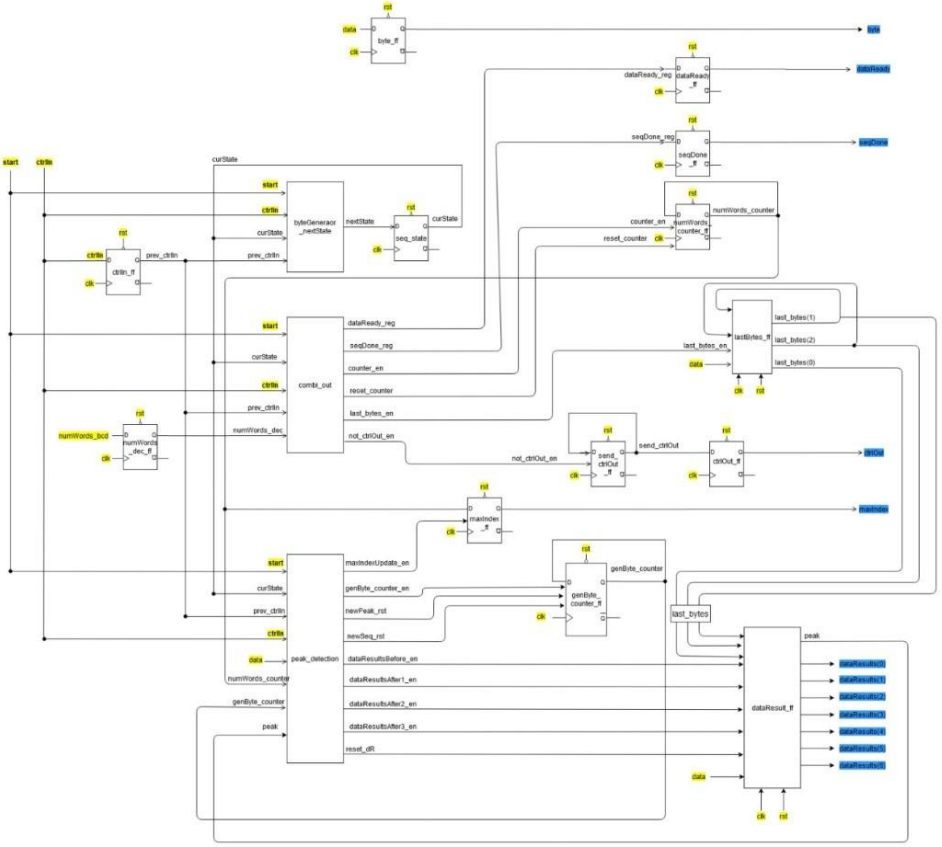
Data Processor of Peak Detector system

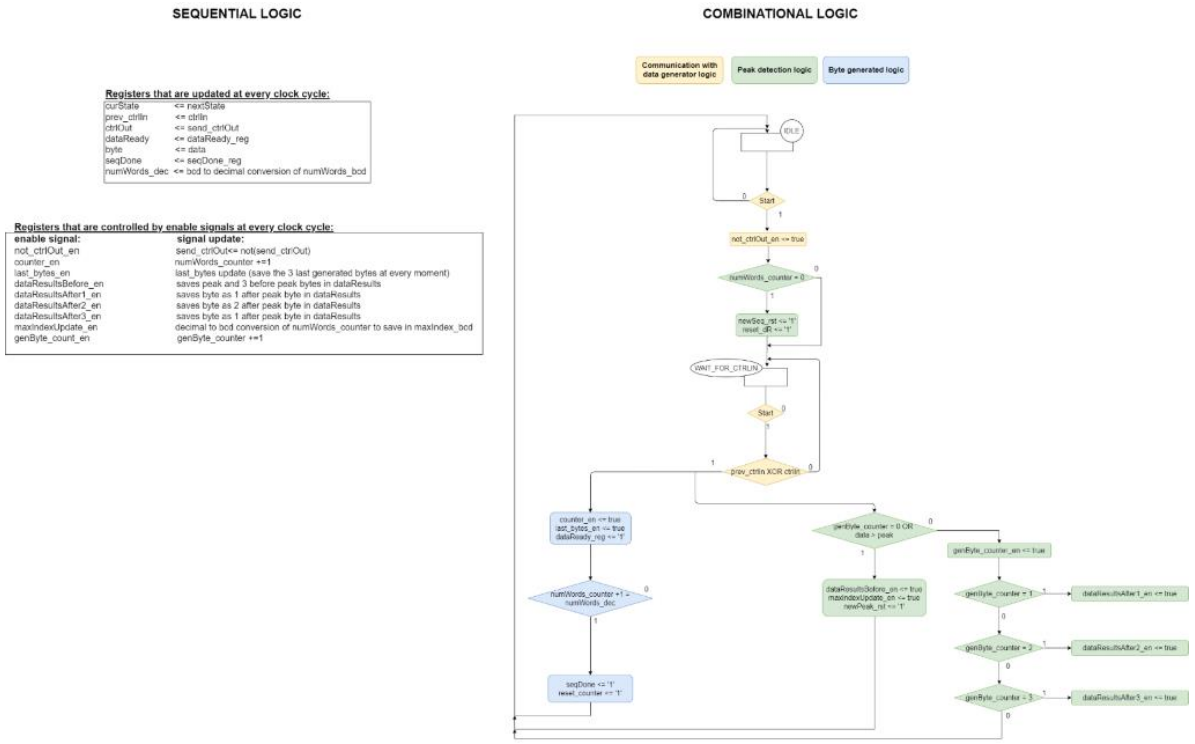
Project Instructions: https://seis.bristol.ac.uk/~sy13201/A2_index.htm

1. The Data Processing module

1.1. Block-level sketch of main logic components



1.2. State diagram for FSM



1.3. Work discussion

The data processor is separated to the combinational and the sequential elements.
The combinational elements consist of 3 main blocks: the **“byteGenerator_nextState”**, the **“combi_out”** and the **“peak_detection”**.
Each of these processes are responsible for a set of important output signals these can be either enable signals which control registers or values to update registers.

- **“byteGenerator_nextState”** is the main logic for the FSM which outputs the next state.
- **“combi_out”** is the block responsible for the setting of values and enable signals which control the following outputs:
  ctrlOut, dataReady, seqDone, numWords_counter, last_bytes
  These signals are used for the communication between the data processor and the byte generator in order to generate a new byte but also to provide the requested outputs to the command processor.
- **“peak_detection”** is responsible for the enable signals of **genByte_counter, dataResults** and **maxIndex**.
  It detects peaks and saves the appropriate bytes into the **dataResults** and **maxIndex** output signals.

For sequential elements there are 3 different kinds.

1. Registers which just update their values at every clock rising edge: **ctrlIn_ff, seq_State_ff, ctrlOut_ff, dataReady_ff, byte_ff, seqDone_ff**
2. Registers which process and update signal values and are controlled by enable signals: **send_ctrlOut, last_bytes_ff, maxIndex_ff, dataRes_ff**
3. Counters: **numWords_counter_ff, genByte_count_ff**

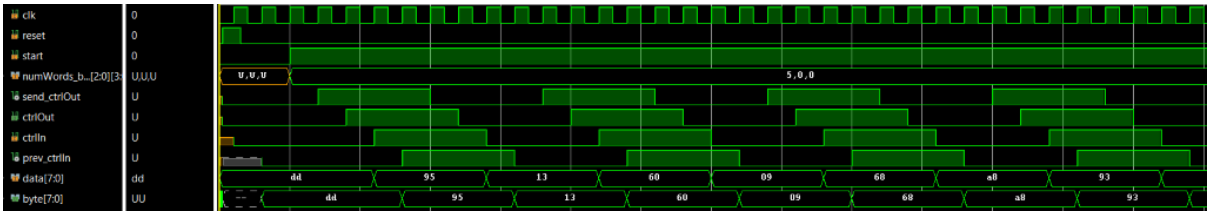
We developed all of these elements and processes together but then we had to simulate it, test it and debug it. For that phase of the project, the university was closed so we had to work individually, sharing our work and progress and finalizing the program.
The first step was to make our program able to simulate and then resolve any errors that were raised by Vivado. We discovered our main problem was that each signal should be updated in only one process; which was something we didn't consider when we were writing the code. From that point on, we were able to test each aspect of the program separately and ensured it was working properly.
Next, we tested and debugged the **"byteGenerator_nextState"** and **"combi_out"** which communicates with the data generator (triggering **ctrlOut** to signal data generation and waiting for **ctrlIn** to transition that signals that the new byte is generated) but also communicates with the command processor (sets the **dataReady** and **seqDone** signal whenever necessary).
Another important step is the BCD to decimal conversion of the **numWords_BCD** signal which comes from the command processor and gives us the number of bytes that need to be generated.
Also, we have a **numWords_counter** which counts the number of bytes generated and checks regularly whether all of the bytes have been generated and the sequence is done.
Last but not least, there is a **last_bytes** signal which saves at every moment in time the 3 last bytes that have been generated, with the aim to have the 3 before peak bytes ready to be saved in case the current new byte is a peak.
The next most important part of the data processor is the peak detection logic which is implemented in the **"peak_detection"** process.
This process is in charge of checking the bytes generated and detecting a peak if there is one.
Also, after detecting a peak, it is responsible of giving the correct enable signals to save the before peak bytes and the peak byte into the **dataResults** but also of saving the 3 consecutive bytes after it that will be generated in order to save the 3 after bytes in the **dataResults** (this is done by the use of **genByte_counter** signal which counts how many bytes have been generated after detecting a peak).
Finally, there is a binary to BCD conversion block of logic which is used when a peak is detected. This logic takes the **numWord_counter** which keeps track of the index of the byte in the sequence and converts it to BCD form in order to save it into the **maxIndex** signal.
After testing each of these blocks we encountered some final problems with the matchings of our results and the testbench results. The problem was that we saved the **dataResults** in backwards, but that was easily solved.
Also, it seems that our program didn’t process all 6 sequences and it had only the correct results for the first two sequences. The issue was resolved after we realized that we had a different understanding of the specifications and after asking a TA realised that the start signal didn’t need to go again to 0 before starting a new sequence.

Finally, as it can be seen in the simulation results in the last section of the report we checked that all sequences had the correct **dataResults** and **maxIndex** at the end of each sequence and the **seqDone** signal was set high for one clock cycle after each sequence.
At the end we also checked the code of any coding optimizations that could be implemented and also adapted our code in order to adhered to all the specifications that were given by the project instructions.

2. Simulation results and test of our implementation

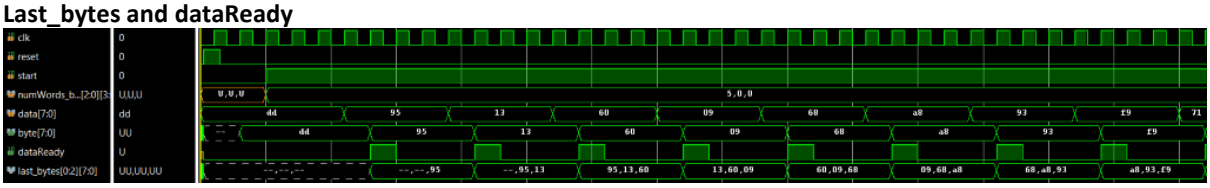
In order to make sure that our data processor worked correctly we tested/simulated it with the testbench provided for unsigned data ("tb_dataConsume_unsigned.vhd")
To test our implementation, we checked the main functionalities below:

Start



In this figure the communication between the data processor and the data generator is shown.
As soon as the start goes high, the ctrlOut changes its current value from 0 to 1 by the signal send_ctrlOut. This transition signals to the data generator that we need a new byte.
In turn the data generator changes the ctrlIn signal value from 0 to 1 to indicate that the new byte has been generated and in the next rising clock edge the data is saved into the byte output signal which will be provided to the command processor.

Last_bytes and dataReady

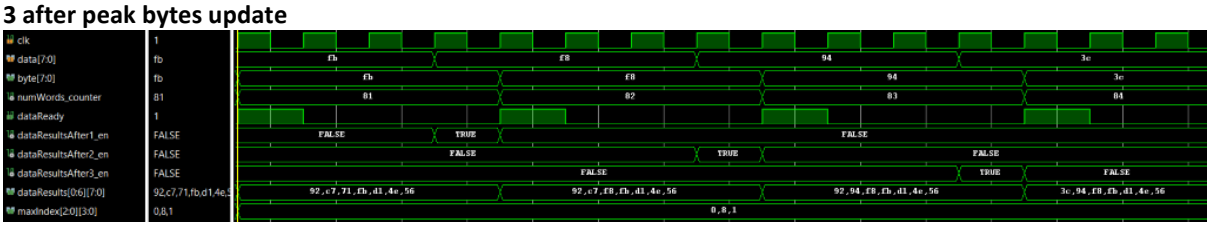


In this figure the functionality of the last_bytes signal (that we added) and of the dataReady signal (that is provided for the command processor) is illustrated.
Last_bytes is a signal that has to save the last 3 bytes generated at every moment in time.
As it is shown, every time that a new byte is generated, after one clock cycle the dataReady signal is set high and also this new byte is saved in the last_bytes signal. With every new byte the previous bytes jumps over to the next position of the signal, and that way at every new byte only the last 3 generated bytes are saved in the last_bytes signal.

BCD to decimal conversion

In this figure it is shown the correct conversion of the numWords_bcd signal to a decimal number saved in the numWords_dec signal.

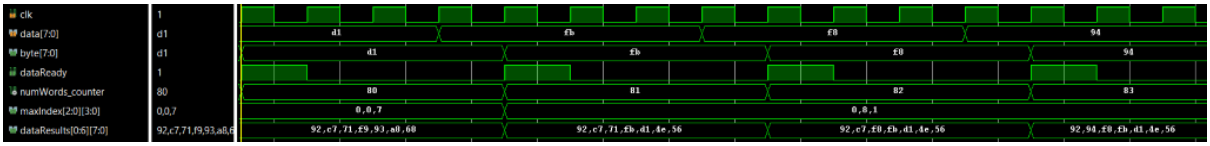
3 after peak bytes update



In this figure the function of the peak detection and 3 after peak bytes update of dataResults is shown.
As soon as a new peak is detected, in this case the 81st byte generated (fb), this new peak is saved as the peak and the 3 last generated bytes are saved in dataResults.
After the update of the peak however it is necessary to check the 3 consecutive bytes in case they are also peaks but if they're not they have to be saved in the dataResults signal as the 3 after peak bytes.

This functionality is correctly illustrated in this figure, since after fb is saved then f8, 94 and 3c bytes are saved in the dataResults signal.

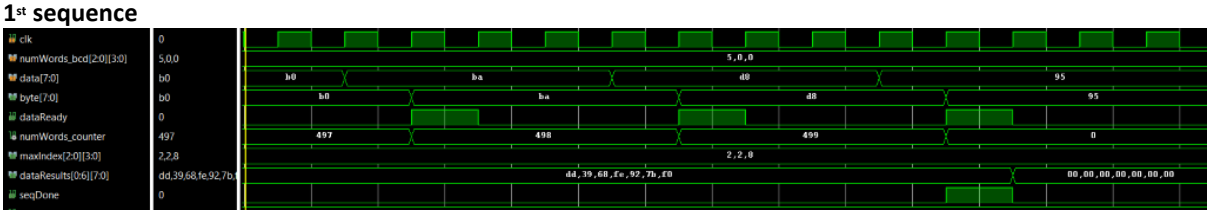
Decimal to BCD conversion



In this figure the decimal to BCD conversion is shown. This conversion is done from getting the decimal number of the numWords_counter and converting it and to BCD form and saving it to the maxIndex signal. As soon as a new peak is detected when numWords_counter is 81, this number is converted to bcd form (0,8,1) and saved into maxIndex.

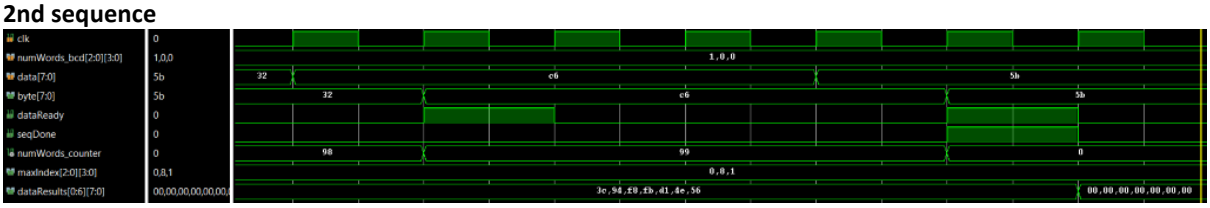
Results of testbench sequences:

1st sequence



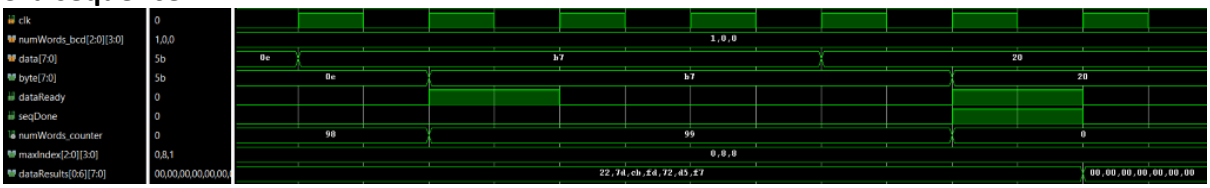
This figure shows the correct results for the 1st sequence which are
maxIndex = 2,2,8 and
dataResults = X"F0", X"7B", X"92", X"FE", X"68", X"39", X"DD "

2nd sequence



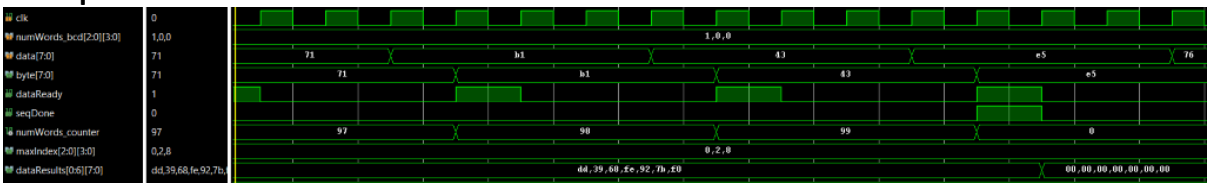
This figure shows the correct results for the 2nd sequence which are
maxIndex = 0,8,1 and
dataResults = X"56", X"4E", X"D1", X"FB", X"F8", X"94", X"3C"

3rd sequence



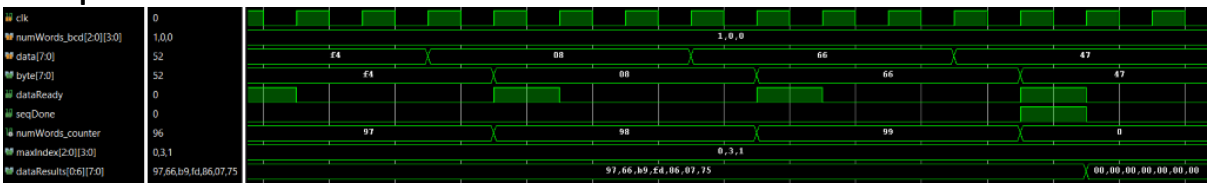
This figure shows the correct results for the 3rd sequence which are
maxIndex = 0,8,8 and
dataResults = X"F7", X"D5", X"72", X"FD", X"CB", X"7D", X"22"

4th sequence



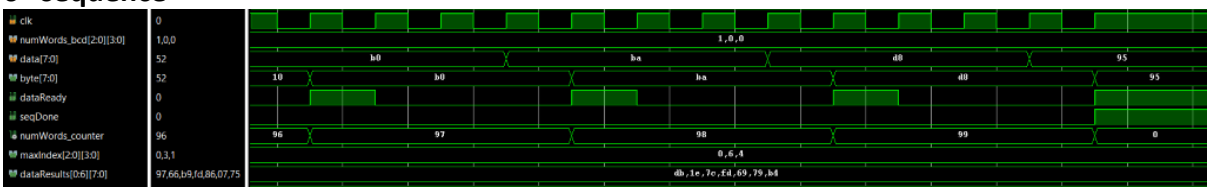
This figure shows the correct results for the 4th sequence which are
maxIndex = 0,2,8 and
dataResults = X"F0", X"7B", X"92", X"FE", X"68", X"39", X"DD"

5th sequence



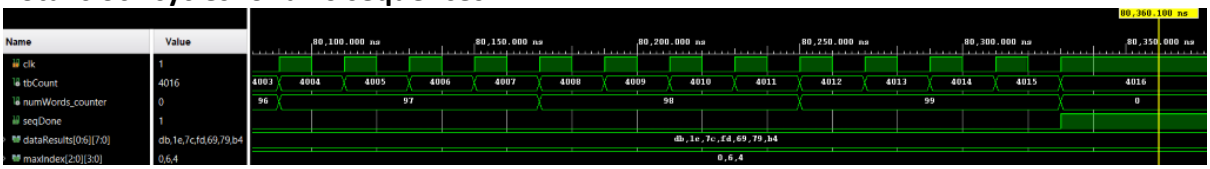
This figure shows the correct results for the 4th sequence which are
maxIndex = 0,3,1 and
dataResults = X"75", X"07", X"86", X"FD", X"B9", X"66", X"97"

6th sequence



This figure shows the correct results for the 6th sequence which are
maxIndex = 0,6,4 and
dataResults = X"B4", X"79", X"69", X"FD", X"7C", X"1E", X"DB"

Total clock cycles for all 6 sequences



max Cycle count is 4500 and our data processor finishes at 4016 clock cycles at around 80. 36µs