Assignment 2 - Peak Detector

Oscar Fuller, Oscar Munday, Will Myers, Reuben Bartley, Frank Gaze

Contents

List of Figures

1 Introduction

1.1 Group Composition:

1) Command Processor
   - *Will Myers*
   -*Oscar Fuller*
   -*Oscar Munday*
2) Data Processor
   -*Reuben Bartley*
   -*Frank Gaze*

1.2 Individual contributions:

Oscar Fuller -
In this project I was responsible for building the shifter required to store the three numbers that follow the initial a/A, and send them to the data processor which correspond the number of bytes needed to be processed. Secondly, I was responsible for receiving the eight-bit data word sent from the data processor in hexadecimal and transmitting it in ascii to the PuTTy Console.

Will Myers -
My contribution to this project was to find an efficient way to format the PuTTY console and detect the end of the byte sequence from the data processor. This required considerable work on the top-level state machine, a look up table to output the different format sequences (multiplexer when synthesized) and a counter for the 'seqDone' signal.

Oscar Munday -
My involvement was in the command processor determining a method to receive the inputted data from the Rx module and check, using a counter, if they formed a correct starting sequence of bytes. Additionally, using a register, I had to make sure each inputted datum was echoed back to the PuTTY terminal.

Reuben Bartley -
Throughout this project I was responsible for the peak detection element of the Data Processor. This required work with the MaxIndex and DataResults signals which communicate directly with the Command Processor. Thus, I was to be greatly responsible for the integration with the Command Processor. This resulted in the usage of multiple registers, counters, and comparators. I was also responsible for making sure the peak detection component would successfully function with the data retrieval section.

Frank Gaze -
In this peak detection project, I was responsible for part of the development of the Data Processor. I was responsible for the Data Processors communications with the Data Generator. This involved integrating the required Two-Phase protocol and working with the 'numwords' and the 'SeqDone' signals.

2 Project Plan

2.1 Group Meetings

We plan to meet every Monday, Tuesday, and Wednesday during our designated lab times. This will provide an opportunity to discuss our project as a group, while also having the opportunity to discuss our design decisions with staff and to receive valuable feedback. Furthermore, starting on week 18 the separate groups will have to organise a weekend session to continue work on their sections. This will aid in meeting our

milestones and objectives. Also, it will mean that any problems encountered can be passed on to staff on Monday which will aid with our project's progression. These weekend sessions will become whole group meetings from week 21 to aid with integration of the processors and to allow for extra assistance on the command processor if required.

2.2 COVID-19 Compromise

Due to the ongoing Covid-19 outbreak, we had to find remote methods to continue to work effectively as a team. This had to allow effective group collaboration, while also avoiding hindering the projects development. This was achieved through weekly meetings held on Zoom (A free video calling platform). Every Sunday we met as a group, which provided an opportunity to discuss our progress on meeting our deadlines and to receive feedback from other members. At the same time, we also organized an additional weekly meeting between our separate groups. This proved to be very effective, as it permitted members to discuss problems in great depths of detail. The productivity resulting from these meetings resulted greatly due to our groups use of GitHub as a safe central storage for our code. This allowed members to review specific sections of code while others discussed the problems they were encountering. This greatly aided in the progression and development of our project.

2.3 Milestones & Objectives

			Week Timetable No.							
TASKS	DURATION (WEEKS)	MEMBERS	16	17	18	19	20	21	22	23
Preparation Phase										
Understanding Vivado	1	All								
Understand System Simulation & Synthesis	2	All								
Data Processor										
Make ASM Chart	1	Frank & Reuben								
Create Data Retrieval	3	Frank. G								
Create Peak Detection	3	Reuben. B								
Combine sections	1	Frank & Reuben								
Command Processor										
Make ASM Chart	1	A.O								
Data Echoing (ANNN)	2	Oscar. M								
Communication with Data Processor	3	Oscar. F								
Formatting Putty Terminal	4	Will. M								
Outputting Byte	3	Oscar. F								
Checking SeqDone	3	Will. M & Oscar. M								
Report										
Data Processor	3	Frank & Reuben								
Command Processor	3	Will Meyers, Oscar. F and Oscar. M								
Integration	2	All								

Figure 1: Gantt Chart

We decided to designate the first week to allowing ourselves to get comfortable with Vivado. At the same time, we also decided we would all start to investigate the full system simulation and synthesis. This is because it would help us later in the integration phase when the two processors are combined. After the second week, we will split into our separate groups and begin work on our ASM charts. Next, the data processor team will have until the end of week 20 to finish their code. This will then allow them to begin work on the report while also aiding with the design of the command processor if required. The objective is to finish the control processor by the end of week 21. However, if this milestone is not completed, then we are prepared to continue working on it in week 22. This will then leave us with a final week to finish the group report and complete the integration of the two processors to complete the full system. These objectives and milestones can be observed in more detail in the Gantt chart above.

3 Command Processor

3.1 Echoing ANNN into PuTTY Console - Oscar Munday

Top Level State Transitions: INIT – CORRECT_WORD
This section dictates the initialisation of the command processor and how it waits for a correct sequence of starting inputs while echoing all inputs back to the PuTTY terminal. In the first state INIT, all value-holders are set to default starting values before moving to the next state, RXNOW_WAIT.
The command processor waits for the rxNow line to go high for one clock cyle – signaling an input byte is ready to be received from the Rx module – before moving to the next state. In COUNT_CHECK the byte is registered. The current numerical value of the counter indicates how many bytes of the input sequence 'aNNN'/ 'ANNN' have been received so far. If the counter is on 0, the processor is still checking for the first byte of the sequence and so the next state is A_CHECK. If the counter is greater than 0, the processor must be checking for a number and so the next state is NUM_CHECK.
In the state A_CHECK, the byte is checked to be the start command: ASCII-code for 'a' or 'A'. If it is not, then the next state will be ERROR; the processor returns to INIT where it is reset, and the above process repeated. If it is, then the processor moves to CORRECT_WORD. Additionally, the txnow line is set high for 1 clock cycle signaling to the Tx module that a byte has been registered and is ready to be received. In turn, the module sends it to the computer, thus the input data is echoed in the PuTTY terminal.
In NUM_CHECK, the input byte is checked to be an ASCII-coded number from 0-9. Similarly to when in A_CHECK, if it is not then the next state will be ERROR and the processor is reset, but if it is then the next state is CORRECT_WORD. As before, txnow is set high to echo the registered byte back to the PuTTY terminal.
In CORRECT_WORD the rxdone line is set high for a clock cycle signaling that the byte has been successfully read from the Rx module and the processor is ready for the next byte. The counter is set to increment by 1 in the next clock cycle to indicate a valid byte of the sequence has been received. At the same time, the counter is checked and if it is currently 0, the processor returns to RXNOW_WAIT. If not, the current byte is a number and so the processor moves to the state SHIFTER. Afterwards it returns to RXNOW_WAIT unless the sequence is complete. Functionality of the SHIFTER state is explained in the next section.

Components required: Output Register
The 8-bit output register stores a byte of data ready to be received by the Tx module, and so be sent to the PuTTY terminal.
D, an 8-bit value-holder, has a value assigned to it. At the same time, load is set high which enables D to be assigned to Q but only on the rising edge of the next clock cycle. Therefore, Q is a clocked value-holder and allows for stable data storage because it can only maintain a single value per clock cycle. In this way, bytes are registered in Q ready for transmission to the Tx module.
If regreset is set high, Q is asynchronously assigned to a default value '11111111'.

Components required: Counter
The 8-bit counter allows the command processor to track how much of a valid input sequence 'aNNN'/ 'ANNN' has been received so far. When an input byte has been successfully checked and is correct, the counter counts it. The counter is controlled with en0 which enables or inhibits incrementation when high or low, respectively.
Within the counter module, its next value is an incrementation of 1 when enable is high. The counter's next value is the same when enable is low. The counter is assigned to its next value only on a rising clock edge, i.e. the counter is clocked, thus is a stable value-holder. In turn, the change in the counter's value incurs the reevaluation of its next value which will be an incrementation if enable is high, or the same value if enable is low.
The counter module has a synchronous reset where it is assigned a default value '000000'. The signal rst_to_4 sets the counter value to '000100' on the rising edge of a clock cycle when high – used in the formatting section later.

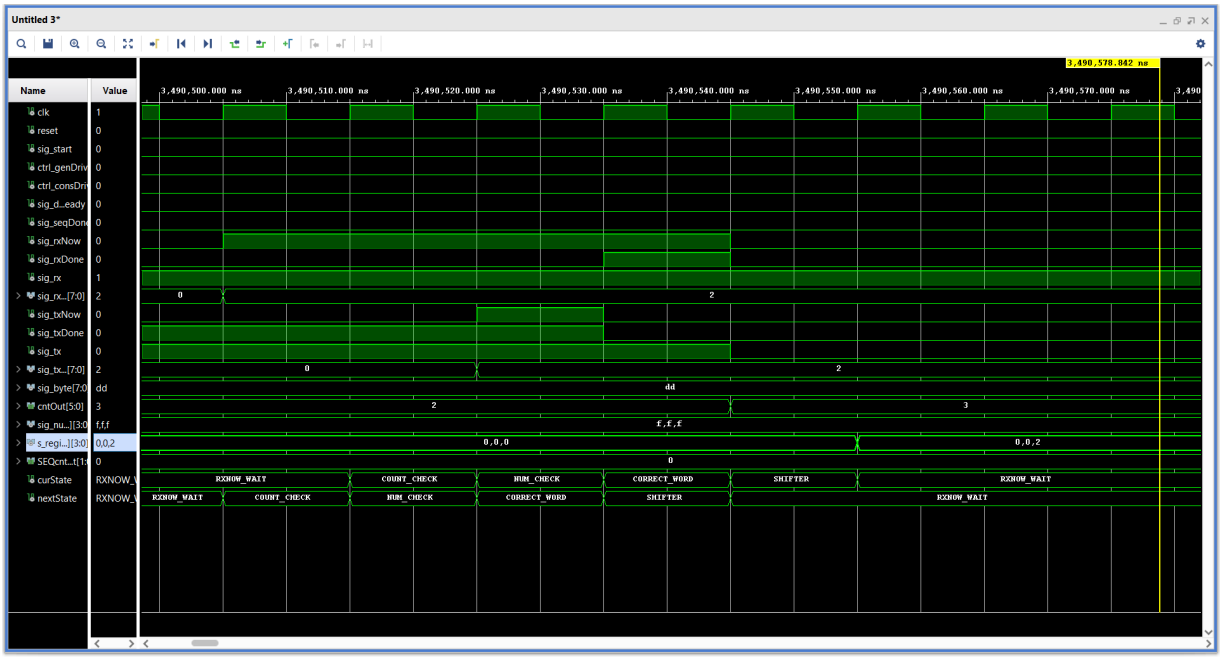

Figure 2: Simulation of ANNN and loading register

3.2 NNN TO Numwords - Oscar Fuller

Top Level State Transitions: CORRECT_WORD – WAIT_SHIFT
In this section a shifter is used to store the numbers coming from the rxData that will become numWords_BCD.
When in the state CORRECT_WORD, if the counter is on 1, 2 or 3, the data coming in is a number required in numWords_BCD so the next state will be SHIFTER.
In the SHIFTER state the shifter is enabled and the number will be loaded into the shifter. This state also checks the counter again to see if it has a value of greater than or equal to 4, this signifies the shifter having all three numbers required. If this is true it will proceed to the WAIT_SHIFT state, if not, the command processor will go back to the RXNOW_WAIT state.
When in the WAIT_SHIFT state load1 goes high to output the numbers stored in the shifter in a BCD array in the signal numWords_BCD to the data processor. This state also waits for txdone to go high to signify the transmitter has finished outputting the final N from NNN and the command processor can proceed to the FORMAT1 state. Also, the counter is reset.

Components required: Shifter
In order to store the numbers sent by Rx so that they could be converted to BCD and sent through numwords, we decided to use a shift register. This shifter will take in three ascii numbers from rxData (after the initial A/a) and then transmit all three numbers in binary coded decimal to the data processor.
The shifter will be made up of three separate registers each corresponding to the significance of the number. Once the shifter is enabled it will store the four least significant bits of the eight-bit ascii character provided into the first register, this is because for all the numbers in ascii only the four least significant bits show the value of the number.
When the shifter is enabled a second time the four bits will be shifted up into the next more significant register and a new four bits will be saved. This carries on until all three numbers are saved.

When load_shift goes high the shifter will output the three numbers in an array of three binary coded decimal numbers.

3.3 Formatting PuTTY Console - Will Myers

After echoing ANNN from Rx back to the PuTTY console, we decided that we should display the retrieved bytes on a new line and have spaces between each byte transmitted. Also, because there is a maximum of 999 bytes that could be outputted, we included functionality to start a new line in PuTTY every 20 bytes to aid in the readability of the displayed data.

Top Level State Transitions: FORMAT1– SEQ_CHECK
The sequence of symbols outputted after the ANNN is:
"SPACE, =, =, SPACE, \, r, \, n, SPACE". As previously mentioned, the WAIT_SHIFT state resets the counter so that we can use the counter to cycle through the look up table.
State FORMAT1 acts as a buffer state to allow the clock driven format look up table time to load the correct value of "000000" at its input and output the correct symbol in FORMAT2. Enable is also set high in FORMAT1 to allow the FSM to iterate through the table.
The next state is FORMAT2 where the output of the format multiplexer is loaded into the output register. This is then sent out of the transmitter modules in FORMAT3.
After this, the command processor waits for the transmitter to finish sending the symbol by waiting until txdone is set high in WAIT_FORMAT.
The next state is FORMAT_CHECK where it loops back to FORMAT1 to output a new symbol, unless cnt0Out is greater than or equal to 8, at which point the final SPACE has been outputted and the controller moves on to start outputting bytes. Please note the remaining decision blocks in the ASM chart during this state will be explained in the last section.
As explained in the following section, the FSM loops back to FORMAT1 after the second digit from byte has been outputted in TRANSMIT2. This causes it to go through the same process, increasing the count and outputting a space between the byte. This is because when the format table receives a number greater than 8 at the address, it outputs a space.
In the next state SEQ_CHECK, the counter is checked again and if it equals 29, then 20 bytes would've been outputted and the processor moves to NEW_LINE (when first byte is outputted the count is 9, and it is incremented with each space between bytes).
In NEW_LINE, reset_to_4 is set high which sets the counter to 4. This results in the sequence "\, r, \, n" being outputted in PuTTY, starting a new line to output the next 20 bits. If the count doesn't equal 29 then the next state equals START1 and a new byte from the data processor will be received. Note the remaining decision block in the ASM chart during this state will be explained in the last section of the command processor.

Components required: FomatMUX
This look up table is used to output various sequences of symbols to format the PuTTY terminal by connecting the address of the table to the counters output. Then, for different counts, the table outputs a different symbol in ascii code.
In the combinational process of the multiplexer, the output is an internal signal q1 instead of formatout. This is then assigned to formatout in the sequential process to ensure that the component is clock driven.

3.4 Outputting Byte - Oscar Fuller

Top Level State Transitions: START1 – TRANSMIT2
In this section the command processor communicates with the data processor and displays the bytes retrieved from the data generator. Once the command processor is in the START1 state it sets the start signal high for a clock cycle, causing the data processor to start retrieving data. Once start is enabled the next state

is DATA_WAIT, this is a wait state that waits for the signal dataready from the data processor to go high, indicating that it has valid data to be read on the byte line.
The next state is WAIT_BYTE. In this state the four most significant bits of the byte signal from the data processor are loaded into the ASCII look up table in the byte multiplexer. This is because the first four bits of byte correspond to the first hexadecimal digit which is then converted into ascii using the table. In the next state LOAD_BYTE, the output of the byte multiplexer is connected to the input of the output register. The registers enable is simultaneously set to high, loading the number across to the output of the register and thus to txData.
In TRANSMIT1 txnow goes high for one clock cycle enabling the transmitter to transmit the ascii coded, hexadecimal digit to the computer.
In the next state called WAIT_TX, the state machine waits for txDone to go high. Also, the remaining four least significant bits of the byte signal are loaded into the byte multiplexer to be converted into ascii code.
Once txDone goes high the command processor goes into the LOAD_BYTE2 state, which has the same function as LOAD_BYTE and then to the final wait state WAIT_TX2, which is the same as WAIT_TX, before looping back to FORMAT1.

Components required: ByteMUX
The byteMux is a multiplexer which converts a four-bit hexadecimal number into ascii code. It functions with each four-bit hexadecimal input having a corresponding eight-bit ascii character. When a four-bit number is inputted for example 1010 which corresponds to an A in hexadecimal the multiplexer will output 01000001, an A in ascii.
Similarly to the functionality of the format multiplexer, in the combinational process of the byte multiplexer, the output is an internal signal q1 is used instead of asciiout. This is then assigned to asciiout in the sequential process.

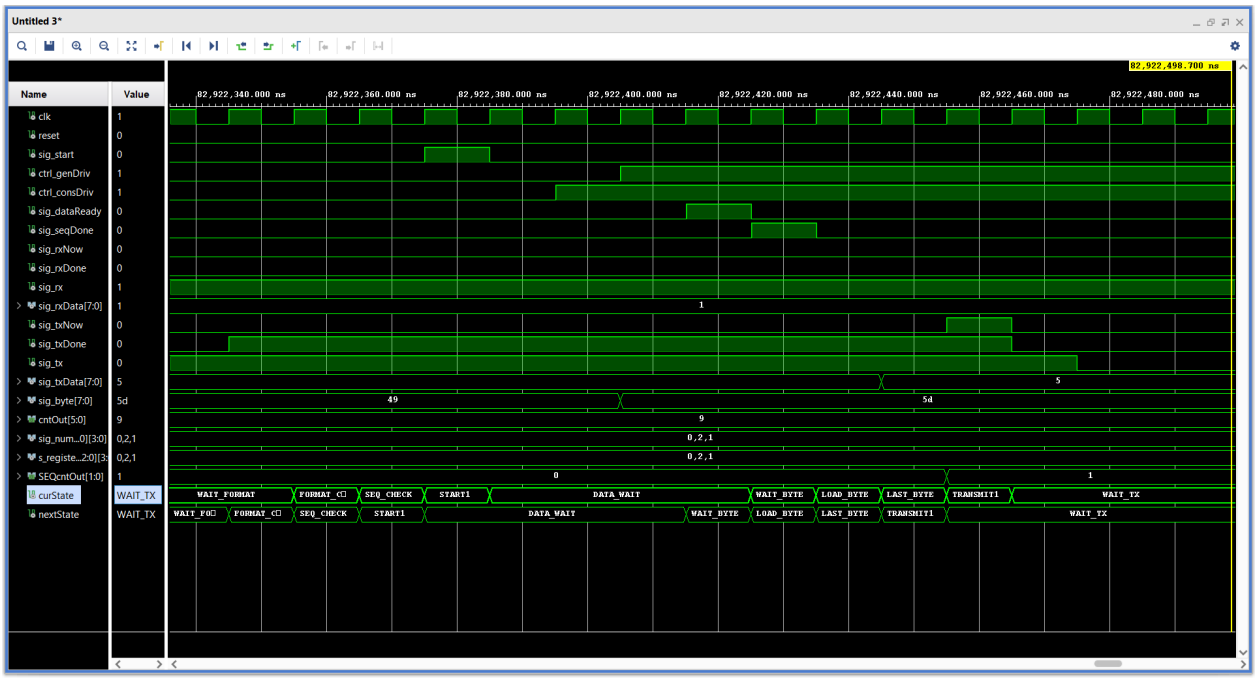

Figure 3: Simulation of receiving seqdone

3.5 Checking for SeqDone– Will Myers

As the data processor sends its last byte to the command processor, it sends the seqDone signal high for just one clock cycle, right after sending dataReady. This therefore requires checking for seqDone and 'remembering' that it has gone high after the last byte has been sent. This 'remembering' will be achieved by using another counter.

Top Level States
While in the LOAD_BYTE the processor also checks if seqDone has gone high in which case the next state will be LAST_BYTE. In LAST_BYTE, en_SeqDone is set high which causes the seqDone counter to count once.
The processor then outputs the final byte and then returns to FORMAT1 before outputting the final SPACE. It then moves to state SEQ_CHECK, where SEQcntOut is checked and if it equals "01", then the next state is FINAL_FORMAT, where the counter is reset to 4 and the seqDone counter counts again to "10". After FINAL_FORMAT, the processor moves to FORMAT1, which begins the final formatting loop to output "\, r, \, n", starting a new line where new commands may be inputted to the console. The FSM finally loops back to the INIT state when the seqDone counter is at "10" and when the main counter is at 8 to signify that the formatting sequence has been completed.

Components required: Seqdone Counter
The architecture of the seqDone counter is the same as that of the main counter except that is only made of two registers as it is only enabled twice.
The byteMux functions when the signal loadDATA goes high, this signifies the data being loaded into the multiplexer and then being outputted on the next clock cycle on the q signal.

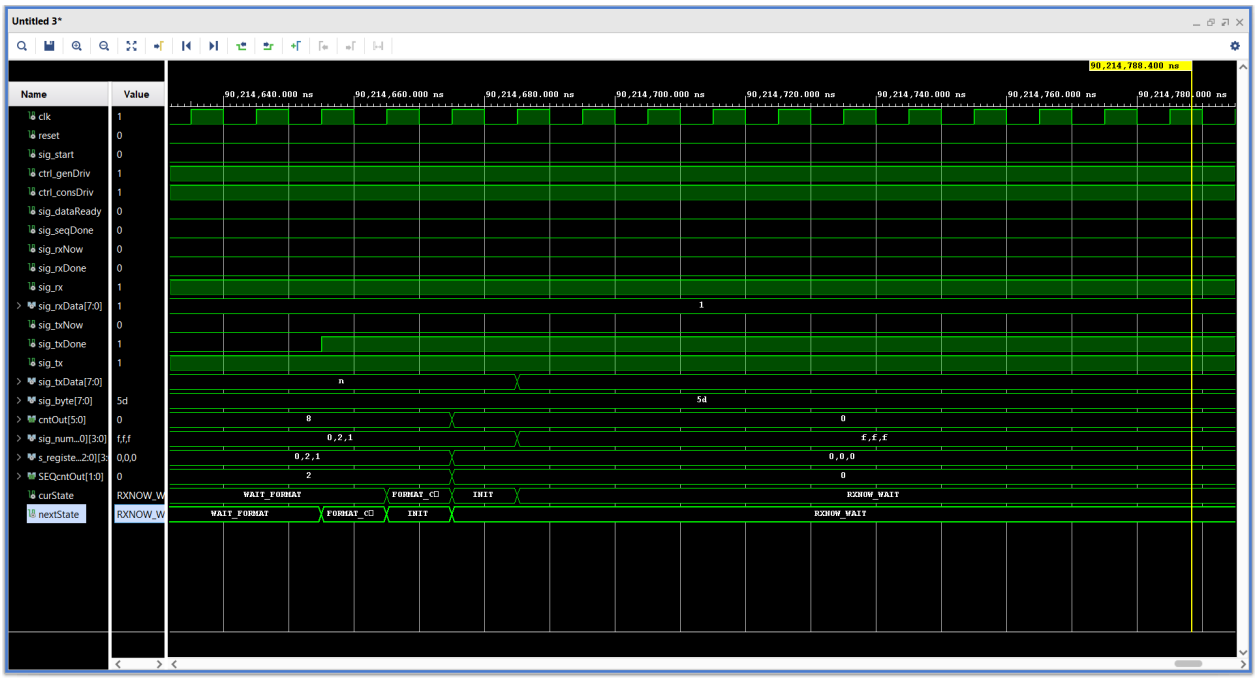

Figure 4: Simulation of formatting

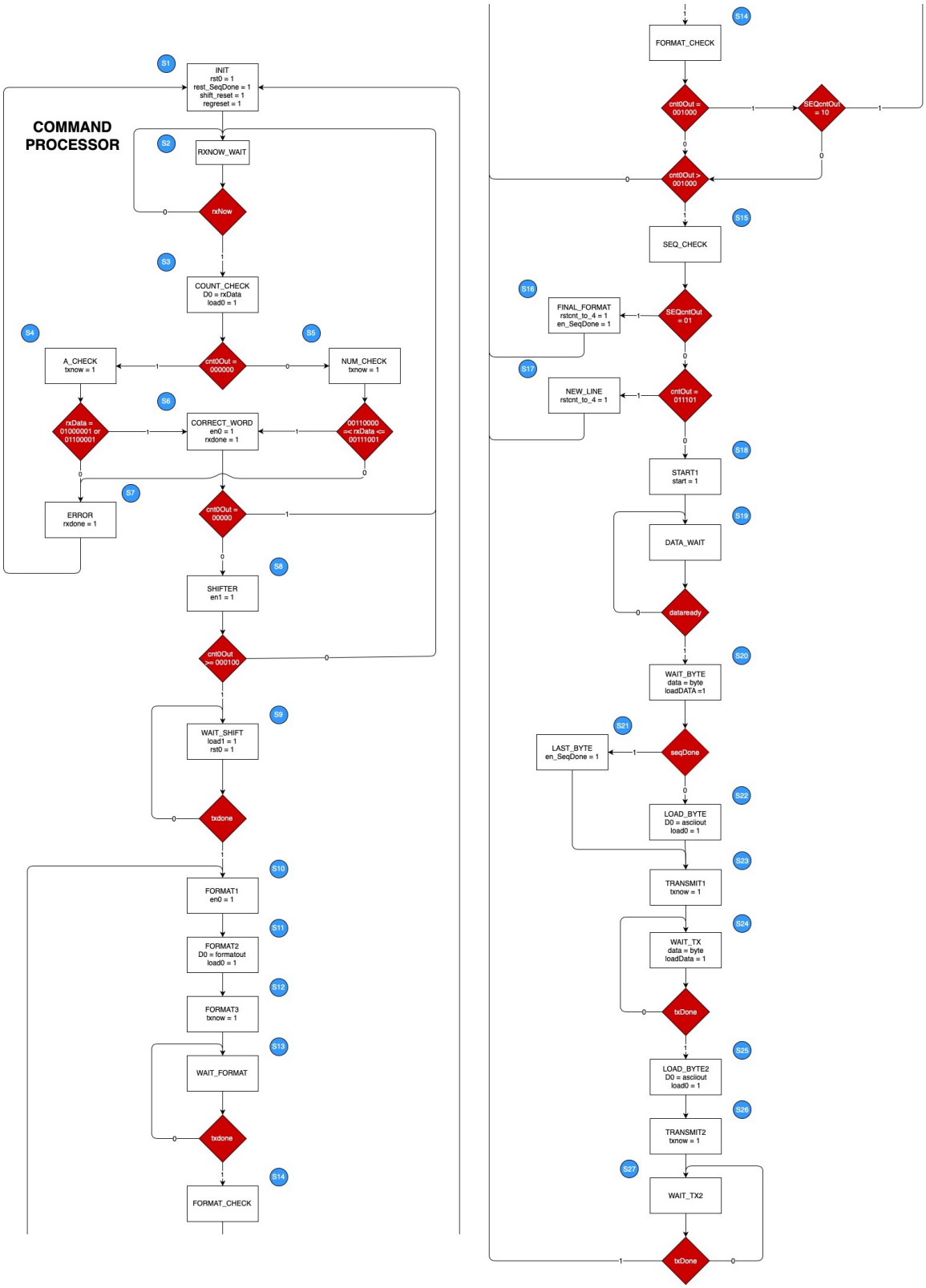

Figure 5: ASM for command processor

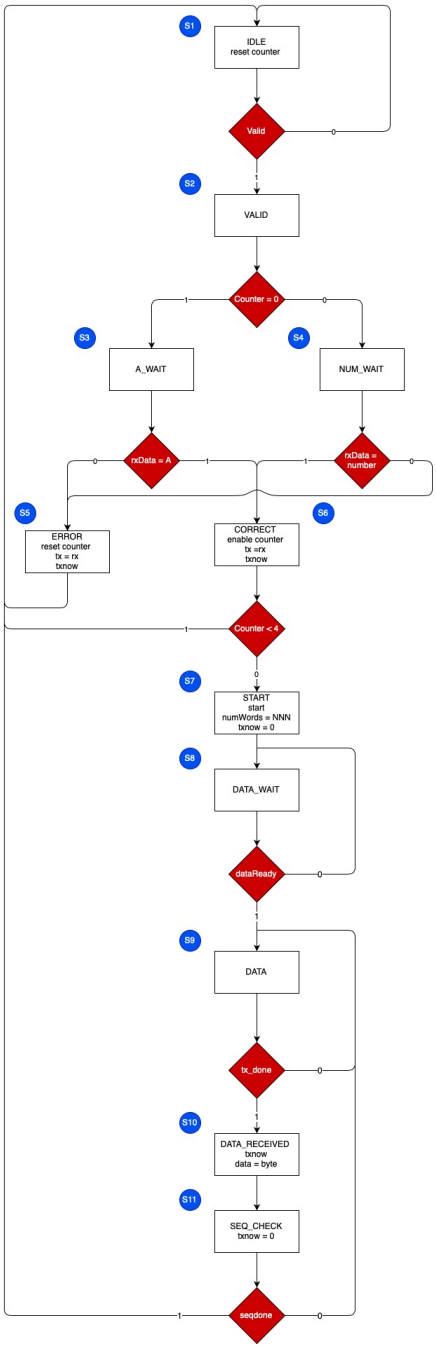

Figure 6: shortened ASM of command processor

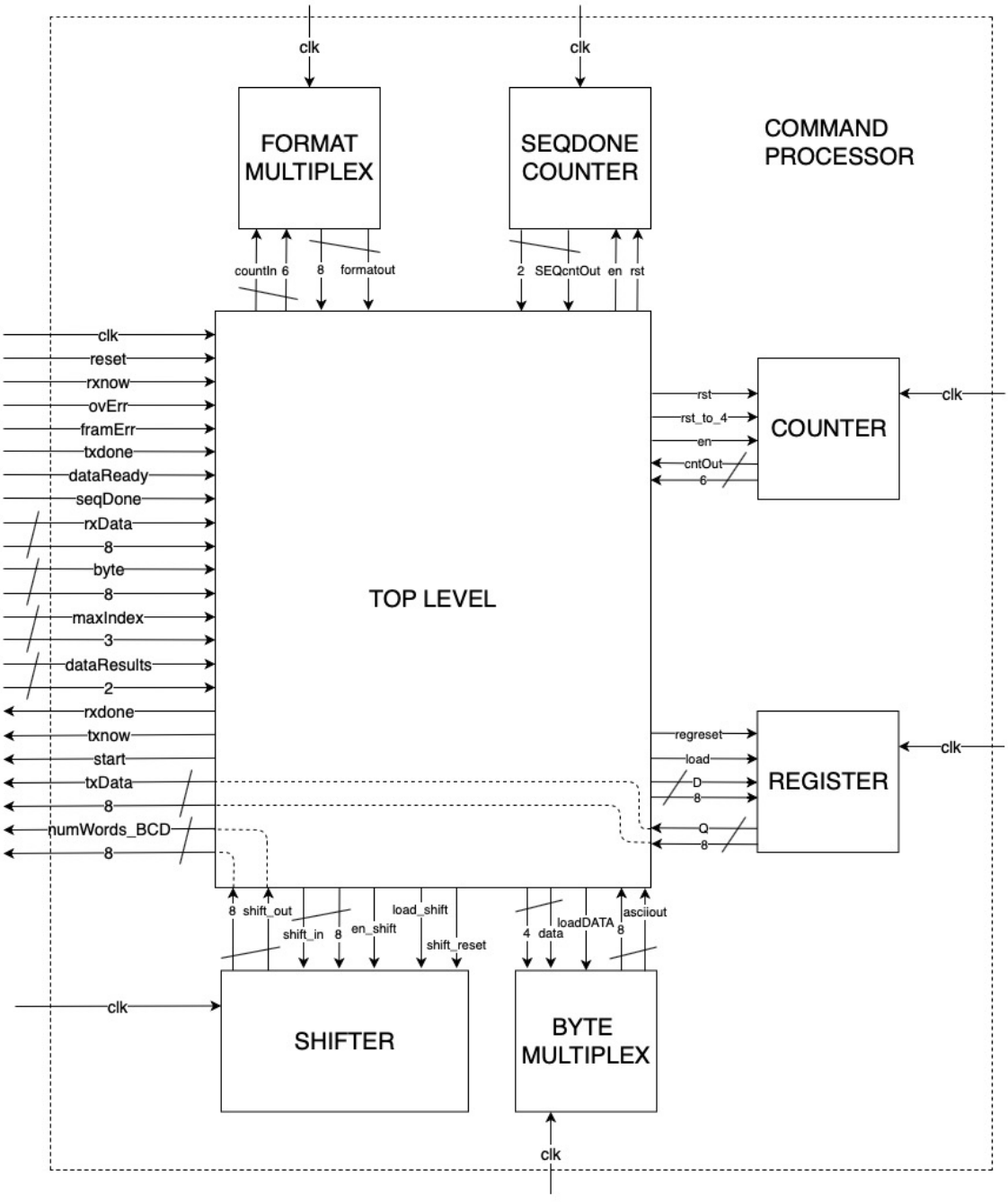

Figure 7: Block level diagram of command processor

4 Data Processor

DIGITAL DESIGN, GROUP PROJECT

ASSIGNMENT 2, DATA PROCESSOR
GROUP 19

Project Start: Mon, 3/9/2020
Display Week: 1

Mar 9, 2020 | Mar 16, 2020 | Mar 23, 2020 | Mar 30, 2020 | Apr 6, 2020

TASK	ASSIGNED TO	START	END
PLANNING			
Understanding Synthesis	Frank & Reuben	3/9/20	3/16/20
Understanding Simulation	Frank & Reuben	3/9/20	3/16/20
Understand DataGen	Frank & Reuben	3/14/20	3/19/20
Make ASM Chart	Frank & Reuben	3/15/20	3/21/20
DATA RETREIVAL			
Make state machine	Frank & Reuben	3/21/20	3/26/20
Build two phase protocol	Frank	3/23/20	3/30/20
Create byte Counter	Frank	3/28/20	3/31/20
NumWords and SeqDone integratio	Frank	3/31/20	4/4/20
Test in simulation	Frank	3/27/20	4/6/20
PEAK DETECTION			
Build shift register	Reuben	3/21/20	3/28/20
Build comparator to find peak	Reuben	3/26/20	3/30/20
Add in a counter for byte storage	Reuben	3/28/20	4/2/20
NumWords and MaxIndex integratio	Reuben	3/30/20	4/5/20
Test in simulation	Reuben	3/28/20	4/7/20
INTEGRATION			
Combine sections	Frank & Reuben	4/7/20	10/04/2020
Comment and organise code	Frank & Reuben	4/8/20	10/04/2021

Figure 8: Gantt chart for data processor

4.1 Data Retrieval – Frank Gaze

Prior to starting our code, we decided to split the Data Processor into two submodules: Data Retrieval and Peak Detection. Two ASM charts were constructed to aid with the coding process and provided a more visual and practical representation of what we wished to implement. The ASM chart for the Data Retrieval submodule is shown below.

Data Gen
To gain information on successfully implementing the two-phase protocol, I devoted time to fully understanding the Data Generator and its single-phase protocol. This provided crucial insight into how to approach changing CtrlOut and also detecting a change in CtrlIn. Especially with the implementation of NOT and XOR gates used in flip flops.

[h!]

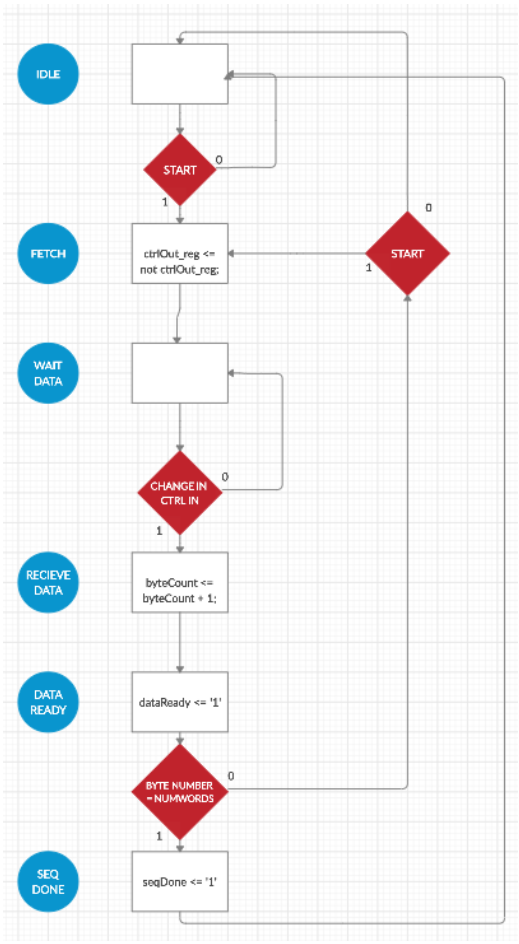

Figure 9: ASM for data retrieval

Two-Phase Protocal

To request a byte from the data generator, a change in CtrlOut was required. This was implemented by using a flip-flop with a NOT gate which affected the CtrlOut signal whenever the system entered the 'FETCH' state. This would only occur after the 'start' signal from the command processor was high on the rising edge of a clock cycle. Else, it would remain in the 'IDLE' state.

Once CtrlOut was changed the system would enter the 'WAIT DATA' state, where it would wait for a change in the CtrlIn signal from the Data Generator. To detect a change in CtrlIn a process was made that assigned the variable of CtrlIn to a new signal on the rising edge of every clock cycle. This new signal and CtrlIn was then used in an XOR gate. If this returned as high, then CtrlIn had changed and meant the data on the data lines was valid and the state machine could enter the 'RECEIVE DATA' state, where the byte counter would increase.

Simulation results

Two-Phase Protocol – The below image shows how 'dataReady' is set high for one clock cycle after a change in 'Ctrl_In' and 'Ctrl_Out 'is detected. This shown at 1,070 ns where both signals have changed from low to high, resulting in dataReady going high. This is again shown at 1,130 ns where both signals have changed from high to low, again resulting in dataReady going high for one full clock cycle.

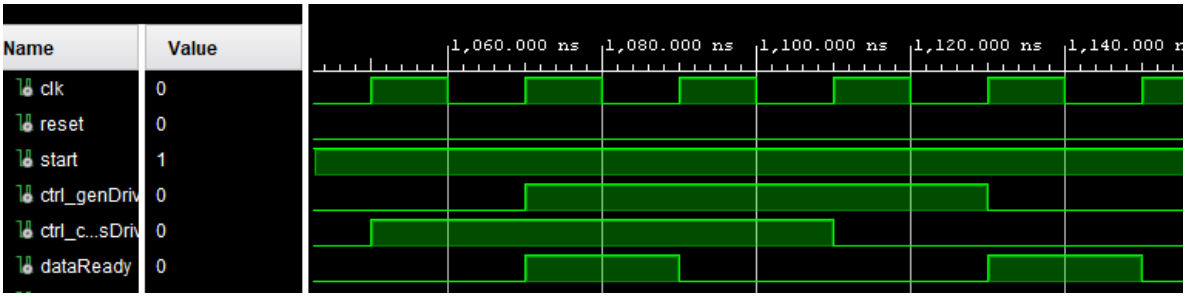

Figure 10: Simulation of two-phase protocol

Byte Counter
A byte counter was important to the implementation of the Data Processor. This is because it is essential in determining when the Peak Detector had reached the end of its given sequence. The byte counter was implemented in a process which was edge triggered and its output varied depending on which state it was currently in. For example, the byte counter would only increase if it entered the process while in the 'RECEIVE DATA' state, as that meant it had just passed through the two phase protocol and had received valid data from the Data Generator. Else, it would either be reset or remain unchanged. Once the machine entered the 'DATA READY' state, the 'dataReady' signal was set high and the data was passed onto the byte lines for the command processor. After this, the system used the byte counters value and a comparator to check if the sequence was complete or not.

Sequence Complete Comparator
A comparator was implemented to detect whether the Peak Detector had finished the given sequence. This was accomplished by comparing the value of the byte counter to that supplied by the 'numwords' signal. This was implemented in an asynchronous process which was triggered by a change in the byte counter. However, to successfully compare the two signals, 'numwords' had to be converted from BCD (Binary Coded Decimal) to an integer. This was performed in a separate asynchronous process which was executed whenever the 'numwords' signal was updated. If the byte counters value was equal to that of which 'numwords' specified, then the machine would enter the 'SEQ DONE' state where the 'SeqDone' signal is set high and 'dataResults' and 'maxIndex' would contain the correct data. However, if the sequence was not complete then the machine would either enter the 'FETCH' or 'IDLE' state depending on the value of the 'start' signal.

4.2 Peak Detector – Reuben Bartley

Peak Detection
A comparator is used to detect whether the most recent byte that will be input into the shift register is greater than the Byte that is currently in middle position of the larger Data Register. If this is the case then the 'peakDetect' signal will go high. A shift register stores the value of the most recent byte and will shift this to the right each time a new byte of data is retrieved for the data generator. Therefore, when 'peakDetect' is high this triggers a process that loads all 4 bytes into the first 4 positions of the dataResults_Reg a register, that is a 7 CHAR Array designed to store the value of the Peak byte in the 3rd CHAR with the 3 bytes that precede and proceed the Peak stored in the first 3 CHAR's and the last 3 CHAR respectively.

Storing the Three Bytes After the Peak
In order to store the 3 bytes after the peak a counter is used. When peakDetect is high this triggers a process, that starts a counter from 0 and counts up each time a new byte of data is stored in the Shift Register. When 'peakCount' = 3 this means that 3 new bytes that proceed the peak byte are now stored inside the shift Register and triggers a process that loads these 3 bytes into the last 3 positions of the dataRegister.

Storing the Peak Values Index
The numWords value is given to the data Processor in BCD and therefore needs to be converted into an integer so that it can be used by the comparator. This is discussed in the previous section. However, the peak value index is required by the command processor in BCD format therefore, an integer to BCD converter was needed. In order to do this each of the different powers needs to be separated from each other and then converted into binary separately. This was done by making use of the modulus and division operators available in the numeric package. Once these were isolated they could be converted to unsigned binary and stored in the corresponding position of the BCD array of Max Index_reg, a register that stores the most recent peak Value index before outputting it back to the Command Processor at the end of the sequence.

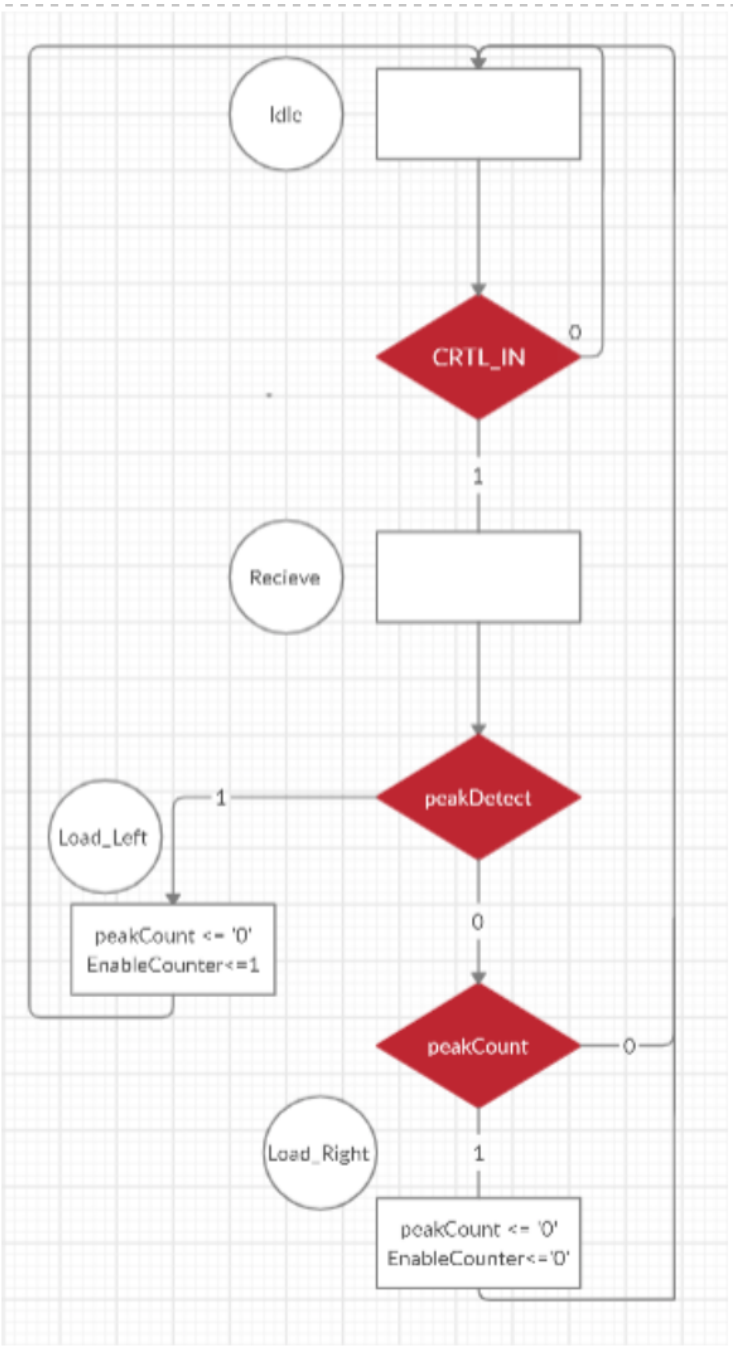

Figure 11: ASM for peak detector