
# Final Report

Stephen "*Network Magic*" Caplan
Javan "*Wired*" Gargus
Kevin "*Encrypt This!*" Hackett
Paul "*The Overload*" Somogyi

Date: November 26, 1998
Instructor: Duncan Elliott
Class: EE 552

# Table of Contents

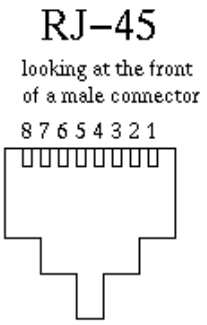
## IC Datasheet

Features:

- Uses Altera Flex10K FPGA on UP1 Prototype Board
- Compatible with IEEE 802.3 ethernet: 10BASE5, and 10BASE-T
- AUI or RJ45 Interface
- Semi-secure XOR based private key encryption
- Bridging Latency: 72 μs
- Bridging Speed: 10 Million bits / second
- Maximum Ethernet Diameter with 2 eSAFE's : 50m (workstation to hub)

RJ45 10BaseT Connector Pinout:

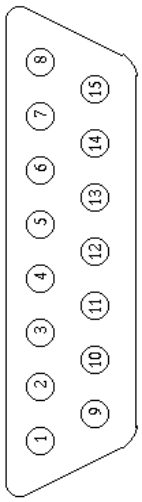
| Pin | Name | Description |
|---|---|---|
| 1 | TX+ | Transmit Data + |
| 2 | TX- | Transmit Data - |
| 3 | RX+ | Receive Data + |
| 4 | n/c | Not Connected |
| 5 | n/c | Not Connected |
| 6 | RX- | Receive Data - |
| 7 | n/c | Not Connected |
| 8 | n/c | Not Connected |




AUI (DB15) 10Base5 Connector Pinout

| Pin | Name | Description |
|---|---|---|
| 1 | n/c | Not Connected |
| 2 | CD+ | Carrier Detect + |
| 3 | TX+ | Transmit Data + |
| 4 | n/c | Not Connected |
| 5 | RX+ | Receive Data + |
| 6 | GND | Ground |
| 7 | n/c | Not Connected |
| 8 | n/c | Not Connected |
| 9 | CD- | Carrier Detect - |
| 10 | TX- | Transmit Data - |
| 11 | n/c | Not Connected |
| 12 | RX- | Receive Data - |
| 13 | 12V | +12V |
| 14 | n/c | Not Connected |
| 15 | n/c | Not Connected |

DB 15



## FPGA and UP1 Header Pinouts

| Pin Name | Pin No | Header No | I/O | Description |
|---|---|---|---|---|
| Enable | 41 | SW1 | I | ON: encyrption enabled<br>OFF: encryption disabled |
| TxE_A | 46 | A 16 | O | Transmit Enable (Ethernet Port A)<br>Held high while transmitting data |
| TxC_A | 49 | A 18 | I | Transmit Clock (Ethernet Port A)<br>10 MHz clock for syncronizing TxD |
| TxD_A | 51 | A 20 | O | Transmit Data (Ethernet Port A)<br>Serial Output data stream |
| RxC_A | 54 | A 22 | I | Receive Clock (Ethernet Port A)<br>10 MHz clock, syncronized with RxD |
| CRS_A | 56 | A 24 | I | Carrier Sense (Ethernet Port A)<br>Active low when carrier is detected |
| RxD_A | 62 | A 26 | I | Receive Data (Ethernet Port A)<br>Serial Input data stream |
| COL_A | 64 | A 28 | I | Collision Detect (Ethernet Port A)<br>Held high when collision detected |
| TxE_B | 78 | A 40 | O | Transmit Enable (Ethernet Port B)<br>Held high while transmitting data |
| TxC_B | 80 | A 42 | I | Transmit Clock (Ethernet Port B)<br>10 MHz clock for syncronizing TxD |
| TxD_B | 82 | A 44 | O | Transmit Data (Ethernet Port B)<br>Serial Output data stream |
| RxC_B | 84 | A 46 | I | Receive Clock (Ethernet Port B)<br>10 MHz clock, syncronized with RxD |
| CRS_B | 87 | A 48 | I | Carrier Sense (Ethernet Port B)<br>Active low when carrier is detected |
| RxD_B | 94 | A 50 | I | Receive Data (Ethernet Port B)<br>Serial Input data stream |
| COL_B | 97 | A 52 | I | Collision Detect (Ethernet Port B)<br>Held high when collision detected |
| CLK | 91 | n/a | I | System Clock |
| RESET | 212 | n/a | I | System Reset |
| LED7L6 | 6 | DIGIT1 : a | O | Seven Segment LED<br>These leds are used to display system status to facilitate problem isolation during testing. These have been used to display:<br>• Frame Count by Carrier Sense<br>• Collision Count<br>• RX Clock (div 5 million)<br>• TX Clock (div 5 million)<br>• Contents of frame data |
| LED7L5 | 7 | DIGIT1 : b | O | |
| LED7L4 | 8 | DIGIT1 : c | O | |
| LED7L3 | 9 | DIGIT1 : d | O | |
| LED7L2 | 11 | DIGIT1 : e | O | |
| LED7L1 | 12 | DIGIT1 : f | O | |
| LED7L0 | 13 | DIGIT1 : g | O | |
| LED7R6 | 17 | DIGIT2 : a | O | |
| LED7R5 | 18 | DIGIT2 : b | O | |
| LED7R4 | 19 | DIGIT2 : c | O | |
| LED7R3 | 20 | DIGIT2 : d | O | |
| LED7R2 | 21 | DIGIT2 : e | O | |
| LED7R1 | 23 | DIGIT2 : f | O | |
| LED7R0 | 24 | DIGIT2 : g | O | |

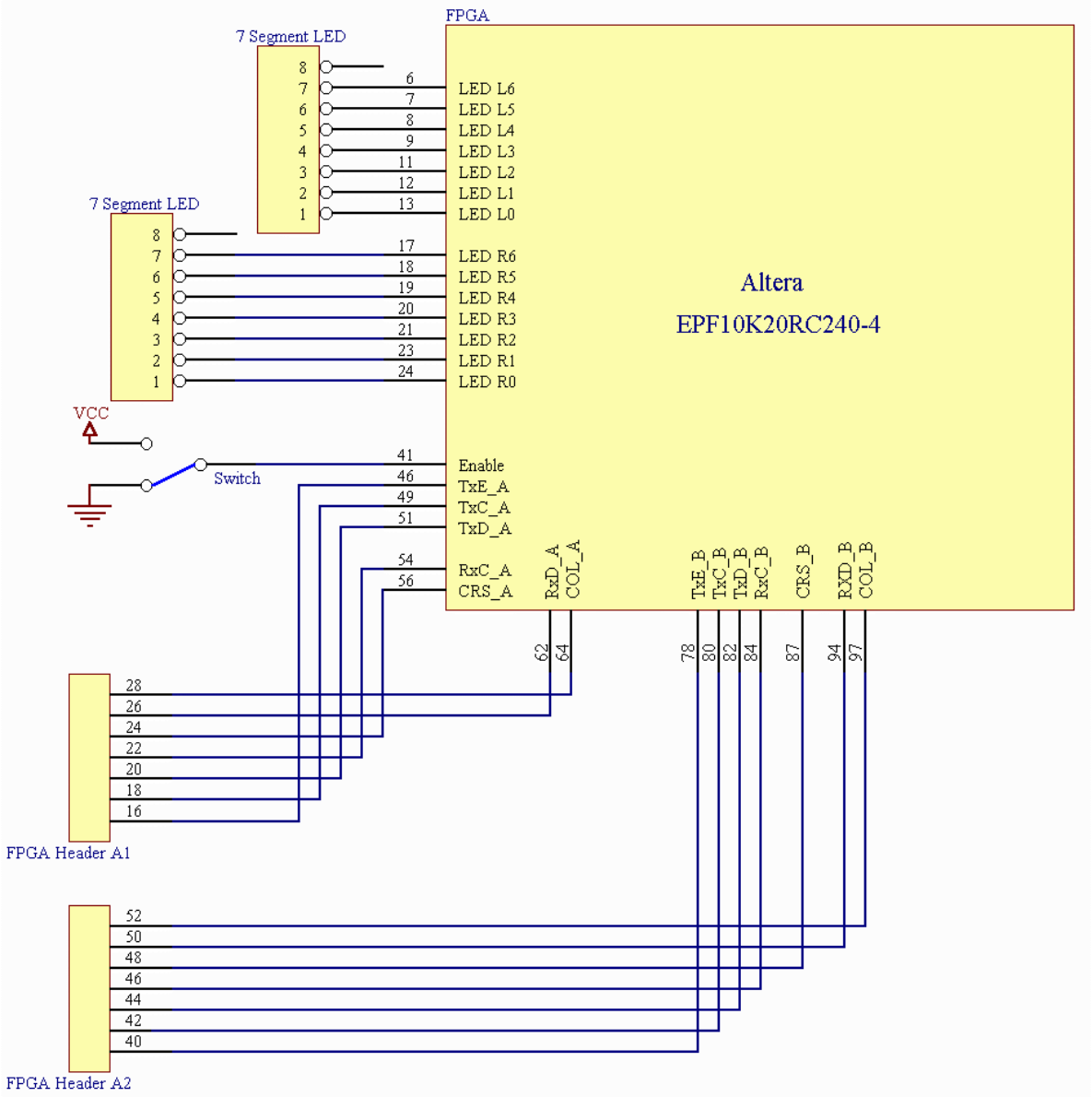
Input Power Requirements:
- 5V to Altera UP1
- 5V to Breadboard Power Bus
- 12V to Breadboard AUI Bus

Absolute Maximum Power Ratings
- Tolerance –2 to 7 V
- Tolerance –0.5 to 7 V
- Tolerance –15 to 15V

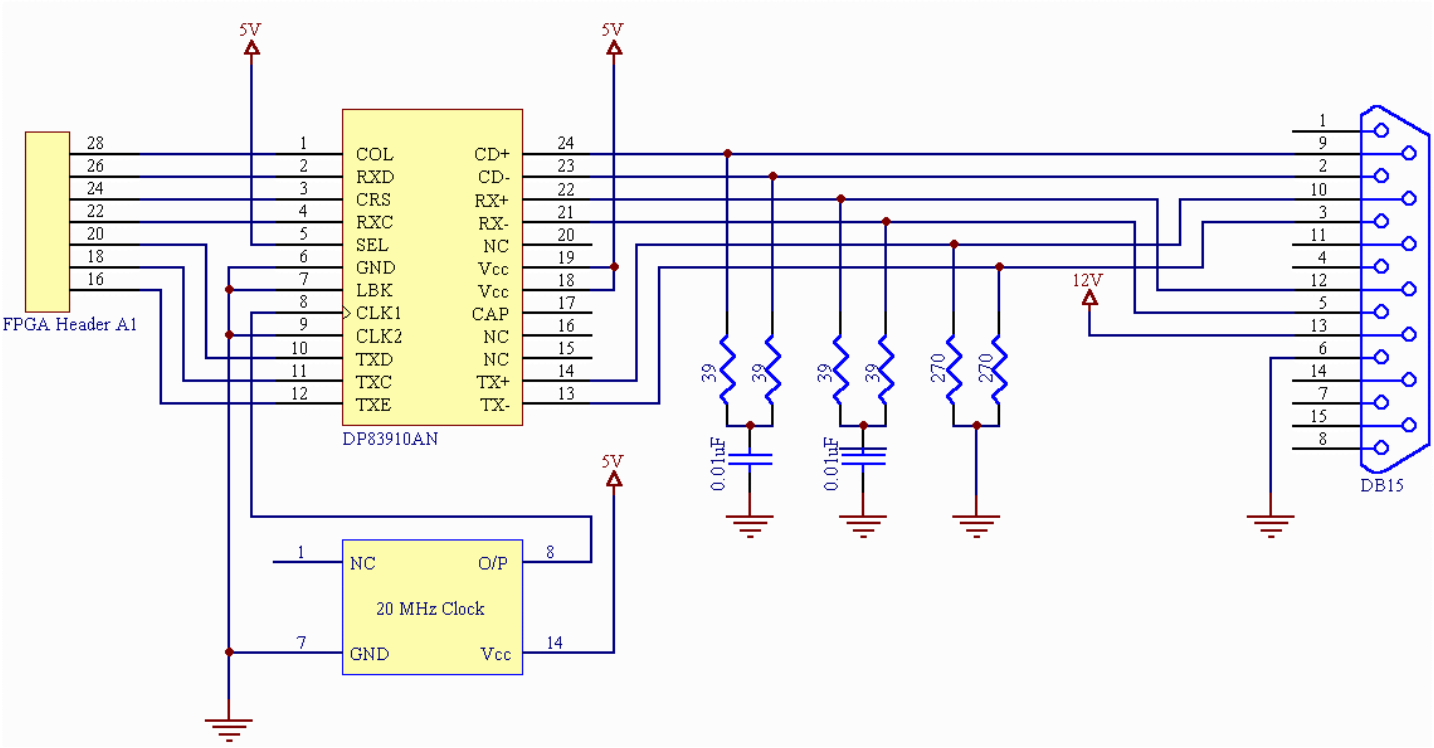
The system uses the following resources of the EPF10k20RC240-4 FPGA:

```
Total dedicated input pins used:          6/6      (100%)
Total I/O pins used:                     10/183    (  5%)
Total logic cells used:                1046/1152   ( 90%)
Total embedded cells used:               0/48     (  0%)
Total EABs used:                         0/6      (  0%)
Average fan-in:                          3.38/4   ( 84%)
Total fan-in:                         3544/4608   ( 76%)
```

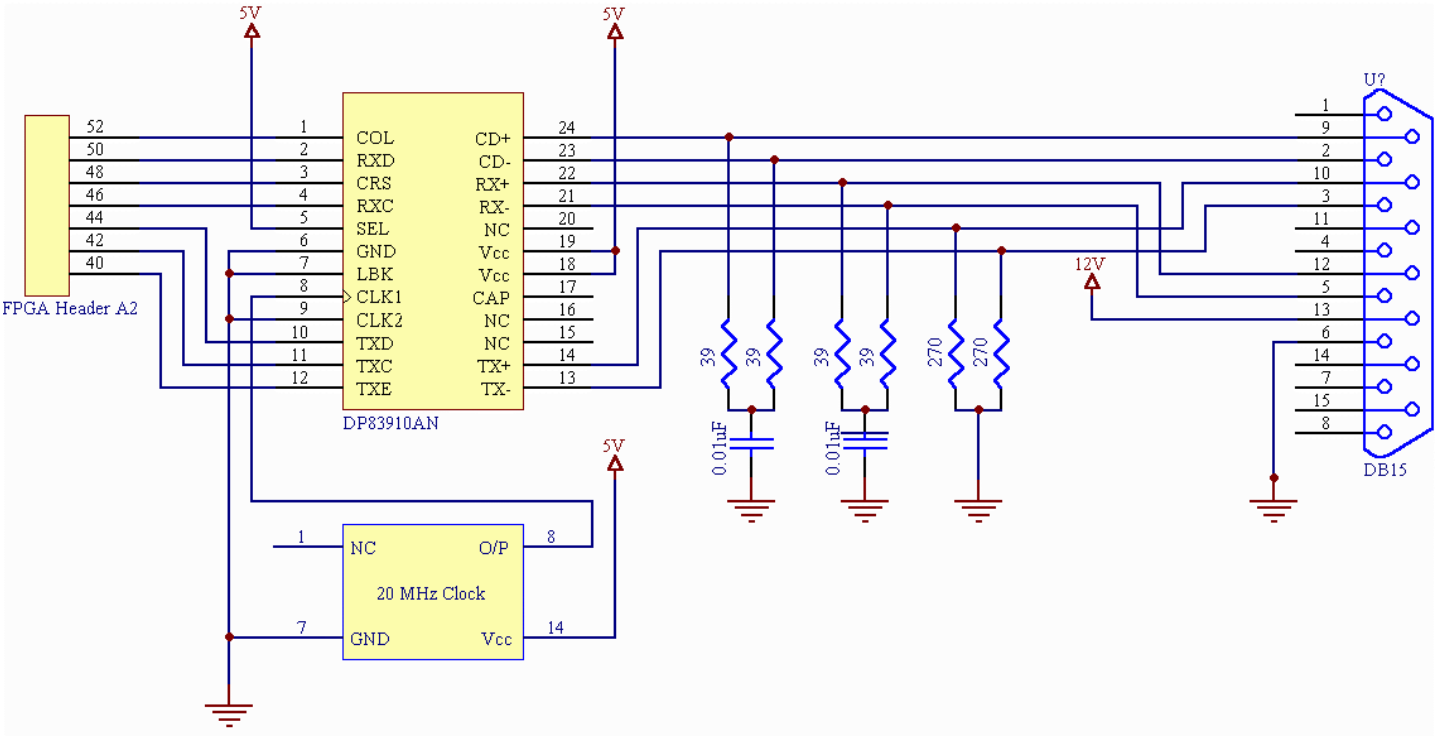
FPGA Pinout and Interconnect Schematic:



External Ethernet Connection: Port A




External Ethernet Connection: Port B



## Abstract

The eSAFE device will read and write to two separate ethernet segments and provide data security by means of a semi-secure encryption algorithm. On the incoming pipe, the data will first be read by our ethernet transceiver. The data portion of the current frame will then be encrypted while the headers are preserved for use on the outgoing pipe. After encryption, the frame will be immediately transmitted onto the outgoing pipe. This is a *cut through* design as opposed to our original *store and forward* proposal.

The goals of this project were to design and implement a digital system capable of transmitting encrypted data. These goals were met with varying degrees of success. First of all, we were forced to implement a much more simplistic encryption algorithm due to size constraints. A simple 'XOR' algorithm was used in place of the intended Blowfish algorithm. Although it was not implemented, the Blowfish encryption modules and all of its subsidiaries are included for completeness and on the chance that it may be implemented in the future.

Although simulation was successful, the prototype currently does not function as intended. Note that simulation is successful for the fully integrated device, as well as each of the separate modules. This implies that the problem lies in the physical implementation of our external interfaces.

Specifically, the problems that we are encountering are that we are not receiving the proper frame preamble; however, if we don't look for preamble, we are able to detect data transfer. We know that we are receiving data when we get a carrier sense. Nevertheless, it is necessary to point out that the eSAFE device does maintain a proper clock cycle on both the transmit and receive components of the ethernet interface.

## Project Description

Network traffic over a LAN or the Internet is inherently vulnerable to a number of forms of attack. Unauthorized access to a private data stream could compromise system security and personal or corporate privacy. The protection of sensitive data is the primary purpose of our project -- the eSAFE network bridge. The eSAFE bridge provides secure end-to-end data flow with bidirectional encryption and decryption of network traffic.

The eSAFE device will read and write to two separate ethernet segments and provide data security by means of a semi-secure encryption algorithm. In the outgoing direction, the data will be read in by our ethernet transceiver. The data portion of the current frame will then be encrypted, while the headers are preserved for use on the outgoing pipe. After encryption, the frame will be immediately transmitted onto the outgoing pipe. This is a *cut through* design as opposed to our original *store and forward* proposal. A near-identical process occurs in the incoming direction, except that the data is instead decrypted.

As reported earlier in our *Resource Requirement* document, a 20k gate chip cannot accommodate our Blowfish components. As a result, we have replaced it with a less secure, but much more efficient XOR-based encryptor. Although not used in our current implementation, this document contains descriptions and simulations of most of our encryption-related components: the Blowfish encryptor, the Blowfish Lookup Table generator, the Blowfish controller, and the Lookup Table arbitrator. While these have been omitted from our final integration, they are still included for completeness. Furthermore, should two 70k cell FPGAs be available, our full project, including Blowfish, can be wholly implemented. As mentioned in previous documents, the LCD display, keypad and TCP features have been dropped from the project.

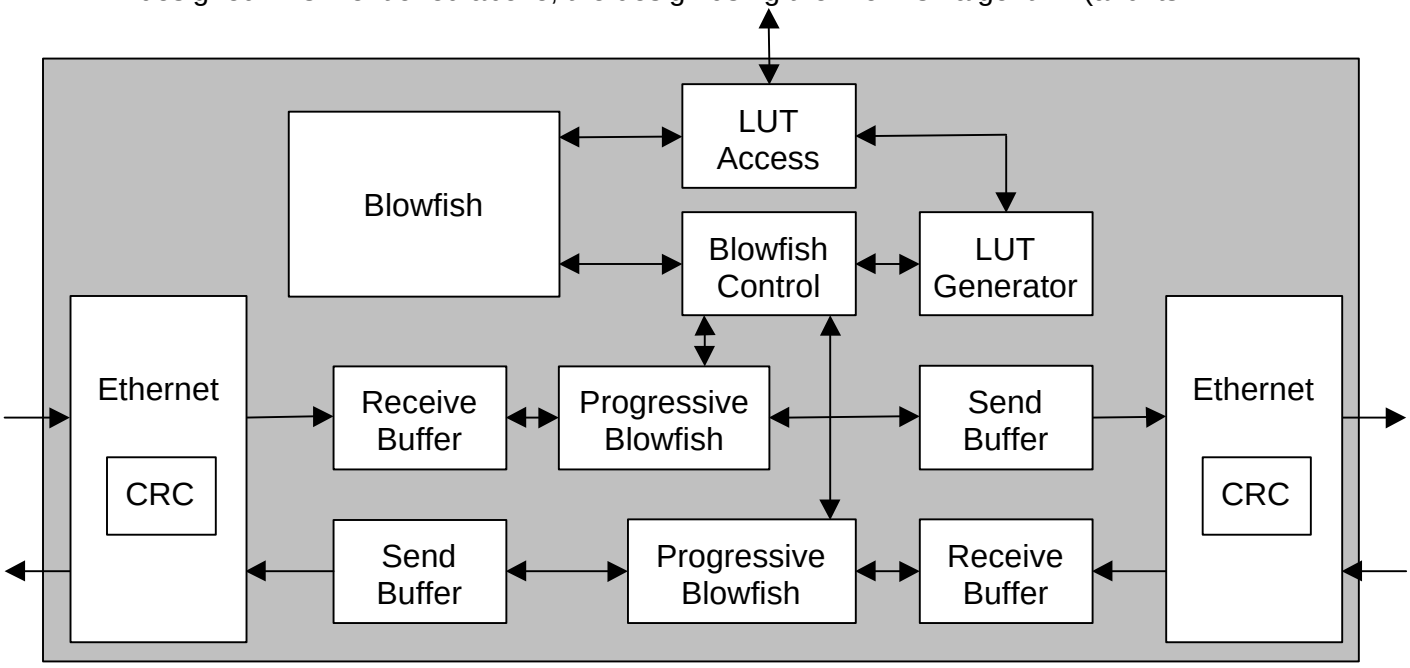
To provide a minimal, but functional system, our current implementation focuses on the ethernet interface including the transmitter, receiver, receive buffer, send buffer and an XOR-based encryptor that will replace Blowfish in our simulations and prototype.

Compilable code is included for the ethernet receiver, transmitter, CRC generator, Blowfish encryptor, Blowfish arbitrator, and Lookup Table arbitrator and memory access interface. Simulations are included for all the above components, along with our integrated system featuring the patent-pending XOR-o-Matic Teknologee.

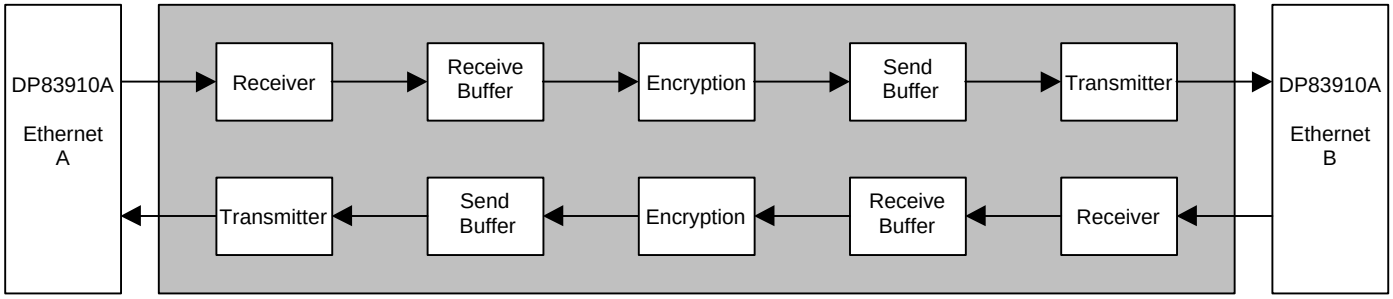
## Integrated System

### Design Details

The following block diagram shows the architecture of the overall chip, as originally designed. As mentioned above, the design using the Blowfish algorithm (and its




associated support modules) would not fit in the 20K gate FPGA, so the encryption was replaced with a byte-wise XOR for testing purposes. The design as tested is shown below:

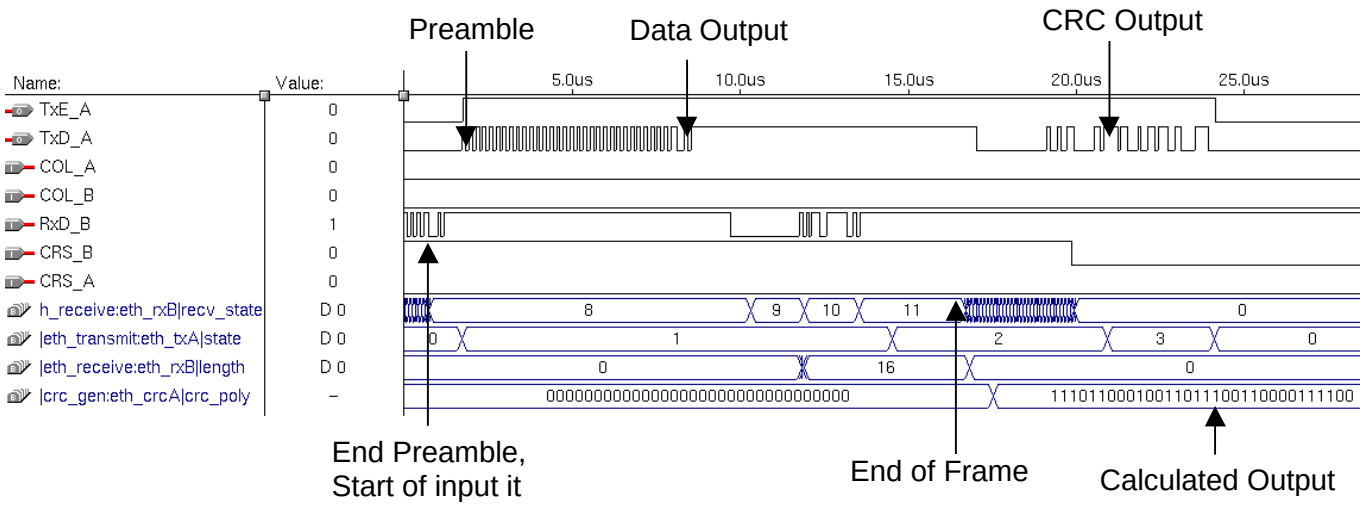



The resource requirements for this design are:

```
Total dedicated input pins used:          6/6      (100%)
Total I/O pins used:                     10/183    (  5%)
Total logic cells used:                1046/1152   ( 90%)
Total embedded cells used:               0/48     (  0%)
Total EABs used:                         0/6      (  0%)
Average fan-in:                         3.38/4    ( 84%)
Total fan-in:                        3544/4608    ( 76%)
```

**Simulation**

Since all other components have already been simulated independently, we will look only at a full end to end case for our integrated simulation. The following waveform illustrates receiving one frame and retransmitting it.




**Testing**

Initial testing on the device has been unsuccessful due to problems interfacing with the ethernet. Basically, we are not receiving the data that we are expecting.

For debugging purposes, we used the two FLEX 7-segment LEDs to display hex numbers.

By counting the number of rising edges on the Carrier Sense signals, we verified that we are receiving the frames. We then counted rising edges on the ethernet receive clock, which is driven by the ethernet chip, and displayed them. This performed as expected; the counters changed whenever a packet was being received and remained constant otherwise. We then displayed nibbles of the incoming data on the LEDs. The values displayed were always ‘0’ or ‘F’, and not the ‘A’ or ‘5’ (for alternating 1’s and 0’s) that would be expected during the preamble. A more thorough extension of this idea would be to store the entire frame in memory, and then use one of the pushbuttons to advance through it on the LEDs byte by byte. This is the most promising idea to pursue for further testing.

We also tested that our crystal oscillators for the transmit side were working. We displayed a count of the rising edges of this clock, divided by 5 million; as expected, the values changed approximately every half second.

When we removed the test for the preamble, and simply took the start of frame to indicate the start of the header, we were able to initiate a frame transmit. However, we are unsure whether the contents of the frame are valid. In order to provide some insight into where the problem may lie, a sniffer could be used to observe the frames we transmit.

## *Ethernet Interface*

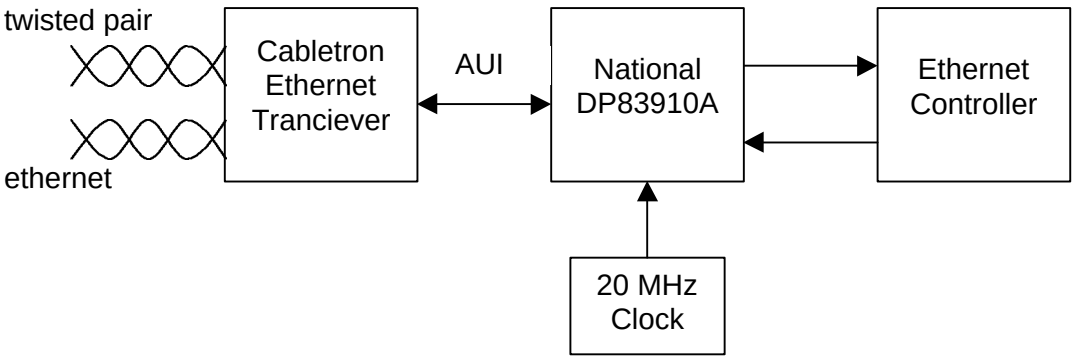
### Overview

This component functions as a controller for the National Semiconductor DP83910AN to provide an interface between the physical ethernet medium and the rest of the eSAFE encryption system. The ethernet controller provides a serialized digital logic data stream along with handshaking and control lines. The ethernet controller is composed of three main functional units: the receive and transmit interface and CRC generator.

The inbound interface is responsible for flagging the frame data payload in order to allow other component to later distinguish between data that should be encrypted and headers that must be left unchanged. The outbound interface will transmit properly formed ethernet packets, in conformance with the IEEE 802.3 specifications for ethernet. Component and integrated design details and annotated simulations are included below.

The national semiconductor chip provides an AUI ethernet connection, which is then attached to a Cabletron ethernet transceiver that is used to read and write data to 10BaseT ethernet on an unshielded twisted pair (UTP) cable, with an RJ45 connector.

The following diagram shows the ethernet interface to the external network components.




### Interface Definition

Receiver:
- Inputs: collision (*COL)*, carrier sence (*CRS*), receive data (*RxD*), and receive clock (*RxC*)
- Outputs: data stream (*eth_recv_data*), receive frame control signal (*eth_recv_frame*), and receive data control signal (*eth_recv_frame_data*)

Transmitter:
- Inputs: transmit clock (*TxC*), Data stream (*eth_send_data*), crc polynomial (*crc_register*)
- Outputs: transmit data (*TxD*), and transmit enable (*TxE)*

CRC Generator:
- Inputs: Data stream (*eth_send_data)*
- Outputs: crc polynomial (*crc_poly*)

**Design Details**

The national semiconductor chip simplifies three important tasks. First, it implements a phased lock loop (PLL) that extracts timing information from the ethernet data. It outputs a clock, with synchronized data for our receiver. All receiver operations within the FPGA rely on this clock for their operation. Second, the chip also provides clocking for our outgoing data, using an external 20MHz clock that is divided down to the 10MHz to which out outgoing data is synchronized. Finally, the chip converts between a non-return to zero (NRZ) serial data stream that can be input or output by the FPGA to the manchester encoded format used in ethernet.

An important modification to our original project specification is the removal of frame buffering on our outgoing interface. While this does provide significant cell reductions, the main motivation for this change was to simplify our overall design. By eliminating full frame buffering, we no longer have to deal with frame retransmission resulting from a collision. Instead on any error, we can safely drop the frame, and leave retransmission to the frame's originator.

An additional benefit is the decrease in the transmission latency of the frame. With our previous *store and forward* model, no transmission would be attempted until the entire packet had been stored. In the best case scenario, with the smallest frame size of 64 bytes, the minimum frame latency would be 512 μs (512 bit times). Typically, this would be significantly higher. However, with our current *cut-through* implementation, the total latency of our device is only 7.2 μs (or 72 bit times).

However, some buffering is still required by the outgoing controller. Since the transmitter is only notified of a new outgoing frame incident with the first bit of the frame address field, a 64 bit shift register is used to accommodate for the ethernet preamble and start delimiter that must be transmitted before the rest of the frame. This same buffer is also used for the crc checksum, which requires 32 bit periods after the last data bit for checksum calculation.

A further update to our original was design is with our handling of collisions. Since the eSAFE works at the network's physical layer collisions really are not our concern. Much like a repeater, the eSAFE now sends all received data on the opposite port, even if it knows data is currently arriving on this port. Obviously, this will create a collision – this is a good thing. Better still, the collision will be heard on *both* ports. That is to say, that the eSAFE is no longer attempting to split ethernet segments or collision domains. Instead, it can now be considered a part of the physical medium of a single segment.

This change is possible because of our change to a low latency cut-through design. According to the 802.3 specification, the maximum round trip diameter of an ethernet segment is limited to maximum of 575-bit periods[1] (ie. An end to end latency of 575 μs). Since each path will include two eSAFEs and we are concerned with the round trip time, the eSAFE device will introduce 288μs (72x2x2) of additional latency.

Effectively, securing a network with eSAFE will reduce the maximum network physical diameter by about 50% ( 288/575 ). The maximum workstation to hub cable length in

[1] Quick Reference Guides to 10 Mbps Ethernet : "Calculating Round Trip Delay Time", http://mojo.ots.utexas.edu/ethernet_old/10quickref/ch7qr_7.html

ethernet is given as 100m. Adding eSAFE security to such a network would decrease this to 50m. Considering the benefits in terms of network security, the sacrifice of in terms of segment diameter is considered to be worthwhile.
As a side note, it would be possible to decrease the end to end latency of the system if the entire preamble was not regenerated. That is, if we received only 60 (of 64) preamble bits, we could retransmit a preamble of only 60 bits. This would decrease the end to end latency of our systems by an order of magnitude to 8 μs, and would allow 95m segments. As a design decision, we have elected to adhead to a strict frame format compliant with the 802.3 specification, and to accept the significant loss in network diameter.

For the testing of our final implemented design, we have removed the CRC generator from the transmitter. Instead, we are preserving the original CRC by considering it to be part of the frame's data. If the data is encrypted, the CRC will obviously be incorrect for the frame – but only until it is decrypted. This was done simply to remove a potential source of error, in attempt to diagnose other problems.

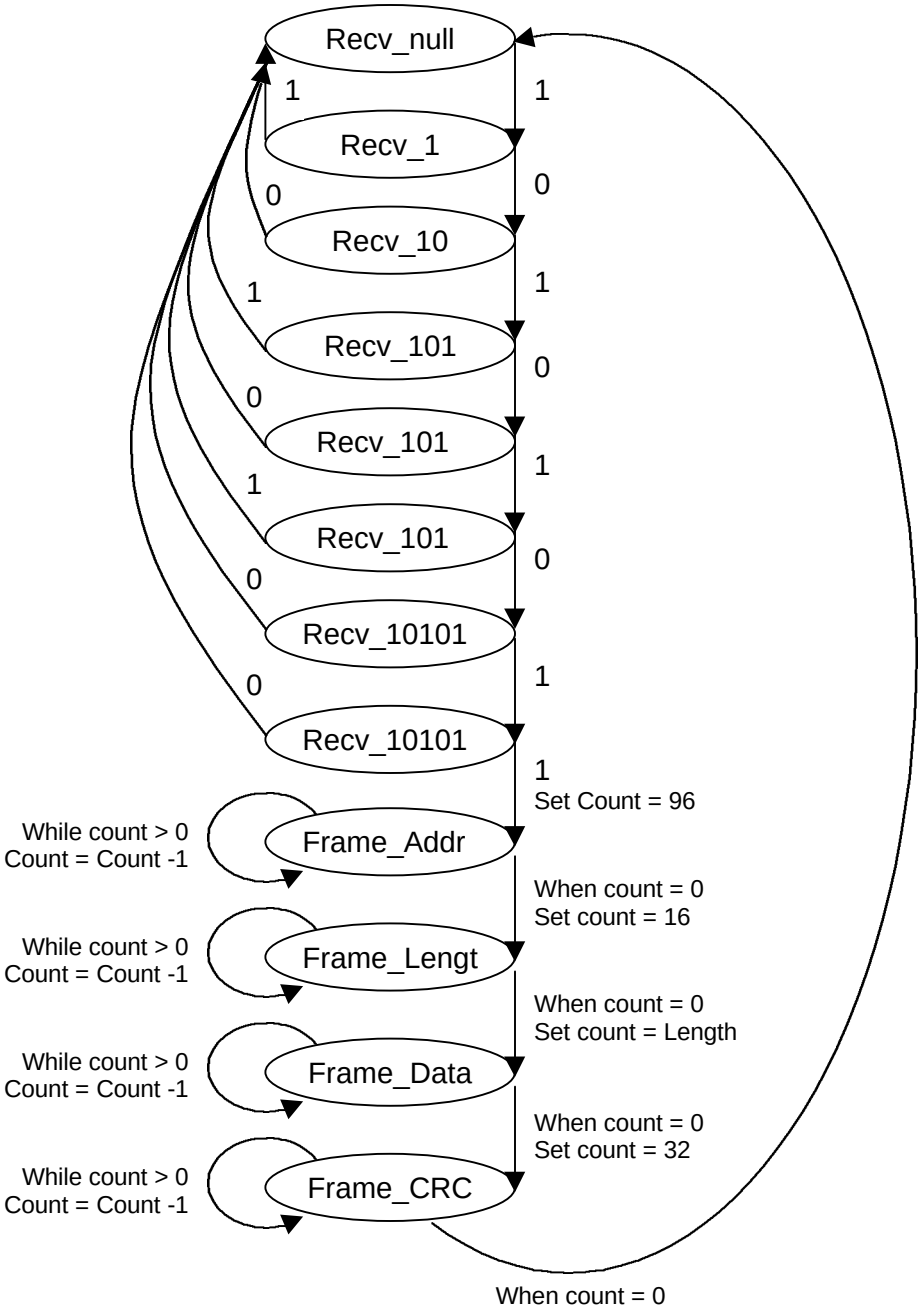
For the purposes of testing within a single non-switched ethernet segment, this will not impede the systems functionality. However, CRC generation will need to be required for use on a switched ethernet or if the system were extended with TCP/IP capabilities. As a result, the design and simulation of CRC is still included for completeness. The simulations of other ethernet components also include operation with CRC.

Currently, the ethernet components require a total of 7 I/O pins for each interface, or 14 pins for the entire system. The crc generator requires 208 logic cells, the transmitter requires 94 and the receiver requires 139 cells. For full functionality we'll use 2 instantiations of each of these, so the total logic cell count for ethernet comes to:

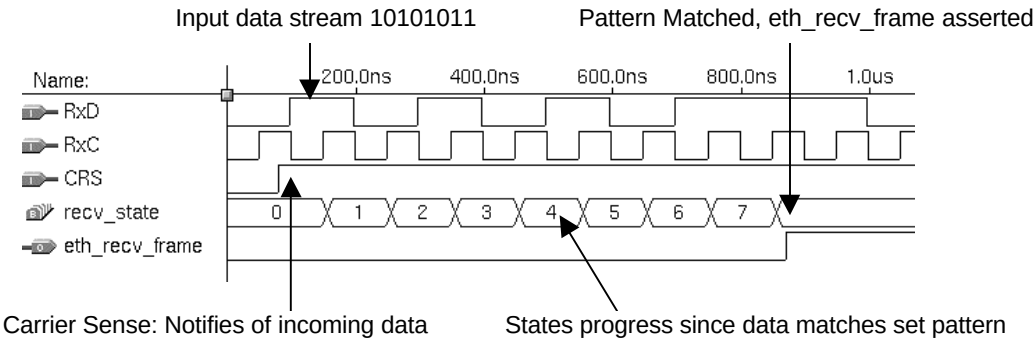
```
Receiver:    139 * 2
Transmitter:  94 * 2
CRC:         208 * 2
--------------------------------
Total       882 cells
```

**Ethernet Receiver Interface**

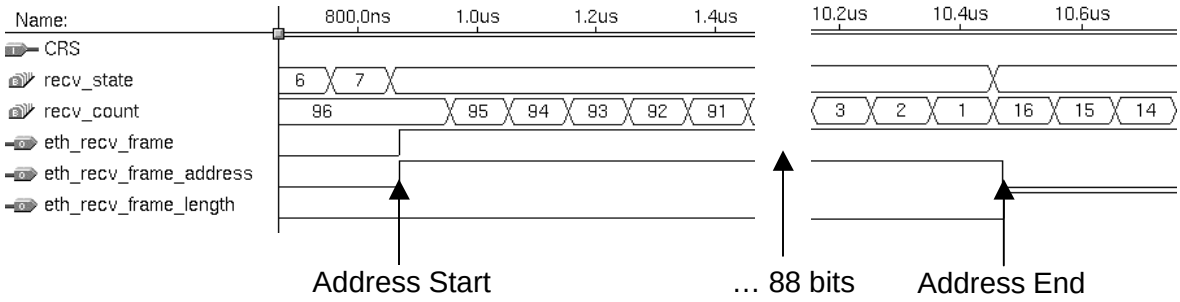
As diagrammed above, this interface receives serial data from the DP83910A chip. The state table for the receiver is shown below:



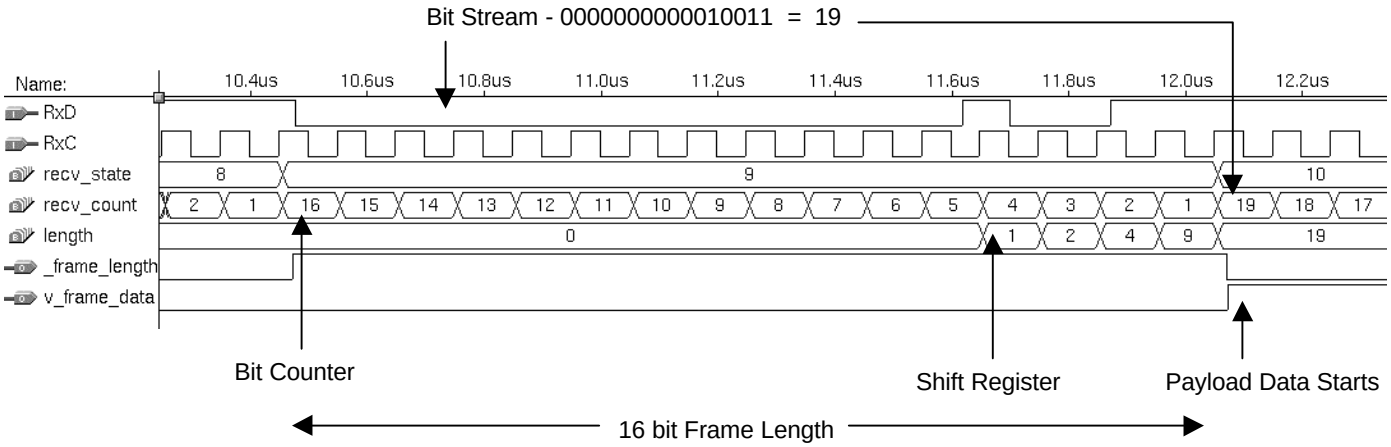
The receiver operation begins when a carrier is detected on the incoming channel (CRS is asserted). It then must remove the ethernet preamble (a 7 bytes string of 1-0-1) which is used by the 83910A to synchronize it's PLL with the data. A simulated start delimiter pattern match is demonstrated below:



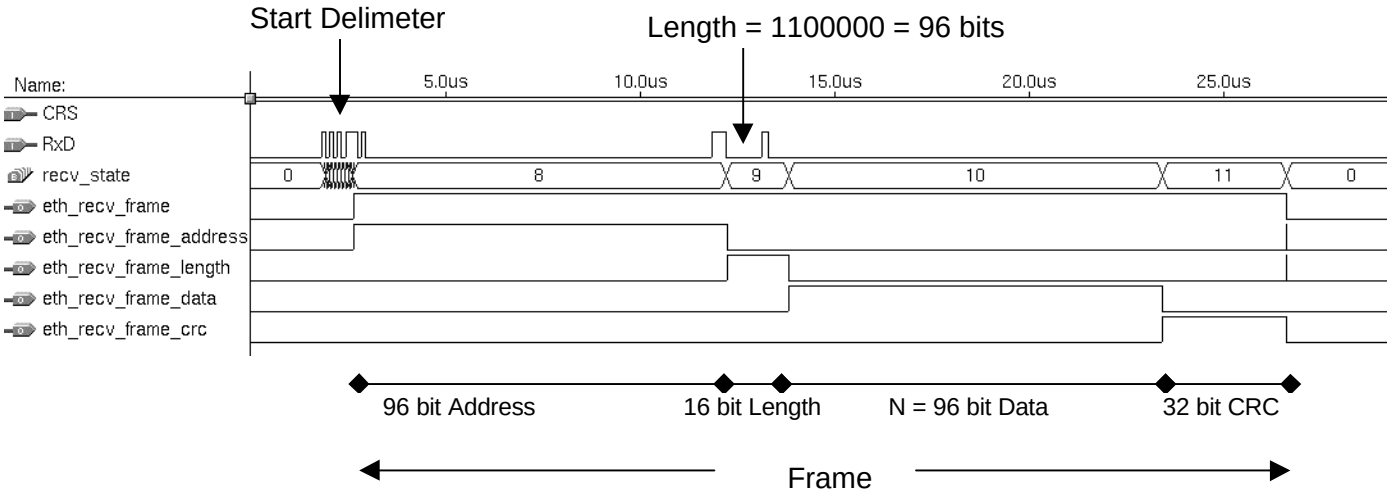

So while the preamble is received, the systems will oscillate between state 0 and 1, but will not assert eth_recv_frame until the start delimiter has been receiver.
After this frame delimiter has been detected, the following 96 bits will correspond to the destination and source ethernet MAC address (6 bytes, 48 bits each).



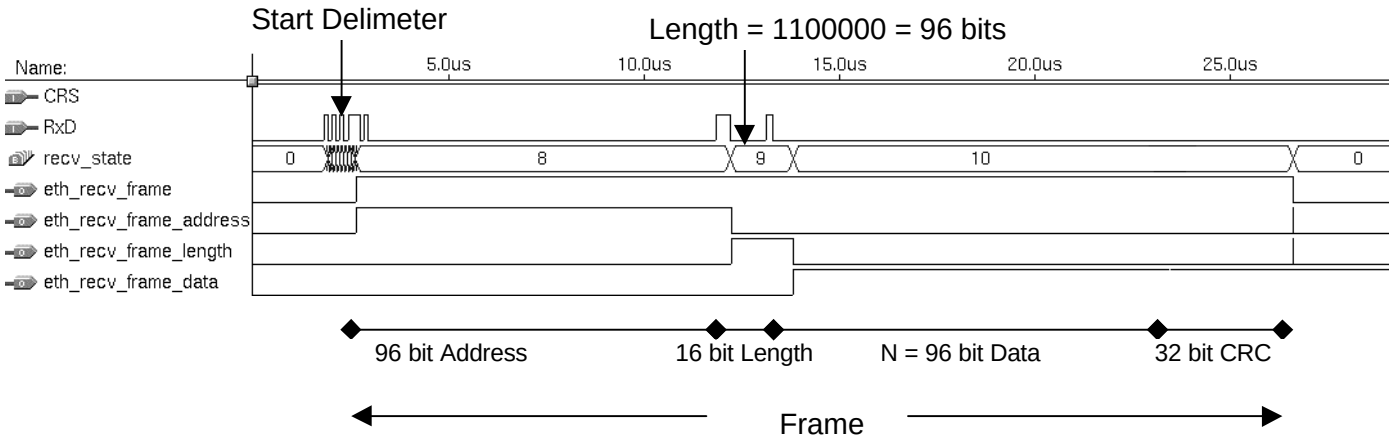

The next 16 bits are the length field that specifies the length of the data field. As the bits arrive they are shifted into a register that will later be used as a counter to mark the end of the data field. In practice if this value is below 384 bits (48 bytes) it indicates that padding is present. So we would always need to count through a minimum of 384 bits. However, in simulation this not practical, so we've allowed shorter values. The simulation below shows the input to the shift register, which is then copied to our bit counter:




After this follows the number of bits of data that was specified in the length field above, and then a 32 bit CRC checksum, so an entire frame might look like this:

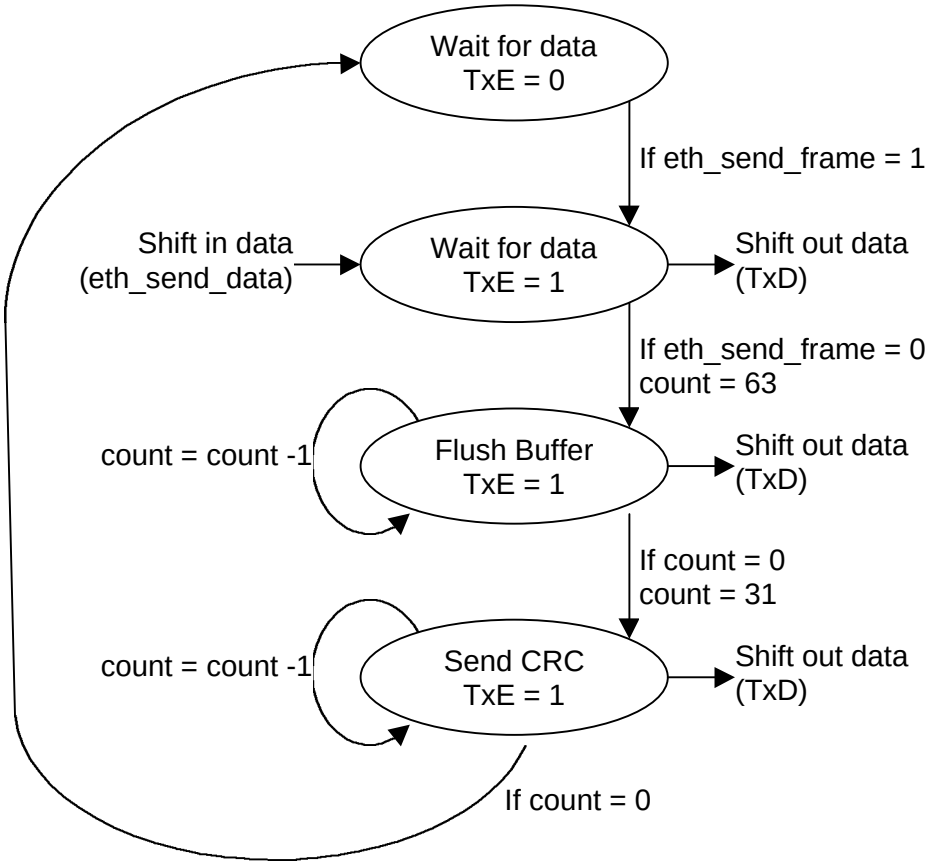



As previously mentioned, for our implementation testing, we are including the CRC as part of the frame data, so the frame will look like this:

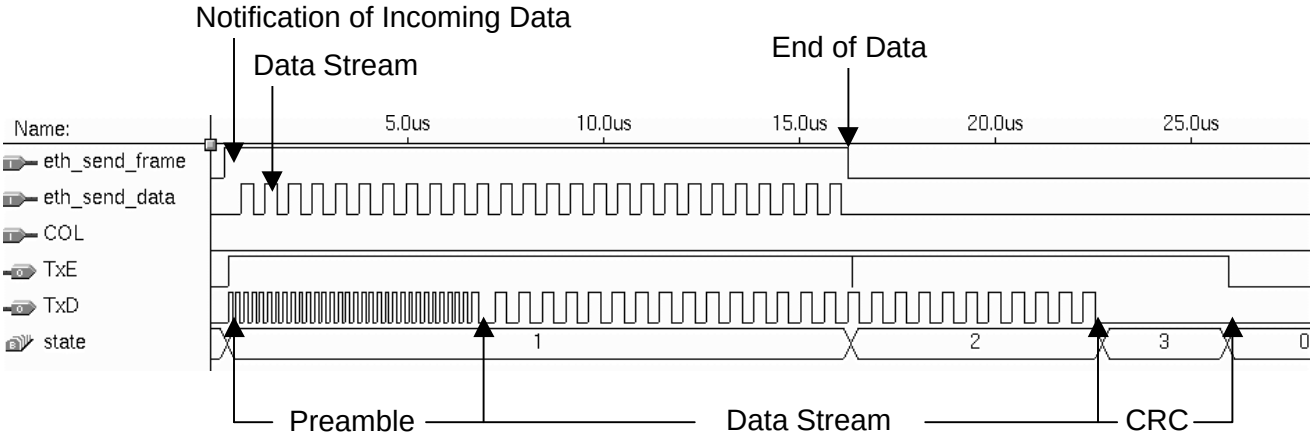


**Ethernet Transmitter Interface**

The state table for the receiver is shown below:

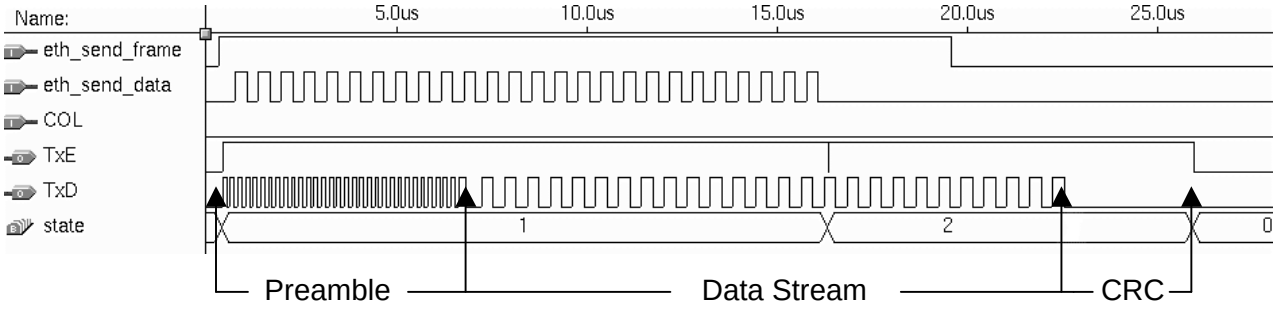


The ethernet transmitter goes into action when the *eth_send_frame* signal is asserted. While this signal is high incoming data will be shifted into a 64 bit buffer, and out onto the outbound interface. This buffer will be pre-loaded with the ethernet preamble, which must precede the rest of the frame. So when data is first shifted in the preamble will be transmitted. Since our output will always be 64 bits behind our input, the transmitter will need to send 64 bits once the input *eth_send_frame* signal is dropped in order to empty the buffer. The final stage is to output the CRC checksum, which will be copied into the last 32 bits of the buffer. When this has been sent, the transmitter will wait for the next frame to be sent. This is simulated below:




*Note: The CRC data output is all zero, since this case hasn't yet been integrated with the CRC generator. That will be shown in the integrated simulation.*

Additionally, it is important to note that in our design for testing the CRC field will be included within the data and state three will be eliminated, as shown:



**CRC Generator[2]**

CRC works by assuming that a data stream is an enormous binary number. If this number is divided by a second fixed binary value, the remainder is our CRC checksum. In order to confirm valid data, the recipient will follow the same process and ensure that the two checksums correspond.

The difficulty with this method comes with the size of the data stream. In an ethernet implementation, this could be as long as 12144 bits, which easily exceeds the capabilities of a standard 32 bit registers. To simplify this, we can do the division a little differently, while considering that we are only interested in the remainder of the division, not the quotient. By working with a register the same size as our target checksum, we are able to shift the serial data through the register, while simply using XORs as linear feedback to the register. Basically, this is a linear feedback shift register that works like this:

```
Load the register with zero bits.
Augment the message by appending N zero bits to the end of
it.

While (more message bits)
      Shift the register left by one bit, reading the next
        bit of the augmented message into register bit
        position 0.
      If (a 1 bit popped out of the register during step 3)
      Then Register = Register XOR Generator Poly.
End

The register now contains the remainder
```

*Source: A Painless Guide to CRC Error Detection Algorithms*
*http://www.repairfaq.org/filipg/LINK/F_crc_v3.html*

The CRC generator was tested with a number of generator polynomials of varying length, on various serial data streams. For simplicity and brevity, only a test case with a 4 bit (N=4) CRCs will be presented here, but this design will scale directly to a 32 bit ethernet implementation.

[2] CRC generation and checking are not included in our implemented design, for reasons previously noted. Simulations and design detail are included for completeness.

We will use the following inputs:

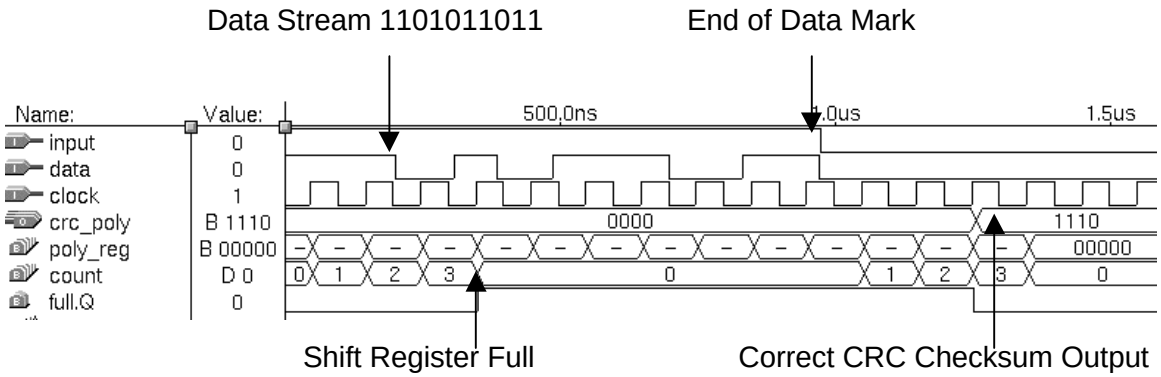
```
Generator Polynomial            :      10011
Original message                : 1101011011
Message after appending N zeros : 11010110110000
```

Through long division, we get the following result:

```
                1100001010 = Quotient
              _______________
10011 )  11010110110000 = Augmented message
         10011,,.,,....
         -----,,.,,....
          10011,.,,....
          10011,.,,....
          -----,.,,....
           00001.,,....
           00000.,,....
           -----.,,....
            00010,,....
            00000,,....
            -----,,....
             00101,....
             00000,....
             -----,....
              01011....
              00000....
              -----....
               10110...
               10011...
               ------...
                01010..
                00000..
                -----..
                 10100.
                 10011.
                 -----.
                  01110
                  00000
                  -----
                   1110 = Remainder = THE CHECKSUM!!!!
```

*Source: A Painless Guide to CRC Error Detection Algorithms*
*http://www.repairfaq.org/filipg/LINK/F_crc_v3.html*

The same result can be seen in the following test case:




It is important to note that the minimal acceptable input length is N (the width of our CRC output). Given less than N bits in the incoming data stream, no CRC output will be produced.

## *Receive Buffer*

### Overview

The receive buffer converts the serial bit stream from the ethernet receiver into bytes, and passes them to the encryption module. Note that the term *buffer* is used loosely here, since it does not buffer move than 2 bytes of data. This component also isolates the ethernet and system clocks.

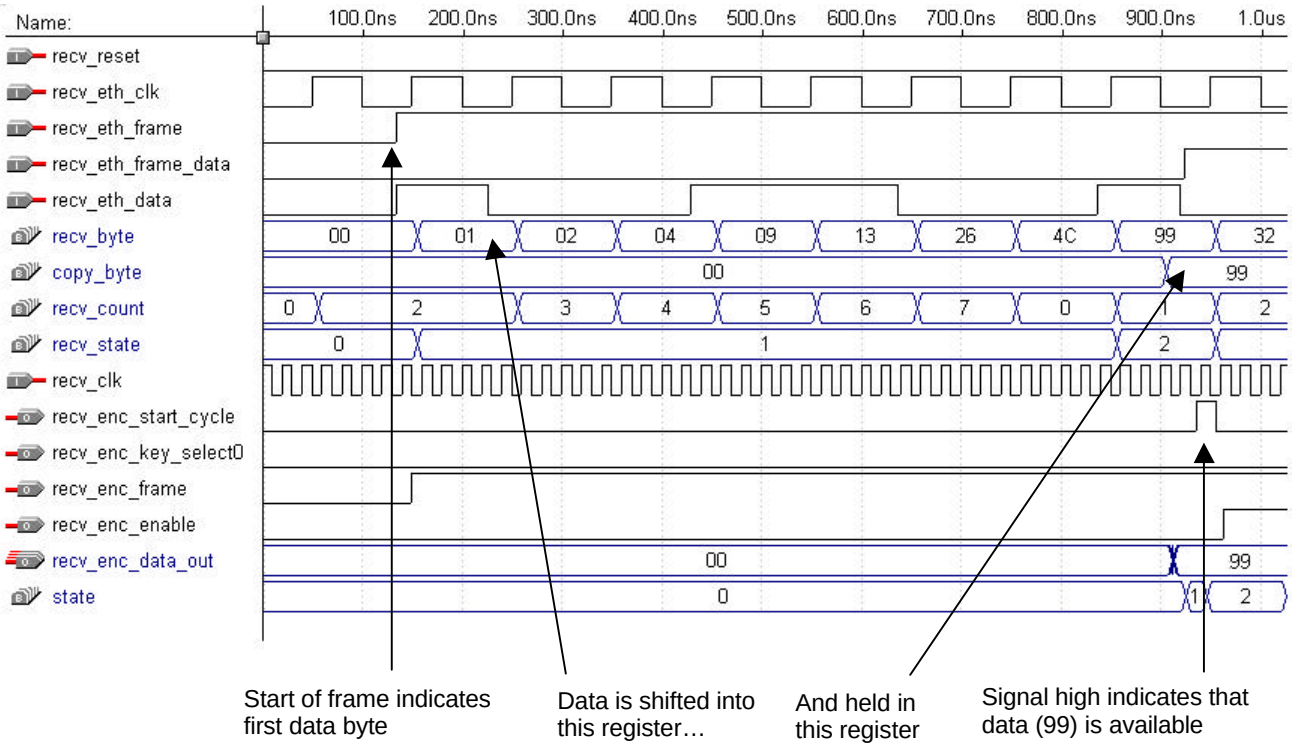
### Interface Definition

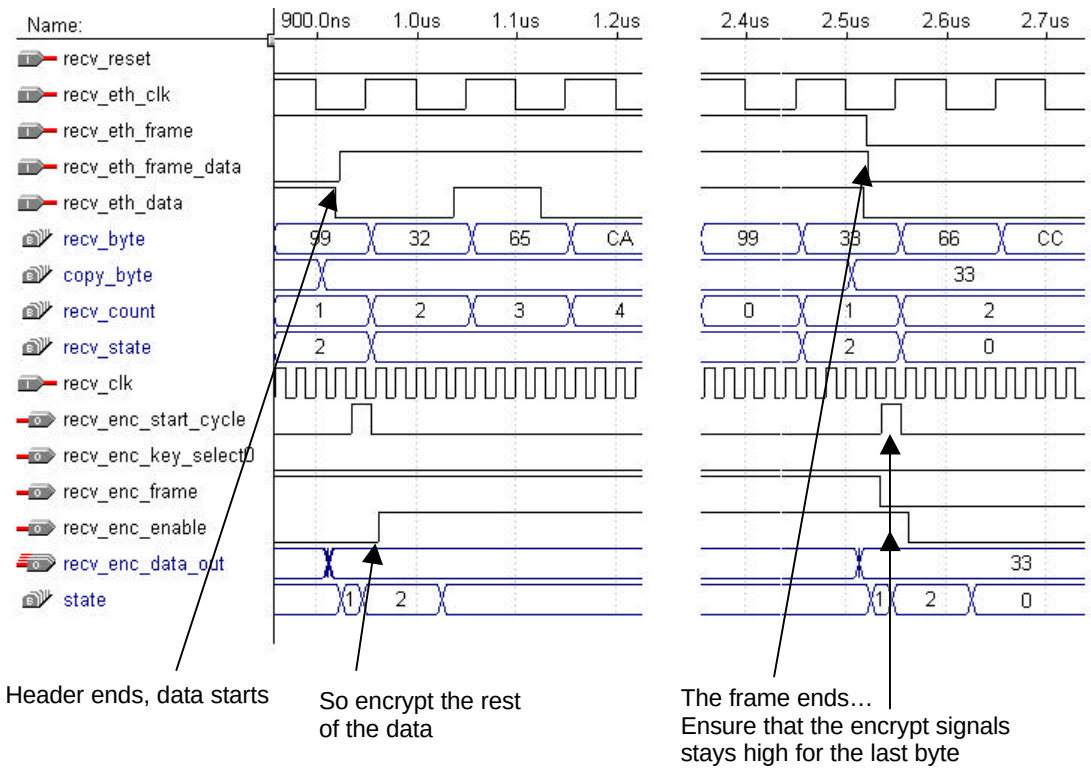
Serial data is accepted on the *recv_eth_data* line, which is clocked with the *recv_eth_clk* signal. The data stream is valid while *recv_eth_frame* is high. The parallelized bytes are asserted on the *recv_end_data_out* signals; the byte is valid when the *recv_enc_start_cycle* signal is high (it remains high for only one clock cycle). The *recv_enc_enable* signal is high if the byte should be encrypted, and low if not.

### Design Details

The ethernet module runs with a 100ns period clock, while the rest of the system runs at a faster clock (up to 20ns period). The bits from the ethernet are shifted into a register; when eight bits have been received, the byte is copied into a second holding register. This is performed by a state machine clocked with the ethernet receive clock. When the system-speed state machine detects that this has happened, it asserts a signal for one clock that indicates a new byte is available. It is assumed that the consumer is waiting for the data, and thus handshaking is unnecessary.

### Simulation






## *Progressive Encryption*

### Overview

The Progressive Encryption module controls encryption/decryption of the incoming data. Each input byte from the receive buffer is conditionally encrypted and passed to the send buffer. This module can easily be modified to support different encryption algorithms.

### Interface Definition

A new input byte on enc_data_in is signaled by asserting *enc_start_cycle*. Output bytes are put on *enc_data_out*, and *enc_done_cycle* is asserted. Again, there is no hand-shaking since each component must run at the required speed for real-time operation.

### Design Details

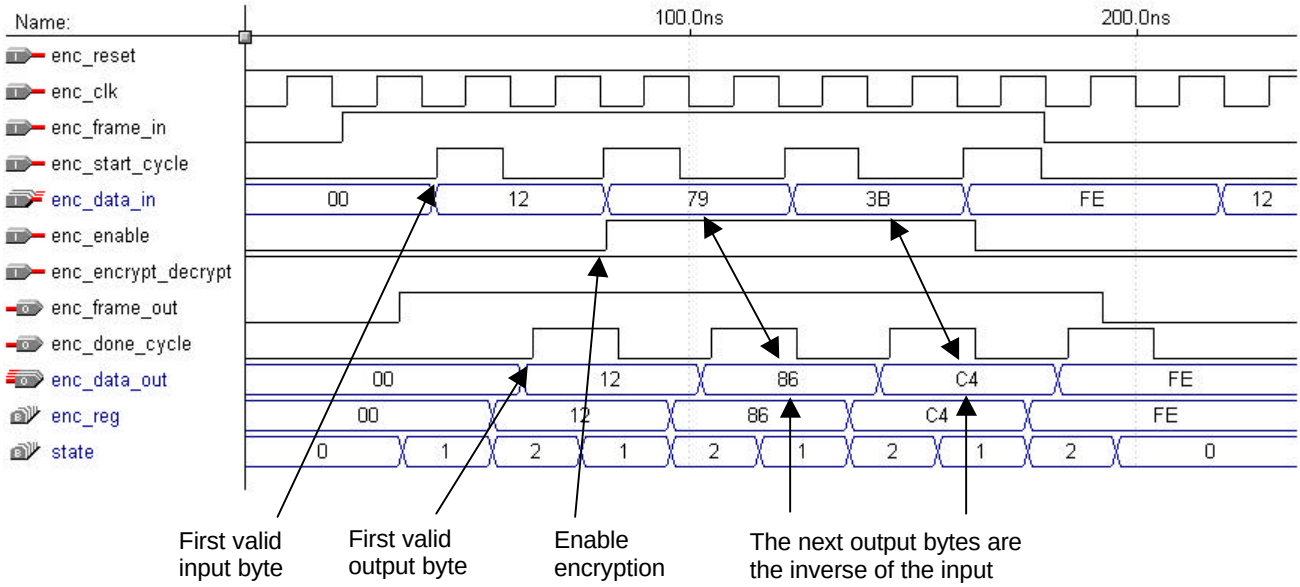
When the Blowfish algorithm is used, this must perform a *progressive* encryption: each input byte is XOR'd with the next byte in a 64-bit register, and the result is both sent to the transmitter and stored back into that byte in the register. The contents of the register are encrypted using Blowfish after every 8 bytes are received. This allows data that is not an even multiple of 8 bytes to be encrypted.

For purposes of demonstration, this module is much simpler. It merely passes all data bytes through to the send buffer, using a simple XOR to encrypt the data when encryption is enabled. An arbitrary 8-bit value is XOR'd with every byte to be encrypted;

for the simulation, this value is all 1's, so that the output is just the inverse of the input, allowing a visual inspection to verify that the data is being encrypted properly. Note that the same process is used to both encrypt and decrypt the data.

**Simulation**

Since we did not implement the full Blowfish encryption, the progressive encryptor to interface to it was not completed. However, the simple XOR, as described above, was implemented and fully tested.




## *Send Buffer*

**Overview**

The send buffer takes the bytes from the encryption module and serializes them to the ethernet transmitter. It is the counterpart of the receive buffer, again not performing much buffering. It also isolates the transmit ethernet and system clocks.

**Interface Definition**

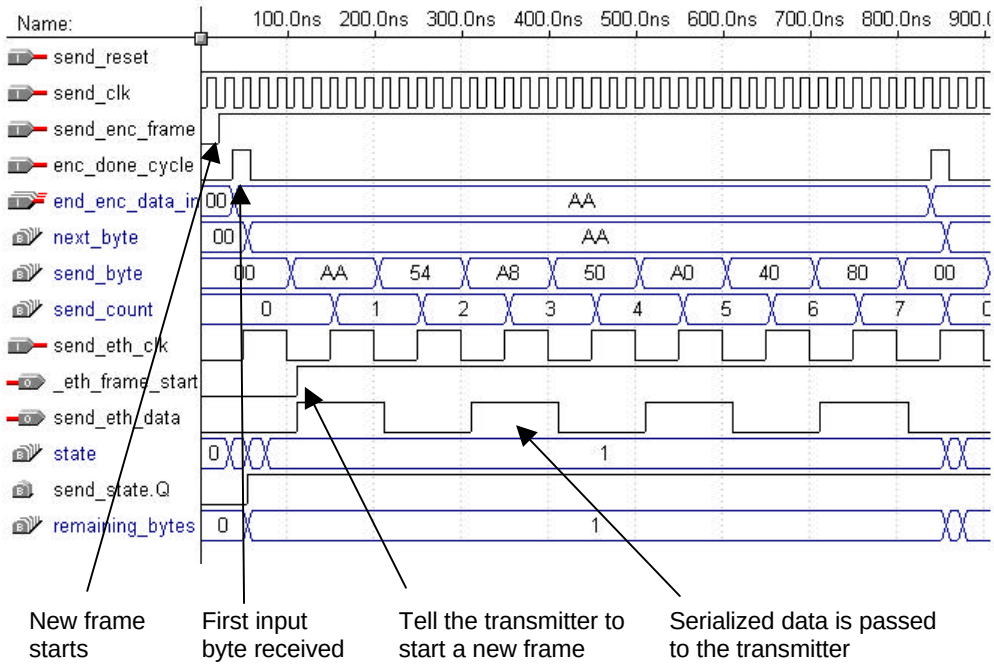
The *send_enc_frame* signals the beginning of a new frame. While this signal is high, a high on the *send_enc_done_cycle* indicates that the next data byte is on the *send_enc_data_in* bus. The *send_eth_frame_start* signal is asserted high when the first bit of the first data byte is ready to be transmitted. While this signal is high, the next data bit is available on *send_eth_data*, coincident with the rising edge of the *send_eth_clk* signal from the ethernet module.

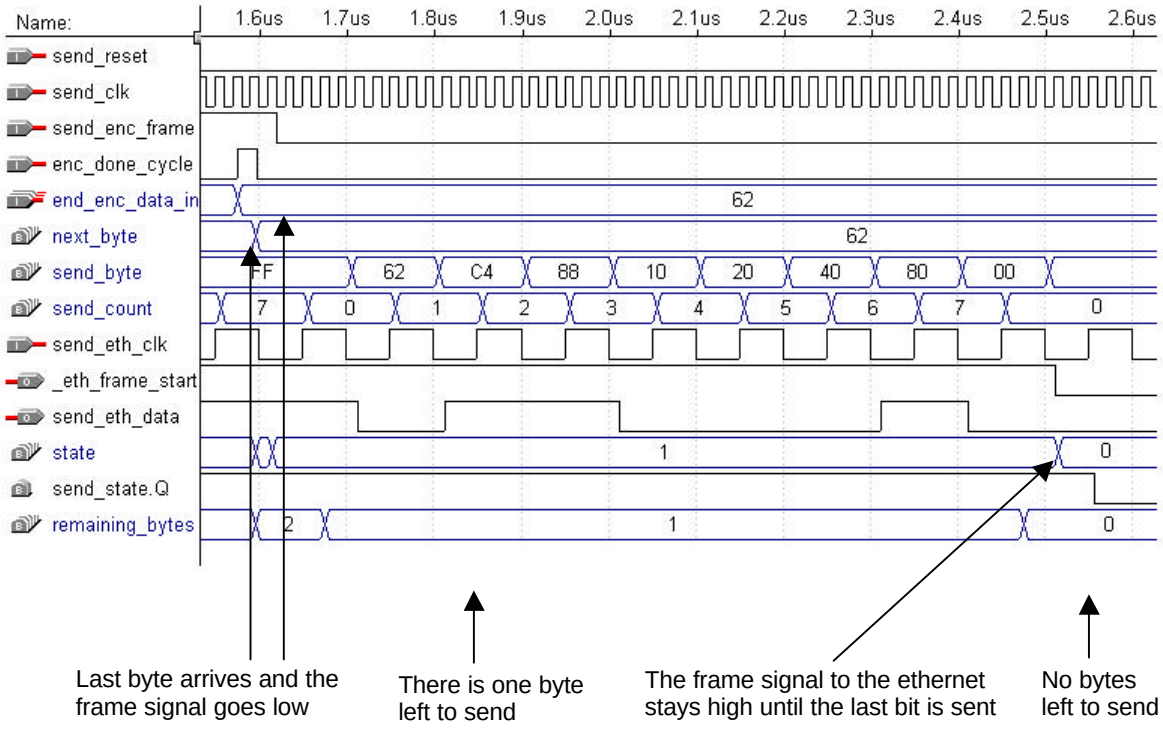
**Design Details**

Each input byte is loaded into a holding register, which is then shifted into the send register when it becomes empty. Bits are shifted out of the send buffer on every ethernet clock. The most significant difficulty encountered was maintaining the frame signal high until the last bit of the last byte is shifted out. This is accomplished by incrementing a counter whenever a new byte is received and decrementing the counter every time a byte has been fully transmitted. This required "converting" the byte_done signal which is valid on the edge of the slower ethernet clock, to a signal that is only high for one edge of the faster system clock.

Similar to the receive buffer, no handshaking is used since the send buffer and ethernet transmitter must be able to process data on every necessary clock edge. If the next byte is available before there is space to buffer it, or if it comes after the last buffered byte is transmitted, then the data is not at the proper rate for transmitting. The first problem will result if the system is not synchronized properly (i.e. we are assuming approximately the same time for each byte to pass through the receive buffer and encryption). The second problem results if the encryption module cannot perform at the required bit-rate (i.e. it has not finished processing the current data when the next data arrives).

**Simulation**




Name:
1.6us
1.7us
1.8us
1.9us
2.0us
2.1us
2.2us
2.3us
2.4us
2.5us
2.6us
send_reset
send_clk
send_enc_frame
enc_done_cycle
end_enc_data_in
next_byte
send_byte
send_count
send_eth_clk
_eth_frame_start
send_eth_data
state
send_state.Q
remaining_bytes
62
62
FF
62
C4
88
10
20
40
80
00
7
0
1
2
3
4
5
6
7
0
1
0
2
1
0
Last byte arrives and the frame signal goes low
There is one byte left to send
The frame signal to the ethernet stays high until the last bit is sent
No bytes left to send

## *Blowfish Encryption Unit*

### Overview

In order to provide encryption of the data, we will implement the Blowfish encryption algorithm. This unit is isolated to encrypting our data once it is received from the ethernet interface. The algorithm was developed by Bruce Schneier, in 1994, and is available in the public domain. He designed the Blowfish algorithm to be fast, compact, simple, and variably secure. With the availability of the algorithm (there is source code available to the public) and the criteria by which it was designed, the Blowfish algorithm seemed like a viable option for our encryption unit.

The Blowfish algorithm is separated into two parts -- Key Expansion and the actual data encryption. Before anything can be encrypted, a large number of subkeys must be calculated. Specifically, these subkeys are a P-array and four S-boxes. The P-array has 18 of these 32-bit subkeys, while each of the 32-bit S-boxes has 256 entries. These terms are important in implementing the algorithm into a design since the data encryption part of the algorithm relies upon them. Once these subkeys are computed, the encryption can take place.

### Interface Definition

In order to implement the Blowfish algorithm, there are a number of input/output pins required. According to the report file that Maxplus2 generates during compilation of the code, there are 111 of 183 input/output pins required (or 60% of the chip). However, since these are all implemented within the chip, this does not affect the wiring capabilities of the eSAFE project.

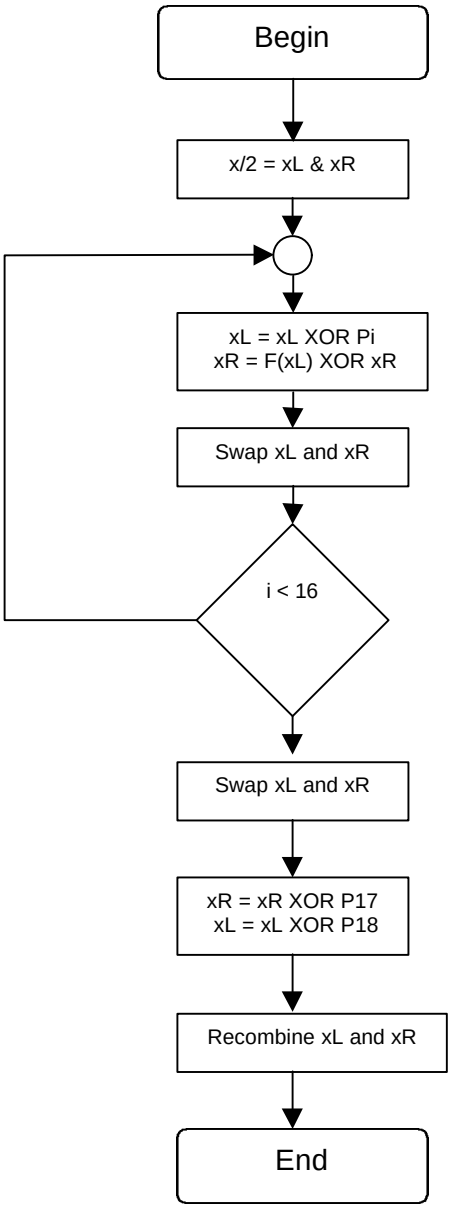
### Design Details

In order to implement the algorithm properly in VHDL, it is necessary to break down the unit into three separate components -- the Blowfish component (**blowfish.vhd**), the Blowfish encryption component (**bf_enc.vhd**), and the Blowfish counter component (**bfcounter.vhd**). The latter two components are necessary in order for the Blowfish component to work properly.

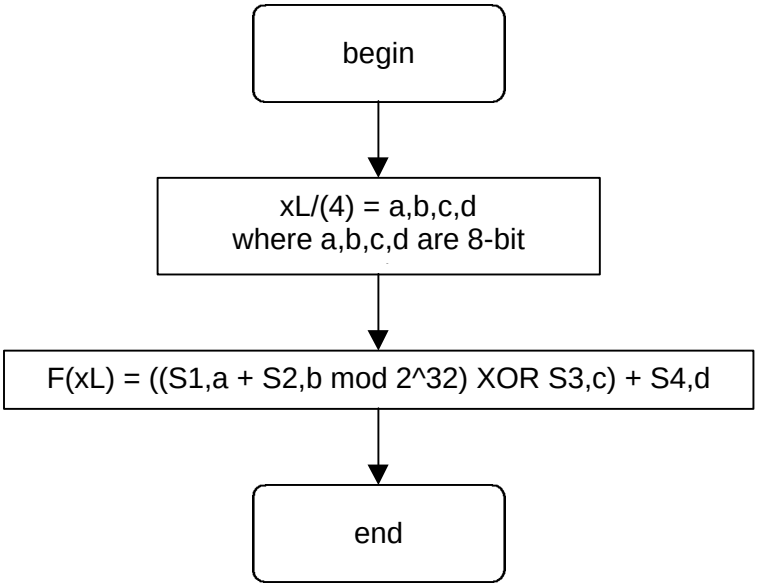
The **blowfish.vhd** component basically starts and ends the encryption process. Upon sending a signal to the ethernet interface, it begins to read in data. This data comes in the form of two 32-bit blocks XL and XR. Once all of the data is read in, it sends a signal (*data_ready_to*) low. Now it is time to actually encrypt the data that has been read in.

The **bf_enc.vhd** component is responsible for encrypting the data. In order to do this, a 'for' loop must be created for values of p-array 0 to 15. In VHDL, we implemented this via a counter (**bfcounter.vhd**). The counter basically counts up for encryption and down for decryption insofar as the p-array is concerned. It does, however, also count the s-boxes in order to implement the function that is referred to by the following flow diagrams:

Note: the original 64-bit data will be represented with and x and the P-array will be denoted Pi.

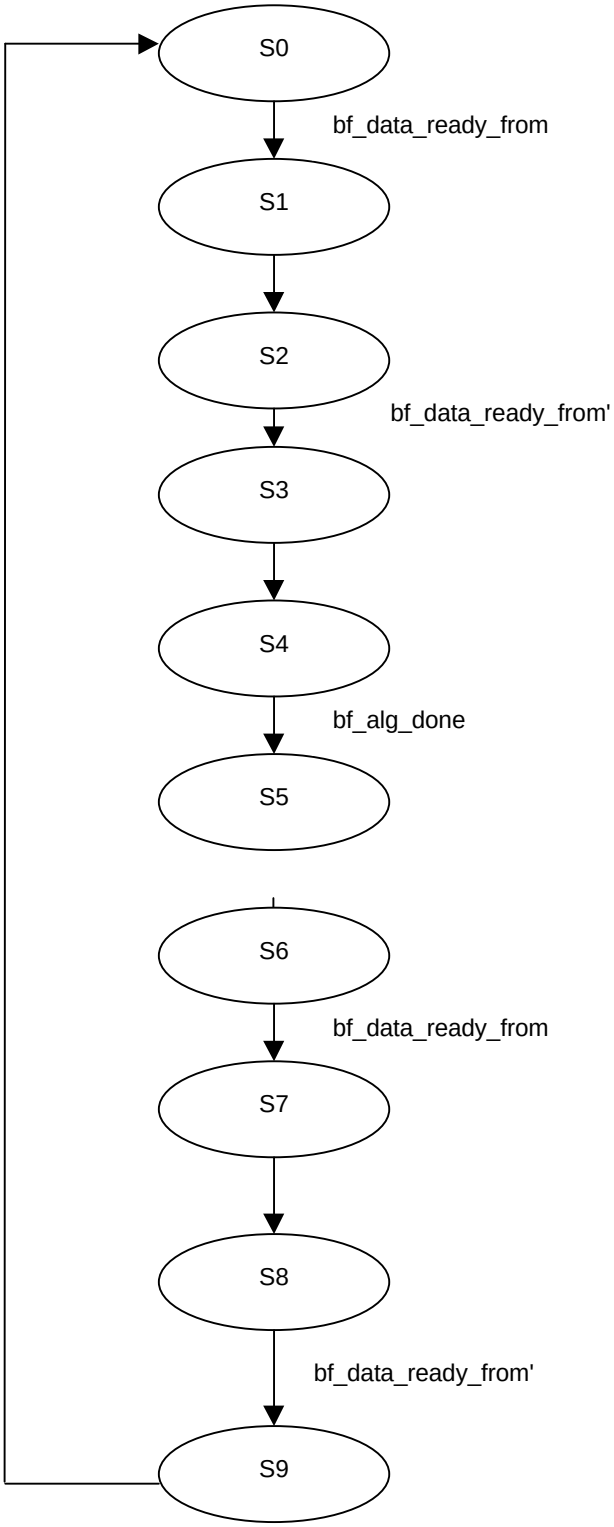


Function F:




The VHDL code for the Blowfish algorithm cannot be included for the web copy of the document. For further information on Canadian export laws on cryptography see http://insight.mcmaster.ca/org/efc/pages/doc/crypto-export.html.

It is possible to analyze the entire encryption unit by analyzing the simulation of the **blowfish.vhd** component. This is because, that component calls the **bf_enc.vhd** and **bfcounter.vhd** components (i.e. the signals under analysis in those two components are included in it). The state diagram for **blowfish.vhd** is depicted below.



In order to show the states at work without having to go to the code, the states are included here in tabular format. This will ameliorate understanding the simulation waveforms found below. Furthermore, it will help to gain an appreciation of the role that **blowfish.vhd** plays in the encryption process.

| ***State*** | ***Description*** | ***Next State*** |
|---|---|---|
| S0 | Goes to S1 if bf_data_ready_from = '1' else wait in S0 | S1 |
| S1 | Set XL_in <= bf_data_in<br>Set bf_data_ready_to <= '1' | S2 |
| S2 | Goes to S3 if bf_data_ready_from = '0' else wait in S2 | S3 |
| S3 | Set XR_in <= bf_data_in<br>Set start <= '1'<br>Set bf_data_ready_to <= '0' | S4 |
| S4 | Goes to S5 if bf_alg_done = '1' else wait in S4 | S5 |
| S5 | Set bf_data_out <= XL_out<br>Set bf_data_ready_to <= '1' | S6 |
| S6 | Goes to S7 if bf_data_from = '1' else wait in S6 | S7 |
| S7 | Set bf_data_out <= XR_out<br>Set bf_data_ready_to <= '0' | S8 |
| S8 | Goes to S9 if bf_data_ready_from = '0' else wait in S8 | S9 |
| S9 | Set bf_done_cycle <= '1' | S0 |

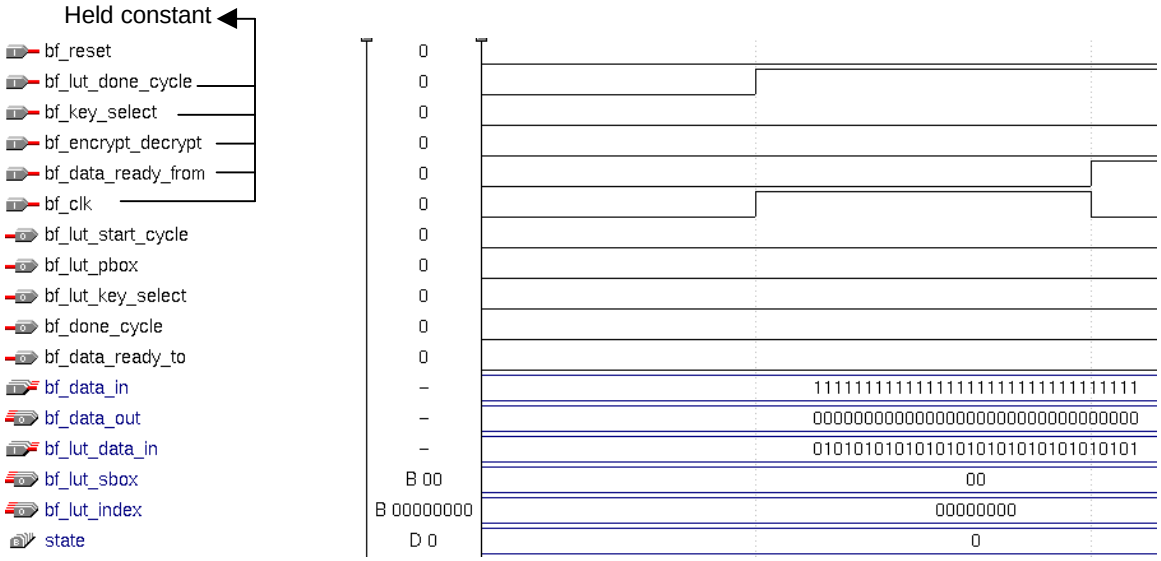
**Design Rationale**

The **blowfish.vhd** encryption module simulates successfully, on its own. However, it does so rather slowly and still takes up quite a bit of the chip. While the cell count is down since its initial coding — previously at 783 of 1152 cells and now at 634 of 1152 cells (or 55% of the chip), it is quite obvious that it will not fit in the 20000 gate chip once all of the components are fully integrated. In order to alleviate this concern, we decided to implement a smaller (albeit less secure) encryption algorithm for use with the Altera board. This algorithm simply 'XORed' the incoming data, basically leaving the encrypted output as the inverse of the input. When compared to the Blowfish encryption algorithm, the amount of space required to implement this simplified encryption technique is negligible. It should also be noted that this unit is fully interchangeable with the Blowfish unit to ease further implementation. Although this method is not nearly as secure as the Blowfish method, it allows us to implement the design onto a 20000-gate chip. The design details will be discussed later.

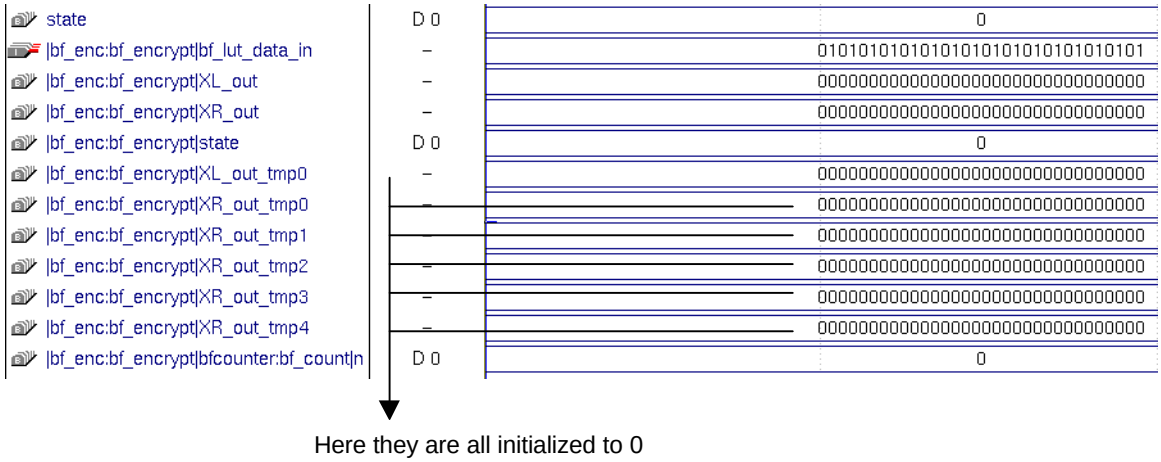
**Simulation**

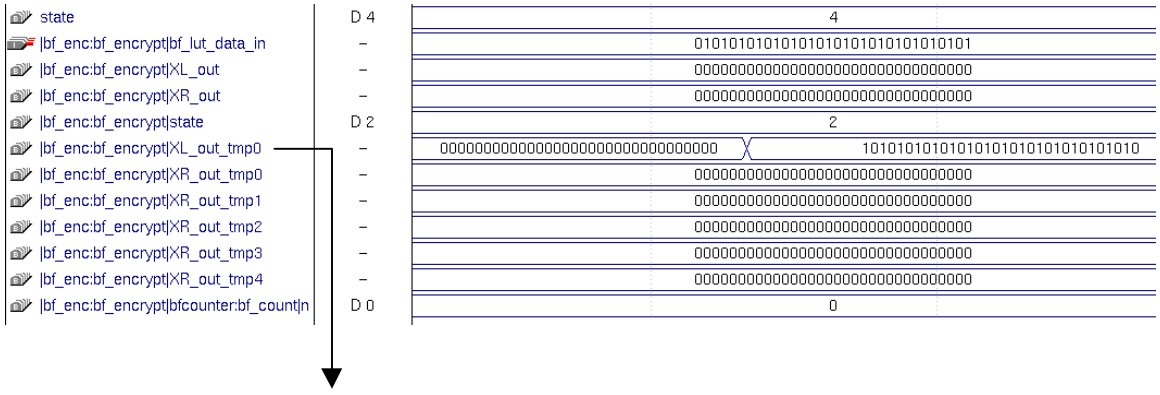
The simulation for the Blowfish algorithm module was successful. Since it has been shown that the **blowfish.vhd** component is responsible for the encryption process, inasmuch as combining the three separate entities (**blowfish.vhd**, **bf_enc.vhd**, and **bfcounter.vhd**), it is possible to analyze the resulting simulation waveforms. Having said that, only the waveform for **blowfish.vhd** will be analyzed as all of the signals under consideration are displayed in that waveform as well as their own. The resulting simulations are displayed below.

Analysis of this unit begins with the inputs as shown. Note that the *bf_data_in* and *bf_lut_data_in* are completely arbitrary values.

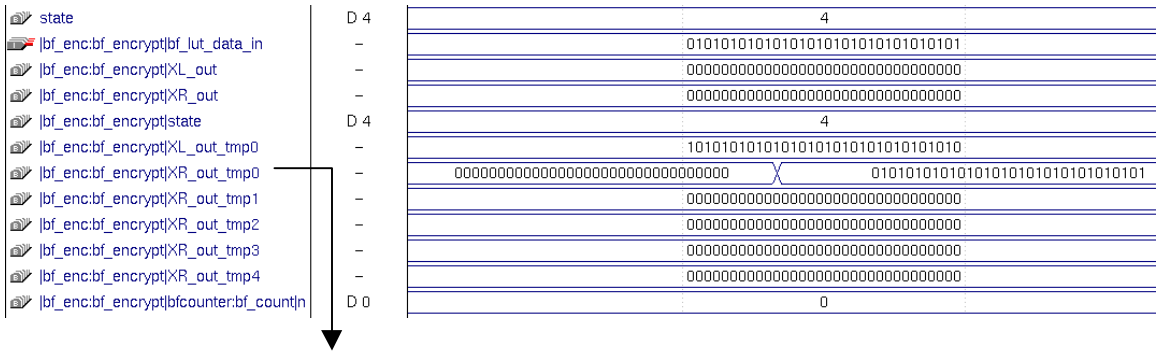



The **bf_enc.vhd** component is required when the **blowfish.vhd** component is ready for encryption. Here, the first step is to separate the 64-bit block into two 32-bit blocks *XL_in* and *XR_in*. In order to implement this algorithm in VHDL, a number of temp values were needed. The evolution of these values (*XL_out_tmp0* and *XR_out_tmp0* to *XR_out_tmp4*) is shown over the next several waveforms.

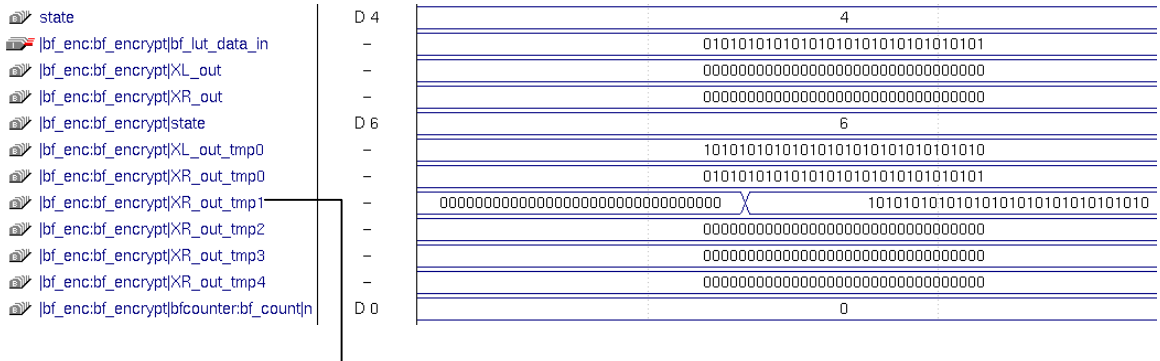




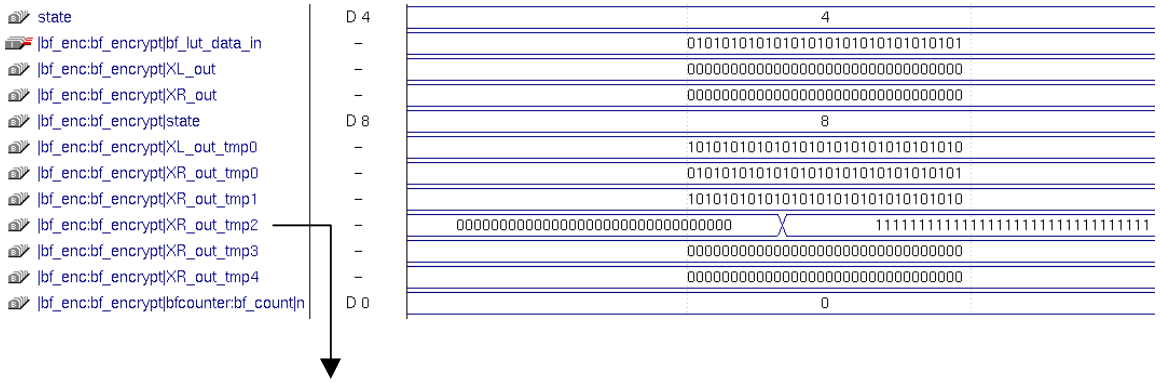

XL_out_tmp0 <= XL_in xor bf_lut_data_in



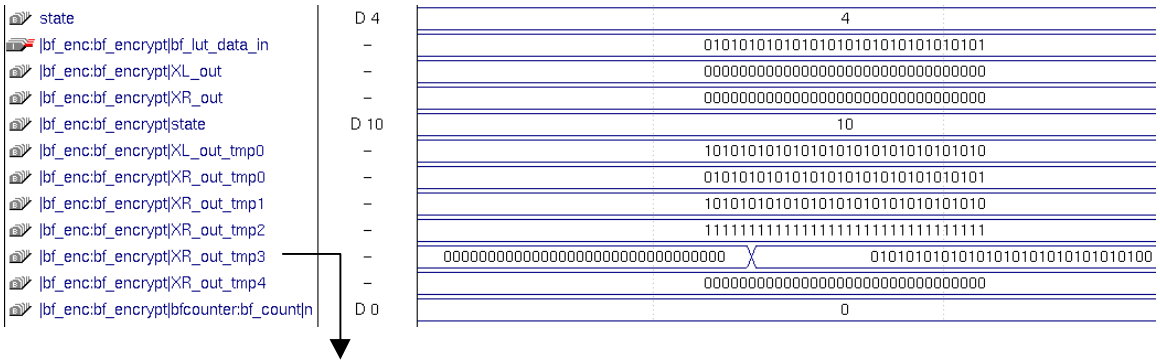

XR_out_tmp0 <= bf_lut_data_in



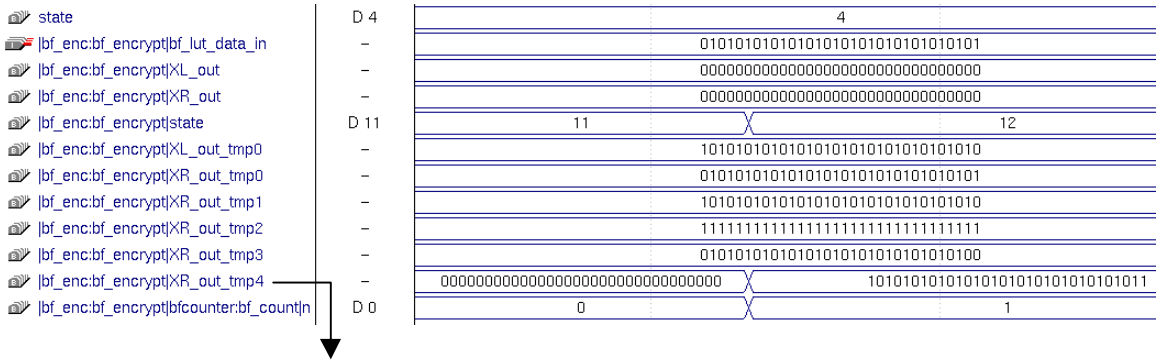

XR_out_tmp1 <= XR_out_tmp0 + bf_lut_data_in




XR_out_tmp2 <= XR_out_tmp1 xor bf_lut_data_in



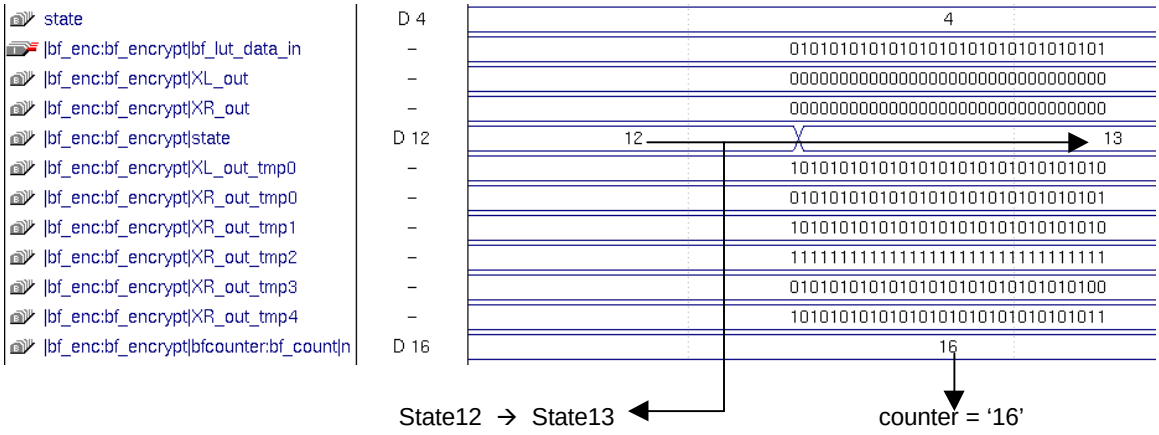

XR_out_tmp3 <= XR_out_tmp2 + bf_lut_data_in



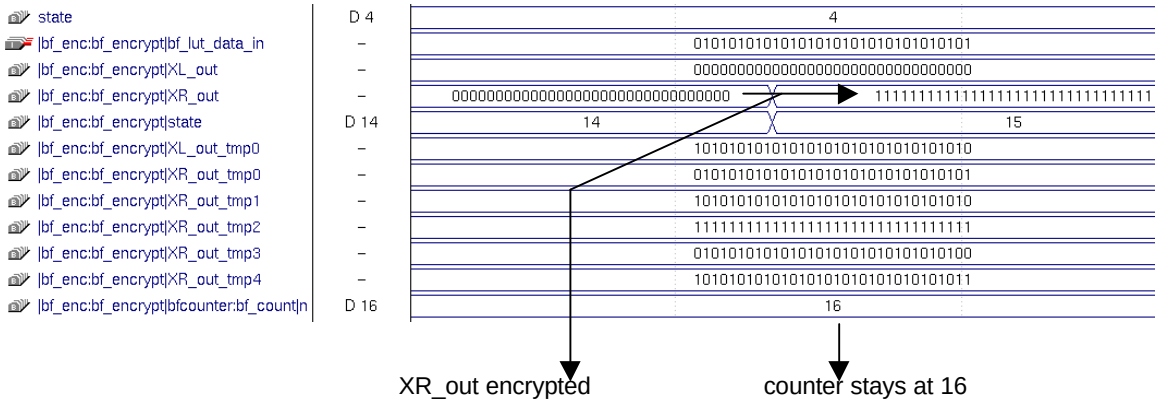

XR_out_tmp4 <= XR_out_tmp3 xor XR_in

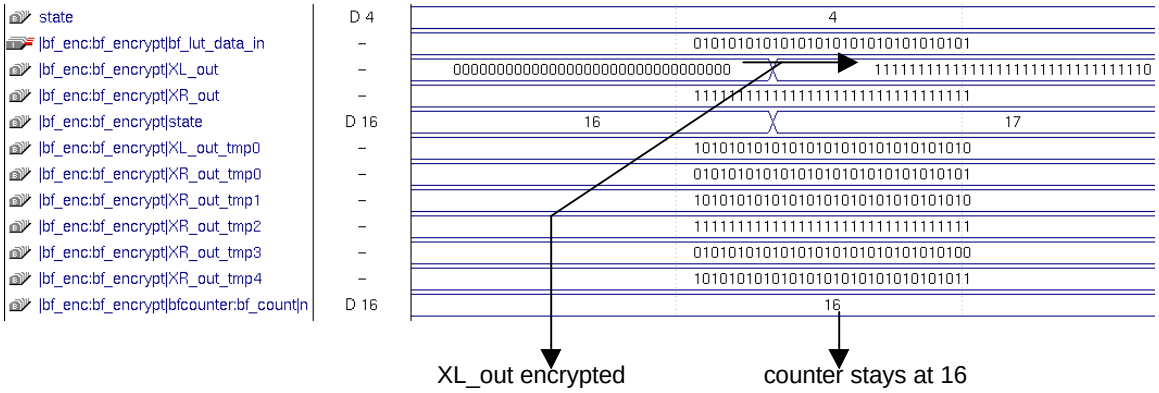
The previous six snapshots of the waveform for the encryption process have shown the first of sixteen executions of the ‘for’ loop explained earlier (and shown in block diagram

format). The remaining executions of this 'for' loop follow along in the same manner as the first and, thus, will not be displayed here. However, in order to show that the loop executes as it should (in correlation to our code), the next waveform shows a snapshot of the **bf_enc.vhd** component going from state 12 to state 13. This only happens when the counter has reached 16 — the number of iterations of the 'for' loop.

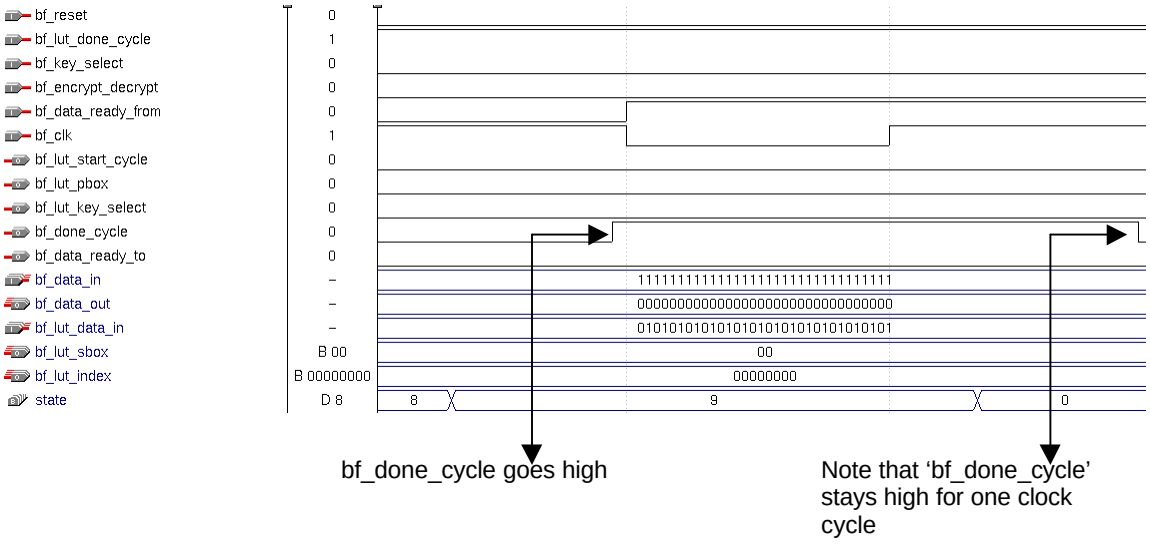



Once the counter has reached 16 and the **bf_enc.vhd** component finally gets out of its loop, it is able to finish encrypting the inputs. That said, the remaining four states (states 13 to 17) simply calculate the output values for the two 32-bit inputs (i.e. *XL_out* and *XR_out*).

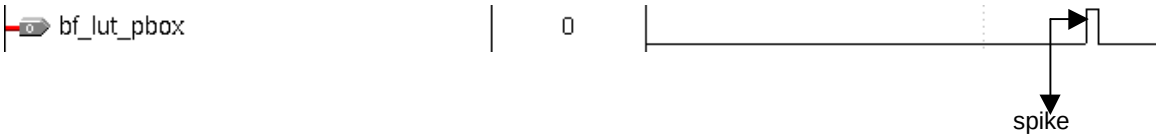





Now that both *XL_out* and *XR_out* have been calculated, the **bf_enc.vhd** component sends a signal *bf_alg_done* to the **blowfish.vhd** component. This tells the **blowfish.vhd** component that encryption is done and it can send data out in the form of its signal *bf_data_out*. Once the **blowfish.vhd** component has sent both 32-bit blocks of data for *XL_out* and *XR_out* (through *bf_data_out*), it asserts another signal — *bf_done_cycle* high.



The simulation runs along relatively smoothly; however, there are a couple of spikes on the signal *bf_lut_pbox*. This glitch does not occur on the rising edge of the clock and, therefore, does not affect a look-up from the look-up table. Furthermore, it occurs at state 12 and has no impact on the result.




## *LUT Access*

### Overview

The Lookup Table Access module arbitrates access to the Blowfish look-up-tables stored in external memory. It arbitrates between three sources: both the Blowfish encryption module and the LUT generator need read access, and the LUT generator needs write access.

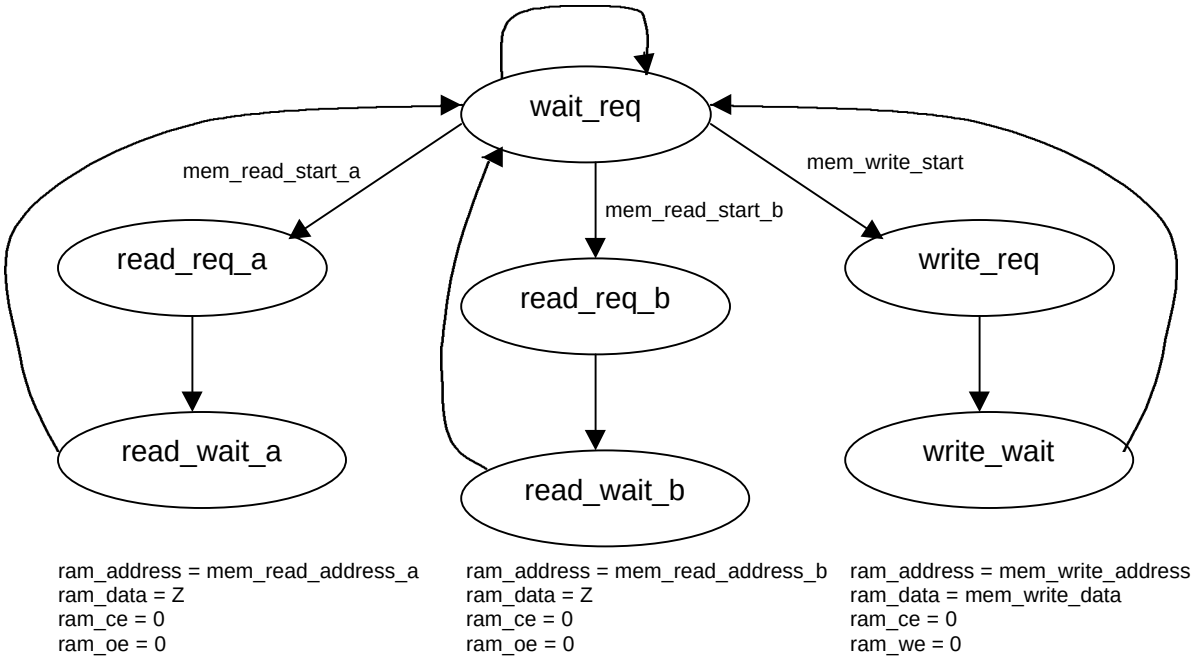
### Interface Definition

Each module that needs to use the LUT Access module is connected with one set of the signals …_*a*, …_*b*, or …_*c*. To request a read, the *mem_read_start_X* line is asserted with the correct address on the corresponding address bus. When the *mem_read_done_X* is high, the memory value is available on the *mem_read_data_X* bus. Similarly for a write, the operation is requested by asserting mem_write_req with the correct address on the *mem_write_address* bus. When the *mem_write_done* signal is asserted, the write operation has completed.

### Design Details

The memory used is asynchronous SRAM (Cypress CY7C199), with an access time of 12ns. Synchronous SRAM would probably have been easier to work with, but the smallest chips available are 1Mbit, which is far beyond the minimum 32Kbit required for the LUT. Since the asynchronous memory responds to falling edges on the control signals, it is necessary to latch all of these signals to prevent glitches; also, the address (and during a write, the data) signals must remain asserted while the chip is enabled, so they must also be latched.

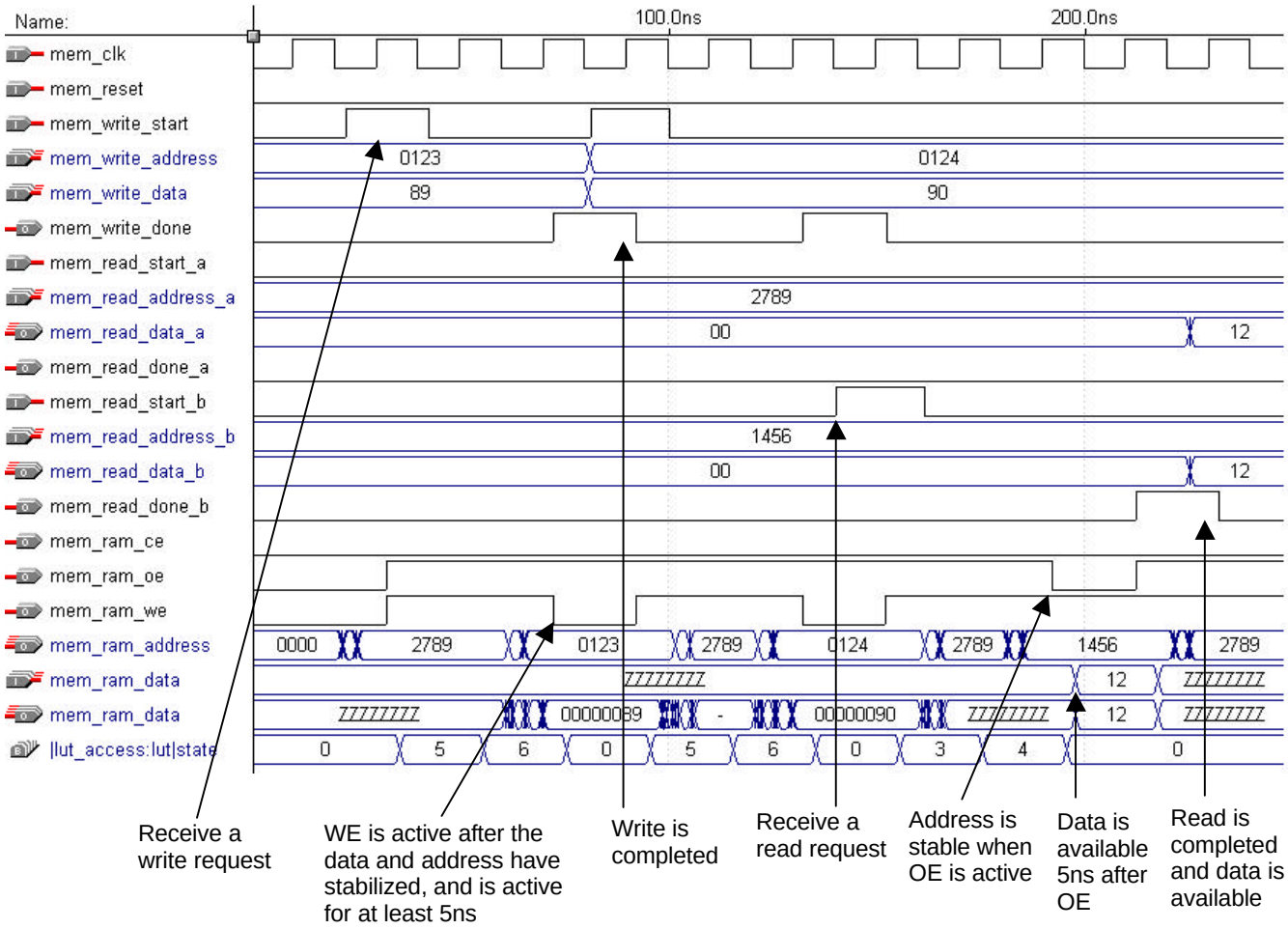
The state machine is shown below. It waits for one of three consumers to request a memory access. When the access is finished, the done signal for that consumer is asserted. Thus, requests from other consumers will block until the current request is completed. Note that there is only one state that chooses which request to acknowledge, which means that the first consumer checked will always get priority. Two more "wait for request" states are required that have different priorities, which would allow for a round-robin type of scheduling.




The current implementation requires 4 clock cycles to complete a read request. Blowfish requires 16 x 4 + 18 memory access, or 18 x 80ns = 6560ns (of course, Blowfish also requires time for its control and logic). This does not meet the timing requirements imposed by the data rate of the 10Mb/s ethernet: 64 bits are received every 6400ns. Further work would be required to reduce this to three clock cycles, which should allow Blowfish to complete in less than 6400ns. Since we did not implement Blowfish, optimizing this code to the required speed was not considered a priority.

**Simulation**

A sample of a series of requests is shown in the waveform on the following page.

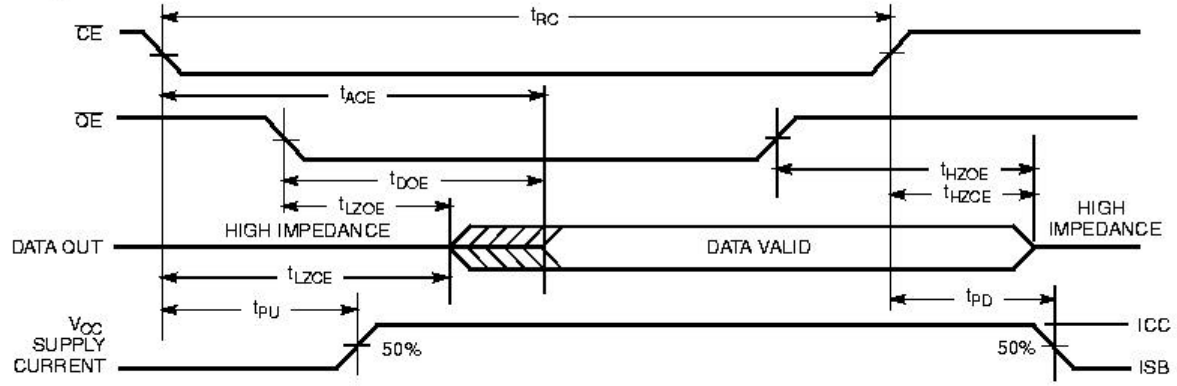



Included below are the read and write cycle timing diagrams (and the table of timing values) from the CY7C199 data sheet.

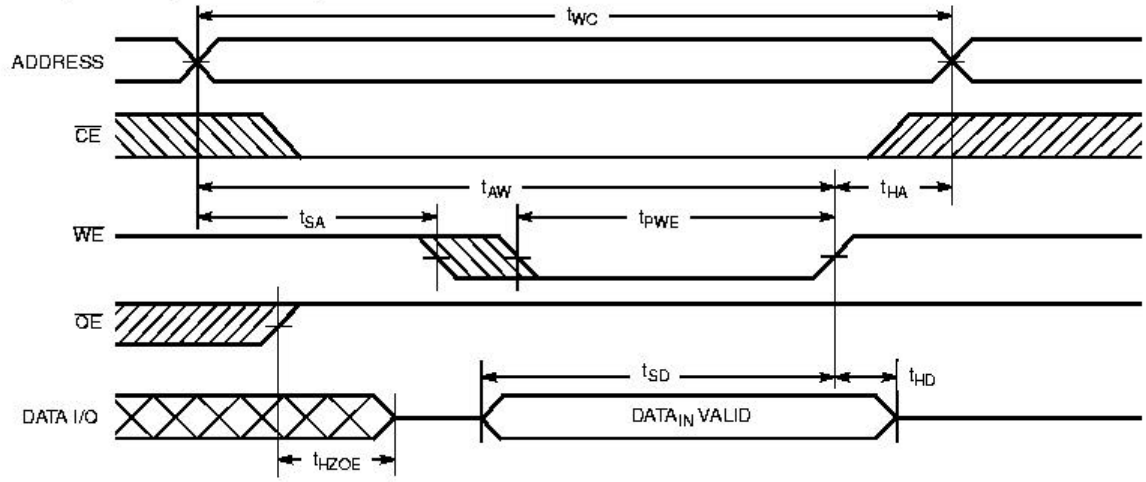
| Param | Description | 7C199-12 Min. | 7C199-12 Max. | Unit |
|---|---|---|---|---|
| **READ CYCLE** | | | | |
| $t_{RC}$ | Read Cycle Time | 12 | | ns |
| $t_{AA}$ | Address to Data Valid | | 12 | ns |
| $t_{OHA}$ | Data Hold from Address Change | 3 | | ns |
| $t_{ACE}$ | $\overline{CE}$ LOW to Data Valid | | 12 | ns |
| $t_{DOE}$ | $\overline{OE}$ LOW to Data Valid | | 5 | ns |
| $t_{LZOE}$ | $\overline{OE}$ LOW to Low Z[8] | 0 | | ns |
| $t_{HZOE}$ | $\overline{OE}$ HIGH to High Z[8, 9] | | 5 | ns |
| $t_{LZCE}$ | $\overline{CE}$ LOW to Low Z[8] | 3 | | ns |
| $t_{HZCE}$ | $\overline{CE}$ HIGH to High Z[8,9] | | 5 | ns |
| $t_{PU}$ | $\overline{CE}$ LOW to Power-Up | 0 | | ns |
| $t_{PD}$ | $\overline{CE}$ HIGH to Power-Down | | 12 | ns |

| Param | Description | 7C199-12 Min. | 7C199-12 Max. | Unit |
|---|---|---|---|---|
| **WRITE CYCLE** | | | | |
| $t_{WC}$ | Write Cycle Time | 12 | | ns |
| $t_{SCE}$ | $\overline{CE}$ LOW to Write End | 9 | | ns |
| $t_{AW}$ | Address Set-Up to Write End | 9 | | ns |
| $t_{HA}$ | Address Hold from Write End | 0 | | ns |
| $t_{SA}$ | Address Set-Up to Write Start | 0 | | ns |
| $t_{PWE}$ | $\overline{WE}$ Pulse Width | 8 | | ns |
| $t_{SD}$ | Data Set-Up to Write End | 8 | | ns |
| $t_{HD}$ | Data Hold from Write End | 0 | | ns |
| $t_{HZWE}$ | $\overline{WE}$ LOW to High Z[9] | | 7 | ns |
| $t_{LZWE}$ | $\overline{WE}$ HIGH to Low Z[8] | 3 | | ns |

**Read Cycle No. 2**




**Write Cycle No. 1 (WE Controlled)**



## *Blowfish Controller*

### Overview

Ideally, an eSAFE device would have two separate and independent Blowfish encryptor blocks, respectively dedicated to encrypting outgoing data and decrypting incoming data. However, due to the large number of logic cells required for a Blowfish encryptor block, it is conceivable that some FPGA devices may not have a sufficient logic cell count to accommodate two separate units. To guard against this problem, the Blowfish controller is introduced to control and mediate access and communications between the Blowfish encryptor and the client components requesting its use. The entity definition for the Blowfish controller is in **bf_control**.

### Interface Definition

The Blowfish controller component waits for one of the three *bfctl_req* lines to be asserted. When one or more of these lines is asserted, the Blowfish controller chooses to service one of them, and asserts the corresponding *bfctl_ack* line. Internally, the **bf_control** unit sets the *bfctl_mux_select* line to the correct value. This multiplexes the signals internally; every incoming and outgoing line of communication between the serviced component and the Blowfish block become connected to each other. The **bf_control** unit will not accept any more connections until the *bfctl_done_cycle* line is asserted by the Blowfish block, signifying that the current request is over. Furthermore, the Blowfish controller will also wait for the request line to be negated. This prevents the serviced unit from immediately requesting another service, thus starving the other clients from using the Blowfish block.

### Design Details

The design of the **bf_control** unit is actually quite simple. As described in the Interface Definition, the unit merely waits for and then accepts a request for service, and acknowledges that request. Internally, the input/output lines are multiplexed in such a fashion as to allow the client component and the Blowfish unit to communicate directly with each other. No internal buffering is performed on these multiplexed lines; thus, there is only a small latency added due to the logic gates in the multiplexers that each signal is routed through. When synthesized, the chip utilization is as follows:

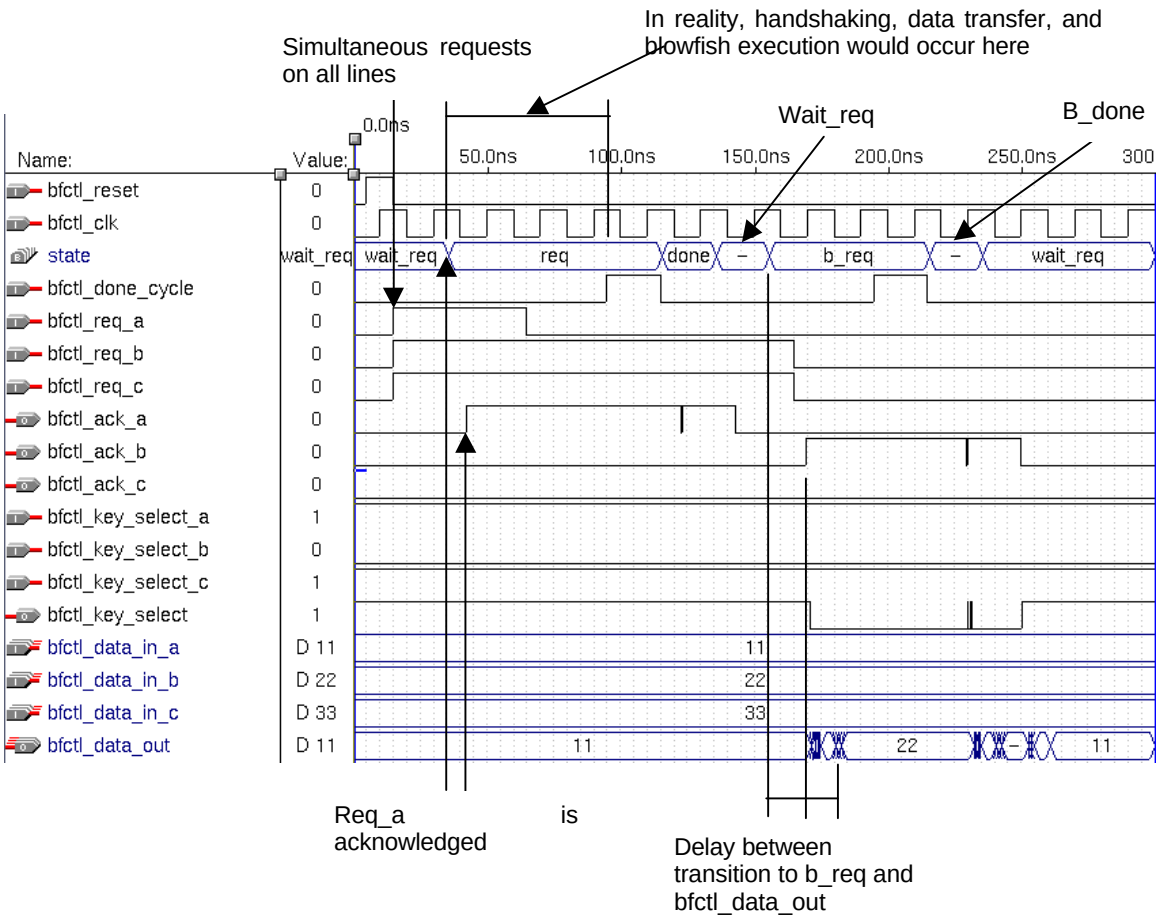
| | |
|---|---|
| Total logic cells required: | 34 |
| Total flipflops required: | 0 |
| Total packed registers required: | 0 |

Since the design only requires one of these units, it can be concluded that this unit does not require a significant percentage of the logic cells, and does not have to be optimized further. However, note that it has, in total, 8 32-bit data lines, which may cause a wiring problem. However, since it is small, we are confident that it will be possible to find a suitable location for this unit on the chip. Furthermore, it can be modified to use 8-bit buses instead, although this will require the addition of buffer stages to accumulate 32 big numbers that the Blowfish unit expects to process.

Testing the **bf_control** unit is quite easy, since it only pays attention to a handful of the many inputs and outputs. All the rest are merely routed via the *bfctl_mux_select* signal. Thus, the *a* group of lines was set to one and the *b* set of lines to zero. The *bfctl_req* line was asserted, and the unit correctly generated the *bfctl_ack_a* and ignored requests from the *bfctl_req_c* line. Then, the unit waited for both *bfctl_done_cycle* to be asserted and *bfctl_req_a* to be negated. Next, when *bfctl_req_b* was asserted, it could be seen that the *bfctl_mux_select* line correctly chose the output to the Blowfish block to be the inputs from the *b* lines (this transition could be seen because all of the *b* inputs were different than the *a* inputs. Note that the *ack* signals were not originally present. However, they were deemed necessary, since the *bf_control* multiplexors could switch over the single lines more quickly than the data bus lines. Thus, it would be possible for the *bfctl_data_ready_to_X* lines to signal that the data was valid before the *bfctl_data_in_X* were fully selected. Thus, the acknowledgements are needed so that the requesting entities can assert their data ready lines after the multiplex switching has been done.

**Simulation and Verification**

The Blowfish control unit arbitrates access to the Blowfish encryption unit between three entities that share an identical interface to the Blowfish controller. These entities, for our current implementation, will consist of two progressive encryption blocks and the lookup table generator. The *bf_control* unit can handle multiple simultaneous requests, as is shown in the simulation below. When a request line is asserted by one of the user entities, the *bf_control* unit chooses one to be granted access, and asserts that unit's *bfctl_ack* line. Internally, this switches over all of the *bf_control* to/from blowfish lines to connect to this user device. Upon receipt of the acknowledgement, the user entity is then directly connected to the Blowfish unit, and modulates the data and control lines to communicate data between each other. Note that there is a significant delay between the transition to the *b_req* state (in which unit *b* is granted control of Blowfish) and the time that the *bfctl_data_in_b* is mapped to *bfctl_data_out*. It is the responsibility of the user unit and the Blowfish unit to account for this. However, if the respective units do not change their *data_to* and *data_from* control lines until an acknowledgement of service is received, then a sufficient delay will be introduced that will allow all lines to be correctly mapped. It is important to note, in the simulation below, that the request states will be much longer than shown. During these states, the serviced unit and the Blowfish unit will be using control lines to load and store data. In addition, the Blowfish unit will be encrypting the data sent by the user entity. Both of these operations will take much longer than shown.




## *Lookup Table Generator*

### Overview

The lookup table generator component is responsible for generating the P-array and S-boxes for a given 64-bit key and storing them in memory. It requires access to ROM (to initialize the P-array and S-boxes) and to RAM (to store the lookup table for use by the Blowfish encryptor block. It also requires access to the Blowfish encryptor block itself (as the algorithm uses Blowfish with the partially generated table to create new entries in the table).

### Interface Definition

The operation of this unit is quite simple. When *lut_key_ready* is asserted, the value on *lut_key_value* is read in for the key. In the two-table case, **lutgen** keeps track internally of which table has the current value, so that it will start table generation in the unused table. Using the defined interfaces and procedures for **bf_control**, **blowfish**, and **lut_access**, the **lutgen** component then follows the lookup table algorithm, and switches over the eSAFE encryptor to use the new table.

**Design Details**

The lookup table generation algorithm, in pseudocode, is as follows:

```
1.  Initialize the P-array and S-boxes from ROM with a fixed
    value common to all eSAFE devices
2.  XOR P1 with the first 32 bits of the 64 bit key, and P2 with
    the second 32 bits of the key
3.  Repeat this for the entire P-array
4.  Encrypt the all-zero string with Blowfish using this table
    and save the 64 bit result in a register.
5.  Set P1 to be the first 32 bits of the result, and P2 to be
    the second 32 bits of the result.
6.  Using the result of the previous Blowfish encryption, repeat
    this process using the latest version of the table, and fill
    up two values from the P-array or S-boxes with the result
    every time.  Repeat until all entries have been updated.
    This will take 521 iterations of Blowfish in total.
```

Since the ROM table is 4kB and Blowfish encryption is performed 521 times, it can clearly be seen that the lookup table requires many memory accesses (both reads and writes) and a significant length of time in order to generate a table that is ready for data encryption. It is therefore important to allow ethernet encryption to progress while lookup table generation is simultaneously being done. In order to accomplish this, the **bf_control** and **lut_access** components were created. The **bf_control** block allows the lookup table generator to share the Blowfish block with both progressive encryptors, whereas the **lut_access** block arbitrates read and write access to the lookup tables in RAM between **blowfish** and **lutgen** (the lookup table generator). Allowing for at least two tables to be stored in RAM would enable the **blowfish** block to use one table (already generated on startup) while the **lutgen** component simultaneously generates the new lookup table in another RAM location. Thus, this design completely eliminates any time constraints from the design of the lookup table generator. This is true because the table generation will proceed cooperatively with the current encryption and there is no maximum time delay specified between new key input and valid table ready.

Actual implementation of this straightforward algorithm in hardware is decidedly non-trivial when attempting to optimize for a limited space. For example, the entire lookup table is scanned through during intialization of the table from ROM and during actual table generation, but taking advantage of this similarity was difficult. However, implementation was abandoned on this component once it was realized that the Blowfish encryptor could not be included in the final project due to space and timing constraints. This was done because successful simulation of the Blowfish component for correctness would not require this component. Also, there was a possiblity that all functional blocks could be optimized to fit on the chip provided. However, it was not probable that the optimization would be enough to accommodate **lutgen** even if **blowfish** managed to fit. In this case, we planned to fall back to ROM-based pregenerated tables, and **lutgen** would still not be required.

## Conclusion

Designing the eSAFE project introduced many challenges. Currently, it is not full functional, as previously discussed. Also discussed throughout the document are possible reasons for this. Nevertheless, although the physical implementation of the device is inoperable, it should be noted that simulations of the wholly integrated device are successful. As well, simulations are successful for the ethernet receiver, transmitter, CRC generator, Blowfish encryptor, and Blowfish arbitrator.

Even though successful simulations do not constitute a successful project, insofar as the physical implementation (as we found out), the eSAFE project was far from unsuccessful. The project chosen was quite complex. This point is especially obvious considering that a much simplified encryption algorithm was required due to size constraints. Although the Blowfish algorithm could not be implemented in the Altera 10K20 chip, the components pertinent to the Blowfish algorithm have been included for completeness and future endeavors.