A technical report on the design and implementation of an external flash memory drive using an FPGA controller

Dominic Spence

170335896

A technical report submitted for the degree of BEng in Electrical and Electronic Engineering

Primary Supervisor: Dr. Alex Bystrov

Secondary Supervisor: Dr. Nick Coleman

21st May 2021

Contents

1. Abstract:

This report explores the theory, design, and implementation of an external flash memory drive, using an FPGA controller. The system allows a user to perform read, write, or erase operations upon a flash memory – as requested by an externally connected PC via an RS232 connection. The result is a is fully functioning system in which a user can enter commands into a serial terminal and perform any of the three flash memory operations stated.

The design of the FPGA controller is comprised of three main components: the RS232 communication interface, the command interpreter, and the flash memory interface. Each of these component blocks are designed as individual VHDL processes, analysed within ModelSim-Altera simulations, and implemented onto an Altera DE1 development board.

The success of this project demonstrates the capabilities for the flash memory to be used as a storage medium for an operating system filesystem.

2. Introduction:

The proposed RS232-to-memory communication interface is designed to allow a user to send, receive or erase blocks of data to or from an externally connected flash memory. This is achieved by sending commands through the serial terminal of a standard PC - connected to an Altera DE1 development board via an RS232 serial port. The DE1 development board used as the hardware for the communication interface contains an on-board RS232 port, a Cyclone II FPGA, and a 4MB flash memory device. The design of the system is coded using VHDL within Quartus II and is synthesized onto the DE1 hardware.

This RS232-to-memory communication interface has the capabilities to be used within many industrial computing systems as it allows a serial device to access data blocks within a memory. By nature, serial devices operate as character devices, meaning data is accessed sequentially byte-by-byte and as such cannot be mounted as a filesystem. By implementing this interface as a block device, an arbitrary filesystem can be created to act as the building blocks of an operating system [1].

This project also has the capabilities to extend to the use of wireless data storage access by implementing an RFCOMM-to-memory interface to allow Bluetooth data transfers instead of through a wired RS232 connection. This was originally planned, however, due to the current climate at the time of writing this, the COVID-19 pandemic has restricted the access of Bluetooth-enabled FPGA boards, thus a wired connection was used. In future developments, this could be easily altered, as the baseline functionality between the RFCOMM and RS232 interfaces are very similar.

2.1 System Architecture and Specification

Figure 1 shows the physical hardware block diagram of the system.

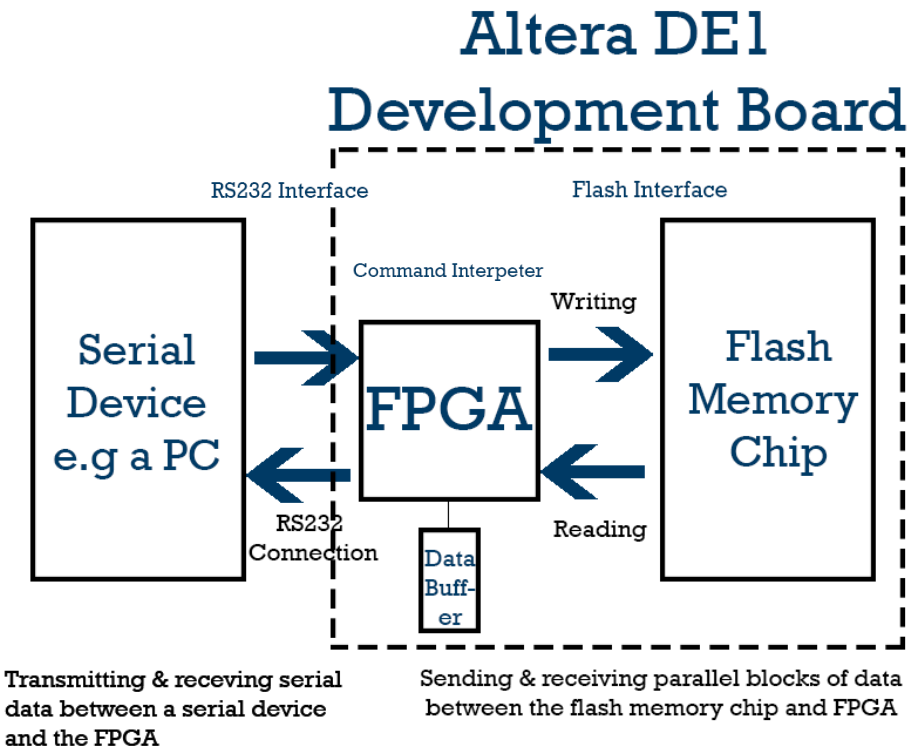


Figure 1: Hardware block diagram of serial-to-memory communication channel

2.1.1 RS232 Interface

The entire system can be broken down into three main sections. The first section is the RS232 interface between the serial device – in this case, a standard PC - and the FPGA. Due to the nature of serial communication, data is transmitted sequentially bit by bit down a single wire. Therefore, for the system to recognise and understand valid bytes of data, an interface is created between the two devices. This is achieved by implementing a UART (Universal Asynchronous Receiver/Transmitter) within the FPGA that drives the RS232 channel. Once this interface is designed, the PC can transmit and receive bytes of data that are used to control the flash operations.

2.1.2 Command Interpreter

Secondly, once the FPGA can receive bytes of data, they need to be manipulated for the system to understand the correct flash mode of operation desired by the user. If a user wanted to read a byte from a certain memory location, the system would need an architecture in place to decipher how the inputs correlate to each various mode of operation. For example, if a user entered “rAAAAA” into the serial terminal, the system needs to know to perform a read operation upon memory location hex AAAAA.

2.1.3 Flash Interface

The final section of the system architecture requires another interface between the FPGA and the flash memory - to store, retrieve or erase the bytes of data requested by the user. This is achieved by creating a VHDL process to understand to results of the command interpreter

and generate the required timing signals that activate the requested mode of operation on the flash memory.

2.2 Aims and Objectives

The aims and objectives for this technical report and project are stated below.

Aims:

- To design and implement an RS232 interface between a serial device and an FPGA that can transmit and receive bytes of data.
- To design and implement a command interpreter that responds to user inputs and selects the requested flash mode of operation.
- To design and implement a flash interface that generates the required timing signals to activate read, write, or erase modes of operation on the flash memory.
- To have a fully functioning communication interface between a serial device and a flash memory that allows users to send, receive or erase blocks of data within the memory.

Objectives:

- Create a VHDL process to emulate an RS232 interface within Quartus II and simulate the design within ModelSim-Altera.
- Implement this VHDL code onto the FPGA and verify the functionality of the interface.
- Create a VHDL program to respond to the incoming bytes and format an architecture to select the correct flash mode of operation.
- Simulate and implement a VHDL code that can successfully perform the three separate read, write and erase flash modes.
- Combine each of the three system architectures into one VHDL program and test the functionality of the entire system.

Further on in this report, the communication protocols and designs used within each section are explained in more detail within the theoretical background section of the report. The implementation and results of these methods are displayed in the results section, and the results are analysed within the discussion section of the report. Finally, the report is concluded by summarising the performance of the fully operational system.

3. Literature review:

The general concepts and theories surrounding this project are mostly well understood, and due to the nature of the unique specification, there is limited research directly relevant to this specific design. The following section is therefore an overview of sources that provide material relevant to the design-specific elements.

3.1 Field Programmable Gate Arrays (FPGAs) and HDLs

FPGAs (Field Programmable Gate Arrays) are semiconductor devices that contain thousands or even millions of individual transistors, each connected by programmable interconnects that form basic configurable logic blocks. These logic blocks and interconnects can be continuously reprogrammed using a hardware description language - such as VHDL or Verilog - to perform complex functions without the need to physically solder wires or components [2]. As mentioned, VHDL is a high-level hardware description language that is written conventionally within software to simulate and program the physical logic blocks, gates, and interconnects of the FPGA [3]. One major benefit to this method of design is that the software simulating the VHDL design will automatically take care of interconnect routing, placement, and timings which eliminates a lot of design analysis [4].

3.2 DE1 Development Board and Quartus II Software

The board used within this project is an Altera DE1 development board which contains a multitude of hardware components that can be configured by synthesizing a VHDL code within the design software called Quartus II. More importantly, the board contains a built-in RS232 port, a Cyclone II FPGA, and a 4MB flash memory [5]. Obtaining and designing on this board means there is no need to buy and solder individual components as each component is already connected to the FPGA. This means that the entire design can be completed within the Quartus II software and synthesized onto the onboard FPGA. The DE1 schematics and signal connections can be found as figure 25 in the appendix.

3.3 Finite State Machines (FSMs)

As with all software development, there are various methods of coding that can be used to perform the same functionality. Although there is no current literature as to how a digital hardware system should be designed using VHDL, a common, efficient method is to use FSMs [6]. FSMs are computational models that are used to control sequential logic within a design - consisting of multiple states and transitions that regulate logical operations [7]. In Moore FSMs commands are executed based upon their current state to control the logical flow of the program and prevent any signal contentions.

3.4 Character and Block Devices

In computing systems, almost every peripheral device is controlled by a code that operates a specific piece of hardware. These codes are called device drivers and are required so the systems knows exactly how to interface and communicate with the device [8]. Typically, the majority of devices are either character devices or block devices. Character devices are generally thought of as devices that transmit data by communicating with the driver as a stream of sequential bytes. Such examples include graphics displays and serial ports – which will be used within this design. Block devices are generally devices that store data and are accessed a block at a time, with the minimum block size of a Linux system consisting of 512

bytes [9]. This project is the design of a driver that can accept inputs from a serial RS232 character device and operate the flash memory as a block device.

3.5 Summary

The review has presented an overview of the theory of low-level computer system architecture and the various protocols and interfaces required to implement an external flash memory drive. The specific hardware and specification have also been reviewed and confirmed to be the most suitable for this design.

4. Theoretical Background:

Figure 2 shows the refined system block diagram, designed to meet the requirements of the specification. The dashed lines represent each of the three components of design. The blue blocks represent each of the VHDL processes that are combined and implemented onto the FPGA.

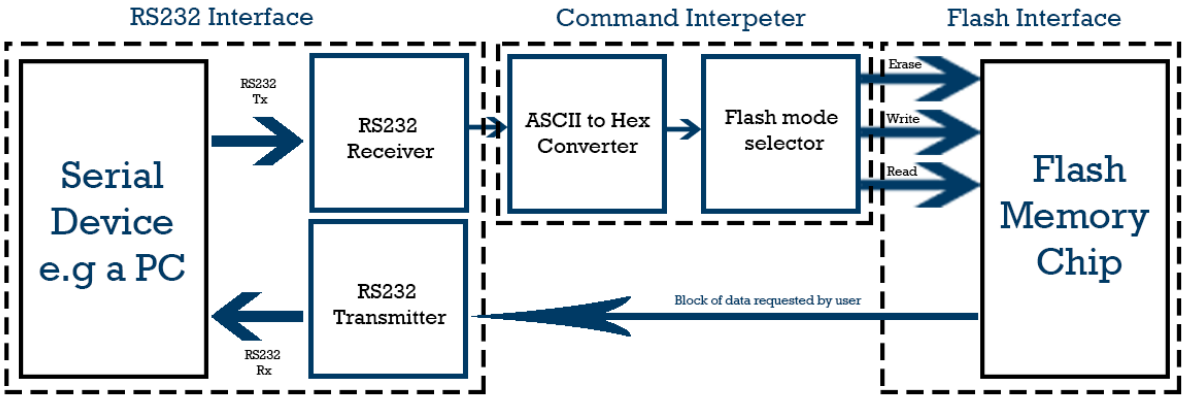


Figure 2: Block diagram of entire system

4.1 RS232 Interface

The first component of the system is an RS232 interface between the serial device and the FPGA. As mentioned, RS232 is a bidirectional asynchronous serial communication interface that sequentially transmits bits of data down a single line to form byte-by-byte transmission of data – at a speed represented by the selected baud rate.

4.1.1 Wire Configuration

For the most basic configuration of a bidirectional RS232 interface, there is a requirement for three wires - Tx, Rx, and GND [10]. These wires represent the transmit, receive, and reference ground signals respectively.

Figure 3: Wiring diagram for the most basic configuration of an RS232 system

4.1.2 RS232 Data Format

As serial communication is often used to transmit data between a variety of different devices, each will have its own clock reference, therefore there is no common clock shared to synchronise the data transfer. For this reason, an asynchronous protocol is used, requiring the stream of data to contain a start and stop bit to act as the synchronization between both devices, meaning that each frame contains of 1 start bit, 8 data bits, and 1 stop bit [11].

4.1.3 RS232 Transmission

When no data is being sent, the line is set at a logic high and idles. When data is to be sent, the line is first pulled down to logic low for 1 bit period to act as the start bit - this informs the receiver that a sequence of data bits is about to follow and allows synchronisation. The data bits are then transmitted sequentially with the least significant bit (LSB) first by setting the line logic high or low, respective to the data bit binary value. Lastly, the line is driven logic high for another bit period to indicate to the receiver that the transmission of the byte has ended. This can be seen in the transmission of ASCII character 'r' in the figure below.

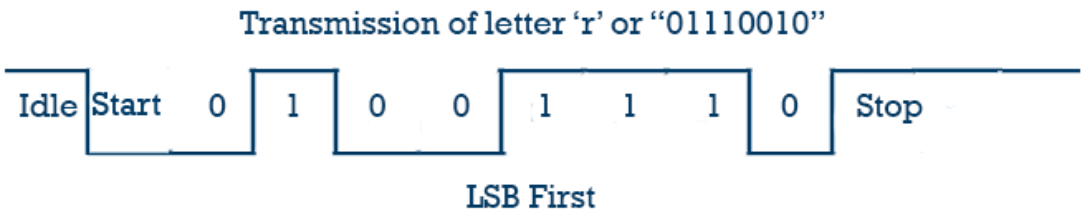


Figure 4: RS232 Transmission of ASCII character 'r'

4.1.4 RS232 Receiver

Similar to the transmitter, when operating as a receiver, the Tx line is monitored and waits for the initial start bit – indicated by a logic low signal for 1 bit period. Once this has been detected, the receiver counts to the midbit of the first data bit and stores the logical state of the line. This is repeated for the remaining 8 data bits and the system returns to idle, waiting for the start of the next byte [12].

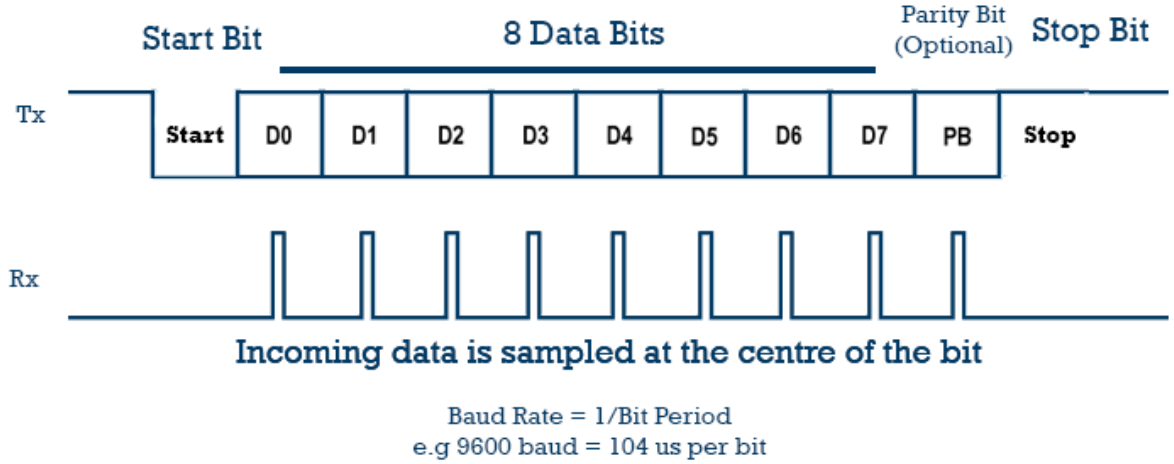


Figure 5: RS232 Data Frame and Receiver

4.1.5 ASCII Characters

For systems to understand transmitted bytes, the 8-bits are represented as ASCII characters. ASCII characters assign every possible letter, number, and character as a stream of 8 binary bits, allowing the possibility of:

$$2^8 = 256\ Combinations \quad \text{(i)}$$

For example, as can be seen in the ASCII table, the character 'r' is assigned the decimal number 114, equivalent to hex x72, or binary "01110010".

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Figure 6: ASCII Character Table [13]

Figure 5 shows a typical waveform diagram for the transmission of character 'r'.

4.1.6 Baud Rate

As mentioned, the baud rate is a parameter that determines the speed of data transfer within serial communication. This rate specifies the number of signal changes that can occur every second. If the two communicating devices have a mismatch in bauds, then they could be sampling the bits at different rates which would result in incorrect data being transmitted or received [14].

$$Bit\ rate\ (b/s) = Baud\ Rate\ (su/s)\ \times Number\ of\ Bits\ per\ Baud \quad \text{(ii)}$$

The DE1 development board used within this project has a baud rate of up to 115200 [5]. As only binary signals are used, the bit rate is equal to the baud rate, meaning 115200 bits can be transmitted every second down the line. This can then be used to find the bit period of a signal, to determine how long the system should set each bit of the signal for. This is dependent on the internal clock used within the device which is a 50MHz oscillator. Thus, the bit period is calculated as:

$$Bit\ period = \frac{Clock\ Frequency(Hz)}{Baud\ Rate\ (bps)} = \frac{50\times10^6}{115200} = 434\ clock\ cycles\ or\ 8.68\mu s \quad \text{(iii)}$$

As each byte of received data requires 10 bits (1 start & stop + 8 data bits), the theoretical maximum data rate can be calculated as:

$$Theoretical\ Maximum\ Data\ Rate\ (bytes/second) = \frac{Bit\ Rate}{Bits\ Transmitted\ per\ Byte} = \frac{115200\ bps}{10} = 11520\ bytes/second \quad \text{(iv)}$$

This is used in the analysis of the results to compare the performance of the communications interface to the theoretical maximum speed.

4.2 Command Interpreter

The second component of the design is the command interpreter. This component is required to format the user inputs and select the correct flash operation. This is done using a finite state machine within the VHDL code that transitions between each flash state, dependent on the user request. Figure 7 shows the pseudocode flowchart for how the command interpreter operates and figure 8 shows the command format that is to be implemented by the user.

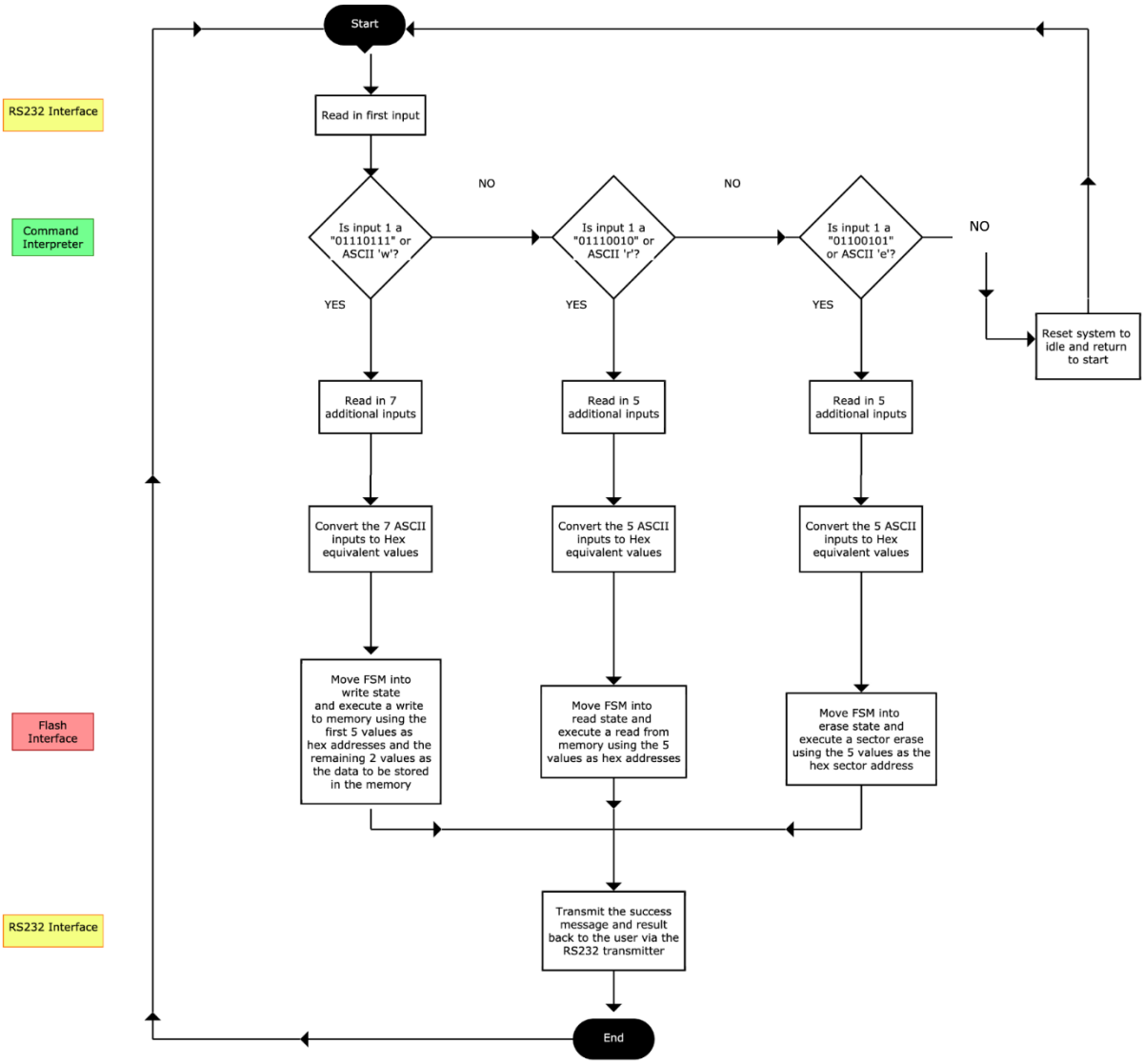


Figure 7: Pseudocode for command interpreter

Command	Input via serial monitor	Explanation of format
Read	‘rXXXXX’	‘r’ followed by 5 hex characters for address to be read from
Erase	‘eXXXXX’	‘e’ followed by 5 hex characters for address of location to erase
Write	‘wXXXXXYY’	‘w’ followed by 5 address characters for address of write location, followed by 2 extra hex characters for the byte of data to be written into memory

Figure 8: Commands to be entered into serial monitor

4.2.1 ASCII to Hex Converter

The bytes of data received by the RS232 interface are not naturally formatted correctly in their ASCII binary state. This is because the flash memory operates using hexadecimal values, not their ASCII equivalents. Therefore, if a user wanted to address location “AAAAA” in the memory, the flash device would expect xAAAAA or “10101010101010101010”. Whereas “AAAAA” entered through the serial interface would transmit the ASCII equivalent of x4141414141. Hence, an ASCII to Hex converter is required within the VHDL code as a subsystem of the command interpreter.

4.3 Flash Interface

As can be seen from the pseudocode flowchart, the system initially checks the user's first input and moves through each branch independently, moving into whichever flash state is requested. To create an interface between the FPGA and the flash memory, VHDL processes are designed within each flash mode state to recreate the signal timing diagrams as per the datasheet requirements. The flash device used is an S29AL032D70TFI04 meaning it runs under speed option 70 in the datasheet tables below.

4.3.1 Flash Memory Operation – Read

To perform a read operation, each individual flash signal seen in figure 9 must be recreated within the timing constraints seen in the table above and the user-requested address is to be set onto the address wires. If these conditions are met, the byte of data within the memory becomes valid at the output of the data bus.

Read Operations

Parameter		Description		Test Setup		Speed Options		Unit
JEDEC	Std					70	90	
t_{AVAV}	t_{RC}	Read Cycle Time (Note 1)			Min	70	90	ns
t_{AVQV}	t_{ACC}	Address to Output Delay		$CE\# = V_{IL}$ $OE\# = V_{IL}$	Max	70	90	ns
t_{ELQV}	t_{CE}	Chip Enable to Output Delay		$OE\# = V_{IL}$	Max	70	90	ns
t_{GLQV}	t_{OE}	Output Enable to Output Delay			Max	30	35	ns
t_{EHQZ}	t_{DF}	Chip Enable to Output High Z (Note 1)			Max	25	30	ns
t_{GHQZ}	t_{DF}	Output Enable to Output High Z (Note 1)			Max	25	30	ns
	t_{OEH}	Output Enable Hold Time (Note 1)	Read		Min	0		ns
			Toggle and Data# Polling		Min	10		ns
t_{AXQX}	t_{OH}	Output Hold Time From Addresses, CE# or OE#, Whichever Occurs First (Note 1)			Min	0		ns

Notes:
1. Not 100% tested.
2. See Figure 12, on page 48 and Table 19 on page 48 for test specifications.

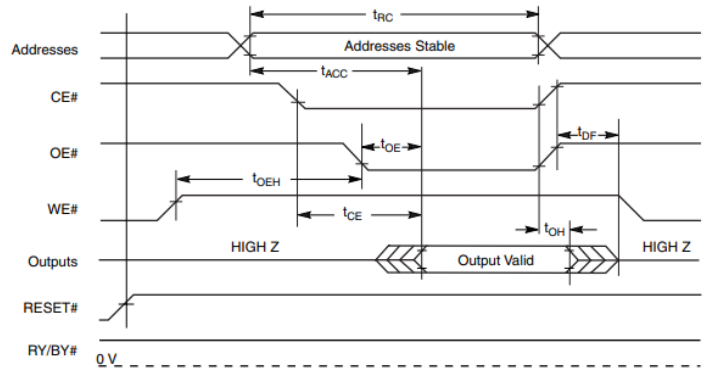


Figure 9: Read operation timing constraints and signal waveforms [15]

4.3.2 Flash Memory Operation – Write

To perform a write operation, a 4-cycle command sequence is required - which entails writing a stream of data values into specific address locations in the memory. On the final cycle, the user entered data is written into the specified address location. This sequence can be found in the 'Program' row of the table in figure 10.

Table I7. S29AL032D Command Definitions — Models 03, 04

Command Sequence (Note 1)			Cycles	Bus Cycles (Notes 2–5)											
				First		Second		Third		Fourth		Fifth		Sixth	
				Addr	Data	Addr	Data	Addr	Data	Addr	Data	Addr	Data	Addr	Data
Read (Note 6)			1	RA	RD										
Reset (Note 7)			1	XXX	F0										
Autoselect (Note 8)	Manufacturer ID	Word	4	555	AA	2AA	55	555	90	X00	01				
		Byte		AAA		555		AAA							
	Device ID, Model 03	Word	4	555	AA	2AA	55	555	90	X01	22F6				
		Byte		AAA		555		AAA		X02	F6				
	Device ID, Model 04	Word	4	555	AA	2AA	55	555	90	X01	22F9				
		Byte		AAA		555		AAA		X02	F9				
	Secured Silicon Sector Factory Protect Model 03, (Note 9)	Word	4	555	AA	2AA	55	555	90	X03	8D/0D				
		Byte		AAA		555		AAA		X06					
	Secured Silicon Sector Factory Protect Model 04, (Note 9)	Word	4	555	AA	2AA	55	555	90	X03	9D/1D				
		Byte		AAA		555		AAA		X06					
	Sector Protect Verify (Note 10)	Word	4	555	AA	2AA	55	555	90	(SA) X02	XX00				
											XX01				
		Byte		AAA		555		AAA		(SA) X04	00				
											01				
Enter Secured Silicon Sector Region		Word	3	555	AA	2AA	55	555	88						
		Byte		AAA		555		AAA							
Exit Secured Silicon Sector Region		Word	4	555	AA	2AA	55	555	90	XXX	00				
		Byte		AAA		555		AAA							
CFI Query (Note 11)		Word	1	55	98										
		Byte		AA											
Program		Word	4	555	AA	2AA	55	555	A0	PA	PD				
		Byte		AAA		555		AAA							
Unlock Bypass		Word	3	555	AA	2AA	55	555	20						
		Byte		AAA		555		AAA							
Unlock Bypass Program (Note 12)			2	XXX	A0	PA	PD								
Unlock Bypass Reset (Note 13)			2	XXX	90	XXX	00								
Chip Erase		Word	6	555	AA	2AA	55	555	80	555	AA	2AA	55	555	10
		Byte		AAA		555		AAA		AAA		555		AAA	
Sector Erase		Word	6	555	AA	2AA	55	555	80	555	AA	2AA	55	SA	30
		Byte		AAA		555		AAA		AAA		555			
Erase Suspend (Note 14)			1	XXX	B0										
Erase Resume (Note 15)			1	XXX	30										

Figure 10: Flash memory command sequence definitions [15]

The write cycles are performed by replicating the waveforms as seen in figure 11. This requires having a separate VHDL process control the required flash signals and place the user inputted data and address values into the final cycle of the command sequence. If this is performed within the timing constraints, the device will write the entered data into the memory.

Erase/Program Operations

Parameter		Description			Speed Options		Unit
JEDEC	Std				70	90	
t_{AVAV}	t_{WC}	Write Cycle Time (Note 1)		Min	70	90	ns
t_{AVWL}	t_{AS}	Address Setup Time		Min	0		ns
t_{WLAX}	t_{AH}	Address Hold Time		Min	45	45	ns
t_{DVWH}	t_{DS}	Data Setup Time		Min	35	45	ns
t_{WHDX}	t_{DH}	Data Hold Time		Min	0		ns
	t_{OES}	Output Enable Setup Time		Min	0		ns
t_{GHWL}	t_{GHWL}	Read Recovery Time Before Write (OE# High to WE# Low)		Min	0		ns
t_{ELWL}	t_{CS}	CE# Setup Time		Min	0		ns
t_{WHEH}	t_{CH}	CE# Hold Time		Min	0		ns
t_{WLWH}	t_{WP}	Write Pulse Width		Min	35	35	ns
t_{WHWL}	t_{WPH}	Write Pulse Width High		Min	30		ns
	$t_{SR/W}$	Latency Between Read and Write Operations		Min	20		ns
t_{WHWH1}	t_{WHWH1}	Programming Operation (Note 2)	Byte	Typ	9		µs
			Word	Typ	11		
t_{WHWH1}	t_{WHWH1}	Accelerated Programming Operation, Word or Byte (Note 2)		Typ	7		µs
t_{WHWH2}	t_{WHWH2}	Sector Erase Operation (Note 2)		Typ	0.7		sec
	t_{VCS}	V_{CC} Setup Time (Note 1)		Min	50		µs
	t_{RB}	Recovery Time from RY/BY#		Min	0		ns
	t_{BUSY}	Program/Erase Valid to RY/BY# Delay		Max	90		ns

AC Characteristics

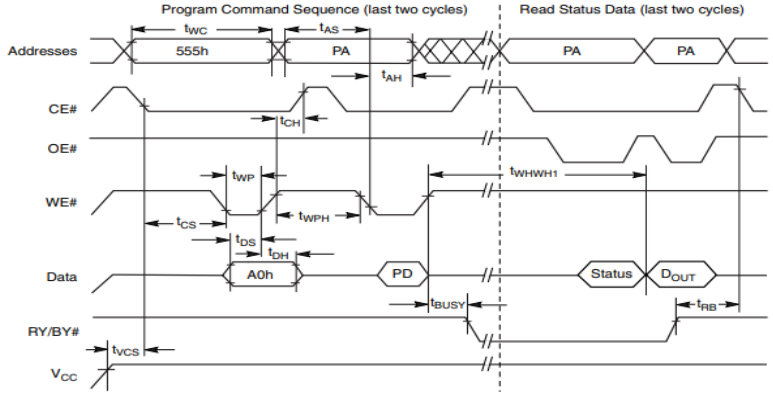
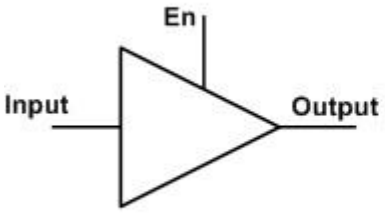


Figure 11: Write operation timing constraints and signal waveforms [15]

4.3.2.1 Tri-state Data Bus Buffer

As the system is using the data wires to read in and write out, a tri-state buffer is required for the data bus to allow bidirectional functionality. By having a tri-state buffer, it ensures that only one direction can drive the bus at any one time, therefore at all other times, the line is driven to high impedance and eliminates any bus contentions [14].

Symbol



Truth Table

En	Input	Output
0	X	Hi-Z
1	0	0
1	1	1

Figure 12: Diagram and truth table for [16]

4.3.4 Flash Memory Operation – Erase

To perform an erase operation, a 6-cycle command sequence is used similarly to the write operation. This sequence can be found under the 'Sector Erase' row within figure 10 and enables a user to clear the entire contents of a 64kb sector. This is especially useful as data cannot be overwritten once set, therefore, if a user wants to change a specific byte in an address location, they must first erase the sector and rewrite the byte.

Analogous to the write operation, the flash signals are to be controlled via a separate VHDL process in which the objective is to replicate the waveforms seen in figure 13. If this is done correctly, ensuring the timing constraints are not exceeded, the user is able to erase the entire contents of a memory sector and reset all values back to default bytes of "11111111".

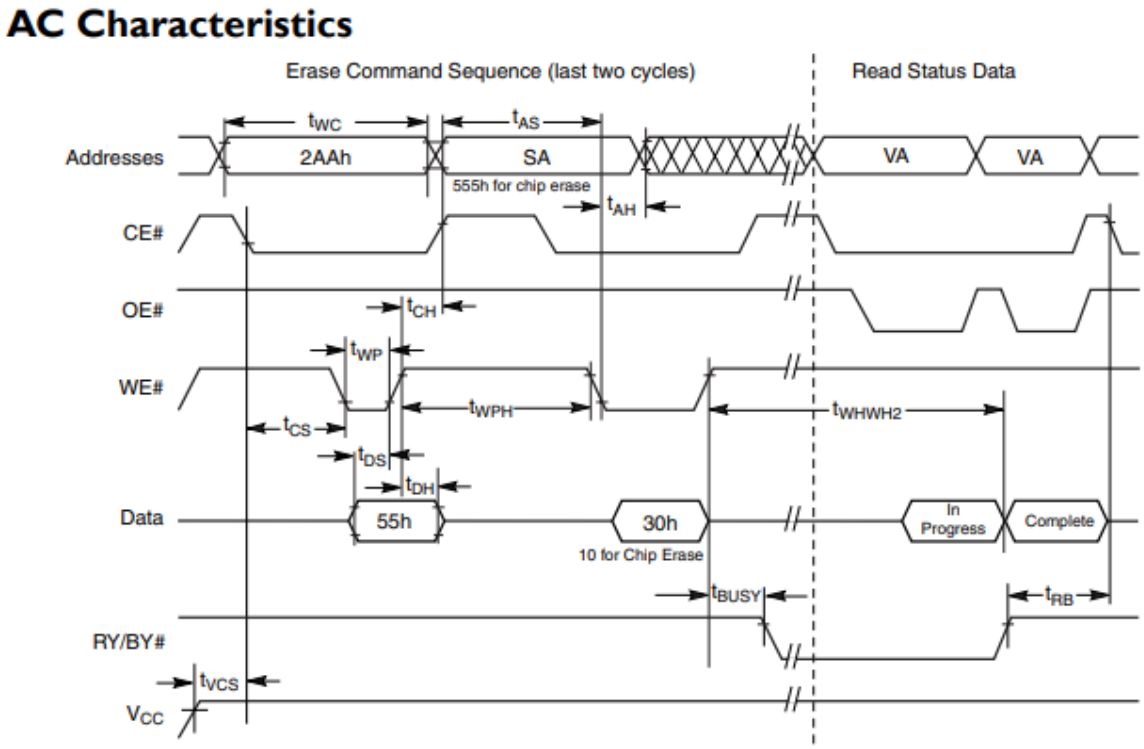


Figure 13: Erase operation signal waveforms [15]

5. Results:

5.1 RS232 Receiver

The VHDL code for the RS232 receiver can be located as figure 26 in the appendix. A simulation of the code within ModelSim-Altera produces the waveform diagram seen in figure 14.

This figure shows the finite state machine, *Rx_state*, transitioning through each of the receiver states in response to the logical values on the *Rx* line. As per the theory in section 4.1, the system waits for the line to drop - indicating the start bit - and uses the *rx_clk_counter* to count 434 clock cycles to the middle of each of the next 8 data bits. At the middle of each data bit, the logical value of the line is stored in the *current_rx_byte* array. Once all 8 bits have been set within the *current_rx_byte* signal, the byte is copied into the *RxBuffer* and stored for later processing.

The simulation suggests, the time taken to move through the entire FSM and read one byte of data is 87us.

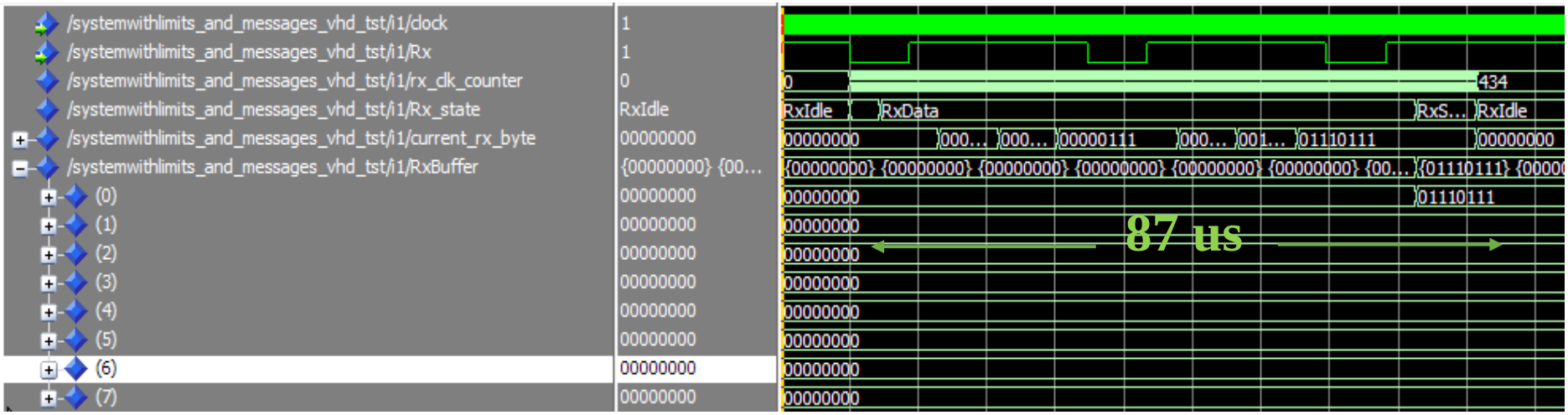


Figure 14: Simulated results for RS232 receiver

5.2 Command Interpreter

The receiver and ASCII to hex conversion VHDL processes are combined and simulated within ModelSim-Altera. The VHDL code can be found as figure 27 in the appendix.

Figure 15 shows the receiver read in 8 consecutive bytes of data via the *Rx* line and store them within the *RxBuffer* as per the results in section 5.1. Once all 8 bytes have been received, the *rx_inputs_complete* flag is set, and the ASCII bytes in *RxBuffer[1]* – *RxBuffer[7]* are converted into the equivalent hexadecimal values using a series of IF statements. The equivalent values are then stored within the *HexBuffer* and the *conversion_complete* flag is triggered.

One important result to note is the time between the *rx_inputs_complete* flag and the *conversions_complete* flag, t_{cc}. This time is basically negligible in comparison to the time taken to receive a command, t_c, emphasising the dominance of the receiver in the system throughput.

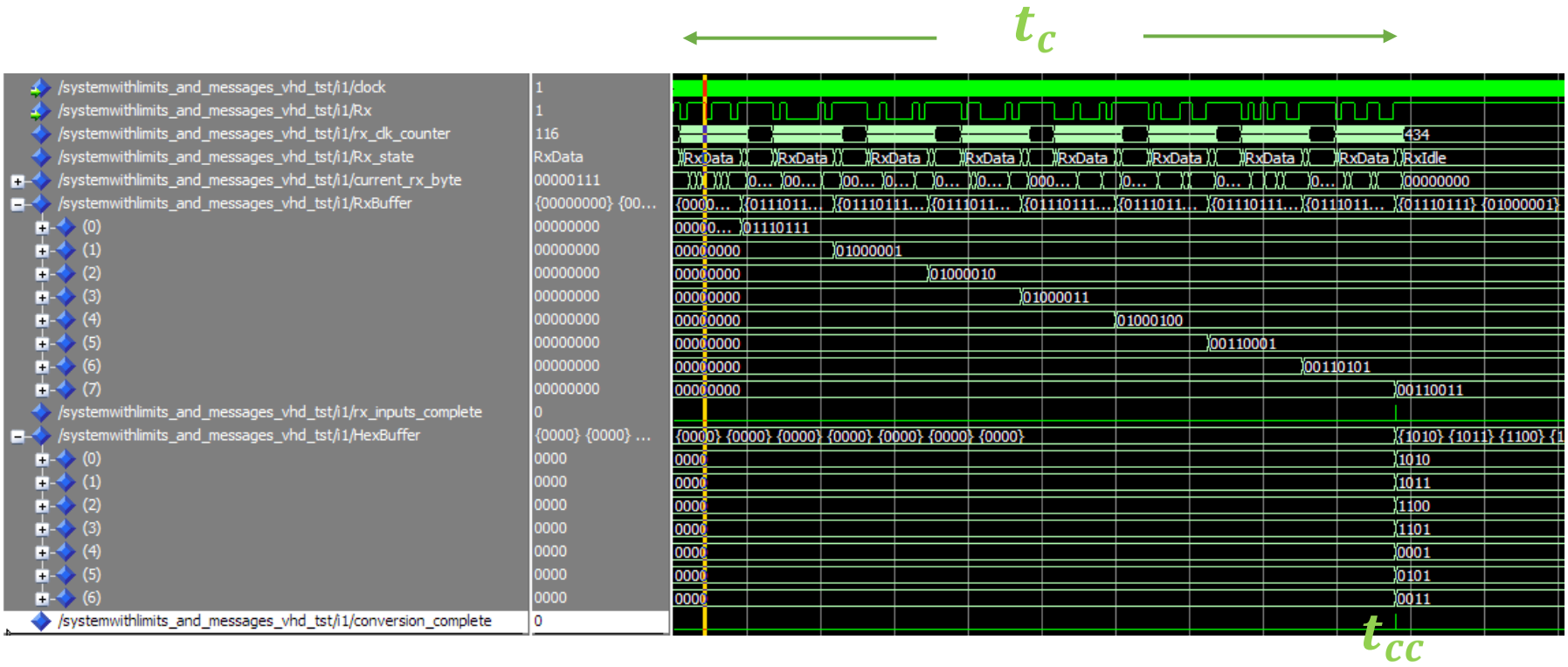


Figure 15: Simulated results for receiving a write command and converting the ASCII user inputs to Hex equivalents.

5.3 Flash Mode Operations

Once the command interpreter has finished the conversion process, the flash state FSM selects the requested flash mode operation – as per the ASCII value in *RxBuffer[0]* - and executes the signal responses explored within the background theory of section 4.3.

5.3.1 Flash Mode Operation – Read

Figure 16 shows the simulation of the VHDL code whilst undergoing a read operation. This code can be found as figure 28 in the appendix.

The receiver firstly reads in the user inputs and performs the conversion as discussed in section 5.2. The system then checks the value in *RxBuffer[0]*. As this value is "01110010" or ASCII 'r', it indicates to the system that the user wants to perform a read operation and thus moves the *Flash_state* FSM into the *Read* state. While in the read state, the FPGA executes the relevant flash signal logic as per the background theory explained in section 4.3.1. The most important feature in figure 16 is the point at which the contents of the *flash_data* signal are copied across to *Tx_output_byte*, 2 clock cycles after the OE signal goes low. This confirms that in simulation, the system can successfully retrieve contents of the flash memory.

The read cycle time, t_{rc} (80ns) is the time taken to perform the read operation on the flash memory. Time, t_{roc} (160ns) is the read operation complete time, which equates the time taken for the system to move through the entire read FSM cycle, complete the read operation and return to an idle state.

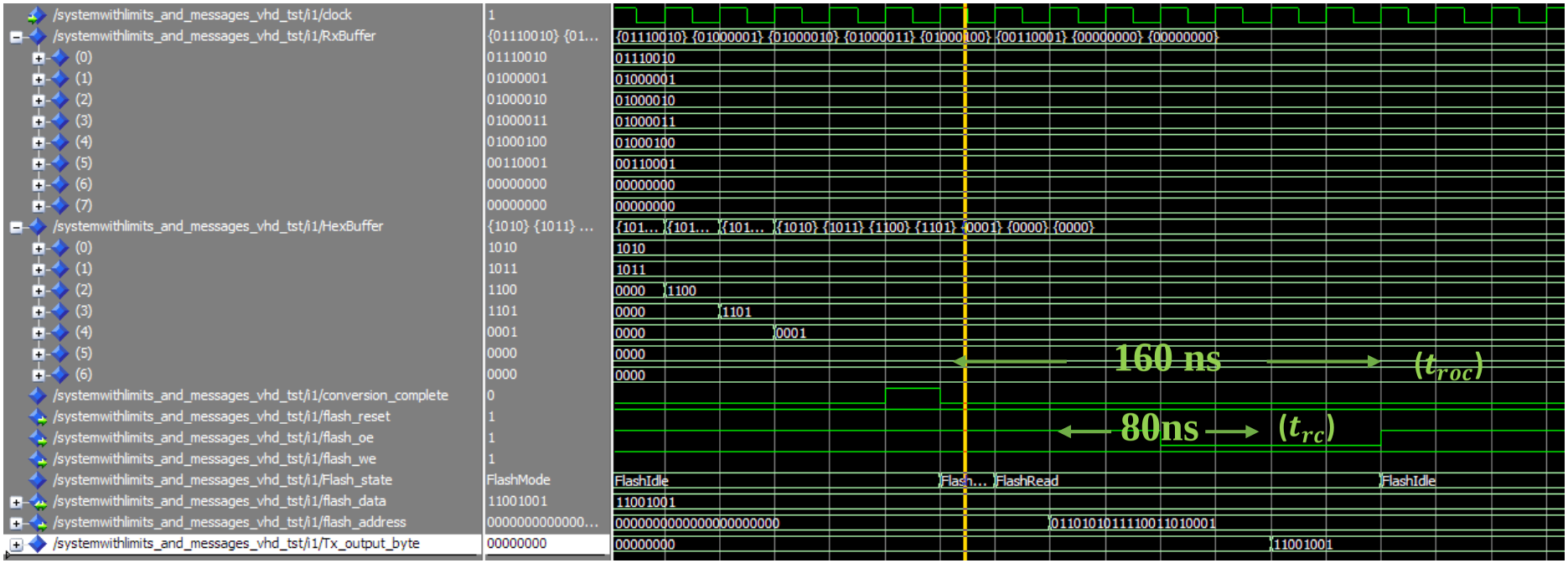


Figure 16: Simulated results for read operation

5.3.2 Flash Mode Operation – Erase

Figure 17 shows the simulation results of the system undergoing an erase operation. The VHDL code can be found as figure 28 in the appendix.

The receiver firstly reads the inputs and completes the hex conversion identically that of the read operation. However, as input 1 in the *RxBuffer[0]* is “01100101” or ASCII character ‘e’, the flash FSM moves into the *Erase* state and sets the flash *Reset*, *OE* and *WE* signals to the correct initial values. The 6-cycle erase sequence - explained in section 4.3.4 - is then observed, comprising of 5 unlock cycles and a 6th sector address cycle in which the user-selected sector is inserted onto the *flash_address* wires. The entire erase process from the conversions completed pulse to the *Flash_state* FSM returning to idle takes 23 clock cycles – equivalent to 460ns.

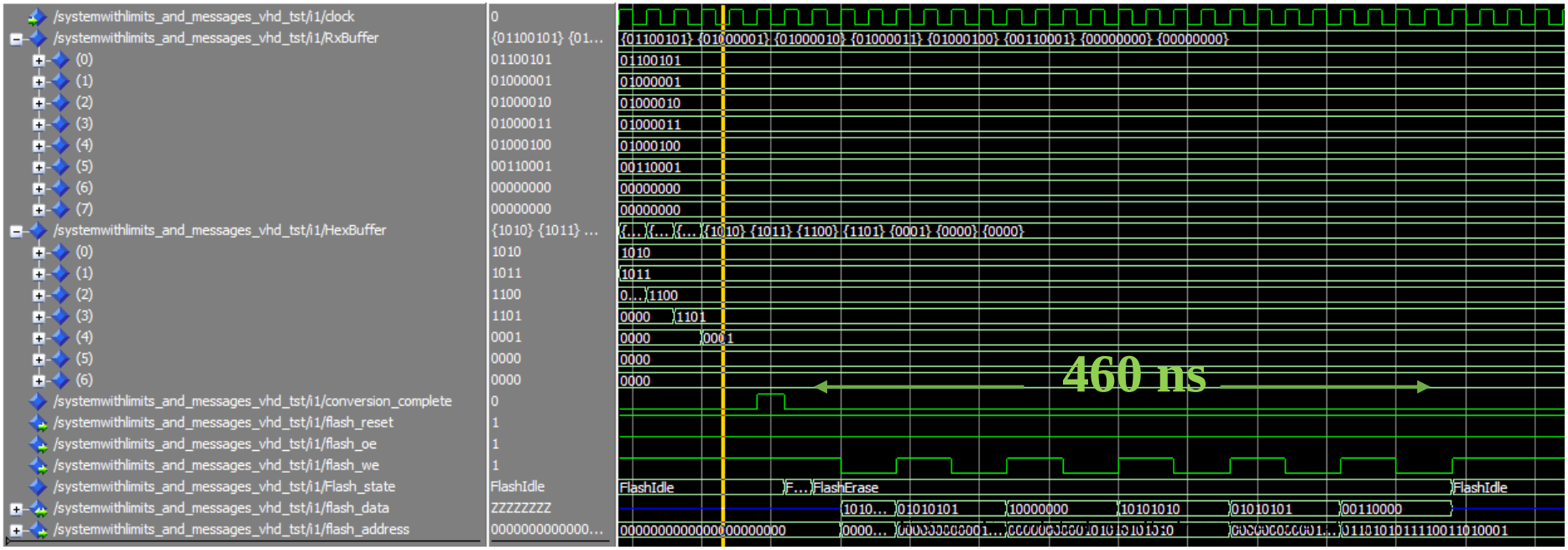


Figure 17: Simulated results for erase operation

5.3.3 Flash mode operation - Write

Figure 18 shows the results of a ModelSim-Altera simulation whilst the system undergoes a write operation. The VHDL code for this process can be found as figure 28 in the appendix.

The simulated waveform in figure 18 completes the exact same receive and convert processes as discussed in the two pervious flash operations. The system checks the contents of the *RxBuffer[0]* for a ‘w’ and transitions the *Flash_state* FSM into the *Write* state. While in the write state, the VHDL code is designed to set the flash signals as per the 4-cycle command sequence explained within section 4.3.2. On the final write cycle, the user inputted data and addresses are set onto the *flash_data* and *flash_address* wires respectively, and the byte of data is written into the memory.

The time taken to complete this entire write operation takes 16 clock cycles – equivalent to 320ns.

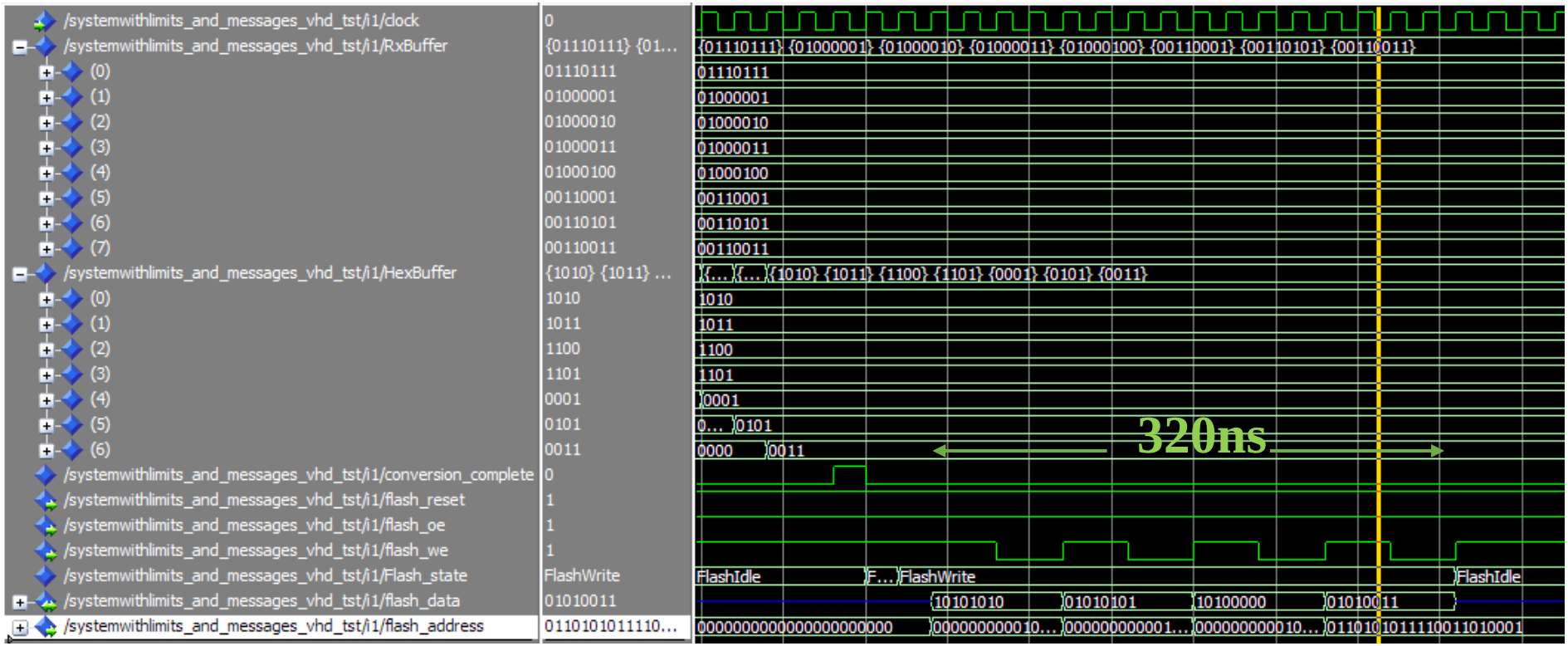


Figure 18: Simulated results for write operation

5.4 RS232 Transmitter

Once the FPGA has successfully completed the requested operation, the transmitter VHDL process is triggered to return either; a success message for an erase or write operation or the requested byte of data from a read operation. Below is the simulated waveform of the transmitter responding to a read operation. The VHDL process can be found as figure 29 in the appendix.

Figure 19 shows the system responding to a requested read operation. In response to this request the transmitter FSM *Tx_state* transitions through *Idle, Start, Data* and *Stop* states and transmits an array of bytes by setting the output of the *Tx* line in accordance with the theory stated in section 4.1. The array of bytes transmitted, *current_read_message*, consists of the read data from memory, *Tx_output_byte*, followed by the ASCII byte equivalent values for "Read complete!". Success messages for write and erase requests were also stored within separate arrays and transmitted upon a successful write or erase operation.

The time taken to transmit the requested byte of data, t_{td}, is 87us whilst the total time taken to transmit the byte of data plus the success message, t_{tm}, is approximately 1.65ms.

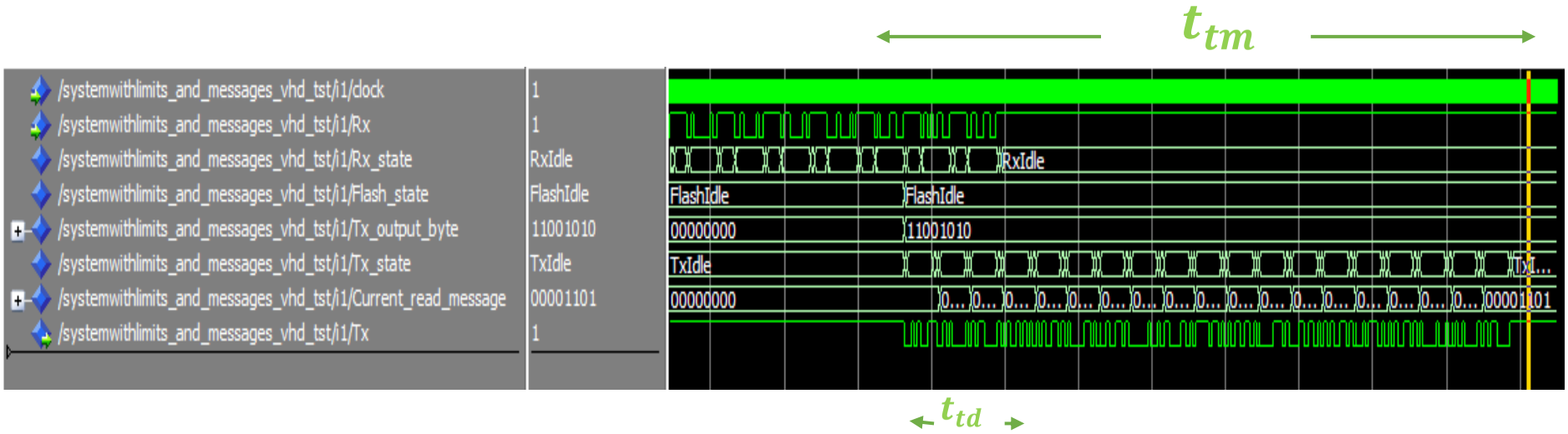


Figure 19: Simulated waveform for transmitting results of a read operation

5.5 Complete System Performance

The table below shows the system performance for each of the 3 flash modes of operation without the additional success messages. The calculations for these timings can be found in the appendix in section 9.3.

Operation	Time taken receive command, *Tc* (us)	*Tc* as a percentage of *Tt* (%)	Time taken to perform flash operation, *To* (ns)	*To* as a percentage of *Tt* (%)	Time taken to transmit result, *Tr* (us)	*Tr* as a percentage of *Tt* (%)	Total time for entire operation, *Tt* (us)
Read	522	85.69	160	0.026	87	14.28	609.16
Erase	522	99.91	460	0.088	X	-	522.46
Write	696	99.95	320	0.046	X	-	696.32

Figure 20: Performance timings for each mode of operation

Figure 21 visualises the effect that each component has on the throughput of the system. As can be seen, the transmitter and receiver account for over 99% of the completion time of each operation and thus demonstrates the dominance these components have on the performance of the system.

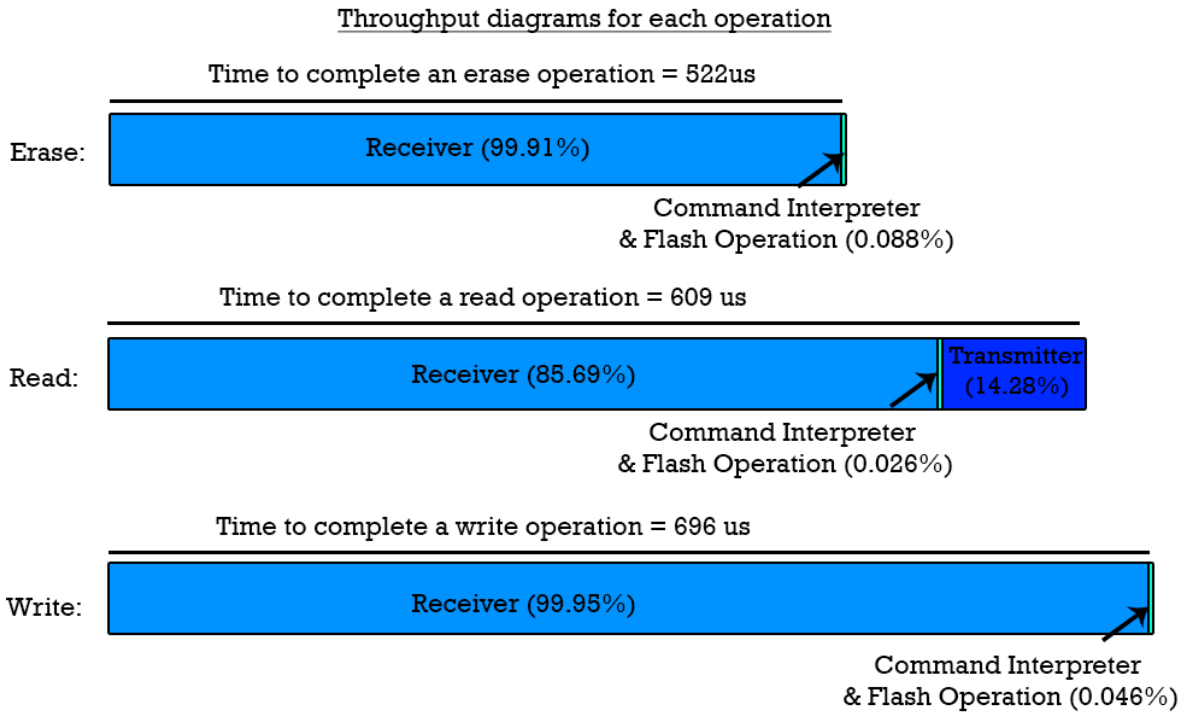


Figure 21: Throughput diagrams for each operation

5.6 System Hardware Implementation

To test the functionality of the entire system, each of the previously simulated VHDL processes are combined into one script, and the code is synthesized onto the DE1 development board. The board is then connected to the PC via an RS232 connection and a serial terminal is opened on the PC. The combined entire VHDL script is found within the appendix.

Firstly, before any read or write operations are executed, the sector AAAAA is erased by typing “eAAAAA” into the serial terminal. Figure 22 shows the response from the transmitter.

Figure 22: System response to an erase operation

Secondly, an arbitrary byte, x4C, or ASCII “L” is written into the memory address AAAAA by typing the command “wAAAAA4C”. The system responded with a write success message as seen in figure 23.

Figure 23: System response to a write operation

Lastly, to confirm the byte has been successfully written, it is then read out using the command “rAAAAA”. As can be seen from figure 24, the system successfully reads out the correct byte, demonstrating the success of the entire functionality of the system.

Figure 24: System response to a read operation

6. Discussions:

6.1 Receiver

6.1.1 Receiver Performance

From the receiver simulation timings in figure 14, the system took approximately 87us to successfully receive and process 10 bytes of data and return to an idle state. This indicates that in theory, the system could operate at a maximum rate of :

$$Data\ Rate\ (bytes/second) = \frac{1}{Bit\ Period} = \frac{1}{87\times10^{-6}} = 11494.3\ bytes/second \quad \text{(v)}$$

in comparison to the theoretical maximum of 11520 bytes per second calculated in equation iv. This means that the interface is operating at an efficiency of approximately 99.78%. The 0.22% loss is likely due to various signal overheads within the VHDL code such as state transitions and signal resets. This high receiver efficiency is extremely important as figure 21 illustrates the dominance the receiver has on the performance of the entire system. Therefore, the high receiver efficiency indicates that the system is operating at almost full capacity.

6.2 Command Interpreter

6.2.1 Command Interpreter Analysis

The simulation shows that time taken to complete the ASCII to hex conversion is negligible in comparison to the time taken for the RS232 receiver to receive a command, thus the interpreter will have little to no impact on the performance of the system.

6.2.2 ASCII to Hex Converter Assumptions

One assumption that should be made is that the user operating the system should understand the format of the commands they are entering. For example, when requesting flash operations, the user must address the locations using hexadecimal values. If a user types a non-hex character such as a “K”, this will be stored as x0 and could generate errors. Therefore, the user must correctly input valid hex addresses for the system to work. With an extension on the scope of this project, a user interface could have been created to transmit the data to the FPGA which could have included an input error mechanism. For this project however, it is assumed that the user knows the input format.

If this is assumed, then figure 15 demonstrates that the command interpreter successfully coverts the ASCII bytes to the respective hex value and thus contributes to the aims and objectives stated in the introduction.

6.3 Flash Mode Operation – Read

6.3.21 Flash Read Operation Performance

In terms of performance, the read simulation can be analysed in comparison to the datasheet timing requirements of the flash memory read operation. As observed in figure 9 within the background theory, the datasheet states that an entire read cycle time, t_{rc}, takes a minimum of 70ns with the most notable delay being the OE to valid output delay, t_{oe}, being a maximum time of 30ns.

From the simulated waveform in figure 16, it can be seen that the read cycle time between setting the address wires and producing a valid output took 80ns - in comparison to the minimum timing constraint of 70ns stated on the datasheet. This appears to be an efficient design, however, does not account for the additional time taken to move through the FSM and reset the system for subsequent operations. This time, t_{roc} , was 160ns however, as mentioned before and shown in figure 21, this time is negligible in comparison to the processing time of the RS232 receiver, therefore has no major impact on the performance of the system.

6.4 Flash Mode Operation – Erase

6.4.3 Flash Erase Performance Analysis

Figure 17 shows the simulated timings of an entire sector erase process lasting 460ns. The minimum possible time taken for this 6-cycle process can be calculated using the timing constraints in figure 13. If the minimum time taken for 1 write cycle is 70ns then, the fastest execution for a 6-cycle process would be 420ns. This infers that the simulation is approximately 91% efficient when performing flash erase operations. However, as previously mentioned and shown within the table of figure 20, the time taken to complete the flash signals is only 0.088% of the entire time of the entire operation. Therefore, the efficiency of the erase process is sufficient to not hinder the performance of the system and thus meets a portion of the third aims and objectives stated within the introduction.

6.5 Flash Mode Operation – Write

6.5.3 Flash Write Operation Performance

The timing requirements in figure 11 state that each write cycle takes a minimum of 70ns, meaning a 4-cycle operation command sequence such as a write operation should take a minimum of 280ns. The entire write process within the simulation in figure 18 took 320ns to complete - which implies that the write operation is 87.5% efficient. Once again - as shown in the table of figure 20 - this performance is virtually negligible as it only accounts for 0.046% of the entire write operation timings. Therefore, the efficiency of this process is sufficient and successfully meets aims and objectives three, stated in the introduction.

6.6 Transmitter

6.6.2 Transmitter Performance

The transmitted success messages are only generated to provide an aesthetic user interface for the design. However, to maximise performance, these are not to be included as they introduce performance penalties for transmitting extra bytes of information. Therefore, the transmitter is only used for read operations. In this case, the simulation took approximately 87us to transmit the result back to the user. This is identical to the speed of the receiver and thus operates at a similar efficiency of 99.78%. This is extremely beneficial considering, this performance accounts for almost 15% of the throughput of a read operation.

7. Conclusion:

Overall, the system can successfully retrieve and interpret user commands from a serial terminal, perform requested read, write, or erase operations, and transmit the results back to the user. The success of these three specifications confirms that every one of the aims and objectives stated within the introduction have been met.

The results of the simulation demonstrate the importance of the performance of the RS232 interface. With the transmitter and receiver accounting for over 99% of the total time of each operation, the effectiveness of the entire communication channel is dominated by these two processes. Thus, a 99% bit rate efficiency on both interfaces means the system is almost working at its theoretical maximum speed, therefore there will be no bottlenecks when operating the interface at high data rates.

At the moment, the user can only access 1 byte at a time, however, to improve the entire system performance, a buffer could be introduced into the system to allow the user to request access to an entire block by stating the address of the start of a block. This slight adjustment would improve the performance of the system as it would remove the need to request each byte of data individually which would remove multiple lots of receiver delays. Also, by introducing block access, the system would have the capabilities to be used directly as an arbitrary filesystem which was as intended. This was not entirely completed as this project was mostly concerned with the actual serial-to-memory interface. However, with slight adjustments to either the user side or internal to the controller, the block access can be easily achieved.

8. References:

[1] J. Schaumann, “Storage Models” in *Principles of System Administration*, 1st ed. 2021, p. 76. [Online]. Available: https://www.netmeister.org/book/principles-of-system-administration.pdf

[2] G. R. Smith, “FPGA Development Phases” in *FPGAs 101*. Newnes, 2010, pp. 43-55. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9781856177061000035

[3] D. M. Harris and S. L. Harris, “Hardware Description Languages” in *Digital Design and Computer Architecture*, 2nd ed. Morgan Kaufmann, 2013, pp. 172-237. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780123944245000045

[4] "Advantages of FPGA | Disadvantages of FPGA", Rfwireless-world.com, 2021. [Online]. Available: https://www.rfwireless-world.com/Terminology/Advantages-and-Disadvantages-of-FPGA.html. [Accessed: 20- May- 2021].

[5] Altera. *DE1 Development and Education Board User Manual v1.2.1*. (2012). Accessed: May,20,2021. [Online]. Available: https://www.intel.com/content/dam/altera-www/global/en_US/portal/dsn/42/doc-us-dsnbk-42-4904342209-de1-usermanual.pdf

[6] N. I. Rafla and B. L. Davis, "A Study of Finite State Machine Coding Styles for Implementation in FPGAs", *IEEE International Midwest Symposium on Circuits and Systems*, vol. 49, p. 337, 2006. Available: https://core.ac.uk/download/pdf/61743673.pdf. [Accessed 20 May 2021].

[7] M. Ben-Ari and F. Mondada, “Finite State Machines” in *Elements of Robotics*, 1st ed. Springer, Cham, 2018, pp. 55-61. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-319-62533-1_4

[8] J. Corbet, A. Rubini and G. Kroah-Hartman, “Block Drivers” in *Linux device drivers*, 3rd ed. O'Reilly, 2005, pp. 464-496. [Online]. Available: https://lwn.net/Kernel/LDD3/

[9] T. Staerk, "Blocks, block devices and block sizes", *Linuxintro.org*, 2008. [Online]. Available: http://www.linuxintro.org/wiki/Blocks,_block_devices_and_block_sizes [Accessed: 20- May- 2021].

[10] J. Main, "How RS232 Works: The Easy Guide to RS232.", Best Microcontroller Projects, 2021. [Online]. Available: https://www.best-microcontroller-projects.com/how-rs232-works.html. [Accessed: 20- May- 2021].

[11] "Serial Communication - learn.sparkfun.com", *Learn.sparkfun.com*, 2021. [Online]. Available: https://learn.sparkfun.com/tutorials/serial-communication/all. [Accessed: 20- May- 2021].

[12] P. An,”RS232 Serial Interface” in *PC interfacing using Centronic, RS232 and game ports*. Oxford: Newnes, 2002, pp. 13-17. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780750636377500031

[13] "ASCII Table,", *LookUpTables*. [Online]. Available: http://www.asciitable.com/, Accessed: May,20,2021.

[14] R. Keim, "UART Baud Rate: How Accurate Does It Need to Be?", *All About Circuits*, 2017. [Online]. Available: https://www.allaboutcircuits.com/technical-articles/the-uart-baud-rate-clock-how-accurate-does-it-need-to-be/. [Accessed: 20- May- 2021].

[15] Spansion. *S29AL032D70TFI040 Datasheet v3*. (2005). Accessed: May,20,2021. [Online]. Available: https://www.alldatasheet.com/datasheet-pdf/pdf/109889/SPANSION/S29AL032D70TFI040.html

[16] Y. Stoyanov, "Digital Buffers And Their Usage - Open4Tech", *Open4Tech*, 2021. [Online]. Available: https://open4tech.com/digital-buffers-and-their-usage/. [Accessed: 20- May- 2021].

9. Appendix:

9.1 Flash to DE1 Connection Schematics

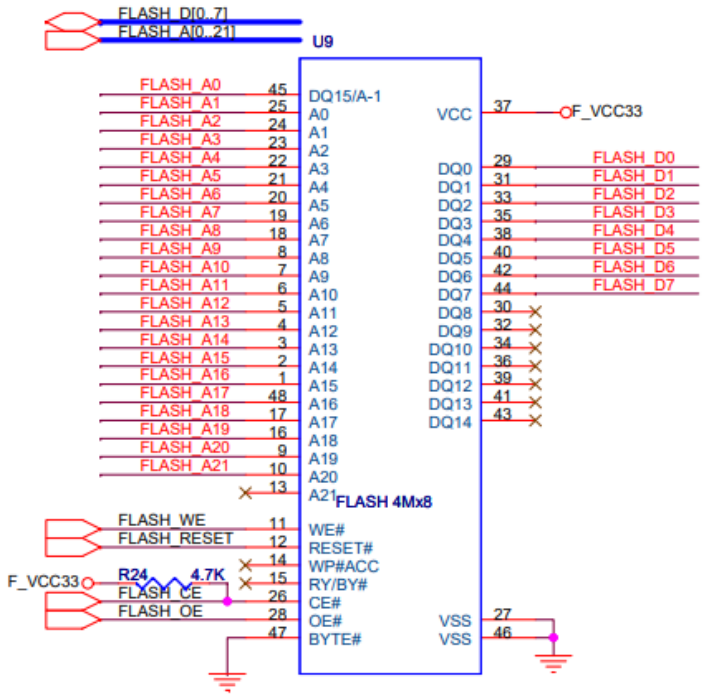


Figure 25: Schematic of flash memory connections to FPGA on the DE1 development board [5]

9.2 VHDL Code

```
Receive: PROCESS (clock)
BEGIN

   IF rising_edge(clock) THEN                                          --At the rising edge of the clock do the following

         CASE Rx_state IS                                              --FSM to receive bytes of data via an RS232 wire

            WHEN RxIdle =>                                             --System waits in idle state until Rx goes low
            current_rx_byte <= "00000000";                             --Initialises current byte to be empty
            IF(Rx = '0') THEN
            rx_clk_counter <= 0;                                       --When Rx goes low, system moves into idle state
            Rx_state <= RxStart;
            ELSE
            Rx_state <= RxIdle;                                        --If Rx remains high, system idles
            END IF;

            WHEN RxStart =>
            IF(rx_clk_counter = (rx_clks_per_bit/2)) THEN               --When in start state, system counts a half bit period and checks if line is still low
               IF Rx = '0' THEN                                        --If the start bit is still low then start is validated and system moves to data state
               rx_clk_counter <= 0;
               Rx_state <= RxData;
               ELSE
               Rx_state <= RxIdle;                                     --If start bit is not low, then start is invali and system moves back to idle state
               END IF;
            ELSE
            rx_clk_counter <= rx_clk_counter + 1;                      --Counting up clock cycles to half bit period
            Rx_state <= RxStart;
            END IF;

            WHEN RxData =>
            IF(rx_clk_counter < rx_clks_per_bit) THEN                  --System counts 1 bit peiod to find middle of data bit
            rx_clk_counter <= rx_clk_counter +1;
            Rx_state <= RxData;
               ELSE IF(rx_databit_index < 8) THEN                      --Stores the line value in next bit of array
               current_rx_byte(rx_databit_index) <= Rx;                --Repeated for all 8 bits
               rx_databit_index <= rx_databit_index +1;
               rx_clk_counter <=0;
               Rx_state <= RxData;
               ELSE
               rx_clk_counter <= 0;                                    --Once system has all 8 data bits
               rx_databit_index <= 0;                                  --Resets clock counter and index
               Tx_ByteLEDs <= current_rx_byte;                         --Shows current byte on LEDs
               RxBuffer(rx_input_counter) <= current_rx_byte;          --Stores byte in appropriate location in receive buffer
               rx_input_counter <= rx_input_counter + 1;
               Rx_state <= RxStop;                                     --Moves to stop state
               END IF;
            END IF;

             WHEN RxStop =>                                                --When in stop state, system counts 1 bit period and validates stop bit
             IF(rx_clk_counter < rx_clks_per_bit) THEN
             rx_clk_counter <= rx_clk_counter +1;
             Rx_state <= RxStop;
             ELSE
             Rx_state <= RxIdle;                                           --Once detected, moved back to idle state
             END IF;

             WHEN OTHERS =>
             Rx_state <= RxIdle;                                           --Error handler, to reset system to idle

             END CASE;


             IF(RxBuffer(0) = x"77") THEN                                  --IF a write operation is required, expect 8 inputs (2 extra for byte to writ
             rx_expected_inputs <= 8;
             ELSE
             rx_expected_inputs <= 6;                                      --Else just expect 6
             END IF;
             --After all 8 inputs, set next stage flags--
             IF(rx_input_counter = rx_expected_inputs) THEN
             rx_inputs_complete <= '1';                                    --After 8 inputs, tell next process to start
             rx_input_counter <= 0;                                        --Reset inputs counter
             END IF;
             IF(buffer_index = rx_expected_inputs-1) THEN                  --If then next stage completes the conversions
             rx_inputs_complete <= '0';                                    --Reset the flag
             END IF;


   END IF;

END PROCESS Receive;
```

Figure 26: VHDL code for receiver

```
Converter: PROCESS (clock,rx_inputs_complete)                                   --Process is activated when the all 8 inputs are stored in receive buffer
BEGIN

   IF rising_edge(clock) THEN

      CASE Conv_state IS                                                        --FSM converts ASCII inputs to hex values to be interpretted by flash device

      WHEN ConvIdle =>                                                          --Process idles & waits for inputs to be complete
      conversion_complete <= '0';                                               --While system idles, the conversions are not complete, therefore the next process is held
      IF(rx_inputs_complete = '1') THEN                                         --If inputs are complete move to next state and begin converting
      Conv_state <= ConvProcessing;
      END IF;

      WHEN ConvProcessing =>
      IF (buffer_index<rx_expected_inputs-1) THEN                                                   --Cycle through the 2nd-8th received byte and convert from ASCII values to hex values
         IF RxBuffer(buffer_index+1) <= "00110000" THEN             --IF 0 (30)       --Converting ASCII values to hex values and storing in hex buffer
         HexBuffer(buffer_index) <= x"0";
         buffer_index <= buffer_index+1;                  --0000
            ELSE IF RxBuffer(buffer_index+1) <= "00110001" THEN         --IF 1 (31)
            HexBuffer(buffer_index) <= x"1";                      --0001
            buffer_index <= buffer_index+1;
               ELSE IF RxBuffer(buffer_index+1) <= "00110010" THEN         --IF 2 (32)
               HexBuffer(buffer_index) <= x"2";                         --0010
               buffer_index <= buffer_index+1;
                  ELSE IF RxBuffer(buffer_index+1) <= "00110011" THEN         --IF 3 (33)
                  HexBuffer(buffer_index) <= x"3";                     --0011
                  buffer_index <= buffer_index+1;
                     ELSE IF RxBuffer(buffer_index+1) <= "00110100" THEN         --IF 4 (34)
                     HexBuffer(buffer_index) <= x"4";                     --0100
                     buffer_index <= buffer_index+1;
                        ELSE IF RxBuffer(buffer_index+1) <= "00110101" THEN         --IF 5 (35)
                        HexBuffer(buffer_index) <= x"5";                     --0101
                        buffer_index <= buffer_index+1;
                           ELSE IF RxBuffer(buffer_index+1) <= "00110110" THEN         --IF 6 (36)
                           HexBuffer(buffer_index) <= x"6";                     --0110
                           buffer_index <= buffer_index+1;
                              ELSE IF RxBuffer(buffer_index+1) <= "00110111" THEN         --IF 7 (37)
                              HexBuffer(buffer_index) <= x"7";                     --0111
                              buffer_index <= buffer_index+1;
                                 ELSE IF RxBuffer(buffer_index+1) <= "00111000" THEN         --IF 8 (38)
                                 HexBuffer(buffer_index) <= x"8";                     --1000
                                 buffer_index <= buffer_index+1;
                                    ELSE IF RxBuffer(buffer_index+1) <= "00111001" THEN         --IF 9 (39)
                                    HexBuffer(buffer_index) <= x"9";                     --1001
                                    buffer_index <= buffer_index+1;
                                       ELSE IF RxBuffer(buffer_index+1) <= "01000001" THEN         --IF A (41)
                                          HexBuffer(buffer_index) <= x"A";                     --1010
                                          buffer_index <= buffer_index+1;
                                             ELSE IF RxBuffer(buffer_index+1) <= "01000010" THEN         --IF B (42)
                                             HexBuffer(buffer_index) <= x"B";                     --1011
                                             buffer_index <= buffer_index+1;
                                                ELSE IF RxBuffer(buffer_index+1) <= "01000011" THEN         --IF C (43)
                                                HexBuffer(buffer_index) <= x"C";                     --1100
                                                buffer_index <= buffer_index+1;
                                                   ELSE IF RxBuffer(buffer_index+1) <= "01000100" THEN         --IF D (44)
                                                   HexBuffer(buffer_index) <= x"D";                     --1101
                                                   buffer_index <= buffer_index+1;
                                                      ELSE IF RxBuffer(buffer_index+1) <= "01000101" THEN         --IF E (45)
                                                      HexBuffer(buffer_index) <= x"E";                     --1110
                                                      buffer_index <= buffer_index+1;
                                                         ELSE IF RxBuffer(buffer_index+1) <= "01000110" THEN         --IF F (46)
                                                         HexBuffer(buffer_index) <= x"F";       --1111
                                                         buffer_index <= buffer_index+1;
      ELSE HexBuffer(buffer_index) <= x"0";                                                                   --IF value inputted is not a hex value(0-F) then set as 0
      buffer_index <= buffer_index+1;
                                                         END IF;
                                                      END IF;
                                                   END IF;
                                                END IF;
                                             END IF;
                                          END IF;
                                       END IF;
                                    END IF;
                                 END IF;
                              END IF;
                           END IF;
                        END IF;
                     END IF;
                  END IF;
               END IF;
            END IF;
         END IF;
      ELSE                                                                  --After buffer has converted all 7 hex values
      buffer_index <= 0;                                                    --Reset the buffer index & move to finishing state
      Conv_state <= ConvFinish;
      END IF;

      WHEN ConvFinish =>                                                    --In finishing state, trigger conversion complete flag for next process and return to idle
      conversion_complete <= '1';
      Conv_state <= ConvIdle;

      WHEN others =>                                                        --Error handler to return to idle
       Conv_state <= ConvIdle;

       END CASE;
    END IF;

END PROCESS Converter;
```

Figure 27: VHDL Code for ASCII to Hex Converter

```
Flash_operation: PROCESS(clock,conversion_complete)                    --Process is triggered after inputs are complete and have been converted to hex values
BEGIN
IF rising_edge(clock) THEN

   CASE Flash_state IS                                                 --Process interprets the inputs and applies the correct mode of operation on the flash memory device

   WHEN FlashIdle =>                                                   --System idles until the hex conversions are completed, then moves into the mode state
   IF (conversion_complete = '1') THEN
   Flash_state <= FlashMode;
   END IF;

   WHEN FlashMode =>                                                   --Mode state interprets the first input and determines which mode of operation to execute
   IF RxBuffer(0) = "01100101" THEN                                    --IF input 1 was an 'e', move to erase state
   Flash_state <= FlashErase;
      ELSE IF RxBuffer(0) = "01110111" THEN                            --IF input 1 was a 'w', move to write state
      Flash_state <= FlashWrite;
         ELSE IF RxBuffer(0) = "01110010" THEN                         --If input 1 was a 'r', move to read state
         Flash_state <= FlashRead;
         END IF;
      END IF;
   END IF;


   WHEN FlashErase =>                                                  --IF an erase operation was selected
   bidir_enable <= '1';                                                --Enable tristate flash data bus signal (write_data_out) to drive flash data bus
   flash_reset <= '1';                                                 --Set the required reset, output enable and write enable for first stage of erase operation
   flash_oe <= '1';
   flash_we <= '0';
   flash_address <= "00" & x"00AAA";                                   --Set the address and data for first stage of erase command cycle
   write_data_out<= x"AA";
   IF (erase_clk_counter < 2) THEN                                     --System sets inputs and holds for 40ns/2 clock cycles (35ns min on data sheet)
   erase_clk_counter <= erase_clk_counter + 1;
   ELSE
   flash_we <= '1';                                                    --Inputs are reset and next data and addresses are set for cycle 2
   flash_address <= "00" & x"00555";
   write_data_out <= x"55";
      IF(erase_clk_counter < 4) THEN                                   --System waits for another 2 clock cycles and enables erase cycle 2
      erase_clk_counter <= erase_clk_counter +1;
      ELSE
      flash_we <= '0';
         IF (erase_clk_counter < 6) THEN
         erase_clk_counter <= erase_clk_counter +1;
         ELSE                                                          --Inputs are reset and next data and addresses are set for cycle 3
         flash_we <= '1';
        flash_we <= '1';
        flash_address <= "00" & x"00AAA";
        write_data_out <= x"80";
           IF(erase_clk_counter < 8) THEN                              --System sets inputs and enables erase cycle 3 for 2 clock cycles (30 ns min)
           erase_clk_counter <= erase_clk_counter +1;
           ELSE
           flash_we <= '0';
              IF (erase_clk_counter < 10) THEN
              erase_clk_counter <= erase_clk_counter +1;
              ELSE
              flash_we <= '1';                                         --Repeated for erase cycle 4
              flash_address <= "00" & x"00AAA";
              write_data_out <= x"AA";
                 IF(erase_clk_counter < 12) THEN
                 erase_clk_counter <= erase_clk_counter +1;
                 ELSE
                 flash_we <= '0';
                    IF (erase_clk_counter < 14) THEN
                    erase_clk_counter <= erase_clk_counter +1;
                    ELSE
                    flash_we <= '1';
                    flash_address <= "00" & x"00555";                  --Repeated for erase cycle 5
                    write_data_out <= x"55";
                       IF(erase_clk_counter < 16) THEN
                       erase_clk_counter <= erase_clk_counter +1;
                       ELSE
                       flash_we <= '0';
                          IF (erase_clk_counter < 18) THEN
                          erase_clk_counter <= erase_clk_counter +1;
                          ELSE
                          flash_we <= '1';                             --Repeated for erase cycle 6
                          flash_address <= "01" & HexBuffer(0) & HexBuffer(1) & HexBuffer(2) & HexBuffer(3) & HexBuffer(4);     --Erase cycle 6 contains address location to be erased, determined
                          write_data_out <= x"30";
                          erase_cycle_complete <= '1';
                             IF(erase_clk_counter < 20) THEN
                             erase_clk_counter <= erase_clk_counter +1;
                             ELSE
                             flash_we <= '0';                          --Sector is erased and inputs are reset
                                IF (erase_clk_counter < 22) THEN
                                erase_clk_counter <= erase_clk_counter +1;
                                ELSE
                                --Clean up--
                                flash_we <= '1';
                                erase_clk_counter <= 0;                --Counters are reset
                                bidir_enable <= '0';                   --Tristate signal disabled and system returns to idle
                                erase_cycle_complete <= '0';
```

*Figure 28 continued onto next page

```
                                Flash_state <= FlashIdle;
                                END IF;
                              END IF;
                            END IF;
                          END IF;
                        END IF;
                      END IF;
                    END IF;
                  END IF;
                END IF;
              END IF;
            END IF;
          END IF;
        END IF;
      END IF;
    END IF;



    WHEN FlashWrite =>                                          --IF user commanded a write operation, the following state is executed
    bidir_enable <= '1';                                        --Enable tristate flash data bus signal (write_data_out) to drive flash data bus
    flash_reset <= '1';                                         --Flash inputs are set
    flash_oe <= '1';
    flash_address <= "00" & x"00AAA";                           --Flash write unlock cycle data and addresses are set
    write_data_out<= x"AA";
      IF (write_clk_counter < 4) THEN
      write_clk_counter <= write_clk_counter + 1;
      ELSE                                                      --System waits for 2 clock cycles to write data to addresses and resets for an exta 2 cycles (35ns min)
      flash_we <= '0';
         IF (write_clk_counter < 8) THEN
          write_clk_counter <= write_clk_counter + 1;
          ELSE                                                  --System sets the second unlock cycle data and addresses
          flash_we <= '1';
          flash_address <= "00" & x"00555";
          write_data_out <= x"55";
           IF(write_clk_counter < 12) THEN                      --Waits 4 clock cycles and write to flash
           write_clk_counter <= write_clk_counter +1;
           ELSE
           flash_we <= '0';
           IF (write_clk_counter < 16) THEN
             write_clk_counter <= write_clk_counter +1;
             ELSE                                               --System sets third write cycle data and addresses
             flash_we <= '1';
             flash_address <= "00" & x"00AAA";
             write_data_out <= x"A0";
               IF(write_clk_counter < 20) THEN
               write_clk_counter <= write_clk_counter +1;
               ELSE                                             --Waits 4 clock cycles and writes to flash
               flash_we <= '0';
                 IF (write_clk_counter < 24) THEN
                 write_clk_counter <= write_clk_counter +1;
                 ELSE                                           --System sets fourth write cycle data and addresses
                 flash_we <= '1';
                 flash_address <= "01" & HexBuffer(0) & HexBuffer(1) & HexBuffer(2) & HexBuffer(3) & HexBuffer(4);          --Addresses are determined by user inputs
                  write_data_out<= HexBuffer(5) & HexBuffer(6);                                                             --Data determined by user inputs
                   write_cycle_complete <= '1';
                   IF(write_clk_counter < 28) THEN
                   write_clk_counter <= write_clk_counter +1;
                   ELSE                                         --System completes cycle
                   flash_we <= '0';
                     IF (write_clk_counter < 32) THEN
                     write_clk_counter <= write_clk_counter +1;
                     ELSE
                     --Clean up--
                     flash_we <= '1';                           --Resets inputs and counters
                     write_clk_counter <= 0;
                     bidir_enable <= '0';                       --Resets tristate signal and returns to idle
                     write_cycle_complete <= '0';
                     Flash_state <= FlashIdle;

                     END IF;
                   END IF;
                 END IF;
               END IF;
             END IF;
           END IF;
         END IF;
       END IF;
     END IF;
   END IF;

    WHEN FlashRead =>                                           --IF user requests a read operation, the following state is executed
    bidir_enable <= '0';                                        --Tristate signal is disabled
    flash_reset <= '1';                                         --Flash inputs are set
    flash_we <= '1';
    flash_oe <= '1';
    flash_address <= "01" & HexBuffer(0) & HexBuffer(1) & HexBuffer(2) & HexBuffer(3) & HexBuffer(4);        --Flash address of memory location to be read is set according to user input
       IF (read_clk_counter < 2) THEN                           --System waits for 2 clock cycles (35ns min) and outputs data in memory
       read_clk_counter <= read_clk_counter + 1;
       ELSE
       flash_oe <= '0';
          IF (read_clk_counter < 4) THEN                        --After 2 clock cycles the data is valid
          read_clk_counter <= read_clk_counter + 1;
          ELSE
          Data_LEDs <= read_data_in;                            --Data on bus is copied to the LEDS & stored as output byte
          Tx_output_byte <= read_data_in;
          read_cycle_complete <= '1';
             IF (read_clk_counter < 6) THEN
             read_clk_counter <= read_clk_counter + 1;
             ELSE
             flash_oe <= '1';                                   --Flash inputs are reset
             read_cycle_complete <= '0';
             read_clk_counter <= 0;                                     --Counters reset and bidirectional signal disabled
             bidir_enable <= '0';
             Flash_state <= FlashIdle;
             END IF;
          END IF;
       END IF;

    WHEN others =>                                              --Error handler to return to idle
    Flash_state <= FlashIdle;

    END CASE;

 END IF;

 END PROCESS Flash_operation;
```

Figure 28: Flash mode operator FSM

```
Transmit_read_result: PROCESS(clock,read_cycle_complete,erase_cycle_complete,write_cycle_complete)    --Process to transmit result of read operation back to user
BEGIN

   IF rising_edge(clock) THEN

         IF(read_cycle_complete = '1') THEN                                                          --Set internal flags if flash processes have been completed
         TxRead <= '1';
            ELSE IF (erase_cycle_complete = '1') THEN
            TxErase <= '1';
               ELSE IF (write_cycle_complete = '1') THEN
               TxWrite <= '1';
               END IF;
            END IF;
         END IF;

         CASE Tx_State IS                                                                            --FSM to transmit message back to user via RS232

         WHEN TxIdle =>
         Tx <= '1';                                                                                  --While in an idle state drive the output line high
         IF (TxRead = '1' OR TxErase = '1' OR TxWrite = '1' OR TxReadMessage = '1') THEN             --When any flash operation completed, move to start state
         Tx_State <= TxStart;
         END IF;

         WHEN TxStart =>
         Tx <='0';                                                                                   --When arriving at start state, set output line low to act as start bit for receiver
         IF (Tx_ClockCounter < Tx_ClocksPerBit) THEN
         Tx_ClockCounter <= Tx_ClockCounter + 1;                                                     --After 1 bit period move to the data state of whichever operation has completed
         Tx_State <= TxStart;
         ELSE
         Tx_ClockCounter <= 0;
            IF(TxRead = '1') THEN
            Tx_State <= TxReadData;
               ELSE IF(TxErase = '1') THEN
               Tx_State <= TxEraseData;
                  ELSE IF(TxWrite = '1') THEN
                  Tx_State <= TxWriteData;
                     ELSE IF(TxReadMessage = '1') THEN
                     Tx_State <= TxReadDataMessage;
                     END IF;
                  END IF;
               END IF;
            END IF;
         END IF;

         WHEN TxEraseData =>                                                                         --If erase operation has been completed
         Current_erase_message <= EraseMessage(EraseMessageIndex);                                   --Cycle through each bit of each erase message letter and transmit it onto the Tx line
         Tx <= Current_erase_message(EraseCharacterIndex);
         IF (Tx_ClockCounter < Tx_ClocksPerBit) THEN
         Tx_ClockCounter <= Tx_ClockCounter +1;
         Tx_State <= TxEraseData;
         ELSE
         Tx_ClockCounter <=0;
            IF(EraseCharacterIndex < 7) THEN                                                           --Go through every letter and increment by 1
            Tx_ClockCounter <= 0;
            EraseCharacterIndex <= EraseCharacterIndex + 1;
            Tx_State <= TxEraseData;
            ELSE                                                                                       --When each byte sent, move to stop state
            Tx_ClockCounter <=0;
            EraseCharacterIndex <= 0;
            Tx_state <= TxStop;
            END IF;
         END IF;

         WHEN TxWriteData =>                                                                           --If write operation has been completed
         Current_write_message <= WriteMessage(WriteMessageIndex);                                     --Cycle through each bit of each write message letter and transmit it onto the Tx line
         Tx <= Current_write_message(WriteCharacterIndex);
         IF (Tx_ClockCounter < Tx_ClocksPerBit) THEN
         Tx_ClockCounter <= Tx_ClockCounter +1;
         Tx_State <= TxWriteData;
         ELSE
         Tx_ClockCounter <=0;
            IF(WriteCharacterIndex < 7) THEN
            Tx_ClockCounter <= 0;
            WriteCharacterIndex <= WriteCharacterIndex + 1;
            Tx_State <= TxWriteData;
            ELSE                                                                                       --When each byte been sent, move to stop state
            Tx_ClockCounter <=0;
            WriteCharacterIndex <= 0;
            Tx_state <= TxStop;
            END IF;
         END IF;


         WHEN TxReadDataMessage =>                                                                     --When if read operation has been completed
         Current_read_message <= ReadMessage(ReadMessageIndex-1);                                      --Cycle through each bit of each read message letter and transmit it onto the Tx line
         Tx <= Current_read_message(ReadCharacterIndex);
         IF (Tx_ClockCounter < Tx_ClocksPerBit) THEN
         Tx_ClockCounter <= Tx_ClockCounter +1;
         Tx_State <= TxReadDataMessage;
         ELSE
         Tx_ClockCounter <=0;
           _IF(ReadCharacterIndex < 7) THEN
            IF(ReadCharacterIndex < 7) THEN
            Tx_ClockCounter <= 0;
            ReadCharacterIndex <= ReadCharacterIndex + 1;
            Tx_State <= TxReadDataMessage;
            ELSE
            Tx_ClockCounter <=0;                                                                     --When each byte has been sent, move to stop state
            ReadCharacterIndex <= 0;
            Tx_state <= TxStop;
            END IF;
         END IF;

         WHEN TxReadData =>                                                  --When in data state, set the output line equal to the bit of data at the current index of the data byte
         Tx <= Tx_output_byte(Tx_DataBit_Index);
         IF (Tx_ClockCounter < Tx_ClocksPerBit) THEN                         --Count for 1 bit period
         Tx_ClockCounter <= Tx_ClockCounter + 1;
         Tx_State <= TxReadData;
         ELSE
         Tx_ClockCounter <=0;
            IF (Tx_DataBit_Index < 7) THEN                                   --Repeat for the remaining 7 bits of the output byte
            Tx_ClockCounter <= 0;
            Tx_DataBit_Index <= Tx_DataBit_Index + 1;
            Tx_State <= TxReadData;
            ELSE                                                             --Once all 8 bits have been transmitted, move to the stop state
            Tx_ClockCounter <= 0;
            Tx_DataBit_Index <= 0;
            TxReadComplete <= '1';
            TxRead <= '0';
            Tx_State <= TxStop;
            END IF;
         END IF;


         WHEN TxStop=>                                                       --When arriving at stop state, drive line high to act as the stop bit for the receiver
         Tx <= '1';
         IF (Tx_ClockCounter < Tx_ClocksPerBit) THEN                         --Count for 1 bit period
         Tx_ClockCounter <= Tx_ClockCounter + 1;
         Tx_State <= TxStop;
         ELSE IF(EraseMessageIndex < 18 AND TxErase = '1') THEN              --System checks whether all bytes have been sent from each success message
         Tx_ClockCounter <= 0;
         EraseMessageIndex <= EraseMessageIndex +1;
         Tx_State <= TxIdle;
         ELSE IF(WriteMessageIndex < 18 AND TxWrite = '1') THEN
         Tx_ClockCounter <= 0;
         WriteMessageIndex <= WriteMessageIndex + 1;
         Tx_State <= TxIdle;
         ELSE IF(ReadMessageIndex < 18 AND TxReadComplete = '1') THEN
```

*Figure 29 continued onto next page

```
620         TxReadMessage <= '1';
621         Tx_ClockCounter <= 0;
622         ReadMessageIndex <= ReadMessageIndex + 1;
623         Tx_State <= TxIdle;
624            ELSE                                                    --If the message are complete reset all counters
625            EraseCharacterIndex <= 0;
626            EraseMessageIndex <= 0;
627            WriteCharacterIndex <= 0;
628            WriteMessageIndex <= 0;
629            ReadCharacterIndex <= 0;
630            ReadMessageIndex <= 0;
631            TxReadComplete <= '0';
632            TxReadMessage <= '0';
633            Tx_ClockCounter <= 0;
634            TxRead <= '0';
635            TxErase <= '0';
636            TxWrite <= '0';
637            Tx_State <= TxIdle;                                       --Return to Idle state
638            END IF;
639         END IF;
640         END IF;
641         END IF;
642
643
644
645         WHEN others =>                                              --Error handler to return to Idle sate
646            Tx_State <= TxIdle;
647
648       END CASE;
649
650    END IF;
651
652 END PROCESS Transmit_read_result;
653
654
655 END rtl;
```

Figure 29: Transmitter interface with success messages

9.3 Calculation of System Performance

$$Total\ time = Time\ to\ receive\ command, Tc + Time\ to\ perform\ operation, To + Time\ to\ transmit\ result, Tr \quad \text{(vi)}$$

$$Tc = Time\ to\ receive\ 1\ byte\ \times inputs\ received \quad \text{(vii)}$$

$$Read: Tt = (87 \times 10^{-6} \times 6) + 160 \times 10^{-9} + 87 \times 10^{-6} = 609.16 \times 10^{-6} \quad \text{(viii)}$$

$$Erase: Tt = (87 \times 10^{-6} \times 6) + 460 \times 10^{-9} = 522.46 \times 10^{-6} \quad \text{(ix)}$$

$$Write: Tt = (87 \times 10^{-6} \times 8) + 320 \times 10^{-9} = 696.32 \times 10^{-6} \quad \text{(x)}$$