CSE

Department of Computer Science & Engineering

802.3 Network Repeater

100BASE-T4

Design Project Specification

Dr. James P. Davis

VLSI Design Laboratory
Department of Computer Science and Engineering
University of South Carolina
Columbia, SC 29208 USA

Version: 23 February 2004

Update from: Nov 21[s], 2000

UNIVERSITY OF SOUTH CAROLINA

1. Ethernet Background

Ethernet is based on the idea that a network between a group of computers should behave as a shared medium. Only one host can transmit data at a time, but more than one host may attempt to transmit data at the same time. If a host wishes to transmit data, it must wait until it detects that the medium is not in use. If two hosts transmit at the same time, a collision occurs and both hosts must stop transmission and try to resend their data after an arbitrary waiting period. This technique is a primary feature of Ethernet and is called carrier sense, multiple access with collision detect (CSMA/CD).

The function of the network repeater is to create the logical equivalent of a shared medium. Each host on the network connects to a common repeater through a segment of twisted pair coaxial cable. The repeater logically joins cable segments to create a larger network. They improve reliability and performance because they isolate hosts with faulty connections and prevent them from disrupting the rest of the network.

Ethernet conforms to the IEEE 802.3 standard concerning network media, physical interfaces, signaling, and network access protocol.

2. The Network Repeater

The basic objectives for a repeater are to:

- Detect carrier activity on ports and receive Ethernet frames on active ports
- Restore the shape, amplitude, and timing of the received frame signals prior to retransmission
- Forward the Ethernet frame to each of the active ports
- Detect and signal a collision event throughout the network
- Extend a network's physical dimensions
- Protect a network from failures of stations, cables, ports, and so forth
- Allow the installation and removal of stations without network disruption
- Support interoperability of various physical layers (10BASE2, 10BASE-T, etc.)
- Provide centralized management of network operations and statistics
- Provide for low-cost network installation, growth, and maintenance
- Partition bad segments

A network repeater's basic function is to retransmit data that is sent from one port to all other ports. Transceivers perform the electrical functions needed for interfacing the host ports to the repeater core logic. Each transceiver interface provides the following signals to the repeater:

- carrier sense (**crs**)
- receive clock (**rx_clk**)
- receive data valid (**rx_dv**)
- receive data error (**rx,_er**)

- three pairs of data (**rxd0 - rxd5**)

The signal **crs** indicates that data is being received by the transceiver. **rx_clk** is the clock recovered from the incoming data by the transceiver and is used to synchronize the data, **rxd0 - rxd5**. The signal **rx_dv** informs the repeater core that the received data, is valid. It is asserted at the start of a data frame (at the data frame's start of frame delimiter, SFD) and deasserted at the end of a frame. The **rx_er** signal indicates that an error was detected in the reception of the data.

3. Protocol

When the carrier sense (**crs**) signal is received from any port, the repeater must buffer the incoming data frame and retransmit the frame to all other functional ports if the following conditions hold true:

- there is not a collision
- the port is not jabbering
- the port is not partitioned

A collision occurs when more than one carrier sense becomes active. In the case of a collision, a **jam** symbol must be generated and transmitted to all ports, including the one previously sending data. The ports that caused the collision will then wait for an arbitrary length of time before attempting to resend data across the network.

A port is jabbering if it continually transmits data for 40,000 to 75,000 bit times. If a port is jabbering, the repeater will stop receiving data from this port by inhibiting **rx_en**. This will free up the network for other ports to send data. The port will be considered to have ceased jabbering after carrier sense is deasserted.

A port is partitioned from the network if it causes 60 or more consecutive collisions. This action is necessary because continued collisions will bring all network communication to a halt. A large number of consecutive collisions can often be attributed to a broken cable or faulty connection. A partitioned port is reconnected if activity on another port occurs for 450 to 560 bit times without the assertion of carrier sense from the partitioned port.

If carrier sense is not asserted by any of the ports, then **idle** symbols are generated and transmitted on all ports. If **rx_er** is asserted by a receiving port while the repeater is retransmitting data to the other ports, then **bad** symbols are generated and transmitted to all other ports until **crs** is deasserted or there is a collision.

To summarize, the repeater receives data from one port and transmits to all other ports. The repeater detects collisions, activity, and errors, and generates the appropriate symbols under these conditions. It also detects jabbering and partition conditions, and asserts **tx_en** (transmit enable) and **rx_en** (receive enable) when appropriate.

4. Data Frame Structure

The frame format is given as follows

Preamble
Start of frame delimiter (SFD)
Destination address
Source address
Length
Data
Pad
Frame check sequence

Ethernet AMC data frame

- **Preamble** - used by the receiving hosts to detect the presence of a carrier and initiate clock recovery (7 bytes, equal to " 1010..." for PLL synchronization)

- **Start of frame delimiter (SFD)** - indicates to the receiving hosts that the next group of bits is the actual data to be transmitted (1 byte)

- **Destination address** - 48-bit address that uniquely identifies which host on the network should receive the frame. A host address is created by taking the 24-bit organizationally unique identifier (OUI) assigned to each organization. The remaining 24 bits are determined internally by network administrators.

- **Source address** - a 48-bit address that uniquely identifies which host is sending the frame.

- **Length** - a 2-byte field that determines how many bytes of data are in the data field.

- **Data** - the minimum data field size is 46 bytes. If fewer than 46 bytes of data need to be sent, additional (pad) characters are added to the end of the data field. The maximum data field size is 1,500 bytes.

- **Frame check sequence (FCS)** - the FCS is a 32-bit cyclic redundancy check (CRC) computed from a standard CRC polynomial. The receiving host computes a CRC from the bits it receives and compares the value to the FCS imbedded in the frame to see whether the data was received error free.

In order to transmit 100 Mb/s over 4 pairs, the frequency of operation must be 25 MHz. The repeater does not check the frame size of frames sent by nodes. It only retransmits the data.

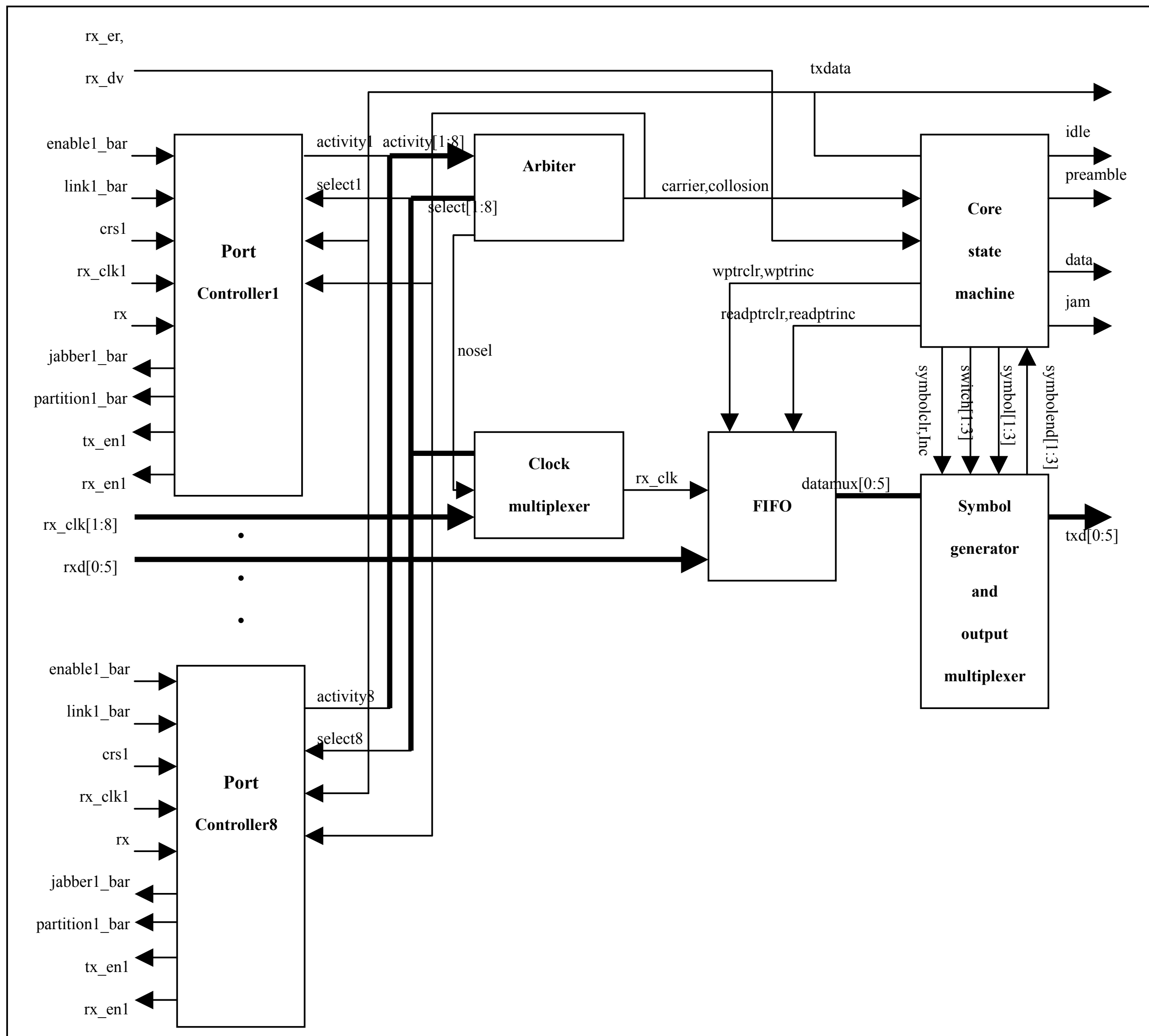
5. Top-level Block Diagram

To summarize the functions of the repeater: 1) In general, the repeater receives data from

one port and retransmits it to the other ports; 2) it detects collisions, activity, and errors, generating and transmitting the appropriate symbols under these conditions; and 3) it detects jabbering and partition conditions, asserting **tx_en**, **rx_en** as appropriate. To accomplish the functions required of the network repeater, the incoming data must be buffered and the correct symbols generated. The buffered data must be multiplexed with other symbols, depending on which data should be transmitted to the active ports. Given the complexity of the problem at hand, the design is broken down into manageable units as follows:

1) Port controller
2) Arbiter
3) Clock Multiplexer
4) Symbol Generator and Output Multiplexer
5) Core Controller

Block diagrams illustrating the top-level ports and the internal connections between components are given as following:



Port Name	Mode	Width	Description
txclk	in	1	Clock for synchronization
areset	in	1	Asynchronous reset
crs	in	1	Carrier sense
enable_bar	in	1	Port enable
link_bar	in	1	PMA_link_OK
selected	in	1	Arbiter select
carrier	in	1	Arbiter carrier
collision	in	1	Arbiter collision
jam	in	1	Control jam
txdata	in	1	Control transmit data
prescale	in	1	Counter prescale
rx_en	buffer	1	Receive enable
tx_en	buffer	1	Transmit enable
activity	buffer	1	Activity present
jabber_bar	buffer	1	Indicates port is jabbering
partition_bar	buffer	1	Indicates port is partitioned

Port descriptions for Port Controller component

6. Port Controller

There are a total of eight port controllers in this design, one for each port of the repeater. Each port synchronizes **crs**, **link_bar**, and **enable_bar** to **tx_clk** on the receive side. The **activity** signal is asserted once a carrier has been detected and synchronized. The arbiter uses this signal to select a port from which to receive data. If a port controller's port has been selected and there is no collision, **rx_en** is asserted. If the port is not the receiving port, **tx_en** is asserted if the core controller indicates that **tx_data** is ready (**link_bar** must also be set and the port cannot be jabbering).

In the event of a collision, **tx_en** is asserted so that the appropriate jam characters may be transmitted to all hosts. The signals **jabber_bar** and **partition_bar** indicate that a port is either

jabbering or partitioned. A timer is used in the port controller to determine how long **crs** has been asserted. If it has been asserted 40,000 to 75,000 bit times, then **jabber_bar** is asserted until **crs** goes low. The number of consecutive collisions is counted to determine whether a port should be partitioned. If 60 or more consecutive collisions have occurred, **partition_bar** is asserted, thereby preventing the port from halting network communication. If another port has been active for more than 450 to 560 bit times without the partitioned port's **crs** signal being asserted, then the port controller deasserts **partition_bar**. These conditions will be determined by the value of **collision** and **carrier**, as well as whether or not the port has been selected.

The **enable_bar** signal is used when a port is not needed. It may be hardwired as deasserted for ports that will not be used.

It is required that **crs**, **link_bar**, and **enable_bar** must be synchronized to **tx_clk**. An asynchronous preset should be used for active low signals(those ending in "**_bar**") and an asynchronous reset otherwise for the active high signal, **crs**. Each of these signals should be registered twice to increase the MTBF (mean time between failure). It is standard practice to use D-type flip-flops for registering signals. Using the synchronizers requires to create three signals (**crsdd, link_bardd, enable_bardd**) for the outputs of the synchronizers (where the suffix "**dd**" indicates that the signal has been twice registered).

The **activity** signal is asserted when a carrier is present, the link is established, and the port is not partitioned or jabbering. An internal signal, **carpres**, should be created for representing the fact that a carrier is present. This signal should be equivalent to the registered **crsdd** signal gated by the registered **enable_bar** signal. Both **activity** and **carpres** can be defined using Boolean equations.

carpres <= **crsdd** and not **enable_bardd**
activity <= carpres and not **link_bardd** and **jabber_bar** and **partition_bar**

Rx_en and **tx_eni** can also be defined with boolean equations. **tx_eni** is an internal signal representing the value of **tx_en** before it is registered, it is asserted when **txdata** is received from the core controller, indicating that this port should transmit data, if it is not jabbering. It must be delayed one clock cycle to synchronize it with the output data.. **tx_eni** must be declared as a signal local to this architecture.

rx_en <= not **enable_bardd** and not **link_bardd** and **selected** and not **collision**;
tx_eni <= not **enable_bardd** and not **link_bardd** and **jabber_bar** and **transmit**;

The **transmit** is the logical AND of two quantities. The first signal is **txdata**, which indicates that the core controller state machine is ready to send data or jam characters. The second is the result of the logical OR of **collision** and a signal **copyd**, which indicates that the arbiter has detected a carrier but that this port is not the selected port. In the other words, the port should transmit jam characters (if a collision occurs) or data characters (to copy, or repeat, data received from another port). As a result, the signal **collision** must be registered (to add a one-cycle delay), so flip-flops are instantiated. All signals must be synchronized.

txdata is one clock cycle behind the other signals because it is triggered by them. Therefore,

the other signals must be delayed:

copyin <= **carrier** and not **selected**;
transmit <= **txdata** and (**copyd** or **collisiond**);

Another condition that the port controller must handle is that of a jabbering port. **jabber_bar** should be asserted (low) if the carrier is present for anywhere between 40,000 and 75,000 bit times. This requires 10,000 clock cycles (75,000 bit times requires 18,750 clock cycles) for 40,000 bit times over four pairs. A counter should be used and incremented while the carrier is present. If it counts approximately 10,000 clock cycles, then **jabber_bar** will be asserted. The counter will be reset when the carrier is not present.

A 14-bit counter is necessary to count to 10,000. To reduce the unnecessary consumption of resources that would result from instantiating a 14-bit counter for each port controller, a common counter can be placed in the core controller. The common counter will count to I K, and run continuously. In each port controller, a 4-bit **prescaled counter** is used that is enabled each time the 10-bit common counter rolls over, or reaches its maximum value. When the 4-bit counter reaches 12, the carrier has been present for 12K clock cycles, plus or minus 1K, clock cycles (10k < 12k +/- 1k < 18.75k). For the port controller design, it is required that the 4-bit counter be instantiated and that signals for counter enable and clear are defined as well as **jabber_bar.**

jabber_bar is asserted when the output of the counter(**jabcnt**) reaches 12, or " 1100" binary. When **jabcnt** reaches 12, which is 1100 in binary, or **jabcnt**(3) and **jabcnt**(2), **jabber_bar** is asserted (low), causing the counter to be disabled. The counter should include a clear signal that is asserted when there is not a carrier present and an enable signal which is asserted when **carpres**, **prescale**, and **jabber_bar** are set.

A state diagram taken from the IEEE 802.3 standard describes the conditions for partitioning or reconnecting a port. It is given on the following page. The states on the left-hand side of the diagram are used to determine whether the port should be partitioned. The **partition_bar** signal is asserted low when port is partitioned. The states on the right-hand side are used to determine whether the port should be reconnected. This state machine controls the clearing and enabling of two counters. One counts collisions (the collision counter) while the other counts the number of clock cycles without a collision. The inputs to the state machine are as follows:

- **copyd**(one clock delay of **copyin**) indicates that another port (not this port) is active and that the port should repeat data being received on another port.
- **quietd** is the opposite of **copyd**. It indicates that no other port is active.
- **carpres** indicates that the carrier sense of this port is active.
- **collisiond** indicates a collision (ports simultaneously trying to transmit)
- **nocoldone** is asserted when the no-collision counter reaches 128. This is the approximately the number of clock cycles needed to transmit a data frame of minimum size. Each clock cycle represents four bit times, and 450 to 560 bit times are needed to transmit a data frame of minimum size.
- **cclimit** is asserted when 64 consecutive collisions have occurred

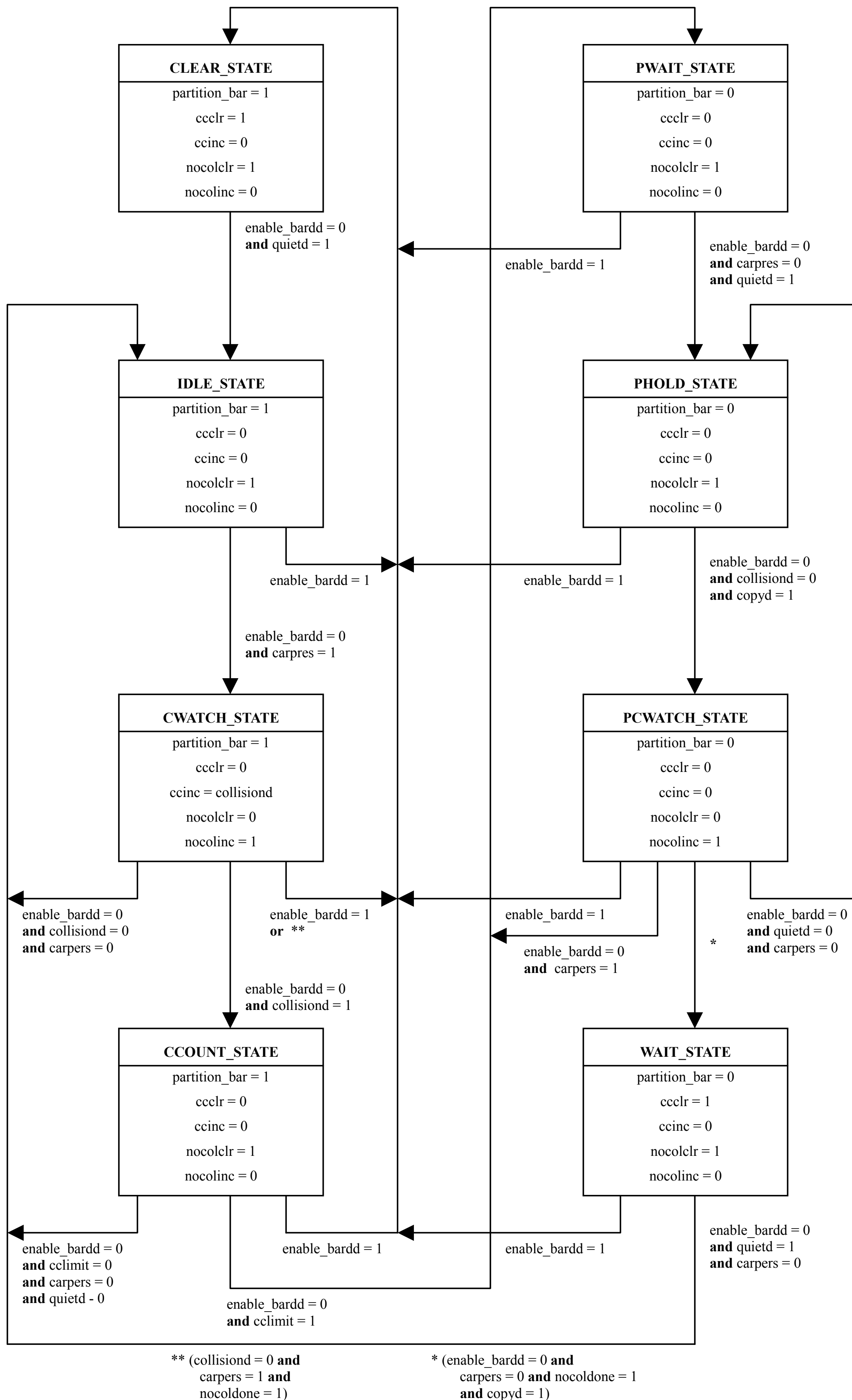
The machine enters the **CLEAR_STATE** after a system reset or if a collision does not occur

for 128 clock cycles while **carpres** is asserted. The counters are cleared in this state. The **IDLE_STATE** resets the no-collision counter. In the collision-watch state, the **CWATCH_STATE** ,if a collision occurs, the collision counter is incremented and a transition is made to the **CCOUNT_STATE**, the collision count state. If no collision occurs, the state machine remains in the **CWATCH_STATE** until **nocoldone** is asserted, at which point a transition is made to the **CLEAR_STATE** and both counters are reset.

The **CCOUNT_STATE** is entered upon a collision. While in the **CCOUNT_STATE**, if the limit for the collision counter is reached (64), then the port is partitioned and a transition is made to the **PWAIT_STATE.** If the limit is not reached, then a transition is made to the **IDLE_STATE** when the offending nodes have backed off and the network is quiet again. From this point, transitions through the states will continue, either additional collisions will be counted or the counters will be cleared.

The **PWAIT_STATE** indicates that the collision counter limit has been reached to justify partitioning the port. The port is partitioned in this state until the network is quiet again, at which point a transition is made to the **PHOLD_STATE**. This state and the **PCWATCH_STATE** (partition collision-watch state) count consecutive clock cycles during which there is no collision and another port is active (**copyd** is asserted). If the port does not cause a collision for 128 clock cycles (450 to 560 bit times) while another port is active, then a transition is made to the **WAIT_STATE**. The state machine remains in the **WAIT_STATE** until the active port is quiet, then a transition is made to the **IDLE_STATE**.

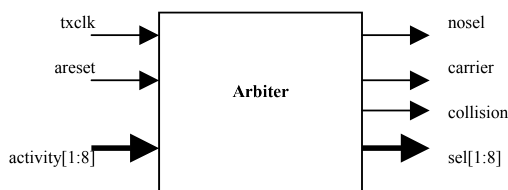
The state transition diagram is given on the next page.


CLEAR_STATE
partition_bar = 1
ccclr = 1
ccinc = 0
nocolclr = 1
nocolinc = 0
PWAIT_STATE
partition_bar = 0
ccclr = 0
ccinc = 0
nocolclr = 1
nocolinc = 0
enable_bardd = 0
and quietd = 1
enable_bardd = 1
enable_bardd = 0
and carpres = 0
and quietd = 1
IDLE_STATE
partition_bar = 1
ccclr = 0
ccinc = 0
nocolclr = 1
nocolinc = 0
PHOLD_STATE
partition_bar = 0
ccclr = 0
ccinc = 0
nocolclr = 1
nocolinc = 0
enable_bardd = 1
enable_bardd = 1
enable_bardd = 0
and collisiond = 0
and copyd = 1
enable_bardd = 0
and carpres = 1
CWATCH_STATE
partition_bar = 1
ccclr = 0
ccinc = collisiond
nocolclr = 0
nocolinc = 1
PCWATCH_STATE
partition_bar = 0
ccclr = 0
ccinc = 0
nocolclr = 0
nocolinc = 1
enable_bardd = 0
and collisiond = 0
and carpers = 0
enable_bardd = 1
or **
enable_bardd = 1
enable_bardd = 0
and carpers = 1
*
enable_bardd = 0
and quietd = 0
and carpers = 0
enable_bardd = 0
and collisiond = 1
CCOUNT_STATE
partition_bar = 1
ccclr = 0
ccinc = 0
nocolclr = 1
nocolinc = 0
WAIT_STATE
partition_bar = 0
ccclr = 1
ccinc = 0
nocolclr = 1
nocolinc = 0
enable_bardd = 0
and cclimit = 0
and carpers = 0
and quietd - 0
enable_bardd = 1
enable_bardd = 1
enable_bardd = 0
and quietd = 1
and carpers = 0
enable_bardd = 0
and cclimit = 1
** (collisiond = 0 and
carpers = 1 and
nocoldone = 1)
* (enable_bardd = 0 and
carpers = 0 and nocoldone = 1
and copyd = 1)

7. Arbiter

The arbiter will use the activity signals of each of the port controllers to supply eight selected signals to the port controllers and clock multiplexer. These signals will indicate which port is receiving data. They will be used to gate the **rx_en** of that port and to choose the appropriate clock for writing to the FIFO. The arbiter also supplies **carrier** and **collision** for use by the port controllers and the core controller. **Nosel** is supplied to the clock multiplexer, indicating that no port is receiving a transmission. Under this condition, all ports of the repeater transmit idle characters.

The block diagram for the arbiter is as given below:



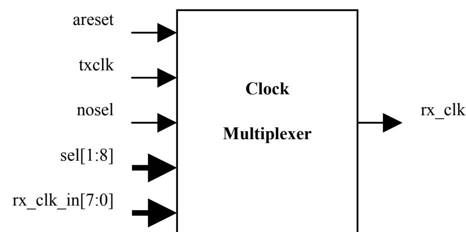
The I/O port description is given below:

Port Name	Port type	Width	Description
txclk	input	1	Transmit clock
areset	input	1	Asynchronous reset
activity	input	8	Port activity
nosel	buffer	1	No port selected
carrier	buffer	1	Carrier detected
collision	buffer	1	Collison detected
sel	buffer	8	Port select

The ports **sel1**- **sel8** are set according to the port activity indicated by the ports **activity1**- **activity8**. The arbiter ensures that only one port is selected as the active receiving port, and identifies collisions as well as an absence of activity. The arbiter does not use a fairness scheme in which each port has equal priority for selection. Instead, **activity1** has the highest priority and **activity8** has the lowest. This means that lower order ports will be selected over higher order ports if activity occurs on multiple ports at about the same time. The **nosel** is set if there is no activity on any port. The **carrier** is set if activity is present on atleast one port. The **collision** is set if activity is present on more than one port. A bank of registers are used to create a pipeline stage. The **carrier**, **collision**, and **nosel** will be used in multiple modules and will propagate through several levels of logic, generating the need for pipeline stage to reduce loading and to maintain a high frequency of operation. The other signals are pipelined to maintain synchronization.

8. Clock Multiplexer

The inputs to the clock multiplexer are the eight receive clocks (**rx_clk7**- **rx_clk0**), the eight selected lines from the arbiter, **nosel** from the arbiter, **areset**, and **txclk**. **Selected** and **nosel** signals are used to select one of the receive clocks as the receive clock, **rx_clk**, or the tranmit clock **tx_clk** (if **nosel**) for use by the FIFO. The block diagram is as given below. The design of the clock multiplexer should be glitch-free. One must be careful as using the output of a combinational function as clock input may cause false clocks due to inputs transitioning at slightly different times or propagating with slightly different delays, even if the simulation indicates that the delays are balanced. In most cases use the output of one flip-flop as the product-term clock for another flip-flop.



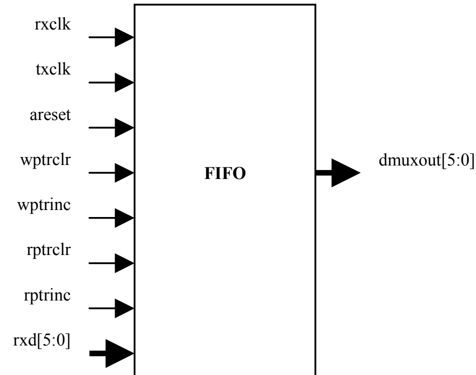
The I/O port description is as given below:

Port Name	Port type	Width	Description
areset	input	1	Asynchronous reset
tx_clk	input	1	Transmit clock
nosel	buffer	1	No port selected
sel	buffer	8	Port select
rx_clk_in	input	8	Receive clock input
rx_clk	output	1	Output receive clock

9. FIFO

The FIFO will capture the incoming data on the receive side, storing six bits of data (**rxd5** – **rxd0**) on the rising edge of **rx_clk. Wptrclr** (write-pointer clear), **wptrinc** (write-pointer increment), **rptrclr** (read-pointer clear), and **rptrinc** (read-pointer increment) are used to advance or clear the

FIFO and to indicate which register to read for the outputs **dmuxout5** – **dmuxout0**. The depth of the FIFO has been set as eight to account for the latency between write and read cycles.
The block diagram and I/O port description of the FIFO are given below:

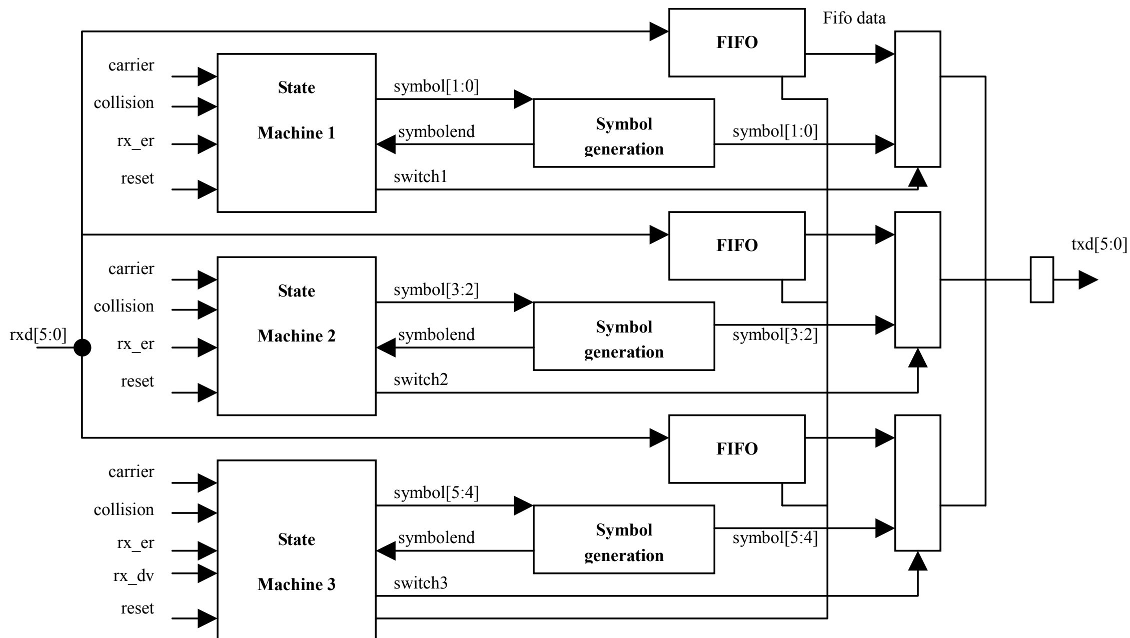


Port Name	Port Type	Width	Description
rxclk	input	1	Receive clock
txclk	input	1	Transmit clock
areset	input	1	Asynchronous reset
wptrclr	input	1	Write-pointer clear
wptrinc	input	1	Write-pointer increment
rptrclr	input	1	Read-pointer clear
rptrinc	input	1	Read-pointer increment
rxd	input	6	FIFO data input
dmuxout	output	6	FIFO data output

10. Core controller

The core controller controls the FIFO read and write pointers as well as symbol generation. It also asserts **tx_data** to indicate to the ports that data is ready.

The function of the core controller is illustrated in figure below.



The core controller determines what data to transmit, data in the FIFO or preamble, idle, jam, or bad characters. To do this, the core controller requires **carrier**, **collision**, **rx_dv**, and **rx_er** as inputs and asserts the FIFO and multiplexer control lines. The signals **rx_dv** and **rx_er** are synchronized to the **tx_clk** with two flip-flops in order to increase the mean time between failures (MTBF).

The core controller basically has three state machines one for each transmit pair of the symbol generator and output multiplexer. Each of the state machines follows the state transition diagram given in fig 1.0. The preamble state actually consists of three preamble substates for the first and third transmit pairs; the second transmit pair has four preamble states (three to transmit SOSA (preamble-“00”), and one to transmit SOSB (SFD-“01”)). The state transition diagram provided illustrates the state transition for transmit pair three. The transition diagram is similar for transmit pair one but for transmit pair two, one more preamble state, **pre4_state** is to be added from transitions are made to **data_state**, **error_state**, **no_sfd**, or **jam_state**. Transitions from one preamble state to the next are based on the **symbolend** for each of the pairs. The **symbolend** is a boundary that is generated from the 3-bit counter placed in the symbol generator to which the clear and increment signals are provided by the core controller. The **carrier** signal should be synchronized with the **tx_clk** and for this it should be registered twice. The state machines use

rx_er and the **rx_dv** as inputs. These signals tranisition from the high-impedance state and are gated with **carrierdd** to filter out glitches. Internal signals, say **rx_dv_in** and **rx_error_in** might be declared and following equations can be used for filtering the glitches.

rx_dv_in <= **carrierdd** and **rx_dv**

rx_error_in <= **carrierdd** and **rx_error;** where **carrierdd** is **carrier** synchronized with **tx_clk**.

These signals can be registered for proper synchronization to obtain registered signals **data_valid** and **error,** which can then be used for state machine control**.** The **rx_dv_in** is to be registered once and **rx_error_in** is to be registered twice.

The FIFO read and write pointers and resets are all controlled by the core controller. The FIFO read and write pointers are continuously incremented. Carrier is an output of the arbiter and indicates that a port is active. However, the write pointer is cleared until valid data is identified and read pointer is cleared until the state machine enters a data state. The write pointer should also be cleared if a collision occurs.

Finally, the prescale counter shared by the port controllers is placed within this design unit. A **prescale** output is generated when the counter reaches its limit (1024). The **prescale** output should be registered once to ensure proper synchronization with the transmit clock.

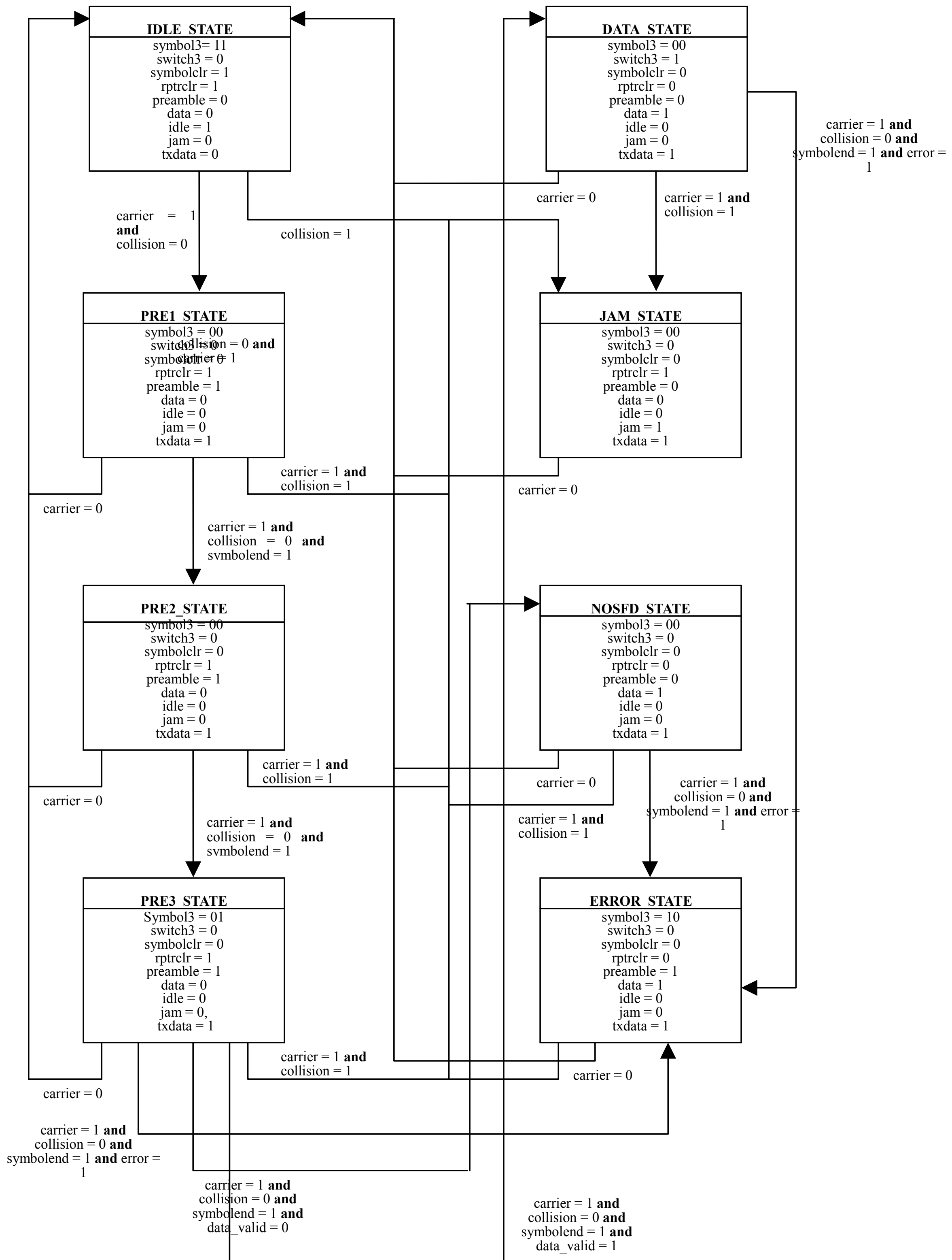
There are several outputs in each state. **Symbolclr** is used to clear the 3-bit counter that controls the symbol boundaries, **preamble** indicates that the core controller is generating preamble, **data** indicates that core controller is transmitting data, **idle** indicates that the network is idle, and **jam** indicates that there is a collision and jam characters are being transmitted. All of these signals are propagated to the top level for observation. **Txdata** is used in each of the port controllers to assert **tx_en**. These signals do not need to be repeated in the design of the two additional state machines that control the other two transmit pairs.

The **symbol** and **switch** outputs are used to control two output multiplexers for each transmit pair. **Symbol** indicates which symbol to generate. **Switch** indicates whether the symbol or FIFO data is selected to transmit to outputs. The **symbol** and **switch** signals are separate for each pair so that one transmit pair may transmit symbols while another transmits data.

The I/O port description of the core controller is given below:

Port Name	Port Type	Width	Description
txclk	input	1	Receive clock
areset	input	1	Asynchronous reset
carrier	input	1	Indicates carrier asserted
collision	input	1	Indicates collision condition
rx_error	input	1	Indicates RX PMA error
rx_dv	input	1	Indicated SFD found in data
symbolend	input	3	Indicates end of symbol line
symbolclr	buffer	1	Reset symbol counter
symbolinc	buffer	1	Increments symbol counter
symbol	buffer	6	Symbol select
switch	buffer	3	Data select
wptrclr	buffer	1	Write-pointer clear
wptrinc	buffer	1	Write-pointer increment
rptrclr	buffer	1	Read-pointer clear
rptrinc	buffer	1	Read-pointer increment
txdata	buffer	1	txdata is ready
idle	buffer	1	Indicates idle generation
preamble	buffer	1	Indicates preamble generation
jam	buffer	1	Indicates jam generation
data	buffer	1	Indicates data generation
prescale	buffer	1	Prescale output to port

Fig 1.0: State Transition diagram



11. Symbol Generator and Output multiplexer

The character symbol generator and output multiplexer will generate symbols. These symbols are the

- bad characters, transmitted to indicate a receive error,
- jam characters, transmitted to indicate collision,
- idle characters, transmitted to indicate there is no activity on the network, and
- preamble characters, transmitted to allow for carrier sensing and clock recovery by the receiving nodes.

A 3-bit counter (symbol counter) is used to time the transmission of the SFD (start of frame delimiter) symbol and subsequent data on transmit. One pair is transmitted when counter reaches the value of 1; the next pair begins at 3, and next at 5. The counter continuously counts rolling over after counting though 6, to indicate symbol boundaries and ensure that the data transmitted is separated by two clock cycles. The identifier **symbolcount** is used to represent the counter output value.

Symbol Generator functionality is given below in a tabular form.

Jam/preamble generation:

symbolcount(0)	**Jam**
0	10
others	01

SOSB and bad character generation:

symbolcount	**sosb1**	**sosb2**	**sosb3**	**bad1**	**bad2**	**bad3**
000	10	01	10	01	10	01
001	01	10	01	01	10	10
010	10	10	01	01	01	10
011	01	01	10	10	01	10
100	01	10	10	10	01	01
101	10	01	01	10	10	01
others	00	00	00	00	00	00

Symbolend1, **symbolend2**, **symbolend3** equations are as given below:
symbolend1 <= **symbolcount**(0) and not **symbolcount**(1) and **symbolcount**(2)
symbolend2 <= **symbolcount**(0) and not **symbolcount**(1) and not **symbolcount**(2)
symbolend3 <= **symbolcount**(0) and **symbolcount**(1) and not **symbolcount**(2)

The clear signal for the symbol counter is set either when **symbolclr** is high or when the counter value reaches 5. So an internal signal such as say **clearcounter** can be used for the symbol counter clear input. It can be defined as given below.

clearcounter <= **symbolend1** or **symbolclr**

The output multiplexer basically has six multiplexers (one for each of the transmit signals of the three transmit pairs). The multiplexers are paired with the transmit pairs, each pair sharing the same select lines. There are five inputs to each multiplexer, so three select lines per pair are required. This is a total of nine select lines for all the multiplexers. The outputs are the transmit signals **txd5** – **txd0**.

The function for each multiplexer listed in a tabular form is given below:

Symbol1 multiplexer:

symbol(1 downto 0)	smuxout
00	jam
01	sosb1
10	bad1
others	00

Symbol2 multiplexer:

symbol(3 downto 2)	smuxout
00	jam
01	sosb2
10	bad2
others	00

Symbol3 multiplexer:

symbol(5 downto 4)	smuxout
00	jam
01	sosb3
10	bad3
others	00

Switch1 multiplexer:

switch(1)	muxout
0	smuxout (1downto 0)
others	dmuxout (1 downto 0)

Switch2 multiplexer:

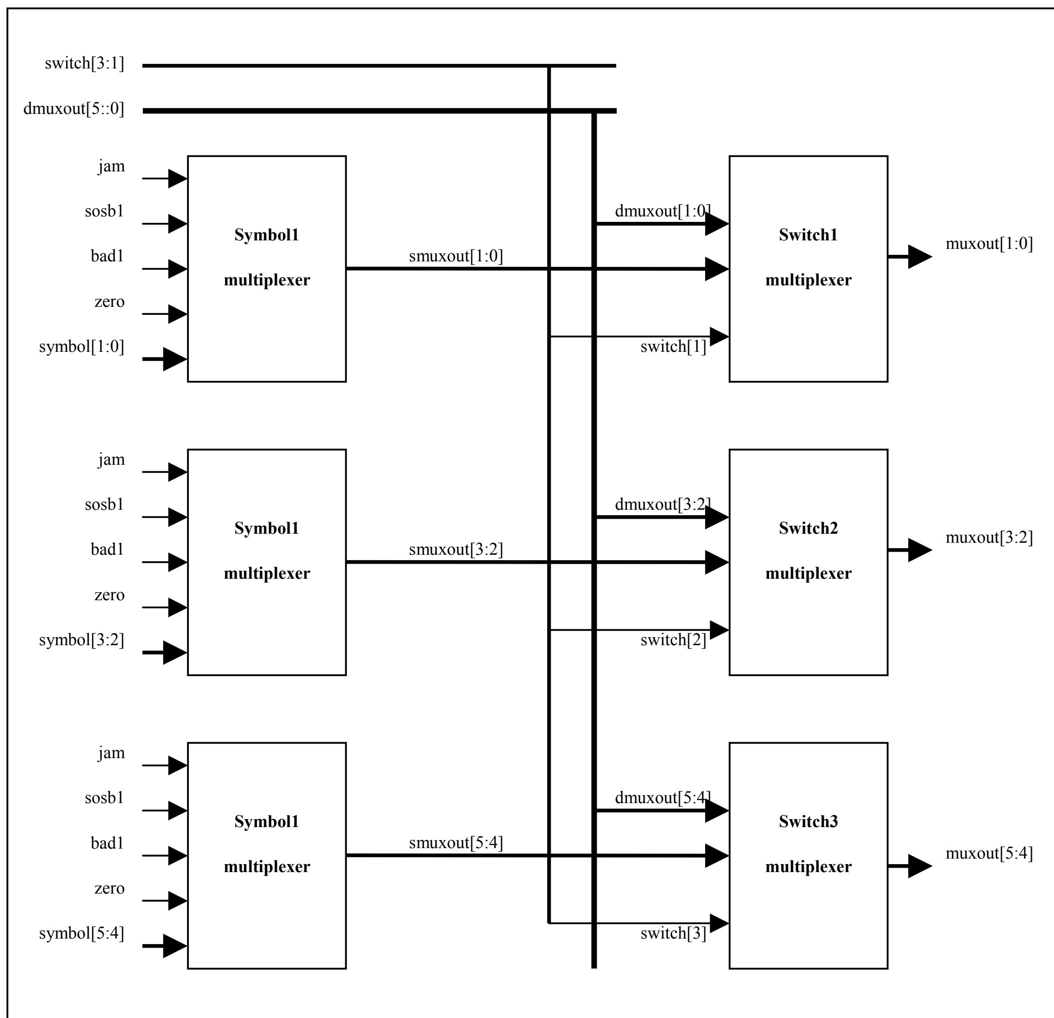
switch(2)	muxout
0	smuxout (3 downto 2)
others	dmuxout (3 downto 2)

Switch3 multiplexer:

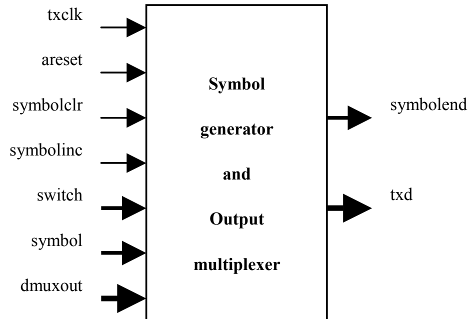
switch(3)	muxout
0	smuxout (5 downto 4)
others	dmuxout (5 downto 4)

The **txd** output is assigned the value of **muxout**, delayed by one clock cycle. The delaying is done to ensure proper synchronization with the transmit clock.

A structural representation of the output multiplexer is given below:



The block diagram and I/O port description of the Symbol generator and Output multiplexer are given below:



Port Name	Port Type	Width	Description
txclk	input	1	Tramsmit clock
areset	input	1	Asynchronous reset
symbolclr	input	1	Symbol counter clear
symbolinc	input	1	Symbol counter increment
switch	input	3	D/S switch control
symbol	input	6	Symbol mux control
dmuxout	input	5	FIFO data input
symbolend	buffer	3	End of line 1,2,3 symbol
txd	buffer	6	Transmit data