6.111 Final Project Report: Encrypted Communications over Ethernet

Hyo Won Kim, Mark Theng

2018

Contents

1 Introduction **3**

2 Overview **4**
- 2.1 External Interfaces 4
- 2.2 Block Diagrams 4
- 2.3 Clock, Memory and Latency Constraints 6

3 Data Flow Interface **6**
- 3.1 "Forward Control" Interfaces 7
- 3.2 "Backward Control" Interfaces 7
- 3.3 "No Control" Interfaces 9

4 UART Drivers (Mark) **9**

5 Networking (Mark) **10**
- 5.1 The Network Stack 10
- 5.2 The FGP Protocol 10
- 5.3 The FFCP Protocol 11
- 5.4 Implementation Overview 12
- 5.5 The RMII Driver 13
- 5.6 Ethernet Implementation 14
- 5.7 FFCP Implementation 15

6 The AES Cryptosystem (Ashley) **16**
- 6.1 The AES Algorithm 16
- 6.2 Encryption/Decryption Blocks 17
- 6.3 Implementation Overview 18
- 6.4 AES Encryption/Decryption 18
  - 6.4.1 SubBytes 18
  - 6.4.2 ShiftRows 19
  - 6.4.3 MixColumns 19

6.4.4 AddRoundKey 20
6.5 Round Key Generation 20
6.6 CBC 21

7 Graphics (Mark) 21

8 Development Process 22

9 Conclusion 24
9.1 Lessons Learned 24
9.2 Possible Extensions 26

A Verilog Code 26
A.1 IP Cores Used 26
A.2 Headers (Includes) 27
A.3 Utilities 28
A.4 Device Drivers 42
A.5 Networking 55
A.6 AES 74
A.7 Graphics 96
A.8 Top Level 97

B Simulation Code 113

C Python Code 115
C.1 Libraries 115
C.2 COE Generation 119
C.3 Communication with FPGA 120

1 Introduction

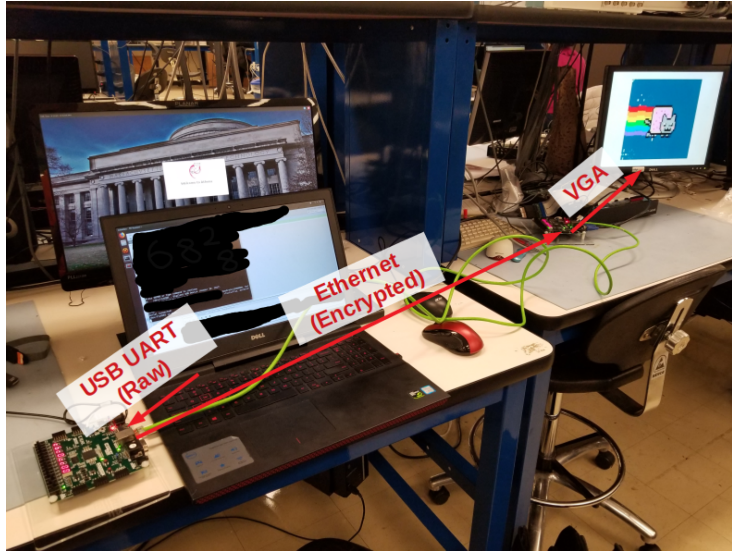

Figure 1: Physical setup and flow of data in our system.

Today's world is defined by constant surveillance, by the government and by the various companies that seek access to our data. Methods for intercepting communications are easily accessible to any motivated entity through exploit kits commonly sold on the dark net. To this crisis of our age there remains but one solution: AES-encrypted Ethernet with FPGAs.

Our system securely[1] and robustly transmits messages from one FPGA to another, complete with end-to-end flow control. While the type of message transmitted is arbitrary, we demonstrate our system by streaming video data: a video is sent frame by frame to an FPGA via the USB UART interface, encrypted, encoded into Ethernet packets, transferred over an Ethernet cable to another FPGA, decrypted, and finally displayed on a VGA monitor.

Ethernet is the most common physical layer networking standard for Internet communications, so using Ethernet as the base communication medium for our cryptosystem would allow our system to communicate with a large range of devices. The Nexys 4 DDR comes with an Ethernet port behind a SMSC 10/100 LAN8720A Ethernet PHY, which exposes up to 100Mpbs Full Duplex Ethernet over RMII (Reduced Media-Independent Interface). In particular,

[1] Up to an extent – due to time constraints, our implementation of AES lacks many features that are important for a fully secure practical implementation, including, but not limited to: metadata encryption, IV exchange, HMAC authentication, side channel attack mitigations, and key negotiation. Do not use this system in production, and never implement your own crypto.

we can interface with Ethernet without worrying about the analog intricacies of digital communication. This ensures that information is transported in a reliable and noise-resistant fashion.

AES (Advanced Encryption Standard) is a secure, time-tested symmetric key encryption algorithm widely used in modern digital systems. Our project implements AES-128 in CBC (Cipher Block Chaining) mode, which ensures that no information is leaked even when the data contains repeating patterns.

2 Overview

2.1 External Interfaces

Our project only interfaces with components on the Nexys 4 DDR board.

- FTDI FT2232HQ USB-UART bridge (RS232 interface, max 12Mbps for USB 2.0 Full Speed)[2]
- SMSC 10/100 LAN8720A Ethernet PHY (RMII, max 100Mbps Full Duplex)[3]
- VGA monitor (800x600, 72Hz mode)

2.2 Block Diagrams

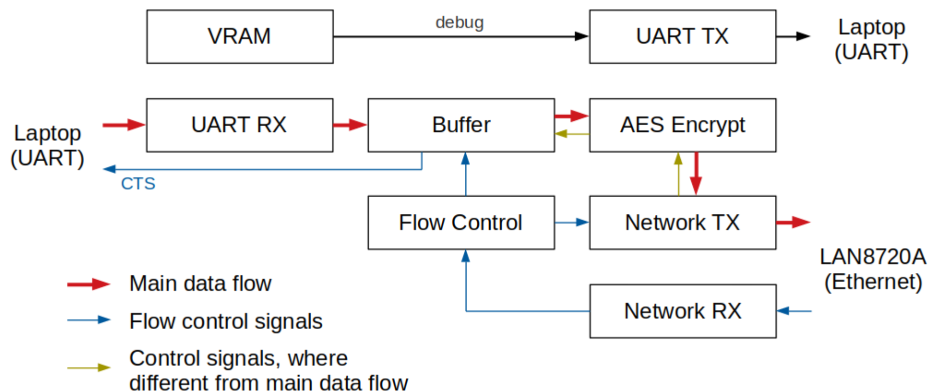

Figure 2: Flow of data in the ”transmit” configuration.

[2]Spec sheet: https://www.ftdichip.com/Support/Documents/DataSheets/ICs/DS_FT2232H.pdf

[3]Spec sheet: http://ww1.microchip.com/downloads/en/devicedoc/8720a.pdf

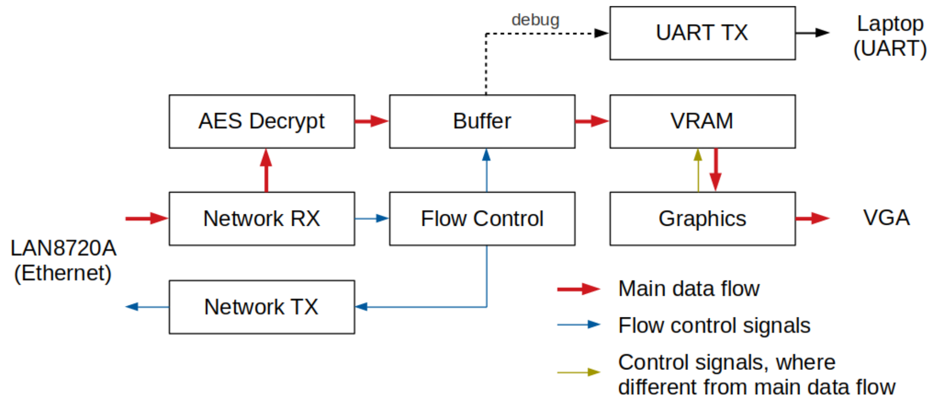

Figure 3: Flow of data in the "receive" configuration.

Our system involves two FPGAs – one on the "transmit" end, and one on the "receive" end. While their overall functionalities are different, they share many of the same components, such as the networking infrastructure required to receive and transmit packets, UART transmission and the AES module. We thus program both FPGAs with the same bitstream, and use a switch to configure each FPGA as either the "transmit" or "receive" end.

For simplicity, our block diagrams show the system in the "transmit" and "receive" configurations separately, omitting the configuration and RAM/AES multiplexing signals.

Our system is, on the fundamental level, a very long daisy chain. The daisy chain is highlighted in the block diagrams by bold red arrows. On top of this is a flow control layer, which buffers data from the laptop and ensures that all packets are processed, in order, by the "receive" end.

There is an additional complication to the daisy chain structure. If data flows from module A to module B, it is not always the case that module A can be thought of as "controlling" module B. For example, in the packet transmission pipeline, the networking module has to transmit packet headers before the payload. It is natural, then, to think of the networking module as "requesting" the payload from the AES encryption module. More information on how this works is provided in Section 3. In the block diagram, this "reversal of control" is indicated by a yellow arrow pointing in the opposite direction of a bold red arrow.

There is also a debug interface that allows us to dump the entire contents of RAM over UART, so that we can inspect it on a computer. Since, for simplicity, only one module is allowed to read from RAM in each configuration, the packet buffer dump interface was disconnected when it was integrated into the system for flow control. This is indicated by the dotted line in Fig. 3.

2.3 Clock, Memory and Latency Constraints

Most of our system uses a 50MHz clock because the 100Mbps Ethernet RMII interface requires a 50MHz clock (2 bits per clock cycle). We chose to operate VGA at 800x600 and 72Hz because the required pixel clock frequency is also 50MHz, obviating the need to cross a clock boundary. Nevertheless, it would be easy to adapt our system for different pixel clocks since the graphics module is separated from the rest of the system by the VRAM buffer, which can be used for synchronization at a clock boundary.

Since both the networking receive and transmit pipelines are fundamentally constrained by the RMII interface, all modules in the pipelines must operate at least as fast as two dibits per clock cycle (or one byte per four clock cycles, or one AES block – that is, 128 bits – per 64 clock cycles). For convenience of implementation, the AES module is further restricted to one AES block per 16 clock cycles (to match one byte per clock cycle), so that we can stream data from byte memory directly into the module without having to worry about synchronization.

There is one part of our system that uses a clock other than 50MHz – the UART interface. We operate UART at 12MBaud, the maximum rate that the Nexys 4 UART interface can handle. In order to receive and transmit at that frequency, the clock frequency should preferably be some integer multiple of 12MHz. For the UART modules, we use a 120MHz clock, along with Xilinx IP FIFO generated cores to synchronize across the clock boundary (ref. Section 4.

Our system contains three sets of BRAMs. There is a 4096-bit ROM, which is used to store constant values used by the networking stack, such as MAC addresses and EtherType values (ref. Section 5). There is a 16384-bit packet buffer RAM used for flow control. It stores 16 packets each up to 1024 bits long. Finally, there is a 16384-word RAM for 12-bit words used as video memory, which holds the 128x128 image (where each pixel is a 12-bit color) that is displayed on the screen.

The main bottleneck in our system is, surprisingly, not the Ethernet interface, but the UART interface. Data is transferred from the laptop to the transmitting FPGA at a maximum rate of 12Mbps (less, in fact, since it has to transmit start and stop bits too), which is slower than the Ethernet 100Mbps. In order for our system to be capable of transmitting video at 60Hz, we only had time to transmit about 128x128 raw pixels per frame. Thus, there was no need to look into memory options beyond BRAM.

3 Data Flow Interface

Since our system is, at its core, a very long daisy chain, a consistent data flow interface was a very important factor in its development. This section describes the central abstractions in the data flow interfaces used throughout our system.

Most module interfaces in our system contain the following signals: ”inclk”, ”in”, ”outclk”, ”out”, and, at times, ”readclk”, ”in_done” and ”done”. **The**

signals labelled ”clk” are not clocks, and could have been better named ”en” for ”enable”. If it helps, only variables *starting with* ”clk” are actually clocks. This phenomenon is an unfortunate artifact of history, and is kept only for the sake of consistency.

3.1 ”Forward Control” Interfaces

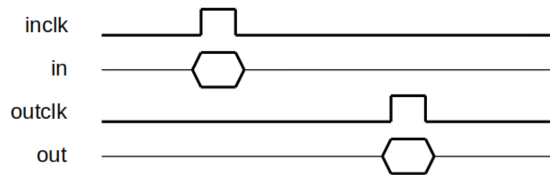

Figure 4: Waveform diagram for a ”forward control” interface.

In general, our system uses two classes of data flow interfaces. The ”forward control” interface is the most natural one. An example of this is the AES module. Data is presented on ”in”, and ”inclk” is asserted for a single clock cycle to indicate that the data on ”in” is valid. Some time later (for example, when the AES module has finished encrypting the block), or even on the same clock cycle, data is presented on ”out”, and ”outclk” is asserted to indicate that the data on ”out” is valid. In some cases, more or less data could be presented on ”outclk/out” than is received on ”inclk/in”, such as the module that unpacks bytes into dibits.

”in_done” and ”done” signals, when present, are used to signify the end of a data stream. ”in_done”, an input, signifies that the ”inclk/in” stream is done, while ”done”, an output, signifies that the ”outclk/out” stream is done. Except in some special cases, ”in_done” (or ”done”) is only valid when ”inclk” (or ”outclk”) is asserted.

This makes daisy-chaining ”forward control” interfaces easy – ”outclk” is fed into ”inclk”, ”out” is fed into ”in”, and ”done” is fed into ”in_done”.

In most cases, a module cannot accept more data on ”inclk/in” until it has presented data on ”outclk/out” – the AES module, for example, cannot process a new block until it has finished processing the current one. There are some cases, however, when this is possible, such as the delay module.

3.2 ”Backward Control” Interfaces

Sometimes, the ”forward control” interface is not sufficient. For example, in the packet transmission pipeline, the networking module must transmit packet headers before transmitting the payload. Here, timing is critical – the networking module must transmit a continuous stream of dibits for a valid Ethernet frame, so the payload must arrive just when the headers have been transmitted.

Trying to arrange for this to happen with a central coordinator breaks modularity, and would result in very complicated code when more layers are added to the network stack. The solution to this is the ”backward control” interface.

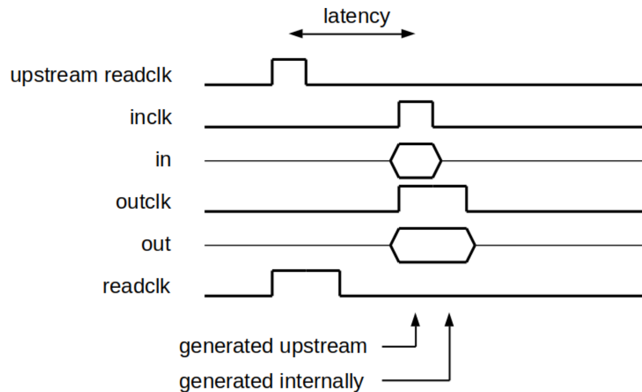

Figure 5: Waveform diagram for a ”backward control” interface. ”readclk” is an input from the downstream module, and ”upstream readclk” is an output to the upstream module.

Suppose we have a chain of modules $A \to B \to C$, where the arrows indicate the direction of data flow. If B has a ”backward control” interface, then it is expected to operate in the following manner: When C is ready for data, it asserts ”readclk” to request data from B. Some time later, B presents the data on ”outclk/out”.

For modules in the packet transmission pipeline, there is an additional latency restriction – the time between when ”readclk” is asserted and when ”outclk” is asserted must always be exactly two clock cycles. When data is ready before that, such as when generated within B itself, a delay buffer must be used. This simplifies the interface, ensuring that the Ethernet module receives data at a consistent rate and allowing it to convert the data into a continuous dibit stream.

The data that B gives to C can come from either B itself, or from A. When the data comes from A, due to the latency restriction, B must assert A's ”readclk” at the same clock cycle when its ”readclk” is asserted, and pass data from A's ”outclk/out” directly to its ”outclk/out” when it arrives. This makes B a Mealy machine by necessity.

As Fig. 5 suggests, modules are usually designed so that C does not need to wait for data from B between consecutive ”readclk” assertions, though this convenience is often not necessary.

Sometimes, it is necessary to convert a ”forward control” module into a ”backward control” module or vice versa. For example, the AES module is used in the packet transmission pipeline, and must be converted into a ”backward” control module. To convert a ”backward control” module to a ”forward control”

module, leave "readclk" always asserted. To convert a "forward control" module to a "backward control" module, pass the "readclk" from the downstream module to the upstream module. When there is a latency restriction, it may be necessary to add a buffer to reduce latency.

3.3 "No Control" Interfaces

Rarely, a "no control" interface is the most natural. For example, the UART transmit driver does not actively request for data to transmit – it only knows that its internal buffers are cleared and is ready to transmit data. In such cases, a "rdy" signal is used instead, which is asserted when a module is ready to consume data. "rdy" cannot be treated as a "readclk" – a module asserts "rdy" when it is ready to consume *one unit* of data, but asserting "readclk" for multiple cycles would result in that many units of data.

"No control" interfaces are, however, incompatible with the rest of our system, so they are usually coupled with a "stream_coord" or "stream_coord_buf" module. This module asserts a single "readclk" when "rdy" is asserted, and waits for one unit of data to arrive before asserting "readclk" again. The buffered version adds a buffer to comply with the latency restriction when necessary.

4 UART Drivers (Mark)

The UART RX (receive) driver receives data over the RS232 UART interface, while the UART TX (transmit) driver transmits data over that interface. Both operate UART at 12MBaud, and thus require a 120Mbps clock. The TX driver is used only to dump the contents of RAM or VRAM for debugging.

Both the TX and RX drivers operate similar to the ones implemented in lab, except that they operate on a clock frequency different from the main system clock. This did not introduce many complications apart from an additional IP FIFO layer to synchronize across the clock boundary.

We previously tried to implement IP-less synchronization since 120Mbps seemed like a nice multiple of 50Mbps. However, the resulting constraint was on the order of 1ns, which was too short even for direct routing. Using an IP FIFO was ultimately the cleanest solution.

Data received over UART is written directly to a buffer. With flow control enabled, this buffer may become full if the networking module is unable to transmit packets successfully. When the buffer is almost full, CTS is used to pause the flow of data from the laptop. A small technical detail is that RTS/CTS handling must be enabled in pyserial on the laptop side for this to work.

5 Networking (Mark)

5.1 The Network Stack

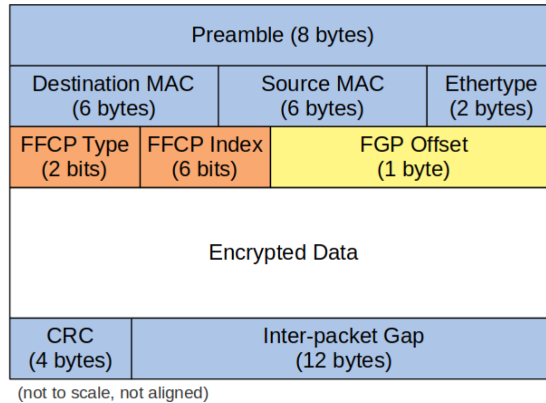

Figure 6: Structure of a complete frame transmitted by the Ethernet module. For clarity, the fields are not to scale or aligned as you might expect from similar diagrams. Blue regions are generated by the Ethernet module, orange regions by the FFCP module and yellow regions by the FGP module.

The network stack used in our system is significantly simpler than a traditional network stack. It comprises three main layers – the Ethernet layer, the FGP (FPGA Graphics Protocol) layer, and the FFCP (FPGA Flow Control Protocol) layer. The Ethernet layer is determined by the Ethernet specification[4].

5.2 The FGP Protocol

The FGP layer is a DMA (Direct Memory Access) protocol invented for this project. It instructs the receiver to write 512 words (given by the encrypted payload) to an offset in VRAM (video memory – the buffer that the VGA module reads from and displays on the screen). Each word is a 12-bit color, and consecutive words are packed into bytes, so the payload would always be $512 * \frac{8}{12} = 768$ bytes long. At the receiving end, a stream transformation module is used to unpack bytes into 12-bit words. The FGP header consists of a single byte, indicating the offset that the data should be written to, divided by 512.

The FGP protocol was designed to transmit video robustly even when the network is unreliable. For example, consider a system that simply transmits video data, split into packets of less than 1500 bytes so as to fit the maximum length of an Ethernet frame. If a single packet is dropped, the rest of the video would be written to the wrong offset in VRAM, causing the the video to be

[4]Ethernet specification: `https://ieeexplore.ieee.org/document/7428776`

rendered out of frame. The FGP protocol avoids this – if a single packet is dropped, the region of memory that the packet was supposed to write to would still contain image data from the previous frame, and subsequent packets would be written to the correct locations in memory.

5.3 The FFCP Protocol

The FFCP layer is a flow control protocol also invented for this project. It is a very stripped-down version of TCP sequence numbers used to ensure a complete, order-preserving data stream. There are three types of FFCP messages – SYN, MSG and ACK. SYN and MSG messages are sent by the transmitting FPGA, while ACK messages are sent by the receiving FPGA.

The FFCP header consists of a single byte. The most significant two bits indicates what type of message it is (0 for SYN, 1 for MSG, 2 for ACK), and the least significant six bits is the sequence number.

SYN and MSG messages are essentially the same, except that SYN is used for the first packet in a stream, while MSG is used for the rest of the packets. Sequence numbers are used to locally identify where in the stream each packet lies. The first packet has sequence number zero, the next has sequence number one, and so on. The sequence number wraps around – for example the 2^6-th packet has sequence number zero.

ACKs also contain a sequence number. An ACK with sequence number n indicates that the receiver has received all messages up to and not including the message with sequence number n.

The transmitting FPGA maintains a transmit window of length 4, starting at the index of the latest ACK it has received. If it receives an ACK outside of the transmit window, the ACK is simply dropped. This mitigates the wrapping around of the sequence number – for example, if the same ACK is received twice, the second ACK is ignored (instead of being treated as an ACK for the $(2^6 - 1)$th message after the start of the window) because it lies outside the transmit window.

The transmitting FPGA attempts to transmit each message in the transmit window in order, even in the absence of ACKs, stopping at the end of the window (or when there are no more packets to transmit). If no ACKs are received for 1ms, it tries transmitting the messages again. This continues for 1s, at which point the transmitting FPGA restarts the stream (by resetting the window to zero and sending out SYN packets). This ensures that the system continues to work even when something catastrophic happens, such as if the receiving FPGA is reset.

The receiving FPGA also maintains a (receive) window of length 4, and drops any packets with a sequence number outside that window. It records which sequence numbers in the window it has received. When the packet at the start of its window has been received, it "commits" the packet (in our case, this means that it executes the FGP write) and advances its window. When it can no longer advance its window, it sends an ACK for the start of its window, indicating that it has received all the packets up to that point.

5.4 Implementation Overview

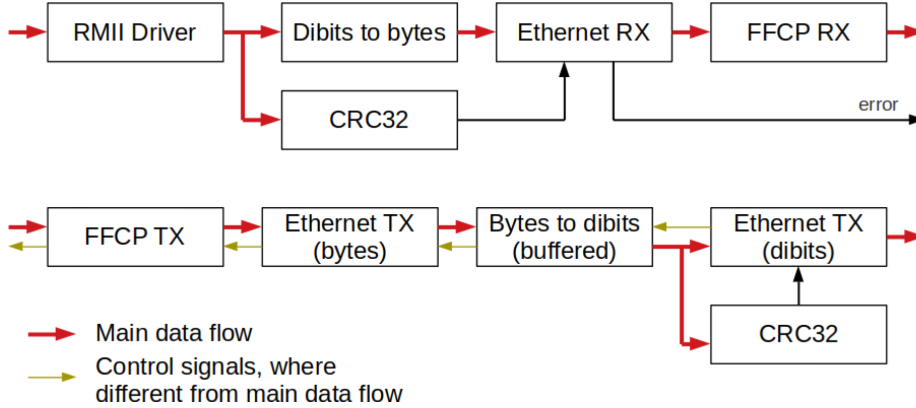

Figure 7: The networking system. The receive pipeline is shown on top, and the transmit pipeline is shown below.

The networking pipeline (Fig. 7) roughly follows the packet structure, with some complications. While FFCP and parts of the Ethernet layer are naturally divided into units of bytes, the Ethernet physical layer is fundamentally composed of dibits, and the Ethernet frame itself has to be transmitted over the dibit-based RMII interface. Stream packing and unpacking modules are used to convert between bytes and dibits in the receive and transmit paths. In the transmit path, the bytes-to-dibits module is buffered to satisfy the latency restriction (ref. Section 3.2).

FGP layer processing is, strictly speaking, not part of the networking pipeline. On the transmit side, the FGP header is generated by the laptop and transmitted over UART, and follows the daisy chain all the way to the RMII interface. FGP processing is used only to split the header from the payload in the packet transmit path, just so that the payload may be encrypted and combined with the header again. On the receive side, the FGP header is again only transiently separated from the payload for payload decryption, and is stored alongside the payload in the packet buffer. It would later be used during the commit stage to determine the offset where the data should be written to the VRAM.

5.5 The RMII Driver

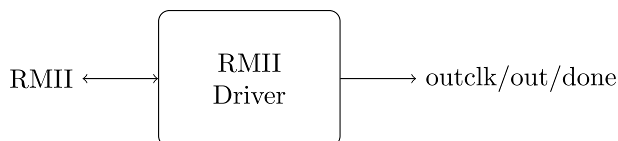

Figure 8: I/O diagram for the RMII driver. This and subsequent module diagrams follow the following conventions: the 50MHz clock (clk) and reset (rst) inputs are omitted, upstream signals are on the left, and downstream signals are on the right.

The RMII driver is responsible for configuring and receiving data from the RMII interface[5]. Since we operate in full duplex mode, there is no need to communicate with the RMII driver to transmit data – the data can be written directly to txen/txd.

A large responsibility of the RMII module is properly resetting the PHY (technically, this isn't part of the RMII specification, but it requires the same signals that the RMII interface uses). The PHY would not work at all without a reset, and could work only partially or in a buggy fashion if the reset is not performed correctly.

There was an interesting technical problem that we ran into when configuring the PHY. The "mode" configuration strap is split across the crsdv and rxd signals (which are tri-state buffers), so it is tempting to do something like this:

```
assign {crsdv, rxd} = rst ? 3'bzzz : DEFAULT_MODE;
```

However, concatenating the signals causes them to lose their tri-state statuses, and the compiler may wrongly treat them as outputs. The solution is to assign to crsdv and rxd separately.

The RMII driver is also responsible for receiving data from the PHY. It splits the crsdv signal into its component crs and dv signals. According to the RMII specification, if crsdv toggles every clock cycle, then dv is asserted while crs is not – otherwise, either both crs and dv are asserted or both aren't, as determined by the value of crsdv. Only the dv (data valid) signal is relevant to us – it indicates when data on rxd is valid.

Another intricacy of the RMII interface is that crsdv is asserted asynchronously at the start of a frame, so some synchronization is needed. There could be some time afterwards before data is presented on rxd, so the RMII driver uses the end of the Ethernet preamble (a long strip of 62 alternating ones and zeroes, ending with two ones) to detect the start of the frame. The frame, with the preamble stripped, is then passed to the Ethernet module.

[5] RMII specification: `http://ebook.pldworld.com/_eBook/-Telecommunications,Networks-/TCPIP/RMII/rmii_rev12.pdf`

5.6 Ethernet Implementation

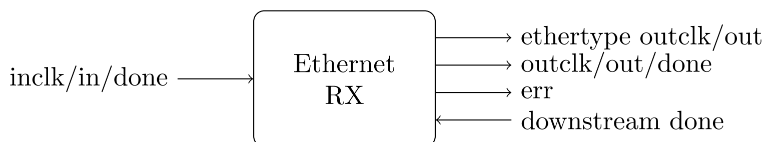

Figure 9: I/O diagram for the Ethernet RX module.

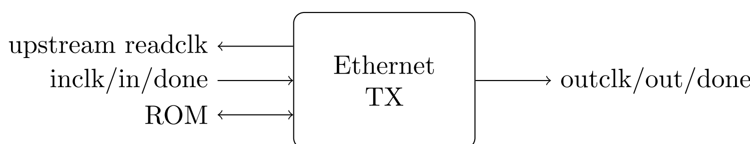

Figure 10: I/O diagram for the Ethernet TX module.

The Ethernet modules are responsible for the MAC address fields, the EtherType field, the CRC, and, in the case of the transmitter module, the preamble and inter-packet gap (consecutive frames should be separated by at least 12 empty octets). It straddles the boundary between dibits and bytes – dibits are used for the CRC module, preamble, and inter-packet gap, while bytes are used for the MAC addresses and EtherType.

The Ethernet RX (receive) module, like most other networking modules in our system, is a simple state machine $mac_{dst} \rightarrow mac_{src} \rightarrow ethertype \rightarrow payload \rightarrow crc \rightarrow done \rightarrow mac_{dst}$, following the structure of an Ethernet frame, with the additional property that any errors in transmission (such as an invalid CRC) moves it directly into *done*. When the EtherType is read, it is presented on the "ethertype outclk/out" interface.

One perhaps unintuitive aspect is that the transition $payload \rightarrow crc$ is triggered by the "downstream done" signal. This happens because the Ethernet frame provides no way to determine the length of the payload, so it is up to the downstream module (in our case, the FFCP RX module) to determine the length of the payload and to inform the Ethernet RX module when the payload is complete.

The Ethernet TX module is also a simple state machine (actually, our implementation splits it into a separate state machine for bytes and for dibits, but this isn't necessary). It multiplexes between reading data from ROM (e.g. for the MAC addresses), reading data from the upstream module (for the payload) and generating data internally (e.g. for the CRC).

Both the RX and TX modules share an implementation of an Ethernet CRC32 module. The transmitter module computes and transmits the CRC,

while the receiver module computes the CRC to verify that the packet has been received correctly.

The CRC module turned out to be one of the more difficult parts of the system, not because of the implementation, but because it was poorly documented. While the CRC algorithm was described in the Ethernet documentation, it was light on details, such as whether the CRC was big- or little-endian (in bits and in bytes). While the exact CRC algorithm does not matter if we only transmitted data between FPGAs, it is important when transmitting data from an FPGA to a laptop since network interface cards (NICs) filter out frames with invalid CRCs. This was a problem because we were using FPGA-laptop communication to verify and experiment with our implementation. We ultimately resorted to comparing our implementation with existing C implementations from online.

We later (too late) found a better way to deal with the CRC problem. It is in fact possible to disable CRC filtering in most NICs via ethtool on Linux using the following command:

```
ethtool -K <interface> rx-all on
```

Even better, most NICs can be programmed to relay the CRC to the operating system (so that you can inspect it with Wireshark, for example) using the following command:

```
ethtool -K <interface> rx-fcs on
```

We arbitrarily chose 0xca12 as the EtherType for FFCP. Using an unused EtherType allows us to filter only for FFCP packets. This is useful for FPGA-laptop debug setups, since the laptop may try to send unrelated data over the Ethernet interface.

5.7 FFCP Implementation

The FFCP RX and TX modules have essentially the same structure as the Ethernet RX and TX modules. Both are state machines that follow the structure of the FFCP packet (in this case, there are only two states – one state for the type/index byte, and one for the payload). The main difference is that the FFCP RX module does not need to check for errors.

The FFCP RX and TX modules are only responsible for FFCP packet parsing and synthesis, not flow control. Flow control is handled by the FFCP RX and TX server modules, which are only provided with metadata summaries of FFCP packets after their corresponding Ethernet frames have been completely received and validated by the Ethernet RX module.

The FFCP RX and TX server modules are mostly defined by the specification outlined in Section 5.3. One implementation detail is that the bit vector on the receiving side which records which sequence numbers have been received is implemented as inferred BRAM. This makes implementation significantly easier since inferred BRAM, unlike IP Coregen BRAM, has no latency on the order of clock cycles.

When a packet is received, the bit in the bit vector corresponding to its sequence number is set. When a packet is committed, the bit in the bit vector corresponding to its sequence number is cleared. In order to ensure that only one write happens at each core cycle, our implementation is designed to commit packets only when no packet is currently being received.

6 The AES Cryptosystem (Ashley)

6.1 The AES Algorithm

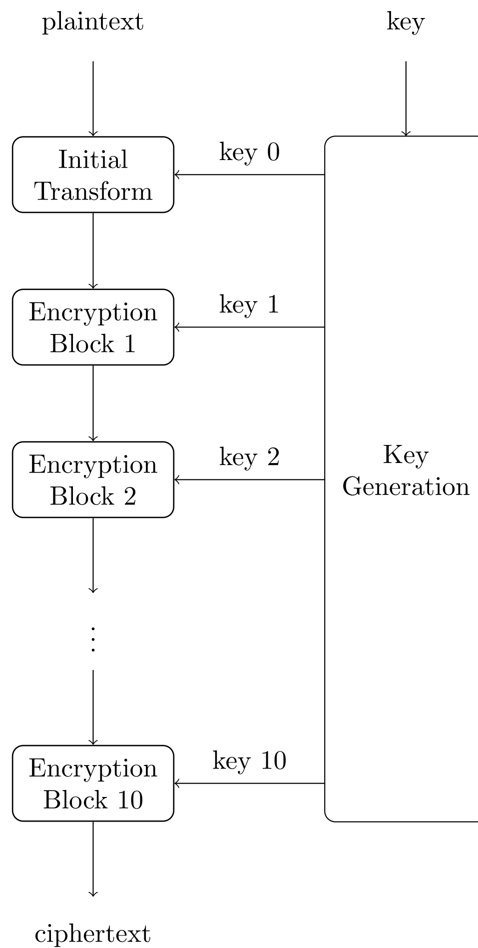

Figure 11: Canonical diagram of the AES algorithm.

The AES encryption algorithm uses a chain of 10 rounds of a single encryption block (Fig. 11). Decryption works similarly, with the encryption blocks replaced with decryption blocks, and the flow of information reversed.

6.2 Encryption/Decryption Blocks

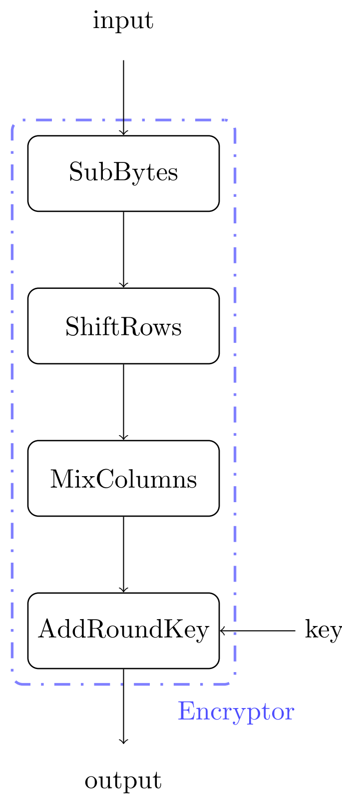

Figure 12: Components of an encryption block.

Each encryption block in the chain looks like the diagram in Figure 12, with the exception of the last round, which skips the MixColumns step.

This skipping of a step gives us a nice property. We can now implement the decryption very similarly, chaining the inverse versions of ShiftRows, SubBytes, AddRoundKey, and MixColumns together, also skipping the MixColumns step in the last round. This gives us a nice consistency that lets us use one module both for encryption and decryption.

6.3 Implementation Overview

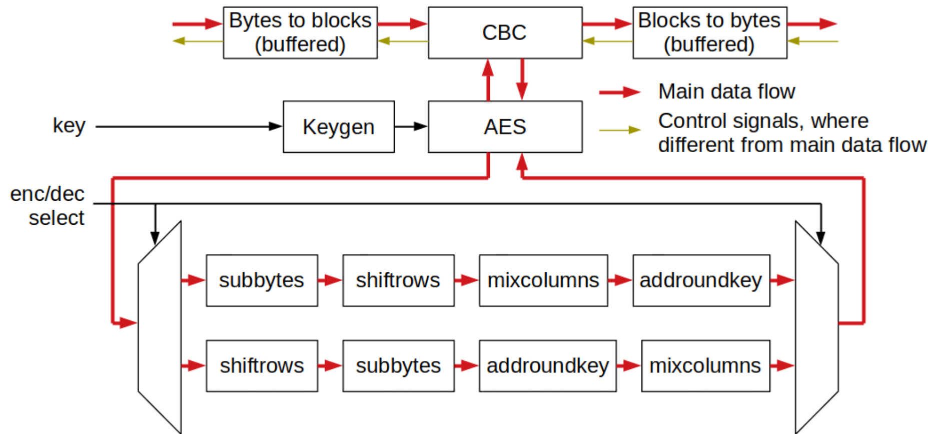

Figure 13: The AES cryptosystem.

Our implementation has a few interesting differences.

We only synthesize one copy of the encryption and decryption blocks. The encryption process takes 10 clock cycles, each corresponding to a single round. The encryption and decryption blocks are reused each round, with the output from each clock cycle used as the input for the next.

Since the components of the encryption and decryption blocks are very similar in both the encryption and decryption paths, just wired in a different order, we originally hoped to reuse them for both encryption and decryption. Thus, each component takes in a "decrypt flag" which configures them for either encryption or decryption. However, since they are purely combinatorial, attempting to multiplex their order in the chain resulted in a combinatorial loop, which is why our final implementation synthesizes the encryption and decryption blocks separately.

Due to timing restrictions (ref. Section 2.3), we pulled the key generation out of the encryption and decryption pathways. We derived the round keys beforehand, and used them for all of the blocks.

We also expanded the system using CBC, or Cipher Block Chaining, which is discussed further in a later section.

6.4 AES Encryption/Decryption

6.4.1 SubBytes

Using a lookup table, each of the 16 bytes of the input is replaced with a substitution byte from a pre-programmed lookup table (a 16x16 grid of 1-byte

values). This requires storing 2 kilobits of memory for both both encryption and decryption.

The substitution is actually derived from taking the inverse of the byte in $GF(2^8)$, and taking a linear transformation of the resulting value. However, as this computation took nontrivial amounts of time and we had no trouble storing the substitution tables in memory, we implemented it as a pure lookup.

We implemented a separate lookup table that was used when the decrypt flag was set.

6.4.2 ShiftRows

We separate the block of 16 bytes that we got as the input into four rows of four bytes, and cyclically shift each column to the right by its index.

When the decrypt flag was set, we simply inverted the operation by shifting each row left by its index.

6.4.3 MixColumns

For each column, we left multiply by the matrix

$$\begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix}$$

to derive a linear transformation. Then, we recombine the columns and reassemble the 128 bits in row-major order, taking the first four bytes of the result to be the first byte of each of the four columns, and analogously for the next twelve bytes.

When the decrypt flag was set, we used the inverse map

$$\begin{bmatrix} 14 & 11 & 13 & 9 \\ 9 & 14 & 11 & 13 \\ 13 & 9 & 14 & 11 \\ 11 & 13 & 9 & 14 \end{bmatrix}$$

instead.

For both directions, however, we needed to remember that the multiplication happens in $GF(2^8)$. Thus, the behavior when naively multiplying modulo 2^8 as natively supported by Verilog, is insufficient.

However, we can represent multiplication by 2 of $byte[7:0]$ as follows:

```
doubled_byte[7:0] = {byte[6 : 0], 1'b0} ^ (8'h1b & {8{byte[7]}});.
```

Then, as all of the coefficients in both the matrices are small, we can simulate the multiplication through a small sequence of bit-shifts and additions.

6.4.4 AddRoundKey

Using the key that was derived using our key generation module, we xor the input with the key we received, and output that result.

6.5 Round Key Generation

Using the initial 16-byte key, we derive the key using a recursive algorithm We split the initial key into four 32-bit blocks. Then, we derive an expanded key, which ends up being 11 128-bit keys concatenated together, as follows.

We define $W_0, ..., W_{4R-1}$ as the 32-bit blocks that make up the expanded key. Then, we utilize two submodules, $RotWord$, which cyclically shifts the input left by a byte, and $SubBytes$, which was a submodule used in AES for substitution. Most generally, each block W_i can be derives as follows.

$$W_i = \begin{cases} K_i & \text{if } i < N \\ W_{i-N} \oplus \text{RotWord}(\text{SubWord}(W_{i-1})) \oplus rcon_{i/N} & \text{if } i \geq N \text{ and } i \equiv 0 \pmod{N} \\ W_{i-N} \oplus \text{SubWord}(W_{i-1}) & \text{if } i \geq N,\ N > 6, \text{ and } i \equiv 4 \pmod{N} \\ W_{i-N} \oplus W_{i-1} & \text{otherwise.} \end{cases}$$

$rcon_i$ indicates the round constant, and is defined by

$$rcon_i = \begin{cases} 1 & \text{if } i = 1 \\ 2 \cdot rcon_{i-1} & \text{if } i > 1 \text{ and } rcon_{i-1} < 80_{16} \\ (2 \cdot rcon_{i-1}) \oplus 11B_{16} & \text{if } i > 1 \text{ and } rcon_{i-1} \geq 80_{16} \end{cases}$$

In our implementation of AES, our block size is 128 bits, and we have only 10 rounds. Thus, we can entirely ignore the third case in the argument.

This allows us to rewrite our key derivation in a much simpler form. If we let key_i, the key for the i'th round, get represented as the concatenation $\{w_{i,0}, w_{i,1}, w_{i,2}, w_{i,3}\}$ of 32-bit words, we can recursively derive

$$\begin{aligned} w_{i,0} &= w_{i-1,0} \oplus rcon_i \oplus \text{RotWord}(\text{SubWord}(w_{i-1,3})) \\ w_{i,1} &= w_{i,0} \oplus w_{i-1,1} \\ w_{i,2} &= w_{i,1} \oplus w_{i-1,2} \\ w_{i,3} &= w_{i,2} \oplus w_{i-1,3} \end{aligned}$$

With this approach, we can compute all ten keys in 10 cycles with fairly simple code.

6.6 CBC

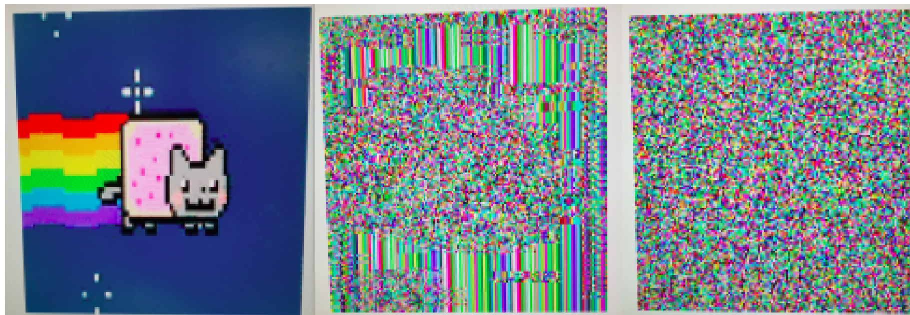

Figure 14: Visual demonstration of the effects of using AES in CBC mode. On the left is the plaintext, which contains many regions of repeated pixels. In the middle is the ciphertext in ECB (Electronic Codebook – AES without modification) mode, and on the right is the ciphertext in CBC mode. The repeated regions are clearly visible in the ECB ciphertext, but not in the CBC ciphertext.

While AES itself is robust, it has the downside that, given a fixed initial key and plaintext, we always get the same output. Thus, we can easily leak the structure of a plaintext that consists of a large number of similar blocks.

To mitigate this problem, a technique called CBC, or *cipher block chaining*, was introduced. The approach simply asks us to take the xor of the plaintext block we are currently encrypting with the ciphertext of the previous block before running it through the encryption algorithm. This system is just as easy to decrypt, as we know what the initial value the plaintext was xor'ed with is. However, this hides similar blocks of plaintext very well, as the value that was encrypted depends on all of the values that came before it.

We used a fixed initial block (called the initialization vector, or IV), and took the xor of the first block with it. Then, we proceeded to chain. While robust, this system has the tradeoff that a single bit error in the entire stream can cause the decryption to fail. Thus, for this to be viable, we needed robust flow control.

In more secure AES implementations, the IV is random to prevent information leakage when the stream is reset. This requires the IV to be transmitted along with the first block in the stream, which requires changes to the networking infrastructure. This was left out due to time constraints.

7 Graphics (Mark)

The graphics module displays data from the VRAM over VGA. While previous labs provided a VGA module, we had to modify it considerably since we were

working in a different VGA mode (800x600 72Hz).

Adapting the VGA module for our preferred mode was not as smooth as expected. It turns out that one additional parameter in a VGA mode is the "polarity". In the lab's 1024x768 60Hz mode, the polarity is such that hsync and vsync are active low. However, in the 800x600 72Hz mode, the polarity is reversed, and hsync and vsync are active high.

Apart from that, the graphics module is responsible for reading pixel data from the VRAM, delaying hsync and vsync to be in sync with the pixel data, and displaying the image scaled up and centered on the screen. All this was rather straightforward.

8 Development Process

Here we describe the steps that we took to bring up our project, which strongly influenced the design of our system. We also summarize the key challenges that we faced over the course of the project.

Throughout the process, we used simulation extensively to verify our designs. These concrete, testable steps made debugging a lot easier, assuring us that code for each step is built on reliable, tested code from the previous step.

Due to the clear separation between the two parts of the project (networking and cryptography), as well as the modular design of all of the components, we could work simultaneously without being blocked by each other, or by small bugs. Furthermore, it meant that we could write tests for expected behavior of the design from a very low-level granularity.

On the networking side, the first step was to configure the Ethernet PHY. Using a laptop to transmit Ethernet frames to the FPGA, we can check that the PHY has been configured correctly using ethtool on Linux. By connecting crsdv/rxd directly to JB, we could use the oscilloscope to check that the PHY was announcing frames.

The next step was to check that the frames received from the RMII interface were of the expected format. This involved streaming the Ethernet frame into BRAM and implementing a UART transmission interface to stream the BRAM to a laptop, so that we could inspect the packets. This step was unexpectedly difficult due to a confluence of several factors.

First, the unintuitive tri-state buffer bug (ref. Section 5.5) caused only one bit of rxd to be treated as a tri-state, resulting in corrupted data (red flag: one bit of rxd was always zero). Second, a typo in the reset timings caused the PHY to only be partially reset, causing it to interlace frames at random times (red flag: stretches of alternating bits that looked like preambles kept appearing in the middle of packets, but this stops when the FPGA is manually reset). Third, Ethernet is little-endian in bits but big-endian in bytes, resulting in a lot of confusion as to what exactly we expected to see in a received packet.

The next step was to transmit Ethernet frames from the FPGA to a laptop. The main difficulty here was figuring out how to compute the CRC correctly

(ref. Section 5.6), since there are many ways to take a 32-bit CRC, even with the same polynomial, and only one is accepted by the laptop's NIC.

Having tested both Ethernet reception and transmission, we then proceeded to try Ethernet communication between FPGAs, which worked without issue.

The next step was to complete the daisy chain, receiving data from a laptop over UART and displaying VRAM over VGA. The main roadblock for this step was the VGA polarity issue (ref. Section 7).

Then, it was time to upgrade the UART RX module from 115200 baud to 12MBaud. Most of the time for this step was wasted trying to manually synchronize signals across the 50MHz/120MHz clock boundary (ref. Section 4). We also upgraded the image from 32x32 to 128x128.

With the full daisy chain working, the final step was flow control. Getting flow control to work well was difficult, since an incorrect flow control implementation is not always immediately obvious in simulation, or even on an FPGA. We eventually had to pore over simulations of the entire system, making sure that every signal behaved as expected. Fortunately, due to the modularity of our system, it was not difficult to integrate flow control into our overall architecture.

On the AES side, the first step was to implement a single encryption block and to verify it against a Python implementation through simulation. This was followed by the decryption block. Each component was individually tested to make sure the inverse pairs did indeed invert each other. Next, multiple blocks were chained together using the same key throughout (that is, without round key generation) and verified. Finally, we implemented round key generation.

After flow control was established, the next step was to implement CBC. While CBC itself was not very difficult, it introduced a significant additional complication.

Originally, the AES encryption module was placed directly after the UART RX module. With CBC, we might need to encrypt the same block two different times with a different previous block – for example, if the stream is reset, the previous block must be the fixed initialization vector. This means that the AES module had to be moved into the packet transmission pipeline, which meant that the AES module had to be converted from a "forward control" module to a "backward control" module with a tighter latency restriction. So, instead of the 20 cycles required for the naive implementation, we only had 16. We fixed this problem by moving the round key generation out of the AES pipeline, and simply storing all of the generated keys before even starting the encryption process.

Another complication is that CBC mode necessitated the idea of a "commit". It was no longer sufficient just to ensure that all packets were received – we had to ensure that all packets were received exactly once, and in order. This required significant changes to the flow control modules (specifically, the FFCP TX/RX servers), though the existing simulation infrastructure we had set up by then meant that the resulting issues were not difficult to debug.

9 Conclusion

Overall, this project was a success, especially with the implementation of our stretch goals. It was very satisfying to watch the system respond robustly to us pulling out the Ethernet cable and plugging it back in, and also work over a 125-foot long Ethernet cable (thanks to Joe for coming up with that). It was also fun to have a live, visual demonstration of the perils of using AES in ECB mode (ref. Section 6.6).

In the end, our project demonstrates that the Ethernet interface on the Nexys 4 can be used to create systems that are interesting both visually and technically. We hope that our project serves both as a reference implementation and inspiration for future 6.111 networking projects.

9.1 Lessons Learned

Apart from the technical lessons described in Section 8:

- Version control (with Git) was critical to the success of our project. It allowed us to feel comfortable refactoring and re-designing large portions of the codebase, knowing that we can always revert to a previous version if things mess up. It also made collaboration a lot easier. Unfortunately, Vivado's handling of IP cores was not designed to be easily integrated with version control, though we got around the issue by gitignoring everything in the project directories except for the top-level .xci and .prj files.

- Simulation is a really important and useful tool for fast development. You don't need to write complete 6.031 unit tests – just being able to run your system and study the waveform diagram is enough. It's a lot faster and easier to debug than synthesizing the code onto the physical FPGA. Additionally, you can work on the project outside lab (unless you're on OS X). Don't be afraid to put your entire top-level module into simulation (ref. Appendix B).

- One tip that makes simulation a lot easier is that you can put Verilog code other than the unit under test into a simulation. This is usually a lot easier than fiddling with procedural code to create the right inputs at the right clock cycles, especially when there are a lot of things going on at once. For example, in our main system test, we reused modules that we previously implemented to simulate the UART stream from the laptop (e.g. the UART TX module).

- In general, don't be hesitant to write code that wouldn't be directly used in the final product. For example, the UART TX debugging interface was an invaluable resource throughout the project.

- At the same time, even when all of our code worked well in simulation, this didn't always translate to things working in the real world. We were fortunate to have started early, giving us time to iron out the difficulties

we faced when interfacing with hardware. For example, we're not aware of any simulation models for the Ethernet PHY, so developing the RMII driver required a lot of debugging with the physical FPGA in lab.

- Consistency is important. Early in the project, before settling on a consistent data flow interface (ref. Section 3), almost every module in the daisy chain had its own naming conventions for inputs and outputs. This made connecting things together very confusing and error-prone. Development was a lot easier when we arbitrarily decided on the inclk-outclk-readclk convention. Of course, it would have been a lot better if we used inen-outen-rden instead, but keeping things consistent was a more important goal.

- Consistency also applies to implementation. For example, settling on a good skeleton specification for the Ethernet RX module (a Mealy machine multiplexing different data sources with a fixed 2-clock-cycle latency, ref. Section 5.6) made subsequently implementing similar modules (e.g. FFCP RX) very simple.

- Modularity helps. At the high-level, there are two ways to go about implementing a project like ours. The first is to synchronize all the components with master coordinator modules, and the second, which was what we opted for, is to make all the modules mostly independent, communicating with each other through data flow interfaces. While the globally coordinated paradigm would have made development a lot faster for a small system, the "locally coordinated" paradigm was extremely convenient especially when the design of the system was still in flux (as is normal in projects like this), and made adding flow control a lot less difficult than it could have been. This is because "local coordination" is more modular – for example, we could move the entire AES encryption module from the UART receive to the Ethernet transmit path without having to worry about timings in the rest of the system.

- One tip for modularity is to use header files to share parameters across a large number of modules. The clog2 function (ref. Appendix A.2) can be used to convert maximum values into bus widths. Good global parameterization allowed us to easily change the VRAM size from 4096 words to 16384 when we increased the image size from 32x32 to 128x128.

- Surprisingly, block diagrams weren't as helpful to our project as one might expect, since the data flow in our system was very simple and linear, but the design was constantly in flux (oftentimes out of necessity – for example, when bringing up the RMII driver, the data flow looked like $ethernet \rightarrow RAM \rightarrow UART$, ref. Section 8). More important to us was a good data flow interface design, which allowed us to cleanly separate the networking and AES parts of the project and shuffle modules around easily.

9.2 Possible Extensions

Some possible extensions that might be of interest:

- Refactor everything to use inen-outen-rden instead of inclk-outclk-readclk.
- Allow the AES module to transmit a randomly generated IV. The SYN packet could be used to transmit just the IV.
- Transmit other forms of data, such as audio alongside video. This just requires an additional multiplexing step on the receive end.
- Implement a key exchange for the AES protocol. Now, instead of programming the same static key into each FPGA, we could have each FPGA generate its own unpredictable key, and derive a shared one through a preliminary handshake. This would require significant improvements to the networking stack, including the design of a key exchange protocol.
- Communicate over MITnet. Both FPGAs could be plugged into the wall Ethernet ports, and could communicate with each other over UDP (perhaps with the assistance of an external server to perform UDP hole punching). This would require implementing a simple DHCP and ARP client.

A Verilog Code

In these appendices, we provide a bare minimum of code required to get the project up and working. The complete repository will soon be uploaded to Github (probably `https://github.com/krawthekrow/fpganet`).

A.1 IP Cores Used

- bit_stream_fifo: FIFO generator, independent clocks distributed RAM, width 1, depth 16.
- byte_stream_fifo: FIFO generator, independent clocks distributed RAM, width 8, depth 16.
- clk_wiz_0: Clocking wizard, clk_out1 at 50MHz, clk_out3 at 120MHz.
- packet_buffer_ram: Block memory generator, simple dual port RAM, width 8, depth 16384, ports always enabled, init file debug.coe.
- packet_synth_rom: Block memory generator, single port ROM, width 8, depth 4096, ports always enabled, init file packet_synth.coe.
- video_cache_ram: Block memory generator, simple dual port RAM, width 12, depth 16384, ports always enabled, init file debug.coe.

A.2 Headers (Includes)

util.vh:

```
function integer clog2(input integer size);
begin
  size = size - 1;
  for (clog2 = 1; size > 1; clog2 = clog2 + 1)
    size = size >> 1;
  end
endfunction
```

params.vh:

```
`include "util.vh"

localparam SYNC_DELAY_LEN = 3;

localparam BYTE_LEN = 8;
localparam COLOR_CHANNEL_LEN = 4;
localparam COLOR_LEN = COLOR_CHANNEL_LEN * 3;
localparam BLOCK_LEN = 128;

localparam PACKET_BUFFER_SIZE = 16384;
// taken from ip summary
localparam PACKET_BUFFER_READ_LATENCY = 2;

localparam PACKET_SYNTH_ROM_SIZE = 4096;
localparam PACKET_SYNTH_ROM_LATENCY = 2;

localparam VIDEO_CACHE_RAM_SIZE = 16384;
localparam VIDEO_CACHE_RAM_LATENCY = 2;

localparam VGA_WIDTH = 800;
localparam VGA_HEIGHT = 600;
```

networking.vh:

```
`include "params.vh"

// measured in bytes/octets
localparam ETH_PREAMBLE_LEN = 8;
localparam ETH_CRC_LEN = 4;
localparam ETH_GAP_LEN = 12;
localparam ETH_MAC_LEN = 6;
localparam ETH_ETHERTYPE_LEN = 2;

localparam ETHERTYPE_FGP = 16'hca11;
```

```
localparam ETHERTYPE_FFCP = 16'hca12;

// measured in bytes
localparam FGP_OFFSET_LEN = 1;
localparam FGP_DATA_LEN = 768;
// measured in 12-bit words (colors)
localparam FGP_DATA_LEN_COLORS = FGP_DATA_LEN * BYTE_LEN / COLOR_LEN;
localparam FGP_LEN = FGP_OFFSET_LEN + FGP_DATA_LEN;

// measured in bits
localparam FFCP_TYPE_LEN = 2;
localparam FFCP_INDEX_LEN = 6;
// measured in bytes
localparam FFCP_METADATA_LEN = 1;
localparam FFCP_DATA_LEN = FGP_LEN;
localparam FFCP_LEN = FFCP_METADATA_LEN + FFCP_DATA_LEN;

localparam FFCP_TYPE_SYN = 0;
localparam FFCP_TYPE_MSG = 1;
localparam FFCP_TYPE_ACK = 2;

localparam FFCP_BUFFER_LEN = 2**FFCP_INDEX_LEN;
localparam FFCP_WINDOW_LEN = 4;

// packet buffer parameters (used in flow control)
// each partition should have enough space to store an entire packet
localparam PB_PARTITION_LEN = 2**clog2(FFCP_LEN);
// number of partitions in the packet buffer queue
localparam PB_QUEUE_LEN = PACKET_BUFFER_SIZE / PB_PARTITION_LEN;
localparam PB_QUEUE_ALMOST_FULL_THRES = PB_QUEUE_LEN * 7 / 8;
```

A.3 Utilities

util.v:

```
// delays signals on in for a DELAY_LEN cycles
// accepts input at any time, no need to wait for an input to
// appear on out
module delay #(
  // number of delay cycles
  parameter DELAY_LEN = 1,
  parameter DATA_WIDTH = 1) (
  input clk, rst, [DATA_WIDTH-1:0] in,
  output [DATA_WIDTH-1:0] out);

// shift register holds input from previous cycles
```

```
reg [DELAY_LEN*DATA_WIDTH-1:0] queue;
assign out = queue[0+:DATA_WIDTH];

always @ (posedge clk) begin
  if (rst)
    queue <= 0;
  // if DELAY_LEN is 1,
  // queue[DATA_WIDTH+:(DELAY_LEN-1)*DATA_WIDTH] would have zero length
  // resulting in wrong behavior, so treat this as a special case
  else if (DELAY_LEN == 1)
    queue <= in;
  else
    queue <= {in, queue[DATA_WIDTH+:(DELAY_LEN-1)*DATA_WIDTH]};
end

endmodule

// modified from code provided in previous labs
module debounce (
  input rst, clk, noisy,
  output reg clean);

reg [19:0] count;
reg prev;

always @(posedge clk) begin
  if (rst) begin
    prev <= noisy;
    clean <= noisy;
    count <= 0;
  end else if (noisy != prev) begin
    prev <= noisy; count <= 0;
  end else if (count == 650000)
    clean <= prev;
  else
    count <= count+1;
end

endmodule

module sync_debounce (
  input rst, clk, in,
  output out);

`include "params.vh"
```

```
wire synced;
delay #(.DELAY_LEN(SYNC_DELAY_LEN)) delay_inst(
  .clk(clk), .rst(rst), .in(in), .out(synced));
debounce debounce_inst(
  .rst(rst), .clk(clk), .noisy(synced), .clean(out));

endmodule

// toggles a signal every 2*BLINK_PERIOD clock cycles
// used to create a blinking LED to check that the system is running
module blinker #(
  parameter BLINK_PERIOD = 50000000) (
  input clk, rst, enable,
  output reg out = 0);

`include "util.vh"

// timer count, increases each clock cycle
reg [clog2(BLINK_PERIOD)-1:0] cnt = 0;

always @(posedge clk) begin
  if (rst || !enable) begin
    cnt <= 0;
    out <= 0;
  end else if (cnt == BLINK_PERIOD-1) begin
    cnt <= 0;
    out <= ~out;
  end else
    cnt <= cnt + 1;
end

endmodule

// out is asserted if in is asserted, or if in was asserted at some
// point in the last EXTEND_LEN clock cycles
module pulse_extender #(
  // time to extend pulse by, default 0.1s
  parameter EXTEND_LEN = 5000000) (
  input clk, rst, in, output out);

`include "util.vh"

// timer count, decreases each clock cycle
reg [clog2(EXTEND_LEN+1)-1:0] cnt = 0;
wire done;
assign done = cnt == 0;
```

```
// assert out if timer has not expired
// include in so that out is asserted on the same clock cycle
// that in is asserted
assign out = in || !done;

always @(posedge clk) begin
  if (rst)
    cnt <= 0;
  else if (in)
    cnt <= EXTEND_LEN;
  else if (!done)
    cnt <= cnt - 1;
end

endmodule

// asserts out for a single clock cycle when in is asserted
// out should be asserted on the same clock cycle as the rising edge of in
module pulse_generator (
  input clk, rst, in, output out);

// pulsed indicates that out has been asserted, and should be deasserted
// thereafter until in is deasserted (and asserted again)
reg pulsed = 0;
assign out = in && !pulsed;

always @(posedge clk) begin
  if (rst)
    pulsed <= 0;
  else if (in)
    pulsed <= 1;
  else
    pulsed <= 0;
end

endmodule

// pulses out for a single clock cycle every PULSE_PERIOD
// out is not, and should not be used as a clock
module clock_divider #(
  parameter PULSE_PERIOD = 4) (
  // only pulses if en is asserted
  input clk, rst, en, output out);

`include "util.vh"
```

```
reg [clog2(PULSE_PERIOD)-1:0] cnt = 0;
assign out = !rst && en && cnt == 0;

always @(posedge clk) begin
  if (rst)
    cnt <= 0;
  else if (cnt == PULSE_PERIOD-1)
    cnt <= 0;
  else
    cnt <= cnt + 1;
end

endmodule

// inclk is not a real clock, and only indicates that the data on
// in is valid
// when inclk is asserted, the buffer is overwritten with in, whether
// or not it is empty
module single_word_buffer #(
  parameter DATA_WIDTH = 1) (
  input clk, rst, clear, inclk, input [DATA_WIDTH-1:0] in,
  // empty indicates if the buffer is empty
  output reg empty = 1, output reg [DATA_WIDTH-1:0] out);

always @(posedge clk) begin
  if (rst)
    empty <= 1;
  else if (inclk) begin
    empty <= 0;
    out <= in;
  // clear simply marks the buffer as empty
  end else if (clear)
    empty <= 1;
end

endmodule
```

clocking.v:

```
// synchronizes a reset signal over a clock boundary using an IP FIFO
module reset_stream_fifo(
  input clka, clkb,
  input rsta, output rstb);

wire fifo_empty, fifo_out;
wire fifo_rden, fifo_prev_rden;
bit_stream_fifo reset_fifo(
```

```
  .rst(1'b0),
  .wr_clk(clka), .rd_clk(clkb),
  .din(rsta), .wr_en(rsta),
  .rd_en(fifo_rden), .dout(fifo_out),
  .empty(fifo_empty));
assign fifo_rden = !fifo_empty;
delay reset_fifo_read_delay(
  .clk(clkb), .rst(rstb), .in(fifo_rden), .out(fifo_prev_rden));
// fifo_out is only valid when rden was asserted the previous clock cycle
assign rstb = fifo_prev_rden && fifo_out;

endmodule
```

stream.v:

```
// convert a stream of words of size S_LEN to a stream of words
// of size L_LEN, where S_LEN and L_LEN are powers of 2
// packing is in little-endian order, as determined by ethernet
// no latency between in and out
module stream_pack #(
  parameter S_LEN = 1,
  parameter L_LEN = 2) (
  input clk, rst, inclk, input [S_LEN-1:0] in, input in_done,
  output outclk, output [L_LEN-1:0] out, output done);

`include "util.vh"

localparam PACK_RATIO = L_LEN/S_LEN;

// shift buffer used to pack small words into large ones
// don't need to store last small word
reg [L_LEN-S_LEN-1:0] shifted;
// number of small words received
reg [clog2(PACK_RATIO)-1:0] cnt = 0;

assign out = {in, shifted};
// data is valid on out when PACK_RATIO small words have been received
assign outclk = inclk && cnt == PACK_RATIO-1;
// last large word is presented when last small word is received
assign done = in_done;

always @(posedge clk) begin
  // usually we have cnt == 0 when in_done is asserted, but
  // reset just in case the data stream was incomplete
  if (rst || in_done)
    cnt <= 0;
  else if (inclk) begin
```

```
    // shift from left since we're assuming little-endian
    shifted <= {in, shifted[S_LEN+:L_LEN-2*S_LEN]};
    cnt <= cnt + 1;
  end
end

endmodule

// convert a stream of words of size L_LEN to a stream of words
// of size S_LEN, where S_LEN and L_LEN should be powers of 2
// no latency between in and out
// unpacking is in little-endian order, as determined by ethernet
// assumes that words are inserted no faster than once every
// PACK_RATIO clock cycles
module stream_unpack #(
  parameter S_LEN = 1,
  parameter L_LEN = 2) (
  input clk, rst, inclk, input [L_LEN-1:0] in, input in_done,
  // readclk requests for data to be presented on out, and is
  // not an actual clock
  input readclk,
  output outclk, output [S_LEN-1:0] out,
  // done is pulsed after in_done when buffer has been cleared
  output rdy, done);

`include "util.vh"

localparam PACK_RATIO = L_LEN/S_LEN;

// shift buffer used to unpack large words into small ones
// need to store entire large word, since readclk may not be asserted
// at the same time as inclk
reg [L_LEN-1:0] shifted;
// as a special case, if readclk and inclk are asserted at the same time,
// present the small word from in directly since there hasn't been time
// to store the large word in the shift buffer
assign out = inclk ? in[0+:S_LEN] : shifted[0+:S_LEN];
reg idle = 1;
assign rdy = idle;
assign outclk = (!idle || inclk) && readclk;

// current offset in large word
// we'd have finished processing the large word when cnt == PACK_RATIO-1
reg [clog2(PACK_RATIO)-1:0] cnt = 0;

// in_done_found indicates that in_done has been asserted,
```

```
// and we should assert done when the buffer clears
reg in_done_found = 0;
assign done = outclk && in_done_found && cnt == PACK_RATIO-1;

always @(posedge clk) begin
  if (rst) begin
    cnt <= 0;
    in_done_found <= 0;
    idle <= 1;
  end else begin
    if (inclk && in_done)
      in_done_found <= 1;
    else if (done)
      in_done_found <= 0;
    if (outclk) begin
      cnt <= cnt + 1;
      // if inclk is asserted at the same time as readclk
      // (and thus outclk), we present the first small word
      // immediately, so only store PACK_RATIO-1 small words
      // from the large word
      if (inclk) begin
        idle <= 0;
        shifted <= {{S_LEN{1'b0}}, in[S_LEN+:L_LEN-S_LEN]};
      end else begin
        shifted <= {{S_LEN{1'b0}}, shifted[S_LEN+:L_LEN-S_LEN]};
        if (cnt == PACK_RATIO-1)
          idle <= 1;
      end
    end else if (inclk) begin
      idle <= 0;
      shifted <= in;
    end
  end
end

endmodule

// convert a dibit stream to a bytestream
module dibits_to_bytes(
  input clk, rst, inclk, input [1:0] in, input in_done,
  output outclk, output [BYTE_LEN-1:0] out, output done);

`include "params.vh"

stream_pack #(.S_LEN(2), .L_LEN(BYTE_LEN)) pack_inst(
  .clk(clk), .rst(rst), .inclk(inclk), .in(in), .in_done(in_done),
```

```
  .outclk(outclk), .out(out), .done(done));

endmodule

// convert a bytestream to a dibit stream
module bytes_to_dibits(
  input clk, rst, inclk, input [BYTE_LEN-1:0] in, input in_done,
  input readclk,
  output outclk, output [1:0] out, output rdy, done);

`include "params.vh"

stream_unpack #(.S_LEN(2), .L_LEN(BYTE_LEN)) unpack_inst(
  .clk(clk), .rst(rst), .inclk(inclk), .in(in), .in_done(in_done),
  .readclk(readclk),
  .outclk(outclk), .out(out), .rdy(rdy), .done(done));

endmodule

// convert a bytestream to a stream of AES blocks
module bytes_to_blocks(
  input clk, rst, inclk, input [BYTE_LEN-1:0] in, input in_done,
  output outclk, output [BLOCK_LEN-1:0] out, output done);

`include "params.vh"

stream_pack #(.S_LEN(BYTE_LEN), .L_LEN(BLOCK_LEN)) pack_inst(
  .clk(clk), .rst(rst), .inclk(inclk), .in(in), .in_done(in_done),
  .outclk(outclk), .out(out), .done(done));

endmodule

// convert a stream of AES blocks to a bytestream
module blocks_to_bytes(
  input clk, rst, inclk, input [BLOCK_LEN-1:0] in, input in_done,
  input readclk,
  output outclk, output [BYTE_LEN-1:0] out, output rdy, done);

`include "params.vh"

stream_unpack #(.S_LEN(BYTE_LEN), .L_LEN(BLOCK_LEN)) unpack_inst(
  .clk(clk), .rst(rst), .inclk(inclk), .in(in), .in_done(in_done),
  .readclk(readclk),
  .outclk(outclk), .out(out), .rdy(rdy), .done(done));

endmodule
```

```
// convert a bytestream to a stream of 12-bit colors
// one clock cycle of latency between in and out
module bytes_to_colors(
  input clk, rst,
  input inclk, input [BYTE_LEN-1:0] in,
  output reg outclk, output reg [COLOR_LEN-1:0] out);

`include "params.vh"

// three states to convert three bytes into two colors
// state indicates number of bytes received for each three-byte block
reg [1:0] state = 0;
// stores the input from the previous inclk
reg [BYTE_LEN-1:0] prev_in;

always @(posedge clk) begin
  if (rst)
    state <= 0;
  else if (inclk) begin
    prev_in <= in;
    case (state)
    // if state == 1, combine previous input with first half of
    // current input
    1: begin
      outclk <= 1;
      out <= {prev_in, in[BYTE_LEN/2+:BYTE_LEN/2]};
    end
    // if state == 2, combine second half of previous input with
    // current input
    2: begin
      outclk <= 1;
      out <= {prev_in[0+:BYTE_LEN/2], in};
    end
    // if state == 0, do nothing since we don't have enough data yet
    default:
      outclk <= 0;
    endcase

    // cycle through three states
    if (state == 2)
      state <= 0;
    else
      state <= state + 1;
  end else
    outclk <= 0;
```

```
end

endmodule

// stream data out of memory
// starts only after start is asserted and then deasserted
module stream_from_memory #(
  parameter RAM_SIZE = PACKET_BUFFER_SIZE,
  parameter RAM_READ_LATENCY = PACKET_BUFFER_READ_LATENCY) (
  input clk, rst, start,
  // read_start and read_end only need to be valid when start is asserted
  // read_end points to one byte after the last byte
  input [clog2(RAM_SIZE)-1:0] read_start, read_end,
  input readclk,
  input ram_outclk, input [BYTE_LEN-1:0] ram_out,
  output ram_readclk,
  output [clog2(RAM_SIZE)-1:0] ram_raddr,
  output outclk, output [BYTE_LEN-1:0] out, output done);

`include "params.vh"

assign outclk = ram_outclk;
assign out = ram_out;

// save read_end since it might change after start is deasserted
reg [clog2(RAM_SIZE)-1:0] read_end_buf;
reg [clog2(RAM_SIZE)-1:0] curr_addr;

// disambiguate reading first and last word in case read_start == read_end
reg first_word = 0;

assign ram_raddr = curr_addr;
// idle indicates that the stream has finished
wire idle;
assign idle = !first_word && ram_raddr == read_end_buf;
// if the stream has finished, don't issue ram reads even if readclk
// is asserted
assign ram_readclk = !idle && readclk;

// delay done so it appears when the last word comes out of ram
wire done_pd;
assign done_pd = readclk && (ram_raddr + 1 == read_end_buf);
delay #(.DELAY_LEN(RAM_READ_LATENCY)) done_delay(
  .clk(clk), .rst(rst), .in(done_pd), .out(done));

always @(posedge clk) begin
```

```
  if (rst) begin
    // stop stream even if readclk is asserted
    // (i.e. make idle = 1)
    curr_addr <= 0;
    read_end_buf <= 0;
    first_word <= 0;
  end else if (start) begin
    curr_addr <= read_start;
    read_end_buf <= read_end;
    first_word <= 1;
  // only increment curr_addr when we issue a read to ram,
  // which only happens when readclk is asserted
  end else if (ram_readclk) begin
    curr_addr <= curr_addr + 1;
    first_word <= 0;
  end
end

endmodule

// create a memory write stream
module stream_to_memory #(
  parameter RAM_SIZE = PACKET_BUFFER_SIZE,
  parameter WORD_LEN = BYTE_LEN) (
  input clk, rst,
  // used to set the offset for a new write stream
  // setoff_val is only valid when setoff_req is asserted
  // the setoff interface can be used at the same clock cycle as
  // inclk, so you can begin a new write stream immediately
  // after a write stream completes
  input setoff_req,
  input [clog2(RAM_SIZE)-1:0] setoff_val,
  input inclk, input [WORD_LEN-1:0] in,
  output reg ram_we = 0,
  output reg [clog2(RAM_SIZE)-1:0] ram_waddr,
  output reg [WORD_LEN-1:0] ram_win);

`include "params.vh"

reg [clog2(RAM_SIZE)-1:0] curr_addr = 0;
always @(posedge clk) begin
  if (rst) begin
    ram_we <= 0;
    curr_addr <= 0;
  end else begin
    if (setoff_req)
```

```
      curr_addr <= setoff_val;
    else if (inclk)
      curr_addr <= curr_addr + 1;

    if (inclk) begin
      // issue ram write
      ram_we <= 1;
      ram_waddr <= curr_addr;
      ram_win <= in;
    end else
      ram_we <= 0;
  end
end

endmodule

// coordinate two stream modules so that upstream data is requested
// only after downstream has received the previous word
module stream_coord(
  input clk, rst,
  input downstream_rdy, downstream_inclk,
  output upstream_readclk);

// waiting indicates that we have issued a read to upstream and we are
// waiting for downstream to receive a word
reg waiting = 0;
assign upstream_readclk = !rst && (waiting ? 0 : downstream_rdy);

always @(posedge clk) begin
  if (rst)
    waiting <= 0;
  // downstream_inclk indicates that a word has been received, and
  // downstream_rdy will be deasserted until downstream is ready
  // for the next word
  else if (downstream_inclk)
    waiting <= 0;
  else if (upstream_readclk)
    waiting <= 1;
end

endmodule

// buffered version of stream_coord, ensures that a word is passed
// out immediately when downstream is ready
module stream_coord_buf #(
  parameter DATA_WIDTH = 1) (
```

```
  input clk, rst,
  input inclk, input [DATA_WIDTH-1:0] in,
  input in_done,
  input downstream_rdy,
  output outclk, output [DATA_WIDTH-1:0] out,
  output done,
  output upstream_readclk);

wire swb_empty;
stream_coord sc_inst(
  .clk(clk), .rst(rst),
  // if downstream (of stream_coord_buf) is ready, the buffer will be
  // cleared, so the buffer (which is downstream of stream_coord) is
  // ready
  .downstream_rdy(swb_empty || downstream_rdy),
  .downstream_inclk(inclk),
  .upstream_readclk(upstream_readclk));
// add a single bit for the done signal
single_word_buffer #(.DATA_WIDTH(DATA_WIDTH+1)) swb_inst(
  // clear the buffer when downstream is ready, since then outclk would
  // be asserted (unless rst or swb_empty is asserted, but in those
  // cases it would be safe to clear the buffer anyway)
  .clk(clk), .rst(rst), .clear(downstream_rdy),
  .inclk(inclk), .in({in, in_done}),
  .empty(swb_empty), .out({out, done}));
// only present data on out when downstream is ready
assign outclk = !rst && !swb_empty && downstream_rdy;

endmodule

// coordinated, buffered version of stream_unpack
module stream_unpack_coord_buf #(
  parameter S_LEN = 1,
  parameter L_LEN = 2) (
  input clk, rst, inclk,
  input [L_LEN-1:0] in,
  input in_done, downstream_rdy,
  output upstream_readclk, outclk,
  output [S_LEN-1:0] out,
  output done);

wire su_rdy, su_inclk, su_in_done;
wire [L_LEN-1:0] su_in;
stream_coord_buf #(.DATA_WIDTH(L_LEN)) su_scb_inst(
  .clk(clk), .rst(rst),
  .inclk(inclk), .in(in),
```

```
  .in_done(in_done), .downstream_rdy(su_rdy && downstream_rdy),
  .outclk(su_inclk), .out(su_in), .done(su_in_done),
  .upstream_readclk(upstream_readclk));
stream_unpack #(.S_LEN(S_LEN), .L_LEN(L_LEN)) su_inst (
  .clk(clk), .rst(rst),
  .inclk(su_inclk), .in(su_in), .in_done(su_in_done),
  .readclk(1'b1),
  .outclk(outclk), .out(out),
  .rdy(su_rdy), .done(done));

endmodule

// coordinated, buffered version of bytes_to_dibits
module bytes_to_dibits_coord_buf(
  input clk, rst, inclk,
  input [BYTE_LEN-1:0] in,
  input in_done, downstream_rdy,
  output upstream_readclk, outclk,
  output [1:0] out,
  output done);

`include "params.vh"

stream_unpack_coord_buf #(.S_LEN(2), .L_LEN(BYTE_LEN)) sucb_inst (
  .clk(clk), .rst(rst),
  .inclk(inclk), .in(in), .in_done(in_done),
  .downstream_rdy(downstream_rdy), .upstream_readclk(upstream_readclk),
  .outclk(outclk), .out(out), .done(done));

endmodule
```

A.4 Device Drivers

display_8hex.v (provided in labs):

```
`timescale 1ns / 1ps

// provided in previous labs
// displays 8 hex numbers on the 7 segment display
// designed by gim hom
module display_8hex(
    input clk,                // system clock
    input [31:0] data,        // 8 hex numbers, msb first
    output reg [6:0] seg,     // seven segment display output
    output reg [7:0] strobe   // digit strobe
    );
```

```
localparam bits = 13;
reg [bits:0] counter = 0;  // clear on power up

wire [6:0] segments[15:0]; // 16 7 bit memorys
assign segments[0]  = 7'b100_0000;
assign segments[1]  = 7'b111_1001;
assign segments[2]  = 7'b010_0100;
assign segments[3]  = 7'b011_0000;
assign segments[4]  = 7'b001_1001;
assign segments[5]  = 7'b001_0010;
assign segments[6]  = 7'b000_0010;
assign segments[7]  = 7'b111_1000;
assign segments[8]  = 7'b000_0000;
assign segments[9]  = 7'b001_1000;
assign segments[10] = 7'b000_1000;
assign segments[11] = 7'b000_0011;
assign segments[12] = 7'b010_0111;
assign segments[13] = 7'b010_0001;
assign segments[14] = 7'b000_0110;
assign segments[15] = 7'b000_1110;

// data and alt values being strobed in
reg [3:0] current_data;

always @(*) begin
case (counter[bits:bits-2])
    3'b000: begin
        current_data = data[31:28];
    end
    3'b001: begin
        current_data = data[27:24];
    end
    3'b010: begin
        current_data = data[23:20];
    end
    3'b011: begin
        current_data = data[19:16];
    end
    3'b100: begin
        current_data = data[15:12];
    end
    3'b101: begin
        current_data = data[11:8];
    end
    3'b110: begin
```

```
            current_data = data[7:4];
        end
        3'b111: begin
            current_data = data[3:0];
        end
    endcase
    end

    always @(posedge clk) begin
    counter <= counter + 1;
    seg <= segments[current_data];
    case (counter[bits:bits-2])
        3'b000: begin
            strobe <= 8'b0111_1111;
        end
        3'b001: begin
            strobe <= 8'b1011_1111;
        end
        3'b010: begin
            strobe <= 8'b1101_1111;
        end
        3'b011: begin
            strobe <= 8'b1110_1111;
        end
        3'b100: begin
            strobe <= 8'b1111_0111;
        end
        3'b101: begin
            strobe <= 8'b1111_1011;
        end
        3'b110: begin
            strobe <= 8'b1111_1101;
        end
        3'b111: begin
            strobe <= 8'b1111_1110;
        end
    endcase
    end

endmodule
```

xvga.v (modified from the one provided in labs):

```
`timescale 1ns / 1ps

// adapted from code provided for previous labs
module xvga(
```

```
  input clk,
  // vga_x and vga_y are only valid when blank is not asserted
  output [clog2(VGA_WIDTH)-1:0] vga_x,
  output [clog2(VGA_HEIGHT)-1:0] vga_y,
  output reg vsync, hsync,
  // vga_* versions of vsync and hsync account for polarity
  output vga_vsync, vga_hsync,
  output reg blank);

`include "params.vh"

// parameters from the VGA spec

// 800x600
localparam VGA_H_FRONT_PORCH = 56;
localparam VGA_H_SYNC = 120;
localparam VGA_H_BACK_PORCH = 64;
localparam VGA_V_FRONT_PORCH = 37;
localparam VGA_V_SYNC = 6;
localparam VGA_V_BACK_PORCH = 23;
localparam VGA_POLARITY = 0;

// 1024x768
// localparam VGA_H_FRONT_PORCH = 24;
// localparam VGA_H_SYNC = 136;
// localparam VGA_H_BACK_PORCH = 160;
// localparam VGA_V_FRONT_PORCH = 9;
// localparam VGA_V_SYNC = 6;
// localparam VGA_V_BACK_PORCH = 23;
// localparam VGA_POLARITY = 1;

// 640x480
// localparam VGA_H_FRONT_PORCH = 16;
// localparam VGA_H_SYNC = 96;
// localparam VGA_H_BACK_PORCH = 48;
// localparam VGA_V_FRONT_PORCH = 10;
// localparam VGA_V_SYNC = 2;
// localparam VGA_V_BACK_PORCH = 33;
// localparam VGA_POLARITY = 1;

// maximum values that hcount and vcount can take
localparam VGA_H_TOT = VGA_WIDTH +
  VGA_H_FRONT_PORCH + VGA_H_FRONT_PORCH + VGA_H_SYNC;
localparam VGA_V_TOT = VGA_HEIGHT +
  VGA_V_FRONT_PORCH + VGA_V_FRONT_PORCH + VGA_V_SYNC;
```

```
reg [clog2(VGA_H_TOT)-1:0] hcount;
reg [clog2(VGA_V_TOT)-1:0] vcount;
// no reason to keep all the information in hcount and vcount for outputs
assign vga_x = hcount[0+:clog2(VGA_HEIGHT)];
assign vga_y = vcount[0+:clog2(VGA_WIDTH)];

assign vga_hsync = hsync ^ VGA_POLARITY;
assign vga_vsync = vsync ^ VGA_POLARITY;

// horizontal: 1344 pixels total
// display 1024 pixels per line
reg hblank,vblank;
wire hsyncon,hsyncoff,hreset,hblankon;
assign hblankon = (hcount == VGA_WIDTH-1);
assign hsyncon = (hcount == VGA_WIDTH+VGA_H_FRONT_PORCH-1);
assign hsyncoff = (hcount == VGA_WIDTH+VGA_H_FRONT_PORCH+VGA_H_SYNC-1);
assign hreset = (hcount == VGA_H_TOT-1);

// vertical: 806 lines total
// display 768 lines
wire vsyncon,vsyncoff,vreset,vblankon;
assign vblankon = hreset & (vcount == VGA_HEIGHT-1);
assign vsyncon = hreset & (vcount == VGA_HEIGHT+VGA_V_FRONT_PORCH-1);
assign vsyncoff = hreset &
  (vcount == VGA_HEIGHT+VGA_V_FRONT_PORCH+VGA_V_SYNC-1);
assign vreset = hreset & (vcount == VGA_V_TOT-1);

// sync and blanking
wire next_hblank,next_vblank;
assign next_hblank = hreset ? 0 : hblankon ? 1 : hblank;
assign next_vblank = vreset ? 0 : vblankon ? 1 : vblank;
always @(posedge clk) begin
  hcount <= hreset ? 0 : hcount + 1;
  hblank <= next_hblank;
  hsync <= hsyncon ? 0 : hsyncoff ? 1 : hsync;  // active low

  vcount <= hreset ? (vreset ? 0 : vcount + 1) : vcount;
  vblank <= next_vblank;
  vsync <= vsyncon ? 0 : vsyncoff ? 1 : vsync;  // active low

  blank <= next_vblank | (next_hblank & ~hreset);
end

endmodule
```

memory.v:

```
// these are wrappers around the ip bram cores
// provides a readclk-outclk interface so that other modules don't need
// to be aware of the bram latency

module video_cache_ram_driver #(
  parameter RAM_SIZE = VIDEO_CACHE_RAM_SIZE,
  parameter READ_LATENCY = VIDEO_CACHE_RAM_LATENCY) (
  input clk, rst,
  input readclk, input [clog2(RAM_SIZE)-1:0] raddr,
  input we, input [clog2(RAM_SIZE)-1:0] waddr,
  input [COLOR_LEN-1:0] win,
  output outclk, output [COLOR_LEN-1:0] out
  );

`include "params.vh"

delay #(.DELAY_LEN(READ_LATENCY)) delay_inst(
  .clk(clk), .rst(rst), .in(readclk), .out(outclk));
video_cache_ram video_cache_ram_inst(
  .clka(clk), .wea(we),
  .addra(waddr), .dina(win),
  .clkb(clk), .addrb(raddr), .doutb(out));

endmodule

module packet_synth_rom_driver #(
  parameter RAM_SIZE = PACKET_SYNTH_ROM_SIZE,
  parameter READ_LATENCY = PACKET_SYNTH_ROM_LATENCY) (
  input clk, rst,
  input readclk, input [clog2(RAM_SIZE)-1:0] raddr,
  output outclk, output [BYTE_LEN-1:0] out
  );

`include "params.vh"

delay #(.DELAY_LEN(READ_LATENCY)) delay_inst(
  .clk(clk), .rst(rst), .in(readclk), .out(outclk));
packet_synth_rom packet_synth_rom_inst(
  .clka(clk),
  .addra(raddr), .douta(out));

endmodule

module packet_buffer_ram_driver #(
  parameter RAM_SIZE = PACKET_BUFFER_SIZE,
  parameter READ_LATENCY = PACKET_BUFFER_READ_LATENCY) (
```

```
  input clk, rst,
  input readclk, input [clog2(RAM_SIZE)-1:0] raddr,
  input we, input [clog2(RAM_SIZE)-1:0] waddr,
  input [BYTE_LEN-1:0] win,
  output outclk, output [BYTE_LEN-1:0] out
  );

`include "params.vh"

delay #(.DELAY_LEN(READ_LATENCY)) delay_inst(
  .clk(clk), .rst(rst), .in(readclk), .out(outclk));
packet_buffer_ram packet_buffer_ram_inst(
  .clka(clk), .wea(we),
  .addra(waddr), .dina(win),
  .clkb(clk), .addrb(raddr), .doutb(out));

endmodule
```

uart.v:

```
// fast receiving, designed for 12MBaud
// 120/50MHz clock boundary
module uart_rx_fast_driver #(
  // 120MBaud * 10 = 120MHz
  parameter CYCLES_PER_BIT = 10) (
  input clk, clk_120mhz, rst,
  input rxd,
  output [7:0] out, output outclk);

`include "params.vh"

// counts the number of consecutive zeroes to detect start bit
// the start bit is detected when rxd has been deasserted for
// CYCLES_PER_BIT/2 cycles
// this ensures that we read the subsequent bits in the middle of each bit
reg [clog2(CYCLES_PER_BIT/2)-1:0] start_bit_cnt = 0;

// shift register to convert bits into bytes
reg [BYTE_LEN-1:0] curr_byte_shifted;

// count the number of clock cycles since we read a bit
// when cycle_in_bit_cnt == CYCLES_PER_BIT-1, it's time to read
// the next bit
reg [clog2(CYCLES_PER_BIT)-1:0] cycle_in_bit_cnt = 0;
// number of bits read for the current byte
// allow bit_in_byte_cnt to go to BYTE_LEN + 2 to detect start/stop bit
// when zero, indicates that we are waiting to receive the start bit
```

```
reg [clog2(BYTE_LEN+2)-1:0] bit_in_byte_cnt = 0;

reg outclk_120mhz = 0;
reg [BYTE_LEN-1:0] out_120mhz;

// ip fifo to synchronize across clock boundaries
wire fifo_empty, fifo_rden;
assign fifo_rden = !fifo_empty;
byte_stream_fifo data_fifo(
  .rst(rst),
  .wr_clk(clk_120mhz), .rd_clk(clk),
  .din(out_120mhz), .wr_en(outclk_120mhz),
  .rd_en(fifo_rden), .dout(out),
  .empty(fifo_empty));
delay fifo_read_delay(
  .clk(clk), .rst(rst), .in(fifo_rden), .out(outclk));
wire rst_120mhz;
reset_stream_fifo reset_fifo_inst(
  .clka(clk), .clkb(clk_120mhz),
  .rsta(rst), .rstb(rst_120mhz));

always @(posedge clk_120mhz) begin
  if (rst_120mhz) begin
    bit_in_byte_cnt <= 0;
    start_bit_cnt <= 0;
    outclk_120mhz <= 0;
  end else begin
    if (rxd)
      start_bit_cnt <= 0;
    // stop incrementing when start_bit_cnt reaches its maximum value
    else if (start_bit_cnt != CYCLES_PER_BIT/2-1)
      start_bit_cnt <= start_bit_cnt + 1;

    if (bit_in_byte_cnt == 0) begin
      // wait for start bit
      if (start_bit_cnt == CYCLES_PER_BIT/2-1) begin
        curr_byte_shifted <= 0;
        bit_in_byte_cnt <= 1;
        cycle_in_bit_cnt <= 0;
      end
      outclk_120mhz <= 0;
    end else begin
      if (cycle_in_bit_cnt == CYCLES_PER_BIT-1) begin
        if (bit_in_byte_cnt == BYTE_LEN + 1) begin
          // rxd should be high at this point
          // but we can't do anything about it otherwise
```

```
          out_120mhz <= curr_byte_shifted;
          outclk_120mhz <= 1;
          bit_in_byte_cnt <= 0;
        end else begin
          bit_in_byte_cnt <= bit_in_byte_cnt + 1;
          // uart is little-endian, so shift in from the left
          curr_byte_shifted <=
            {rxd, curr_byte_shifted[1+:BYTE_LEN-1]};
        end
        cycle_in_bit_cnt <= 0;
      end else
        cycle_in_bit_cnt <= cycle_in_bit_cnt + 1;
    end
  end
end

endmodule

// expose a stream interface to request one byte at a time
module uart_tx_fast_stream_driver(
  input clk, clk_120mhz, rst, start,
  input inclk, input [7:0] in,
  output txd, output upstream_readclk);

wire driv_rdy;
uart_tx_fast_driver uart_driv_inst(
  .clk(clk), .clk_120mhz(clk_120mhz), .rst(rst),
  .inclk(inclk), .in(in), .txd(txd), .rdy(driv_rdy));
stream_coord sc_inst(
  .clk(clk), .rst(rst || start),
  .downstream_rdy(driv_rdy), .downstream_inclk(inclk),
  .upstream_readclk(upstream_readclk));

endmodule

// fast transmitting, designed for 12MBaud
// 120/50MHz clock boundary
module uart_tx_fast_driver #(
  // 120MBaud * 10 = 120MHz
  parameter CYCLES_PER_BIT = 10) (
  input clk, clk_120mhz, rst,
  input inclk, input [7:0] in,
  output txd,
  // rdy is asserted to request for a new byte to transmit
  output rdy);
```

```
`include "params.vh"

// two extra bits for start/stop bits
// initialize to all ones to indicate that nothing is being transmitted
reg [BYTE_LEN+2-1:0] curr_byte_shifted = ~0;
assign txd = curr_byte_shifted[0];

// number of bits in the current byte left to transmit
// allow bits_left_cnt to go to BYTE_LEN + 2 to send start/stop bits
reg [clog2(BYTE_LEN+2)-1:0] bits_left_cnt = 0;

wire [BYTE_LEN-1:0] in_120mhz;
wire inclk_120mhz;

wire rst_120mhz;
reset_stream_fifo reset_fifo_inst(
  .clka(clk), .clkb(clk_120mhz),
  .rsta(rst), .rstb(rst_120mhz));

// not actually a clock
// used as a timer to ensure that bits are transmitted at the correct rate
// for uart
wire tx_clk;
clock_divider #(.PULSE_PERIOD(CYCLES_PER_BIT)) tx_clock_divider(
  .clk(clk_120mhz), .rst(rst_120mhz), .en(1'b1), .out(tx_clk));

// ip fifo to synchronize across clock boundary
wire fifo_empty, fifo_full, fifo_rden;
assign fifo_rden = !fifo_empty && tx_clk && bits_left_cnt == 0;
byte_stream_fifo data_fifo(
  .rst(rst),
  .wr_clk(clk), .rd_clk(clk_120mhz),
  .din(in), .wr_en(inclk),
  .rd_en(fifo_rden), .dout(in_120mhz),
  .full(fifo_full), .empty(fifo_empty));
delay fifo_read_delay(
  .clk(clk_120mhz), .rst(rst_120mhz),
  .in(fifo_rden), .out(inclk_120mhz));
// delay tx_clk by one cycle to account for fifo read time
wire tx_clk_delayed;
delay tx_clk_delay(
  .clk(clk_120mhz), .rst(rst_120mhz),
  .in(tx_clk), .out(tx_clk_delayed));

assign rdy = !fifo_full;
```

```
always @(posedge clk_120mhz) begin
  if (rst_120mhz) begin
    bits_left_cnt <= 0;
    curr_byte_shifted <= ~0;
  end else if (tx_clk_delayed) begin
    if (bits_left_cnt == 0 && inclk_120mhz) begin
      bits_left_cnt <= BYTE_LEN + 2 - 1;
      curr_byte_shifted <= {1'b1, in_120mhz, 1'b0};
    end else begin
      if (bits_left_cnt != 0)
        bits_left_cnt <= bits_left_cnt - 1;
      curr_byte_shifted <= {1'b1, curr_byte_shifted[1+:BYTE_LEN+1]};
    end
  end
end

endmodule
```

rmii.v:

```
// receives ethernet frames from the rmii interface
// ethernet frames start with a preamble of alternating ones and zeroes,
// ending with two ones
// frames presented on out will not include the preamble
module rmii_driver(
  input clk, rst,
  // rxd and crsdv double as configuration straps when resetting the phy
  // we use rxerer and intn only for reset configuration
  inout crsdv_in,
  inout [1:0] rxd_in,
  output rxerr, intn,
  output reg rstn = 0,
  output reg [1:0] out = 0,
  // done is pulsed at the same time as the last dibit of a full packet,
  // not on the last byte presented on out, giving a chance for the
  // driver to throw an error if the crc check fails
  output reg outclk = 0, output done);

`include "params.vh"

// should have 25ms delay from power supply up before
// nRST assertion, but we assume that power supplies have
// long been set up already
// according to spec: need 100us before nRST deassertion
// and 800ns afterwards
localparam RESET_BEFORE = 5000;
localparam RESET_AFTER = 40;
```

```
localparam RESET_SEQUENCE_LEN = RESET_BEFORE + RESET_AFTER;
reg [clog2(RESET_SEQUENCE_LEN)-1:0] rst_cnt = 0;
wire rst_done;
assign rst_done = rst_cnt == RESET_SEQUENCE_LEN;

// RESET CONFIGURATION

// 100Base-TX Full Duplex, auto-negotiation disabled,
// CRS active during receive
localparam DEFAULT_MODE = 3'b011;
// PHY address, leave at zero
localparam DEFAULT_PHYAD = 0;
// REF_CLK in (we control the clocking)
localparam DEFAULT_NINTSEL = 1;

assign crsdv_in = rst_done ? 1'bz : DEFAULT_MODE[2];
assign rxd_in = rst_done ? 2'bzz : DEFAULT_MODE[1:0];
assign rxerr = rst_done ? 1'bz : DEFAULT_PHYAD;
assign intn = rst_done ? 1'bz : DEFAULT_NINTSEL;

wire crsdv;
wire [1:0] rxd;

// assertion of CRS_DV is async wrt the REF_CLK, so synchronization needed
delay #(.DELAY_LEN(SYNC_DELAY_LEN-1)) crsdv_sync(
  .clk(clk), .rst(rst), .in(crsdv_in), .out(crsdv));
// delay rxd to be in time with crsdv
delay #(.DELAY_LEN(SYNC_DELAY_LEN-1), .DATA_WIDTH(2)) rxd_sync(
  .clk(clk), .rst(rst), .in(rxd_in), .out(rxd));

localparam STATE_IDLE = 0;
// crsdv has been asserted, waiting for data to start streaming in
localparam STATE_WAITING = 1;
// reading preamble
localparam STATE_PREAMBLE = 2;
localparam STATE_RECEIVING = 3;
reg [1:0] state = STATE_IDLE;

// store previous crsdv to check for toggling
// a toggling crsdv indicates that dv is asserted but not crs
reg prev_crsdv;
always @(posedge clk) begin
  if (rst_done)
    prev_crsdv <= crsdv;
end
```

```
// distinguish crs and dv signals
// only to be used when state == STATE_RECEIVING
// this will give the wrong value for crs on the rising edge of crsdv,
// but this is fine since we aren't using crs
// this will give the wrong value for dv on the falling edge of crsdv,
// but this is fine since systems based on ethernet should be robust to
// zeroes padded to the end of a frame
wire crsdv_toggling, crs, dv;
assign crsdv_toggling = prev_crsdv != crsdv;
assign crs = crsdv_toggling ? 0 : crsdv;
assign dv = crsdv_toggling ? 1 : crsdv;
assign done = (state == STATE_RECEIVING) && !dv;

always @(posedge clk) begin
  if (rst) begin
    rst_cnt <= 0;
    rstn <= 0;
    state <= STATE_IDLE;
    out <= 0;
    outclk <= 0;
  end else if (~rst_done) begin
    if (rst_cnt == RESET_BEFORE - 1)
      rstn <= 1;
    rst_cnt <= rst_cnt + 1;
  end else case(state)
  STATE_IDLE:
    if (crsdv)
      state <= STATE_WAITING;
  STATE_WAITING:
    // drop back to idle if crsdv stops being asserted
    if (!crsdv)
      state <= STATE_IDLE;
    else if (rxd == 2'b01) begin
      state <= STATE_PREAMBLE;
    end
  STATE_PREAMBLE:
    // drop back to idle if crsdv stops being asserted
    if (!crsdv)
      state <= STATE_IDLE;
    else if (rxd == 2'b11) begin
      state <= STATE_RECEIVING;
    end
  STATE_RECEIVING:
    if (!dv) begin
      state <= STATE_IDLE;
      outclk <= 0;
```

```
    end else begin
      outclk <= 1;
      out <= rxd;
    end
  endcase
end

endmodule
```

A.5 Networking

ethernet.v:

```
// generate an ethernet frame on the level of bytes
// CRC needs to be generated on the level of dibits,
// which is handled in eth_tx
// expects payload and rom delay of PACKET_SYNTH_ROM_DELAY
// exposes readclk to out latency of PACKET_SYNTH_ROM_DELAY
module eth_body_tx #(
  // size of rom holding constants, e.g. mac addresses
  parameter RAM_SIZE = PACKET_SYNTH_ROM_SIZE) (
  input clk, rst, start, in_done,
  input inclk, input [BYTE_LEN-1:0] in,
  input readclk,
  input ram_outclk, input [BYTE_LEN-1:0] ram_out,
  output ram_readclk, output reg [clog2(RAM_SIZE)-1:0] ram_raddr,
  output outclk, output [BYTE_LEN-1:0] out,
  output upstream_readclk, done);

‘include "packet_synth_rom_layout.vh"
‘include "networking.vh"

localparam STATE_IDLE = 0;
localparam STATE_MAC_DST = 1;
localparam STATE_MAC_SRC = 2;
localparam STATE_ETHERTYPE = 3;
localparam STATE_PAYLOAD = 4;

// "pre-delayed" versions of output data generated internally by this
// module, to be combined with ram and upstream read results
wire outclk_pd;
wire [BYTE_LEN-1:0] out_pd;
// out_premux is out, but may be overwritten with ram read result
wire [BYTE_LEN-1:0] out_premux;
// if we requested a payload (upstream) read, then we want to use the
// payload read result instead
```

```
// if we requested a read from ram, then we want to use the ram read
// result instead
assign out =
  inclk ? in :
  ram_outclk ? ram_out :
  out_premux;

// outclk for data that is generated internally by this module
wire outclk_internal;
assign outclk = outclk_internal || ram_outclk || inclk;
delay #(.DELAY_LEN(PACKET_SYNTH_ROM_LATENCY)) outclk_delay(
  .clk(clk), .rst(rst || start),
  .in(outclk_pd), .out(outclk_internal));
delay #(.DELAY_LEN(PACKET_SYNTH_ROM_LATENCY),
  .DATA_WIDTH(BYTE_LEN)) out_delay(
  .clk(clk), .rst(rst || start), .in(out_pd), .out(out_premux));
assign done = in_done;

reg [2:0] state = STATE_IDLE;
// number of bytes transmitted for the current stage
reg [2:0] cnt;

// multiplex different sources of data
assign upstream_readclk = state == STATE_PAYLOAD && readclk;
assign ram_readclk = (
  (state == STATE_MAC_DST) ||
  (state == STATE_MAC_SRC) ||
  (state == STATE_ETHERTYPE)) &&
  readclk;
// these signals would be used if we generated any data internally
// we don't in this module, but keep this anyway to maintain a consistent
// implementation across different networking modules
assign outclk_pd = 0;
assign out_pd = 0;

always @(posedge clk) begin
  if (rst)
    state <= STATE_IDLE;
  else if (start) begin
    state <= STATE_MAC_DST;
    ram_raddr <= MAC_RECV_OFF;
    cnt <= 0;
  end else if (outclk) begin
    if (state == STATE_MAC_DST && cnt == ETH_MAC_LEN-1) begin
      cnt <= 0;
      state <= STATE_MAC_SRC;
```

```
      ram_raddr <= MAC_SEND_OFF;
    end else if (state == STATE_MAC_SRC && cnt == ETH_MAC_LEN-1) begin
      cnt <= 0;
      state <= STATE_ETHERTYPE;
      ram_raddr <= ETHERTYPE_FFCP_OFF;
    end else if (state == STATE_ETHERTYPE &&
        cnt == ETH_ETHERTYPE_LEN-1) begin
      state <= STATE_PAYLOAD;
    end else if (state == STATE_PAYLOAD && in_done) begin
      cnt <= 0;
      state <= STATE_IDLE;
    end else begin
      cnt <= cnt + 1;
      ram_raddr <= ram_raddr + 1;
    end
  end
end

endmodule

module crc32(
  input clk, rst,
  // allow application to reuse out as a shift register
  // when shift is asserted, shift the data on out by 2 bits
  input shift, inclk, input [1:0] in,
  output [31:0] out);

localparam CRC_INIT = 32'hffffffff;
// polynomial is reflected since we operate in
// least-significant-bit first for simplicity
localparam CRC_POLY = 32'hedb88320;

reg [31:0] curr;
// leave at least significant bit first, since crc will be
// shifted out from the end
assign out = ~curr;

// optimized dibit CRC step
// step1: XOR in both inputs at once
// step2: first division by poly
// step3: second division by poly
wire [31:0] step1, step2, step3;
assign step1 = {curr[2+:30], curr[0+:2] ^ in};
assign step2 = step1[1+:31] ^ (step1[0] ? CRC_POLY : 0);
assign step3 = step2[1+:31] ^ (step2[0] ? CRC_POLY : 0);
```

```
always @(posedge clk) begin
  if (rst)
    curr <= CRC_INIT;
  else if (shift)
    curr <= {2'b11, curr[2+:30]};
  else if (inclk)
    curr <= step3;
end

endmodule

// creates continuous dibit stream for an ethernet frame
module eth_tx #(
  // size of rom holding constants, e.g. mac addresses
  parameter RAM_SIZE = PACKET_SYNTH_ROM_SIZE) (
  input clk, rst, start, in_done,
  input inclk, input [BYTE_LEN-1:0] in,
  input ram_outclk, input [BYTE_LEN-1:0] ram_out,
  output ram_readclk,
  output [clog2(RAM_SIZE)-1:0] ram_raddr,
  output outclk, output [1:0] out,
  output upstream_readclk, done);

`include "networking.vh"

// main_rdy indicates that main processing is ready for the frame body
wire main_rdy;
wire eth_tx_readclk, eth_tx_outclk, eth_tx_done;
wire [BYTE_LEN-1:0] eth_tx_out;
wire btd_outclk, btd_done;
wire [1:0] btd_out;
wire crc_rst, crc_shift;
wire [31:0] crc_out;
eth_body_tx eth_tx_inst(
  .clk(clk), .rst(rst), .start(start), .in_done(in_done),
  .inclk(inclk), .in(in), .readclk(eth_tx_readclk),
  .ram_outclk(ram_outclk), .ram_out(ram_out),
  .ram_readclk(ram_readclk), .ram_raddr(ram_raddr),
  .outclk(eth_tx_outclk), .out(eth_tx_out),
  .upstream_readclk(upstream_readclk), .done(eth_tx_done));
bytes_to_dibits_coord_buf btd_inst(
  .clk(clk), .rst(rst || start),
  .inclk(eth_tx_outclk), .in(eth_tx_out), .in_done(eth_tx_done),
  .downstream_rdy(main_rdy), .upstream_readclk(eth_tx_readclk),
  .outclk(btd_outclk), .out(btd_out), .done(btd_done));
crc32 crc32_inst(
```

```
  .clk(clk), .rst(crc_rst), .shift(crc_shift),
  .inclk(btd_outclk), .in(btd_out), .out(crc_out));

localparam STATE_IDLE = 0;
// we need to transmit the preamble
localparam STATE_PREAMBLE = 1;
localparam STATE_BODY = 2;
localparam STATE_CRC = 3;
// "transmit" an inter-packet gap to ensure that packets are always
// sufficiently spaced apart
localparam STATE_GAP = 4;

reg [2:0] state = STATE_IDLE;
// number of dibits transmitted for the current stage
reg [5:0] cnt;

// sfd indicates that we have reached the end of the preamble
wire sfd, crc_done, gap_done;
// multiply by 4 since we're measuring in dibits
assign sfd = (state == STATE_PREAMBLE) && (cnt == ETH_PREAMBLE_LEN*4-1);
assign crc_done = (state == STATE_CRC) && (cnt == ETH_CRC_LEN*4-1);
assign crc_shift = state == STATE_CRC;
assign gap_done = (state == STATE_GAP) && (cnt == ETH_GAP_LEN*4-1);
assign out =
  (state == STATE_BODY) ? btd_out :
  sfd ? 2'b11 :
  (state == STATE_PREAMBLE) ? 2'b01 :
  crc_out[0+:2];
// reset the crc module when we enter the crc-protected region, which
// starts when the preamble ends
assign crc_rst = rst || sfd;
assign main_rdy = state == STATE_BODY;
assign done = gap_done;
assign outclk = btd_outclk ||
  (state == STATE_PREAMBLE) ||
  (state == STATE_CRC);

always @(posedge clk) begin
  if (rst)
    state <= STATE_IDLE;
  else if (start) begin
    cnt <= 0;
    state <= STATE_PREAMBLE;
  end else if (sfd)
    state <= STATE_BODY;
  else if (state == STATE_BODY && btd_done) begin
```

```
    state <= STATE_CRC;
    cnt <= 0;
  end else if (crc_done) begin
    state <= STATE_GAP;
    cnt <= 0;
  end else if (gap_done)
    state <= STATE_IDLE;
  else
    cnt <= cnt + 1;
end

endmodule

module eth_rx(
  input clk, rst, inclk,
  input [1:0] in, input in_done,
  // downstream_done must be asserted on the same cycle as outclk
  // on the last byte of payload processing for correct error detection
  input downstream_done,
  output outclk, output [BYTE_LEN-1:0] out,
  output ethertype_outclk,
  output [ETH_ETHERTYPE_LEN*BYTE_LEN-1:0] ethertype_out,
  // done is pulsed at the same time as the last dibit of a full packet,
  // not on the last byte presented on out, giving a chance for the
  // module to throw an error if the crc check fails
  // done is only pulsed when a complete frame has been received
  // without errors
  output err, done);

`include "networking.vh"

wire frame_rst;
// reset everything not just on rst but also when the flow of data breaks
assign frame_rst = rst || !inclk;

wire dtb_outclk, dtb_done;
wire [BYTE_LEN-1:0] dtb_out;
dibits_to_bytes dtb_inst(
  .clk(clk), .rst(frame_rst),
  .inclk(inclk), .in(in), .in_done(in_done),
  .outclk(dtb_outclk), .out(dtb_out), .done(dtb_done));
wire crc_shift;
wire [31:0] crc_out;
crc32 crc32_inst(
  .clk(clk), .rst(frame_rst), .shift(crc_shift),
  .inclk(inclk), .in(in), .out(crc_out));
```

```
localparam STATE_MAC_DST = 0;
localparam STATE_MAC_SRC = 1;
localparam STATE_ETHERTYPE = 2;
localparam STATE_PAYLOAD = 3;
localparam STATE_CRC = 4;
localparam STATE_DONE = 5;

reg [2:0] state = STATE_MAC_DST;
// records number of bytes received for the current stage
reg [2:0] cnt = 0;
// idle indicates that we're not currently processing a frame
reg idle = 1;
// throw an error if the flow of data breaks while we're processing a frame
assign err = (!idle && !inclk) ||
  // we should not be receiving additional data after the crc
  (state == STATE_DONE && inclk && in != 2'b00) ||
  // crc check
  (state == STATE_CRC && inclk && in != crc_out[1:0]);

assign out = dtb_out;
assign outclk = state == STATE_PAYLOAD && dtb_outclk;
wire ethertype_done;
assign ethertype_done =
  state == STATE_ETHERTYPE && cnt == ETH_ETHERTYPE_LEN-1;
wire crc_done;
assign crc_done =
  state == STATE_CRC && cnt == ETH_CRC_LEN-1;
// a crc error may be thrown on the last dibit, so check for that
// before asserting done
assign done = dtb_outclk && crc_done && !err;

// holds the shifted ethertype, so that the ethertype can be transmitted
// one byte at a time
reg [(ETH_ETHERTYPE_LEN-1)*BYTE_LEN-1:0] ethertype_shifted;
assign ethertype_outclk = dtb_outclk && ethertype_done;
assign ethertype_out = {ethertype_shifted, dtb_out};

assign crc_shift = inclk && state == STATE_CRC;

always @(posedge clk) begin
  if (frame_rst) begin
    state <= STATE_MAC_DST;
    cnt <= 0;
    idle <= 1;
  // stop processing packets immediately when an error is thrown
```

```
  end else if (err)
    state <= STATE_DONE;
  else if (dtb_outclk) begin
    if (state == STATE_MAC_DST && cnt == ETH_MAC_LEN-1) begin
      state <= STATE_MAC_SRC;
      cnt <= 0;
    end else if (state == STATE_MAC_SRC && cnt == ETH_MAC_LEN-1) begin
      state <= STATE_ETHERTYPE;
      cnt <= 0;
    end else if (ethertype_done)
      state <= STATE_PAYLOAD;
    // the length of the payload is determined by the downstream
    // module, as is standard in networking over ethernet
    else if (state == STATE_PAYLOAD && downstream_done) begin
      state <= STATE_CRC;
      cnt <= 0;
    end else if (crc_done) begin
      state <= STATE_DONE;
      idle <= 1;
    end else if (state != STATE_DONE) begin
      idle <= 0;
      cnt <= cnt + 1;
      ethertype_shifted <=
        ethertype_out[0+:(ETH_ETHERTYPE_LEN-1)*BYTE_LEN];
    end
  end
end

endmodule
```

fgp.v:

```
// FGP: FPGA Graphics Protocol
// simple invented-here DMA protocol used to transmit graphics information
// sits just above the ethernet layer
// should be encrypted in actual packet
// format: [ offset (1 byte) | data (768 bytes) ]
// 768 bytes = 512 colors

// produces outputs with a latency of LATENCY
// essentially the same structure as eth_tx
module fgp_tx #(
  parameter LATENCY = PACKET_SYNTH_ROM_LATENCY) (
  input clk, rst, start, in_done,
  input inclk, input [BYTE_LEN-1:0] in,
  input [BYTE_LEN-1:0] offset,
  input readclk,
```

```
  output outclk, output [BYTE_LEN-1:0] out,
  output upstream_readclk, done);

`include "networking.vh"

reg [BYTE_LEN-1:0] offset_buf;

wire outclk_pd;
wire [BYTE_LEN-1:0] out_pd, out_premux;
assign out =
  inclk ? in :
  out_premux;
wire outclk_internal;
assign outclk = outclk_internal || inclk;
delay #(.DELAY_LEN(LATENCY)) outclk_delay(
  .clk(clk), .rst(rst || start),
  .in(outclk_pd), .out(outclk_internal));
delay #(.DELAY_LEN(LATENCY),
  .DATA_WIDTH(BYTE_LEN)) out_delay(
  .clk(clk), .rst(rst || start), .in(out_pd), .out(out_premux));
assign done = in_done;

localparam STATE_OFFSET = 0;
localparam STATE_DATA = 1;

reg [0:0] state = STATE_OFFSET;
reg [9:0] cnt = 0;

assign upstream_readclk = (state == STATE_DATA) && readclk;
assign outclk_pd =
  (state == STATE_OFFSET) &&
  readclk;
assign out_pd =
  (state == STATE_OFFSET) ? offset_buf : 0;

always @(posedge clk) begin
  if (rst || start) begin
    state <= STATE_OFFSET;
    offset_buf <= offset;
    cnt <= 0;
  end else if (readclk) begin
    if (state == STATE_OFFSET && cnt == FGP_OFFSET_LEN-1) begin
      state <= STATE_DATA;
      cnt <= 0;
    end else
      cnt <= cnt + 1;
```

```
  end
end

endmodule

// parser drives ram directly for now, no error detection
module fgp_rx(
  input clk, rst, inclk,
  input [BYTE_LEN-1:0] in,
  output done,
  output offset_outclk,
  output [BYTE_LEN-1:0] offset_out,
  output outclk, output [BYTE_LEN-1:0] out);

`include "networking.vh"

localparam STATE_OFFSET = 0;
localparam STATE_DATA = 1;

reg [0:0] state = STATE_OFFSET;
reg [9:0] cnt = 0;

assign done = inclk && state == STATE_DATA && cnt == FGP_DATA_LEN-1;
assign offset_outclk = inclk && state == STATE_OFFSET &&
  cnt == FGP_OFFSET_LEN-1;
assign offset_out = in;
assign outclk = inclk && state == STATE_DATA;
assign out = in;

always @(posedge clk) begin
  if (rst || done) begin
    state <= STATE_OFFSET;
    cnt <= 0;
  end else if (inclk) begin
    if (state == STATE_OFFSET && cnt == FGP_OFFSET_LEN-1) begin
      state <= STATE_DATA;
      cnt <= 0;
    end else
      cnt <= cnt + 1;
  end
end

endmodule
```

ffcp.v:

```
// FFCP: FGPA Flow Control Protocol
```

```
// simple invented-here protocol for flow control
// format: [ type (2 bits) | index (6 bits) | FGP data (769 bytes) ]
// type is 0 (syn), 1 (msg) or 2 (ack)
// The index works like the sequence number in TCP
// FFCP's flow control is a very simplified version of TCP's, where
// the window size is fixed and data flows in only one direction
// FGP data is omitted in ack

// produces outputs with a latency of LATENCY
// essentially the same structure as eth_tx
module ffcp_tx #(
  parameter LATENCY = PACKET_SYNTH_ROM_LATENCY) (
  input clk, rst, start, in_done,
  input inclk, input [BYTE_LEN-1:0] in,
  // ffcp_type/index are valid only when start is asserted
  // these are the values written to the FFCP header
  input [FFCP_TYPE_LEN-1:0] ffcp_type,
  input [FFCP_INDEX_LEN-1:0] ffcp_index,
  input readclk,
  output outclk, output [BYTE_LEN-1:0] out,
  output upstream_readclk, done);

`include "networking.vh"

// the metadata is the type and index combined into a single byte
reg [BYTE_LEN-1:0] metadata_buf;
// the type field in metadata_buf
wire [FFCP_TYPE_LEN-1:0] metadata_buf_type;
assign metadata_buf_type = metadata_buf[FFCP_INDEX_LEN+:FFCP_TYPE_LEN];
wire is_ack;
assign is_ack = metadata_buf_type == FFCP_TYPE_ACK;

wire outclk_pd;
wire [BYTE_LEN-1:0] out_pd, out_premux;
assign out =
  inclk ? in :
  out_premux;
wire outclk_internal;
assign outclk = outclk_internal || inclk;
delay #(.DELAY_LEN(LATENCY)) outclk_delay(
  .clk(clk), .rst(rst || start),
  .in(outclk_pd), .out(outclk_internal));
delay #(.DELAY_LEN(LATENCY),
  .DATA_WIDTH(BYTE_LEN)) out_delay(
  .clk(clk), .rst(rst || start), .in(out_pd), .out(out_premux));
```

```
localparam STATE_METADATA = 0;
localparam STATE_DATA = 1;

reg [0:0] state = STATE_METADATA;
reg [9:0] cnt = 0;

wire metadata_done;
assign metadata_done =
  state == STATE_METADATA && cnt == FFCP_METADATA_LEN-1;
// if we're transmitting an ack, then skip the entire payload stage
// by asserting done
wire ack_done;
delay #(.DELAY_LEN(LATENCY)) done_delay(
  .clk(clk), .rst(rst || start),
  .in(readclk && metadata_done && is_ack), .out(ack_done));
assign done = in_done || ack_done;

assign upstream_readclk = (state == STATE_DATA) && readclk;
assign outclk_pd = (state == STATE_METADATA) && readclk;
assign out_pd =
  (state == STATE_METADATA) ? metadata_buf : 0;

always @(posedge clk) begin
  if (rst || start) begin
    state <= STATE_METADATA;
    metadata_buf <= {ffcp_type, ffcp_index};
    cnt <= 0;
  end else if (readclk) begin
    if (metadata_done) begin
      state <= STATE_DATA;
      cnt <= 0;
    end else
      cnt <= cnt + 1;
  end
end

endmodule

module ffcp_rx(
  input clk, rst, inclk,
  input [BYTE_LEN-1:0] in,
  output done,
  // ffcp_type/index valid only when metadata_outclk is asserted
  output metadata_outclk,
  output [FFCP_TYPE_LEN-1:0] ffcp_type,
  output [FFCP_INDEX_LEN-1:0] ffcp_index,
```

```
  output outclk, output [BYTE_LEN-1:0] out);

`include "networking.vh"

localparam STATE_METADATA = 0;
localparam STATE_DATA = 1;

reg [0:0] state = STATE_METADATA;
reg [9:0] cnt = 0;

wire metadata_done;
assign metadata_done =
  state == STATE_METADATA && cnt == FFCP_METADATA_LEN-1;
assign done = inclk && (
  (metadata_done && in[FFCP_INDEX_LEN+:FFCP_TYPE_LEN] == FFCP_TYPE_ACK) ||
  (state == STATE_DATA && cnt == FFCP_DATA_LEN-1));
assign metadata_outclk = inclk && metadata_done;
assign ffcp_type = in[FFCP_INDEX_LEN+:FFCP_TYPE_LEN];
assign ffcp_index = in[0+:FFCP_INDEX_LEN];
assign outclk = inclk && state == STATE_DATA;
assign out = in;

always @(posedge clk) begin
  if (rst || done) begin
    state <= STATE_METADATA;
    cnt <= 0;
  end else if (inclk) begin
    if (metadata_done) begin
      state <= STATE_DATA;
      cnt <= 0;
    end else
      cnt <= cnt + 1;
  end
end

endmodule

// manages flow control for the receiving end
// the index used in the FFCP header in this case is the same as the
// offset where the data lives in the packet buffer queue
// there are two downstreams here -- the packet transmission pipeline for
// acks, and the FGP DMA pipeline for commits

// specifically, when a packet is received, a bit is set indicating that
// it has been received; if it is at the head of the receive window,
// the packet is committed to the FGP DMA pipeline; when the commit is
```

```
// done, the window is advanced, and an ack is sent out if there are
// no more packets to commit

module ffcp_rx_server(
  // syn indicates that a syn is received
  input clk, rst, syn,
  // inclk indicates that a packet has been received with a sequence
  // number (FFCP index field) given by in_index
  input inclk, input [FFCP_INDEX_LEN-1:0] in_index,
  // downstream_done indicates that downstream has finished sending an
  // ack, and we can send another one
  input downstream_done,
  // commit_done indicates that downstream has finished committing a
  // packet, and we can commit another one
  input commit_done,
  // when commit is asserted, downstream processes the packet at the
  // index given by commit_index as part of the data stream
  // packets should be committed in order, except in the case of syn
  output commit, output [FFCP_INDEX_LEN-1:0] commit_index,
  // outclk indicates that we should send an ack for the index given by
  // out_index
  output outclk, output [FFCP_INDEX_LEN-1:0] out_index);

`include "networking.vh"

// bit vector recording whether we have received the packet with each
// sequence number in the window
reg received[FFCP_BUFFER_LEN-1:0];
// used in the reset procedure, which clears the received vector
reg [clog2(FFCP_BUFFER_LEN)-1:0] rst_cnt = 0;
// the start of the receive window
reg [clog2(FFCP_BUFFER_LEN)-1:0] queue_head;
wire curr_received;
assign curr_received = received[queue_head];
wire rst_done;
assign rst_done = rst_cnt == FFCP_BUFFER_LEN-1;

reg downstream_rdy = 1;
always @(posedge clk) begin
  if (rst)
    downstream_rdy <= 1;
  // if we send an ack with outclk, downstream won't be ready until
  // it asserts done
  else if (outclk)
    downstream_rdy <= 0;
  else if (downstream_done)
```

```
    downstream_rdy <= 1;
end

reg commit_rdy = 1;
always @(posedge clk) begin
  if (rst)
    commit_rdy <= 1;
  // if we request a commit, downstream won't be ready until
  // it asserts done
  else if (commit)
    commit_rdy <= 0;
  else if (commit_done)
    commit_rdy <= 1;
end

// ignore all messages other than those in receive window
// be careful of wraparound
wire ignore, receiving;
// offset in the receive window of the incoming packet
// use an explicitly declared net to truncate before comparison,
// so that it will always be positive (this is a circular buffer,
// so a negative offset is equivalent to a large positive one)
wire [clog2(FFCP_BUFFER_LEN)-1:0] in_index_head_off;
assign in_index_head_off = in_index - queue_head;
assign ignore = in_index_head_off >= FFCP_WINDOW_LEN;
assign receiving = inclk && !ignore;

// ack_buf indicates that we should send an ack as soon as we get
// the chance to
// try to ack as many indices as possible, so wait until the queue head
// has advanced past all received packets before acking
// also wait until self and downstream are ready
// also wait until any pending commits have finished, because the window
// is only advanced after a commit has completed, but curr_received
// is not asserted in the interim
reg ack_buf = 0;
assign outclk = !curr_received && commit_rdy &&
  downstream_rdy && ack_buf &&
  !rst && !syn && rst_done && !receiving;
// ack index should be the first index that we have not yet received
// when curr_received is not asserted, this is just the head of the window
assign out_index = queue_head;
// only commit when the packet at the head of the window is received
// wait until self and downstream are ready
assign commit = !outclk && curr_received && commit_rdy &&
  !rst && !syn && rst_done && !receiving;
```

```
assign commit_index = queue_head;

always @(posedge clk) begin
  if (rst || syn) begin
    queue_head <= 0;
    rst_cnt <= 0;
    ack_buf <= syn;
    // clear this now so we don't need to waste a rst_cnt bit for
    // an extra clear cycle
    received[FFCP_BUFFER_LEN-1] <= 0;
  end else if (!rst_done) begin
    // if ack_buf is set, then the reset was because of a syn, so
    // the first packet has been received
    received[rst_cnt] <= (ack_buf && rst_cnt == 0) ? 1 : 0;
    rst_cnt <= rst_cnt + 1;
  end else begin
    if (commit_done)
      queue_head <= queue_head + 1;
    if (receiving)
      received[in_index] <= 1;
    else if (outclk)
      // outclk indicates that an ack has been transmitted, so
      // clear ack_buf
      ack_buf <= 0;
    else if (commit) begin
      // if we commit a packet, then we should ack it, but only
      // after the commit is done and the window is advanced
      ack_buf <= 1;
      received[queue_head] <= 0;
    end
  end
end

endmodule

// the ffcp_queue manages the packet buffer (PB), intended for use
// with ffcp_tx_server or ffcp_rx_server
// we only need to be able to advance the head/tail by one index each time,
// or overwrite the head
module ffcp_queue(
  input clk, rst,
  input advance_head, advance_tail,
  // inclk indicates that we should overwrite the head
  // in_head is what we should overwrite the head to
  input inclk, input [clog2(PB_QUEUE_LEN)-1:0] in_head,
  output almost_full,
```

```
  output reg [clog2(PB_QUEUE_LEN)-1:0] head, tail);

`include "networking.vh"

wire [clog2(PB_QUEUE_LEN)-1:0] space_used = tail - head;
assign almost_full = space_used >= PB_QUEUE_ALMOST_FULL_THRES;

always @(posedge clk) begin
  if (rst) begin
    tail <= 0;
    head <= 0;
  end else begin
    if (inclk)
      head <= in_head;
    else if (advance_head)
      head <= head + 1;
    if (advance_tail)
      tail <= tail + 1;
  end
end

endmodule

// keeps track of acknowledgement from receiving FPGA
// ensures that packets only packets up to WINDOW_LEN after the last
// ack are transmitted
// cycles through the transmit window every RESEND_TIMEOUT if no acks
// are received
// starts an entirely new stream if no acks have been received for
// RESYN_TIMEOUT
module ffcp_tx_server(
  input clk, rst,
  // the packet buffer queue head follows the ffcp_tx_server's head,
  // but indexes into the packet buffer queue instead of the
  // transmit sequence numbers window
  // the packet buffer queue tail points to (one after) the last packet
  // that has been received from the laptop
  input [clog2(PB_QUEUE_LEN)-1:0] pb_head, pb_tail,
  input downstream_done,
  // inclk indicates that a packet has been received with sequence number
  // (FFCP index) given by in_index
  input inclk, input [FFCP_INDEX_LEN-1:0] in_index,
  // outclk indicates that we should send a packet with sequence number
  // given by out_index, with payload in the packet buffer queue
  // partition given by out_buf_pos; if out_syn is set, this packet
  // should additionally be a syn instead of a msg
```

```
  output outclk, out_syn,
  output [FFCP_INDEX_LEN-1:0] out_index,
  output [clog2(PB_QUEUE_LEN)-1:0] out_buf_pos,
  // the pb outputs are the interface to ffcp_queue
  // outclk_pb requests the ffcp_queue head to be overwritten with
  // out_pb_head
  output outclk_pb,
  output [clog2(PB_QUEUE_LEN)-1:0] out_pb_head);

`include "networking.vh"

reg downstream_rdy = 1;
always @(posedge clk) begin
  if (rst)
    downstream_rdy <= 1;
  // if we send an ack with outclk, downstream won't be ready until
  // it asserts done
  else if (outclk)
    downstream_rdy <= 0;
  else if (downstream_done)
    downstream_rdy <= 1;
end

// the head of the transmit window
// this is different from the head of the packet buffer queue,
// which is a separate queue that buffers packets from the laptop
// except in the case of a syn, we want the queue_head and packet buffer
// queue head to advance at the same time -- the queue_head advances
// when we receive an ack for a packet, indicating that we no longer
// need the packet in the packet buffer queue
reg [clog2(FFCP_BUFFER_LEN)-1:0] queue_head = 0;
// index in the transmit window of the packet that we are transmitting
// (or just transmitted)
reg [clog2(FFCP_BUFFER_LEN)-1:0] curr_index = 0;

// (one more than) the end of the transmit window
// used to ensure that the addition wraps around (i.e. is truncated)
// properly for correct comparison
wire [clog2(FFCP_BUFFER_LEN)-1:0] window_end;
wire at_end;
assign window_end = queue_head + FFCP_WINDOW_LEN;
// offset from the packet buffer queue head where the payload of the
// packet that we are transmitting is stored
wire [clog2(PB_QUEUE_LEN)-1:0] curr_index_pb;
// this is, apart from an offset of pb_head - queue_head, the same as
// curr_index
```

```
assign curr_index_pb = curr_index - queue_head + pb_head;
// we have transmitted all packets in the current cycle if we are
// at the end of the transmit window, or there are no more packets
// to transmit
assign at_end = curr_index == window_end || curr_index_pb == pb_tail;

// ignore all acks other than those in transmit window
// be careful of wraparound
wire ignore, receiving;
wire [clog2(FFCP_BUFFER_LEN)-1:0] in_index_head_off;
// offset in the transmit window of the incoming ack
// use an explicitly declared net to truncate before comparison,
// so that it will always be positive (this is a circular buffer,
// so a negative offset is equivalent to a large positive one)
assign in_index_head_off = in_index - queue_head;
assign ignore = in_index_head_off >= FFCP_WINDOW_LEN;
assign receiving = inclk && !ignore;

// overwrite the packet buffer queue head when an ack is received
assign outclk_pb = receiving;
assign out_pb_head = in_index - queue_head + pb_head;

// syn_buf indicates that the next packet sent should be a syn
// first packet should be a syn once reset is done
reg syn_buf = 1;
assign out_syn = syn_buf && out_index == 0;
assign out_index = curr_index;
assign out_buf_pos = curr_index - queue_head + pb_head;
assign outclk = !rst && downstream_rdy && !at_end;

localparam TESTING = 0;
// unit test values are 4 and 20
localparam RESEND_TIMEOUT = TESTING ? 100 : 500000;
localparam RESYN_TIMEOUT = TESTING ? 60000 : 50000000;

// use pulse extenders as timers
// resyn indicates that the data stream should be reset
wire resend_disable, resyn_disable, resyn;
// wait for 10ms before trying again
pulse_extender #(.EXTEND_LEN(RESEND_TIMEOUT)) resend_timer (
  .clk(clk), .rst(rst), .in(!at_end), .out(resend_disable));
// wait for 1s before trying to re-establish the connection
pulse_extender #(.EXTEND_LEN(RESYN_TIMEOUT)) resyn_timer (
  .clk(clk), .rst(rst), .in(inclk), .out(resyn_disable));
// only assert resyn for one clock cycle, don't resyn again until
// we hear an ack from the receiving FPGA
```

```
pulse_generator resyn_pg (
  .clk(clk), .rst(rst), .in(!resyn_disable), .out(resyn));

// difference between in_index and curr_index, but truncated to
// take care of wraparound
wire [clog2(FFCP_BUFFER_LEN)-1:0] in_index_curr_index_off;
assign in_index_curr_index_off = in_index - curr_index;

always @(posedge clk) begin
  // reset is similar to resyn since in both cases we are
  // starting a new data stream
  if (rst || resyn) begin
    queue_head <= 0;
    curr_index <= 0;
    syn_buf <= 1;
  end else begin
    if (receiving) begin
      // once we get an ack, we no longer want to syn
      syn_buf <= 0;
      // move the transmit window up to the ack index
      queue_head <= in_index;
    end
    // if the transmit window has shifted beyond the packet
    // we're currently transmitting, just skip ahead
    if (receiving && in_index_curr_index_off < FFCP_WINDOW_LEN)
      curr_index <= in_index;
    // when we transmit a packet, go to the next packet
    else if (outclk)
      curr_index <= curr_index + 1;
    // when the resend timeout expires, go back to the head of the
    // transmit window
    else if (!resend_disable)
      curr_index <= queue_head;
  end
end

endmodule
```

A.6 AES

aes.v:

```
// inclk is asserted when a block is presented on in
// outclk should be asserted when a block is presented on out
// always takes the same number of clock cycles per block
module aes_combined(
```

```
  input clk, rst,
  input inclk, input [BLOCK_LEN-1:0] in, key,
  output outclk, output [BLOCK_LEN-1:0] out,
  input decr_select);

`include "params.vh"

  reg [127:0] aes_in;
  wire [127:0] aes_key;
  wire [127:0] aes_out;
  reg [4:0] count = 0;
  reg crypting = 0;
  reg [4:0] key_counter = 0;

  always @(posedge clk) begin
    if(rst) begin
        count <= 0;
        crypting <= 0;
        key_counter <= 0;
    end
  else if (key_counter < 12)
    key_counter <= key_counter + 1;
    else if (inclk) begin
        count <= 0;
        crypting <= 1;
        aes_in <= in;
    end
    else if (crypting && count == 0) begin
            aes_in <= aes_in ^ aes_key;
            count <= count + 1;
    end
    else if (crypting && count < 11) begin
        count <= count + 1;
        aes_in <= aes_out;
    end
    else crypting <= 0;
  end
  aes_block block(.in(aes_in), .key(aes_key), .block_num(count-1), .out(aes_out), .decr_sele
  keygen round_key(.clk(clk), .start(key_counter == 0), .key(key), .keyout_sel(decr_select ?

  assign out = (count == 10) ? aes_out : 0;
  assign outclk = (count == 10);

endmodule

// implements CBC mode
```

```
// when encrypting, XOR the previous output (ciphertext) with the current
// input (plaintext)
// when decrypting, XOR the previous input (ciphertext) with the current
// output (plaintext)
module aes_chain(
  input clk, rst,
  input inclk, input [BLOCK_LEN-1:0] in, key,
  output outclk, output [BLOCK_LEN-1:0] out,
  // only perform CBC if cbc_enable is asserted, to allow us to
  // demonstrate difference with CBC enabled and disabled
  input decr_select, cbc_enable);

`include "params.vh"

// need additional buffering for prev when decrypting -- some time
// passes between inclk and outclk, so we want the plaintext from
// two inclks before
reg [BLOCK_LEN-1:0] prev_prev = 0;
reg [BLOCK_LEN-1:0] prev = 0;
wire [BLOCK_LEN-1:0] aes_out;
aes_combined aes_inst(
  .clk(clk), .rst(rst),
  .inclk(inclk), .in((!decr_select && cbc_enable) ? (in ^ prev) : in),
  .key(key),
  .outclk(outclk), .out(aes_out), .decr_select(decr_select));
assign out = (decr_select && cbc_enable) ? (aes_out ^ prev_prev) : aes_out;

always @(posedge clk) begin
  if (rst) begin
    prev_prev <= 0;
    prev <= 0;
  end else if (decr_select && inclk) begin
    prev_prev <= prev;
    prev <= in;
  end else if (!decr_select && outclk)
    prev <= out;
end

endmodule

// buffered version of bytes interface for aes_combined
// for use with network stack
module aes_combined_bytes_buf(
  input clk, rst,
  input inclk, input [BYTE_LEN-1:0] in, input in_done,
  input [BLOCK_LEN-1:0] key,
```

```
  input readclk,
  output outclk, output [BYTE_LEN-1:0] out, output done,
  output upstream_readclk,
  input decr_select, cbc_enable);

`include "params.vh"

wire btbl_outclk, btbl_done;
wire [BLOCK_LEN-1:0] btbl_out;
bytes_to_blocks btbl_inst(
  .clk(clk), .rst(rst), .inclk(inclk), .in(in), .in_done(in_done),
  .outclk(btbl_outclk), .out(btbl_out), .done(btbl_done));
wire btbl_swb_empty;
wire btbl_swb_outclk, btbl_swb_done;
wire [BLOCK_LEN-1:0] btbl_swb_out;
single_word_buffer #(.DATA_WIDTH(BLOCK_LEN+1)) btbl_swb_inst(
  .clk(clk), .rst(rst), .clear(btbl_swb_outclk),
  .inclk(btbl_outclk), .in({btbl_out, btbl_done}),
  .empty(btbl_swb_empty), .out({btbl_swb_out, btbl_swb_done}));

wire aes_outclk;
wire [BLOCK_LEN-1:0] aes_out;
aes_chain aes_inst(
  .clk(clk), .rst(rst),
  .inclk(btbl_swb_outclk), .in(btbl_swb_out), .key(key),
  .outclk(aes_outclk), .out(aes_out),
  .decr_select(decr_select), .cbc_enable(cbc_enable));
reg aes_done_buf = 0;
always @(posedge clk) begin
  if (btbl_swb_outclk && btbl_swb_done)
    aes_done_buf <= 1;
  else if (aes_outclk)
    aes_done_buf <= 0;
end

wire bltb_swb_empty;
wire bltb_swb_outclk, bltb_swb_done;
wire [BLOCK_LEN-1:0] bltb_swb_out;
single_word_buffer #(.DATA_WIDTH(BLOCK_LEN+1)) bltb_swb_inst(
  .clk(clk), .rst(rst), .clear(bltb_swb_outclk),
  .inclk(aes_outclk), .in({aes_out, aes_done_buf}),
  .empty(bltb_swb_empty), .out({bltb_swb_out, bltb_swb_done}));
wire bltb_rdy;
blocks_to_bytes bltb_inst(
  .clk(clk), .rst(rst),
  .inclk(bltb_swb_outclk), .in(bltb_swb_out), .in_done(bltb_swb_done),
```

```
  .readclk(readclk), .outclk(outclk), .out(out), .done(done),
  .rdy(bltb_rdy));
// read out from the blocks-to-bytes buffer when the blocks-to-bytes
// module is ready
assign bltb_swb_outclk = readclk && bltb_rdy && !bltb_swb_empty;
// the bytes-to-blocks module generates blocks no faster than once
// every 16 clock cycles, so the aes module will always be ready when
// the bytes-to-blocks buffer is written to
// so we just read out from the bytes-to-blocks buffer whenever
// the blocks-to-bytes buffer is ready
assign btbl_swb_outclk = (bltb_swb_outclk || bltb_swb_empty) &&
  !btbl_swb_empty;
// send a read to upstream when the bytes-to-blocks buffer is being
// cleared or is already empty
assign upstream_readclk = btbl_swb_outclk || btbl_swb_empty;

endmodule

module aes_block(input [127:0] in,
                 input [127:0] key,
                 input [3:0] block_num,
                 input decr_select,
                 output [127:0] out);

    wire [127:0] sb_out_e;
    wire [127:0] sr_out_e;
    wire [127:0] mc_out_e;
    wire [127:0] rc_out_e;
    wire [127:0] sb_out_d;
    wire [127:0] sr_out_d;
    wire [127:0] mc_out_d;
    wire [127:0] rc_out_d;

    subbytes a_e(.in(in), .out(sb_out_e), .decrypt(1'b0));
    shiftrows b_e(.in(sb_out_e), .out(sr_out_e), .decrypt(1'b0));
    mixcolumns c_e(.in(sr_out_e), .out(mc_out_e), .decrypt(1'b0));
    addroundkey d_e(
    .in((block_num == 9) ? sr_out_e : mc_out_e),
    .out(rc_out_e), .key(key));

    shiftrows b_d(.in(in), .out(sr_out_d), .decrypt(1'b1));
    subbytes a_d(.in(sr_out_d), .out(sb_out_d), .decrypt(1'b1));
    addroundkey d_d(.in(sb_out_d), .out(rc_out_d), .key(key));
    mixcolumns c_d(.in(rc_out_d), .out(mc_out_d), .decrypt(1'b1));

  assign out = decr_select ? (
```

```
    (block_num == 9) ? rc_out_d : mc_out_d) : rc_out_e;

endmodule

module keygen(input clk,
              input start,
              input [127:0] key,
              input [3:0] keyout_sel,
              output [127:0] keyout_selected);
    reg [127:0] rcon;

    reg generating;
    reg [3:0] round_num;
    reg [31:0] w0, w1, w2, w3;

    reg [127:0] keyout [10:0];

    wire [31:0] temp;

    assign keyout_selected = keyout[keyout_sel];
    always @(posedge clk) begin
        if (start) begin
            generating <= 1;
            round_num <= 1;
            keyout[0] <= key;
            w0 <= key[127:96];
            w1 <= key[95:64];
            w2 <= key[63:32];
            w3 <= key[31:0];
        end
        else if (generating && round_num <= 10) begin
            w0 <= w0^temp^rcon;
            w1 <= w0^temp^rcon^w1;
            w2 <= w0^temp^rcon^w1^w2;
            w3 <= w0^temp^rcon^w1^w2^w3;

            keyout[round_num][127:96] <= w0^temp^rcon;
            keyout[round_num][95:64] <= w0^temp^rcon^w1;
            keyout[round_num][63:32] <= w0^temp^rcon^w1^w2;
            keyout[round_num][31:0] <= w0^temp^rcon^w1^w2^w3;
            round_num <= round_num + 1;
        end
    end

    wire [7:0] temp0, temp1, temp2, temp3;
```

```
    sbox q0( .in(keyout[round_num-1][63:56]),.out(temp0));
    sbox q1( .in(keyout[round_num-1][71:64]),.out(temp1));
    sbox q2( .in(keyout[round_num-1][79:72]),.out(temp2));
    sbox q3( .in(keyout[round_num-1][87:80]),.out(temp3));
    assign temp = {temp1,temp2,temp3,temp0};

    // rc_i top byte of rcon
    // rc_i = rc_{i-1} *2 if <= rc_{i-1} h'80 else rc_{i-1} *2  ^ h'11b
    always @(*) begin
        case (round_num)
            4'h1: rcon=128'h01_00_00_00_00;
            4'h2: rcon=128'h02_00_00_00_00;
            4'h3: rcon=128'h04_00_00_00_00;
            4'h4: rcon=128'h08_00_00_00_00;
            4'h5: rcon=128'h10_00_00_00_00;
            4'h6: rcon=128'h20_00_00_00_00;
            4'h7: rcon=128'h40_00_00_00_00;
            4'h8: rcon=128'h80_00_00_00_00;
            4'h9: rcon=128'h1b_00_00_00_00;
            4'ha: rcon=128'h36_00_00_00_00;
            default: rcon = 0;
        endcase
    end

endmodule

module addroundkey(input [127:0] in,
                   input [127:0] key,
                   output [127:0] out);
    assign out = in ^ key;
endmodule

module inv_sbox(input [7:0] in,
            output [7:0] out);

            reg [7:0] inv_sbox;
            always @(*) begin
                case (in)
          8'h0 : inv_sbox = 8'h52;
          8'h1 : inv_sbox = 8'h9;
          8'h2 : inv_sbox = 8'h6A;
          8'h3 : inv_sbox = 8'hD5;
          8'h4 : inv_sbox = 8'h30;
          8'h5 : inv_sbox = 8'h36;
          8'h6 : inv_sbox = 8'hA5;
          8'h7 : inv_sbox = 8'h38;
```

```
8'h8 : inv_sbox = 8'hBF;
8'h9 : inv_sbox = 8'h40;
8'hA : inv_sbox = 8'hA3;
8'hB : inv_sbox = 8'h9E;
8'hC : inv_sbox = 8'h81;
8'hD : inv_sbox = 8'hF3;
8'hE : inv_sbox = 8'hD7;
8'hF : inv_sbox = 8'hFB;
8'h10 : inv_sbox = 8'h7C;
8'h11 : inv_sbox = 8'hE3;
8'h12 : inv_sbox = 8'h39;
8'h13 : inv_sbox = 8'h82;
8'h14 : inv_sbox = 8'h9B;
8'h15 : inv_sbox = 8'h2F;
8'h16 : inv_sbox = 8'hFF;
8'h17 : inv_sbox = 8'h87;
8'h18 : inv_sbox = 8'h34;
8'h19 : inv_sbox = 8'h8E;
8'h1A : inv_sbox = 8'h43;
8'h1B : inv_sbox = 8'h44;
8'h1C : inv_sbox = 8'hC4;
8'h1D : inv_sbox = 8'hDE;
8'h1E : inv_sbox = 8'hE9;
8'h1F : inv_sbox = 8'hCB;
8'h20 : inv_sbox = 8'h54;
8'h21 : inv_sbox = 8'h7B;
8'h22 : inv_sbox = 8'h94;
8'h23 : inv_sbox = 8'h32;
8'h24 : inv_sbox = 8'hA6;
8'h25 : inv_sbox = 8'hC2;
8'h26 : inv_sbox = 8'h23;
8'h27 : inv_sbox = 8'h3D;
8'h28 : inv_sbox = 8'hEE;
8'h29 : inv_sbox = 8'h4C;
8'h2A : inv_sbox = 8'h95;
8'h2B : inv_sbox = 8'hB;
8'h2C : inv_sbox = 8'h42;
8'h2D : inv_sbox = 8'hFA;
8'h2E : inv_sbox = 8'hC3;
8'h2F : inv_sbox = 8'h4E;
8'h30 : inv_sbox = 8'h8;
8'h31 : inv_sbox = 8'h2E;
8'h32 : inv_sbox = 8'hA1;
8'h33 : inv_sbox = 8'h66;
8'h34 : inv_sbox = 8'h28;
8'h35 : inv_sbox = 8'hD9;
```

```
8'h36 : inv_sbox = 8'h24;
8'h37 : inv_sbox = 8'hB2;
8'h38 : inv_sbox = 8'h76;
8'h39 : inv_sbox = 8'h5B;
8'h3A : inv_sbox = 8'hA2;
8'h3B : inv_sbox = 8'h49;
8'h3C : inv_sbox = 8'h6D;
8'h3D : inv_sbox = 8'h8B;
8'h3E : inv_sbox = 8'hD1;
8'h3F : inv_sbox = 8'h25;
8'h40 : inv_sbox = 8'h72;
8'h41 : inv_sbox = 8'hF8;
8'h42 : inv_sbox = 8'hF6;
8'h43 : inv_sbox = 8'h64;
8'h44 : inv_sbox = 8'h86;
8'h45 : inv_sbox = 8'h68;
8'h46 : inv_sbox = 8'h98;
8'h47 : inv_sbox = 8'h16;
8'h48 : inv_sbox = 8'hD4;
8'h49 : inv_sbox = 8'hA4;
8'h4A : inv_sbox = 8'h5C;
8'h4B : inv_sbox = 8'hCC;
8'h4C : inv_sbox = 8'h5D;
8'h4D : inv_sbox = 8'h65;
8'h4E : inv_sbox = 8'hB6;
8'h4F : inv_sbox = 8'h92;
8'h50 : inv_sbox = 8'h6C;
8'h51 : inv_sbox = 8'h70;
8'h52 : inv_sbox = 8'h48;
8'h53 : inv_sbox = 8'h50;
8'h54 : inv_sbox = 8'hFD;
8'h55 : inv_sbox = 8'hED;
8'h56 : inv_sbox = 8'hB9;
8'h57 : inv_sbox = 8'hDA;
8'h58 : inv_sbox = 8'h5E;
8'h59 : inv_sbox = 8'h15;
8'h5A : inv_sbox = 8'h46;
8'h5B : inv_sbox = 8'h57;
8'h5C : inv_sbox = 8'hA7;
8'h5D : inv_sbox = 8'h8D;
8'h5E : inv_sbox = 8'h9D;
8'h5F : inv_sbox = 8'h84;
8'h60 : inv_sbox = 8'h90;
8'h61 : inv_sbox = 8'hD8;
8'h62 : inv_sbox = 8'hAB;
8'h63 : inv_sbox = 8'h0;
```

```
8'h64 : inv_sbox = 8'h8C;
8'h65 : inv_sbox = 8'hBC;
8'h66 : inv_sbox = 8'hD3;
8'h67 : inv_sbox = 8'hA;
8'h68 : inv_sbox = 8'hF7;
8'h69 : inv_sbox = 8'hE4;
8'h6A : inv_sbox = 8'h58;
8'h6B : inv_sbox = 8'h5;
8'h6C : inv_sbox = 8'hB8;
8'h6D : inv_sbox = 8'hB3;
8'h6E : inv_sbox = 8'h45;
8'h6F : inv_sbox = 8'h6;
8'h70 : inv_sbox = 8'hD0;
8'h71 : inv_sbox = 8'h2C;
8'h72 : inv_sbox = 8'h1E;
8'h73 : inv_sbox = 8'h8F;
8'h74 : inv_sbox = 8'hCA;
8'h75 : inv_sbox = 8'h3F;
8'h76 : inv_sbox = 8'hF;
8'h77 : inv_sbox = 8'h2;
8'h78 : inv_sbox = 8'hC1;
8'h79 : inv_sbox = 8'hAF;
8'h7A : inv_sbox = 8'hBD;
8'h7B : inv_sbox = 8'h3;
8'h7C : inv_sbox = 8'h1;
8'h7D : inv_sbox = 8'h13;
8'h7E : inv_sbox = 8'h8A;
8'h7F : inv_sbox = 8'h6B;
8'h80 : inv_sbox = 8'h3A;
8'h81 : inv_sbox = 8'h91;
8'h82 : inv_sbox = 8'h11;
8'h83 : inv_sbox = 8'h41;
8'h84 : inv_sbox = 8'h4F;
8'h85 : inv_sbox = 8'h67;
8'h86 : inv_sbox = 8'hDC;
8'h87 : inv_sbox = 8'hEA;
8'h88 : inv_sbox = 8'h97;
8'h89 : inv_sbox = 8'hF2;
8'h8A : inv_sbox = 8'hCF;
8'h8B : inv_sbox = 8'hCE;
8'h8C : inv_sbox = 8'hF0;
8'h8D : inv_sbox = 8'hB4;
8'h8E : inv_sbox = 8'hE6;
8'h8F : inv_sbox = 8'h73;
8'h90 : inv_sbox = 8'h96;
8'h91 : inv_sbox = 8'hAC;
```

```
8'h92 : inv_sbox = 8'h74;
8'h93 : inv_sbox = 8'h22;
8'h94 : inv_sbox = 8'hE7;
8'h95 : inv_sbox = 8'hAD;
8'h96 : inv_sbox = 8'h35;
8'h97 : inv_sbox = 8'h85;
8'h98 : inv_sbox = 8'hE2;
8'h99 : inv_sbox = 8'hF9;
8'h9A : inv_sbox = 8'h37;
8'h9B : inv_sbox = 8'hE8;
8'h9C : inv_sbox = 8'h1C;
8'h9D : inv_sbox = 8'h75;
8'h9E : inv_sbox = 8'hDF;
8'h9F : inv_sbox = 8'h6E;
8'hA0 : inv_sbox = 8'h47;
8'hA1 : inv_sbox = 8'hF1;
8'hA2 : inv_sbox = 8'h1A;
8'hA3 : inv_sbox = 8'h71;
8'hA4 : inv_sbox = 8'h1D;
8'hA5 : inv_sbox = 8'h29;
8'hA6 : inv_sbox = 8'hC5;
8'hA7 : inv_sbox = 8'h89;
8'hA8 : inv_sbox = 8'h6F;
8'hA9 : inv_sbox = 8'hB7;
8'hAA : inv_sbox = 8'h62;
8'hAB : inv_sbox = 8'hE;
8'hAC : inv_sbox = 8'hAA;
8'hAD : inv_sbox = 8'h18;
8'hAE : inv_sbox = 8'hBE;
8'hAF : inv_sbox = 8'h1B;
8'hB0 : inv_sbox = 8'hFC;
8'hB1 : inv_sbox = 8'h56;
8'hB2 : inv_sbox = 8'h3E;
8'hB3 : inv_sbox = 8'h4B;
8'hB4 : inv_sbox = 8'hC6;
8'hB5 : inv_sbox = 8'hD2;
8'hB6 : inv_sbox = 8'h79;
8'hB7 : inv_sbox = 8'h20;
8'hB8 : inv_sbox = 8'h9A;
8'hB9 : inv_sbox = 8'hDB;
8'hBA : inv_sbox = 8'hC0;
8'hBB : inv_sbox = 8'hFE;
8'hBC : inv_sbox = 8'h78;
8'hBD : inv_sbox = 8'hCD;
8'hBE : inv_sbox = 8'h5A;
8'hBF : inv_sbox = 8'hF4;
```

```
8'hC0 : inv_sbox = 8'h1F;
8'hC1 : inv_sbox = 8'hDD;
8'hC2 : inv_sbox = 8'hA8;
8'hC3 : inv_sbox = 8'h33;
8'hC4 : inv_sbox = 8'h88;
8'hC5 : inv_sbox = 8'h7;
8'hC6 : inv_sbox = 8'hC7;
8'hC7 : inv_sbox = 8'h31;
8'hC8 : inv_sbox = 8'hB1;
8'hC9 : inv_sbox = 8'h12;
8'hCA : inv_sbox = 8'h10;
8'hCB : inv_sbox = 8'h59;
8'hCC : inv_sbox = 8'h27;
8'hCD : inv_sbox = 8'h80;
8'hCE : inv_sbox = 8'hEC;
8'hCF : inv_sbox = 8'h5F;
8'hD0 : inv_sbox = 8'h60;
8'hD1 : inv_sbox = 8'h51;
8'hD2 : inv_sbox = 8'h7F;
8'hD3 : inv_sbox = 8'hA9;
8'hD4 : inv_sbox = 8'h19;
8'hD5 : inv_sbox = 8'hB5;
8'hD6 : inv_sbox = 8'h4A;
8'hD7 : inv_sbox = 8'hD;
8'hD8 : inv_sbox = 8'h2D;
8'hD9 : inv_sbox = 8'hE5;
8'hDA : inv_sbox = 8'h7A;
8'hDB : inv_sbox = 8'h9F;
8'hDC : inv_sbox = 8'h93;
8'hDD : inv_sbox = 8'hC9;
8'hDE : inv_sbox = 8'h9C;
8'hDF : inv_sbox = 8'hEF;
8'hE0 : inv_sbox = 8'hA0;
8'hE1 : inv_sbox = 8'hE0;
8'hE2 : inv_sbox = 8'h3B;
8'hE3 : inv_sbox = 8'h4D;
8'hE4 : inv_sbox = 8'hAE;
8'hE5 : inv_sbox = 8'h2A;
8'hE6 : inv_sbox = 8'hF5;
8'hE7 : inv_sbox = 8'hB0;
8'hE8 : inv_sbox = 8'hC8;
8'hE9 : inv_sbox = 8'hEB;
8'hEA : inv_sbox = 8'hBB;
8'hEB : inv_sbox = 8'h3C;
8'hEC : inv_sbox = 8'h83;
8'hED : inv_sbox = 8'h53;
```

```
          8'hEE : inv_sbox = 8'h99;
          8'hEF : inv_sbox = 8'h61;
          8'hF0 : inv_sbox = 8'h17;
          8'hF1 : inv_sbox = 8'h2B;
          8'hF2 : inv_sbox = 8'h4;
          8'hF3 : inv_sbox = 8'h7E;
          8'hF4 : inv_sbox = 8'hBA;
          8'hF5 : inv_sbox = 8'h77;
          8'hF6 : inv_sbox = 8'hD6;
          8'hF7 : inv_sbox = 8'h26;
          8'hF8 : inv_sbox = 8'hE1;
          8'hF9 : inv_sbox = 8'h69;
          8'hFA : inv_sbox = 8'h14;
          8'hFB : inv_sbox = 8'h63;
          8'hFC : inv_sbox = 8'h55;
          8'hFD : inv_sbox = 8'h21;
          8'hFE : inv_sbox = 8'hC;
          8'hFF : inv_sbox = 8'h7D;
          default : inv_sbox = 8'h0;
        endcase
            end
            assign out = inv_sbox;
endmodule

module sbox(input [7:0] in,
            output [7:0] out);

            reg [7:0] sbox;
            always @(*) begin
                case (in)
                    8'h0 : sbox = 8'h63;
                    8'h1 : sbox = 8'h7C;
                    8'h2 : sbox = 8'h77;
                    8'h3 : sbox = 8'h7B;
                    8'h4 : sbox = 8'hF2;
                    8'h5 : sbox = 8'h6B;
                    8'h6 : sbox = 8'h6F;
                    8'h7 : sbox = 8'hC5;
                    8'h8 : sbox = 8'h30;
                    8'h9 : sbox = 8'h1;
                    8'hA : sbox = 8'h67;
                    8'hB : sbox = 8'h2B;
                    8'hC : sbox = 8'hFE;
                    8'hD : sbox = 8'hD7;
                    8'hE : sbox = 8'hAB;
                    8'hF : sbox = 8'h76;
```

```
8'h10 : sbox = 8'hCA;
8'h11 : sbox = 8'h82;
8'h12 : sbox = 8'hC9;
8'h13 : sbox = 8'h7D;
8'h14 : sbox = 8'hFA;
8'h15 : sbox = 8'h59;
8'h16 : sbox = 8'h47;
8'h17 : sbox = 8'hF0;
8'h18 : sbox = 8'hAD;
8'h19 : sbox = 8'hD4;
8'h1A : sbox = 8'hA2;
8'h1B : sbox = 8'hAF;
8'h1C : sbox = 8'h9C;
8'h1D : sbox = 8'hA4;
8'h1E : sbox = 8'h72;
8'h1F : sbox = 8'hC0;
8'h20 : sbox = 8'hB7;
8'h21 : sbox = 8'hFD;
8'h22 : sbox = 8'h93;
8'h23 : sbox = 8'h26;
8'h24 : sbox = 8'h36;
8'h25 : sbox = 8'h3F;
8'h26 : sbox = 8'hF7;
8'h27 : sbox = 8'hCC;
8'h28 : sbox = 8'h34;
8'h29 : sbox = 8'hA5;
8'h2A : sbox = 8'hE5;
8'h2B : sbox = 8'hF1;
8'h2C : sbox = 8'h71;
8'h2D : sbox = 8'hD8;
8'h2E : sbox = 8'h31;
8'h2F : sbox = 8'h15;
8'h30 : sbox = 8'h4;
8'h31 : sbox = 8'hC7;
8'h32 : sbox = 8'h23;
8'h33 : sbox = 8'hC3;
8'h34 : sbox = 8'h18;
8'h35 : sbox = 8'h96;
8'h36 : sbox = 8'h5;
8'h37 : sbox = 8'h9A;
8'h38 : sbox = 8'h7;
8'h39 : sbox = 8'h12;
8'h3A : sbox = 8'h80;
8'h3B : sbox = 8'hE2;
8'h3C : sbox = 8'hEB;
8'h3D : sbox = 8'h27;
```

```
8'h3E : sbox = 8'hB2;
8'h3F : sbox = 8'h75;
8'h40 : sbox = 8'h9;
8'h41 : sbox = 8'h83;
8'h42 : sbox = 8'h2C;
8'h43 : sbox = 8'h1A;
8'h44 : sbox = 8'h1B;
8'h45 : sbox = 8'h6E;
8'h46 : sbox = 8'h5A;
8'h47 : sbox = 8'hA0;
8'h48 : sbox = 8'h52;
8'h49 : sbox = 8'h3B;
8'h4A : sbox = 8'hD6;
8'h4B : sbox = 8'hB3;
8'h4C : sbox = 8'h29;
8'h4D : sbox = 8'hE3;
8'h4E : sbox = 8'h2F;
8'h4F : sbox = 8'h84;
8'h50 : sbox = 8'h53;
8'h51 : sbox = 8'hD1;
8'h52 : sbox = 8'h0;
8'h53 : sbox = 8'hED;
8'h54 : sbox = 8'h20;
8'h55 : sbox = 8'hFC;
8'h56 : sbox = 8'hB1;
8'h57 : sbox = 8'h5B;
8'h58 : sbox = 8'h6A;
8'h59 : sbox = 8'hCB;
8'h5A : sbox = 8'hBE;
8'h5B : sbox = 8'h39;
8'h5C : sbox = 8'h4A;
8'h5D : sbox = 8'h4C;
8'h5E : sbox = 8'h58;
8'h5F : sbox = 8'hCF;
8'h60 : sbox = 8'hD0;
8'h61 : sbox = 8'hEF;
8'h62 : sbox = 8'hAA;
8'h63 : sbox = 8'hFB;
8'h64 : sbox = 8'h43;
8'h65 : sbox = 8'h4D;
8'h66 : sbox = 8'h33;
8'h67 : sbox = 8'h85;
8'h68 : sbox = 8'h45;
8'h69 : sbox = 8'hF9;
8'h6A : sbox = 8'h2;
8'h6B : sbox = 8'h7F;
```

```
8'h6C : sbox = 8'h50;
8'h6D : sbox = 8'h3C;
8'h6E : sbox = 8'h9F;
8'h6F : sbox = 8'hA8;
8'h70 : sbox = 8'h51;
8'h71 : sbox = 8'hA3;
8'h72 : sbox = 8'h40;
8'h73 : sbox = 8'h8F;
8'h74 : sbox = 8'h92;
8'h75 : sbox = 8'h9D;
8'h76 : sbox = 8'h38;
8'h77 : sbox = 8'hF5;
8'h78 : sbox = 8'hBC;
8'h79 : sbox = 8'hB6;
8'h7A : sbox = 8'hDA;
8'h7B : sbox = 8'h21;
8'h7C : sbox = 8'h10;
8'h7D : sbox = 8'hFF;
8'h7E : sbox = 8'hF3;
8'h7F : sbox = 8'hD2;
8'h80 : sbox = 8'hCD;
8'h81 : sbox = 8'hC;
8'h82 : sbox = 8'h13;
8'h83 : sbox = 8'hEC;
8'h84 : sbox = 8'h5F;
8'h85 : sbox = 8'h97;
8'h86 : sbox = 8'h44;
8'h87 : sbox = 8'h17;
8'h88 : sbox = 8'hC4;
8'h89 : sbox = 8'hA7;
8'h8A : sbox = 8'h7E;
8'h8B : sbox = 8'h3D;
8'h8C : sbox = 8'h64;
8'h8D : sbox = 8'h5D;
8'h8E : sbox = 8'h19;
8'h8F : sbox = 8'h73;
8'h90 : sbox = 8'h60;
8'h91 : sbox = 8'h81;
8'h92 : sbox = 8'h4F;
8'h93 : sbox = 8'hDC;
8'h94 : sbox = 8'h22;
8'h95 : sbox = 8'h2A;
8'h96 : sbox = 8'h90;
8'h97 : sbox = 8'h88;
8'h98 : sbox = 8'h46;
8'h99 : sbox = 8'hEE;
```

```
8'h9A : sbox = 8'hB8;
8'h9B : sbox = 8'h14;
8'h9C : sbox = 8'hDE;
8'h9D : sbox = 8'h5E;
8'h9E : sbox = 8'hB;
8'h9F : sbox = 8'hDB;
8'hA0 : sbox = 8'hE0;
8'hA1 : sbox = 8'h32;
8'hA2 : sbox = 8'h3A;
8'hA3 : sbox = 8'hA;
8'hA4 : sbox = 8'h49;
8'hA5 : sbox = 8'h6;
8'hA6 : sbox = 8'h24;
8'hA7 : sbox = 8'h5C;
8'hA8 : sbox = 8'hC2;
8'hA9 : sbox = 8'hD3;
8'hAA : sbox = 8'hAC;
8'hAB : sbox = 8'h62;
8'hAC : sbox = 8'h91;
8'hAD : sbox = 8'h95;
8'hAE : sbox = 8'hE4;
8'hAF : sbox = 8'h79;
8'hB0 : sbox = 8'hE7;
8'hB1 : sbox = 8'hC8;
8'hB2 : sbox = 8'h37;
8'hB3 : sbox = 8'h6D;
8'hB4 : sbox = 8'h8D;
8'hB5 : sbox = 8'hD5;
8'hB6 : sbox = 8'h4E;
8'hB7 : sbox = 8'hA9;
8'hB8 : sbox = 8'h6C;
8'hB9 : sbox = 8'h56;
8'hBA : sbox = 8'hF4;
8'hBB : sbox = 8'hEA;
8'hBC : sbox = 8'h65;
8'hBD : sbox = 8'h7A;
8'hBE : sbox = 8'hAE;
8'hBF : sbox = 8'h8;
8'hC0 : sbox = 8'hBA;
8'hC1 : sbox = 8'h78;
8'hC2 : sbox = 8'h25;
8'hC3 : sbox = 8'h2E;
8'hC4 : sbox = 8'h1C;
8'hC5 : sbox = 8'hA6;
8'hC6 : sbox = 8'hB4;
8'hC7 : sbox = 8'hC6;
```

```
8'hC8 : sbox = 8'hE8;
8'hC9 : sbox = 8'hDD;
8'hCA : sbox = 8'h74;
8'hCB : sbox = 8'h1F;
8'hCC : sbox = 8'h4B;
8'hCD : sbox = 8'hBD;
8'hCE : sbox = 8'h8B;
8'hCF : sbox = 8'h8A;
8'hD0 : sbox = 8'h70;
8'hD1 : sbox = 8'h3E;
8'hD2 : sbox = 8'hB5;
8'hD3 : sbox = 8'h66;
8'hD4 : sbox = 8'h48;
8'hD5 : sbox = 8'h3;
8'hD6 : sbox = 8'hF6;
8'hD7 : sbox = 8'hE;
8'hD8 : sbox = 8'h61;
8'hD9 : sbox = 8'h35;
8'hDA : sbox = 8'h57;
8'hDB : sbox = 8'hB9;
8'hDC : sbox = 8'h86;
8'hDD : sbox = 8'hC1;
8'hDE : sbox = 8'h1D;
8'hDF : sbox = 8'h9E;
8'hE0 : sbox = 8'hE1;
8'hE1 : sbox = 8'hF8;
8'hE2 : sbox = 8'h98;
8'hE3 : sbox = 8'h11;
8'hE4 : sbox = 8'h69;
8'hE5 : sbox = 8'hD9;
8'hE6 : sbox = 8'h8E;
8'hE7 : sbox = 8'h94;
8'hE8 : sbox = 8'h9B;
8'hE9 : sbox = 8'h1E;
8'hEA : sbox = 8'h87;
8'hEB : sbox = 8'hE9;
8'hEC : sbox = 8'hCE;
8'hED : sbox = 8'h55;
8'hEE : sbox = 8'h28;
8'hEF : sbox = 8'hDF;
8'hF0 : sbox = 8'h8C;
8'hF1 : sbox = 8'hA1;
8'hF2 : sbox = 8'h89;
8'hF3 : sbox = 8'hD;
8'hF4 : sbox = 8'hBF;
8'hF5 : sbox = 8'hE6;
```

```
                8'hF6 : sbox = 8'h42;
                8'hF7 : sbox = 8'h68;
                8'hF8 : sbox = 8'h41;
                8'hF9 : sbox = 8'h99;
                8'hFA : sbox = 8'h2D;
                8'hFB : sbox = 8'hF;
                8'hFC : sbox = 8'hB0;
                8'hFD : sbox = 8'h54;
                8'hFE : sbox = 8'hBB;
                8'hFF : sbox = 8'h16;
                default : sbox = 8'h0;
            endcase
        end
        assign out = sbox;
endmodule

module subbytes(input [127:0] in,
                input decrypt, // flag, when set to 1 work in decrypt mode
                output [127:0] out);

     wire [127:0] out_e;
     wire [127:0] out_d;

     sbox q0( .in(in[127:120]),.out(out_e[127:120]) );
     sbox q1( .in(in[119:112]),.out(out_e[119:112]) );
     sbox q2( .in(in[111:104]),.out(out_e[111:104]) );
     sbox q3( .in(in[103:96]),.out(out_e[103:96]) );

     sbox q4( .in(in[95:88]),.out(out_e[95:88]) );
     sbox q5( .in(in[87:80]),.out(out_e[87:80]) );
     sbox q6( .in(in[79:72]),.out(out_e[79:72]) );
     sbox q7( .in(in[71:64]),.out(out_e[71:64]) );

     sbox q8( .in(in[63:56]),.out(out_e[63:56]) );
     sbox q9( .in(in[55:48]),.out(out_e[55:48]) );
     sbox q10(.in(in[47:40]),.out(out_e[47:40]) );
     sbox q11(.in(in[39:32]),.out(out_e[39:32]) );

     sbox q12(.in(in[31:24]),.out(out_e[31:24]) );
     sbox q13(.in(in[23:16]),.out(out_e[23:16]) );
     sbox q14(.in(in[15:8]),.out(out_e[15:8]) );
     sbox q15(.in(in[7:0]),.out(out_e[7:0]) );

     inv_sbox iq0( .in(in[127:120]),.out(out_d[127:120]) );
     inv_sbox iq1( .in(in[119:112]),.out(out_d[119:112]) );
     inv_sbox iq2( .in(in[111:104]),.out(out_d[111:104]) );
```

```
     inv_sbox iq3( .in(in[103:96]),.out(out_d[103:96]) );

     inv_sbox iq4( .in(in[95:88]),.out(out_d[95:88]) );
     inv_sbox iq5( .in(in[87:80]),.out(out_d[87:80]) );
     inv_sbox iq6( .in(in[79:72]),.out(out_d[79:72]) );
     inv_sbox iq7( .in(in[71:64]),.out(out_d[71:64]) );

     inv_sbox iq8( .in(in[63:56]),.out(out_d[63:56]) );
     inv_sbox iq9( .in(in[55:48]),.out(out_d[55:48]) );
     inv_sbox iq10(.in(in[47:40]),.out(out_d[47:40]) );
     inv_sbox iq11(.in(in[39:32]),.out(out_d[39:32]) );

     inv_sbox iq12(.in(in[31:24]),.out(out_d[31:24]) );
     inv_sbox iq13(.in(in[23:16]),.out(out_d[23:16]) );
     inv_sbox iq14(.in(in[15:8]),.out(out_d[15:8]) );
     inv_sbox iq15(.in(in[7:0]),.out(out_d[7:0]) );

     assign out = decrypt ? out_d : out_e;
endmodule

module shiftrows(input [127:0] in,
                input decrypt, // flag, when set to 1 work in decrypt mode
                output [127:0] out);

    wire [7:0] bytes [15:0];

    genvar i;
    generate
        for (i = 0; i < 16; i=i+1) begin : gen_bytes
            assign bytes[15-i] = in[i*8+7:i*8];
        end
    endgenerate

    assign out = decrypt ?  {bytes[0], bytes[1], bytes[2], bytes[3],
                             bytes[7], bytes[4], bytes[5], bytes[6],
                             bytes[10], bytes[11], bytes[8], bytes[9],
                             bytes[13], bytes[14], bytes[15], bytes[12]}
                : {bytes[0], bytes[1], bytes[2], bytes[3],
                  bytes[5], bytes[6], bytes[7], bytes[4],
                  bytes[10], bytes[11], bytes[8], bytes[9],
                  bytes[15], bytes[12], bytes[13], bytes[14]};
endmodule

module mixcolumns(input [127:0] in,
                input decrypt, // flag, when set to 1 work in decrypt mode
                output [127:0] out);
```

```
 wire [7:0] bytes [15:0];
 wire [7:0] dbytes [15:0];       // doubled bytes
 wire [7:0] ddbytes [15:0];       // bytes * 4
 wire [7:0] dddbytes [15:0];       // bytse * 8
 wire [7:0] mixed_bytes_e [15:0];
 wire [7:0] mixed_bytes_d [15:0];


 genvar i;
 generate
     for (i = 0; i < 16; i=i+1) begin : gen_bytes
         assign bytes[i] = in[i*8+7:i*8];
         assign dbytes[i] = {bytes[i][6 : 0], 1'b0} ^
             (8'h1b & {8{bytes[i][7]}});
         assign ddbytes[i] = {dbytes[i][6 : 0], 1'b0} ^
             (8'h1b & {8{dbytes[i][7]}});
         assign dddbytes[i] = {ddbytes[i][6 : 0], 1'b0} ^
             (8'h1b & {8{ddbytes[i][7]}});
     end
 endgenerate


/*
For encryption, we left multiply each column by

2 3 1 1
1 2 3 1
1 1 2 3
3 1 1 2

in order to generate a new matrix. This means we can go column by column,
taking every fourth byte.

multiplication by 2 is equivalent to a left shift of 1,
multiplication by 3 is a left shift of 1 xored with the initial value.
using this, we can implement this transformation with no multipliers.

The inverse matrix (for decryption) is

0E 0B 0D 09
09 0E 0B 0D
0D 09 0E 0B
0B 0D 09 0E
*/
```

```
        genvar j;
        generate
            for (j = 0; j < 4; j=j+1) begin : gen_column_mix
                assign mixed_bytes_d[j] = dbytes[j]^ddbytes[j]^dddbytes[j]^
                                            bytes[j+4]^dbytes[j+4]^dddbytes[j+4]^
                                            bytes[j+8]^ddbytes[j+8]^dddbytes[j+8]^
                                            bytes[j+12]^dddbytes[j+12];
                assign mixed_bytes_d[j+4] = bytes[j]^dddbytes[j]^
                                            dbytes[j+4]^ddbytes[j+4]^dddbytes[j+4]^
                                            bytes[j+8]^dbytes[j+8]^dddbytes[j+8]^
                                            bytes[j+12]^ddbytes[j+12]^dddbytes[j+12];
                assign mixed_bytes_d[j+8] = bytes[j]^ddbytes[j]^dddbytes[j]^
                                            bytes[j+4]^dddbytes[j+4]^
                                            dbytes[j+8]^ddbytes[j+8]^dddbytes[j+8]^
                                            bytes[j+12]^dbytes[j+12]^dddbytes[j+12];
                assign mixed_bytes_d[j+12] = bytes[j]^dbytes[j]^dddbytes[j]^
                                            bytes[j+4]^ddbytes[j+4]^dddbytes[j+4]^
                                            bytes[j+8]^dddbytes[j+8]^
                                            dbytes[j+12]^ddbytes[j+12]^dddbytes[j+12];

                assign mixed_bytes_e[j] = dbytes[j]^dbytes[j+4]^bytes[j+4]^
                            bytes[j+8]^bytes[j+12];
                assign mixed_bytes_e[j+4] = bytes[j]^dbytes[j+4]^dbytes[j+8]^
                            bytes[j+8]^bytes[j+12];
                assign mixed_bytes_e[j+8] = bytes[j]^bytes[j+4]^dbytes[j+8]^
                            dbytes[j+12]^bytes[j+12];
                assign mixed_bytes_e[j+12] = dbytes[j]^bytes[j]^bytes[j+4]^
                            bytes[j+8]^dbytes[j+12];
            end
        endgenerate

        assign out = decrypt ?
            {mixed_bytes_d[15], mixed_bytes_d[14], mixed_bytes_d[13], mixed_bytes_d[12],
            mixed_bytes_d[11], mixed_bytes_d[10], mixed_bytes_d[9], mixed_bytes_d[8],
            mixed_bytes_d[7], mixed_bytes_d[6], mixed_bytes_d[5], mixed_bytes_d[4],
            mixed_bytes_d[3], mixed_bytes_d[2], mixed_bytes_d[1], mixed_bytes_d[0]}
            :
            {mixed_bytes_e[15], mixed_bytes_e[14], mixed_bytes_e[13], mixed_bytes_e[12],
            mixed_bytes_e[11], mixed_bytes_e[10], mixed_bytes_e[9], mixed_bytes_e[8],
            mixed_bytes_e[7], mixed_bytes_e[6], mixed_bytes_e[5], mixed_bytes_e[4],
            mixed_bytes_e[3], mixed_bytes_e[2], mixed_bytes_e[1], mixed_bytes_e[0]};
endmodule
```

A.7 Graphics

graphics.v

```
module graphics_main #(
  parameter RAM_SIZE = PACKET_BUFFER_SIZE) (
  input clk, rst, blank,
  input [clog2(VGA_WIDTH)-1:0] vga_x,
  input [clog2(VGA_HEIGHT)-1:0] vga_y,
  // vga_hsync/vsync should be delayed by the latency of this module
  // so that the output pixel colors would be synchronized
  input vga_hsync_in, vga_vsync_in,
  input ram_outclk, input [COLOR_LEN-1:0] ram_out,
  output ram_readclk, output [clog2(RAM_SIZE)-1:0] ram_raddr,
  output [COLOR_LEN-1:0] vga_col,
  output vga_hsync_out, vga_vsync_out);

`include "params.vh"

wire blank_delayed;
delay #(.DELAY_LEN(VIDEO_CACHE_RAM_LATENCY)) hsync_delay(
  .clk(clk), .rst(rst), .in(vga_hsync_in), .out(vga_hsync_out));
delay #(.DELAY_LEN(VIDEO_CACHE_RAM_LATENCY)) vsync_delay(
  .clk(clk), .rst(rst), .in(vga_vsync_in), .out(vga_vsync_out));
delay #(.DELAY_LEN(VIDEO_CACHE_RAM_LATENCY)) blank_delay(
  .clk(clk), .rst(rst), .in(blank), .out(blank_delayed));

// number of pixels per image pixel in each direction
localparam RESOLUTION = 4;
// height and width of the image
localparam IMAGE_SIZE = 128;

// position in image to display, in terms of image pixels
wire [clog2(IMAGE_SIZE)-1:0] image_x, image_y;
// position in image to display, in terms of screen pixels
wire [clog2(VGA_WIDTH)-1:0] image_x_pix;
wire [clog2(VGA_HEIGHT)-1:0] image_y_pix;
// center image in screen
assign image_x_pix = vga_x - (VGA_WIDTH/2 - IMAGE_SIZE*RESOLUTION/2);
assign image_y_pix = vga_y - (VGA_HEIGHT/2 - IMAGE_SIZE*RESOLUTION/2);
// divide by RESOLUTION
assign image_x = image_x_pix[clog2(RESOLUTION)+:clog2(IMAGE_SIZE)];
assign image_y = image_y_pix[clog2(RESOLUTION)+:clog2(IMAGE_SIZE)];
// display the image in the center and white everywhere else
assign ram_readclk = !blank &&
  vga_x >= VGA_WIDTH/2 - IMAGE_SIZE*RESOLUTION/2 &&
  vga_x < VGA_WIDTH/2 + IMAGE_SIZE*RESOLUTION/2 &&
```

```
  vga_y >= VGA_HEIGHT/2 - IMAGE_SIZE*RESOLUTION/2 &&
  vga_y < VGA_HEIGHT/2 + IMAGE_SIZE*RESOLUTION/2;
assign ram_raddr = {image_y, image_x};
assign vga_col = blank_delayed ? 12'h0 :
  ram_outclk ? ram_out : 12'hfff;

endmodule
```

A.8 Top Level

main.v:

```
// SW[0]: reset
// SW[1]: master configure: on for transmit, off for receive
// SW[2]: UART_CTS override (to test flow control)
// SW[3]: enable CBC mode
// SW[15:8]: last bits of key for aes
// BTNC: dump ram over uart (from vram, only in transmit configuration)
module main(
  input CLK100MHZ,
  input [15:0] SW,
  input BTNC, BTNU, BTNL, BTNR, BTND,
  output [7:0] JB,
  output [3:0] VGA_R,
  output [3:0] VGA_B,
  output [3:0] VGA_G,
  output VGA_HS,
  output VGA_VS,
  output LED16_B, LED16_G, LED16_R,
  output LED17_B, LED17_G, LED17_R,
  output [15:0] LED,
  output [7:0] SEG,  // segments A-G (0-6), DP (7)
  output [7:0] AN,  // Display 0-7
  inout ETH_CRSDV, ETH_RXERR,
  inout [1:0] ETH_RXD,
  output ETH_REFCLK, ETH_INTN, ETH_RSTN,
  input UART_TXD_IN, UART_RTS,
  output UART_RXD_OUT, UART_CTS,
  output ETH_TXEN,
  output [1:0] ETH_TXD,
  output ETH_MDC, ETH_MDIO);

////// INCLUDES

`include "networking.vh"
```

```
localparam RAM_SIZE = PACKET_BUFFER_SIZE;
localparam VRAM_SIZE = VIDEO_CACHE_RAM_SIZE;
localparam ROM_SIZE = PACKET_SYNTH_ROM_SIZE;

// fixed key to use; last 8 bits will be overwritten by switches
localparam KEY = 128'h4b42_4410_770a_ee13_094d_d0da_1217_7bb0;

////// CLOCKING

wire clk_50mhz;

// the main clock for FPGA logic will be 50MHz
wire clk;
assign clk = clk_50mhz;

wire clk_120mhz;

// 50MHz clock for Ethernet receiving
clk_wiz_0 clk_wiz_inst(
  .reset(0),
  .clk_in1(CLK100MHZ),
  .clk_out1(clk_50mhz),
  .clk_out3(clk_120mhz));

////// CONFIGURATION AND RESET

wire sw0, sw1, sw2, sw3;
wire [7:0] swkey;
delay #(.DELAY_LEN(SYNC_DELAY_LEN)) sw0_sync(
  .clk(clk), .rst(1'b0), .in(SW[0]), .out(sw0));
delay #(.DELAY_LEN(SYNC_DELAY_LEN)) sw1_sync(
  .clk(clk), .rst(1'b0), .in(SW[1]), .out(sw1));
delay #(.DELAY_LEN(SYNC_DELAY_LEN)) sw2_sync(
  .clk(clk), .rst(1'b0), .in(SW[2]), .out(sw2));
delay #(.DELAY_LEN(SYNC_DELAY_LEN)) sw3_sync(
  .clk(clk), .rst(1'b0), .in(SW[3]), .out(sw3));
delay #(.DELAY_LEN(SYNC_DELAY_LEN), .DATA_WIDTH(8)) swkey_sync(
  .clk(clk), .rst(1'b0), .in(SW[8+:8]), .out(swkey));

wire config_transmit;
assign config_transmit = sw1;
wire cbc_enable;
assign cbc_enable = sw3;

reg prev_sw1 = 0;
reg prev_sw3 = 0;
```

```
reg [7:0] prev_swkey = 0;
always @(posedge clk) begin
  prev_sw1 <= sw1;
  prev_sw3 <= sw3;
  prev_swkey <= swkey;
end

// reset device when configuration is changed
wire config_change_reset;
assign config_change_reset =
  sw1 != prev_sw1 ||
  sw3 != prev_sw3 ||
  swkey != prev_swkey;

wire rst;
// ensure that reset pulse lasts a sufficient long amount of time
// skip reset when testing
localparam TESTING = 0;
localparam RESET_TIMEOUT = TESTING ? 1 : 5000000;
pulse_extender #(.EXTEND_LEN(RESET_TIMEOUT)) reset_pe(
  .clk(clk), .rst(1'b0), .in(sw0 || config_change_reset), .out(rst));

////// HEX DISPLAY

wire [31:0] hex_display_data;
wire [6:0] segments;

display_8hex display(
  .clk(clk), .data(hex_display_data), .seg(segments), .strobe(AN));

assign SEG[7] = 1'b1;
assign SEG[6:0] = segments;

////// LEDS

assign LED16_R = BTNL; // left button -> red led
assign LED16_G = BTNC; // center button -> green led
assign LED16_B = BTNR; // right button -> blue led
assign LED17_R = BTNL;
assign LED17_G = BTNC;
assign LED17_B = BTNR;

////// BUTTONS

wire btnc_raw, btnl_raw, btnc, btnl;
sync_debounce sd_btnc(
```

```
  .rst(rst), .clk(clk), .in(BTNC), .out(btnc_raw));
sync_debounce sd_btnl(
  .rst(rst), .clk(clk), .in(BTNL), .out(btnl_raw));

pulse_generator pg_btnc(
  .clk(clk), .rst(rst), .in(btnc_raw), .out(btnc));
pulse_generator pg_btnl(
  .clk(clk), .rst(rst), .in(btnl_raw), .out(btnl));

////// VGA

wire [clog2(VGA_WIDTH)-1:0] vga_x;
wire [clog2(VGA_HEIGHT)-1:0] vga_y;
// allow for hsync and vsync to be delayed before sending on wire
wire vga_hsync, vga_vsync, vga_hsync_predelay, vga_vsync_predelay, blank;

xvga xvga_inst(
  .clk(clk), .vga_x(vga_x), .vga_y(vga_y),
  .vga_hsync(vga_hsync_predelay), .vga_vsync(vga_vsync_predelay),
  .blank(blank));

// vga_col should be set by some downstream module, indicating the
// color of the current pixel
wire [COLOR_CHANNEL_LEN-1:0] vga_r_out, vga_g_out, vga_b_out;
wire [COLOR_LEN-1:0] vga_col;
assign {vga_r_out, vga_g_out, vga_b_out} = vga_col;

// buffer all outputs to enforce timing constraints
delay #(.DATA_WIDTH(COLOR_CHANNEL_LEN)) vga_r_sync(
  .clk(clk), .rst(rst), .in(vga_r_out), .out(VGA_R));
delay #(.DATA_WIDTH(COLOR_CHANNEL_LEN)) vga_g_sync(
  .clk(clk), .rst(rst), .in(vga_g_out), .out(VGA_G));
delay #(.DATA_WIDTH(COLOR_CHANNEL_LEN)) vga_b_sync(
  .clk(clk), .rst(rst), .in(vga_b_out), .out(VGA_B));
delay vga_hs_sync(
  .clk(clk), .rst(rst), .in(vga_hsync), .out(VGA_HS));
delay vga_vs_sync(
  .clk(clk), .rst(rst), .in(vga_vsync), .out(VGA_VS));

////// BRAM

// the ram_rst signals allow us to clear any pending reads
wire ram_rst;
wire ram_readclk, ram_outclk, ram_we;
wire [clog2(RAM_SIZE)-1:0] ram_raddr, ram_waddr;
wire [BYTE_LEN-1:0] ram_out, ram_win;
```

```
packet_buffer_ram_driver ram_driv_inst(
  .clk(clk), .rst(rst || ram_rst),
  .readclk(ram_readclk), .raddr(ram_raddr),
  .we(ram_we), .waddr(ram_waddr), .win(ram_win),
  .outclk(ram_outclk), .out(ram_out));

wire vram_rst;
wire vram_readclk, vram_outclk, vram_we;
wire [clog2(VRAM_SIZE)-1:0] vram_raddr, vram_waddr;
wire [COLOR_LEN-1:0] vram_out, vram_win;
video_cache_ram_driver vram_driv_inst(
  .clk(clk), .rst(rst || vram_rst),
  .readclk(vram_readclk), .raddr(vram_raddr),
  .we(vram_we), .waddr(vram_waddr), .win(vram_win),
  .outclk(vram_outclk), .out(vram_out));

wire rom_rst, rom_readclk, rom_outclk;
wire [clog2(ROM_SIZE)-1:0] rom_raddr;
wire [BYTE_LEN-1:0] rom_out;
packet_synth_rom_driver psr_inst(
  .clk(clk), .rst(rst || rom_rst),
  .readclk(rom_readclk), .raddr(rom_raddr),
  .outclk(rom_outclk), .out(rom_out));

////// AES

// use switches for last 8 bits of key
wire [BLOCK_LEN-1:0] aes_key;
assign aes_key = {KEY[8+:BLOCK_LEN-8], swkey};

wire aes_rst, aes_inclk, aes_outclk, aes_in_done, aes_done;
wire [BYTE_LEN-1:0] aes_in, aes_out;
wire aes_upstream_readclk, aes_readclk;
wire aes_decr_select;
assign aes_decr_select = !config_transmit;
aes_combined_bytes_buf aes_inst(
  .clk(clk), .rst(rst || aes_rst),
  .inclk(aes_inclk), .in(aes_in), .in_done(aes_in_done),
  .key(aes_key),
  .readclk(aes_readclk),
  .outclk(aes_outclk), .out(aes_out), .done(aes_done),
  .upstream_readclk(aes_upstream_readclk),
  .decr_select(aes_decr_select), .cbc_enable(cbc_enable));

// multiplex aes for encrypt and decrypt paths
```

```
wire aes_encr_rst, aes_decr_rst;
wire aes_encr_inclk, aes_decr_inclk, aes_encr_outclk, aes_decr_outclk;
wire aes_encr_in_done, aes_decr_in_done, aes_encr_done, aes_decr_done;
wire aes_encr_readclk, aes_decr_readclk;
wire aes_encr_upstream_readclk, aes_decr_upstream_readclk;
wire [BYTE_LEN-1:0] aes_encr_in, aes_decr_in, aes_encr_out, aes_decr_out;
assign aes_rst = aes_decr_select ? aes_decr_rst : aes_encr_rst;
assign aes_inclk = aes_decr_select ? aes_decr_inclk : aes_encr_inclk;
assign aes_in = aes_decr_select ? aes_decr_in : aes_encr_in;
assign aes_in_done = aes_decr_select ? aes_decr_in_done : aes_encr_in_done;
assign aes_readclk = aes_decr_select ? aes_decr_readclk : aes_encr_readclk;
assign aes_encr_outclk = aes_decr_select ? 0 : aes_outclk;
assign aes_decr_outclk = aes_decr_select ? aes_outclk : 0;
assign aes_encr_out = aes_decr_select ? 0 : aes_out;
assign aes_decr_out = aes_decr_select ? aes_out : 0;
assign aes_encr_done = aes_decr_select ? 0 : aes_done;
assign aes_decr_done = aes_decr_select ? aes_done : 0;
assign aes_encr_upstream_readclk =
  aes_decr_select ? 0 : aes_upstream_readclk;
assign aes_decr_upstream_readclk =
  aes_decr_select ? aes_upstream_readclk : 0;

////// RAM MULTIPLEXING

// old uart ram interface for debugging, no longer used
wire uart_ram_rst, uart_ram_readclk, uart_ram_outclk;
wire [clog2(RAM_SIZE)-1:0] uart_ram_raddr;
wire [BYTE_LEN-1:0] uart_ram_out;
assign uart_ram_outclk = 0;

wire rx_ram_rst, uart_ram_we, rx_ram_readclk, rx_ram_outclk;
wire [clog2(RAM_SIZE)-1:0] uart_ram_waddr, rx_ram_raddr;
wire [BYTE_LEN-1:0] uart_ram_win, rx_ram_out;
wire ffcp_ram_rst, rx_ram_we, ffcp_ram_readclk, ffcp_ram_outclk;
wire [clog2(RAM_SIZE)-1:0] rx_ram_waddr, ffcp_ram_raddr;
wire [BYTE_LEN-1:0] rx_ram_win, ffcp_ram_out;
assign ram_rst = config_transmit ? ffcp_ram_rst : rx_ram_rst;
assign ram_we = config_transmit ? uart_ram_we : rx_ram_we;
assign ram_waddr = config_transmit ? uart_ram_waddr : rx_ram_waddr;
assign ram_win = config_transmit ? uart_ram_win : rx_ram_win;
assign ram_readclk = config_transmit ? ffcp_ram_readclk : rx_ram_readclk;
assign ram_raddr = config_transmit ? ffcp_ram_raddr : rx_ram_raddr;
assign rx_ram_outclk = config_transmit ? 0 : ram_outclk;
assign ffcp_ram_outclk = config_transmit ? ram_outclk : 0;
assign rx_ram_out = ram_out;
assign ffcp_ram_out = ram_out;
```

```
wire uart_vram_rst;
wire vga_vram_readclk, vga_vram_outclk;
wire [clog2(VRAM_SIZE)-1:0] vga_vram_raddr;
wire [COLOR_LEN-1:0] vga_vram_out;
wire uart_vram_readclk, uart_vram_outclk;
wire [clog2(VRAM_SIZE)-1:0] uart_vram_raddr;
wire [COLOR_LEN-1:0] uart_vram_out;
assign vram_rst = config_transmit ? uart_vram_rst : 0;
assign vram_readclk =
  config_transmit ? uart_vram_readclk : vga_vram_readclk;
assign vram_raddr = config_transmit ? uart_vram_raddr : vga_vram_raddr;
assign vga_vram_outclk = config_transmit ? 0 : vram_outclk;
assign uart_vram_outclk = config_transmit ? vram_outclk : 0;
assign vga_vram_out = vram_out;
assign uart_vram_out = vram_out;

////// RMII

assign ETH_REFCLK = clk;
assign ETH_MDC = 0;
assign ETH_MDIO = 0;
wire rmii_outclk, rmii_done;
wire [1:0] rmii_out;
rmii_driver rmii_driv_inst(
  .clk(clk), .rst(rst),
  .crsdv_in(ETH_CRSDV), .rxd_in(ETH_RXD),
  .rxerr(ETH_RXERR),
  .intn(ETH_INTN), .rstn(ETH_RSTN),
  .out(rmii_out),
  .outclk(rmii_outclk), .done(rmii_done));

wire eth_txen;
wire [1:0] eth_txd;
// buffer the outputs so that eth_txd calculation would be
// under timing constraints
delay eth_txen_delay(
  .clk(clk), .rst(rst), .in(eth_txen), .out(ETH_TXEN));
delay #(.DATA_WIDTH(2)) eth_txd_delay(
  .clk(clk), .rst(rst), .in(eth_txd), .out(ETH_TXD));

////// ETHERNET TX <= RAM

wire ffcp_tx_start;
wire [FFCP_TYPE_LEN-1:0] ffcp_tx_type;
wire [FFCP_INDEX_LEN-1:0] ffcp_tx_index;
```

```
wire [clog2(RAM_SIZE)-1:0] ffcp_tx_sfm_read_start;
wire ffcp_tx_sfm_readclk;
wire ffcp_tx_sfm_outclk, ffcp_tx_sfm_done;
wire [BYTE_LEN-1:0] ffcp_tx_sfm_out;
assign ffcp_ram_rst = ffcp_tx_start;
stream_from_memory #(.RAM_SIZE(RAM_SIZE),
  .RAM_READ_LATENCY(PACKET_BUFFER_READ_LATENCY)) ffcp_tx_sfm_inst(
  .clk(clk), .rst(rst), .start(ffcp_tx_start),
  .read_start(ffcp_tx_sfm_read_start),
  .read_end(ffcp_tx_sfm_read_start + FGP_LEN),
  .readclk(ffcp_tx_sfm_readclk),
  .ram_outclk(ffcp_ram_outclk), .ram_out(ffcp_ram_out),
  .ram_readclk(ffcp_ram_readclk), .ram_raddr(ffcp_ram_raddr),
  .outclk(ffcp_tx_sfm_outclk), .out(ffcp_tx_sfm_out),
  .done(ffcp_tx_sfm_done));

wire ffcp_tx_fgp_offset_outclk;
wire [BYTE_LEN-1:0] ffcp_tx_fgp_offset_out;
wire ffcp_tx_fgp_outclk, ffcp_tx_fgp_done;
wire [BYTE_LEN-1:0] ffcp_tx_fgp_out;
// use an fgp_rx to split the data from the offset since we are
// only encrypting the data
fgp_rx fgp_rx(
  .clk(clk), .rst(rst),
  .inclk(ffcp_tx_sfm_outclk), .in(ffcp_tx_sfm_out),
  .offset_outclk(ffcp_tx_fgp_offset_outclk),
  .offset_out(ffcp_tx_fgp_offset_out),
  .outclk(ffcp_tx_fgp_outclk), .out(ffcp_tx_fgp_out),
  .done(ffcp_tx_fgp_done));
reg fgp_tx_reading_metadata = 0;
always @(posedge clk) begin
  if (rst || ffcp_tx_fgp_offset_outclk)
    fgp_tx_reading_metadata <= 0;
  else if (ffcp_tx_start)
    fgp_tx_reading_metadata <= 1;
end
assign ffcp_tx_sfm_readclk =
  fgp_tx_reading_metadata || aes_encr_upstream_readclk;

assign aes_encr_rst = ffcp_tx_start && ffcp_tx_type == FFCP_TYPE_SYN;
assign aes_encr_inclk = ffcp_tx_fgp_outclk && !fgp_tx_reading_metadata;
assign aes_encr_in = ffcp_tx_fgp_out;
assign aes_encr_in_done = ffcp_tx_fgp_done;

wire fgp_tx_start;
```

```
assign fgp_tx_start = ffcp_tx_fgp_offset_outclk;
// delay the aes readclk since every transmit component should have
// exactly two clock cycles of delay, but aes has no latency
wire aes_encr_readclk_pd;
delay #(.DELAY_LEN(PACKET_SYNTH_ROM_LATENCY)) aes_encr_readclk_delay(
  .clk(clk), .rst(rst || fgp_tx_start),
  .in(aes_encr_readclk_pd), .out(aes_encr_readclk));

// glue the fgp header back on
wire fgp_tx_readclk, fgp_tx_outclk, fgp_tx_done;
wire [BYTE_LEN-1:0] fgp_tx_out;
fgp_tx fgp_tx_inst(
  .clk(clk), .rst(rst),
  .start(fgp_tx_start), .offset(ffcp_tx_fgp_offset_out),
  .inclk(aes_encr_outclk), .in(aes_encr_out),
  .in_done(aes_encr_done),
  .readclk(fgp_tx_readclk),
  .outclk(fgp_tx_outclk), .out(fgp_tx_out), .done(fgp_tx_done),
  .upstream_readclk(aes_encr_readclk_pd));

wire ffcp_tx_readclk, ffcp_tx_outclk, ffcp_tx_done;
wire [BYTE_LEN-1:0] ffcp_tx_out;
ffcp_tx ffcp_tx_inst(
  .clk(clk), .rst(rst), .start(ffcp_tx_start),
  .inclk(fgp_tx_outclk), .in(fgp_tx_out),
  .in_done(fgp_tx_done),
  .ffcp_type(ffcp_tx_type), .ffcp_index(ffcp_tx_index),
  .readclk(ffcp_tx_readclk),
  .outclk(ffcp_tx_outclk), .out(ffcp_tx_out),
  .upstream_readclk(fgp_tx_readclk), .done(ffcp_tx_done));

wire eth_tx_done;
eth_tx eth_tx_inst(
  .clk(clk), .rst(rst), .start(ffcp_tx_start),
  .in_done(ffcp_tx_done),
  .inclk(ffcp_tx_outclk), .in(ffcp_tx_out),
  .ram_outclk(rom_outclk), .ram_out(rom_out),
  .ram_readclk(rom_readclk), .ram_raddr(rom_raddr),
  .outclk(eth_txen), .out(eth_txd),
  .upstream_readclk(ffcp_tx_readclk), .done(eth_tx_done));

////// ETHERNET RX => VRAM

wire eth_rx_downstream_done, eth_rx_outclk, eth_rx_err, eth_rx_done;
wire [BYTE_LEN-1:0] eth_rx_out;
wire eth_rx_ethertype_outclk;
```

```
wire [ETH_ETHERTYPE_LEN*BYTE_LEN-1:0] eth_rx_ethertype_out;
eth_rx eth_rx_inst(
  .clk(clk), .rst(rst),
  .inclk(rmii_outclk), .in(rmii_out),
  .in_done(rmii_done),
  .downstream_done(eth_rx_downstream_done),
  .outclk(eth_rx_outclk), .out(eth_rx_out),
  .ethertype_outclk(eth_rx_ethertype_outclk),
  .ethertype_out(eth_rx_ethertype_out),
  .err(eth_rx_err), .done(eth_rx_done));
// reset all downstream modules when a receive error occurs
wire eth_rx_downstream_rst;
assign eth_rx_downstream_rst = rst || eth_rx_err;

// ffcp_rx_en is asserted (and held until the next packet is received)
// only when the packet is an ffcp packet, acting as a filter
reg ffcp_rx_en = 0;
always @(posedge clk) begin
  if (eth_rx_downstream_rst)
    ffcp_rx_en <= 0;
  else if (eth_rx_ethertype_outclk)
    ffcp_rx_en <= eth_rx_ethertype_out == ETHERTYPE_FFCP;
end
wire ffcp_rx_eth_done;
assign ffcp_rx_eth_done = ffcp_rx_en && eth_rx_done;

wire ffcp_rx_done;
assign eth_rx_downstream_done = ffcp_rx_done;
wire ffcp_rx_metadata_outclk;
wire [FFCP_TYPE_LEN-1:0] ffcp_rx_type;
wire [FFCP_INDEX_LEN-1:0] ffcp_rx_index;
wire ffcp_rx_outclk;
wire [BYTE_LEN-1:0] ffcp_rx_out;
ffcp_rx ffcp_rx_inst(
  .clk(clk), .rst(eth_rx_downstream_rst),
  .inclk(eth_rx_outclk && ffcp_rx_en), .in(eth_rx_out),
  .done(ffcp_rx_done),
  .metadata_outclk(ffcp_rx_metadata_outclk),
  .ffcp_type(ffcp_rx_type), .ffcp_index(ffcp_rx_index),
  .outclk(ffcp_rx_outclk), .out(ffcp_rx_out));

// signals that tell us what kind of ffcp packet we received
wire ffcp_rx_ack_outclk, ffcp_rx_syn_outclk, ffcp_rx_msg_outclk;
assign ffcp_rx_ack_outclk =
  ffcp_rx_metadata_outclk && ffcp_rx_type == FFCP_TYPE_ACK;
assign ffcp_rx_syn_outclk =
```

```
  ffcp_rx_metadata_outclk && ffcp_rx_type == FFCP_TYPE_SYN;
assign ffcp_rx_msg_outclk =
  ffcp_rx_metadata_outclk && ffcp_rx_type == FFCP_TYPE_MSG;

// we no longer need to use the ffcp_queue interface in the receive
// configuration
wire pb_advance_tail_rx, pb_advance_head_rx;
assign pb_advance_tail_rx = 1'b0;
assign pb_advance_head_rx = 1'b0;

// stream the ffcp payload into ram

// records the number of bytes of the ffcp payload (which is an fgp
// datagram) we have received
reg [clog2(FGP_LEN)-1:0] fgp_rx_cnt = 0;
assign rx_ram_waddr = {ffcp_rx_index_buf, fgp_rx_cnt};
assign rx_ram_we = ffcp_rx_outclk;
assign rx_ram_win = ffcp_rx_out;
assign pb_rst_rx = 1'b0;
always @(posedge clk) begin
  if (eth_rx_downstream_rst)
    fgp_rx_cnt <= 0;
  else if (ffcp_rx_eth_done)
    fgp_rx_cnt <= 0;
  else if (ffcp_rx_outclk)
    fgp_rx_cnt <= fgp_rx_cnt + 1;
end

// the signal that we send to the ffcp_rx_server to inform it of a syn
// this is different from the syn_outclk because we need to assert this
// at the same time as when we assert the ffcp_rx_server's inclk, which
// only happens when the complete ethernet frame is received
wire ffcp_rx_serv_syn;
// if the ffcp_rx_server receives a syn, then we have started a new stream,
// so all downstream modules should be reset
wire ffcp_rx_serv_downstream_rst;
assign ffcp_rx_serv_downstream_rst = rst || ffcp_rx_serv_syn;

// signals related to the ffcp_rx_server
wire ffcp_rx_commit, ffcp_rx_commit_done;
wire [clog2(FFCP_BUFFER_LEN)-1:0] ffcp_rx_commit_index;

// at some point, the ffcp_rx_server will commit the packet
// and we stream the packet from the packet buffer to VRAM
wire [clog2(RAM_SIZE)-1:0] ffcp_rx_sfm_read_start;
assign ffcp_rx_sfm_read_start =
```

```
  {ffcp_rx_commit_index, {clog2(FGP_LEN){1'b0}}};
wire ffcp_rx_sfm_readclk, ffcp_rx_sfm_outclk;
wire [BYTE_LEN-1:0] ffcp_rx_sfm_out;
stream_from_memory #(.RAM_SIZE(RAM_SIZE),
  .RAM_READ_LATENCY(PACKET_BUFFER_READ_LATENCY)) ffcp_rx_sfm_inst(
  .clk(clk), .rst(ffcp_rx_serv_downstream_rst), .start(ffcp_rx_commit),
  .read_start(ffcp_rx_sfm_read_start),
  .read_end(ffcp_rx_sfm_read_start + FGP_LEN),
  .readclk(ffcp_rx_sfm_readclk),
  .ram_outclk(rx_ram_outclk), .ram_out(rx_ram_out),
  .ram_readclk(rx_ram_readclk), .ram_raddr(rx_ram_raddr),
  .outclk(ffcp_rx_sfm_outclk), .out(ffcp_rx_sfm_out),
  .done(ffcp_rx_commit_done));
assign rx_ram_rst = ffcp_rx_serv_downstream_rst;
assign ffcp_rx_sfm_readclk = aes_decr_upstream_readclk;

// the packets are stored in the packet buffer along with the fgp header
// we use the fgp header to determine where to stream the data to in vram

wire fgp_rx_setoff_req;
wire [BYTE_LEN+clog2(FGP_DATA_LEN_COLORS)-1:0] fgp_rx_setoff_val;
wire fgp_rx_outclk;
wire [BYTE_LEN-1:0] fgp_rx_out, fgp_rx_offset_out;
wire fgp_rx_done;
fgp_rx fgp_rx_inst(
  .clk(clk), .rst(ffcp_rx_serv_downstream_rst),
  .inclk(ffcp_rx_sfm_outclk && !config_transmit), .in(ffcp_rx_sfm_out),
  .done(fgp_rx_done),
  .offset_outclk(fgp_rx_setoff_req), .offset_out(fgp_rx_offset_out),
  .outclk(fgp_rx_outclk), .out(fgp_rx_out));
assign fgp_rx_setoff_val = {fgp_rx_offset_out,
  {clog2(FGP_DATA_LEN_COLORS){1'b0}}};

assign aes_decr_rst = ffcp_rx_serv_downstream_rst;
assign aes_decr_inclk = fgp_rx_outclk;
assign aes_decr_in = fgp_rx_out;
assign aes_decr_readclk = 1'b1;

wire fgp_btc_outclk;
wire [COLOR_LEN-1:0] fgp_btc_out;
bytes_to_colors fgp_btc_inst(
  .clk(clk), .rst(ffcp_rx_serv_downstream_rst),
  .inclk(aes_decr_outclk), .in(aes_decr_out),
  .outclk(fgp_btc_outclk), .out(fgp_btc_out));
assign aes_decr_in_done = 1'b0;
stream_to_memory
```

```
  #(.RAM_SIZE(VRAM_SIZE), .WORD_LEN(COLOR_LEN)) fgp_stm_inst(
  .clk(clk), .rst(ffcp_rx_serv_downstream_rst),
  .setoff_req(fgp_rx_setoff_req),
  .setoff_val(fgp_rx_setoff_val[clog2(VRAM_SIZE)-1:0]),
  .inclk(fgp_btc_outclk), .in(fgp_btc_out),
  .ram_we(vram_we), .ram_waddr(vram_waddr),
  .ram_win(vram_win));

////// FFCP FLOW CONTROL (RECEIVE CONFIGURATION)

// buffer ffcp header information so that we can provide them to the
// ffcp_rx_server after the complete ethernet frame has been received
reg ffcp_syn_buf;
reg [FFCP_INDEX_LEN-1:0] ffcp_rx_index_buf;
always @(posedge clk) begin
  if (ffcp_rx_syn_outclk || ffcp_rx_msg_outclk) begin
    ffcp_rx_index_buf <= ffcp_rx_index;
    ffcp_syn_buf <= ffcp_rx_syn_outclk;
  end
end

wire ffcp_ack_start;
wire [FFCP_INDEX_LEN-1:0] ffcp_ack_index;
assign ffcp_rx_serv_syn = ffcp_rx_eth_done && ffcp_syn_buf;
ffcp_rx_server ffcp_rx_serv_inst(
  .clk(clk), .rst(rst), .syn(ffcp_rx_serv_syn),
  .inclk(ffcp_rx_eth_done),
  .in_index(ffcp_rx_index_buf),
  .downstream_done(eth_tx_done),
  .commit(ffcp_rx_commit), .commit_index(ffcp_rx_commit_index),
  .commit_done(ffcp_rx_commit_done),
  .outclk(ffcp_ack_start), .out_index(ffcp_ack_index));

////// FFCP FLOW CONTROL (TRANSMIT CONFIGURATION)

wire [clog2(PB_QUEUE_LEN)-1:0] pb_head, pb_tail;
// pb_advance_head_rx, pb_advance_tail_rx and pb_rst_rx
// already declared earlier
wire pb_advance_tail, pb_advance_tail_tx;
wire pb_advance_head, pb_advance_head_tx;
wire pb_rst, pb_rst_tx;
assign pb_advance_tail =
  config_transmit ? pb_advance_tail_tx : pb_advance_tail_rx;
assign pb_advance_head =
  config_transmit ? pb_advance_head_tx : pb_advance_head_rx;
assign pb_rst = config_transmit ? pb_rst_tx : pb_rst_rx;
```

```
wire pb_inclk;
wire [clog2(PB_QUEUE_LEN)-1:0] pb_in_head;
wire pb_almost_full;
ffcp_queue ffcp_queue_inst(
  .clk(clk), .rst(rst || pb_rst),
  .advance_head(pb_advance_head), .advance_tail(pb_advance_tail),
  .inclk(pb_inclk && config_transmit), .in_head(pb_in_head),
  .almost_full(pb_almost_full),
  .head(pb_head), .tail(pb_tail));
wire ffcp_msg_start;
wire [FFCP_INDEX_LEN-1:0] ffcp_msg_index;
wire ffcp_tx_syn;
wire [clog2(PB_QUEUE_LEN)-1:0] ffcp_tx_buf_pos;
ffcp_tx_server ffcp_tx_serv_inst(
  .clk(clk), .rst(rst),
  .pb_head(pb_head), .pb_tail(pb_tail),
  .downstream_done(eth_tx_done),
  .inclk(ffcp_rx_ack_outclk), .in_index(ffcp_rx_index),
  .outclk(ffcp_msg_start), .out_syn(ffcp_tx_syn),
  .out_index(ffcp_msg_index),
  .out_buf_pos(ffcp_tx_buf_pos),
  .outclk_pb(pb_inclk), .out_pb_head(pb_in_head));
assign pb_rst_tx = 1'b0;
assign ffcp_tx_sfm_read_start = {ffcp_tx_buf_pos, {clog2(FGP_LEN){1'b0}}};

assign ffcp_tx_start = config_transmit ? ffcp_msg_start : ffcp_ack_start;
assign ffcp_tx_type = config_transmit ?
  (ffcp_tx_syn ? FFCP_TYPE_SYN : FFCP_TYPE_MSG) : FFCP_TYPE_ACK;
assign ffcp_tx_index = config_transmit ? ffcp_msg_index : ffcp_ack_index;

////// UART RX => RAM (TRANSMIT CONFIGURATION)

wire uart_cts;
// request laptop to stop transmitting when the packet buffer is
// almost full, or when we manually override cts
assign uart_cts = sw2 || pb_almost_full;
assign UART_CTS = uart_cts;
wire [7:0] uart_rx_out;
wire uart_rx_outclk;
uart_rx_fast_driver uart_rx_inst(
  .clk(clk), .clk_120mhz(clk_120mhz), .rst(rst),
  .rxd(UART_TXD_IN), .out(uart_rx_out), .outclk(uart_rx_outclk));
wire uart_rx_active;
// reset downstream modules if nothing is received for 1ms, and not
// because we told upstream to pause transmitting
pulse_extender #(.EXTEND_LEN(50000)) uart_rx_active_pe(
```

```
  .clk(clk), .rst(rst),
  .in(uart_rx_outclk || uart_cts), .out(uart_rx_active));
wire uart_rx_downstream_rst;
assign uart_rx_downstream_rst = rst || !uart_rx_active;

// streams data from uart to ram
reg [clog2(FGP_LEN)-1:0] uart_rx_cnt = 0;
assign uart_ram_waddr = {pb_tail, uart_rx_cnt};
assign uart_ram_we = uart_rx_outclk;
assign uart_ram_win = uart_rx_out;
// when we have received a complete fgp packet, advance the tail and
// start streaming to the next partition in the packet buffer queue
assign pb_advance_tail_tx = uart_rx_outclk && uart_rx_cnt == FGP_LEN-1;
always @(posedge clk) begin
  if (uart_rx_downstream_rst)
    uart_rx_cnt <= 0;
  else if (uart_rx_outclk)
    uart_rx_cnt <= pb_advance_tail_tx ? 0 : uart_rx_cnt + 1;
end
assign pb_advance_head_tx = 1'b0;

////// UART TX <= RAM/VRAM

wire uart_tx_inclk, uart_tx_readclk;
wire [BYTE_LEN-1:0] uart_tx_in;
wire uart_tx_start;
assign uart_tx_start = btnc;
// if config_transmit is set, stream debug output from vram
// otherwise, stream it from ram
wire uart_sfm_ram_readclk;
wire [clog2(RAM_SIZE)-1:0] uart_sfm_ram_raddr;
stream_from_memory uart_sfm_inst(
  .clk(clk), .rst(rst), .start(uart_tx_start),
  .read_start(0), .read_end(config_transmit ? VRAM_SIZE : RAM_SIZE),
  .readclk(uart_tx_readclk),
  .ram_outclk(config_transmit ? uart_vram_outclk : uart_ram_outclk),
  .ram_out(config_transmit ? uart_vram_out[BYTE_LEN-1:0] : uart_ram_out),
  .ram_readclk(uart_sfm_ram_readclk), .ram_raddr(uart_sfm_ram_raddr),
  .outclk(uart_tx_inclk), .out(uart_tx_in));
assign uart_vram_readclk = uart_sfm_ram_readclk;
assign uart_ram_readclk = uart_sfm_ram_readclk;
assign uart_vram_raddr = uart_sfm_ram_raddr;
assign uart_ram_raddr = uart_sfm_ram_raddr;
assign uart_vram_rst = uart_tx_start;
assign uart_ram_rst = uart_tx_start;
uart_tx_fast_stream_driver uart_tx_inst(
```

```
  .clk(clk), .clk_120mhz(clk_120mhz), .rst(rst), .start(uart_tx_start),
  .inclk(uart_tx_inclk), .in(uart_tx_in), .txd(UART_RXD_OUT),
  .upstream_readclk(uart_sfm_readclk));
assign uart_tx_readclk = 1'b1;

////// VRAM => VGA (RECEIVE CONFIGURATION)

graphics_main graphics_main_inst(
  .clk(clk), .rst(rst), .blank(blank),
  .vga_x(vga_x), .vga_y(vga_y),
  .vga_hsync_in(vga_hsync_predelay), .vga_vsync_in(vga_vsync_predelay),
  .ram_outclk(vga_vram_outclk), .ram_out(vga_vram_out),
  .ram_readclk(vga_vram_readclk), .ram_raddr(vga_vram_raddr),
  .vga_col(vga_col),
  .vga_hsync_out(vga_hsync), .vga_vsync_out(vga_vsync));

////// DEBUGGING SIGNALS

wire aes_rst_pe;
pulse_extender aes_rst_pe_inst(
  .clk(clk), .rst(rst), .in(aes_rst), .out(aes_rst_pe));

wire blink;
blinker blinker_inst(
  .clk(clk), .rst(rst),
  .enable(1), .out(blink));

assign LED = {
  SW[15:6],
  blink,
  aes_rst_pe,
  cbc_enable,
  uart_cts,
  config_transmit,
  rst
};

assign hex_display_data = {
  pb_head[3:0],
  ram_raddr[clog2(RAM_SIZE)-12+:12],
  pb_tail[3:0],
  ram_waddr[clog2(RAM_SIZE)-12+:12]
};

assign JB = {
  8'h0
```

```
};

endmodule
```

B Simulation Code

test_full.v:

```
‘timescale 1ns / 1ps

module test_daisy_chain();

‘include "networking.vh"
‘include "packet_synth_rom_layout.vh"

reg clk_100mhz = 0;
initial forever #5 clk_100mhz = ~clk_100mhz;

reg clk_50mhz = 0;
initial forever #10 clk_50mhz = ~clk_50mhz;

reg clk_120mhz = 0;
initial forever #4.16667 clk_120mhz = ~clk_120mhz;

reg uart_rst = 1;

wire uart_rom_rst, uart_rom_readclk, uart_rom_outclk;
wire [clog2(PACKET_SYNTH_ROM_SIZE)-1:0] uart_rom_raddr;
wire [BYTE_LEN-1:0] uart_rom_out;
packet_synth_rom_driver uart_psr_inst(
  .clk(clk_50mhz), .rst(uart_rst || uart_rom_rst),
  .readclk(uart_rom_readclk), .raddr(uart_rom_raddr),
  .outclk(uart_rom_outclk), .out(uart_rom_out));
wire uart_tx_inclk, uart_tx_readclk;
wire [BYTE_LEN-1:0] uart_tx_in;
wire uart_tx_start;
assign uart_rom_rst = uart_tx_start;
stream_from_memory uart_sfm_inst(
  .clk(clk_50mhz), .rst(uart_rst), .start(uart_tx_start),
  .read_start(SAMPLE_PAYLOAD_OFF),
  .read_end(SAMPLE_PAYLOAD_OFF + SAMPLE_PAYLOAD_LEN),
  .readclk(uart_tx_readclk),
  .ram_outclk(uart_rom_outclk), .ram_out(uart_rom_out),
  .ram_readclk(uart_rom_readclk), .ram_raddr(uart_rom_raddr),
  .outclk(uart_tx_inclk), .out(uart_tx_in));
wire uart_txd;
```

```
uart_tx_fast_stream_driver uart_tx_inst(
  .clk(clk_50mhz), .clk_120mhz(clk_120mhz), .rst(uart_rst),
  .start(uart_tx_start),
  .inclk(uart_tx_inclk), .in(uart_tx_in), .txd(uart_txd),
  .upstream_readclk(uart_tx_readclk));

reg [13:0] sw_prefix = 14'b0100_1110_0000_10;
reg rst = 1;
wire [15:0] sw_tx, sw_rx;
assign sw_tx = {sw_prefix, 1'b1, rst};
assign sw_rx = {sw_prefix, 1'b0, rst};
wire eth_tx_crsdv, eth_rx_crsdv;
wire [1:0] eth_tx_rxd, eth_rx_rxd;

main main_inst_tx(
  .CLK100MHZ(clk_100mhz), .SW(sw_tx),
  .BTNC(1'b0), .BTNU(1'b0), .BTNL(1'b0), .BTNR(1'b0), .BTND(1'b0),
  .ETH_CRSDV(eth_tx_crsdv), .ETH_RXD(eth_tx_rxd),
  .ETH_TXEN(eth_rx_crsdv), .ETH_TXD(eth_rx_rxd),
  .UART_TXD_IN(uart_txd), .UART_RTS(1'b0));
main main_inst_rx(
  .CLK100MHZ(clk_100mhz), .SW(sw_rx),
  .BTNC(1'b0), .BTNU(1'b0), .BTNL(1'b0), .BTNR(1'b0), .BTND(1'b0),
  .ETH_CRSDV(eth_rx_crsdv), .ETH_RXD(eth_rx_rxd),
  .ETH_TXEN(eth_tx_crsdv), .ETH_TXD(eth_tx_rxd),
  .UART_TXD_IN(1'b1), .UART_RTS(1'b0));

reg uart_tx_start_manual = 0;
assign uart_tx_start = uart_tx_start_manual;

initial begin
  #2000
  uart_rst = 0;
  rst = 0;
  #400

  uart_tx_start_manual = 1;
  #20
  uart_tx_start_manual = 0;
  // 12mbaud = 84ns per bit, for a frame of 914 bytes
  // multiply by 2 to account for overhead
  #(2 * 84 * 914 * 8)

  uart_tx_start_manual = 1;
  #20
  uart_tx_start_manual = 0;
```

```
  #(2 * 84 * 914 * 8)

  // test resyn
  #2000000

  uart_tx_start_manual = 1;
  #20
  uart_tx_start_manual = 0;
  #(2 * 84 * 914 * 8)

  uart_tx_start_manual = 1;
  #20
  uart_tx_start_manual = 0;
  #(2 * 84 * 914 * 8)

  $stop();
end

endmodule
```

C Python Code

C.1 Libraries

fpga_serial.py (modified from the one provided in previous labs):

```
'''Automatically find USB Serial Port
jodalyst 9/2017
'''

import serial.tools.list_ports

def get_usb_port():
  usb_port = list(serial.tools.list_ports.grep('USB-Serial Controller'))
  if len(usb_port) == 1:
    print('Automatically found USB-Serial Controller: {}'.format(usb_port[0].description))
    return usb_port[0].device
  else:
    ports = list(serial.tools.list_ports.comports())
    port_dict = {i:[ports[i],ports[i].vid] for i in range(len(ports))}
    usb_id = None
    for p in port_dict:
      print('{}:   {} (Vendor ID: {})'.format(p,port_dict[p][0],port_dict[p][1]))
      if port_dict[p][1]==1027:
        usb_id = p
    if usb_id == None:
```

```
      return None
    else:
      print('USB-Serial Controller: Device {}'.format(p))
      return port_dict[usb_id][0].device

def do_serial(callback):
  serial_port = get_usb_port()
  if serial_port is None:
    raise Exception('USB-Serial Controller Not Found')

  with serial.Serial(port = serial_port,
    # 12mbaud
    baudrate=12000000,
    parity=serial.PARITY_NONE,
    stopbits=serial.STOPBITS_ONE,
    bytesize=serial.EIGHTBITS,
    # enable rts/cts processing
    rtscts=True,
    timeout=0) as ser:

    print(ser)
    print('Serial Connected!')

    if ser.isOpen():
      print(ser.name + ' is open...')

    callback(ser)
```

image_bytes.py:

```
from PIL import Image

def colors_to_bytes(arr):
  res = []
  for i in range(len(arr)//2):
    col1, col2 = arr[i*2], arr[i*2+1]
    res += [
      col1 >> 4,
      ((col1 & 0xf) << 4) | (col2 >> 8),
      col2 & 0xff
    ]
  return bytes(res)

def image_to_colors(fin_name, width, height):
  im = Image.open(fin_name).convert('RGB')
  im = im.resize((width, height))
  colors = []
```

```
  for i in range(height):
    for j in range(width):
      r, g, b = im.getpixel((j, i))
      r >>= 4
      g >>= 4
      b >>= 4
      colors += [(r << 8) | (g << 4) | b]
  im.close()
  return colors

def image_to_bytestream(fin_name, width, height):
  return colors_to_bytes(image_to_colors(fin_name, width, height))
```

crc32.py:

```
def reflect(x):
  res = 0
  for i in range(32):
    res = (res << 1) | ((x >> i) & 1)
  return res

def reflect_bytes(x):
  res = 0
  for i in range(4):
    res = (res << 8) | ((x >> (i*8)) & 0xff)
  return res

POLY = reflect(0x04c11db7)
INIT = reflect(0xffffffff)
MASK = 0xffffffff

def crc(frame):
  curr = INIT
  for i in range(len(frame) * 8):
    curr_bit = (frame[i//8] >> (i%8)) & 1
    curr ^= curr_bit
    multiple = POLY if (curr & 1) == 1 else 0
    curr = ((curr >> 1) ^ multiple) & MASK
  return reflect_bytes((~curr) & MASK)
```

eth.py:

```
from socket import *
import image_bytes
import crc32

MAC_LEN = 6
```

```
MAC_ZERO = bytes.fromhex('000000000000')
MAC_SEND = bytes.fromhex('DEADBEEFCAFE')
MAC_RECV = bytes.fromhex('C0FFEEDAD101')
MAC_BROADCAST = bytes.fromhex('FFFFFFFFFFFF')

ETHERTYPE_LEN = 2
ETHERTYPE_FGP = bytes.fromhex('ca11')
ETHERTYPE_FFCP = bytes.fromhex('ca12')
ETHERTYPE_IP = bytes.fromhex('0800')
ETHERTYPE_ARP = bytes.fromhex('0806')

HEADER_LEN = 2 * MAC_LEN + ETHERTYPE_LEN

def gen_eth_body(dst, src, eth_type, payload):
  assert(len(src) == MAC_LEN and len(dst) == MAC_LEN)
  assert(len(eth_type) == ETHERTYPE_LEN)
  return dst + src + eth_type + payload

def gen_eth(dst, src, eth_type, payload):
  body = gen_eth_body(dst, src, eth_type, payload)
  crc = crc32.crc(body)
  return body + bytes([
    crc >> 24,
    (crc >> 16) & 0xff,
    (crc >> 8) & 0xff,
    crc & 0xff
  ])

# fpga to fpga
def gen_eth_f2f(eth_type, payload):
  return gen_eth(MAC_RECV, MAC_SEND, eth_type, payload)

def gen_eth_fgp_payload(offset, colors):
  assert(len(colors) == 512)
  return (bytes([offset//512]) +
    image_bytes.colors_to_bytes(colors))

def gen_eth_fgp(offset, colors):
  return gen_eth_f2f(ETHERTYPE_FGP,
    gen_eth_fgp_payload(offset, colors))

def sendeth(frame, interface = "enp2s0"):
  s = socket(AF_PACKET, SOCK_RAW)
  s.bind((interface, 0))
  # remove crc
  return s.send(frame[:-4])
```

```
def get_ethertype(frame):
  return frame[2*MAC_LEN:2*MAC_LEN+2]

def get_src_mac(frame):
  return frame[MAC_LEN:2*MAC_LEN]
```

C.2 COE Generation

generate_debug_coe.py:

```
NUM_ELEMENTS = 16384;
arr = list(range(NUM_ELEMENTS))

with open('debug.coe', 'w') as f:
  f.write('memory_initialization_radix=16;\n')
  f.write('memory_initialization_vector=\n')
  for i in range(NUM_ELEMENTS):
    f.write('%02x%c\n' % (arr[i] % 2**8,
      ',' if i != NUM_ELEMENTS - 1 else ';'))
```

generate_packet_synth_coe.py (this also generates the packet_synth_rom_layout.vh header file):

```
import sys
sys.path.append('../lib/')
import os
import eth
import image_bytes

sample_image_data = list(range(512))
# arbitrarily choose an offset of 6 for testing
sample_payload = eth.gen_eth_fgp_payload(6*512, sample_image_data)
sample_frame = eth.gen_eth_f2f(eth.ETHERTYPE_FGP, sample_payload)

class Memory:
  def __init__(self):
    self.curr_bytes = []
    self.curr_index = 0

  def append(self, val):
    off = self.curr_index
    self.curr_bytes += val
    self.curr_index += len(val)
    return off

mem = Memory()
```

```
mac_send_off = mem.append(eth.MAC_SEND)
mac_recv_off = mem.append(eth.MAC_RECV)
ethertype_fgp_off = mem.append(eth.ETHERTYPE_FGP)
ethertype_ffcp_off = mem.append(eth.ETHERTYPE_FFCP)
sample_frame_off = mem.append(sample_frame)
sample_payload_off = mem.append(sample_payload)
sample_image_data_off = mem.append(sample_image_data)

NUM_ELEMENTS = 4096;
arr = list(range(NUM_ELEMENTS))

def write_localparam(f, key, val):
  f.write('localparam %s = %d;\n' % (key, val))

rom_layout_filename = os.path.join(
  os.path.dirname(__file__), '../hdl/inc/packet_synth_rom_layout.vh')
with open(rom_layout_filename, 'w') as f:
  f.write('// This is a generated file. DO NOT edit this directly.\n\n')
  f.write('// sender and receiver MAC addresses\n')
  write_localparam(f, 'MAC_SEND_OFF', mac_send_off)
  write_localparam(f, 'MAC_RECV_OFF', mac_recv_off)
  f.write('\n')
  write_localparam(f, 'ETHERTYPE_FGP_OFF', ethertype_fgp_off)
  write_localparam(f, 'ETHERTYPE_FFCP_OFF', ethertype_ffcp_off)
  f.write('\n')
  write_localparam(f, 'SAMPLE_PAYLOAD_OFF', sample_payload_off)
  write_localparam(f, 'SAMPLE_PAYLOAD_LEN', len(sample_payload))
  write_localparam(f, 'SAMPLE_FRAME_OFF', sample_frame_off)
  write_localparam(f, 'SAMPLE_FRAME_LEN', len(sample_frame))
  write_localparam(f, 'SAMPLE_IMG_DATA_OFF', sample_image_data_off)
  write_localparam(f, 'SAMPLE_IMG_DATA_LEN', len(sample_image_data))

for i in range(len(mem.curr_bytes)):
  arr[i] = mem.curr_bytes[i]

with open('packet_synth.coe', 'w') as f:
  f.write('memory_initialization_radix=16;\n')
  f.write('memory_initialization_vector=\n')
  for i in range(NUM_ELEMENTS):
    f.write('%02x%c\n' % (arr[i] % 2**8,
      ',' if i != NUM_ELEMENTS - 1 else ';'))
```

C.3 Communication with FPGA

serial-receiver.py:

```
import sys
sys.path.append('../lib/')
import fpga_serial

def listen(ser):
  with open('dump.log', 'wb') as f:
    while True:
      data = ser.read(128)
      if len(data) > 0:
        print('received %d bits' % len(data))
        f.write(data)
        f.flush()

fpga_serial.do_serial(listen)
```

serial-send-cycle.py:

```
import sys
import os
import os.path
import time
sys.path.append('../lib/')
import eth
import image_bytes
import fpga_serial

STOP_EARLY = False
image_dir = 'images/nyan/'
# image_dir = 'images/rickroll/'
IMAGE_WIDTH = 128
IMAGE_HEIGHT = 128
FRAME_PERIOD = 1/12

def send_cycle(ser):
  # Only cycle a few times if testing (i.e. STOP_EARLY == True)
  cnt = 0
  images = sorted(os.listdir(image_dir))
  prev_time = time.time()
  while True:
    if STOP_EARLY and cnt == 5:
      break
    for fin_name in images:
      im = image_bytes.image_to_colors(
        os.path.join(image_dir, fin_name),
        IMAGE_WIDTH, IMAGE_HEIGHT)
      # im = [a % 256 for a in list(range(128*128))]
      for i in range(len(im)//512):
```

```
        num_written = ser.write(
          eth.gen_eth_fgp_payload(i*512, im[i*512:(i+1)*512]))
        # uncomment this to transmit one packet per second
        # ser.flush()
        # time.sleep(1)
      # only flush once per complete frame for better throughput
      ser.flush()
      # print("%d bytes written" % num_written)
      curr_time = time.time()
      time_diff = curr_time - prev_time
      if time_diff < FRAME_PERIOD:
        time.sleep(FRAME_PERIOD - time_diff)
      prev_time = curr_time
    cnt = cnt + 1

fpga_serial.do_serial(send_cycle)
```