. So, after cmdProc sent valid $numWords$ to dataConsume, cmdProc must clear $reg\_seqDone$ before it starts to grab byte from dataConsume as $reg\_seqDone$ would register $seqDone$ when $seqDone$ is 1 until justification state $FINAL\_WAIT$. If we do not clear $reg\_seqDone$ after $ANNN$ state (send all $numWords$ to dataConsume) before $FOURTH$ state (give $start$ signal to dataConsume), the output would be very strange and it only outputs one byte then stop immediately. After $CHECK$ and $SAFE$ states (these are used for checking $reg\_seqDone$ and clear $reg\_seqDone$ to 0), in the next two states, cmdProc will set $start$ to 1 for one system clock cycle and wait for $dataReady$ in the next state. After that, cmdProc enters states to print out byte, which these states utilise a function called $hexa\_to\_ascii$. This direct printing will cause latch since it links the input and output together but we do not have any good idea to improve it now. Every time, after cmdProc outputs one byte and space, cmdProc will enter $FINAL\_WAIT$ state for checking $reg\_seqDone$. If $reg\_seqDone$ is 1 ($seqDone$ will automatically register into $reg\_seqDone$ in $regLogic$ process in every rising edge clock if $seqDone$ is 1), in $FINAL\_WAIT$ state, cmdProc will assign next state to LF (line feed state with a carriage return state behind) and back to all initial state ($INIT\_ECHO$). Otherwise, it will get into $FOURTH$ state and keep grabbing byte from dataConsume.

```
when PRINT_BYTE_UPPER => -- byte_upper
if txdone = '1' then
    txnow <= '1';
    txData <= hexa_to_ascii(byte(7 downto 4)); -- upper
    nextState <= PRINT_BYTE_LOWER;
else -- txdone = '0'
    txnow <= '0';
    nextState <= PRINT_BYTE_UPPER;
end if;

when PRINT_BYTE_LOWER => -- byte_lower
if txdone = '1' then
    txnow <= '1';
    txData <= hexa_to_ascii(byte(3 downto 0)); -- lower
    nextState <= PRINT_BYTE_SPACE;
else -- txdone = '0'
    txnow <= '0';
    nextState <= PRINT_BYTE_LOWER;
end if;

when PRINT_BYTE_SPACE => -- byte_space
if txdone = '1' then
    txnow <= '1';
    txData <= "00100000"; -- space character
    nextState <= FINAL_WAIT;
else -- txdone = '0'
    txnow <= '0';
    nextState <= PRINT_BYTE_SPACE;
end if;
```

3.1.3 L and P Printing-Yumu Xie

L and P Printing: in this part, it can be split into two parts, L Printing and P Printing. There are two initialisation signals *ini_maxIndex* and *ini_dataResults*. They are utilised when I press the *reset* button in chip, all L and P results should be clear to initialisation state.

L Printing: Multiple global signals are created for better implementation of functions: *resultL*, *ascii_result*, *reg_maxIndex* and *reg_dataResults*. In *regLogic* process, *seqDone* will be registered into *reg_seqDone* if *seqDone* is 1. Besides, *dataResults* are only available when *seqDone* is 1, and *seqDone* only lasts 1 for one system clock cycle. *seqDone* will be detected in *regLogic* process and register *maxIndex* and *dataResults* into *reg_maxIndex* and *reg_dataResults* for further implementations. After Rx pattern recognizer recognizes L command, the next state will be L_LF with a L_CR (line feed and carriage return) before getting into L state. In L state, *reg_seqDone* will be loaded into *resultL* and *lLogic* will detect the change of *resultL* and execute transfer operations by using *hexa_to_ascii* function. During *nextStateLogic* (the state machine which proceeds states), L Printing has many states to be used for printing out the seven bytes of L command. They are generally can be called L_UPPER, L_LOWER and L_SPACE. After all seven bytes have been printed out, it gets into LF state (line feed state with a carriage return state behind before getting back to $INIT\_ECHO$ initial state). In L Printing logic, a process called *lLogic* is also used for transferring type from hexadecimal to ascii for the sake of printing. In this process, every time when rising edge clock, an embedded for loop will be used in order to transfer hexadecimal bytes stored in *resultL* to ascii bytes into *ascii_result*. Then, once cmdProc enters states to print out seven bytes for L command, *ascii_result* will be utilised and printed out.

P Printing: there are also global signals initialised for better implementation of functions: *peakP*, *indexP*, *reg_maxIndex* and *reg_dataResults*. Just like what I mentioned, *reg_maxIndex* and *reg_dataResults* are used to register *maxIndex* and *dataResults* when *seqDone* is 1. Once Rx pattern recognizer recognizes P command, the next state would be P_LF and P_CR with a P state behind. In P state, *peakP* will load *reg_seqDone*(3) and *indexP* will load *reg_maxIndex* into this signal. During printing state, $PEAK\_UPPER$, $PEAK\_LOWER$, $PEAK\_SPACE$, $FIRST\_INDEX$, $SECOND\_INDEX$, $THIRD\_INDEX$, the function *hexa_to_ascii* will be used during these printing states for the sake of transferring type from hexadecimal to ascii by implementing *peakP* and *indexP* with this function.

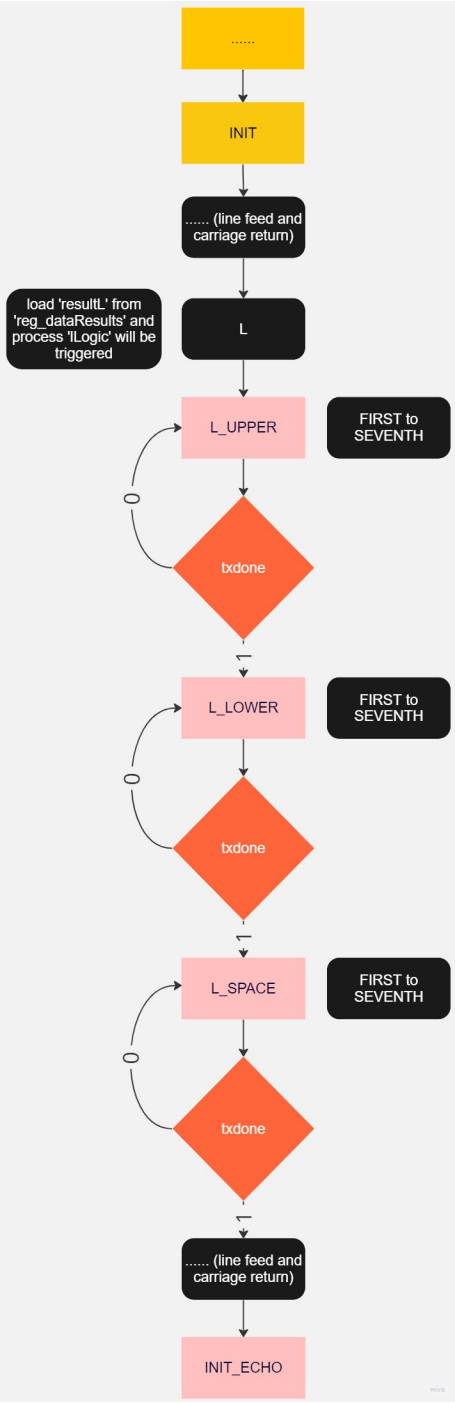


Fig. 3 L printing

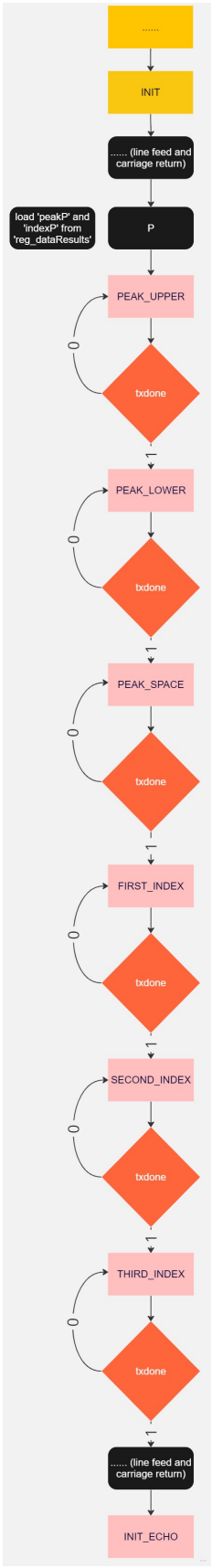


Fig. 4 P printing

4 Design and Simulation of Data Processor

4.1 Data Processor Overall Design

Signal

The signal curState and nextState reflect the FSM's current state and next state respectively. '*counter*' records the number of data that currently received; '*num*' is the data number that should be sent, which was converted by function *bcd_to_int*.; '*peak_index*' is the current peak index.

'window', '*cur_window*', '*max_window*' are arrays of 7characters. '*window*' shift window starts from index 3 to left for one bit; '*cur_window*' shifts window one bit every time; '*max_window*' is current max window.

There are signals '*ctrl_detect*', '*ctrlIn_reg*', '*ctrlOut_reg*'. All of them are used to detect whether the *ctrlIn* has changed and *ctrlIn_reg* is the register of the *ctrlIn*, *ctrlOut_reg* is the register of the ctrlOut. '*start_done*'(which is used to detect start signal), '*final_enable*', '*rest_enable*'(these two are used to trigger process in the 'compare' state and '*final*' state.

```
type state_type is (INIT, s1,dready,compare, final);--five states
signal curState, nextState: state_type;
signal counter, num, peak_index: integer:= 0; ---counter:count current times of data received
----num:number of data need to be send
--peak_index: current PEAK INDEX
signal max_num: std_logic_vector(7 downto 0) := "00000000";
signal window,cur_window,max_window:  CHAR_ARRAY_TYPE(6 downto 0):=(others => (others => '0')); ---shift window start from index 3 to left for one bit
 ---cur_window:current window shift from 0 index to left for one bit
--max_window: current max window
signal ctrl_detect, ctrlIn_reg, ctrlOut_reg, start_done, final_enable, rest_enable: std_logic:= '0'; --detect whether the ctrlIn changed
 ---ctrlin_reg:register ctrlIn
   ----register ctrlOut
```

Process

Firstly, the *star_detect* process is used to detect the start signal. Once it sent by command process, if the '*start*' is 1, assign *start_done* <= 1, and vice versa.

4.1.1 *Two-Phase Control*-Jiaying Shen

Next, the '*alter_ctrlOut*' is triggered by the clk, reset and *rest_enable* signals. '*alter_ctrlOut*' takes charge in '*ctrlOut_reg*' and '*num*'. When either '*reset*' or '*rest_enable*' is 1, '*ctrlOut_reg*' and '*num*' will be reset.

When there is a rising edge and start = 1, 'not *ctrlOut_reg*' assign to '*ctrlOut_reg*'. After the end of process, assign the '*ctrlOut_reg*' to '*ctrlOut*'.

```
alter_ctrlOut: PROCESS(clk, reset, rest_enable)----used to change the ouput of ctrlOut
BEGIN
        IF reset='1'OR rest_enable='1' THEN
                ctrlOut_reg <= '0';
                num <= 0,0,0;
        ELSIF rising_edge(clk) THEN
                IF start = '1' THEN
                     num <= 100*bcd_to_int(numWords_bcd(2))+10*bcd_to_int(numWords_bcd(1)) + bcd_to_int(numWords_bcd(0));
                     ctrlOut_reg <= not ctrlOut_reg;
            END IF;
     END IF;
END PROCESS;
ctrlOut <= ctrlOut_reg;
```

We set a *REG_ctrlIn*' process making '*ctrlIn*' registered to detect change in *ctrlIn* (which is triggered by *ctrlIn_reg*, clk and reset signals), which used to register *ctrlIn*, in order to compare it later. When '*reset*' or *curState* is *INIT*, the process sets '*ctrlIN_reg*' to 0. And the process assigns the value of '*ctrlIn*' to '*ctrlIn_reg*' when clk signal is rising edge. The '*ctrl_detect*' then would be set to the result of the XOR operation between '*ctrlIn_reg*' and '*ctrlIn*', which comparing whether the *ctrlIn* is changed.

```
REG_ctrlIn: PROCESS(ctrlIn_reg, clk, reset) --------used to register ctrlIn, in order to compare it later
BEGIN
IF reset = '1' OR curState = INIT THEN
        ctrlIn_reg <= '0';
    ELSIF rising_edge(clk) THEN
             ctrlIn_reg <= ctrlIn;
    END IF;
END PROCESS;
ctrl_detect <= ctrlIn_reg xor ctrlIn;---comparing whether the ctrlIn is changed
```

4.1.2 Peak Detection-Jiaying Shen, Jiaxin Fan

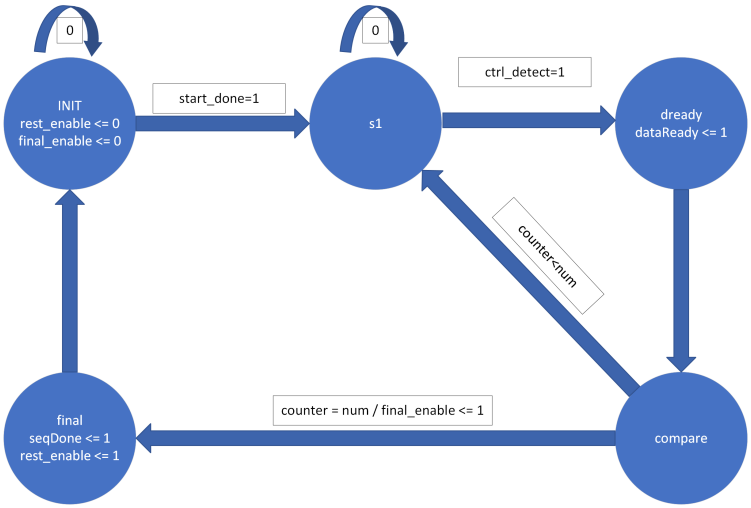


Fig. 5 FSM Chart for State

As I mentioned there are 5 states which are $INIT$, $s1$, $dready$, $compare$ and $final$. These 5 states are all triggered by $curState$, $data$, $start$ and $ctrl\_detect$. The initial state is $INIT$, which is used to initialize all enable signal: when '$star\_done$' signal is 1. We are waiting for $ctrlIn$ change in $s1$ state. The process will go next state '$dready$' unless the detect signal ($ctrldetect$) is 1, or it repeats detecting the change in $s1$ until '$ctrl\_detect$' is 1. The state '$dready$' is in order to keep one clock data ready. Thus, we set '$dataReady$' to 1 and transition to next state ($comparestate$). In the $compare$ state, we will compare the new data and current max data, which is the core of the whole process. In the '$compare$', the next State would back to $s1$ if '$counter$' is smaller than 'num', otherwise go on to next state '$final$ and reset '$final\_enable$' signal to 1 in the mean while. We process output $seqDone$ and assigned value in the '$final$' state. In the '$final$', sets '$seqDone$' and '$rest\_enable$' to 1 and then returns to $INIT$ state and to be ready for the next turn.

'$detect\_ctrlIn$' process is triggered by clk, reset and $rest\_enable$ signals. The purpose of the '$detect\_ctrlIn$' is used to detect whether the $ctrlIn$ changed or not. While it changed, shift the window and make counter +1. In '$detect\_ctrlIn$', when there is a rising edge signal from clk: when '$reset$' or '$reset\_enable$' is 1, the '$counter$', '$window$' and 'cur_window' would be reset; when '$ctrl\_detect$' is 1, indicates that '$ctrlIn$' has changed, then the '$window$' and 'cur_window' will shift to the left by one bit. And the shifted position would be appended by '$data$'.

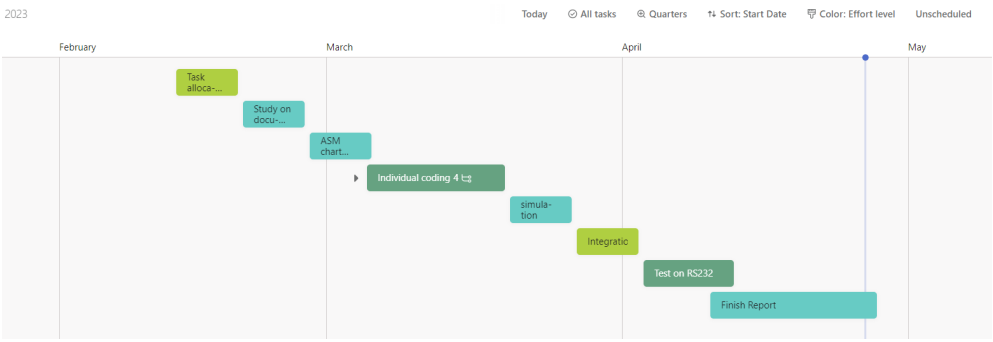
```
detect_ctrlIn: PROCESS(clk, reset, rest_enable) --------used to detect whether the ctrlIn changed or not, if changed, shift the window
BEGIN
IF rising_edge(clk) THEN
        IF reset = '1' OR rest_enable = '1' THEN
              counter <= 0;
              window <= (others => (others => '0'));
          cur_window <= (others => (others => '0'));
        ELSIF ctrl_detect = '1' THEN
              counter <= counter + 1;
          window(6 downto 3) <= window(5 downto 3) & data; ---start from index 3 shift to left for one bit
          cur_window <= cur_window(5 downto 0) & data;  --shift one bit every time
        END IF;
END IF;
END PROCESS;
```

If there is a new data comes into '*shift_window*", and if the number > current max number. The window will be shifted to new data. And the *shift_window* is triggered by *clk*, *reset*, *curState*, *rest_enable* signal. If clk has a rising edge: if '*reset*' or '*rest_enable*' is 1, *shift_window* will reset the '*max_num*', '*peak_index*' and '*max_window*'; or if the '*curState*' is '*dready*', it compares the current data with the '*max_num*'. If current value is bigger than the maximum value, it will be updated as '*max_num*', '*peak_index*' and '*max_window*' with the new data, window and index. It adds the value of '*cur_window*'(2 downto 0) to '*max_window*'(2 downto 0) while index 3 of '*cur_window*' and '*max_window*' have the same value.

According to an assign process, we assign value to dataresults and maxindex afterall numbers come out. The assign process is triggered by *clk*, *reset*, *final_enable* and *rest_enable* signal. When there exists a rising edge on clk signal, and one of the '*reset*' or '*rest_enable*' is 1, it resets the '*dataResults*' and '*maxIndex*'. Then sets all elements as 0. And if '*final_enable*' is 1: It copies the value from '*max_window*' to 'dataResults' in a loop iterating from index 6 down to 0; or it converts the '*peak_index*' integer to a series of bcd_num value and assigns them to the '*maxIndex*'. In '$maxIndex(2)$, divides '*peak_index*' by 100 and converts it to a 4-bit integer and then converts it to a 4-bit *std_logic_vector*. In '$maxIndex(1)$, divide '*peak_index*' by 100, get the remainder. Then divides the result by 10, then converts it to a 4-bit integer and then converts it to a 4-bit *std_logic_vector*. In '$maxIndex(0)$', divides it by 10, which retains the last digit of '*peak_index*' and get the remainder. Then converts it to a 4-bit integer and then converts it to a 4-bit std_logic_vector. When '*final_enable*' is set to 1, the maximum window's data is updated in the 'dataResults', and the '*peak_index*' is converted to *bcd_num* and assigned to '*maxIndex*'.

5 Project Timeline

Following is our Gantt Chart for project management:

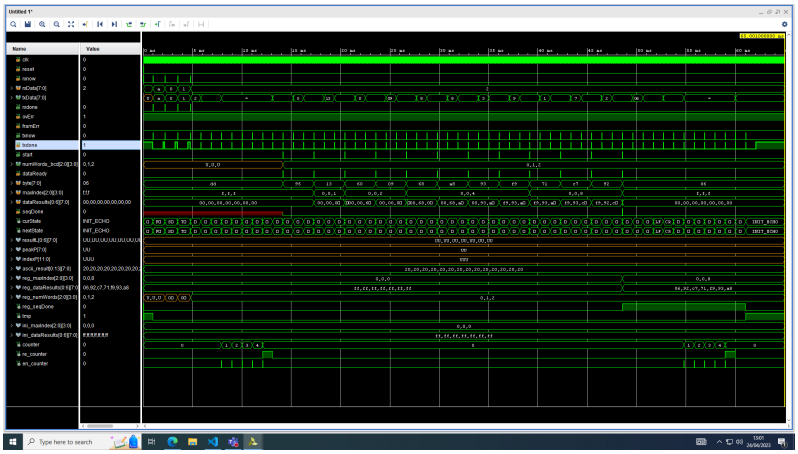


6 Peer Assessment

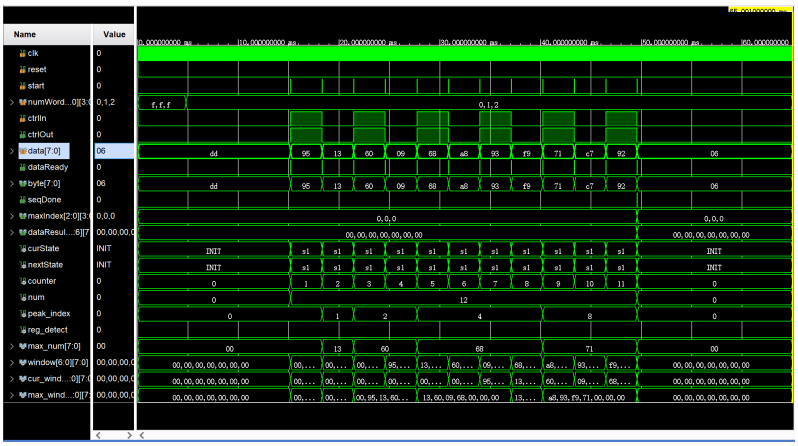
Group Member	1	2	3	4	5	spec (12.5 in total)
Name	Jiaying Shen	Yiyao Wang	Yumu Xie	Jiaxin Fan	Shiyu Song	
Username (e.g. ab12345)	ax21078	sz21463	po21744	yy21834	wf19101	
Leadership	5	4	2	1.5	0	12.5
Team Engagement	4	2	4.5	2	0	12.5
Carrying out technical work	4	3	4	1.5	0	12.5
Contributing to the report	3	3.5	3	3	0	12.5
Total / member	4	3.125	3.375	2	0	50

7 Conclusion

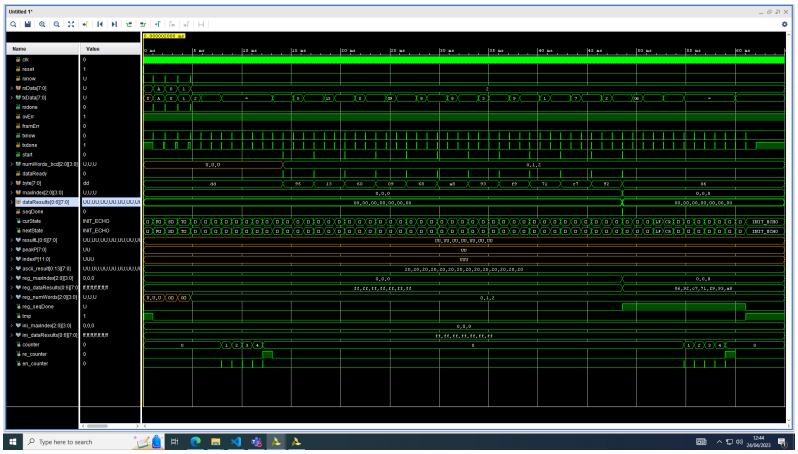
Here is behavioural simulation result of command processor:

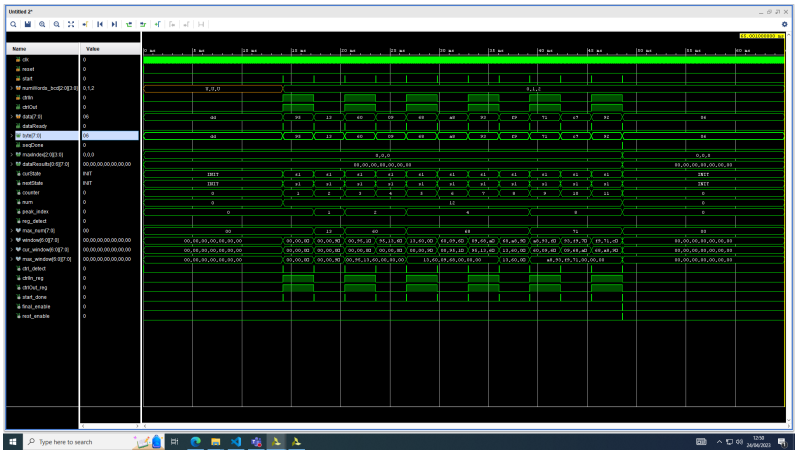


Here is behaviour simulation result of data processor:



Here is simulation result of full system (peak detector):





Here is output of terminal:

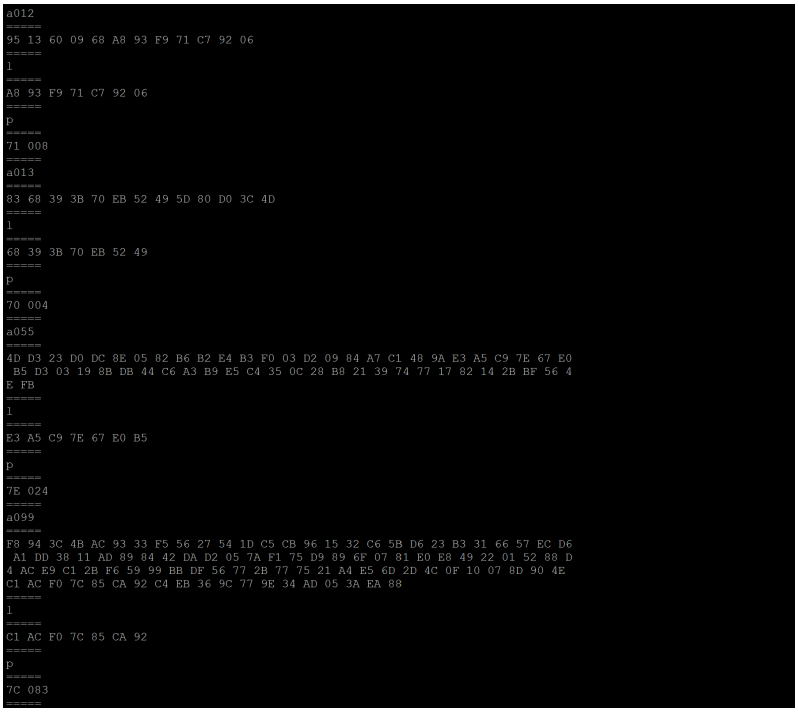


Fig. 6 aNNN/ANNN/l/L/p/P