Digital Design Peak Detector - Group 12 Report

Authors

jz22966 - Charlie Nasiadka, **qh22044** - Joshua Featherstone, **zo20876** - Damian Stone, **qk21907** - Billy Usher, **my22652** - Samuel Ogunmwonyi

Abstract

This report details the development of a peak detector system designed and implemented using VHDL by our team of five. This report documents the design, project planning, implementation, and integration processes

Table of Contents

1. Introduction

1.1 Tasks Description

We were given a task of designing, building and testing a peak detector in VHDL as a group of five members. Our final deliverable is code that passes three test benches in simulation as well as latch free synthesis onto an Artix 7 FPGA board.

Some functional components were provided including dataGen, Rx and Tx modules. We had to create implementations of the data processor and a command processor.

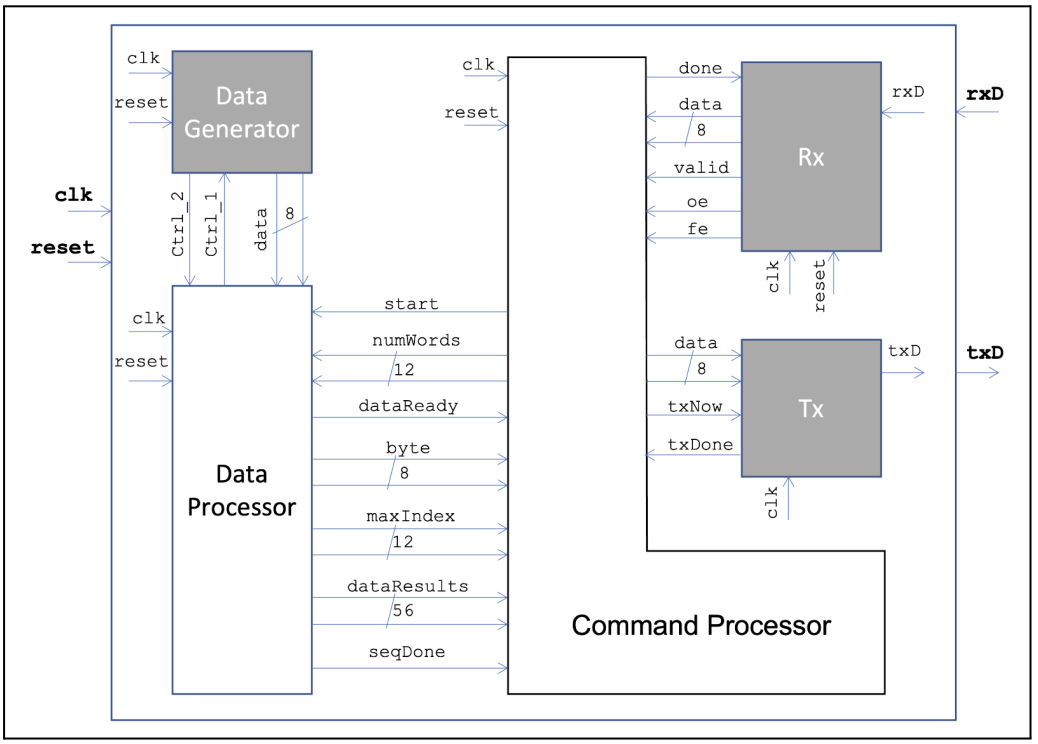


Figure 1 - Design Overview

Data processor role:

- Starting in sync with the command processor, using the 'start' signal
- Requesting data from the data generator, using 'ctrl1' and 'ctrl2' signals
- Informing the command processor to read data
- Analysing the peak and storing samples (7 samples, including the peak and 3 bytes on each side of it)
- Providing results

Command processor role:

- Managing UART links
- Command validation
- Requesting data from data processor
- Printing all input commands and results
- Converting and transmitting bytes as ASCII

1.2 Group Composition & Individual Task Division

There were two separate teams for Data processor (Charlie/Damian) and Command processor (Josh/Billy/Samuel).

Data Processor:

The group worked on creating an initial FSM to understand what we had to code. Then we split up the tasks to complete work. Charlie worked on implementing a peak detection algorithm, fixed two-phase logic causing bugs, decreased the number of states to pass tests, tested synthesis between the separate processors on the FPGA board. Damian worked on the data conversions, two phase protocol, handled dataReady and seqDone signals, removed latches from code, created testbenches to ensure the code works with edge cases and cleaned up the code for readability. Overall, the work was divided pretty equally, often helping each other and pair programming to complete an objective, as we found that this was more effective.

Command Processor:

Within the Command Processor group, a similar approach was carried out. We created our initial FSM then decomposed it into three key segments (A/P/L) that could be split between the group. Initially the code was built in three separate files cmdA, cmdP and cmdL each with an individual FSM. Billy handled the A command, looking for Tx input of the form aNNN, interacting with the data processor and outputting received bytes as two hexadecimal characters in ASCII to Tx. He wrote an individual test bench for this module tb_cmdA. Samuel was responsible for implementing command “P”, as well as developing a “Printer” component that would be used throughout the command processor for a consistent way to interface with the transmitter. He also assisted in the creation of the central “cmdProc” as well as debugging in order to produce a product that worked in simulation/synthesis. Similarly, he made tb_cmdP for this module. Josh implemented the “L” command and tb_cmdL test bench. He then fully integrated Billy’s A command code into the central FSM for final submission, which through gradual debugging and restructuring to ensure compliance, ending with the cmdProc passing the tb_CmdProcessor_Interim test bench.

Integration Plan

Before the integration of the two we first ensured both the Data/Command Processor worked with and modelled the behaviour of the .edn files. This naturally made integration of the two a simple process as they were fully functional and already interfaced with each other.

2. Project Plan

2.1 Project Plan and Meetings

The data processing group was able to communicate through frequent texts and calls to describe any changes to the code and any issues. This was very straightforward as there were just two working on this section. They held multiple pair-programming sessions, especially when determining the logic of the code or when faced with bugs.

The command processing group's method of communication was online meetings, along with texts to schedule them. This communication plan allowed the group to work with frequent communication even through the reading week/spring break, allowing us to tackle bugs that came with higher levels of integration.

In order to share code and any changes, we used git for version control. We made sure to create branches for any changes we make so as to not disrupt the development on the main branch. This approach allowed for smoother collaboration, as we could review each other's code through pull requests before merging them, thereby maintaining code integrity and minimising code conflicts.

2.2 Gantt Chart

Both teams agreed to a certain plan of action to follow with milestones shown below.

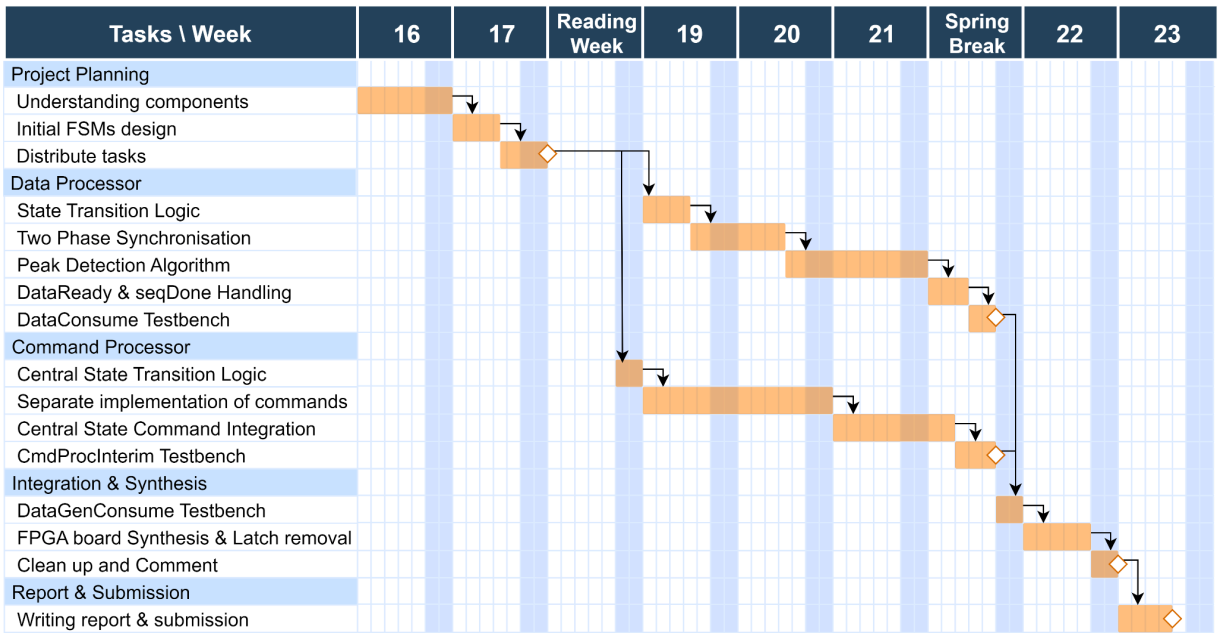


Figure 2 - Project Timeline using Gantt Chart

3. Data Processor

3.1 Architecture Breakdown

The 'dataConsume' architecture consists of several main logic components which include an entity, signal declarations, processes and a state machine.

Main Logic Components

- **State Machine**: Guides the program through a series of states,each responsible for different parts of the peak detection. (more detail in section 3.2)
- **Two Phase Protocol**: Identifies whether the ctrlIn signal has flipped to synchronise data processing events. Manages the ctrlOut signal based on the program's current state and operational conditions.
- **Peak Detection Algorithm**: Detects peaks, updates dataResults and maxIndex if a new peak is found and tracks the last three bytes processed.

Processes

- **StateMachine**: transitions to the next state.
- **combi_next:** defines the combinatorial logic of state transitions.
- **combi_out:** defines the combinatorial logic for state machine outputs.
- **CtrlInEdgeDetect:** detects the ctrlIn signal toggle.
- **CtrlOutToggle**: toggles the ctrlOut signal.
- **UpdateCounter:** increments counter to keep track of current index of byte.
- **BCDtoINT:** converts numWords_bcd input signal from BCD to integer.
- **ByteOutput:** outputs data bytes in sync with the clock.
- **PeakDetection:** contains the peak detection logic.

Signal Definitions

State Management Signals
- **Curr_state:** represents the current operational state of the module
- **Next_state:** determines the next state to transition to.

Two Phase Protocol Signals
- **prev_ctrlIn:** Holds the previous value of the ctrlIn input signal.
- **edge_detected_ctrlIn:** Set to high when the ctrlIn signal is toggled.
- **ctrlOut_state:** Determines the current state of the ctrlOut output signal.

Data Handling and Counter Signals
- **numWords_int:** Holds the integer value of numWords_bcd
- **counter:** Tracks the current number of bytes processed and serves as a peak index

Peak Detection Signals
- **Peak_value:** Holds the current peak value detected in the data stream.
- **lastThreeBytes:** Acts like a buffer, storing the last three bytes of data processed.
- **Update_next_values:** Manages the logic for storing the next three values after a peak is detected.

Enable signals

- **en_updateCounter:** Enables the increment of the counter signal.
- **en_bcdToInt:** Enables the conversion process from BCD to integer for the numWords_bcd signal.
- **en_peakDetection:** Enables the peak detection algorithm to update peak values and manage related buffers and indices.
- **en_byteOutput:** Enables the output of data bytes to the command processor.
- **en_reset:** Forces a reset or initialization of critical signals and counters.

3.2 State Machine of Data Processor

The following diagram **Figure 3** describes the structure of the FSM (Finite State Machine) used for the Data Processor. Each of the four states is designed to manage specific tasks and transitions based on input signals.

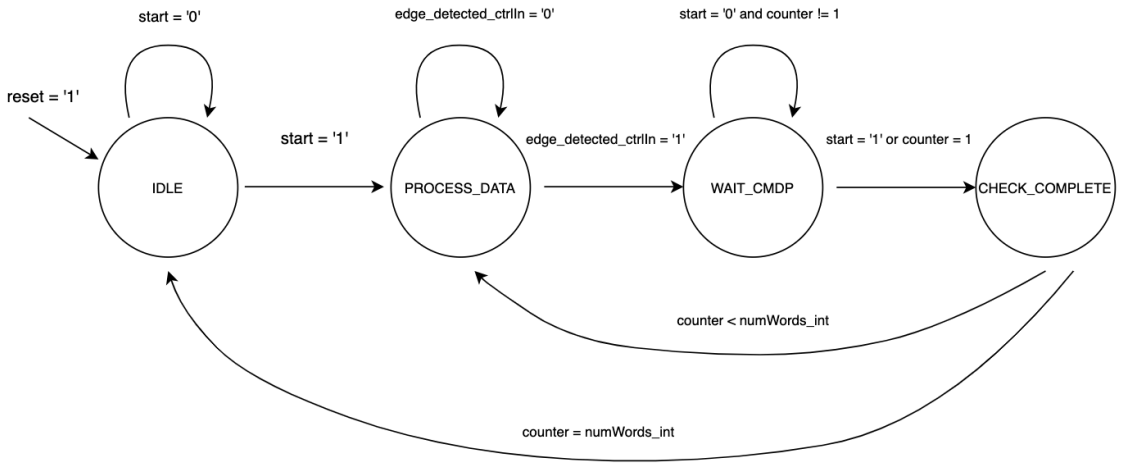


Figure 3 - Data Processor Finite State Machine

We decided to use a Mealy architecture, where the outputs depend on both the current state and the inputs, as this allows a quick reaction to changes in the input signals which is crucial for real-time control and synchronisation between the Data Processor and the Data Generator. We believe this improves communication efficiency in systems that require dynamic responses based on changing input conditions.

For the implementation of the code, the transitions are carried out through a process called combi_next, which handles only the logic of the state transitions while combi_out deals with the outputs and additional processes for each state. In this way, two crucial logics for the functioning of our FSM are divided, making the code more readable and easier to debug.

3.3 Explanation of Code

As shown earlier, the FSM connects the 4 main components for data processing, so this section seeks to explain in more detail how the program functions at the time of execution. First we reset signals to avoid latches in our code. Then the State Machine goes as follows:

1. The program starts in the **IDLE** state, which enables 'en_reset' to reset the output signals such as: dataResults and maxIndex. Signals created internally like: counter, peak_value, update_next_values, and lastThreeBytes are also reset. This ensures a clean start for each execution, clearing any residual data from previous operations. The program remains in the IDLE state until the Command Processor toggles the 'start' signal high. Upon receiving this signal, the system prepares to transition to the next state. Before moving on, it converts the 'numWords_bcd' signal into an integer and toggles 'ctrlOut' through the CtrlOutToggle process to request data from the Data Generator, which initiates the two-phase protocol.

2. In **PROCESS_DATA**, the program waits until the edge_detected_ctrlIn signal is high, which indicates that the Data Generator has sent new data through a two-phase protocol system. Once new data is received, the en_updateCounter and en_peakDetection signals are enabled, activating two essential processes UpdateCounter and PeakDetection.

UpdateCounter increments the internal counter of the Data Processor to keep track of how many bytes we have processed from the Data Generator.

PeakDetection is the process that carries out the execution of the algorithm. If a new peak is found, maxIndex and dataResults are updated by adding the new peak at the middle (at index 3), the three bytes that came before the peak and resets the values of the next 3 bytes, as shown in Figure 4.

```
if counter = 0 or signed(data) > peak_value then
    peak_value <= signed(data); -- update peak value for future comparison
    maxIndex(0) <= std_logic_vector(to_unsigned(counter mod 10, 4)); -- Update max index using counter
    maxIndex(1) <= std_logic_vector(to_unsigned((counter / 10) mod 10, 4));
    maxIndex(2) <= std_logic_vector(to_unsigned((counter / 100) mod 10, 4));
    dataResults(6) <= std_logic_vector(lastThreeBytes(2));-- Update the three values before the peak
    dataResults(5) <= std_logic_vector(lastThreeBytes(1));
    dataResults(4) <= std_logic_vector(lastThreeBytes(0));
    dataResults(3) <= std_logic_vector(signed(data)); -- Update the peak value
    dataResults(2) <= (others => '0'); -- Reset the three values after the peak
    dataResults(1) <= (others => '0');
    dataResults(0) <= (others => '0');
    update_next_values <= 3; -- indicating that we need to store the next three values
```

Figure 4 - Peak Detection

The update_next_values signal is also set to the value of 3 upon finding a new peak, which indicates that we must store the next 3 bytes that will be processed in the future.

```
update_next_values <= 3; -- indicating that we need to store the next three values
```

Figure 5 - Setting Update_next_values to 3

This ensures that dataResults are filled in with their corresponding values as the data is processed without having to store all processed data, making the algorithm work with low space complexity.

```
if update_next_values > 0 then
  dataResults(update_next_values - 1) <= std_logic_vector(signed(data));
  update_next_values <= update_next_values - 1;
end if;
```

Figure 6 - Filling in next three bytes

Finally, it always saves the last three processed values in a buffer called lastThreeBytes, so we can update dataResults(6 downto 4) when a peak is found.

```
lastThreeBytes(2) <= lastThreeBytes(1);
lastThreeBytes(1) <= lastThreeBytes(0);
lastThreeBytes(0) <= signed(data);
```

Figure 7 - updating buffer

To ensure the functioning of an efficient algorithm that works correctly in any case, the algorithm was first developed in Python and then tested using different test cases.

3. After processing the new data, the program moves to the next state **WAIT_CMDP**, where the en_byteOutput signal is enabled to send the newly processed byte to the Command Processor through the ByteOutput process. Then it waits for the Command Processor to send the start signal again, indicating that it is ready to proceed to the next state.

4. Once the start signal is received, the program proceeds to the **CHECK_COMPLETE** state, where it is decided whether to continue processing data, checking if the counter equals numWords_int. If the total of words needed to process has not been reached, the program will move back to the **PROCESS_DATA** state to continue processing data, otherwise the program will end moving to the **IDLE** state.

4. Command Processor

4.1 Architecture Breakdown

The *cmdProc* architecture *arch* encompasses our aNNN and P/L command logic. Within this there are multiple processes with individual functionality.

The *combi_nextState* process contains the state logic for our central FSM. Upon receiving an a/A character from the *Rx* module it routes down through states *N2, PRINT_N2, N1, PRINT_N1, N0, PRINT_N0* responsible for checking for valid digit inputs and echoing. After carrying out carriage return and line feed, the command processor then signals the data processor via the *start* signal in state *STARTING.* Receiving bytes one at a time, the command processor outputs each 4 bit nibble as a hexadecimal character in the *Tx* using hexadecimal format in states *HEX1, HEX2*. Checking at this point for signal *seq_Done* to go high, the FSM decides whether to print a space in *SPACE* or halt execution and return to the *INIT* state. Upon receiving a P/L character at any point during states *INIT, N2, N1, N0* the FSM instead hands execution to the cmdP or cmdL component, staying in states *P/L*

```
-----------------------------
-- Combinatorial Inputs
-----------------------------
combi_nextState: PROCESS(curState, rxnow_reg, rxData_reg, seq_Available, doneL, doneP, finished, dataReady_reg)
BEGIN
    CASE curState IS
        ------------------------------------------------------------------------------------------
        -- Central FSM
        ------------------------------------------------------------------------------------------
        WHEN RESTART => -- Wait for printing after invalid input…
        WHEN INIT => -- Wait for rxNow to give valid rx input…
        WHEN VALID => -- Check if A/P/L or otherwise restart…
        ------------------------------------------------------------------------------------------
        -- A command Inputs
        ------------------------------------------------------------------------------------------
        WHEN PRINT_A => -- Wait for A to print…
        WHEN N2 => -- Wait for next input…
        WHEN PRINT_N2 =>…
        WHEN N1 => -- Wait for next input…
        WHEN PRINT_N1 =>…
        WHEN N0 => -- Wait for next input…
        WHEN PRINT_N0 =>…
        ------------------------------------------------------------------------------------------
        -- Carriage Return and Line Feed
        ------------------------------------------------------------------------------------------
        WHEN WAIT_CARRIAGE => -- Prep printing for carriage return…
        WHEN CARRIAGE_RETURN => -- Wait for carriage return to print…
        WHEN WAIT_LINE =>…
        WHEN LINE_FEED =>…
        ------------------------------------------------------------------------------------------
        -- Data Processor communication and bytes printed
        ------------------------------------------------------------------------------------------
        WHEN STARTING => -- Set start and numWords…
        WHEN DATAPROC => -- Wait for data from data Processor…
        WHEN PREP_HEX1 => -- Prep printing for first Hex Digit…
        WHEN HEX1 => -- Wait for printing for first Hex Digit…
        WHEN PREP_HEX2 => -- Prep printing for first Hex Digit…
        WHEN HEX2 => -- Wait for printing for second Hex Digit…
        WHEN PREP_SPACE => -- Prep printing for space…
        WHEN SPACE => -- Wait for space to be printed…
        ------------------------------------------------------------------------------------------
        -- Carriage Return and Line Feed
        ------------------------------------------------------------------------------------------
        WHEN WAIT_CARRIAGE2 => -- Prep printing for carriage return…
        WHEN CARRIAGE_RETURN2 => -- Wait for carriage return to print…
        WHEN WAIT_LINE2 =>…
        WHEN LINE_FEED2 =>…
        ------------------------------------------------------------------------------------------
        -- P and L commands
        ------------------------------------------------------------------------------------------
        WHEN PRINT_P => -- Wait for valid P character to print…
        WHEN PRINT_L => -- Wait for valid L character to print…
        WHEN PREP_P =>…
        WHEN PREP_L =>…
        WHEN P => -- P component active…
        WHEN L => -- L component active…
        WHEN OTHERS =>…
    END CASE;
END PROCESS;
```

Figure 5 - Collapsed combi_nextState process

The *combi_out* process works to provide a mealy FSM implementation. We took care to avoid latches by ensuring any outputs either have default values defined outside the case structure or used enable based register processes. These latches became apparent initially when running synthesis so we acted to adapt our code and remove these using set VHDL principles. The *combi_out* process alongside *store_NNN, store_byte, seq_numWords* manage registering NNN digits, driving start high, storing bytes received along the byte signal from the data processor and populating the printer for Tx communication.

```
en <= '0';
en_storedByte <= '0';
enP <= '0';
enL <= '0';
en_NNN_2 <= '0';
en_NNN_1 <= '0';
en_NNN_0 <= '0';
en_numWords <= '0';
rxDone <= '0';
start <= '0';
NNN_val <= "0000";
dataIn <= "00000000";
```

```
-----------------------------
-- Sequential storage logic for NNN/ store_Byte
-----------------------------
store_NNN: PROCESS(clk)
BEGIN
    IF clk'EVENT AND clk='1' THEN
      IF en_NNN_2='1' THEN
        NNN(2) <= NNN_val;
      ELSIF en_NNN_1='1' THEN
        NNN(1) <= NNN_val;
      ELSIF en_NNN_0='1' THEN
        NNN(0) <= NNN_val;
      END IF;
    END IF;
END PROCESS;

store_byte: PROCESS(clk)
BEGIN
    IF clk'EVENT AND clk='1' THEN
      IF en_storedByte='1' THEN
        storedByte <= UNSIGNED(byte_reg);
      END IF;
    END IF;
END PROCESS;

seq_numWords: PROCESS(clk)
BEGIN
    IF clk'EVENT AND clk='1' THEN
        IF en_numWords = '1' THEN
            numWords_bcd <= NNN;
        END IF;
    END IF;
 END PROCESS;
```

Figure 6 - Default Signals and Register Processes

The *sequencing* and *data* processes are responsible for storing values, required later in P/L, received from the data processor that are only driven high for one clock cycle

```
-----------------------------
-- seq_Available Managing
-----------------------------
sequencing: PROCESS (clk)
BEGIN
    IF clk'EVENT AND clk='1' THEN
        IF RESET = '1' THEN
            seq_Available <= '0';
        ELSIF curState = STARTING THEN
            seq_Available <= '0';
        ELSIF seqDone_reg = '1' THEN
            seq_Available <= '1';
        END IF;
    END IF;
END PROCESS;

-----------------------------
-- Data Results and Max Index Managing
-----------------------------
data: PROCESS (clk)
BEGIN
    IF clk'EVENT AND clk='1' THEN
        IF RESET = '1' THEN
            maxIndex_stored <= ("0000", "0000", "0000");
            dataResults_stored <= ("00000000", "00000000", "00000000", "00000000", "00000000", "00000000", "00000000");
        ELSIF seqDone_reg = '1' THEN
            maxIndex_stored <= maxIndex_reg;
            dataResults_stored <= dataResults_reg;
        END IF;
    END IF;
END PROCESS;
```

Figure 7 - Sequence Available, Data Results and Max Index

4.2 Component Composition

To provide modularity we implemented three components (*cmdP, cmdL* and *printer*) to carry out specific functions. These are each in separate VHDL files and have their own FSMs. Components *cmdP* and *cmdL* are enabled from cmdProc using the *enP/enL* signals. They handle their functions separately, outputting to *Tx*. The P command (peak printing) prints the peak byte and its corresponding index in the sequence. Whereas the L command (listing) lists the 7 bytes surrounding and including the peak.

```
-------------------------
-- Component Definitions
-------------------------
COMPONENT printer IS
    PORT (
      clk:          IN std_logic;                             --i
      reset:        IN std_logic;                             --i
      en:           IN std_logic;                             --i
      dataIn:       IN std_logic_vector (7 DOWNTO 0);         --i
      txDone:       IN std_logic;                             --i
      txData:       OUT std_logic_vector (7 DOWNTO 0);        --o
      txnow:        OUT std_logic;                            --o
      finished:     OUT std_logic                             --o
    );
END COMPONENT printer;

COMPONENT cmdP IS
    PORT (
       clk:         IN std_logic;                             --i
       reset:       IN std_logic;                             --i
       en:          IN std_logic;                             --i
       peakByte:    IN std_logic_vector (7 DOWNTO 0);         --i
       maxIndex:    IN BCD_ARRAY_TYPE(2 DOWNTO 0);            --i
       txdone:      IN std_logic;                             --i
       txData:      OUT std_logic_vector (7 DOWNTO 0);        --o
       txnow:       OUT std_logic;                            --o
       doneP:       OUT std_logic                             --o
    );
END COMPONENT cmdP;

COMPONENT cmdL IS
    PORT (
      clk:          IN std_logic;                             --i
      reset:        IN std_logic;                             --i
      enL:          IN std_logic;                             --i
      dataResults:  IN CHAR_ARRAY_TYPE(0 TO RESULT_BYTE_NUM-1); --i
      txdone:       IN std_logic;                             --i
      txData:       OUT std_logic_vector (7 DOWNTO 0);        --o
      txnow:        OUT std_logic;                            --o
      doneL:        OUT std_logic                             --o
    );
END COMPONENT cmdL;
```

Figure 8 - Components

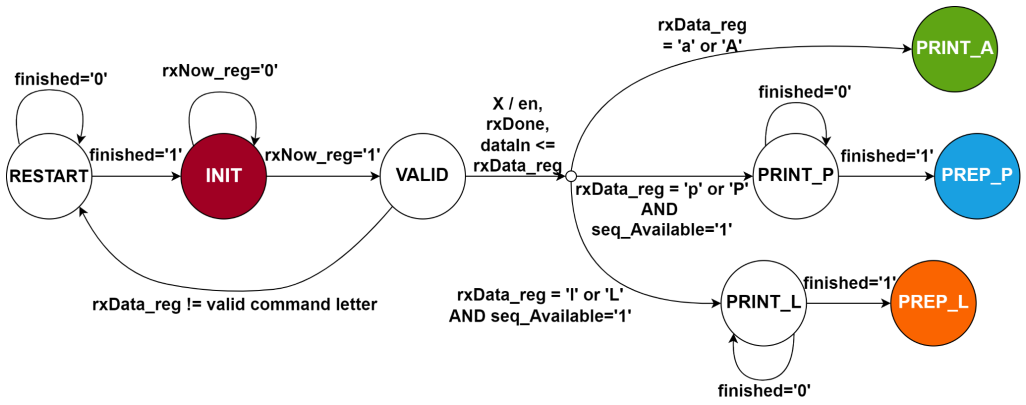
The *printer* module is a generic printer used throughout our implementation. There is an instance of this in both the central FSM and P/L. It acts to replace signals *txData, txNow* and *txDone* with new signals *dataIn, en* and *finished*. It can be defined as a component throughout the code and prints to *Tx* the value of *dataIn* whenever *en* is driven high. Because of our multiple components, there are many signals which could be the current value of *txData* and *txNow* inputs. We handle these multiple drivers state dependently.

```
----------------------------
--  Multiple driver control
----------------------------
assign_tx: PROCESS(curState, txNow_M, txNow_L, txNow_P, txData_M, txData_L, txData_P)
BEGIN
    IF curState = L THEN
        txData <= txData_L;
        txNow <= txNow_L;
    ELSIF curState = P THEN
        txData <= txData_P;
        txNow <= txNow_P;
    ELSE
        txData <= txData_M;
        txNow <= txNow_M;
    END IF;
END PROCESS;
```

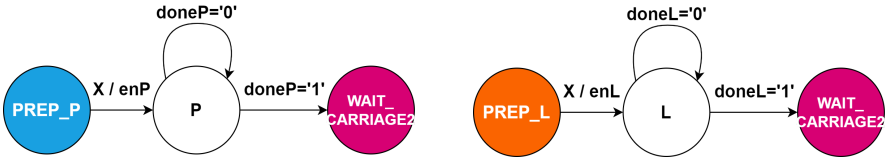
Figure 9 - Multiple driver control

4.3 State Machine of Command Processor

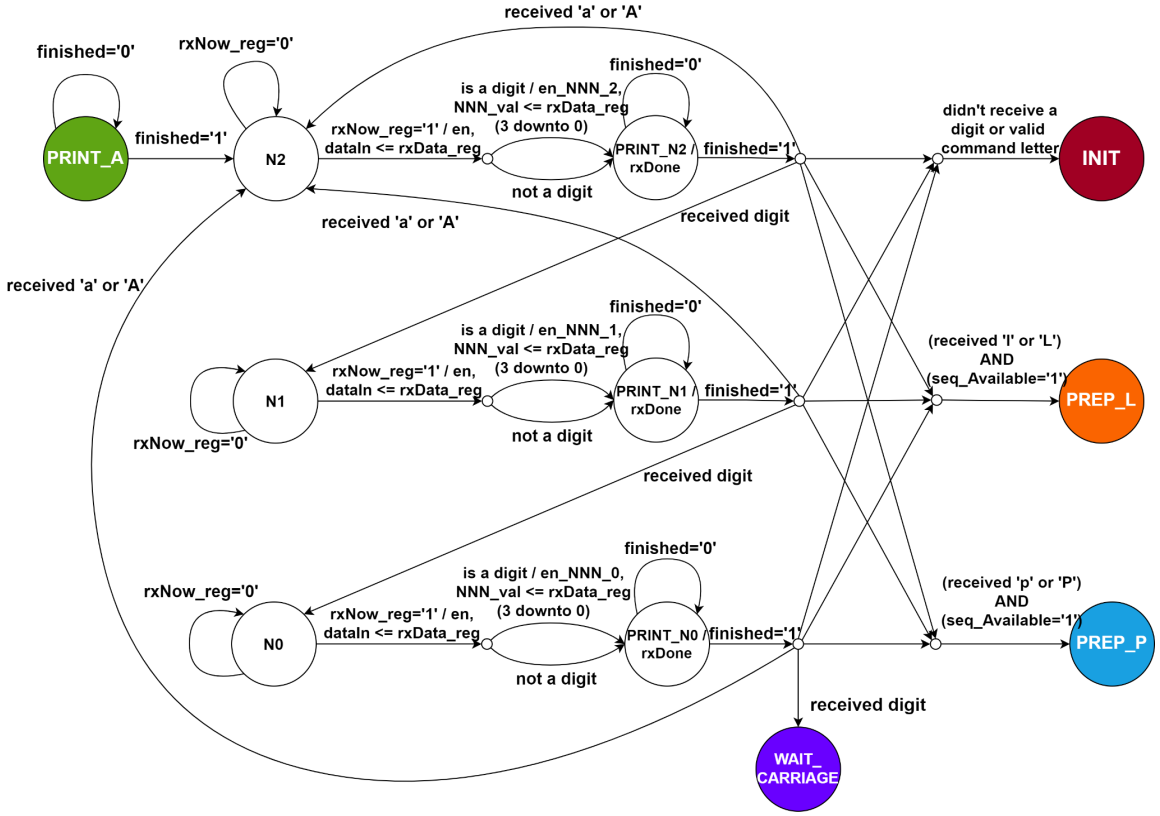
Initial:



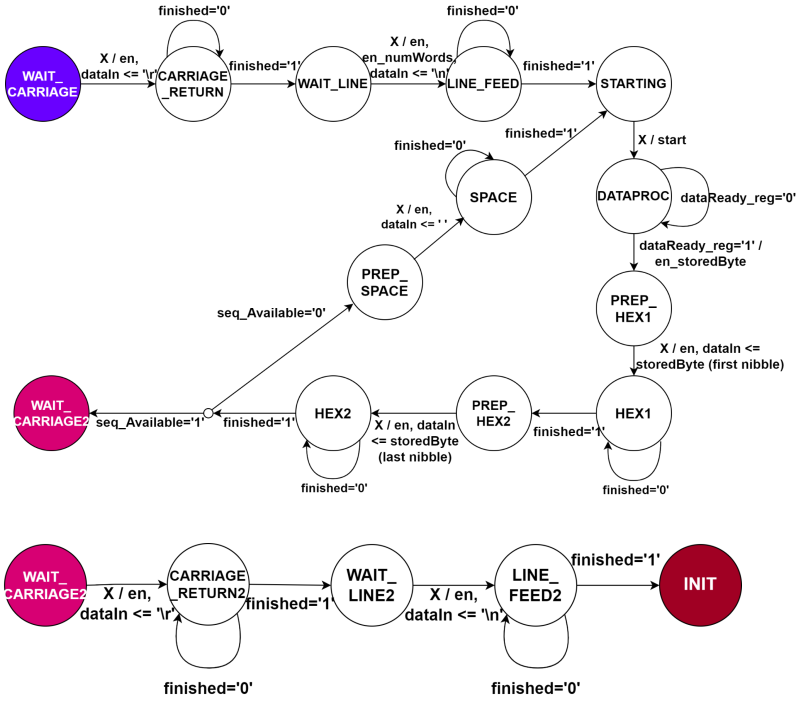
P and L states:



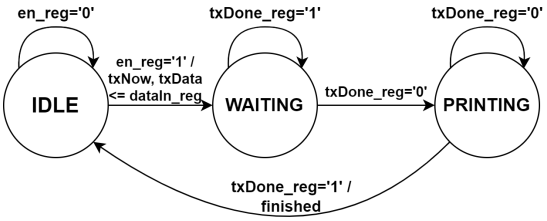
aNNN:



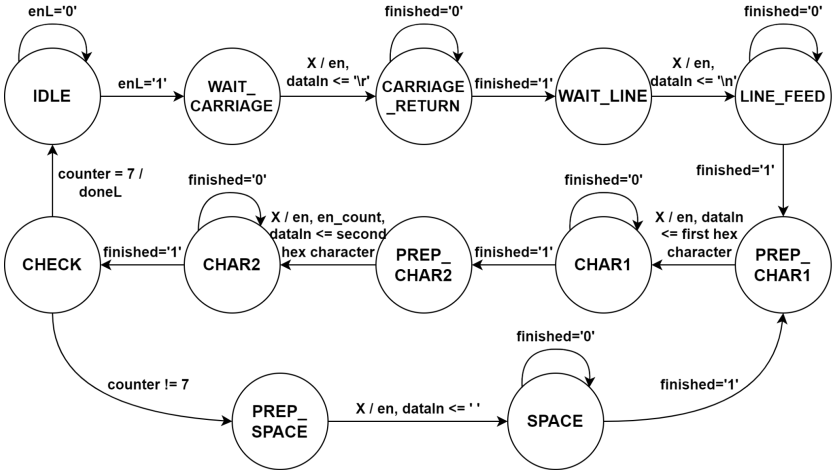
Data Processing:



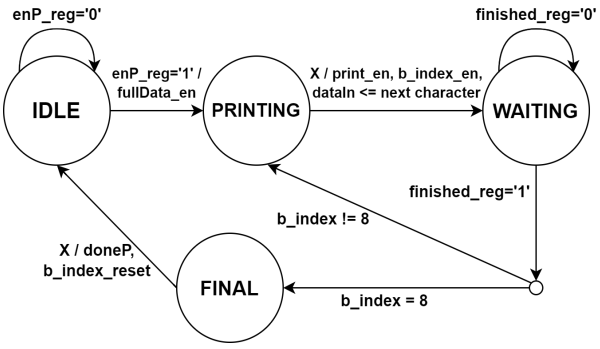
Printer:



cmdL:



cmdP:



5. Integration, System Evaluation and Conclusion

Test Benches

Once we had completed our parts and ensured they worked with the tb_dataConsume and tb_cmdProcessor_Interim testbenches, we moved on to synthesising the data and command processor separately on the FPGA board with .edn files to ensure correct functionality. We moved onto integration of the two components. Replacing the .edn files in simulation, our combined code passed the tb_dataGenConsume test bench.

Latches and Waveform

This led to several issues being uncovered despite our code working with the testbenches in simulation. These issues were primarily caused by inferred latches that altered the synthesised result in unpredictable ways. These proved difficult to debug as these did not cause errors in simulation, yet did in synthesis because of how the FPGA synthesises code.
To remove these latches we ensured that signals in combinatorial processes had values specified in all branches of our combinatorial processes. As some signals needed to be persistent, we replaced these with enable signals that activated edge triggered clocked processes to hold them. This ensured they would not be inferred as latches, but flip flops. Once we fixed our code and it synthesised correctly, we received the following waveform.

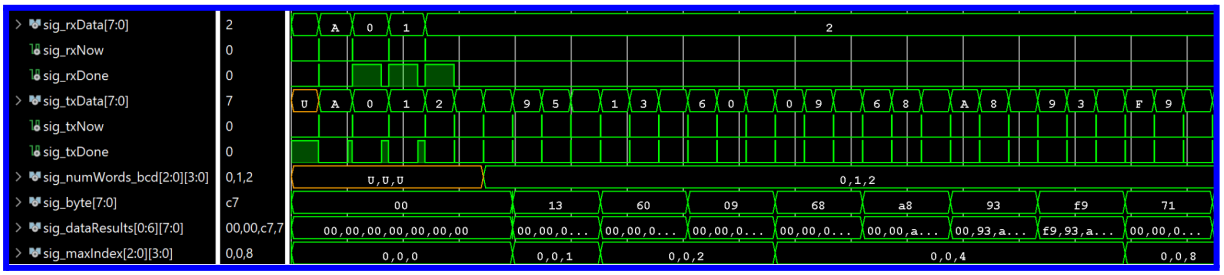


Figure X - Simulation Waveform Example

Putty Terminal Final Synthesis

We finally synthesised the code onto the FPGA and ran the following commands in the Putty terminal

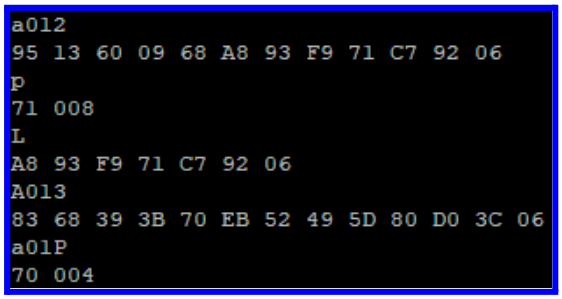


Figure X - PUTTY terminal demonstration

As shown, the Command Processor successfully receives a case-insensitive aNNN command and the Data Processor is able to return the requested bytes. The P and L commands display the correct information required. The final two commands show the ability to re-enter the ANNN command, as well as show how the aNNP/aNNL edge case has been taken into consideration. The beginning/end of each command is formatted with line feed and carriage returns.

6. Conclusion

In conclusion, this report has documented the successful interpretation, design, implementation, and integration of our VHDL-based peak detector system. We adhered to a schedule and effectively divided tasks among team members, meeting deadlines. We met all project outcomes and requirements, including the successful passing of various simulation tests and the implementation on an FPGA board. We overcame challenges such as inferred latches during synthesis, timing in simulation and synchronisation between components. This project showcases our technical skills in VHDL and system integration and our problem-solving and team collaboration skills.

7. Peer Assessment

Group Member	1	2	3	4	5
Name	Charlie	Damian	Josh	Billy	Samuel
Username	jz22966	zo20876	qh22044	qk21907	my22652
Leadership	2.5	2.5	3	2.5	2
Team engagement	2.5	2.5	2.5	2.5	2.5
Carrying out technical work	2.25	2	2.75	2.5	3
Contributing to the report	2.5	2.5	2.5	2.5	2.5
Total / member	9.75	9.5	10.75	10	10