Department of Electronic and Telecommunication Engineering

University of Moratuwa

EN4020: Advanced Digital Systems

Serial Bus Protocol Design

Name	**Index number**
R.Y.J. Abeygunawardena	170006N
B.H.M. Imdaad	170232D
M.N.F. Nusha	170415R
K.H.M.T.M.S. Samarakoon	170538V

Supervisor
Dr. Subramaniam Thayaparan

This report is submitted in partial fulfillment of the requirements for the module EN4020: Advanced Digital Systems.

2022-02-26

Abstract

Design and implementation of a serial bus with multiple masters and slaves, arbiter and top-level design that can carry out a specific set of tasks, including data transfer, priority-based and split arbiter transaction & top-level verification.

This project involves the design and implementation of a serial bus with multiple master and slave modules, a centralized bus arbiter and four types of data transfer: read, burst read, write, and burst write. Further, priority-based as well as delay-based split transaction is implemented in this project. The system also has external UART communication capability, with a dedicated master-slave pair for this purpose.

The project was carried out by: (1)Designing the internal bus protocol; (2)Simulation and testing of internal bus; (3)Implementation of the same; and (4)Design, simulation and implementation of the external bus protocol. The design of the internal bus protocol involves designing the master-slave communication, the master-arbiter communication, the top-level communication, as well as the external communication; including specific bit patterns used to convey information via a serial wire. Next, the modules were created using SystemVerilog HDL and tested individually, after which each communication protocol was tested. This testing and debugging was done using Quartus Prime 18.1, and the simulation software ModelSim-Altera as well as QuestaSim. The system bus was then implemented on Cyclone IV Altera DE2-115 FPGA board, with the use of the above-mentioned Quartus Prime software.

This report details the architecture and description of each module, a brief discussion of the code used to implement their functionality, and the internal and external communication protocol. It also presents the waveforms and RAM blocks that are obtained during simulation of the system, and the methodology of implementing the serial bus protocol on an FPGA board.

This report details the architecture and description of each module, a brief discussion of the code used to implement their functionality, and the internal and external communication protocol. It also presents the waveforms and RAM blocks that are obtained during simulation of the system, and the methodology of implementing the serial bus protocol on an FPGA board.

Contents

Acronyms

ACK	acknowledgement.
BRAM	block RAM.
FPGA	Field Programmable Logic Array.
GPIO	General Purpose Input Output.
I/O	input/output.
LCD	liquid crystal display.
NAK	no-acknowledgement.
RAM	random access memory.
Rx	receiver.
Tx	transmitter.
UART	universal asynchronous receiver-transmitter.

List of Figures

List of Tables

1 Introduction

A system bus can be described as a shared communication link between various components of a system. While facilitating communication is a system bus protocols primary purpose, it also allows easy connection of new components.

At a basic level, a system bus can be parallel or serial, depending on the type of wires used for data transmission. It can also be defined as internal or external, based on whether it communicates with external peripheral devices.

A bus can be defined using a set of *design parameters*, which includes the bus connection type, arbitration technique, the timing and synchronization approach used, the bus dimensions as well as the types of data transfer supported by the bus.

In our system, several aspects of the system are parameterized, including the number of master and slave components, the depth of their block RAMs (BRAMs), the data word size etc. Some of these parameters as used in this report, and for the implementation and demonstration are included in the tables below.

1.1 Design

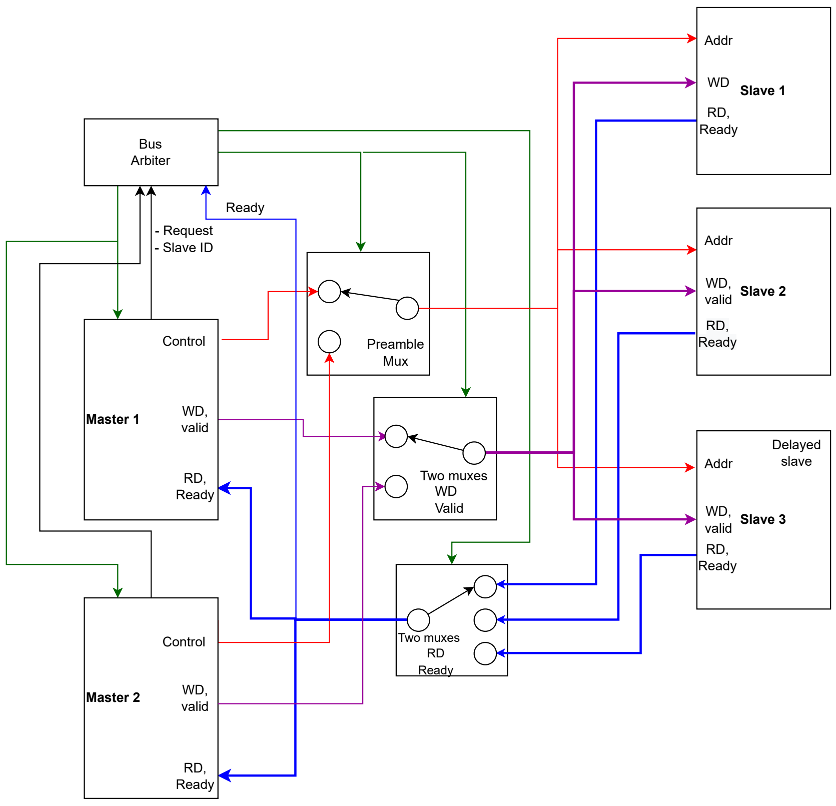


Figure 1.1: System architecture block diagram.

Table 1.1: High-level bus parameters: internal

No. of Masters (Internal)	:	2
No. of Slaves (Internal)	:	3
Memory size of slaves	:	4k, 4k, 2k
Data Transfer Types	:	Read, Write, Read Burst, Write Burst
Arbitration	:	Centralized
Bus Width - Address	:	12 bits
Bus Width - Data	:	16 bits

One of the parameters of the three internal communication ”slave” components is the random access memory (RAM) access *delay*. For demonstration purposes, one of the internal slaves is set to have a considerable delay in RAM access, which causes the bus to be occupied for this duration without any data transfer happening. This occurrence is appropriately handled by the arbiter.

Table 1.2: System components

Name	**Description**	**No. of instances**
Top Module	Allows user to control system, instantiates and connects all the other components	1
Arbiter	Handles the selection of master that is given access to the bus	1
Master	Instantiates and carries out communication via the bus	3
Slave	Receives communication request and carries it out	4

Table 1.3: Slave modules: design parameters

Slave ID	**Depth**	**Delay (RAM read)**
Slave 1	2K	0
Slave 2	2K	0
Slave 3	4K	1200 Clock cycles

The external communication is handled by a dedicated master-slave pair. The system uses universal asynchronous receiver-transmitter (UART) for external communication, and can be connected to two different boards simultaneously; receiving data from one board and sending data to the other. Each of these board-to-board connections is a two-way connection with a pair of UART transmitter-receiver on both ends. A data transfer from an external device to the system is always acknowledged on reception. Similarly, when data is sent from the system, acknowledgement is expected.

Table 1.4: External communication : design parameters

External Comm Protocol	:	UART
Baud Rate	:	19200
Received data display time	:	5s
ACK wait time	:	1ms
No. of re-transmissions	:	5

1.2 Report Overview

Section 2 of this report details the methodology to implement this project in an Field Programmable Logic Array (FPGA), configure the required internal or external communication, and observe the results using the FPGA displays. Section 3 takes a look at each module of the serial bus system in our design; the module as implemented in the Quartus Prime software, main input/output (I/O) ports of the module, its functionality and the state machines used to implement this function. Sections 4 and 5 of this report describe the internal and external communication protocols, with the pseudo code algorithm as well the waveforms obtained by running a testbench simulation. Finally, section 6 is a discussion of the customization and improvements done when implementing this project to be able to have a smoother and more flexible functional serial bus.

2 Methodology

This SystemVerilog design is created such that the following aspects of the serial bus protocol can be tested and demonstrated as required:

Internal communication protocol

Master slave selection:

- Single master - single slave communication
- Multi master - single slave communication
- Multi master - multi slave communication

The above selection can be made to demonstrate:

- Handling *split transaction* when communication with a slow slave is ongoing (sect. 4.2.4).
- Handling high priority master requests during low priority master communication; *priority transfer* (sect. 4.2.2).

The type of data transfer for each of the above communication can be selected:

- Single read
- Burst read
- Single write from master BRAM
- Single write with a user-input data
- Burst write from master BRAM
- Burst write from a user-input data

External communication protocol

- Initialize a communication to send data to another FPGA board.
- Receive and send the communication initiated by another FPGA board.

2.1 Internal communication

First the user needs to provide the requirements to setup the communication variables. These inputs are taken using the 18 switches in the FPGA board in several consecutive states. The requirements and the given value by the user is visible on the LCD display (See figure 2.1). Push buttons are used to go to the next state in order to give the next input.

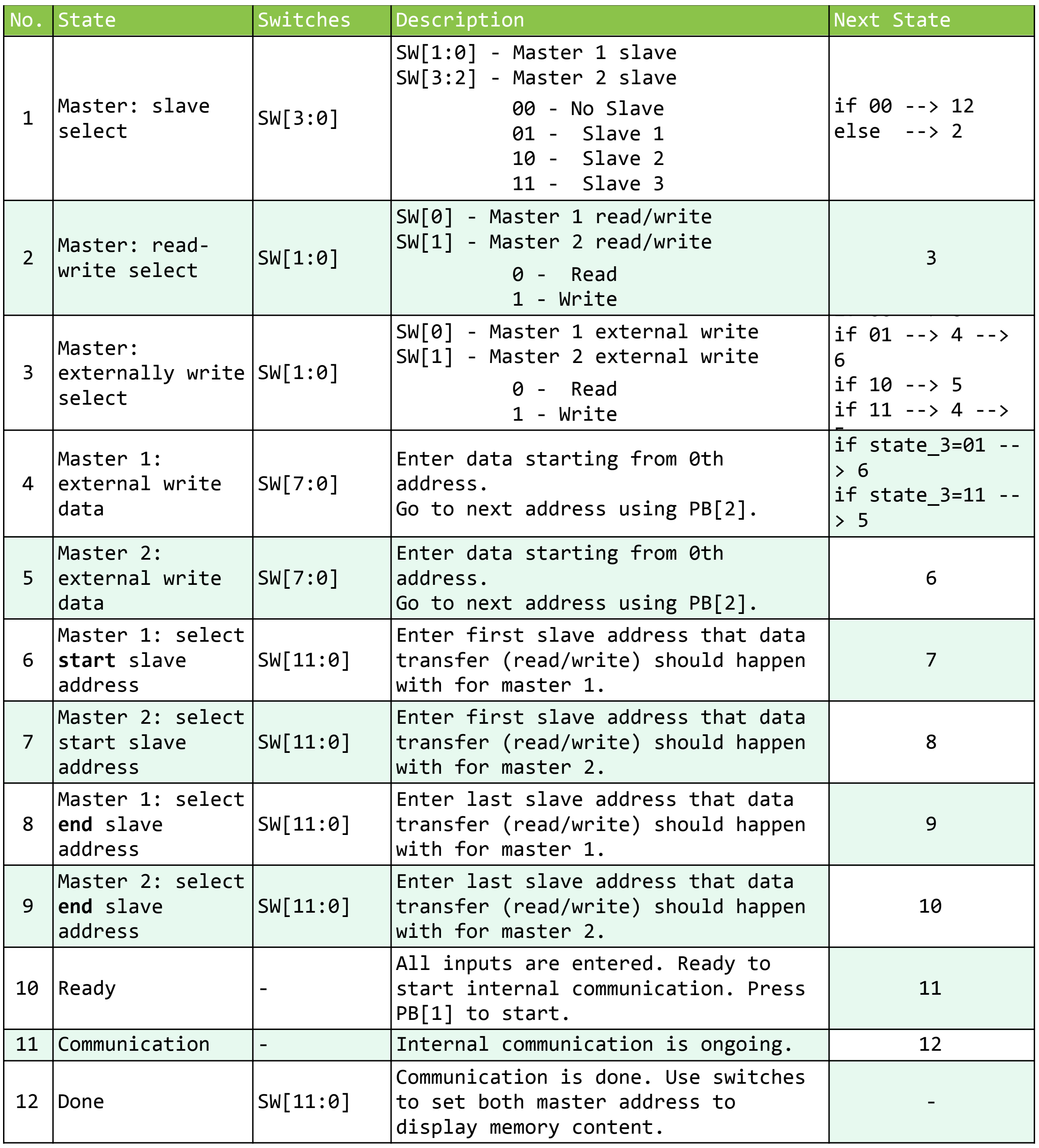
The usage of the push buttons is as follows.

Table 2.1: Push button controls

PB[0]	-	Reset system to initial state
PB[1]	-	Go to next state
PB[2]	-	Go to next address (Used during externally write only)
PB[3]	-	Start external communication

The user inputs can be taken in the appropriate states. The table 2.2 describes the states that the user see during internal communication testing.

Table 2.2: States used to configure FPGA for internal communication



No.	State	Switches	Description	Next State

1	Master: slave select	SW[3:0]	SW[1:0] - Master 1 slave
			SW[3:2] - Master 2 slave
			00 - No Slave
			01 - Slave 1
			10 - Slave 2
			11 - Slave 3
				if 00 --> 12
				else --> 2

2	Master: read-write select	SW[1:0]	SW[0] - Master 1 read/write SW[1] - Master 2 read/write 0 - Read 1 - Write	3

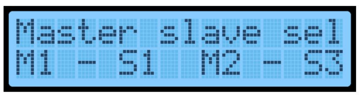
3	Master: externally write select	SW[1:0]	SW[0] - Master 1 external write
			SW[1] - Master 2 external write
			0 - Read
			1 - Write
				if 01 --> 4 --> 6
				if 10 --> 5
				if 11 --> 4 -->

4	Master 1: external write data	SW[7:0]	Enter data starting from 0th address.
			Go to next address using PB[2].
				if state_3=01 --> 6
				if state_3=11 --> 5

5	Master 2: external write data	SW[7:0]	Enter data starting from 0th address. Go to next address using PB[2].	6
6	Master 1: select **start** slave address	SW[11:0]	Enter first slave address that data transfer (read/write) should happen with for master 1.	7
7	Master 2: select start slave address	SW[11:0]	Enter first slave address that data transfer (read/write) should happen with for master 2.	8
8	Master 1: select **end** slave address	SW[11:0]	Enter last slave address that data transfer (read/write) should happen with for master 1.	9
9	Master 2: select **end** slave address	SW[11:0]	Enter last slave address that data transfer (read/write) should happen with for master 2.	10
10	Ready	-	All inputs are entered. Ready to start internal communication. Press PB[1] to start.	11
11	Communication	-	Internal communication is ongoing.	12
12	Done	SW[11:0]	Communication is done. Use switches to set both master address to display memory content.	-

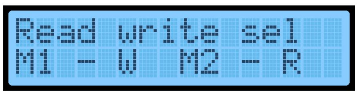
If no slaves are selected for both masters in state 1, the rest of the states will not be activated. Instead, the system directly goes to the "done" state as described in table 2.2.

The user can use this to display the memory content of the masters before initializing the communication. If the user does not select any data transfer that externally changes master's memory content in state 3, states 4 & 5 may not appear accordingly.



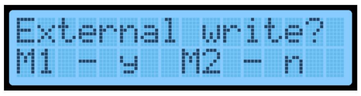
```
Master slave sel
M1 - S1  M2 - S3
```

(a) Master slave select



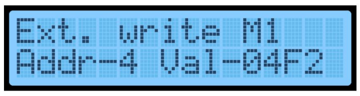
```
Read write sel
M1 - W  M2 - R
```

(b) Read / Write select



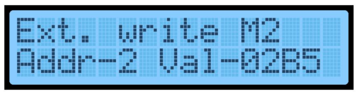
```
External write?
M1 - y  M2 - n
```

(c) External write select



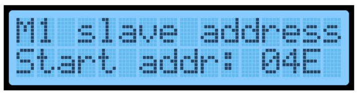
```
Ext. write M1
Addr-4 Val-04F2
```

(d) Master_1 external write



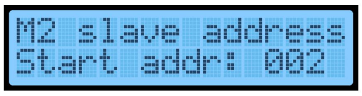
```
Ext. write M2
Addr-2 Val-02B5
```

(e) Master_2 external write



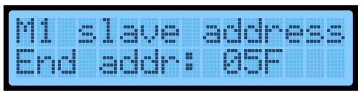
```
M1 slave address
Start addr: 04E
```

(f) M_1 slave start address



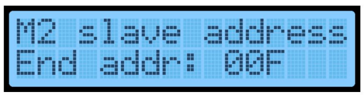
```
M2 slave address
Start addr: 002
```

(g) M_2 slave start address



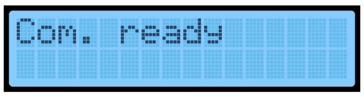
```
M1 slave address
End addr: 05F
```

(h) M_1 slave end address



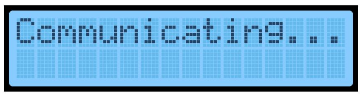
```
M2 slave address
End addr: 00F
```

(i) M_2 slave end address



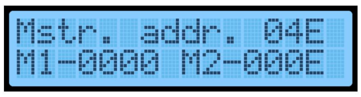
```
Com. ready
```

(j) Ready to start



```
Communicating...
```

(k) Communicating



```
Mstr. addr. 04E
M1-0000 M2-000E
```

(l) Communication over

Figure 2.1: Example LCD display values in different states

2.1.1 Communication between the Top Module and Masters

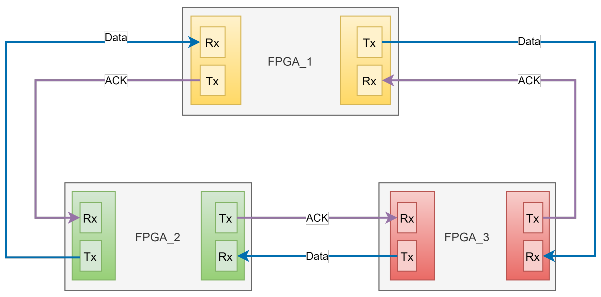
The top module and masters communicate using a parallel bus protocol. It happens in several steps as given below.

- Configuring masters using user provided data.
- Start internal/external communication protocol by a push button press.
- Indicate end of the communication of each master.
- Visualize memory content of masters to the user.
- Visualize data taken from an external device through external communication protocol.

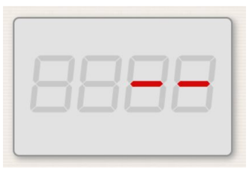
During configuring masters as shown in the figure 2.1, the user provided the necessary data that is used to setup the communication procedure. All the provided details are stored in the top module until the user data acquisition is finished. After that (between state (i) and (j) in figure 2.1) the gathered user details are sent to each master one after the other as detailed in Section 4.1.

2.2 External Communication

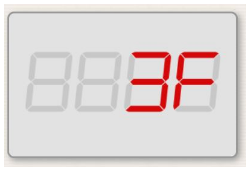
The used external communication protocol is UART protocol. In order to demonstrate, at lease two external FPGAs should be connected. These connected devices also should have the same communication protocol. The following UART specifications should be satisfied by them.



(a) Communication between devices



(b) Default display



(c) Display: $3F$

Figure 2.2: External communication system

- Baud-rate - 19200
- 1 start bit
- 1 end bit
- 8 data bits
- No parity bits

To acknowledge the transmitted device about the received data, a specific bit pattern is used.

- Acknowledgement bit pattern - $8'b11001100$

To connect a device for external communication, the ground pin should be connected to a common ground. Next, the UART should be connected through General Purpose Input Output (GPIO) pins as shown in the sub-figure 2.2a. The GPIO pins are used as follows.

- Rx pins:
  - GPIO[0] - Receive data. (an 8 bit value)
  - GPIO[1] - Transmit acknowledgement for the received data. (Acknowledgement bit pattern)
- Tx pins:
  - GPIO[2] - Receive acknowledgement for transmitted data. (Acknowledgement bit pattern)
  - GPIO[3] - Transmit data. (an 8 bit value)

Connect the Rx data pin (GPIO[0]) and acknowledgement (GPIO[1]) to the Tx data pin and acknowledgement pin respectively of the external device, and vice versa.

To initialize the external communication, first set the initial value (from 0 to 63) using *SW[5:0]* switches. Press *BP[3]* button to start external communication. Then the initialized value will be displayed on the seven segment display for 5 seconds and after increment by one, the value will be sent to the next FPGA board.

When a value from a external FPGA is received by this FPGA board, that value will be displayed on the seven segment for 5 seconds, then increment by one and sends to the other FPGA board via UART protocol.

For further information about external communication protocol, please refer Chapter 5.

3 Modules

3.1 Top Module

This is the top level of the design hierarchy. All other modules are instantiated within this module. It is a state machine which changes its states according to user inputs and internal modules' outputs. This module does following tasks.

1. Act as a user interface to:
    - indicate current state of the overall process.
    - visualize user input requirement to setup master modules before communication process begin.
    - visualize user inserted inputs.
    - visualize the results after the communication process.
2. Instantiate modules related to communication process.

Table 3.1: Modules instantiated in top module

Communication Modules	**User Interface Modules**
Arbiter Module	Debouncers
Master Module	Liquid crystal display (LCD)
Slave Module	Seven segment display
Bus interconnect Module	
External communication Modules	

3.2 User Interface Modules

The design has several modules to interact with the user. There are modules used to get inputs from the user through push buttons and switches in the FPGA board as well as modules to indicate current state and visualize provided inputs by the user.

3.2.1 Debouncer Module

As the clock is 50 MHz, When giving user inputs through push buttons (*KEY[0:3]* in FPGA board), the button keep at pressed position more than 1 clock cycle. If that input is fed to the system as it is, it may lead to corrupted results. This module is used to remove that effect.

Each of the push buttons in the FPGA board are connected to a separate instance of this module. The output of this module is connected to the main system instead of directly connecting push buttons to the system. When user press a button the output will change its state (asynchronously), but it will go back to its initial state just after one clock period. Further more the user is restricted not to give more than one input through same push button within 0.5S of time. This is to ensure high reliability of the overall process.

Table 3.2: Top module inputs and outputs

Ports	Description
CLOCK_50	50 MHz clock used to run the system
KEY[3:0]	Push buttons used to give user inputs. (ex:- start, reset, go to next state/address)
SW[17:0]	Switches used to give user inputs. (ex:- Set the master module memory values, select slave addresses etc.)
LEDG[7:0]	Green LEDs. Used to indicate current state
LEDR[7:0]	Red LEDs. Used to indicate each switch is on or off.
[5 : 0]HEX[1:0]	Seven segment displays. Used to show values related to external communication.
LCD_data	Data to the LCD to show in the display
LCD_RW LCD_EN LCD_RS LCD_BLON LCD_ON	Control LCD module in the board.

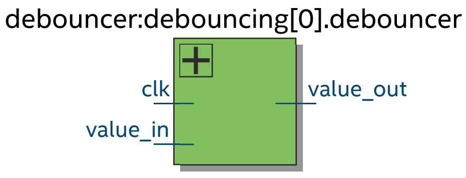


Figure 3.1: Debouncer module

3.2.2 LCD Module

This module is used to give all the indications related to the internal communication protocol.

The figure 2.1 shows example LCD display values in all the states. The values change when user go to the next state/ next address using push buttons or user gives a new input using switches on the FPGA board.

This LCD module consists of several SystemVerilog modules as shown below.

1. **LCD_interface** - Used to interface the LCD module with the main system. Acquire the data needed to be displayed and the control inputs. Generate necessary string patterns as bit streams to send to the display at every state of the main system. Sends data and control signals designed by lower level modules to the LCD on the FPGA board.

2. **LCD_TOP** - Used to process the bit patterns that needed to be displayed at a given time.

3. **LCD_Controller** - Used to control the LCD clock cycle by clock cycle. Process one cell of the display at a time. Gen-

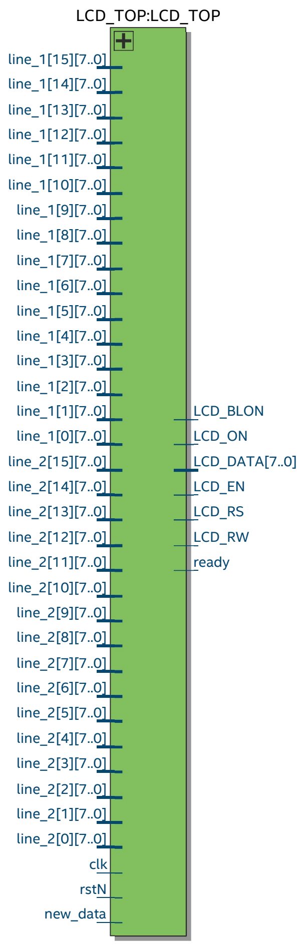
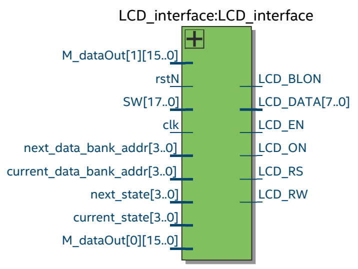
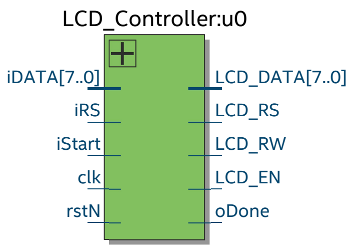


Figure 3.2: LCD display top module

erate necessary control signals and bit patterns to control the LCD.



(a) LCD interface



(b) LCD controller

Figure 3.3: LCD related modules

3.2.3 Hex Display

This module is mainly used to do the indications related to the external communication (with other FPGAs). Only the left most 2 seven segment units are used as only needs to show a 8 bit value. When new value is received from external FPGA board, that value is shown as a hexadecimal number for 5 seconds as shown in figure 2.2c. Rest of the time it will show 2 dash lines only. This module gets the binary number and the control signals to start displaying as inputs. It output the control signals to the seven segment displays on the FPGA board to show the value. The block diagram of the module is shown in figure 3.4.

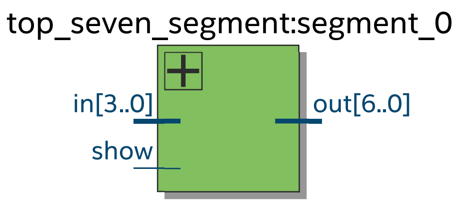


Figure 3.4: Seven segment display block diagram

3.3 Arbiter

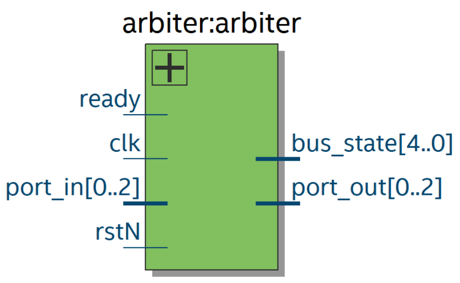


Figure 3.5: Arbiter module

This is the top module of the arbiter which manages the connections between all the master ports with the use of a centralized controller. It is a fully parameterized module to instantiate any number of master ports with any number of slaves.

Table 3.3: Arbiter ports

	Name		Description	Default
from			Inputs	
Top	clk	:	Clock input	-
	rstN	:	Negative edge reset input	H
Master	port_in	:	An unpacked array input which is a combination of wires from all the masters that are connected	-
Slave	ready	:	A single logic wire which informs whether the currently connected internal slave is ready to send data	H
to			Outputs	
Bus Interconnect	bus_state	:	This set of wires output the current bus state in order to drive the bus multiplexers to required path	H

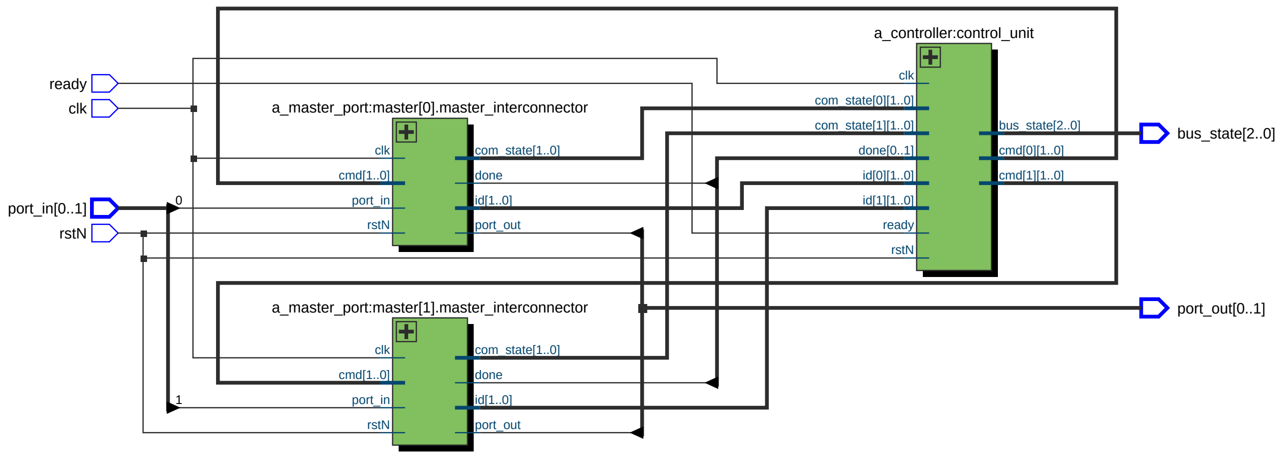


Figure 3.6: arbiter complete module

3.3.1 Controller

This module is the central arbiter controller module who controls the bus depending on the requests comes from masters. This is a 7 layer state machine.

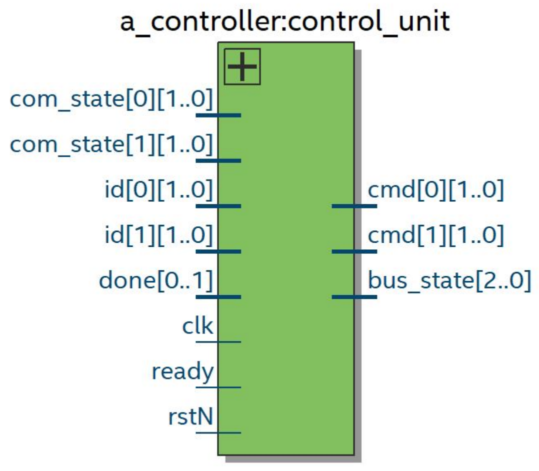


Figure 3.7: Controller module

Table 3.4: Controller ports

Name		Description	Default
		Inputs	
com_state	:	unpacked logic wires which comes from each master port about the current communication state	10-WAIT
id	:	unpacked logic wires which comes from each master port about the requested slave id	00-NO SLAVE
done	:	done state notification from master to inform the controller to handle the bus	L
		Outputs	
cmd	:	unpacked array which goes from controller to each master port, sending information about their requests	-

Table 3.5: Controller state machine

State		Description	Next state
RST	:	Reset the initial parameters to their default values	START
START	:	Listen to requests from masters and update current master and slave parameters	ALLOC
ALLOC	:	Send clear command	ACK
ACK		Wait until master sends the acknowledgment:	
	:	Receive nak	OVER
	:	Receive ack; allocate the bus	COM
COM	:	Interrupt occurs: update the master slave parameters	DONE
	:	Receive the end com command: reset the bus to default	OVER
DONE	:	Wait for the done command	ALLOC
OVER	:	Interrupt mode	ALLOC
	:	Normal mode	START

3.3.2 Master port

This intermediate module will connect each master with the controller. Inside the arbiter this port will be communicating with controller on behalf of the corresponding master. Only essential commands are send to the controller and the rest is taken care by the port. This is a 7 layer state machine (table 3.6.

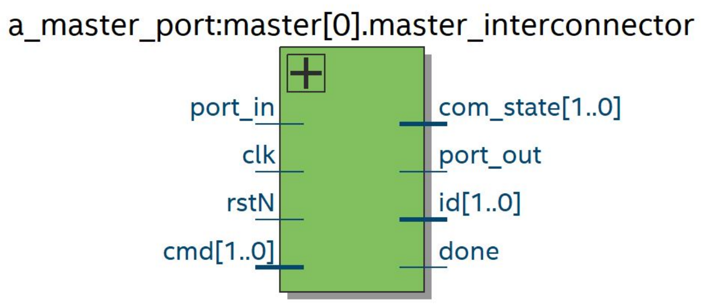


Figure 3.8: Master port

Table 3.6: Master port state machine

State		Description	Next state
RST	:	Reset the initial parameters to their default values	START
START	:	Listen to master request	ALLOC1
ALLOC1	:	Save the slave ID	ALLOC2
ALLOC2	:	Send clear signal to master	ACK
ACK		Wait until master sends the acknowledgment:	
	:	Receive nak	START
	:	Receive ack	COM
COM	:	Receive interrupt stop command	DONE
	:	Receive the communication end signal	START
DONE	:	Receive the done signal	ALLOC2

3.3.3 Priority selector

This module will lookout for any master requests. Depending on the state, controller gets to know for whom the bus should be allocated next.

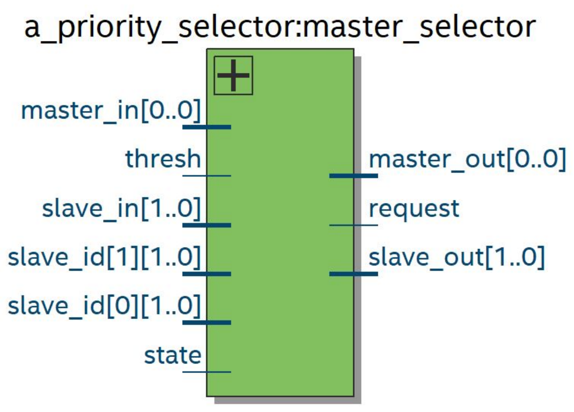


Figure 3.9: Priority selector module

Table 3.7: Priority selector ports

Name		Description	Default
		Inputs	
master_in	:	Current master and current slave for whom the bus is allocated to	-
slave_in	:	Current master and current slave for whom the bus is allocated to	-
state	:	Informs what the current state is: normal state or the interrupt state (split, priority)	0-NRML
thresh	:	Input from threshold counter	-
		Outputs	
master_out	:	Next master for whom the bus should be allocated to	-
slave_out	:	Next slave for whom the bus should be allocated to	-
request	:	Triggers if there is at least one request	L

How this module works :
The priority selector is responsible for instructing the controller to allocate bus interconnect, based on the priority of the master. It functions in two modes:

- Normal state: listen to requests from masters and instruct the controller to allocate the bus to the *highest priority* master that requested connection.
- Interrupt state:
  - If the threshold is not triggered: instruct the controller to allocate the bus to the master that requested if that master's priority is higher than the current master's.
  - If the threshold is triggered: (Split state) instruct the controller to allocate the bus to the requested master if the requested slave is not the current active slave.

3.3.4 Threshold counter

This module measures the delay when the slave that is currently active is in a processing state, with no data transfer happening; i.e. the bus is idle. If the delay is higher than the set threshold value, a different lower priority master will be allocated the bus for communication if any such particular master has requested. (split transaction - sect. 4.2.4).

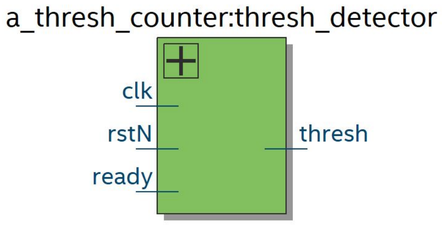


Figure 3.10: Threshold counter module

3.3.5 Write buffer

This shift register module will write the master according to the protocol designed.

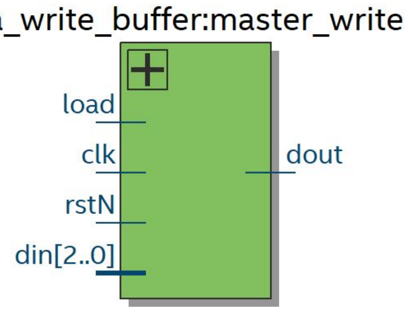


Figure 3.11: Write buffer

Table 3.8: Write buffer module ports.

Name		Description	Default
		Inputs	
load	:	Inform whether there is a control signal to be received	L
din	:	Receive control signals from arbiter	-
		Outputs	
dout	:	Least significant bit of the register is connected to this output port which will be sent to the respective master	-

3.4 Master Module

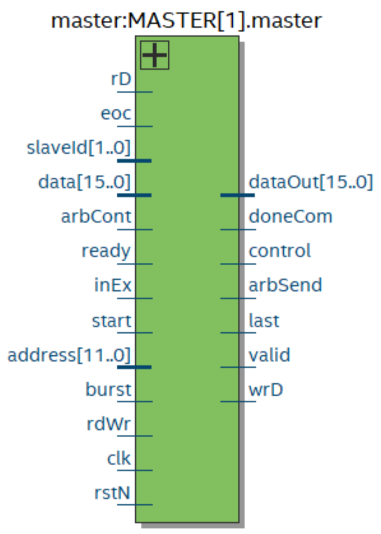


Figure 3.12: Master module

This is the main module in which the master port is implemented. It is also a fully parameterized module which can be used to implement a master module with a memory of any depth and width. The master module has the capability of performing read, write, read-burst, and write-burst. The top module has the full authority on managing on which state the master should function. The table 3.9 discusses the input and output ports of the master module.

The master comprises of three state machines in order to provide the functionalities of user interaction and internal communication. The first state machine which is used to provide user interaction is as follows.

- idle: Waiting for the top module to send control signal to initialize the module
- startConfig: Start of the configuration mode of the master with the top module. The top module will send data to be written into the master if the user requires to do so and describe the mode of functioning for the internal communication.
- startEndConfig: The end of the configuration state of the master with the top module.
- startCom: Internal communication process.
- done: After the internal communication process is over.

Table 3.9: Master ports

	Name		Description	Default
from			Inputs	
Top	clk	:	Clock input.	-
	rstN	:	Negative edge reset input.	H
	burst	:	Used for external burst write from the top module.	-
	rdWr	:	Read or write signal for the internal communication between master and slave.	-
	inEx	:	Used to inform the master whether data will be written from the top module.	-
	data	:	n-bit data wire from top module to master.	-
	address	:	n-bit address wire used to give the start and end address of internal communication.	-
	slaveId	:	Information about which slave the master should communicated is provided through this wire.	-
	start	:	Used by top module to control the configuration of the master prior to internal communication.	L
	eoc	:	Used when it is required for the master to not have any internal communication.	L
Arbiter	arbCont	:	1-bit wire in which the arbiter send control signals to master during internal communication.	L
Slave	rD	:	1-bit wire in which the data read from the slave are sent to the master	-
	ready	:	used by the slave to inform master when to start accepting the data sent through the rD wire	H
to			Outputs	
Top	doneCom	:	Used to inform the top module when the internal communication is over.	L
	dataOut	:	n-bit data wire used to send data to be displayed after the communication is over.	-
Arbiter	arbSend	:	1-bit wire in which the control signal from master to arbiter is sent.	L
Slave	control	:	1-bit wire in which the master sends the control information to the slave during the setup of communication	L
	wrD	:	1-bit wire which is used to stream data to slave during "write" process	-
	valid	:	used by the master to inform the slave when take the data sent through the wrD wire during "write" process	L
	last	:	used by the master to inform the slave during a burst read or write that the last byte is being processed	L

The master module will start the next state machine after the configuration of the master is completed by the top module. This is a sub-state machine which is used to handle the internal communication and interactions between the master and arbiter.

- idleCom: Default state during the start of the internal communication.
- reqCom: Process of requesting arbiter for communication with a slave.

- reqAck: Acknowledgement for the arbiter when it grants communication between the master and the slave.
- masterCom: Process of carrying out read, write, read-burst or write-burst with the slave.
- masterHold: When the arbiter interrupts the communication process due to priority switching in the middle of an exchange of a byte; master's request state to finish the current exchange before pausing the communication process.
- masterDone: Process of informing the arbiter it is clear to hand over the bus to the high priority master for communication.
- masterSplit: State in which a split transaction occurs in the middle of the communication process.
- splitComContinue: Continuation of the communication process after the split transaction.
- over: End of the internal communication.

The next state machine is used inside the masterCom state mentioned above. This state machine controls which type of communication will take place between the master and slave.

- checkState: Checks whether the master should continue with a read, write, read-burst or write-burst.
- controlSignal: Process of sending the control signal to the slave.
- singleRead: Process of doing a single read.
- burstRead: Process of doing a burst read.
- singleWrite: Process of doing a single write.
- burstWrite: Process of doing a burst write.

3.4.1 Block ram

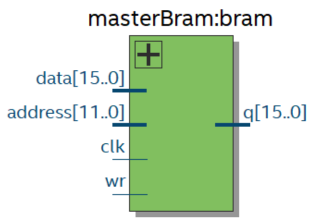


Figure 3.13: BRAM module

This module is used inside the master to instantiate a BRAM of depth and width as required by the top module. The following table shows the input and output port declaration.

Table 3.10: BRAM module ports.

Name		Description	Default
		Inputs	
clk	:	Clock input.	-
wr	:	Read/write enable signal	L
data	:	n-bit wire used to write data of n-width to the bram	-
address	:	m-bit wire used to give the address of the data entry to read/write	-
		Outputs	
q	:	n-bit wire used to read data of n-width from the bram	-

3.5 Slave Module

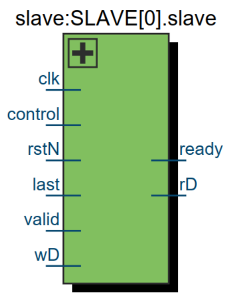


Figure 3.14: Slave module

The slave module acts as the target of the communication intiated by the master. It has the capability to carry out read, write, read burst and write burst operations, which is further discussed in section 4.3. The input and output ports of this module are discussed in table 3.11 below. Further, the module instantiates a BRAM to store and retrieve data.

Table 3.11: Slave Ports

	Name		Description	Default
from			Inputs	
Top	clk	:	Clock input	-
	rstN	:	Negative edge reset input	H
Master	control	:	Receive control signals	-
	wD	:	Receive Write data	-
	valid	:	Check whether Write data is valid	L
	last	:	Check whether current burst Read-/Write byte is last	L
to			Outputs	
Master	rD	:	Send Read data	-
Master, Arbiter	ready	:	Slave ready for communication	H

Other than the clock input for synchronization, and rstN for asynchronous reset function, this module only communicates with the master via the interconnect. The master handles the initiation of communication and the data transfer with this module. It uses a single state machine to implement the data transfer modes mentioned above:

- INIT : Initializes the components of the slave module
- IDLE : Waiting for control signal to initiate data transfer
- RECONFIG : Data transfer mode changed by master during another process; re-configures mode
- CONFIG : Configuring a new data transfer
- CONFIG_NEXT : Processing configuration (ready is set to low during this state)
- READ : Serially transmitting read data to master
- READB_GET : Buffering read data for burst read from memory (ready is low)
- READB : Transferring read data during burst; automatically increments address
- WRITE : Receiving serial write data from master
- WRITEB : Receiving a continuous string of write data from master and storing in memory
- WRITEB_END : Stores the last write burst data

In addition to a state-machine, the slave module makes use of registers; in particular, buffers and counters, to enable serial data transfer. The control signal is sent as a contінous string of data, and it is read and stored using the special registers *config_counter* and *config_buffer*. Read and write are also each done with the use of special registers; *rD_buffer*, *rD_counter*, *wD_buffer*, and *wD_counter*. They both use a common *address* register to access the BRAM. Further, the function of RAM access delay is implemented with the use of *delay_counter*.

The module implementation is fully parameterized for easier modification and handling. These parameters include the memory depth, the data width and the delay expected to be made when accessing the memory. The number of slaves in the system and its own slave ID are also entered as parameters for easier configuration as these values will also be fixed when the system is initialized.

3.6 Bus Inter-connect

This module sets up the connection between the Master and Slave modules, based on the selection of Master and Slave by the arbiter.

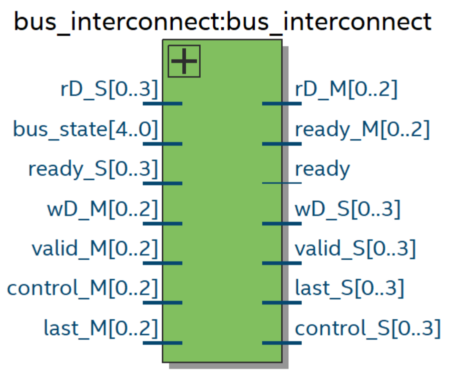


Figure 3.15: Bus interconnect module

Table 3.12: Bus interconnect ports

	Name		Description	Default
from			Inputs	
Arbiter	bus_state	:	Arbiter informs interconnect of which master and slave need to be connected	-
Master	control_M	:	Unpacked array that connects all ”control” input from all master	L
	valid_M	:	Unpacked array that connects all ”valid” input from all master	L
	wD_M	:	Unpacked array that connects all ”wD” or ”write data” input from all master	L
	last_M	:	Unpacked array that connects all ”last” input from all master	L
Slave	ready_S	:	Unpacked array that connects all ”ready” input from all slave	H
	rD_S	:	Unpacked array that connects all ”rD” or ”read data” input from all slave	L
to			Outputs	
Arbiter	ready	:	Informs slave whether the currently connected slave is ready	H
Master	ready_S	:	”ready” input from currently connected slave	H
	rD_S	:	”rD” or ”read data” input from currently connected slave	L
Slave	control_S	:	”control” output to currently connected slave	L
	valid_S	:	”valid” output to currently connected slave	L
	wD_S	:	”wD” or ”write data” output to currently connected slave	L
	last_S	:	”last” output to currently connected slave	L

This module has no clock input, and therefore makes the required connection at the same clock cycle that the arbiter assigns the connection. The module uses *assign* statements to assign the correct master and slave wires to each other. For the other master and slave inputs, it directly assigns the default value.

3.7 External Master Module

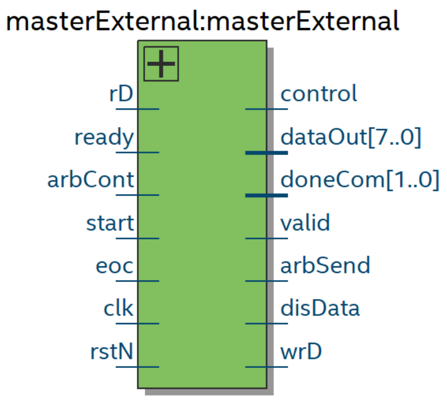


Figure 3.16: External master module

This is the module which is used to communicate with the slave module that communicates with the external boards. Its main functionalities are:

- Initialize external communication when required.
- Display the data (numerical) received through external boards.
- Increment the data received by 1 and send it to the slave that communicates with the external boards.

It is also a fully parameterized module. This module does not support burst communication as the maximum size of the data exchange is of 1-byte. This module comprises of two state machines to carry out the functionalities required out of it. The first state machine mainly handles the interactions between the this module with the top module.

- configMaster: Used by the top module to initialize the module using the top module.
- idle: Intermediate idle state in the case top module wants to stop the external communication from the start.
- write_data: The state in which the master increment the data by one and send it to the slave for external communication.
- read_data: Process of reading the data sent through an external board to current board.
- displayData: The state in which the data is displayed for n seconds.
- end_com: Signifies the end of external communication when it is stopped by external means.

Table 3.13: External Master Ports.

	Name		Description	Default
from			Inputs	
Top	clk	:	Clock input.	-
	rstN	:	Negative edge reset input.	H
	start	:	Used by top module to control the configuration of the External master prior to external communication as well as to initiate the external communication from the current board when required.	L
	eoc	:	Used when it is required for the master to stop external communication.	L
Arbiter	arbCont	:	1-bit wire in which the arbiter send control signals to master during internal communication.	L
Slave	rD	:	1-bit wire in which the data read from the slave are sent to the master	-
	ready	:	used by the slave to inform master when to start accepting the data sent through the rD wire	H
to			Outputs	
Top	doneCom	:	2-bit wire used to inform the top module when the external communication is idle/in progress/over.	2'b00
	dataOut	:	n-bit data wire used to send data to be displayed during external communication.	-
	disData	:	1-bit wire used to inform the top module when the data is available to be displayed	L
Arbiter	arbSend	:	1-bit wire in which the control signal from master to arbiter is sent.	L
Slave	control	:	1-bit wire in which the master sends the control information to the slave during the setup of communication	L
	wrD	:	1-bit wire which is used to stream data to slave during "write" process	-
	valid	:	used by the master to inform the slave when take the data sent through the wrD wire during "write" process	L

The next state machine is used to communicate with the arbiter and carry out the communication between the external slave and master.

- idleCom: Default state during the start of the internal communication.
- reqCom: Process of requesting arbiter for communication with a slave.
- reqAck: Acknowledgement for the arbiter when it grants communication between the master and the slave.
- masterCom: Process of carrying out read, write, read-burst or write-burst with the slave.
- masterHold: When the arbiter interrupts the communication process due to priority switching in the middle of an exchange of a byte; master's request state to finish the current exchange before pausing the communication process.

- masterDone: Process of informing the arbiter it is clear to hand over the bus to the high priority master for communication.
- masterSplit: State in which a split transaction occurs in the middle of the communication process.
- splitComContinue: Continuation of the communication process after the split transaction.
- checkAck: Checks if the external communication was successful or not.
- over: End of the internal communication.

3.8 External Slave Module

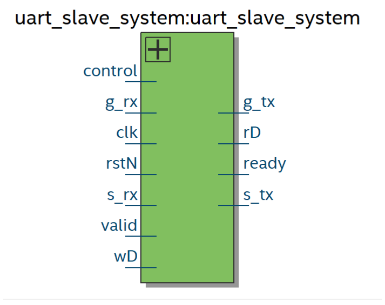


Figure 3.17: External slave module

Table 3.14: External Slave Ports

	Name		Description	Default
from			Inputs	
Top	clk	:	Clock input	-
	rstN	:	Negative edge reset input	H
Master	control	:	Receive control signals	-
	wD	:	Receive Write data	-
	valid	:	Check whether Write data is valid	L
External Devices	g_rx	:	Get_rx - receive serial data	-
	s_rx	:	Send_rx - receive acknowledgement when data is sentT	-
to			Outputs	
Master	rD	:	Send Read data	-
	ready	:	Slave ready for communication	H
External Devices	g_tx	:	Get_tx - send acknowledgement	-
	s_tx	:	Send_tx - send data to be transmitted	-

The external communication slave consists of two sets of UART receiver (Rx) and transmitter (Tx) modules responsible for handling communication with external devices via UART protocol, a baud rate generator and the uart_slave module that handles the state machine and communication between the UART modules and the internal master.

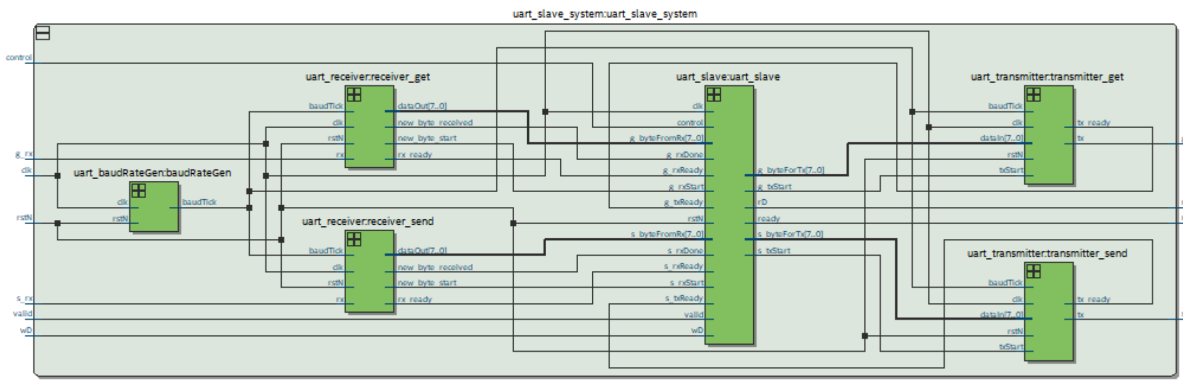

uart_slave_system:uart_slave_system
uart_receiver:receiver_get
uart_baudRateGen:baudRateGen
uart_receiver:receiver_send
uart_slave:uart_slave
uart_transmitter:transmitter_get
uart_transmitter:transmitter_send

Figure 3.18: External slave module: Components

3.8.1 UART-to-Internal Slave Module

This module communicates with the internal master similar to the internal slave-to-master communication. However, it does not have a BRAM or memory component, and considering several factors, was only implemented with single-read and single-write capability. They include:

1. By design, the time taken for a complete UART transmission or reception is significantly larger than that taken for an internal communication. This leads to a long-time occupation of the bus, making a split transfer (sect. 4.2.4) highly likely.
2. The external communication modules are designed such that the master ?? sets the slave to read mode whenever a write is not happening, so that any incoming data can be read.
3. Unlike the internal slave, this system is set up to continue communication with external devices whenever the bus is busy with other communication. This is possible because the external master and slave will only ever communicate with each other.

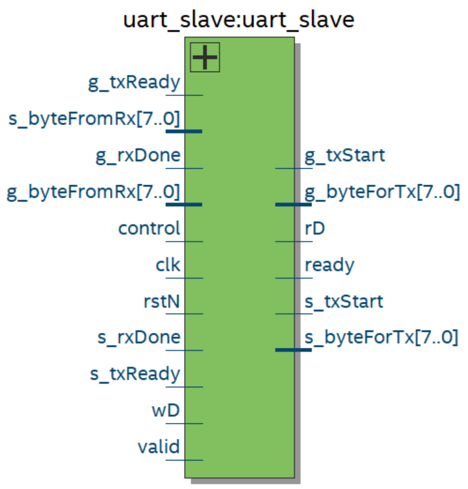


Figure 3.19: UART-to-internal slave module

Due to the above reasons, the slave will be constantly in a state of read or write, unless the master sends an instruction otherwise, which makes a burst transfer meaningless. Instead, the slave will carry out the read/write and wait for the master to initiate a different transfer as required.

Similar to the internal slave module, this module uses a single state machine to implement its design. The states used are as follows:

Table 3.15: States of UART slave module

Name	Description	Next State
INIT	At system initialization, sets values for the logic registers & outputs of the module	IDLE
IDLE	Wait for the control signal from master for a data transfer	Default state
RECONFIG	Triggered by receiving a control high (1). Reconfigures state when control signal is received	CONFIG_NEXT or any state
CONFIG_NEXT	When full instruction is received, processes the control signal and waits for comm. state to be comm. or ready from UART	SEND_ACK or WRITE or IDLE
SEND_ACK	When 'read' data is received from UART Rx, sends the acknowledgement that data was received through UART Tx	READ
READ	Sends data received by UART Rx to master	IDLE
WRITE	After write instruction, if master in comm. mode, receives data from master and sends via UART Tx	GET_ACK
GET_ACK	Waits to receive acknowledgement after a Write; re-transmits up to 5 times in case it is not sent	CHECK_ACK
CHECK_ACK	If a response is received from the data-out UART, checks if it is an ACK	IDLE

Reconfiguring the slave

1. The slave module may receive a control bit pattern from the master module during any of the above modules. This could be a 'HOLD' bit pattern, a 'CONTINUE' bit pattern, or a new read/write data transfer bit pattern.

2. The module goes to RECONFIG state and receives the control bit pattern. It then stores this control bit pattern in the *config_buffer* and returns to the previous state to finish carrying out what it was doing. Here, the slave sets ready low and changes the *control_state* to *stored*.

3. If the control instruction stored is a termination of connection i.e., a HOLD, the slave completes the task, but waits for CONTINUE to resume communication/data transfer with the master.

4. However, if it is a data transfer initiation, the slave carries out this stored read/write initiation after completing the current task.

3.8.2 UART Modules

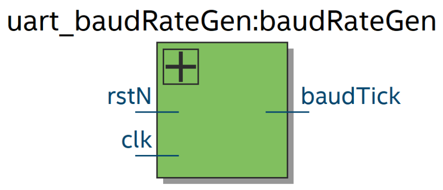


Figure 3.20: Baud rate generator module

The UART baud rate generator module is responsible for generating a clock that can provide timing signals for UART transmission and reception. The baud rate used in this project is 19200.

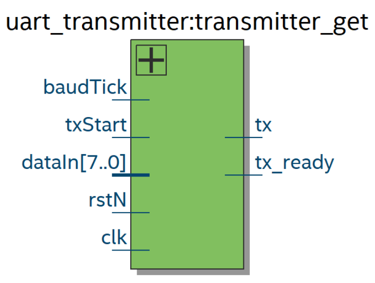
Table 3.16: UART transmitter ports.

Name		Description	Default
		Inputs	
baudTick	:	Receive timing information	L
txStart	:	Signal that current dataIn is valid for transmission	L
dataIn	:	Byte-sized array of wires that sends data for transmission	-
		Outputs	
tx_ready	:	informs when module is available for communication	H
tx	:	sends out data	-

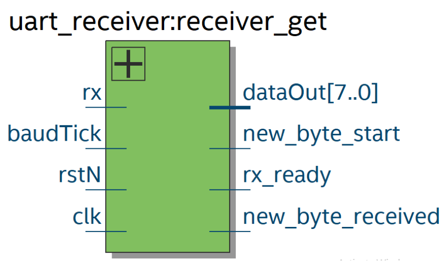
Table 3.17: UART receiver ports.

Name		Description	Default
		Inputs	
baudTick	:	Receive timing information	L
rx	:	Receive UART data	-
		Outputs	
rx_ready	:	Informs module is ready to receive data	H
dataOut	:	Byte-sized array of wires that contains data that was received	-
new_byte_start	:	Signals start of receiving new byte	L
new_byte_received	:	Signals that the full new byte has been received	L

This system consists of two sets of UART modules; one to transmit data and the other to receive data. The system cannot receive and transmit at the same time, as the data-in (read) mode or data-out (write) mode will be initiated by the master. The protocol is set up, however, to set the system at a default data-in (read) mode.



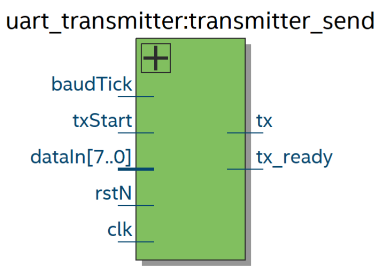
(a) Data-in Transmitter



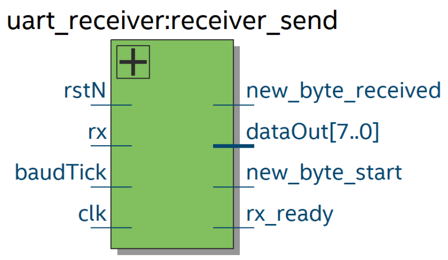
(b) Data-in Receiver

Figure 3.21: Data-in UART modules

When the slave is in data-in (read) mode, it waits for the UART receiver to get a full byte of data, after which the slave reads the data and sends an acknowledgement to the same device via the UART transmitter.



(a) Data-out Transmitter



(b) Data-out Receiver

Figure 3.22: Data-out UART modules

The data-out UART modules are used when the external_slave is in write mode; the data sent by the master is transmitted to the external device through the UART Tx and the acknowledgement from the same device is received through the Rx module.

4 Internal Communication protocol

4.1 Top-Master

The user input that is configured via push buttons and switches, as discussed in Section 2.1.1 is implemented using the logic shown below.

```
initiate Master config
for each Master
    State I
    (1) Start : HIGH (for one clock cycle)
    (2) Send Slave_ID.
    (3) Send action (read/write).
    (4) Send Slave's starting address.
    (5) Send first data packet set by the user.

    if (burst read or burst write)
    State II
    (1) Start : LOW
    (2) Send next data packet set by the user. (once per N clock
    cycles)

    State III
    (1) Start : LOW (for one clock cycle)
    (2) Send Slave's last address.

end Master config
```

The module transfers information regarding Read/Write action, the Slave_ID initially, as well as the data input by the user (if there is one) initially. Next, in case there is a Burst Read/Write action, the top module proceed to state 2 as shown, followed by the prompt to finish the Burst Read/Write action in state 3.

4.2 Master-Arbiter

The main wires interconnecting the master and arbiter are as follows:

Table 4.1: Master - Arbiter module ports.

Name		Description	Default
		Arbiter to Master	
arbCont	:	1-bit wire in which the arbiter send control signals to master during internal communication.	L
		Master to Arbiter	
arbSend	:	1-bit wire in which the control signal from master to arbiter is sent.	L

In order to start the communication with a particular slave the master has to first request the arbiter. The master will send "REQUEST" signal to the arbiter requesting for communication. This consists of 3 1's followed by the slave ID of the slave in which the master wants communicate with. The arbiter will cache this request and allocate the bus for the master depending on the priority and the availability of the bus. The arbiter will send a "CLEAR" signal to master to inform the allocation of the bus. The master will respond with an "ACK" signal to acknowledge the received "CLEAR" signal and begin the communication with slave as explain in the section 4.3.1.

111	Slave_Id

Figure 4.1: Master to arbiter request.

The general communication process between the master and arbiter is explained in the section 4.2.1.

When a master requests the arbiter for communication with a slave, when the bus is occupied by another master, three main scenarios can occur. If the master that requests has a higher priority, the bus would be allocated to the new master that requested for communication; while asking the master that was on communication to hold it's communication. This procedure is explained in the section 4.2.2.

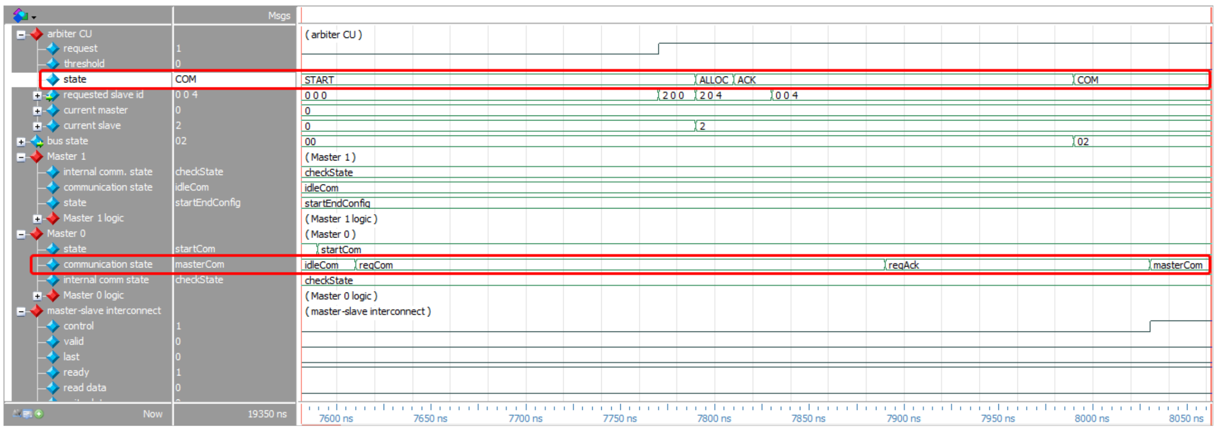
In the case where the master that request for communication has LOW priority, bus will only be allocated to master if it is possible for it to have a split transaction; else, the arbiter will wait till the bus becomes idle to allocate it to the next master who has higher priority. This procedure is explained in the section 4.2.4.

4.2.1 General

```
initiate REQUEST                              : Master
    Sends request Start|Slave_ID
if bus IDLE and not allocated                 : Arbiter
    (1) Send CLEAR (110)
    (2) Set arbCont LOW
else if bus NOT IDLE                          : Arbiter
    if priority of the master is HIGH
       //Follow the procedure for priority transaction
    else
        Wait till bus becomes IDLE
else if bus IDLE and allocated                :Arbiter
    if bus idle for time =< (split_threshold)
        Continue
    if bus idle for time > (split_threshold)
        Follow the procedure for split transaction
    else
        Wait till bus becomes IDLE
if CLEAR                                      : Master
    (1) Send ACK (101)
    (2) Set arbSend HIGH
```

```
     (3) Wait for ACK_received
if ACK                                      : Arbiter
     (1) Send ACK_received
     (2) Set arbCont HIGH
     (3) Allocate the bus to the master
if ACK_received                             : Master
     (1) Set arbSend HIGH
     (2) Start communication with the slave
end REQUEST
```

The wave diagram of the simulation related to state changes on arbiter and master for the above request procedure is shown in the following figure 4.2.



```
                                Msgs
arbiter CU                                   ( arbiter CU )
  request                       1
  threshold                     0
  state                         COM          START  ALLOC  ACK  COM
  requested slave id            0 0 4        0 0 0  2 0 0  2 0 4  0 0 4
  current master                0            0
  current slave                 2            0  2
bus state                       02           00  02
Master 1                                     ( Master 1 )
  internal comm. state          checkState   checkState
  communication state           idleCom      idleCom
  state                         startEndConfig  startEndConfig
  Master 1 logic                             ( Master 1 logic )
Master 0                                     ( Master 0 )
  state                         startCom     startCom
  communication state           masterCom    idleCom  reqCom  reqAck  masterCom
  internal comm state           checkState   checkState
  Master 0 logic                             ( Master 0 logic )
master-slave interconnect                    ( master-slave interconnect )
  control                       1
  valid                         0
  last                          0
  ready                         1
  read data                     0
Now                     19350 ns     7600 ns  7650 ns  7700 ns  7750 ns  7800 ns  7850 ns  7900 ns  7950 ns  8000 ns  8050 ns
```

Figure 4.2: Request: Master initiate the request

After the communication with the slave is over, the master will end the communication by sending the arbiter the control signal OVER (011). After that the master will set arbSend to LOW and goes to END COM state.

```
initiate END COM                            : Master
     (1) Sends OVER
     (2) Go to END COM State
if OVER                                     : Arbiter
     (1) Terminates the connection between the master and slave
     (2) Reset the bus and goes to the begin state and then
     starts listening to master requests
end
END COM
```

The wave diagram of the simulation related to state changes on arbiter and master for the above communication *END procedure* is shown in the following figure 4.3.

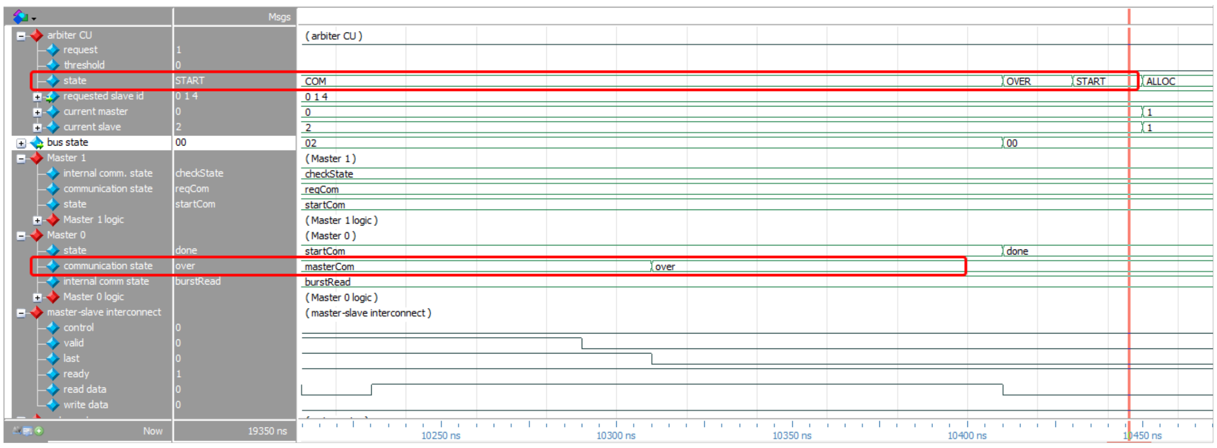


Figure 4.3: End COM: master inform the arbiter about the end of com.

4.2.2 Priority Select

```
begin PRIORITY SELECT
if bus NOT IDLE                               : Arbiter
    if priority of the master is HIGH
        (1) send priority stop (000) to the LOW priority master
        (2) waits for DONE from low priority master
        if DONE received                      : Arbiter
            (1) Send CLEAR (110) to HIGH priority master
            (2) Follow the General procedure
    else
        Wait till bus becomes IDLE
else if bus IDLE and allocated                : Arbiter
    if bus idle for time =< (split_threshold)
        Continue
    if bus idle for time > (split_threshold)
        (1) follow the procedure for split transaction
    else
        Wait till bus becomes IDLE
end PRIORITY SELECT
```

The complete wave diagram of the simulation related to state changes on arbiter and master for the above *PRIORITY* communication is shown in the following figure 4.4.

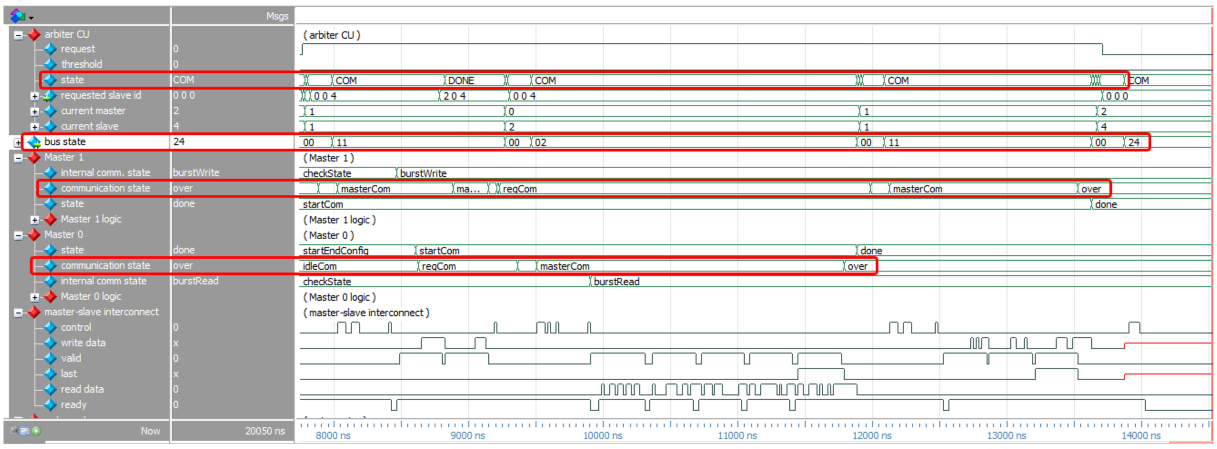


Figure 4.4: Priority SELECT: Waveform of priority com.

When a priority transaction takes place, the arbiter will inform the master that is using the bus to pause the communication in order to allocate the bus for the high priority master. The communication process between the arbiter and the master that is currently utilizing the bus is as follows:

```
initiate PRIORITY Interrupt                  : Arbiter
    Sends master PRIORITY STOP (000) control signal
if PRIORITY STOP is received                 : Master
    if the communication is in the middle of a byte
      and not the last byte
        (1) Send HOLD to arbiter
        (2) Finish the current byte
        (3) Pause the communication with the slave
        (4) Send DONE to arbiter
    if the communication is in the middle of a byte
      and it is the last byte
        (1) Send HOLD to arbiter
        (2) Finish the last byte
        (3) Follow the step for END COM described above
    if DONE received                         : Arbiter
        (1) Terminate the current bus allocation
        (2) send CLEAR (110) to HIGH priority master
        (3) Follow  the General procedure
end PRIORITY Interrupt
```

The wave diagram of the simulation related to state changes on arbiter and master for the above *PRIORITY interrupt* communication is shown in the following figure 4.5.

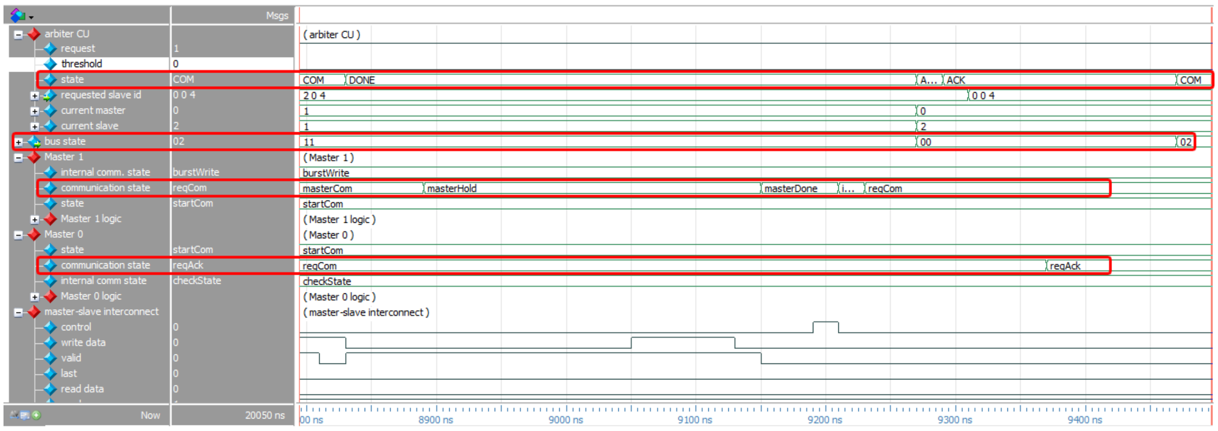


Figure 4.5: Priority SELECT: High priority master interrupt com.

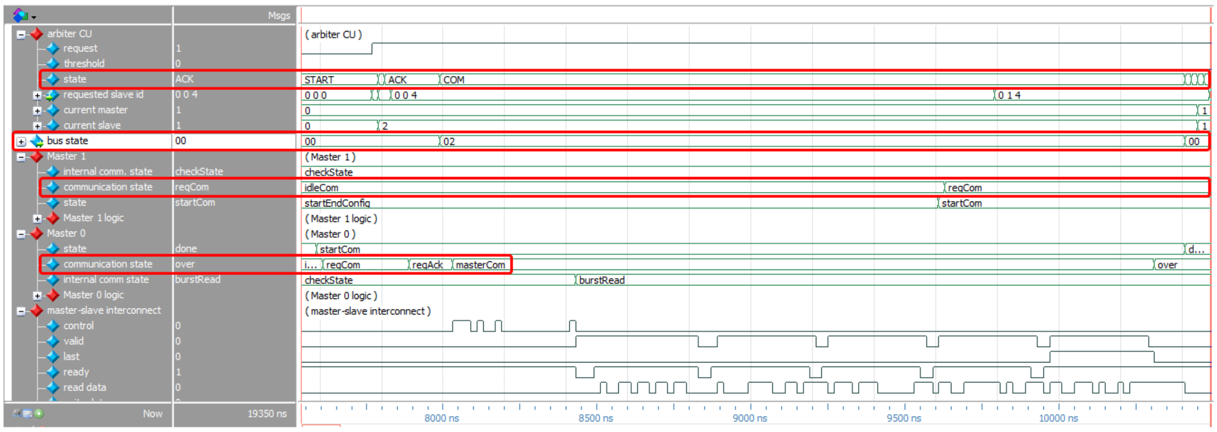


Figure 4.6: Priority SELECT: Low priority master interrupt com.

Figure4.7 shows the waveform of an instance of priority transfer, including the master-arbiter communication and the corresponding master-slave communication for an instance where a read burst being carried out by a low priority master is interrupted by a high priority master (which is carrying out a write burst in a different slave). This image also shows the relevant logic, states and data/control signals of the involved slaves, masters, and arbiter.

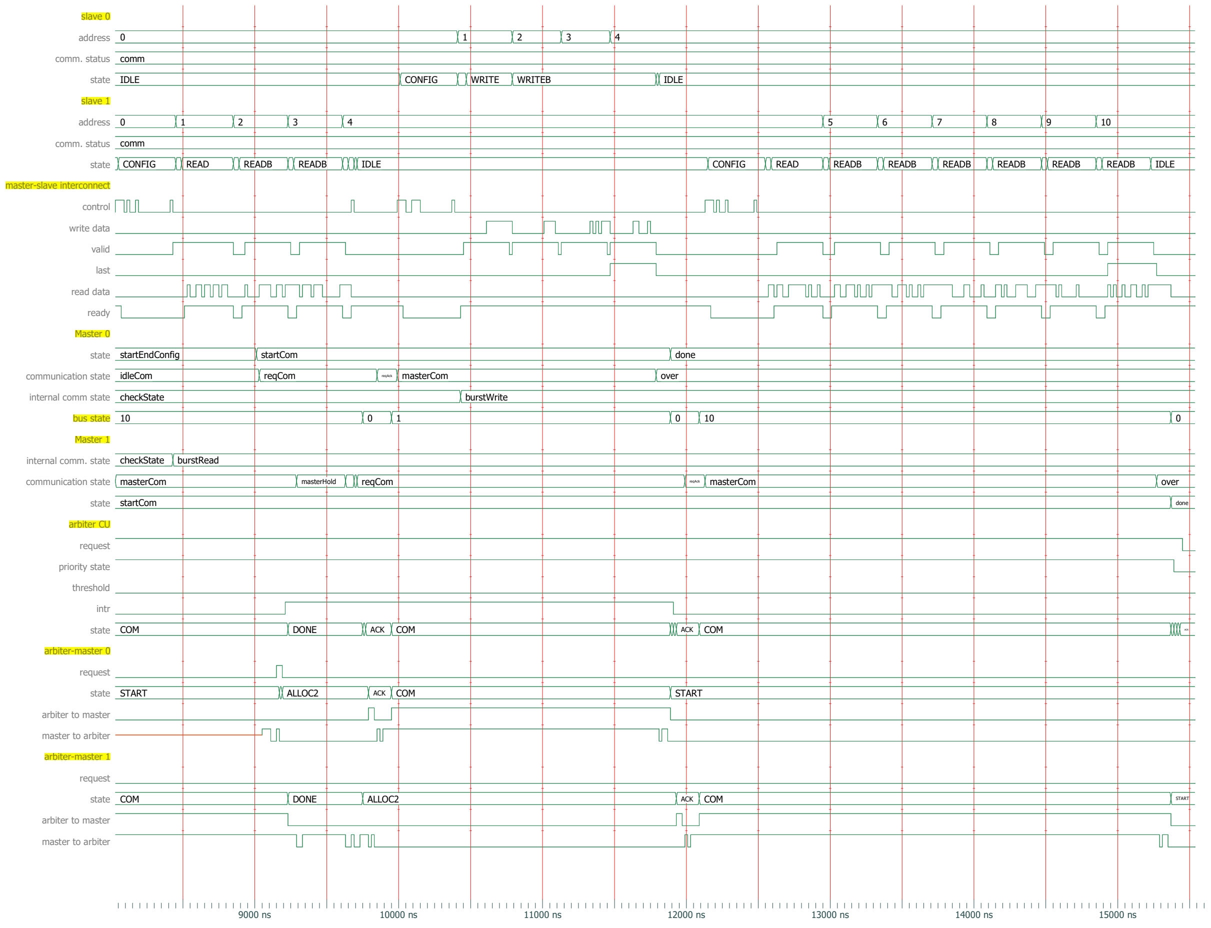


Figure 4.7: Priority transfer: Waveform

4.2.3 Split Transaction

```
begin SPLIT
if bus IDLE but allocated to another master : Arbiter
    if bus idle for time =< (split_threshold)
        Continue
    if bus idle for time > (split_threshold)
        (1) Send SPLIT STOP (010) to the master that
            is currently utilizing the bus
        (2) Wait for DONE
        if DONE received                    : Arbiter
        (1) Terminate the current bus allocation
        (2) send CLEAR (110) to new master
        (3) Follow  the General procedure
    else
        Wait till bus becomes IDLE
end SPLIT
```

The complete wave diagram of the simulation related to state changes on arbiter and master for the above *SPLIT* communication is shown in the following figure 4.8

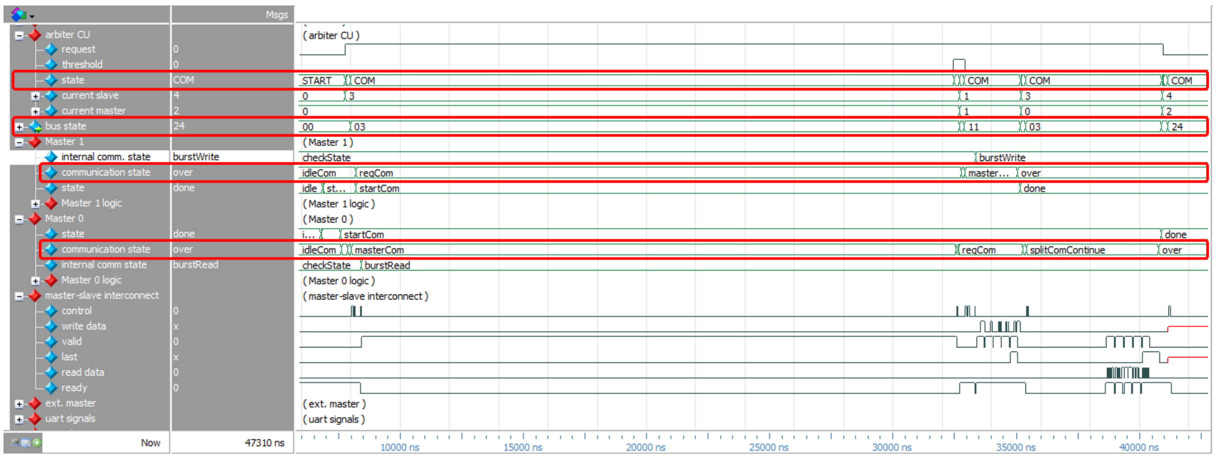


Figure 4.8: SPLIT: end-end split transaction

In order to notify the current master that is occupying the bus, the arbiter will send a "SPLIT STOP" to notify the master to pause the communication and hand over the bus. The procedure of it is as follows:

4.2.4 Split Continue

```
initiate SPLIT Continue                    : Arbiter
    Send CLEAR OLD (100)
if CLEAR OLD                               : Master
    Send ACK (101)
if ACK received                            : Arbiter
    Allocate the bus for the old master
end SPLIT Continue
```

The wave diagram of the simulation related to state changes on arbiter and master for the above *SPLIT* communication is shown in the following figure 4.9

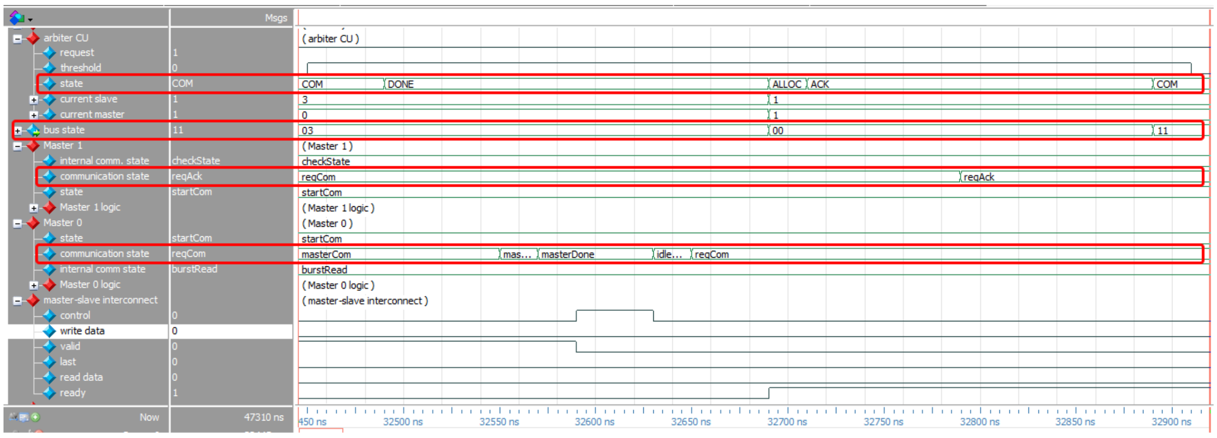


Figure 4.9: SPLIT: pause the current master

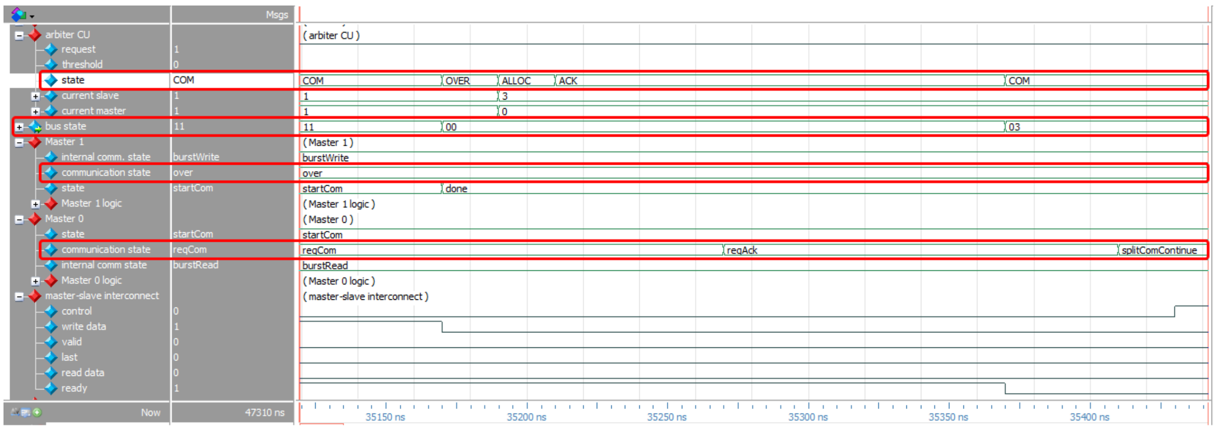


Figure 4.10: SPLIT: handover the bus to the first master

4.3 Master-Slave

The main wires interconnecting the master and slave via the interconnect are discussed above; they include:

Table 4.2: Master-Slave Communication Signals

Name		Description	Default
		Master to Slave	
control	:	Used to set up and terminate communication	-
wD	:	Used to transmit data to be written in slave	-
valid	:	Used to signal whether wD is valid	L
last	:	Used to signal the last byte in a burst communication	L
		Slave to Master	
rD	:	Used to transmit data being read from slave	-
ready	:	Used to signal that slave is ready for communication and that rD is valid	H

4.3.1 Initiating Communication

In order to initiate communication with a slave, master sends a control signal "START" which consists of 3 1's. After this, the master sends the slave ID to of the slave to which it expects to communicate, followed by one bit signalling whether the communication is a Read or Write, and another bit informing whether it is a Burst communication or not. Finally, the address in the slave memory that the master wants to initially communicate with is sent.

Start (111)	Slave_Id	R/W	B	Initial Address

Figure 4.11: Sequence of Control Signal

Figure 4.12: figure caption

4.3.2 Read

The Read operation, when initiated by the master, consists of reading some value stored in the slave's memory and storing it in the master. Slave uses "rD" wire which is set aside to send Read data serially, and the "ready" wire is used to signal the availability and correctness of the rD output.

```
initiate read                         : Master
    Send control Start|Slave_ID|00|address
if read data in buffer                : Slave
    (1) Set ready HIGH
    (2) Start sending read data
if ready HIGH                         : Master
    (1) Start reading read data
    (2) Set valid HIGH at next clock cycle
    (3) Read data into buffer
    (4) Store read data in BRAM
end read
```

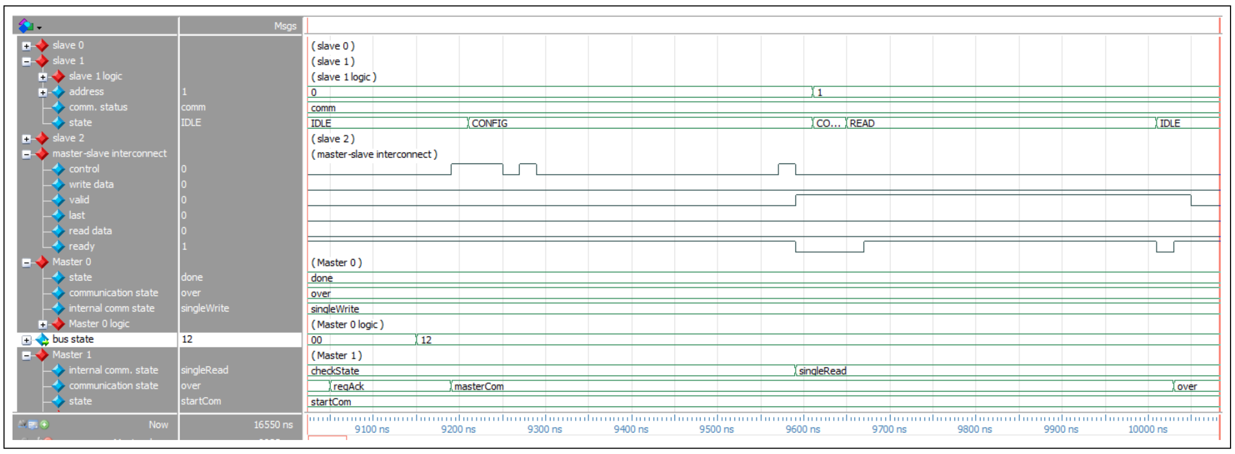


Figure 4.13: Simulation of read between master and slave

4.3.3 Read Burst

Read burst operation is used in internal communication to Read a set of data from a memory block in slave BRAM, without having to enter every address in that block. Only the initial address is sent to the slave; the end of address is signalled by setting the "last" wire to high.

```
initiate read burst                   : Master
    Send control Start|Slave_ID|01|address
if not last read burst address
    Set last LOW
    if read data in buffer            : Slave
        (1) Set ready HIGH
        (2) Start sending read data
```

```
          (3) Increment address
          (4) Send next data
     if ready HIGH                      : Master
          (1) Start reading read data
          (2) Set valid HIGH at next clock cycle
          (3) Read data into buffer
          (4) Store read data in BRAM
          (5) Read next data
          (6) Increment address and Store
if last read burst address              : Master
     Set last HIGH
     Repeat read for one more data byte
end read burst
```

Simulation

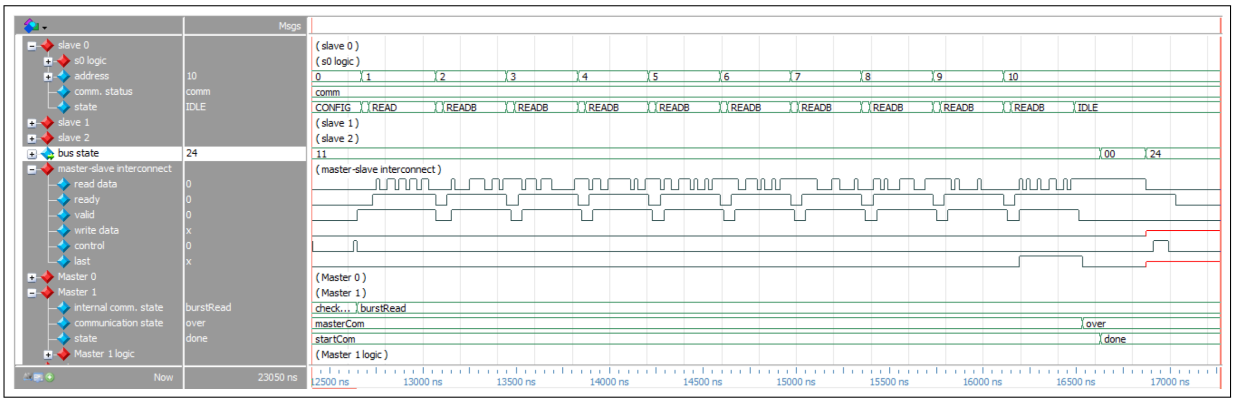
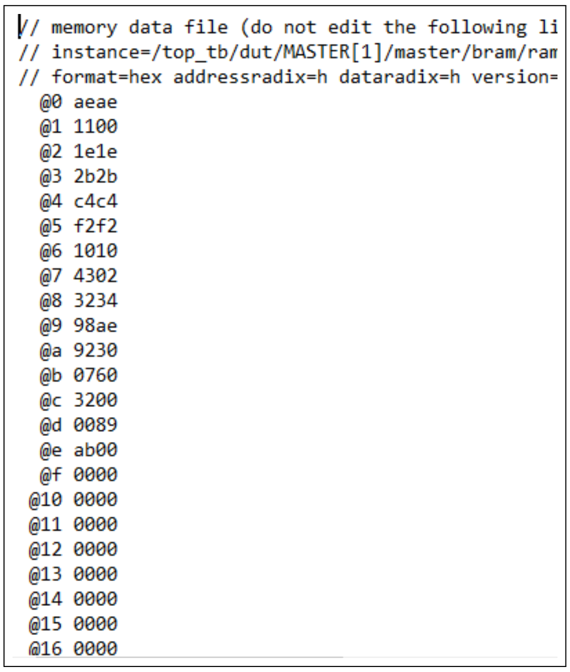
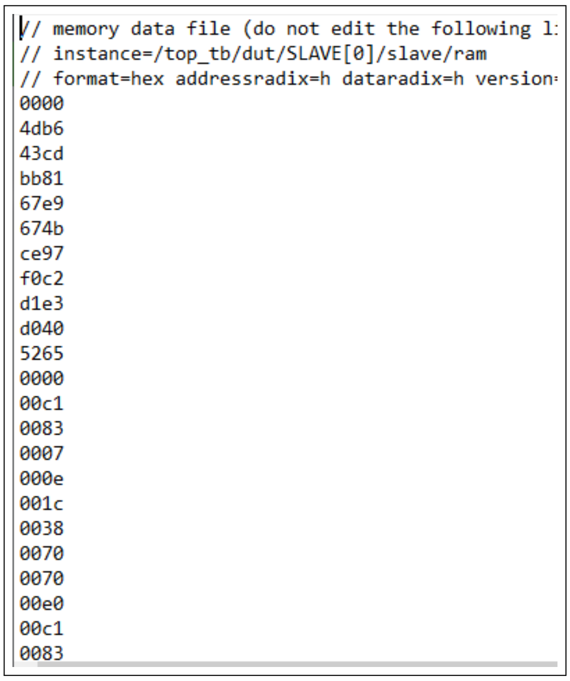


Figure 4.14: Simulation of read burst between master and slave



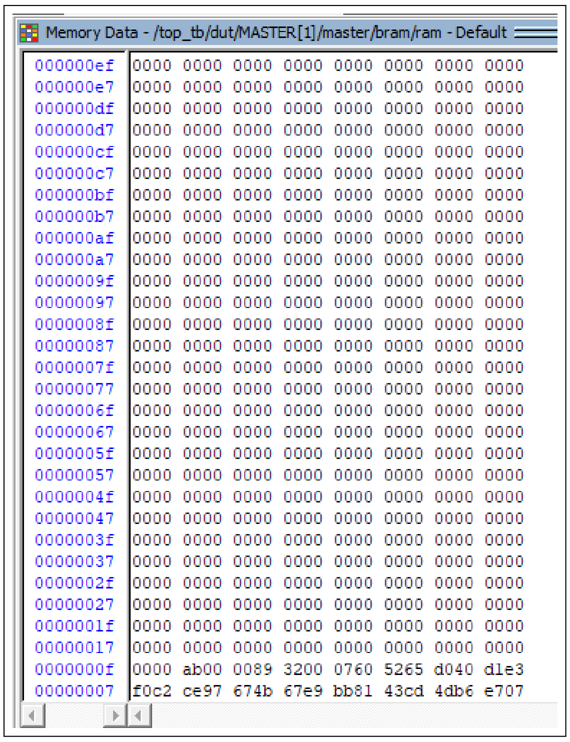
```
// memory data file (do not edit the following li
// instance=/top_tb/dut/MASTER[1]/master/bram/ram
// format=hex addressradix=h dataradix=h version=
  @0 aeae
  @1 1100
  @2 1e1e
  @3 2b2b
  @4 c4c4
  @5 f2f2
  @6 1010
  @7 4302
  @8 3234
  @9 98ae
  @a 9230
  @b 0760
  @c 3200
  @d 0089
  @e ab00
  @f 0000
 @10 0000
 @11 0000
 @12 0000
 @13 0000
 @14 0000
 @15 0000
 @16 0000
```

(a) Initialized master memory block



```
// memory data file (do not edit the following l:
// instance=/top_tb/dut/SLAVE[0]/slave/ram
// format=hex addressradix=h dataradix=h version:
0000
4db6
43cd
bb81
67e9
674b
ce97
f0c2
d1e3
d040
5265
0000
00c1
0083
0007
000e
001c
0038
0070
0070
00e0
00c1
0083
```

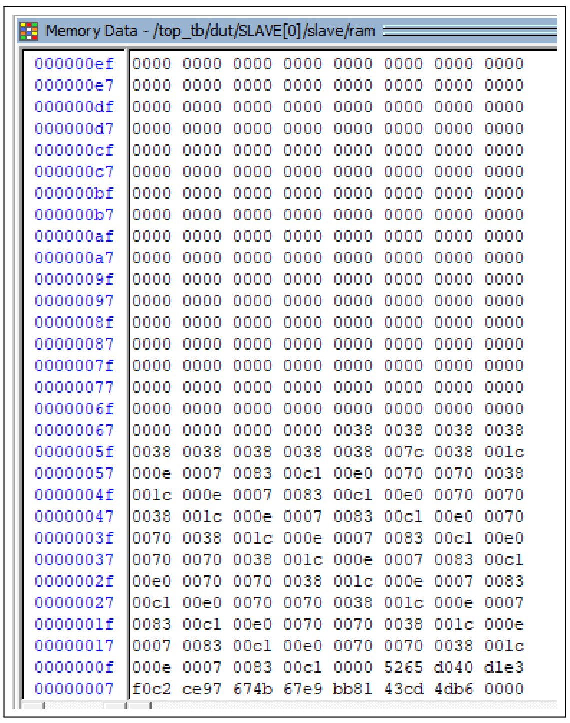
(b) Initialized slave memory block



Memory Data - /top_tb/dut/MASTER[1]/master/bram/ram - Default

```
000000ef 0000 0000 0000 0000 0000 0000 0000 0000
000000e7 0000 0000 0000 0000 0000 0000 0000 0000
000000df 0000 0000 0000 0000 0000 0000 0000 0000
000000d7 0000 0000 0000 0000 0000 0000 0000 0000
000000cf 0000 0000 0000 0000 0000 0000 0000 0000
000000c7 0000 0000 0000 0000 0000 0000 0000 0000
000000bf 0000 0000 0000 0000 0000 0000 0000 0000
000000b7 0000 0000 0000 0000 0000 0000 0000 0000
000000af 0000 0000 0000 0000 0000 0000 0000 0000
000000a7 0000 0000 0000 0000 0000 0000 0000 0000
0000009f 0000 0000 0000 0000 0000 0000 0000 0000
00000097 0000 0000 0000 0000 0000 0000 0000 0000
0000008f 0000 0000 0000 0000 0000 0000 0000 0000
00000087 0000 0000 0000 0000 0000 0000 0000 0000
0000007f 0000 0000 0000 0000 0000 0000 0000 0000
00000077 0000 0000 0000 0000 0000 0000 0000 0000
0000006f 0000 0000 0000 0000 0000 0000 0000 0000
00000067 0000 0000 0000 0000 0000 0000 0000 0000
0000005f 0000 0000 0000 0000 0000 0000 0000 0000
00000057 0000 0000 0000 0000 0000 0000 0000 0000
0000004f 0000 0000 0000 0000 0000 0000 0000 0000
00000047 0000 0000 0000 0000 0000 0000 0000 0000
0000003f 0000 0000 0000 0000 0000 0000 0000 0000
00000037 0000 0000 0000 0000 0000 0000 0000 0000
0000002f 0000 0000 0000 0000 0000 0000 0000 0000
00000027 0000 0000 0000 0000 0000 0000 0000 0000
0000001f 0000 0000 0000 0000 0000 0000 0000 0000
00000017 0000 0000 0000 0000 0000 0000 0000 0000
0000000f 0000 ab00 0089 3200 0760 5265 d040 d1e3
00000007 f0c2 ce97 674b 67e9 bb81 43cd 4db6 e707
```

(c) Master memory block after read burst



Memory Data - /top_tb/dut/SLAVE[0]/slave/ram

```
000000ef 0000 0000 0000 0000 0000 0000 0000 0000
000000e7 0000 0000 0000 0000 0000 0000 0000 0000
000000df 0000 0000 0000 0000 0000 0000 0000 0000
000000d7 0000 0000 0000 0000 0000 0000 0000 0000
000000cf 0000 0000 0000 0000 0000 0000 0000 0000
000000c7 0000 0000 0000 0000 0000 0000 0000 0000
000000bf 0000 0000 0000 0000 0000 0000 0000 0000
000000b7 0000 0000 0000 0000 0000 0000 0000 0000
000000af 0000 0000 0000 0000 0000 0000 0000 0000
000000a7 0000 0000 0000 0000 0000 0000 0000 0000
0000009f 0000 0000 0000 0000 0000 0000 0000 0000
00000097 0000 0000 0000 0000 0000 0000 0000 0000
0000008f 0000 0000 0000 0000 0000 0000 0000 0000
00000087 0000 0000 0000 0000 0000 0000 0000 0000
0000007f 0000 0000 0000 0000 0000 0000 0000 0000
00000077 0000 0000 0000 0000 0000 0000 0000 0000
0000006f 0000 0000 0000 0000 0000 0000 0000 0000
00000067 0000 0000 0000 0000 0038 0038 0038 0038
0000005f 0038 0038 0038 0038 0038 007c 0038 001c
00000057 000e 0007 0083 00c1 00e0 0070 0070 0038
0000004f 001c 000e 0007 0083 00c1 00e0 0070 0070
00000047 0038 001c 000e 0007 0083 00c1 00e0 0070
0000003f 0070 0038 001c 000e 0007 0083 00c1 00e0
00000037 0070 0070 0038 001c 000e 0007 0083 00c1
0000002f 00e0 0070 0070 0038 001c 000e 0007 0083
00000027 00c1 00e0 0070 0070 0038 001c 000e 0007
0000001f 0083 00c1 00e0 0070 0070 0038 001c 000e
00000017 0007 0083 00c1 00e0 0070 0070 0038 001c
0000000f 000e 0007 0083 00c1 0000 5265 d040 d1e3
00000007 f0c2 ce97 674b 67e9 bb81 43cd 4db6 0000
```

(d) Slave memory block after read burst

Figure 4.15: Memory files: Read burst data transfer

4.3.4 Write

The Write operation is used by the Master to write some data into the slave's BRAM. This data can be given from an external source (for example, the input from FPGA) or from the Master's BRAM. The dedicated "wrD" write Data wire is used to send the Write data in series, and the validity of this input is signalled using the "valid" wire.

```
initiate write                                : Master
    Send control Start|Slave_ID|10|address
if write data in buffer
    (1) Set valid HIGH
    (2) Start sending write data
if valid HIGH                                 : Slave
    (1) Start reading write data
    (2) write data into buffer
    (3) Store write data in BRAM
end write
```

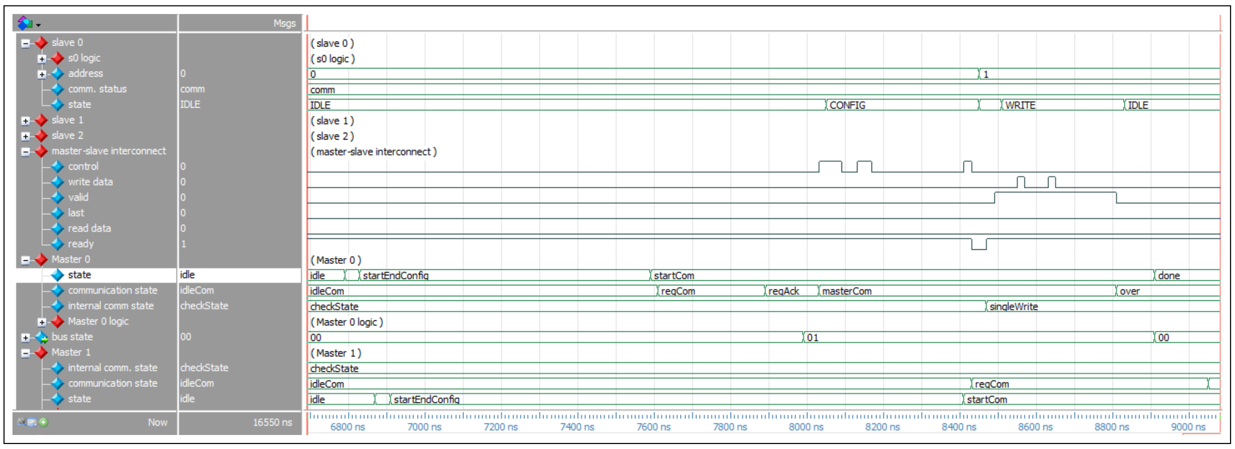


Figure 4.16: Simulation of single write between master and slave

4.3.5 Write Burst

The Write Burst operation is used to write a set of data from the master to a continuous block of the slave BRAM without having to specify each address. Similar to a single Write, the ”wrD” and ”valid” wires are used to send data, and the ”last” wire is used to signal the last address in the Write Burst operation.

```
initiate write burst                    : Master
    Send control Start|Slave_ID|11|address
if not last write burst address
    Set last LOW
    if write data in buffer
        (1) Set valid HIGH
        (2) Start sending write data
        (3) Increment address
        (4) Send next data
    if valid HIGH                       : Slave
        (1) Start reading write data
        (2) Write data into buffer
        (3) Store write data in BRAM
        (4) Receive next data
        (6) Increment address and Store
if last write burst address              : Master
    Set last HIGH
    Repeat write for one more data byte
end write burst
```

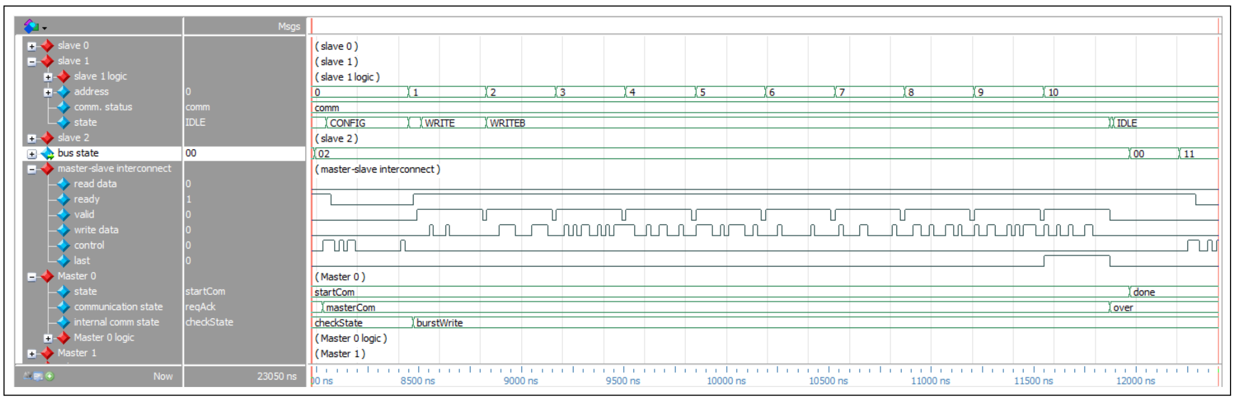
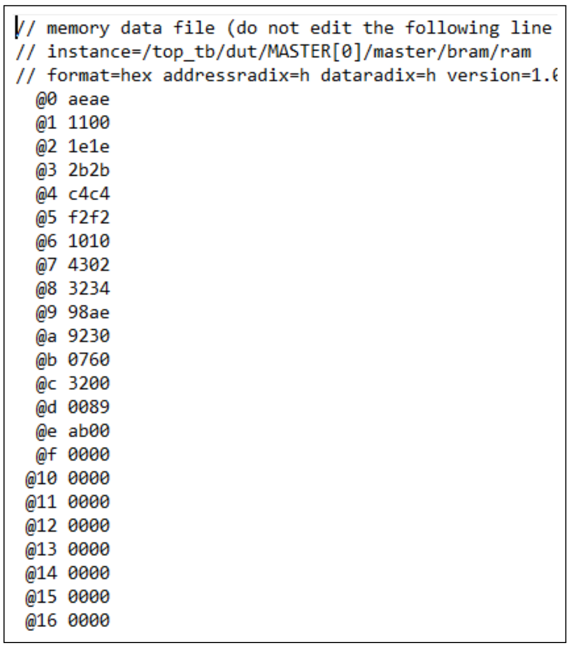
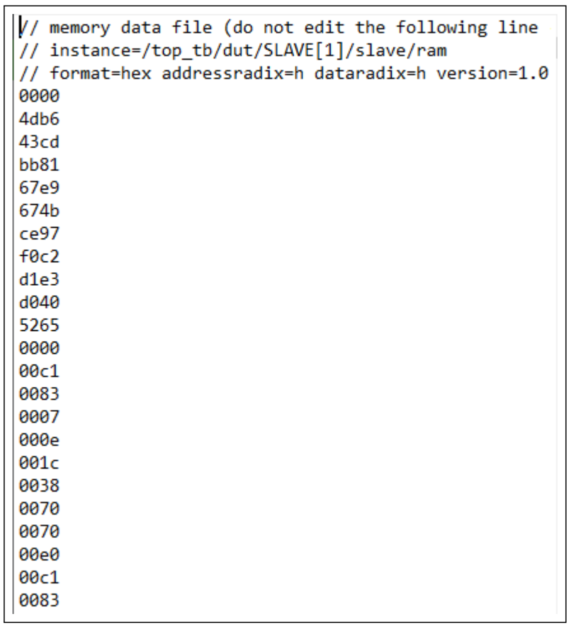


Figure 4.17: Simulation of write burst between master and slave



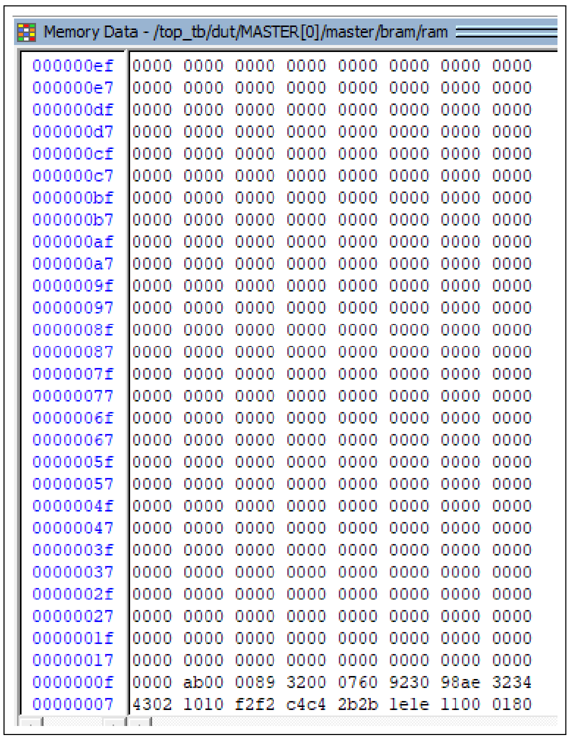
```
// memory data file (do not edit the following line
// instance=/top_tb/dut/MASTER[0]/master/bram/ram
// format=hex addressradix=h dataradix=h version=1.
  @0 aeae
  @1 1100
  @2 1e1e
  @3 2b2b
  @4 c4c4
  @5 f2f2
  @6 1010
  @7 4302
  @8 3234
  @9 98ae
  @a 9230
  @b 0760
  @c 3200
  @d 0089
  @e ab00
  @f 0000
 @10 0000
 @11 0000
 @12 0000
 @13 0000
 @14 0000
 @15 0000
 @16 0000
```

(a) Initialized master memory block



```
// memory data file (do not edit the following line
// instance=/top_tb/dut/SLAVE[1]/slave/ram
// format=hex addressradix=h dataradix=h version=1.0
0000
4db6
43cd
bb81
67e9
674b
ce97
f0c2
d1e3
d040
5265
0000
00c1
0083
0007
000e
001c
0038
0070
0070
00e0
00c1
0083
```

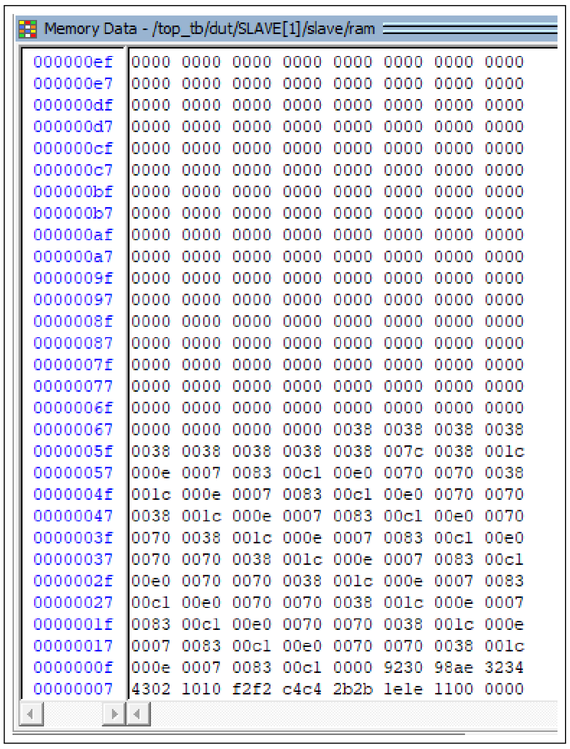
(b) Initialized slave memory block



Memory Data - /top_tb/dut/MASTER[0]/master/bram/ram

```
000000ef 0000 0000 0000 0000 0000 0000 0000 0000
000000e7 0000 0000 0000 0000 0000 0000 0000 0000
000000df 0000 0000 0000 0000 0000 0000 0000 0000
000000d7 0000 0000 0000 0000 0000 0000 0000 0000
000000cf 0000 0000 0000 0000 0000 0000 0000 0000
000000c7 0000 0000 0000 0000 0000 0000 0000 0000
000000bf 0000 0000 0000 0000 0000 0000 0000 0000
000000b7 0000 0000 0000 0000 0000 0000 0000 0000
000000af 0000 0000 0000 0000 0000 0000 0000 0000
000000a7 0000 0000 0000 0000 0000 0000 0000 0000
0000009f 0000 0000 0000 0000 0000 0000 0000 0000
00000097 0000 0000 0000 0000 0000 0000 0000 0000
0000008f 0000 0000 0000 0000 0000 0000 0000 0000
00000087 0000 0000 0000 0000 0000 0000 0000 0000
0000007f 0000 0000 0000 0000 0000 0000 0000 0000
00000077 0000 0000 0000 0000 0000 0000 0000 0000
0000006f 0000 0000 0000 0000 0000 0000 0000 0000
00000067 0000 0000 0000 0000 0000 0000 0000 0000
0000005f 0000 0000 0000 0000 0000 0000 0000 0000
00000057 0000 0000 0000 0000 0000 0000 0000 0000
0000004f 0000 0000 0000 0000 0000 0000 0000 0000
00000047 0000 0000 0000 0000 0000 0000 0000 0000
0000003f 0000 0000 0000 0000 0000 0000 0000 0000
00000037 0000 0000 0000 0000 0000 0000 0000 0000
0000002f 0000 0000 0000 0000 0000 0000 0000 0000
00000027 0000 0000 0000 0000 0000 0000 0000 0000
0000001f 0000 0000 0000 0000 0000 0000 0000 0000
00000017 0000 0000 0000 0000 0000 0000 0000 0000
0000000f 0000 ab00 0089 3200 0760 9230 98ae 3234
00000007 4302 1010 f2f2 c4c4 2b2b 1e1e 1100 0180
```

(c) Master memory block after write burst



Memory Data - /top_tb/dut/SLAVE[1]/slave/ram

```
000000ef 0000 0000 0000 0000 0000 0000 0000 0000
000000e7 0000 0000 0000 0000 0000 0000 0000 0000
000000df 0000 0000 0000 0000 0000 0000 0000 0000
000000d7 0000 0000 0000 0000 0000 0000 0000 0000
000000cf 0000 0000 0000 0000 0000 0000 0000 0000
000000c7 0000 0000 0000 0000 0000 0000 0000 0000
000000bf 0000 0000 0000 0000 0000 0000 0000 0000
000000b7 0000 0000 0000 0000 0000 0000 0000 0000
000000af 0000 0000 0000 0000 0000 0000 0000 0000
000000a7 0000 0000 0000 0000 0000 0000 0000 0000
0000009f 0000 0000 0000 0000 0000 0000 0000 0000
00000097 0000 0000 0000 0000 0000 0000 0000 0000
0000008f 0000 0000 0000 0000 0000 0000 0000 0000
00000087 0000 0000 0000 0000 0000 0000 0000 0000
0000007f 0000 0000 0000 0000 0000 0000 0000 0000
00000077 0000 0000 0000 0000 0000 0000 0000 0000
0000006f 0000 0000 0000 0000 0000 0000 0000 0000
00000067 0000 0000 0000 0000 0038 0038 0038 0038
0000005f 0038 0038 0038 0038 0038 007c 0038 001c
00000057 000e 0007 0083 00c1 00e0 0070 0070 0038
0000004f 001c 000e 0007 0083 00c1 00e0 0070 0070
00000047 0038 001c 000e 0007 0083 00c1 00e0 0070
0000003f 0070 0038 001c 000e 0007 0083 00c1 00e0
00000037 0070 0070 0038 001c 000e 0007 0083 00c1
0000002f 00e0 0070 0070 0038 001c 000e 0007 0083
00000027 00c1 00e0 0070 0070 0038 001c 000e 0007
0000001f 0083 00c1 00e0 0070 0070 0038 001c 000e
00000017 0007 0083 00c1 00e0 0070 0070 0038 001c
0000000f 000e 0007 0083 00c1 0000 9230 98ae 3234
00000007 4302 1010 f2f2 c4c4 2b2b 1e1e 1100 0000
```

(d) Slave memory block after write burst

Figure 4.18: Memory files: Write burst data transfer

4.3.6 Communication control

Just like initiation, the master has full control over the communication. In situations like the split transfer (sect. 4.2.4) and priority transfer (sect. 4.2.2), the master-slave communication will be temporarily put on hold, or terminated. This situation is handled with the use of control signal to send various bit patterns:

- START (111) : Initiates a completely new communication (sect. 4.3.1).
- ABORT (100): Stops current communication and both go to default mode - usually used during priority transfers.
- HOLD (110): Master and slave are disconnected. The slave can continue reading from memory or any other action and wait for master to restart communication.
- CONTINUE (101) : Re-starts a communication that was put on hold.

The latter two of these bit patterns are used during split transfer, which occurs when the slave is taking too long (more than the threshold value) to read data from memory and another master is waiting to access the bus interconnect to communicate with a different slave. Hence, the slave continues attempting to read data and has already got, or is closer to getting, the data into the buffer ready to send to master upon initiating communication.

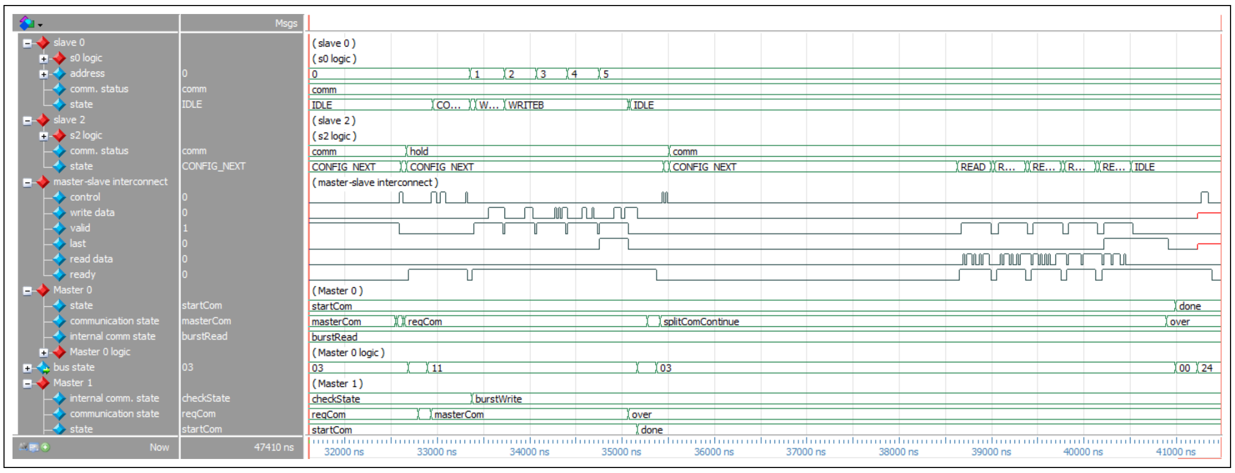


Figure 4.19: Simulation of data transfer with split transaction: master and slave

5 External Communication Protocol

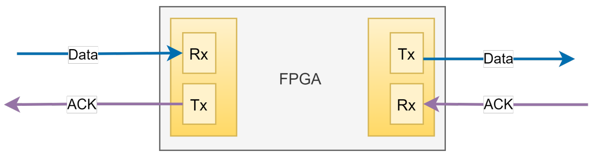


Figure 5.1: External communication from FPGA

Table 5.1: External communication ports

Name		Description
		Internal to (External)
g_rx	:	Serial communication wire to receive data-in
s_rx	:	Serial communication wire to receive acknowledgement for data-out
		External to Internal
g_tx	:	Serial communication wire to transmit acknowledgement for data-in
s_tx	:	Serial communication wire to transmit data-out

5.1 Initiating the communication

When the device boots up, the external master's default mode is 'read' mode. When it gets connected to the external slave, it sets the slave up to receive data from an external device.

In order to implement an external write, an 'external write' signal is sent to the external master, after which the master sends the write data to the external (UART) slave for transmission.

5.2 Minimizing the bus-interconnect allocation

As the external master is assigned the lowest priority among the masters, a request from another master will result in a priority selected transfer.

Hence, when the external master is in 'read' mode, if the arbiter conducts a transfer:

1. If a data transfer is ongoing, the master completes the data read and then sends a 'HOLD' signal to the slave.
2. If a data transfer is not ongoing, the master directly sends a 'HOLD' signal to the slave.
3. Then, it sends a 'Done' signal to the arbiter.

When the master is in 'write' mode, each time the single data write from master to slave is over, the master then sends a 'HOLD' signal to the slave and returns the bus interconnect to the arbiter. The slave then continues transmission, receives acknowledgement or handles re-transmission, and waits for the master to initiate the next read in order to update the master about the acknowledgement (ACK)/no-acknowledgement (NAK) status.

5.3 Data Read

```
Initiate Read                           : Master
    Send control Start|Slave_ID|0
if get_Rx sends start signal            : Slave
    (1) Send the acknowledgement through get_Tx
    (2) Set ready HIGH
    (3) Send four bits of ACK or NAK of the previous write
    (4) Start sending Read data to internal master
if ready HIGH                           : Master
    (1) Start reading Read data
    (2) Set valid HIGH at next clock cycle
    (3) Read data into buffer
    (4) Display data
    (5) Increment by 1
End Read
```

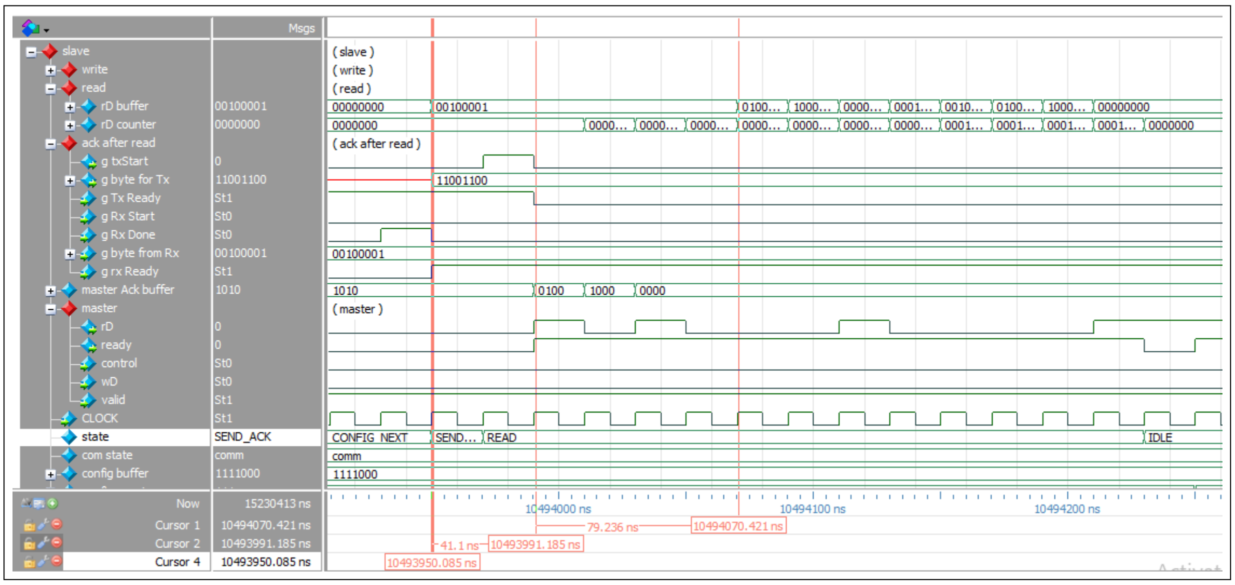


Figure 5.2: Read : Sending ACK followed by 8-bit data to master

5.4 Data Write

```
Initiate Write                          : Master
    Send control Start|Slave_ID|10|address
if Write data in buffer
```

```
    (1) Set valid HIGH
    (2) Start sending Write data
if valid HIGH                          : Slave
    (1) Start reading Write data
    (2) Write data into buffer
    (3) Send data to send_Tx
    if received ACK
        Send ACK to master with the next read data
    if not received ACK
        (1) Wait for 1 ms
        (2) Retransmit data through send_Tx
        if not received ACK after 5 retransmissions
            Send NAK to master with the next read data
End Write
```

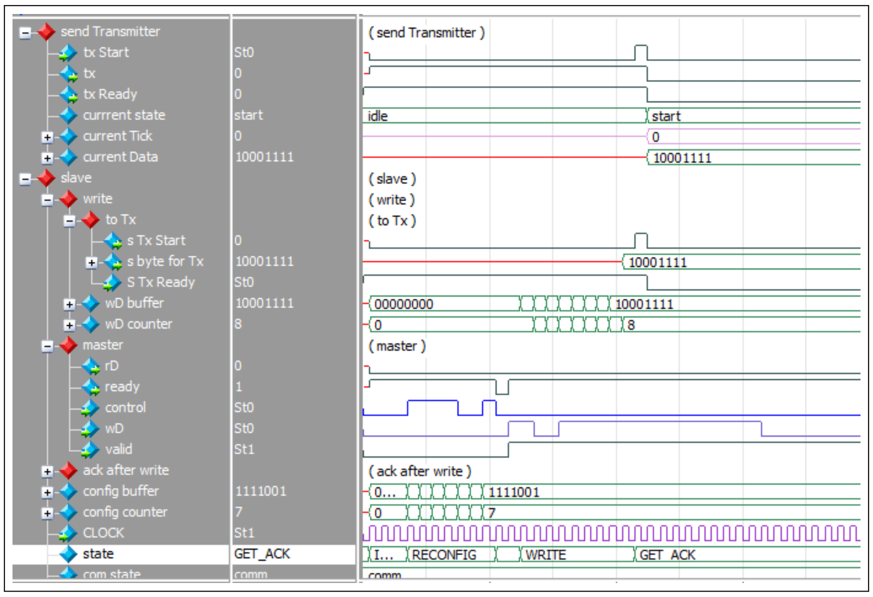


Figure 5.3: Write: Receive data from Master and transmit

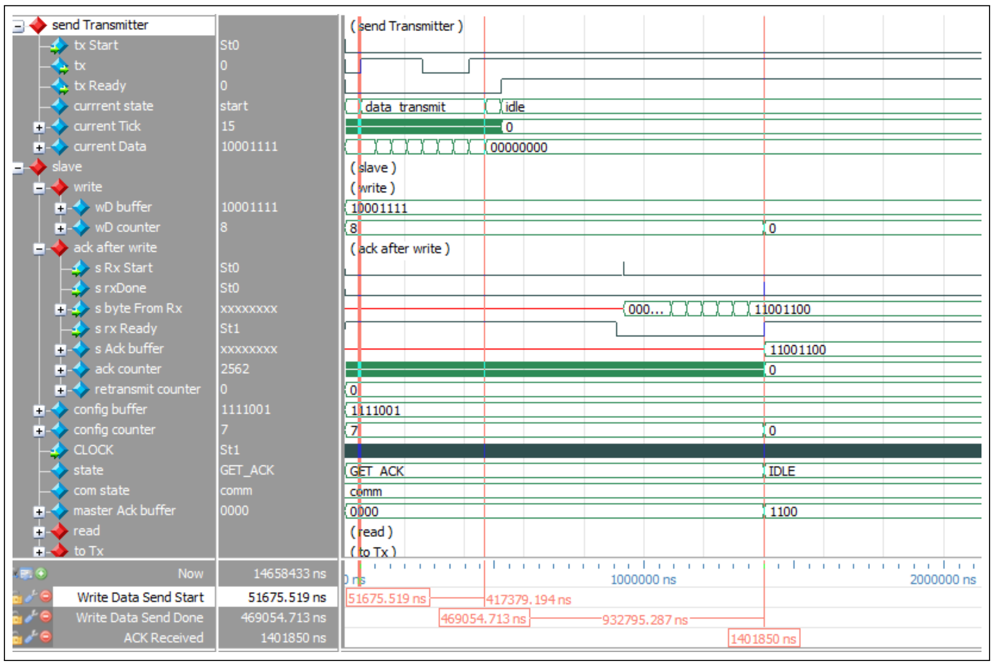


Figure 5.4: Write: Send data to UART and receieve ACK

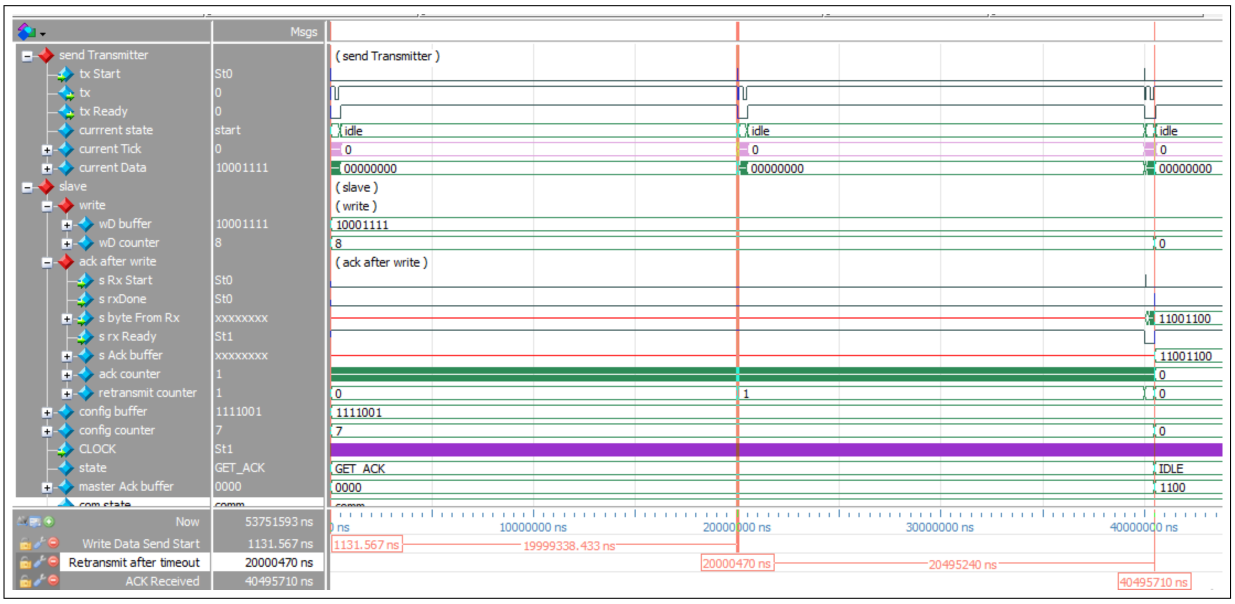


Figure 5.5: Write: Send data to UART - receive ACK after retransmitting

6 Discussion & Conclusion

6.1 Parameterization

The design has various variables that can be set before the compilation and after the compilation during run time. In our design most of the variables that should be decided before the compilation are parameterized. Therefore the whole system can be easily changed just by changing the parameter value given in the top module (*top.sv*) only. Some of the variables that are parameterized are given below.

- Internal communication related parameters
  - Number of masters
  - Number of slaves
  - Data width of the memories and serial transmission data packets.
  - The memory depth of each slave
  - Master who start communication first. (Useful to demonstrate high priority situations.)
  - Latency (in clock cycle count) of each slave before read from slave. (Useful to demonstrate split transaction.)
- External communication related parameters
  - UART data width
  - UART baud-rate
  - Number of re-transmissions until receive the acknowledgement before stop communication.
  - Acknowledgement timeout period. (in milliseconds)

Even though we have done the parameterization for many of the variables, due to non-parameterization of some of the variables, we could not achieve advantages of parameterization of some variables. This limitation is further discussed in the section 6.3.1.

6.2 Problems and Solutions

Various issues identified during the testing and debugging stages were experimented with, and appropriate solutions were implemented.

6.2.1 UART acknowledgement timeout not sufficient

The baud-rate for the external communication in this implementation is 19200. It takes around 0.5 milliseconds to send a one 8-bit data packet (including the starting bit and end bit). The acknowledgement timeout period was set to 1 millisecond in the initial design i.e., around twice the time taken for 1 data packet transmission. During the experiments we found that this timeout period was not enough as the external device that we used for testing - Arduino-Mega - takes more than 1 millisecond to responds to the received data packet. As a solution, we could increase the timeout period for the acknowledgement or increase the baud-rate to 115200 or higher value. But, in order to keep the communication protocol specifications close to the original design parameters, we increased the acknowledgement timeout period to 10 milliseconds.

6.2.2 Sending control signal during a data transfer

The system is designed such that the bus interconnect can be handed over mid-data-transfer to another master, thereby necessitating the master to terminate or put on-hold the current data transfer. Hence, the master-slave communication is designed so that the master can initiate and terminate a data transfer at any time, irrespective of whether the slave is currently handling a previous data transfer or not. However, while the master and slave are not connected, the slave may still have to continue its own processing.

For instance, when the slave with a high memory-access *delay* is interrupted with a split transfer (sect. 4.2.4), the slave has continue processing, and wait for the master to be connected to hand over the data.

Another instance is in the external master-slave communication, where the slave is either waiting for an acknowledgement, carrying out retransmission, or waiting to receive a data.

6.3 Limitations of the System

The implemented protocol still has several limitations, which could be improved for a more advanced and flexible design.

6.3.1 Limitations in parameterization

Our design is parameterized such that most of the variables can be easily changed just by changing a number in the top module (section 3.1). But due to the method we use to acquire user inputs in the top module, **the parameter "INT_MASTER_COUNT" (number of masters used for internal communication) can not be changed**. The default value for this parameter is two. We acquire some of the user data to configure all the internal masters at once. (ex:- select read or write.) User inputs for these data are visible in the LCD. Due to limitations of the space in the LCD, we can not increase the above mentioned parameter.

But as a solution, get user inputs for each master module can be acquired separately. Then the above mentioned parameter can be changed. (This method is currently being designed.) But user may have to provide necessary inputs for a long time and it may be not good implementation strategy for demonstration purposes.

6.3.2 Limitations of the split transaction

The arbiter in our design will carry out the split or priority communication (described comprehensively in section 4.2.2) as follows:

1. Pause the first master

2. Communication of the second master
3. Handing over the bus to the first master

Consider the following scenario: an interrupt occurs while the second master is communicating. Then, the second master should give away the bus to the third. If an interrupt occurs again, while third master is communicating, then fourth master will get the bus... and this continues.Then, after each master ends its communication, the arbiter should handover the bus to the previous master whose communication was interrupted up until the awaiting first master. However, interrupts can occur even at this stage. This will create a situation like a recursion within a recursion, which will lead to memory leakage and highly complex hardware design. Therefore, in order to avoid this, we designed the arbiter such that it does not to listen to any interrupts while the bus is given to the second master. So the current design will not lead to split leakage issue. However, any master who requested while the design was in split stage will get the bus after split transaction ends. Therefore, **any master who requests the bus will eventually get the bus at some point.**

6.3.3 Limitations in external data transfer

The system is design with two different *data_out* and *data_in* UART modules. While each of these modules have an additional Rx and Tx connection used for receiving and sending acknowledgements respectively, this restricts the direction of data transfer. Hence, one pair of pins can only be used for transmitting data and receiving acknowledgement, while the other can be used for the opposite.

6.3.4 Limitation in sending acknowledgement to master

This implementation of bus protocols uses master to slave connection only in one direction (i.e., from the master module to the slave module). Thus, in the implementation of external communication, any communication initiation has to happen from the master to the slave. The slave thus cannot send the acknowledgement once it is received. Hence, the acknowledgment for the *previous write* is only sent with the *next read* from the slave module to the master module.

6.4 Conclusion

The serial bus protocol with both internal and external communication, is capable of handling priority based transfer and delay based split transfer described in this report and it is implemented and demonstrated on an FPGA.

The internal communication can be demonstrated by sending instructions to the masters using the switches and push buttons on the board, with each master being assigned a maximum of one data transfer at a time. The accuracy of a read-type or write-type data transfer can be confirmed by following it up with an appropriate data transfer (i.e., write after a read, read after a write) to or from the same slave address. Similarly, a range of addresses can be input for burst read or write-type data transfers.

The external communication can be demonstrated by connecting to other FPGA boards as discussed in section 5, then initiating a data transfer of an integer from one of the boards, after which the boards connected in a loop will continuously increment and forward this value.

During the design and coding of this project, in addition to functionality and accuracy of data transfer, the implementation of a parameterized, and thereby flexible bus was paid attention to. In the testing and debugging stage of the project simulation, various scenarios were tested and corrected if there were errors in the initial design logic. Similarly, errors were identified and corrected during the implementation testing stage. Finally, the serial bus protocol was successfully implemented to carry out basic data transfers in both internal and external communication.