# UC BERKELEY

**CS150 Components and Design Techniques for Digital Systems**

**Spring 2008 Project: Wireless Videoconferencing**
**Prof. Pister**
**Prof. McGeer**

**TEAM ICHIBAN: Timothy Liu and Bradley Lyons**
**May 5, 2008**

# Table of Contents

# Abstract

The Spring 2008 CS150 project was a wireless videoconferencing system built on a CaLinx2 board featuring a Xilinx Virtex-E FPGA. The project focused on using the FPGA to rapidly design a system to communicate between the Micron MT48LC16M16A2TG -7E SDRAM chips, the Analog Devices ADV7194 video encoder and ADV7185 video decoder, and the Chipcon CC2420 RF transceiver. The final product allowed a user to see locally sampled video, video received over the wireless protocol, and a GUI displaying information about the current operating configuration. The full project took seven weeks from introduction to final completion.

# Overview

The project was split up into four separate checkpoints. To begin the project, we wrote a SDRAM controller. A bug-free, fully verified memory controller was absolutely essential because any flaws would result a cascade of failures through the whole project.

The second checkpoint consists of three modules. The main module, an arbiter, interfaces with the memory controller and handles the read and write requests of the other two modules, the video processor and camera processor. The video processor buffers data read from memory and outputs it to the video encoder, which subsequently sends the appropriate data to the video screen. The camera processor buffers data from the camera and writes to memory once a full burst of data is in the buffer.

For the third checkpoint, the details of interfacing with the CC2420 are abstracted away with the Transceiver module. It handles the CC2420's initialization, configuration, and all the steps required to send and receive data. The Transceiver asserts Ready when it is able to service a new packet transmission. It also asserts EndSession when a valid packet has been received and put on the BigOut bus. Much like the memory controller, a reliable implementation allows us to use its interface to easily send and receive data without having to worry about the complex details of interfacing with the CC2420.

The final checkpoint involved tying the previous three checkpoints together, along with a new datapath to compress, send, receive, and decompress video data. First, the arbiter was modified to handle two read and two write requests, with a wireless sending processor reading from memory and outputting to the Transceiver, and a wireless receive processor writing data to memory. The wireless send processor is connected to a datapath that compresses the video data and converts it into 248-bit packets for wireless transmission. When we receive a new packet, it is decompressed and sent to the wireless receive processor, which writes the received video data to memory.

# Checkpoint 1: SDRAM

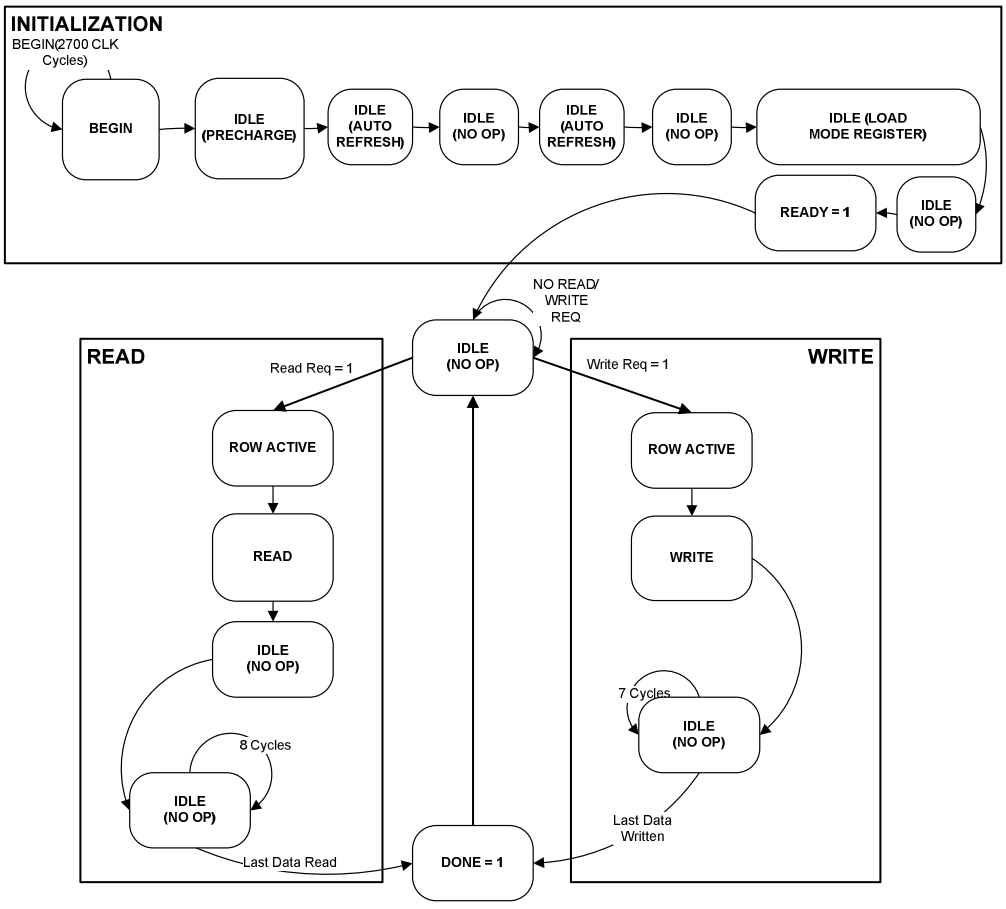
## Objective

The objective of this checkpoint was to develop a module to initialize the CaLinx2's SDRAM chips with our desired configuration and then handle the timing and control signals necessary for issuing read and write requests.

## SDRAM Controller

After a reset, the SDRAM controller will do a one-time initialization of the SDRAM chips, configure their operation, and then assert a Ready signal to indicate that the SDRAM chips are ready for operation. At this point, the memory controller will be ready for either a read or write request. Because the SDRAM arbiter is the only module that interfaces with the memory controller, we were able to assume that the memory controller would not receive a write and read request in the same cycle. On a memory read, the controller will assert the DataValid signal in order to indicate that a valid word has been read from memory and put on the data bus. The DataValid signal's timing allows it to be tied to the write enable port on a FIFO, simplifying the logic necessary to store the data read from memory. A write request requires more logic, as the issuer must correctly use the OutputEnable signal to switch the direction of the tri-state bus between the SDRAM and the memory controller. A Done signal is used to signal that the last word of data is being written or read during a burst and the memory controller will be ready for a new command on the next cycle. This signal is used by the arbiter to determine when a read or write has been completed.

The SDRAM is configured to use eight-word bursts when reading and writing. This reduces the frequency of read and write requests within the memory subsystem.



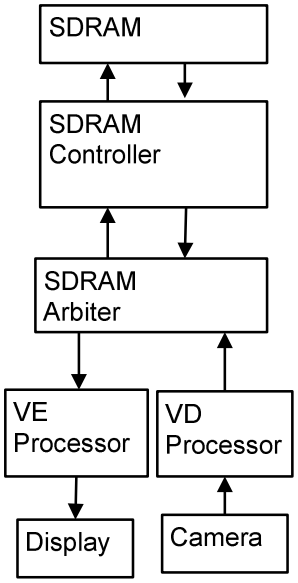
# Checkpoint 2: Local Video System

## Objective

Because we will be simultaneously sending data to the video encoder to display on the screen and writing data to memory from the camera, an arbiter will be necessary to handle the simultaneous read and write requests of the two modules. Also, a module will be necessary for buffering data from the camera and for buffering the data that will be sent to the video encoder.

## SDRAM Arbiter

The arbiter is a relatively simple module that waits for a read or write request, then multiplexes the appropriate signals to the SDRAM controller when the read or write begins. At the end of the burst, the arbiter pulses an enable signal to the address counter in the VD or VE processor.

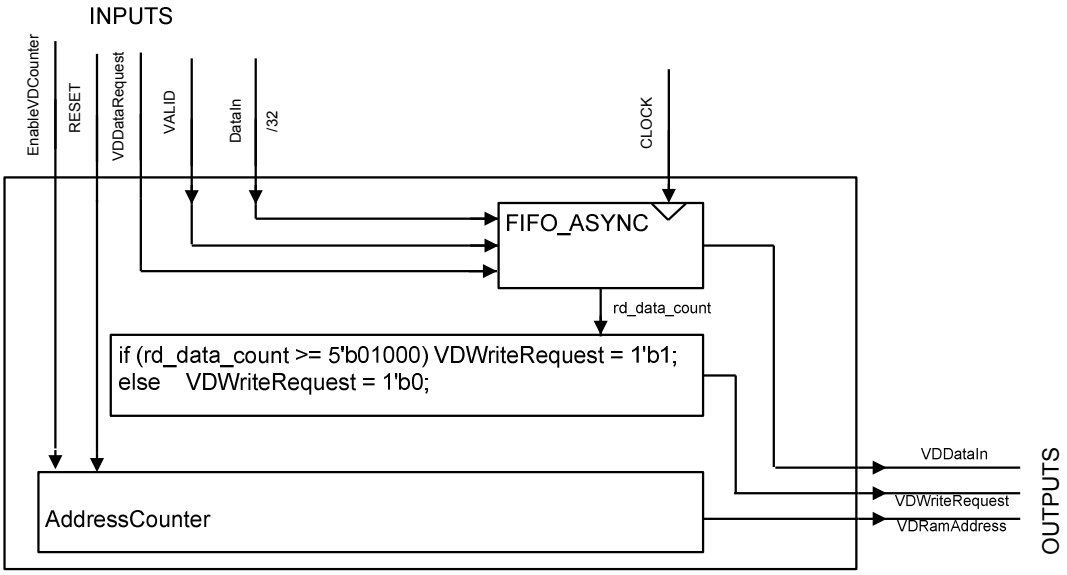



## VEProcessor

The video encoder, which takes data in the YCbYCr format and sends it to the LCD, was modified to use the ITU 601 standard, which specifies 507 lines of active video and 720 pixels per line. The use of this standard allowed us to buffer data from memory in the correct order before the video encoder requests it. We represent this order in our address counter. To buffer the data read from RAM before being sent to the video encoder, we used a FIFO. The read request signal to the arbiter is asserted when the FIFO's data count indicates that it is low on data.

## VDProcessor

This module is very similar to the VEProcessor, simply working in the opposite order. Data from the video decoder is written into a FIFO; once we have a full burst of data to be sent, a write request signal is asserted. An identical address counter is used because our video encoder and decoder were both modified to follow the same ITU 601 standard.




## AddressCounter

The AddressCounter is used to determine the address in memory that will be read from or written to when the VE or VD processor signals a read or write request. In both modules, the address counter is incremented by an enable signal from the arbiter at the end of a read or write burst. When the enable signal is asserted, the counter increments by eight because of our use of eight-word bursts when reading and writing from memory.

# Checkpoint 3: Wireless Transceiver

## Objective

The Transceiver module is meant to provide a reliable, simple interface to the CC2420 wireless transceiver on our CaLinx2 board. The process of receiving data will be abstracted away into a single BigOut bus and a EndSession signal to indicate that the data on that bus is valid. A Ready-Start handshake will be used to communicate when the Transceiver is ready to send data and when the module using the Transceiver has a valid packet of data to send.

## CC2420 Transceiver

The Transceiver starts by initializing the CC2420 by enabling its voltage regulator, pulsing its reset signal, activating its crystal oscillator, then configuring its control registers. Because our Transceiver module would handle address recognition, we disabled hardware address recognition. The CC2420 was then set to the channel from the DIP switches on the CaLinx2 board. The last important change was to set the threshold of the FIFOP signal to 37 bytes, the size of a single packet of data in our protocol. In this configuration, the FIFOP signal will be asserted when we have received a full packet of data.

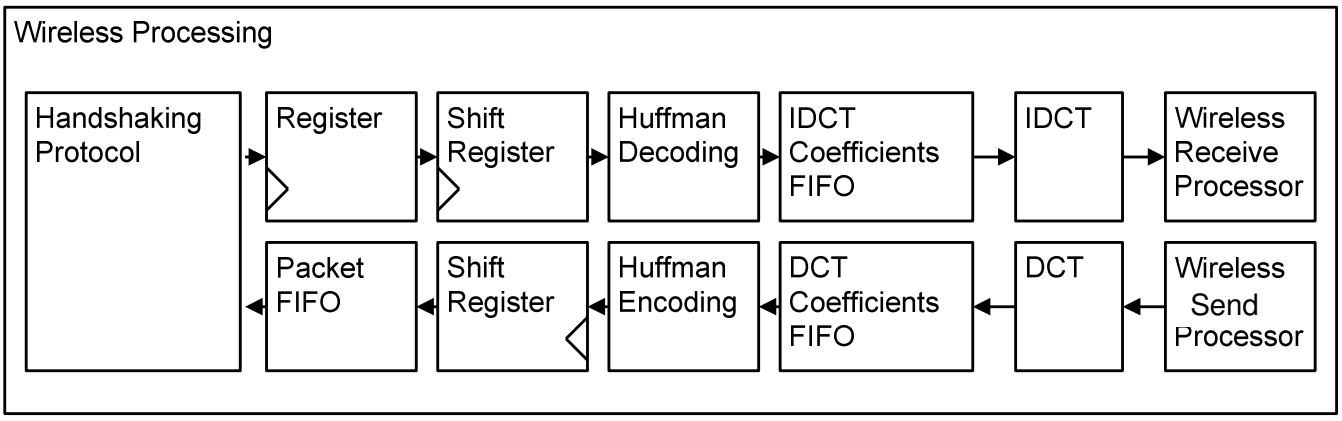
On a packet receive, the Transceiver will shift 37 bytes of data out of the CC2420's RXFIFO. Then the packet is validated by checking its declared length, source and destination addresses, and the CRC bits. If the values are all correct, the EndSession signal will be asserted, indicating that a valid packet has been received and put onto the BigOut bus.

Transmitting data requires a more complicated process. After shifting our packet into the TXFIFO, we have to check whether the CCA signal indicates that the channel is clear. If the channel is not clear, we use a counter with scrambled bits to generate a random amount of time to wait before checking the channel again. This pseudo-random wait time is necessary because it helps to avoid issues where two transceivers wait the same amount of time after seeing a unclear channel and then transmit simultaneously. During this random wait, our Transceiver FSM returns to its wait state. This allows our Transceiver to process any packets we may receive during this random wait time. Also, we use a latch that asserts its output during the random wait; this prevents Ready from being asserted if a packet is waiting to be transmitted. Once CCA indicates a clear channel, we issue the STX_ON to initiate the wireless transmission of our packet.

# Checkpoint 4: Wireless Video Conferencing

## Objective

Unlike previous checkpoints, the specification for this checkpoint provided a high-level description of the final project while leaving its implementation open-ended. A compression scheme consisting of the discrete cosine transform and Huffman encoding was required before sending data, and an analogous decompression datapath was necessary to convert the data back to the YCbYCr format. Also, a wireless protocol was designed to compensate for channel congestion and packet loss.




## Wireless Send Processor

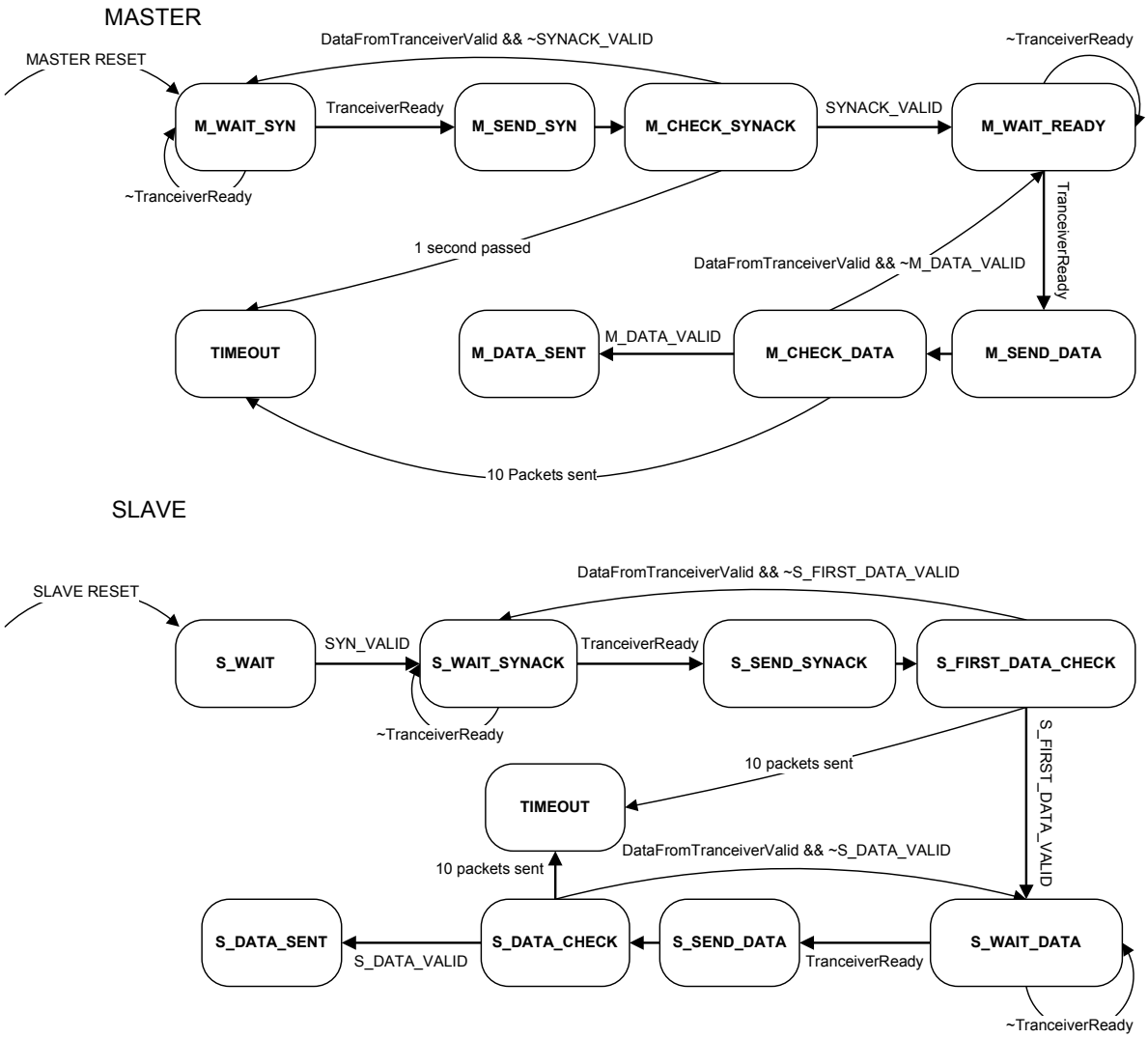
After one subsampled frame of local video has been written to SDRAM, the Wireless Send Processor begins buffering the video data into a 32x32 FIFO. Next, the send processor regulates the input to the DCT blackbox, which discards small high-frequency components for lossy video compression. Once the DCT has received 64 input values, it will output a stream of 16-bit coefficients. We ignore all bits except for bits 11 to 3 and write these 9-bit values to a 9x256 FIFO. This is quantization, another form of compression, which effectively ignores the frequencies that contribute the least to the compressed frame.

## Huffman Encoder

Each coefficient generated by the DCT blackbox is transformed from twos complement to an encoded sequence of bits. Huffman Encoding provides prefix-free code in which no bit sequences are prefixes of any other. This variable length encoding schema allows for more DCT coefficients to be included in each packet. The encoder outputs a single bit at a time; this is used as the input to a 248-bit-wide shift register. When the shift register has shifted in a total of 248 bits, the packet is written to a 248x16 FIFO.

## Wireless Protocol

When transmitting wirelessly, each Transceiver will behave as either Master or Slave. The master initiates a connection by sending a SYN packet. Upon reception of a SYN packet, the slave accepts the connection by sending a SYNACK packet. After the master has received a SYNACK packet, it will begin sending packets of encoded data. The transmission of data packets requires both master and slave to maintain a lossless transfer. The master increments a sequence number each time it receives a valid packet from the slave, and the slave confirms reception by sending its own video data with the sequence number from the last received packet. If the master or slave does not receive a packet within a 50ms time period, it will re-transmit the last packet. This protocol allows for seamless transfer despite congested channels and dropped connections.




```
wire ReceiveAnythingValid = (RecipientAddress == 8'hFF);
wire ReceiveAddrValid = (RecipientAddress == MySrcAddr);
wire SenderAddrValid = (SenderAddress == MyDestAddr);
wire SynPacketValid = (DataFromTranceiverValid && (DataReceived[255:251] == 5'd1));
wire SynackPacketValid = (DataFromTranceiverValid && (DataReceived[255:251] == 5'd2));
wire DataPacketValid = (DataFromTranceiverValid && (DataReceived[255:251] == 5'd3));
wire CountValid = (DataReceived[250:248] == SequenceNumber[2:0]);
wire CountPlusValid = (DataReceived[250:248] == (SequenceNumber[2:0] + 1'b1));

assign SYNACK_VALID= ReceiveAddrValid && SynackPacketValid;
assign M_DATA_VALID = ReceiveAddrValid && SenderAddrValid && CountValid && DataPacketValid;

assign SYN_VALID       = SynPacketValid && ReceiveAnythingValid;
assign S_FIRST_DATA_VALID = ReceiveAddrValid && SenderAddrValid && DataFromTranceiverValid &&          DataPacketValid;
assign S_DATA_VALID = ReceiveAddrValid && SenderAddrValid && DataFromTranceiverValid &&          DataPacketValid &&
CountPlusValid;
```

## Huffman Decoding, IDCT, and Wireless Receive Processor

When a data packet is received, the data is latched in and sent to a shift register. We then shift bits into our Huffman Decoder, which reassembles the coefficients from the encoded bitstream. The output of the decoder is buffered in a 16x256 FIFO. These coefficients are then sent to the IDCT blackbox, which outputs the luma values for the received video. The luma values are put in the YCbYCr format in the Wireless Receive Processor. Once the Wireless Receive Processor receives a full burst of data, it requests a write from the arbiter. To prevent the

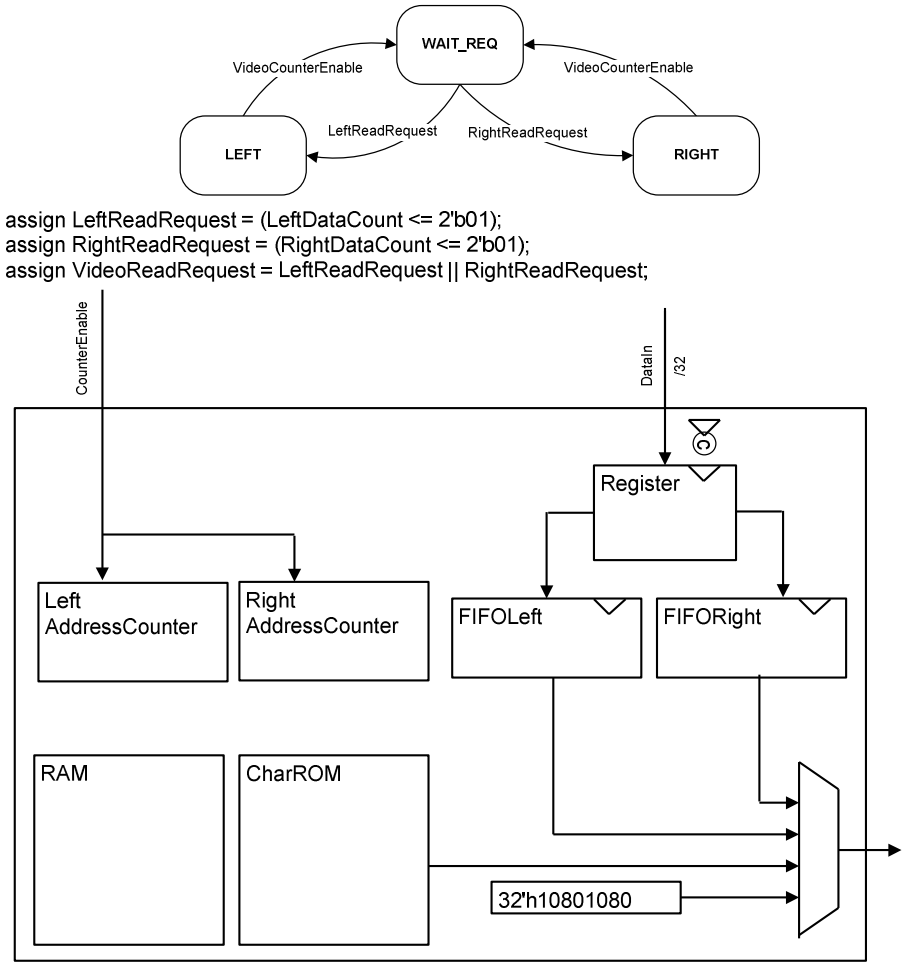
end user from seeing the frame data as it is being updated in memory, the Wireless Receive Processor and the Video Processor alternate which bank is being displayed and which bank is being written to with received video data

## Camera Processor

Because of the limited bandwidth of the 802.15.4 protocol, it was necessary to reduce the resolution of our video to 160x120. This was done by modifying the VDProcessor from checkpoint two. One address counter was modified to track which line and pixel pair was being asserted by the video decoder. Based on this address counter, we determined whether or not to write the YCbYCr data to the FIFO. Another address counter was used to provide the memory address of the next write request. Also, the Camera Processor was modified to only sample a frame of video on a Reset signal or when the Wireless Send Processor requests a new frame of data.

## Video Processor

The VEProcessor of checkpoint 2 was modified to display the local and received video data, along with a simple GUI describing the project's state. Two address counters and two FIFOs were used to separately buffer each frame of data. A simple FSM arbitrates between the two FIFOs' read requests. The GUI is generated by filling an SRAM module with the ASCII characters we want to display. We map 16x16 blocks of the screen to each location in the SRAM. Then the output of the SRAM is used as the input to the provided CharROM module. The offset of the current line and pixel pair determines which bit of the CharROM's output to use to set the color of the pixel pair. Edge detectors are used to signal when the source or destination address, channel, or the Master/Slave switch has been changed. An FSM then updates the corresponding values in SRAM, thus changing what the user sees on the screen.



# Design Metrics

| Logic Utilization | Used | Available | Utilization |
|---|---|---|---|
| Number of Slice Flip Flops | 12,117 | 38,400 | 31% |
| Number of 4 input LUTs | 9,768 | 38,400 | 25% |
| **Logic Distribution** | | | |
| Number of occupied Slices | 8,145 | 19,200 | 42% |
| **Total Number of 4 input LUTs** | 10,904 | 38,400 | 28% |
| Number used as logic | 9,768 | | |
| Number used as a route-thru | 707 | | |
| Number used as Shift registers | 429 | | |

**FPGA Resource Utilization**

| Checkpoint | Design Time | Coding Time | Debugging Time | Total Time |
|---|---|---|---|---|
| 1 | 3 hrs | 2 hrs | 0 hrs | 5 hrs |
| 2 | 6 hrs | 6 hrs | 3 hrs | 15 hrs |
| 3 | 8 hrs | 10 hrs | 7 hrs | 25 hrs |
| 4 | 5 hrs | 18 hrs | 72 hrs | 95 hrs |
| Total | 23 hrs | 36 hrs | 82 hrs | 140 hrs |

**Project Time Allocation**

**Design Analysis**

For checkpoints 1-3, we spent a good balance of time in design, coding, and debugging. In checkpoint 4, however, very little time was spent planning the details, and this resulted in a significantly greater amount of time in coding and debugging. However, the open-ended nature of checkpoint 4 meant that we had very little material to use when doing our design document. Because planning ahead was virtually impossible, it was necessary to design each part of checkpoint 4 as we went along.

# Conclusion

### Checkpoint 1

The memory controller is operating on our relatively slow system clock, 27MHz. While this makes it much easier to meet timing constraints, our memory bandwidth is nowhere near the capabilities of the SDRAM chips, which are designed to operate at speeds of up to 133MHz. However, our memory controller will read eight 32-bit words in 11 cycles, which translates to roughly 75MB/sec. Write speeds are about the same. The use of fixed-length read and write bursts required us to account for that throughout the rest of our project, significantly reducing the amount of abstraction between the SDRAM controller and the rest of the project. Furthermore, any change to the memory clock speed or the burst length would require refactoring a significant portion of the rest of the project.

### Checkpoint 2

Because this checkpoint of the project was implemented without knowledge of future checkpoints, we wrote the arbiter and the VE and VD processors without any future modifications in mind. Checkpoint 4 required us to significantly redesign everything written in this checkpoint. Another difficulty in this checkpoint was tight coupling between the arbiter and the memory controller. Small tweaks to the operation of the memory controller required changes in the arbiter's logic as well.

### Checkpoint 3

The implementation of the Transceiver was far more difficult than the abstract design. The smaller details, such as setting and resetting our flags and counters and using the handshaking protocols correctly in all cases, ended up requiring much more design time than we had planned. One phenomena encountered by many students was the Transceiver failing on transmissions unless the two CRC bytes were not shifted into the end of every packet. This goes against what our documentation specified, but the fix worked for us and a number of other students. Lastly, the wireless transceiver is the bottleneck of the whole system, since the theoretical maximum bandwidth of the 802.15.4 protocol is a mere 250kbps.

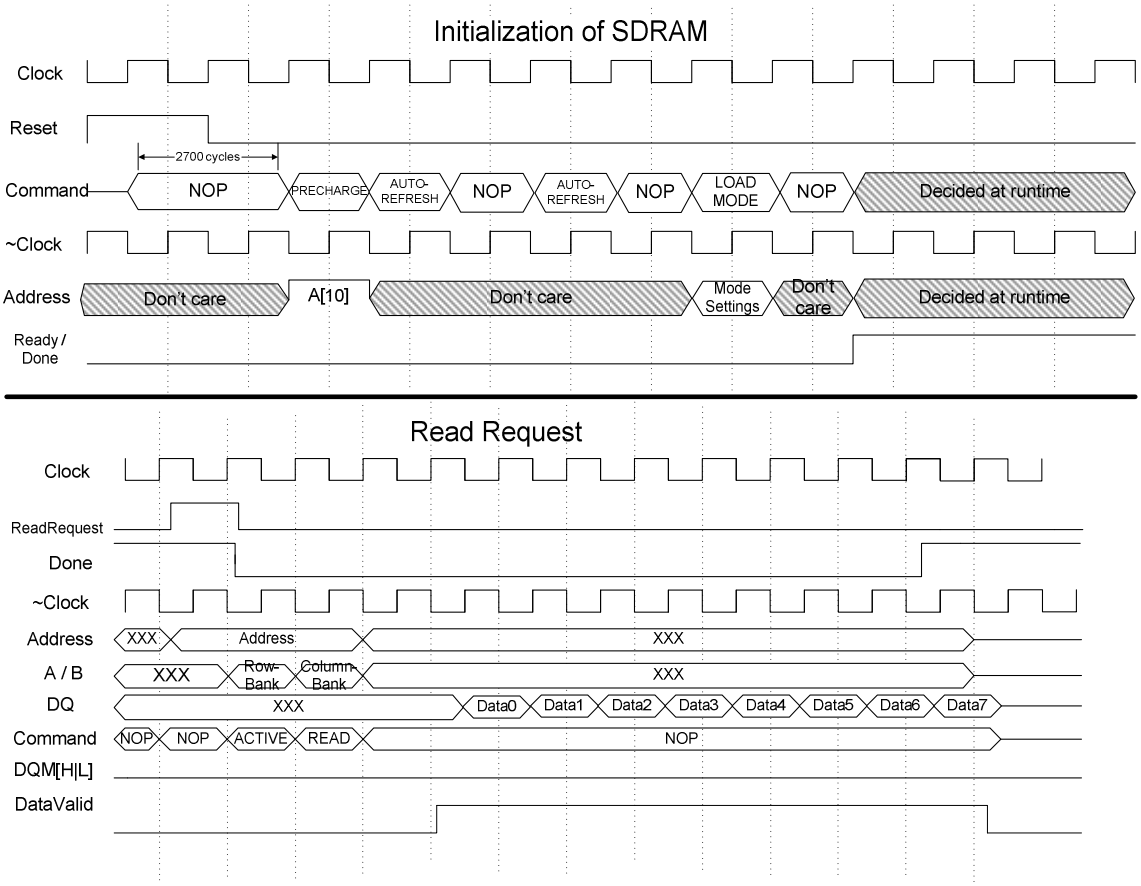
### Checkpoint 4

Completing the last steps of the final project was far more difficult than the rest of the project combined. First, the sheer complexity of tying together all the modules we had previously written required each component to be working perfectly. A single bug in our project would require many hours of debugging because of the time spent simply trying to find where the error came from. The debugging methodology was invented on the fly, though other students shared many of their own debugging tricks. On average, our final project synthesized in fifteen minutes, significantly hindering our ability to make quick fixes. Chipscope often broke our project by introducing too much delay into our design; this inspired some ingenuity in using the seven-segment LEDs to output debugging values.

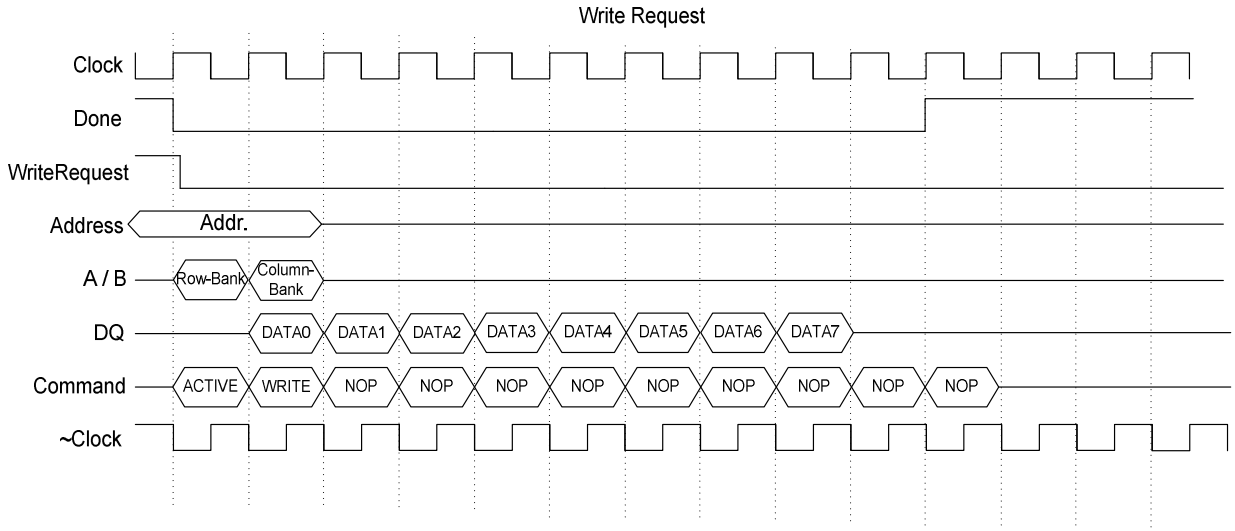
# Suggestions

- Provide a TA solution that works with the specifications, and quickly provide a fixed TA solution as soon as possible when someone points out a flaw.
- Like most students, we felt that this project was weighted far too much towards the end. Our best suggestion would be to replace checkpoint 2 with checkpoint 2.5.
- Ensure better communication between the TAs. The same question with different TAs would result in conflicting answers every once in a while.
- Ban anybody from playing music/watching videos on their laptop through speakers; ban any movies or TV shows on the big screen; ban any video game systems from the lab. We aren't big fans of Linkin Park and listening to that one kid playing it over and over again was rather cruel; nor do we like listening to the same video off YouTube over and over again.
- The TAs better have taken CS150. The TAs this year definitely knew the intricacies of this project and prevented us from running into a lot of issues that plagued previous semesters.
- Please consider potential conflicts between project deadlines and classroom deadlines (problem sets or midterms). We escaped a checkpoint being due the day after a midterm, that could have been ugly.
- More lectures centered on topics that are relevant to the project. We had one lecture on RAM and one lecture on how wireless transceivers work.
- Bring a couch in the lab so that we can get a bit of sleep while working long hours in the lab.
- Include design portions of the project that incorporate something other than finite state machines. That seems to be all we did for this entire project.
- Reading case studies or having lectures about how hardware works in the real world would definitely provide insight into the applicability of what we do in lab with what happens in industry. Pister mentioned some of the real-world design processes in lecture, but he didn't lecture too often.

# Appendix




**SDRAM Initialization and Read Timing Diagram**




**SDRAM Write Timing Diagram**

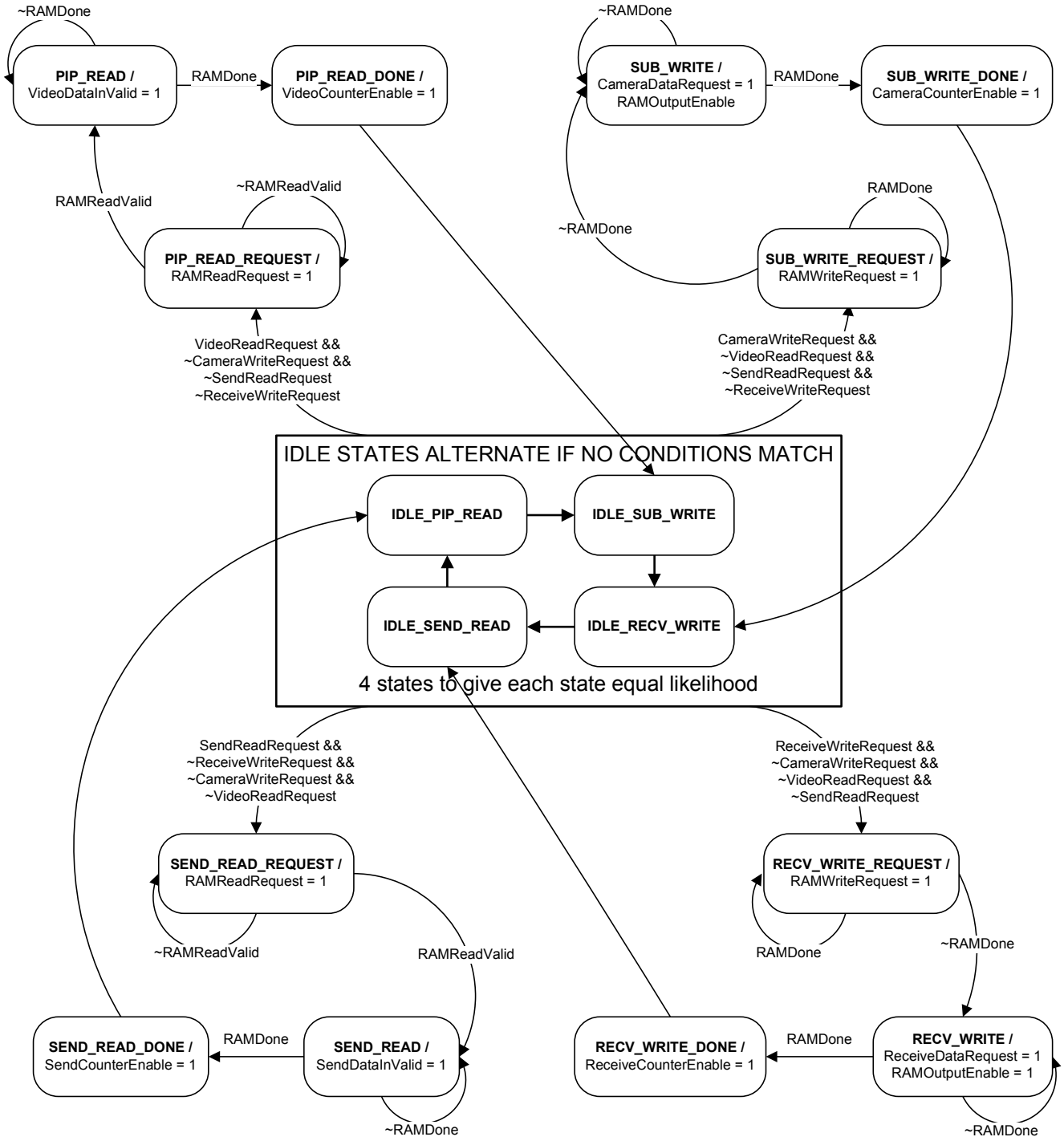
| STATE | ACTION/COMMAND | DataValid | DONE | READY | RAM_CS | RAM_RAS | RAM_CAS | RAM_WE | RAM_A | RAM_DQ |
|---|---|---|---|---|---|---|---|---|---|---|
| BEGIN | | 0 | 0 | 0 | 1 | X | X | X | XXX | XXX |
| 1st IDLE | PRECHARGE | 0 | 0 | 0 | 0 | 0 | 1 | 0 | RAM_A[10] = 1 | XXX |
| 2nd IDLE | AUTO REFRESH | 0 | 0 | 0 | 0 | 0 | 0 | 1 | XXX | XXX |
| 3rd IDLE | NO OP | 0 | 0 | 0 | 0 | 1 | 1 | 1 | XXX | XXX |
| 4th IDLE | AUTO REFRESH | 0 | 0 | 0 | 0 | 0 | 0 | 1 | XXX | XXX |
| 5th IDLE | NO OP | 0 | 0 | 0 | 0 | 1 | 1 | 1 | XXX | XXX |
| 6th IDLE | LOAD MODE REGISTER | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0000000100011 | XXX |
| 7th IDLE | NO OP | 0 | 0 | 0 | 0 | 1 | 1 | 1 | XXX | XXX |
| Ready | NO OP/(part of previous cycle) | 0 | 1 | 1 | 0 | 1 | 1 | 1 | XXX | XXX |
| IDLE | NO OP | 0 | 0 | 1 | 0 | 1 | 1 | 1 | XXX | XXX |
| ACTIVE | ROW ACTIVE | 0 | 0 | 1 | 0 | 0 | 1 | 1 | Address[23:11] | XXX |
| READ | READ | 0 | 0 | 1 | 0 | 1 | 0 | 1 | Address[8:0] RAM_A[10] = 1 | XXX |
| CAS - IDLE | NO OP | 0 | 0 | 1 | 0 | 1 | 1 | 1 | XXX | XXX |
| Data 0 - IDLE | NO OP | 1 | 0 | 1 | 0 | 1 | 1 | 1 | XXX | Data0 |
| Data 1 - IDLE | NO OP | 1 | 0 | 1 | 0 | 1 | 1 | 1 | XXX | Data1 |
| ... | NO OP | 1 | 0 | 1 | 0 | 1 | 1 | 1 | XXX | Data2 |
| Data 7 - IDLE | NO OP | 1 | 1 | 1 | 0 | 1 | 1 | 1 | XXX | Data7 |
| IDLE | NO OP | 0 | 1 | 1 | 0 | 1 | 1 | 1 | XXX | XXX |
| IDLE | NO OP | 0 | 1 | 1 | 0 | 1 | 1 | 1 | XXX | XXX |
| IDLE | NO OP | 0 | 0 | 1 | 0 | 1 | 1 | 1 | XXX | XXX |
| ACTIVE | ROW ACTIVE | 0 | 0 | 1 | 0 | 0 | 1 | 1 | Address[23:11] | XXX |
| WRITE | WRITE | 0 | 0 | 1 | 0 | 1 | 0 | 0 | Address[8:0] | XXX |
| CAS - IDLE | NO OP | 0 | 0 | 1 | 0 | 1 | 1 | 1 | XXX | XXX |
| Data 0 - IDLE | NO OP | 1 | 0 | 1 | 0 | 1 | 1 | 1 | XXX | Data0 |
| Data 1 - IDLE | NO OP | 1 | 0 | 1 | 0 | 1 | 1 | 1 | XXX | Data1 |
| ... | NO OP | 1 | 0 | 1 | 0 | 1 | 1 | 1 | XXX | Data2 |
| Data 7 - IDLE | NO OP | 1 | 1 | 1 | 0 | 1 | 1 | 1 | XXX | Data7 |
| IDLE | NO OP | 0 | 1 | 1 | 0 | 1 | 1 | 1 | XXX | XXX |
| IDLE | NO OP | 0 | 1 | 1 | 0 | 1 | 1 | 1 | XXX | XXX |

**SDRAM Control Output Chart**

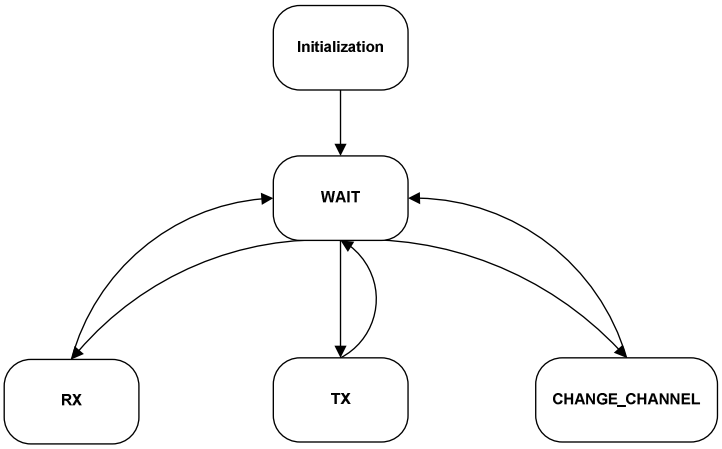
Outputs

SDRAM Arbiter

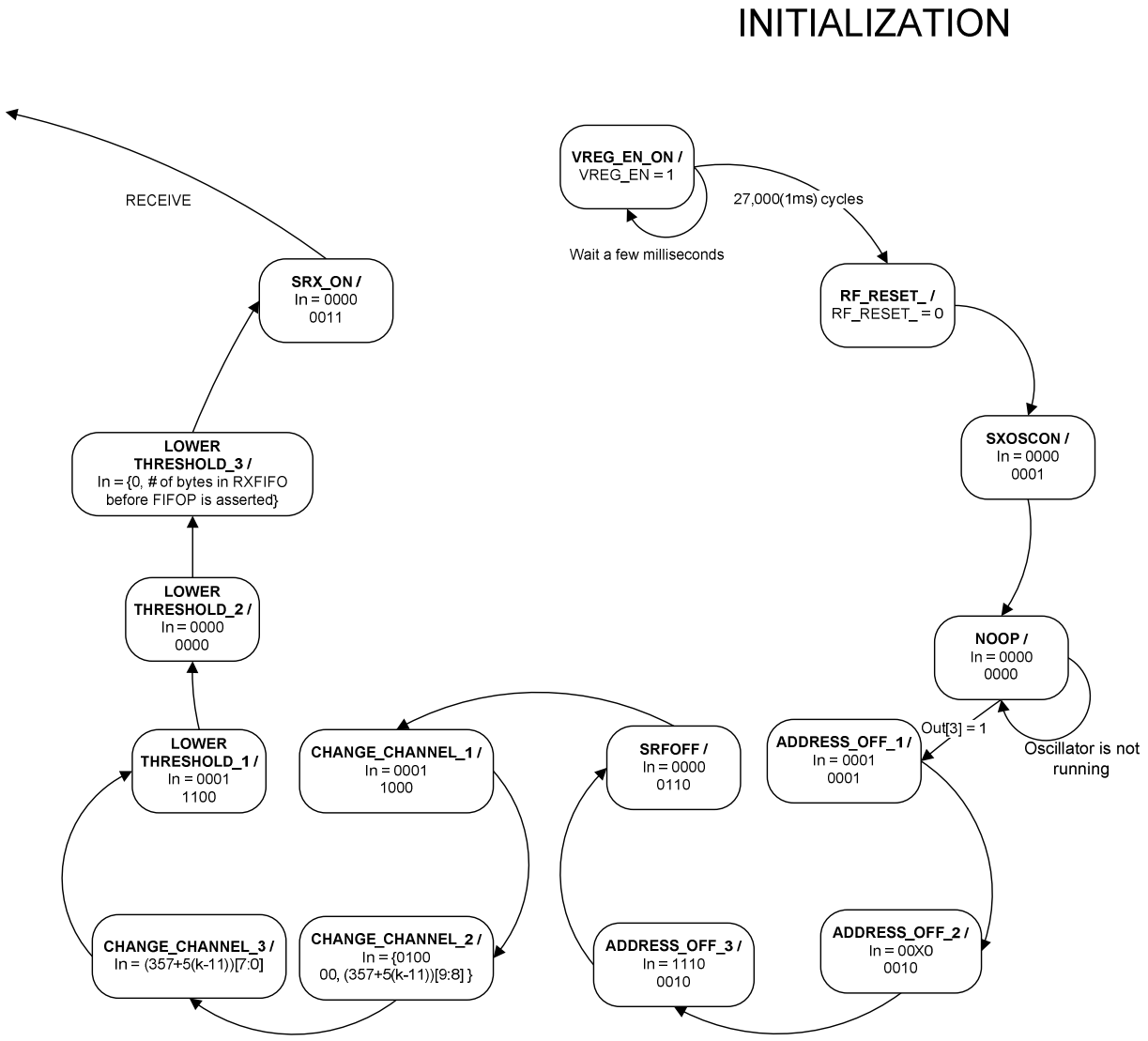
```
assign VideoDataOut = RAM_DQ;
assign SendDataIn = RAM_DQ;
always @( * ) begin
        if ((CS == RECV_WRITE_REQUEST) || (CS == RECV_WRITE)) WriteData = ReceiveDataOut;
        else if ((CS == SUB_WRITE_REQUEST) || (CS == SUB_WRITE)) WriteData = CameraDataIn;
        else WriteData = 32'hXXXXXXXX;
end
```



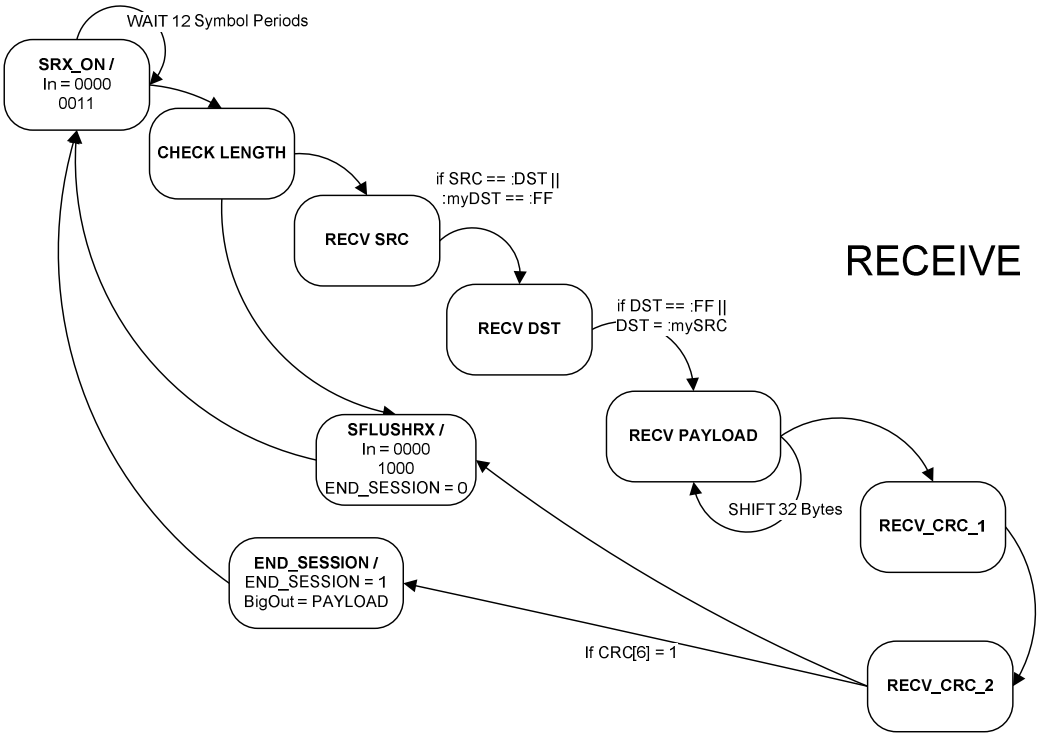

**SDRAM Arbiter State Diagram**



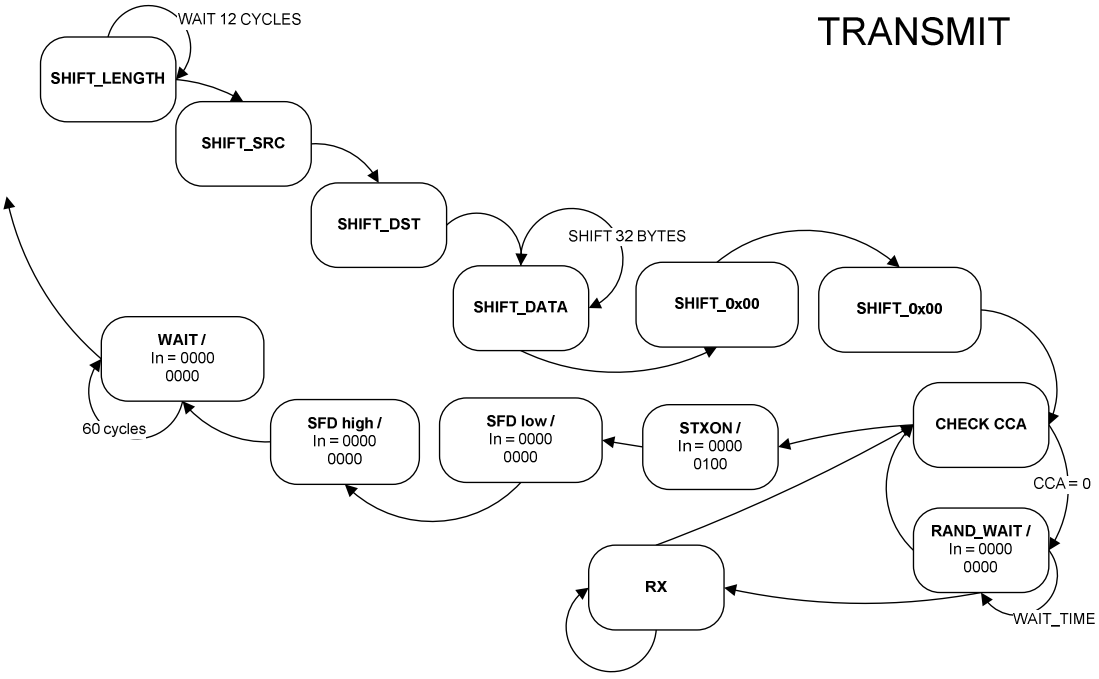

**Transceiver FSM Overview**




**Transceiver Initialization State Diagram**

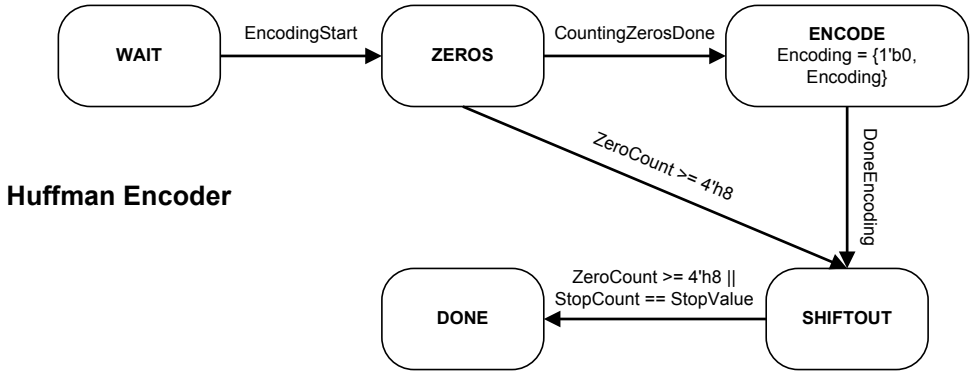



**Transceiver Receive State Diagram**




**Transceiver Transmit State Diagram**

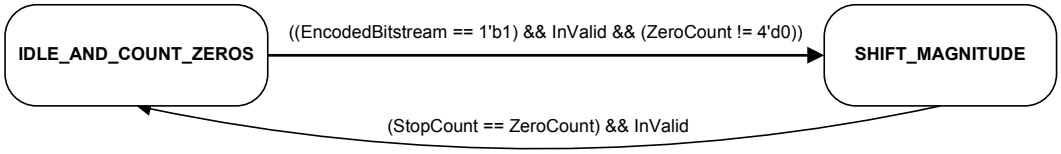
```
assign Encoding = (LumaValue[8]) ? {((LumaValue[7:0] ^ 8'hFF) + 1'b1), 1'b1} : {LumaValue[7:0], 1'b0}
assign StopValue = (16 – 2*ZeroCount);
assign HuffmanReady = (CS == WAIT);
assign Done = (CS == DONE);
```




**Huffman Encoder**

```
assign Done = ((CurrentState == SHIFT_MAG) && (StopCount == ZeroCount) || Zero) && InValid;
assign Zero = (~ZeroCountDone && InValid && (EncodedBitstream == 1'b1)) && (CurrentState !=
SHIFT_MAG) && (ZeroCount == 4'b0000);
assign DCTCoefficients = (Zero) ? 16'h0000 : TwosComplement;
assign TwosComplement = (EncodedBitstream && InValid) ? ((Magnitude ^ 16'hFFFF) + 1) : Magnitude;
```

**Huffman Decoder**




**Huffman Encoder and Decoder State Diagrams**