Digital Design Assignment 2

Report – Group 18

1. Group composition and work division

- Group members work distribution:
  - Data Processor - Kulsoom Asif, Alexander Kalchev;
  - Command Processor – Mohamed El Debawy, Yazeed Awad Mohamed, Said Al Kathairi.

- Individual contributions:

- Kulsoom Asif

As part of the data processor developers' team, my contribution to this assignment was to design the processes for the counters, registers and comparator for the data processor. Our design made use of two counters where the primary counter evaluates the number of bytes being fed into the data processor and the secondary counter counts to three so that it can trigger the end of the sequence. I designed three registers for the data processor with the first being a shift register with a serial input and parallel output and the second being the current peak register that stores the peak byte along with three bytes preceding and following it. The third register stores the index of the peak byte. In addition to this, we included a subtractor in our design and its function is to subtract three from the index of the rightmost byte in order to calculate the index of the peak byte. The last process I worked on was the comparator where the current peak is compared against the new middle byte of the shift register. My strategy with the design was to create and test the processes individually with separate testbenches and then implement them together. I found that this approach made debugging easier as I was able to catch errors early in development.

- Alexander Kalchev

My contributions include designing the Finite State Machine logic and implementing it in the Data Processor. I proposed the internal counting to be done in binary for with Binary Coded Decimal conversions before input and output as well as the use of a secondary counter. I also implemented a comparator to signal the FSM logic when the requested number of bytes have been processed, paired up with a value locking mechanism to provide an accurate index counting for the peak without disrupting the state logic after the requested bytes have been counted. In addition, I devised and implemented two small logic blocks which convert the two-phase protocol signals from and to the Data Generator into signals which the FSM logic can operate with. Team meeting organization, work scheduling and the block diagram of the Data processor are also a part of my contributions.

- Mohamed El Debawy

First, I designed the FSM chart, made a list of the processes required to achieve the functionality of the command processor and distributed the tasks among my team. I wrote the code for implementing the state logic and the control signals responsible for successful communication between different entities (Receiver, Transmitter, Data Processor), as well as the code for conversion of byte received from the data processor to ASCII code bytes equivalent to the hexadecimal characters that represent the byte. At the end, I tested and simulated the code, where I identified the source of errors and eliminated them. Also, I collaborated with the data processor team to integrate the code and debug it.

- Yazeed Awad Mohamed

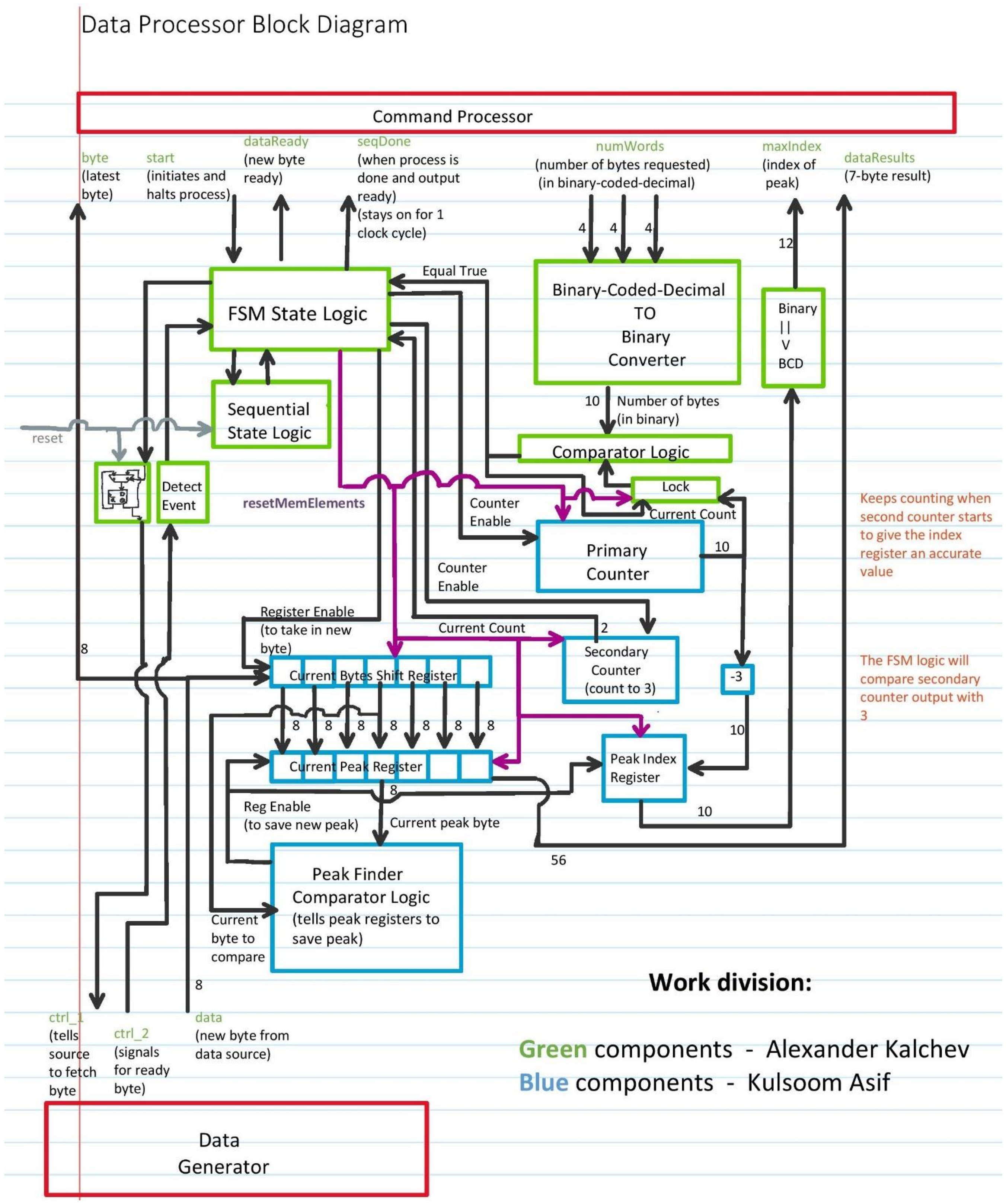
I designed the processes for a 32-bit shift register and a hexadecimal shift register. The 32bit register is used to store the incoming bytes representing the ANNN input. This register will be used as input to another process namely the

ANNN checker. Furthermore, the design of the hexadecimal shift register to convert the bytes from binary to hexadecimal ready to be displayed at the pc's terminal. I was also involved in the testing and debugging of the command processor modules.

- Said Al Kathairi

I was part of the command processing team and I dealt with validating user input according to the ANNN format by creating a process called ANNNvalidate which sends a flag to reflect the command requirements have been met. Moreover, I designed a function that converts the ascii characters into a binary coded decimal (BCD), collecting the 3 bytes NNN into a 12bit word consisting of 3X4bit BCDs. I was also a part of a collective team testing and debugging the command processor functions.

2. Data Processing module architecture



Data Processor Block Diagram
Command Processor
byte (latest byte)
start (initiates and halts process)
dataReady (new byte ready)
seqDone (when process is done and output ready) (stays on for 1 clock cycle)
numWords (number of bytes requested) (in binary-coded-decimal)
maxIndex (index of peak)
dataResults (7-byte result)
4
4
4
12
FSM State Logic
Equal True
Binary-Coded-Decimal TO Binary Converter
Binary || V BCD
Sequential State Logic
reset
Detect Event
resetMemElements
10
Number of bytes (in binary)
Comparator Logic
Lock
Current Count
Counter Enable
Primary Counter
10
Counter Enable
Keeps counting when second counter starts to give the index register an accurate value
Register Enable (to take in new byte)
Current Count
2
Secondary Counter (count to 3)
-3
8
Current Bytes Shift Register
The FSM logic will compare secondary counter output with 3
8
8
8
8
8
8
8
10
Current Peak Register
Peak Index Register
8
Reg Enable (to save new peak)
Current peak byte
10
56
Peak Finder Comparator Logic (tells peak registers to save peak)
Current byte to compare
8
Work division:
Green components - Alexander Kalchev
Blue components - Kulsoom Asif
ctrl_1 (tells source to fetch byte)
ctrl_2 (signals for ready byte)
data (new byte from data source)
Data Generator

- Kulsoom Asif:

In our design we defined 1 byte as a type 'std_logic_vector' consisting of 8 bits.
The processes for the registers and counters monitor the clock's rising edge and enable signal to ensure the timely execution of the sequential statements. They also have a synchronous reset by FSM logic so that each component changes to their default values when triggered.

Shift Register
The process for the shift register has an 8-bit (1-byte) serial input and a 56-bit (7-byte) parallel output. Initially, we set all 56 bits to a default value of all zeros. The middle byte (31 down to 24) is assigned to 'data1' so that it can be fed into the comparator. With every clock cycle, 8 new bits (1 byte) is introduced to the register. In order to accommodate them, the register shifts the existing bits by 8 positions. This shifting results in the middle byte to be unique thus a new byte is fed into the comparator after each shift. It is important to note that the shift convention is to the right in descending order as the latest bits occupies the range 55 down to 48. In addition to 'data1' the register also assigns the current 7-byte sequence to 'Output_Reg' and this is fed into the Current peak register.
The design for this register was inspired by:
https://www.intel.com/content/www/us/en/programmable/quartushelp/13.0/mergedProjects/hdl/vhdl/vhdl_pro_registers.htm

Current Peak Register
The process for the Current Peak register simply stores the current peak byte along with the three previous and three following bytes. It has a 56-bit (7-byte) parallel input. This register monitors the 'Reg2_enable' signal coming from the comparator each rising clock edge and only updates when it receives a positive (1) enable signal. It stores the 56 bits it receives from the shift register until the next positive 'Reg2_enable' signal. The 56 bits are assigned to a signal 'data2' which is fed back into a comparator and the signal 'dataResults'.

Peak Index Register
The Index register stores the index of the peak byte. This register also monitors the 'Reg2_enable' signal at each rising clock edge as the index value is updated when it receives a positive (1) enable signal. The process includes a simple signal assignment where the signal received from 'the Subtractor' is assigned to 'indexReg_out' and then fed to the Binary to BCD converter.

Primary Counter
The process is designed to count the number of bytes being processed by the data processor. We defined a variable 'count' within the process to specify the type (integer) and range (max 1000 bytes) of the output. The primary counter's default value is '0' and when counter is enabled, the count value increases by 1 every clock cycle. At the end of the process, the variable 'count' is assigned to a signal called 'counter1_out'. This signal outputs to 'the Subtractor'.

Secondary Counter
In this counter's process, when the counter is enabled, the count value increases by 1 every clock cycle until it reaches a value of '3'. This is done to shift the last three bytes in the shift register so they can be compared as peaks. We defined a variable 'index' within the process to specify the type (integer) and range (max 3) of the output. The final output of the process is 'counter2_out' which is type 'unsigned std_logic_vector' and is fed into the FSM combinational logic for next state.

Subtractor
This combinational statement subtracts four from the primary counter's output to give the index of the current peak. This value is fed into the current peak register. This subtraction is performed in the 'unsigned' type. We found that it was more efficient to implement this action using the 'ieee.numeric_std' package rather than the 'ieee.std_logic_arith' package.

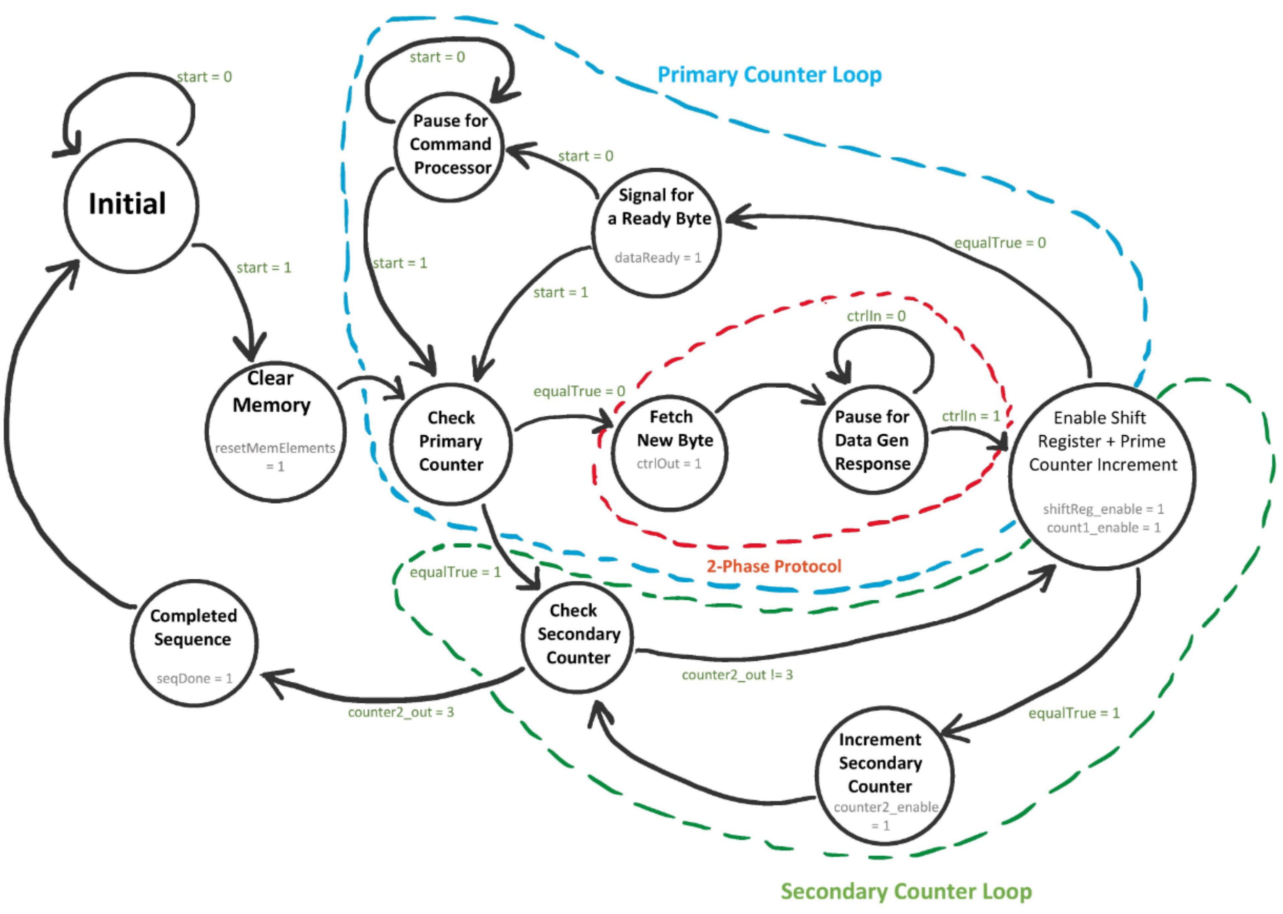
Comparator

This combinational process compares the new middle byte from the shift register and current peak byte from the peak register. Our comparator was inspired by the process that was provided as part of the assignment, however, we defined variables within this process so that we could convert the incoming bytes into type ‘signed std_logic_vector’ as our specification required it. If the new byte is greater than the current peak, then a positive (1) ‘Reg2_enable’ signal is sent out from the comparator to the current peak register so that it can be updated.

to_dataResults

This is a simple combinational process that takes the 56-bit long input (‘data2’) coming from the current peak register and assigns it to the ‘dataResults’ signal. The ‘dataResults’ signal has been defined as a ‘CHAR_ARRAY_TYPE’ type so we use a loop to iterate through the 56-bit input and assign the relevant ‘data2’ bit ranges to the appropriate ‘dataResults’ index.

Data Processor FSM Chart



- Alexander Kalchev

Finite State Machine Logic

In order to justify the state outputs of the FSM logic, their interconnections and relevance to the larger system, the way of operation of the Data Processor needs to be explained. In order to store a byte as a peak, along with the 3 bytes before and after it – bytes which have not been received at the time of receiving the peak byte from the Data Generator, the easiest option was to compare each byte with a three-byte-requests-long lag. In other words, every new requested byte is stored in a shift register that has capacity for seven bytes and the byte in the middle of that sequence is the one that is compared for peak. That means that while the processor receives and passes on a new byte, it also compares the byte that was received three bytes earlier. To ensure that every byte is compared at the end of a sequence, when all necessary bytes have been received, a second loop to the Finite State logic needs to be introduced that shifts the shift register three more times without requesting new bytes, so that each of the last three bytes goes through the middle of the register where it is compared for a potential peak.

Operation

To start with, it is worth noting that this is a Moore state machine: its outputs depend only on its current state. Since the specifications require the processor to keep its data results asserted after the end of a requested sequence and a global reset signal to be able to clear any previous results, the "Initial" state of the state machine is not the one during which the Data Processor clears its memory elements. Instead, the processor waits for sequence request at its Initial state with its old results asserted and the state immediately following it is a "Clear Memory" state, where an internal reset signal is asserted to clear all of the Data Processor's memory elements, except for the 2-phase protocol handling flip-flops.

After that, the state machine goes into its first counting loop where it requests new bytes of the amount specified by the Command Processor. The sequence of events for this loop are as follows: first, the count comparator's output signal is checked. Its job is to compare the current output of the primary counter which counts the processed bytes to the number of requested bytes and output a 'high' 'equalTrue' signal when they match. If it still has bytes left to process, it goes through a state to request a new byte from the Data Generator via 'ctrlOut' and head to a pause state where it waits for a response from it. After receiving confirmation of a new valid byte, the state logic heads to an "Enable Shift Register" state where it enables the shift register to receive the new byte while also incrementing the primary counter to account for the byte. Then, it heads to a separate state to signal the Data Processor to read its new byte through 'dataReady' going high for one clock cycle and depending on the state of 'start', it either heads to another pause state or bypasses it straight onto checking the comparator again.

In the second loop of states, the comparator would indicate that the requested number of bytes has been processed in which case it heads to a state of checking the output of the secondary counter which needs to reach a value of 3 to indicate 3 completed register shifts. If there are bytes to shift, the next state is to enable the shift register. As the state diagram shows, the primary and secondary state loops share the same "Enable Shift Register" state. That is because in both loops the output for that state would be the same so the state can safely be reused with the state machine keeping track of which loop it is in by the state of the 'equalTrue' signal. The register needs to be shifted and the primary counter still needs to be incremented because its output is sampled for the index of those bytes. After shifting and passing through a state to increment the secondary counter it checks its value again and if it indicates the completed shift of three bytes it can go on to end the sequence by asserting 'high' on 'seqDone' and heading to its initial state where it can idle with the results asserted on the appropriate output signals.

2-phase protocol

The combinational state logic itself cannot detect and transmit events in the form of clock-independent signal edges. For that reason, individual components were designed to translate the events of the control signals to and from the Data Generator into 'high' signal states that last for one clock cycle – a format that the state logic can operate with. For the detection of the input signal event, a flip flop stores the signal's value, thus delaying it by a clock cycle and then comparing that stored signal with the original using an XOR gate so that a signal difference in time would be detected and transmitted as a 'high' signal state at least until the next clock cycle. This implementation was largely borrowed from the instructional videos on the topic on blackboard and the provided code for the Data Generator.

The output signal, however, is a 2 to 1 multiplexer, the output of which is connected to the input of a flip flop, its output and NOT output connected to the multiplexer's inputs. The state logic drives the control input of the mux so that when it asserts a '1' on that signal, the multiplexer switches its output to the NOT output of the flip flop, which becomes its new output after the next clock edge. The flip flop's output is our 'ctrlOut' signal which sees a change of state.

BCD to binary to BCD conversions

A decision was made to handle all signals related to counting internally in binary format instead of Binary Coded Decimal. That makes for a less complex implementation for the primary counter and simpler implementation of arithmetic operations where the number 4 needs to be subtracted from the primary counter's output in order to give an accurate value for the index of the peak. This approach also saves signal lines as the BCD number is stored on 12 bits while its binary equivalent (provided it should not exceed 999) needs only 10 bits. That necessitated two additional components to convert the value of 'numWords' to binary and to convert the value of the peak index from binary to BCD to transmit through 'maxIndex'.

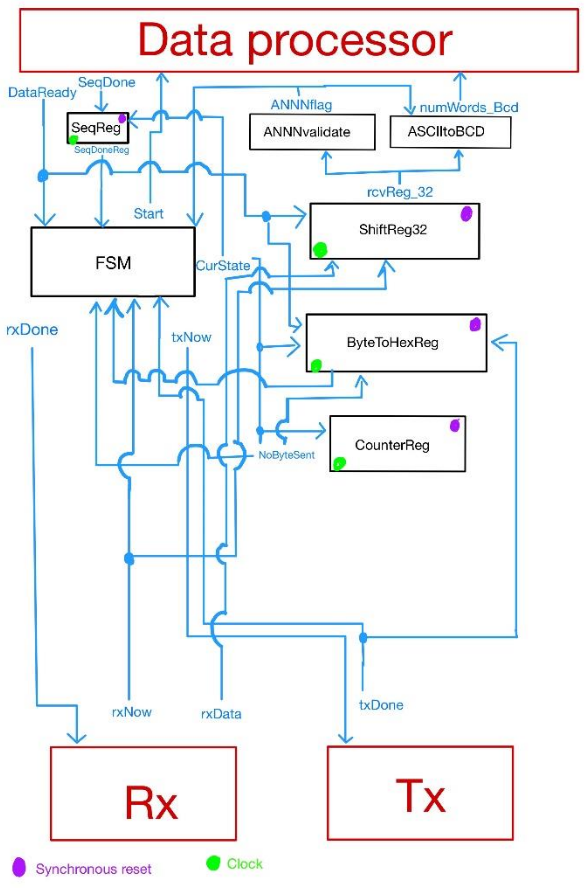
The implementation of a BCD-to-binary conversion process is simple: each of the BCD digits is multiplied by its respective power of 10 (the digit of the hundreds is multiplied by 100, etc.) and the resulting products are summed up to get an unsigned binary result. The binary to BCD converting process, however, was more difficult: it utilizes the “Double Dabble” or “Shift and add 3” algorithm: A sequence of 12 bits is “added” to the left side of the binary number, each four bits representing a BCD digit, and the algorithm loops 12 times. Each time, each of the BCD digits’ value is checked and 3 is added to it if it is 5 or greater and then the whole sequence along with the binary number is shifted to the left by one bit. The result after the loop is our BCD number in the space of the 12 added bits. The implementation of this sub-component was heavily inspired by the Wikipedia article on the algorithm: https://en.wikipedia.org/wiki/Double_dabble

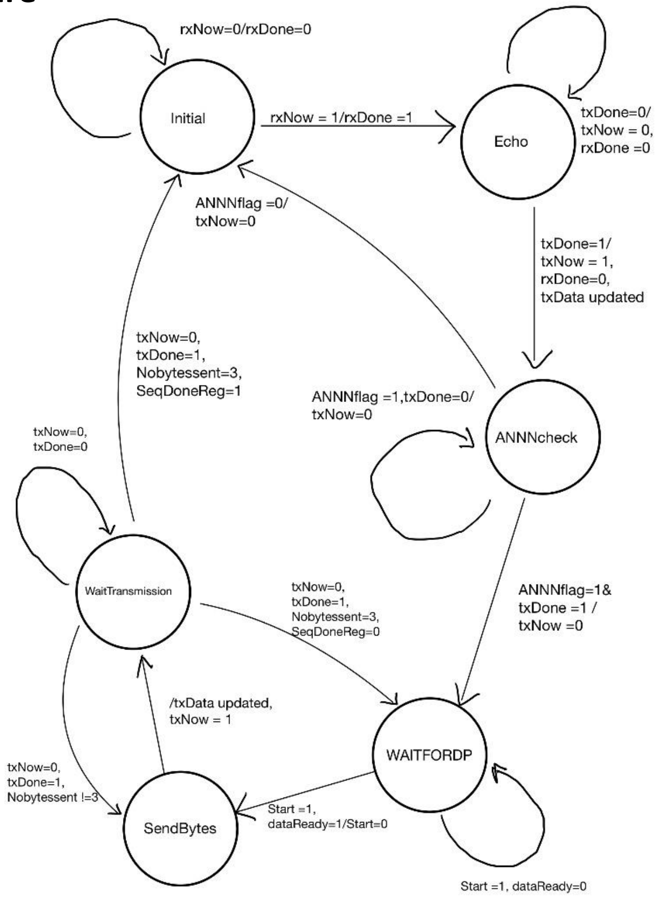
Comparator and Counter Lock

The comparator process compares two std_logic_vector values and outputs a ‘high’ signal ‘equalTrue’ when they are equal. It does so by comparing the bits of the signals one by one and changing the value of a flag variable if a difference is found at any point.

There is an issue with this setup, however: the Finite State Machine logic relies heavily on the value of ‘equalTrue’ to know which state loop it is operating in at any given time. When the comparator finds that the counter’s output has reached the requested number of bytes it will keep its output value at ‘1’. However, while the last three bytes are shifted during the secondary state loop, the primary counter also increases its output value since the index of those bytes comes from it and we need an accurate index if any one of them happens to become our new peak. That would cause the comparator’s output to go to ‘0’ since its inputs are no longer equal and make the FSM logic go back to thinking it still has more bytes to process. The easiest solution would be to lock the output of the comparator when it is asserted ‘high’. The problem with this is that when the components are reset, there could be a point at which the binary version of ‘numWords’ and the primary counter’s output are both 0 before the start of a sequence at which point the comparator may assert a ‘high’ signal for detection of equal values, locking it at ‘high’ for the rest of the sequence. Instead, our implemented solution involves a new “lock” process that operates in the path between the output of the primary counter and the comparator that is essentially just a small register. It locks in place the value of the counter that the comparator receives when ‘equalTrue = 1’, in other words, when the comparator detects matching inputs. This solves the initialization issue since if the “lock” component’s locking function is activated by the comparator seeing 0s at its inputs, it will be immediately released by the binary ‘numWords’ value changing to the actual requested by the Command Processor value.

3. Command Processing module architecture





- Said Al Kathairi :

The process ANNNvalidate is designed to validate the user input and check that it meets the format of ANNN. Only if the user inputs “A” (01000001) or “a” (01100001) followed by three decimal digits according to ascii binary encodings does the process produce a flag confirming the requirements have been met. This functionality is achieved by breaking down the received 32bit word from the shift register into 4 different bytes stored into 4 variables namely A, N1, N2 and N3. Firstly, A must equal either the capital or lower-case representation of its ascii binary representation. Because all ascii decimal digits begin with a 0011 encoding, the ANNNvalidate process check that this format is met to ensure it’s a decimal digit. After checking for the first 0011 the vector of the full byte is broken down into three other variables int1 , int2, int3. These variables are unsigned 4bit vectors which are used to check that each N does not exceed the value of 9 in decimal as it is a requirement that the peak detector does not process more than 999 datapoints, therefore if any of these vectors = 1111 (12 in decimal) the input is invalid. All these requirements must be met for the process to flag a valid input.

ASCIITOBCD, the function of this process is to compress the bytes that represent the three different NNN digits into a smaller 3X4 bit word which will later be combined and passed to the data processor as a 12bit word called “numWords”. Decimal digits in ASCII binary encodings are represented as 0011XXXX where the last 4 bits XXXX represent the decimal value of the character. Therefore, we can break down each individual byte (8bit vector) and take only the last 4 bits as one BCD, for example 00110111 (equivalent to 7) is broken down to just 0111. And hence a 12-bit word containing NNN = 012 can be represented as (0000) (0001) (0010). After splitting the original bytes and concatenating the bits we need together as a new signal, “numWords” is now ready to be used by the data processor.

- Yazeed Mohamed:

A Shift register to store the incoming bytes that represent ANNN with a process having clock in the sensitivity list alongside a set of conditional statements, two IF statements within the process block, one to clear the register if reset is set to high, and one to store the incoming bytes where each letter is represented by a byte of data that comes into the Rx Module from the PC’s UART, a Shift Register to shift and save values in hexadecimal format.

If RxNow is high, and if the reset signal is low, and if the current state is the initial state.

A process is initiated for a shift register to store bytes in hexadecimal form with clock in the sensitivity list with a reset button to clear the values when reset is high.

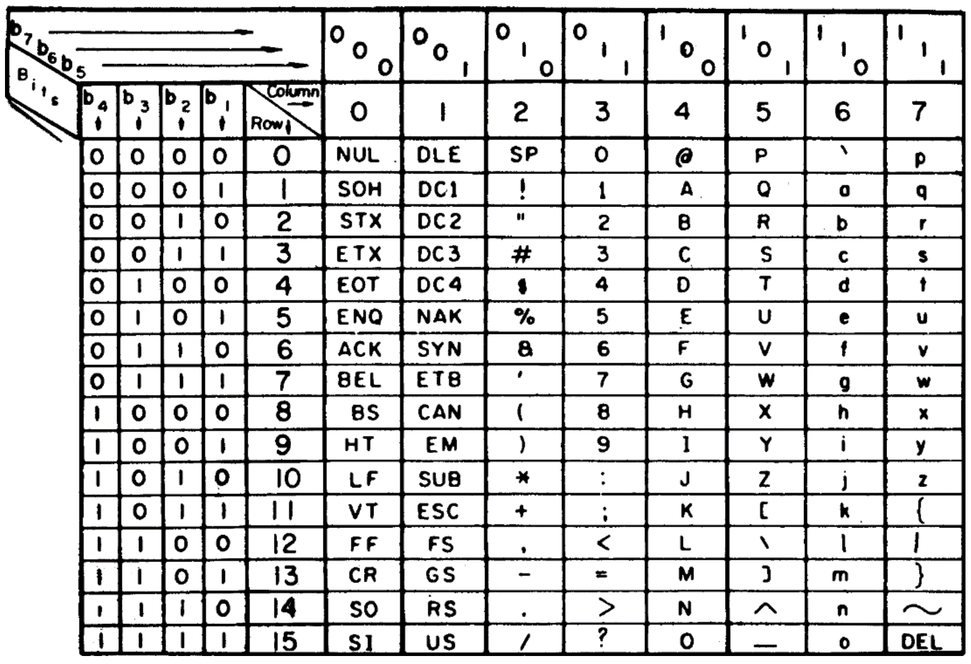
When the DataReady signal between the RxModule and the Command processor is set to High we shift the 32 bit register and add the new byte from the serial line connected to the PC’s UART into the freed space, RxDone is set to high, the byte is “read” and that way we move from the initial state to the “Echo Keystroke” state, if the 32 bit shift register is indeed in the ANNN form we proceed to convert the numerical component of the echoed command “NNN” into a BCD format and we save it to the NNN register switching the Start signal ON/high which takes us to the Next State where we wait for the data processor.

If the shift register however is not in the required form “ANNN” we go back to the initial state to check whether DataReady signal is high in order to check if we have received the required sequence of bytes.

- Mohamed ElDebawy:

I wrote the code for ByteToHexreg block shown in block diagram above. The task was to convert the byte received from the data processor to ASCII code that represents the hexadecimal characters and assign it to the hexreg signal to be able to send it to the PC. To achieve this, the byte received is converted to three bytes two for each nibble and one for a space character to be printed between each word.

USASCII code chart



b_7					0	0	0	0	1	1	1	1
b_6					0	0	1	1	0	0	1	1
b_5					0	1	0	1	0	1	0	1
b_4	b_3	b_2	b_1	Row ↓ / Column →	0	1	2	3	4	5	6	7
0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	x
1	0	0	1	9	HT	EM	)	9	I	Y	i	y
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
1	0	1	1	11	VT	ESC	+	;	K	[	k	{
1	1	0	0	12	FF	FS	,	<	L	\	l	|
1	1	0	1	13	CR	GS	-	=	M	]	m	}
1	1	1	0	14	SO	RS	.	>	N	^	n	~
1	1	1	1	15	SI	US	/	?	O	—	o	DEL

To convert the nibble, if its decimal equivalent is in the range 1 – 9 then, the byte is coded in this format:

0011 + decimal equivalent of the nibble

If the decimal equivalent is greater than 9, byte follows this format:

0100 + (unsigned equivalent - 9)

I derived this logic from the chart based on the observation for both sequences 1 to 9 and A, B, C, D, E, F each sequence has the msb nibble fixed and lsb nibble for each character is higher by one than the character preceded.

Moreover, I added the capability of the process of shifting the HexReg signal to the left by a byte to send the next byte, since txData can only access the msb in HexReg signal, thus shifting is required.

To avoid this shifting and sending process to run continuously, I created the process CounterReg which is a counter for the bytes sent, thus after the third bytes is sent, we return to the data processor requesting the next byte.

Furthermore, in the designing stage, I noted that the signal seqDone is set high for one clock cycle by the data processor. however, when the signal is checked by the command processor the time window of seqDone being set high would have already elapsed as it is checked after a period that exceeds that one clock cycle.

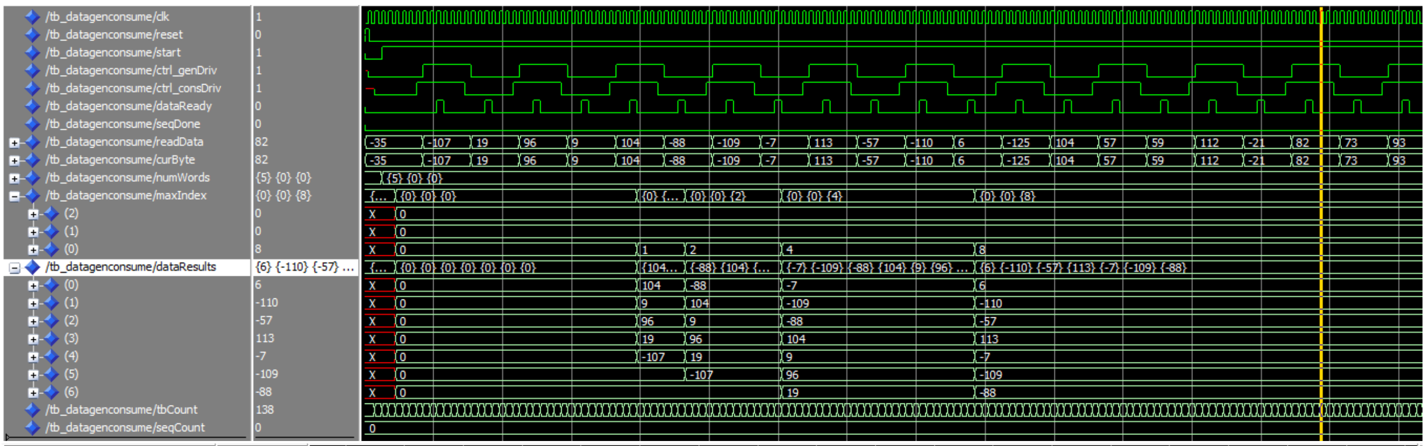
The solution was to add a 1-bit register (flip flop) named SeqDone_Reg that switches its signal (seqDonreg) to high when seqDone is high and reset it after being checked.

Added to that, I wrote cmd_nextstate process, which implements the FSM chart designed to achieve the next state logic and manipulate the control signals (rxdone, txnow, start). The state logic can be defined as trigger and wait logic. To elaborate, it triggers the other components and wait for them to finish and then move to next step.

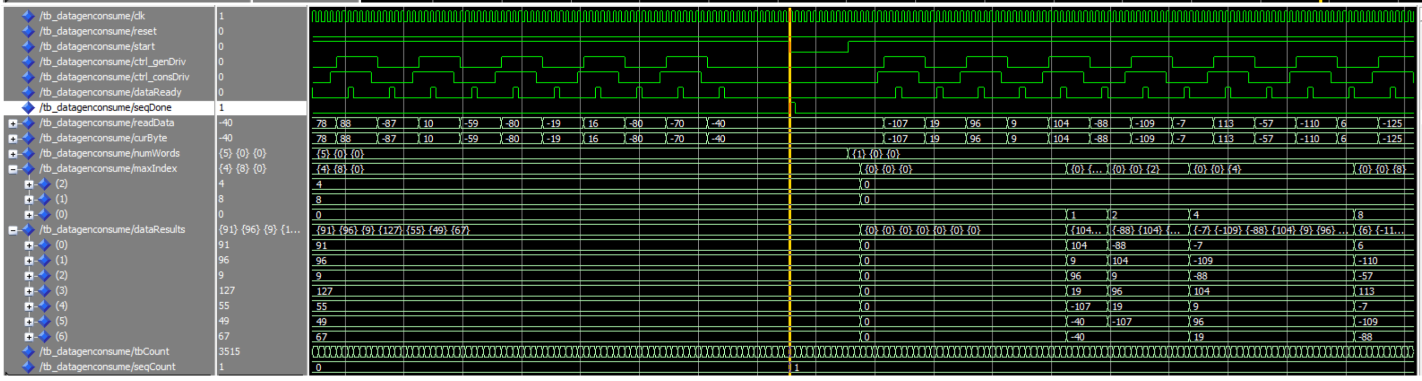
The logic begins with INTIAL state waiting for a character to be typed in. Once a byte is sent by a receiver, we acknowledge receiving the byte by setting rxDone to high allowing the receiver to fetch the next byte typed on the PC and echo this byte to the PC. Then if ANNN or aNNN is detected we move from the receiving stage (INTIAL, ECHO) to the transmitting stage (WAITFORDP, SENDBYTES, WAITFORTRANSMISSION) where the command processor communicates with data processor to request bytes and transmit them, otherwise it returns to the receiving stage. In the transmitting stage the following cycle occurs. In WAITFORDP, a byte is requested from the data processor by setting start signal high. After byte is received and dataReady is high, it moves from the current state where command processor was waiting for the data processor to retrieve the byte to SENDBYTES. Tx is triggered to send a byte in this state. In WAITFORTRANSMISSION, it waits for the byte to be sent. An important thing to note out, is that in every cycle we keep switching between SENDBYTES and WAITFORTRANSMISSION states to send the three bytes in the hexReg signal mentioned above. This cycle stops when seqDone goes high, indicating that requested number of bytes have been processed and the code returns to the INTIAL state.

4. Testing and simulation results

- Data Processor Standalone Simulation



```
/tb_datagenconsume/clk            1
/tb_datagenconsume/reset          0
/tb_datagenconsume/start          1
/tb_datagenconsume/ctrl_genDriv   1
/tb_datagenconsume/ctrl_consDriv  1
/tb_datagenconsume/dataReady      0
/tb_datagenconsume/seqDone        0
/tb_datagenconsume/readData       82
/tb_datagenconsume/curByte        82
/tb_datagenconsume/numWords       {5} {0} {0}
/tb_datagenconsume/maxIndex       {0} {0} {8}
  (2)                             0
  (1)                             0
  (0)                             8
/tb_datagenconsume/dataResults    {6} {-110} {-57} ...
  (0)                             6
  (1)                             -110
  (2)                             -57
  (3)                             113
  (4)                             -7
  (5)                             -109
  (6)                             -88
/tb_datagenconsume/tbCount        138
/tb_datagenconsume/seqCount       0
```

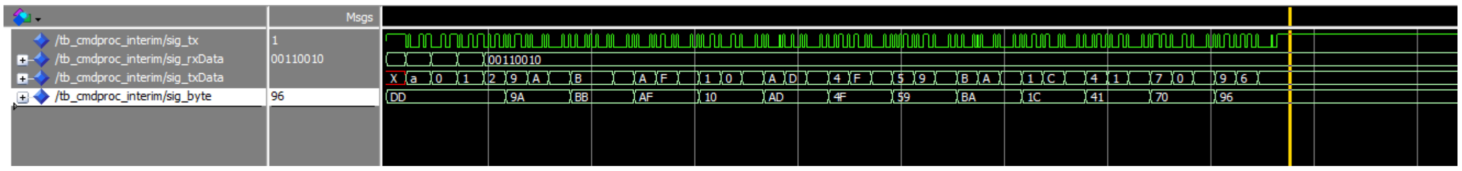


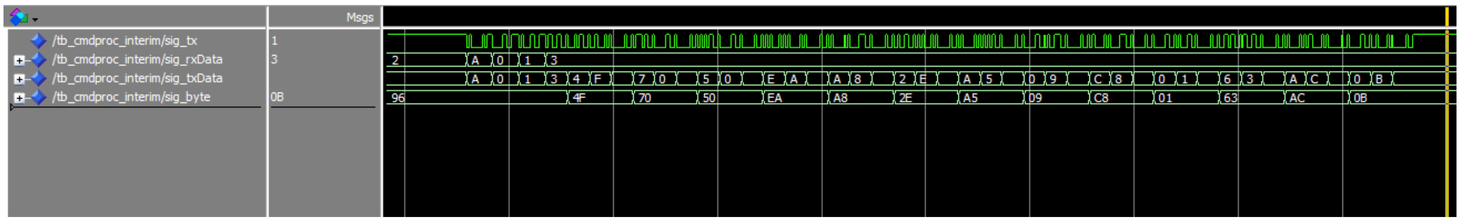
```
/tb_datagenconsume/clk            1
/tb_datagenconsume/reset          0
/tb_datagenconsume/start          0
/tb_datagenconsume/ctrl_genDriv   0
/tb_datagenconsume/ctrl_consDriv  0
/tb_datagenconsume/dataReady      0
/tb_datagenconsume/seqDone        1
/tb_datagenconsume/readData       -40
/tb_datagenconsume/curByte        -40
/tb_datagenconsume/numWords       {5} {0} {0}
/tb_datagenconsume/maxIndex       {4} {8} {0}
  (2)                             4
  (1)                             8
  (0)                             0
/tb_datagenconsume/dataResults    {91} {96} {9} {1...
  (0)                             91
  (1)                             96
  (2)                             9
  (3)                             127
  (4)                             55
  (5)                             49
  (6)                             67
/tb_datagenconsume/tbCount        3515
/tb_datagenconsume/seqCount       1
```

Screenshots of a simulation of the Data Processor using the provided in the Code Archive testbench tb_dataGenConsume.vhd" and Data Generator implementation. We needed to adjust the entity declaration of our component to match the port mappings of the testbench and fix an issue that caused the Data Processor to assign an index value to the peak byte that was one higher than the correct one. Our architecture then started behaving as expected: upon request for a new sequence through the 'start' signal being asserted high, the Data Processor starts requesting new bytes via the 2-phase protocol, and passing them to the Command Processor with 'dataReady' being driven 'high'. After the entire sequence of bytes has been processed and transmitted, and results related to the peak byte are ready at the output signals, 'seqDone' is asserted high to indicate sequence completion.

Important notes: In the first screenshot it can be observed that signals such as 'dataResults' and 'maxIndex' are initialized one clock cycle after the global reset signal has been asserted. That is because internal memory elements are cleared by an internal reset signal tied to an FSM state that follows the initial state. Another thing to note is that our architecture processes a sequence of 500 bytes in 3515 clock cycles, as can be seen on the second simulation screenshot.

- Command Processor Standalone Simulation

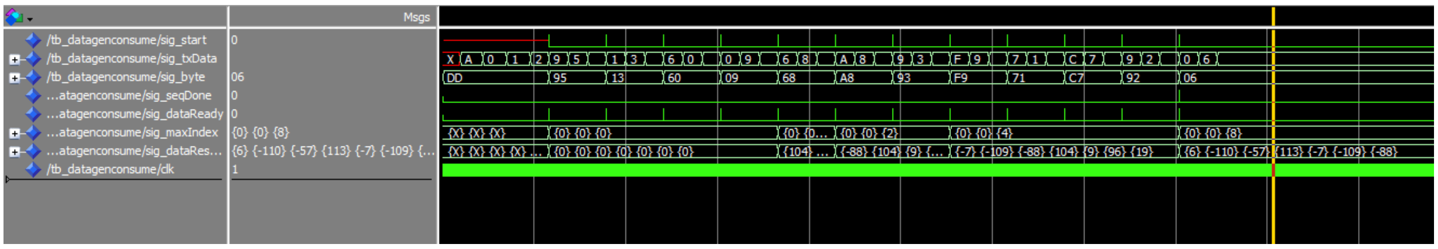

/tb_cmdproc_interim/sig_tx
/tb_cmdproc_interim/sig_rxData
/tb_cmdproc_interim/sig_txData
/tb_cmdproc_interim/sig_byte
Msgs
1
00110010
96

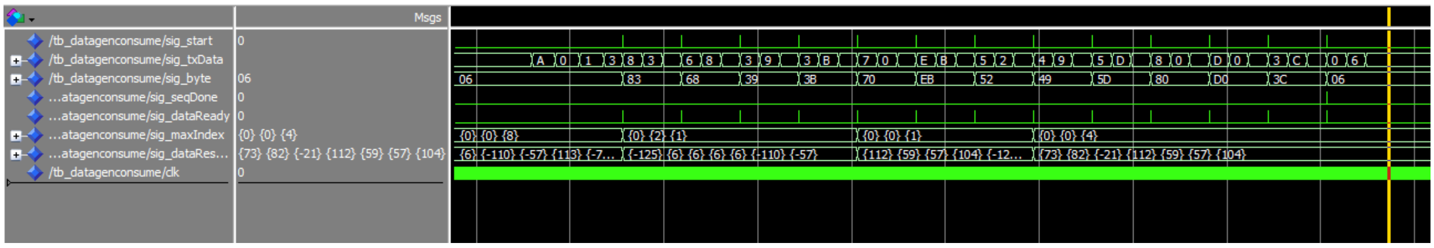

/tb_cmdproc_interim/sig_tx
/tb_cmdproc_interim/sig_rxData
/tb_cmdproc_interim/sig_txData
/tb_cmdproc_interim/sig_byte
Msgs
1
3
0B

```
constant dataSequence : CHAR_ARRAY_TYPE(0 to SEQ_LENGTH-1) := (
      X"9A", X"BB", X"AF", X"10", X"AD", X"4F", X"59", X"BA", X"1C", X"41",
      X"70", X"96", X"4F", X"70", X"50", X"EA", X"A8", X"2E", X"A5", X"09",
      X"C8", X"01", X"63", X"AC", X"0B",
      X"6F", X"2D", X"95", X"31", X"62",
```

As shown in the simulation above the characters typed out by the user are echoed successfully. Furthermore, ANNN is detected appropriately detecting both “a” and “A” as displayed above with a012 and A013 respectively. The decimal digits inputted are evaluated correctly (for example request 12 datapoint and get 12 in return). The bytes from the Data processor are sent in ascii code representing the hexadecimal equivalent of the byte with a space in-between characters (the space is shown as a blank above).

- Complete System Simulation Results


/tb_datagenconsume/sig_start
/tb_datagenconsume/sig_txData
/tb_datagenconsume/sig_byte
...atagenconsume/sig_seqDone
...atagenconsume/sig_dataReady
...atagenconsume/sig_maxIndex
...atagenconsume/sig_dataRes...
/tb_datagenconsume/clk
Msgs
0
06
0
0
{0} {0} {8}
{6} {-110} {-57} {113} {-7} {-109} {...
1


/tb_datagenconsume/sig_start
/tb_datagenconsume/sig_txData
/tb_datagenconsume/sig_byte
...atagenconsume/sig_seqDone
...atagenconsume/sig_dataReady
...atagenconsume/sig_maxIndex
...atagenconsume/sig_dataRes...
/tb_datagenconsume/clk
Msgs
0
06
0
0
{0} {0} {4}
{73} {82} {-21} {112} {59} {57} {104}
0

An issue we encountered with simulating the entire system was that we would reach the first transmitted byte and the process would just halt indefinitely. What was happening was that since the specifications did not describe how long the ‘start’ and ‘dataReady’ signals should be asserted for after the Data Processor transmits a new byte, the Command Processor was designed in such a way that after receiving a ‘high’ ‘dataReady’ signal, it waited for ‘dataReady’ to go ‘low’ again before processing the byte and asserting ‘start=1’ to request a new byte. But the Data Processor was designed to keep ‘dataReady’ high until it receives a new byte request. That necessitated for the modification of one of the components’ state machines so that this loop did not occur. The shown screenshots are from the following working integration of the Data Processor with the Command Processor. As shown, the requested number of bytes are provided by the Data Processor and translated and passed on by the Command Processor.