Wireless Video Conferencing System

with video compression and instant messaging

Chris Fletcher, Ilia Lebedev
University of California, Berkeley

May 2, 2008

Contents

1 Design Overview **2**

2 System Description **4**
2.1 SDRAM Interface . . . 4
2.1.1 SDRAM Controller . . . 4
2.1.2 SDRAM Arbiter . . . 5
2.2 Graphics System . . . 6
2.2.1 Video Decoder Processor . . . 6
2.2.2 Video Encoder Processor . . . 9
2.2.3 User Interface . . . 10
2.2.4 Graphics Transport Engine . . . 10
2.3 Communications System . . . 14
2.3.1 RF Transceiver Interface . . . 14
2.3.2 Network Controller . . . 14
2.3.3 Packets: Video and Instant Messaging. . . . 16

3 Design Metrics **17**

4 Conclusion **17**

5 Suggestions **18**

6 Acknowledgments and References **19**

Abstract

The Wireless Video Conferencing System is a term project for the Spring 2008 CS150 at the University of California, Berkeley. The system is an implementation of a visual communications device on a Xilinx ViretexE platform, the CaLinxII. The design interfaces with a 802.15.4 radio, Micron SDRAM, as well as a ITU656/601 video encoder and decoder. The system consists of a multi-port SDRAM interface, Video Encoder and Decoder interfaces, video compression and decompression and decompression engines, and the wireless radio interface. The required functionality allowed for a very slow (on the order of 1 FPS) compressed video-only grayscale telecommunication between two peers with unique addresses on any of the 16 802.15.4 radio channels, with some on-screen statistics displayed as text. The design does not include support for more than two peers per channel (due to bandwidth limitations), audio, or color. The required functionality has been extended somewhat for extra credit: instant messaging with reliable delivery, an N64 controller interface, an on-screen keyboard, and full-screen local video features were added to the system. PS/2 keyboard and video over ethernet features were designed, but not implemented due to time constraints and availability of FPGA fabric. Through careful design, testing, and good division of labor, we have a telecommunications system with an impressive set of features and a refined user interface.

1 Design Overview

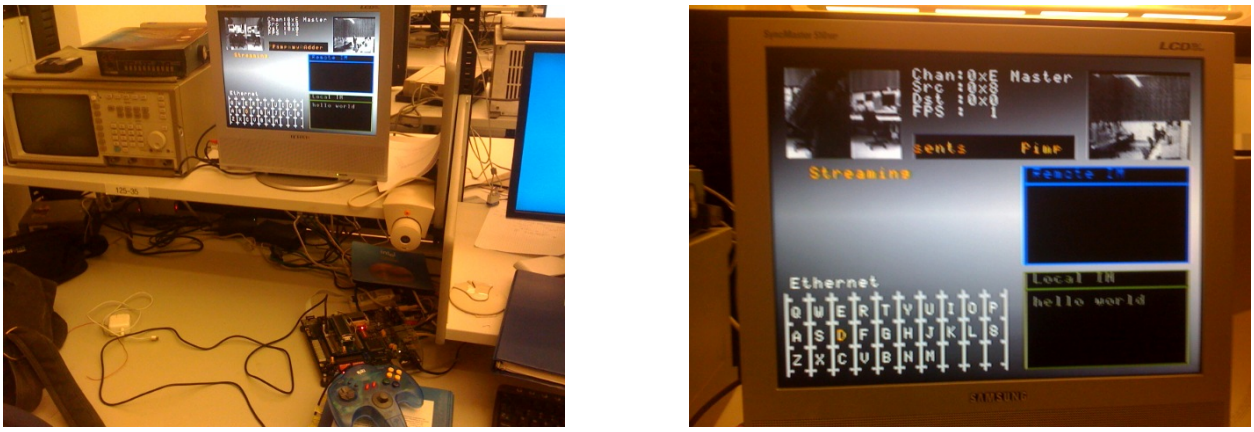


Figure 1: GUI and workstation.

This project was designed in five discrete milestones:

1. SDRAM Controller: allows for reads and writes to a virtually limitless supply of off-board memory
2. SDRAM Arbiter & Local Video: creates a means for SDRAM to arbitrate between multiple clients and establishes the video pipeline through the Video Decoder and Encoder
3. Sub-sampled Video: establishes a compressed video scheme to send video data over a relatively slow wireless network

4. Wireless Transceiver: enables data (video and text) to be sent over the wireless network
5. Graphics, Compression, and Communications Engines: interfaces sub-sampled video with the wireless transceiver to allow for wireless video conferencing

The whole of this system is shown in figure 2.

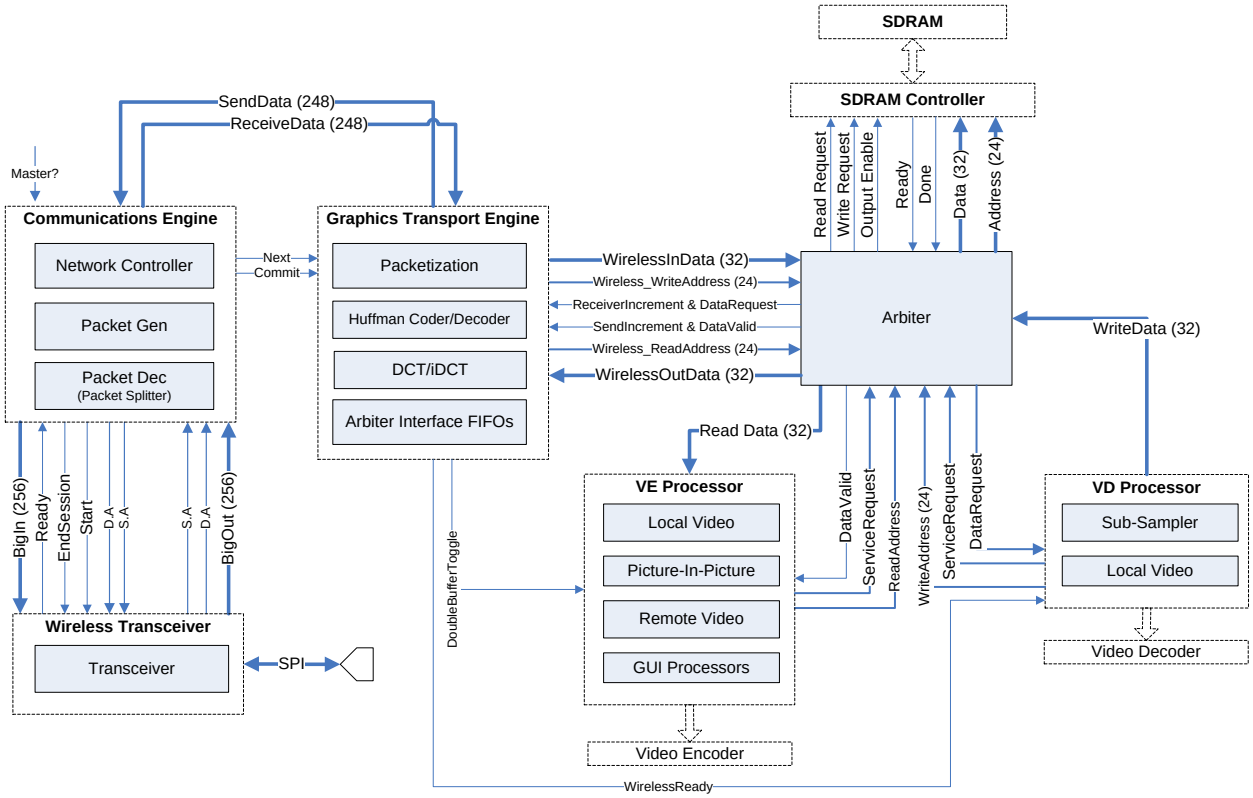


Figure 2: Top level block diagram.

Additionally, the following functionally was added alongside the Graphics Systems:

1. On-screen Keyboard
2. N64 Interface
3. Scrolling text
4. Local video mode (with toggle to wireless conferencing mode)
5. Instant messaging with reliable delivery

The project GUI and workstation is shown in figure 1.

2 System Description

2.1 SDRAM Interface

The main reason an SDRAM interface is present in this project is because it was necessary for the early stages of the Graphics engine. In its current form, however, the SDRAM is only used for a local video feature implemented in addition to the required feature set. Local video demands a fair amount of buffering since the video encoder and decoder are not guaranteed to be in sync. The Xilinx VirtexE 2000 device does not present nearly enough on-chip ram to buffer video, thus an external memory interface is required. Given the fact that video undergoes sub-sampling and compression before it is stored in memory, however, and given the relatively small resulting frame size, the entire required feature set could be implemented without an SDRAM controller. Nevertheless, this interface is important for educational reasons, as well as to facilitate any expansion of the Wireless Video Conferencing System feature set. The interface provides two read ports and two wirie ports, utilized by the Video Encoder and Decoder, and by the Graphics System.

2.1.1 SDRAM Controller

The SDRAM interface presents a usable, FIFO-like interface to the Wireless Video Conferencing System. The goal here is to encapsulate initialization and chip-specific details of SDRAM, exposing a trivial address-data-like interface. All SRDAM commands are issued from within the module, while the system is communicates to SDRAM through a simple ready-start handshake. Since our Wireless Video Conferencing System runs on a relatively slow (27MHz) clock, most of the timing constraints imposed by the Micron SDRAM interface do not ever come into play. Since our system only requires the most basic read/write functionality, all the SDRAM controller must do is preform stateful initialization of the Micron SDRAM, and issue read and write commands. More advanced features of the Micron SDRAM such as data masking are not utilized. Our system takes advantage of 8-word bursts offered by the Micron SDRAM. Since data is refreshed by our system often, we do not concern ourselves with refreshing the data.

During initialization, the SDRAM controller follows the exact procedure outlined in the Micron datasheet. We allow just over 0.1ms after a system reset, pre-charge, auto-refresh twice, load the mode register value, and finally assume an idle state, waiting for a read or a write command. The controller favors write requests over read requests.

The SDRAM controller preforms a write with auto precharge upon receiving a read request. The controller follows the procedure outlined in the Micron SDRAM data sheet: issue "Active," followed by the write command, specifying the first chunk of data. NOP is held for eight more cycles in order to transport the remaining data, and to accommodate for the required NOP. The controller then reenters the idle state, ready for another request.

The SDRAM controller preforms a read with auto precharge upon receiving a read request (If no write request is issued). The controller follows the procedure outlined in the Micron SDRAM data sheet: issue "Active," followed by the read command. The next cycle is a NOP to accommodate for the CAS delay. After that, the controller spends eight cycles issuing NOP while the requested data is read out. The controller then reenters the idle state, ready for another request.

2.1.2 SDRAM Arbiter

The SDRAM Arbiter's job is to provide an interface between SDRAM and the rest of the wireless video conferencing system. Specifically, the Arbiter must arbitrate between different SDRAM read and write requests, given that some requests come from modules requiring more immediate attention than others. Additionally, the Arbiter handles SDRAM read/write timing requirements and thus simplifies read/write operations from the prospective of its client modules.

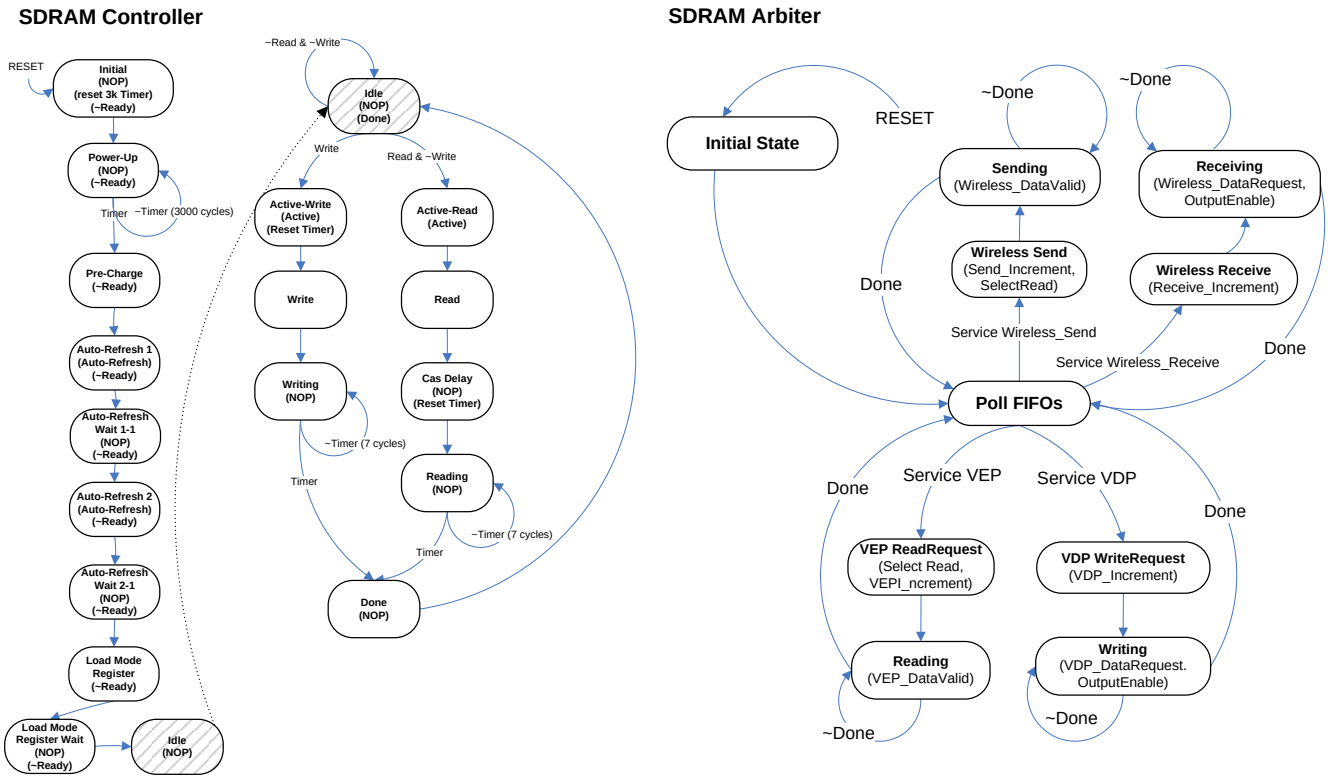


Figure 3: The SDRAM Controller and Arbiter.

The Arbiter interprets different requests as 1-bit signals from client modules. These requests originate from one of: the Video Decoder Processor (writes), Video Encoder Processor (reads), and the Graphics Transport Engine (reads and writes). Once the Arbiter decides which client to service, it performs one 8-word burst worth of work for that client, returns to idle mode, and polls its clients for the next request. If a client contains multiple sub-modules that need service (the PIP and remote video FIFOs in the Video Encoder Processor, for example), it is the client's job to decide which sub-module gets serviced once the Arbiter decides to service that client.

Given simultaneous requests from multiple client modules, the Arbiter provides service based on a strict priority scheme given in table 1. The two local video processors receive the highest priority because they

have to be serviced most often and change state faster than the FIFOs in the Graphics Transport Engine (given that wireless operates slower than the TV or Camera). Wireless Receive is given higher priority than Wireless Send because if incoming packets overflow their buffer, the wireless conferencing link is lost. By contrast, if wireless data is not sent with as great urgency, the greatest loss will be speed, rather than stability.

Module	**Priority**
Video Encoder Processor	1
Video Decoder Processor	2
Wireless Receive	3
Wireless Send	4

Table 1: Arbitration priority scheme.

2.2 Graphics System

The graphics system is responsible for managing all graphics data in the design. Graphics interfaces with the camera (Video Decoder Processor, VDP), TV (Video Encoder Processor, VEP), and wireless transceiver (Graphics Transport Engine, GTE). Additionally, graphics is responsible for generating the user interface, which is presented to the video encoder, independently of what is being read from the camera. These systems, not including the GTE, comprise the local graphics sub-system (see figure 4). The GTE interfaces with the wireless send/receive protocol system and the local graphics sub-system to enable wireless video conferencing.

The basic unit of data in the Graphics System is the 32-bit pixel pair. The pixel pair is interpreted by the system as a Y, Cr, Y, Cb value, as defined by the ITU 656 and ITU 601 video standards, where Y indicates luminance, and Cr/Cb indicates red chrominance and blue chrominance, respectively. Luminance corresponds to pixel brightness and chrominance corresponds to different components of color.

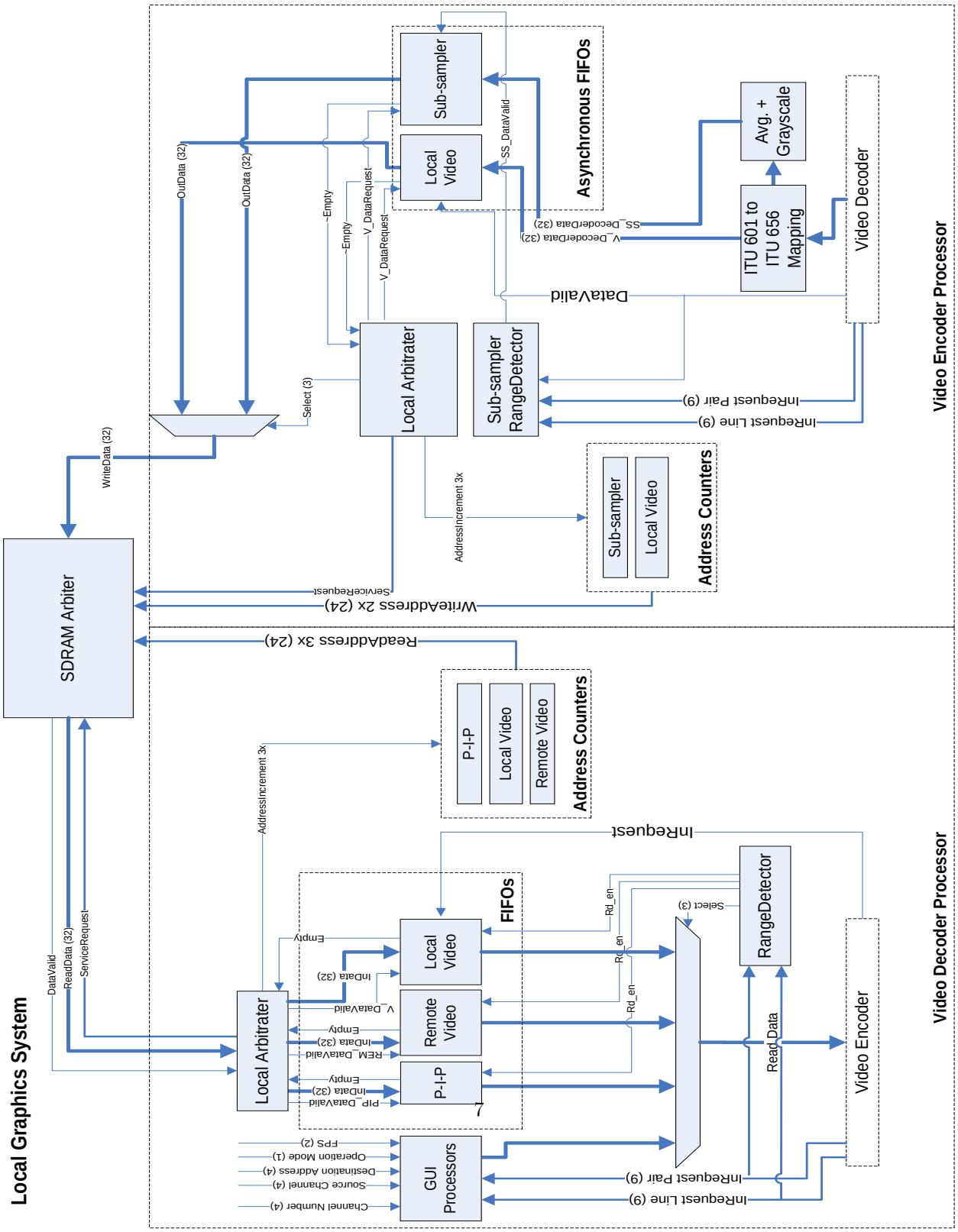
The Graphics System can interpret local uncompressed video as well as Sub-sampled video. Local uncompressed video is made up of 32-bit ITU 601/656 pixel pairs. Sub-sampled video, while reformed into 32-bit pixel pairs, is transported as 8-bit luma values (given by the average of the two luma values that make up the pixel pair). The Sub-sampled chroma is a constant for all Sub-sampled pairs, and is thus not transported at all. The uncompressed and Sub-sampled pixel pairs make up the fundamental data units in the graphics system.

Given a fundamental unit of transport, the remainder of this section will discuss the Graphics System's major components in greater detail. As it is key to transporting video data from local SDRAM to the wireless system, video compression will also be discussed. Finally, the GUI system, specifically the text generation modules, are included at the section's conclusion.

2.2.1 Video Decoder Processor

The Video Decoder Processor (VDP) is responsible for buffering data from the video decoder (camera interface) and storing it in SDRAM for when it is needed by the VEP. Specifically, the VDP must sample

Figure 4: The Local Graphics System (w/o the Graphics Transport Engine)



uncompressed local video data and sub-sampled grayscale video data. The uncompressed video data is used to display uncompressed local video (unaltered video) and the sub-sampled data what is actually sent over wireless and displayed remotely.

Data Buffering Since the camera and video decoder constantly output video data, regardless of the state of the SDRAM arbiter, all output video data must be buffered before it is written to SDRAM. All VDP buffering was accomplished by 32-wide/32-deep asynchronous FIFOs with a 5-bit data count interface. The FIFOs are asynchronous because the video decoder runs on a different clock than the main system[1]. Because the main system cannot halt the video decoder, the write enable on these FIFOs is not based on whether or not they can accept more data. As such, it is the arbiter's responsibility to ensure that the FIFOs do not overflow.

Service Requests The FIFOs request service from the Arbiter whenever they are not empty. In other words, the VDP requests service with the OR of each FIFO's empty signal. Because the asynchronous FIFOs output a 5-bit data count, and the main system defines empty as 2'b00 (indicating that there are between 0 and 1 bursts of data left in the 32-word-deep FIFO), data count mapping combinational logic is used to translate the 5-bit data count to a 2-bit data-count before it is interpreted by the service request logic.

The Sub-sampler The Sub-sampler's (SS) job is to sample a compressed frame of video data and store it to a separate bank in SDRAM, relative to uncompressed local video. It is the sub-sampled data that will be sent out over wireless[2]. This is accomplished by only keeping every fourth pair of every fourth line coming out of the video decoder. All other pairs are discarded before they enter the SS FIFO. A dedicated RangeDetector module decides whether or not to keep pixel pairs coming into the SS FIFO by ANDing the `DataValidFromDecoder` line (which is the VDP dedicated write enable line) with a `SampleNow` flag that is only asserted when current pair out of the decoder satisfies these conditions:

1. The line and pair are both divisible by 4
2. The line is in the range of lines 1-240 or 255-494
3. The pair is in the range of pairs 1-320

The VDP skips lines 241-255 because the `InRequestLine/Pair` are determined by an artificial counter that doesn't take interleaving into account[3]. This implies that a sub-sampled frame is exactly 80 pairs and 120 lines in size.

Further compression: Grayscale In addition to the Sub-sampler only keeping certain lines and pairs, all sub-sampled pairs are actually the average of the original pixel pair's luma values, appended to a fixed chroma value of 8'h80. The chroma value maps the pixel pair to grayscale while averaging the luma reduces the sharpness of the video, which is tolerable in a smaller SS image.

[1] Both clocks run at 27 Mhz, however, are potentially out of phase with each other.
[2] A sub-sampled frame of video is on the order of 75 kilobits whereas an uncompressed frame is 5700 kilobits.
[3] In other words, lines 1-240 correspond to every even line in a frame and lines 255-494 correspond to every odd line.

Address Counters As both the uncompressed local video FIFO and the SS FIFO must be serviced by the SDRAM Arbiter independently, and they both write to different banks, each must have its own address counter. The address counter for uncompressed local video acts on RAM bank 0 and counts 0-359 pairs (inclusive), increases the line by 1, starts over, and continues until line 506 (inclusive). When the address counter reaches line 506 and pair 359, it resets to line 0 and pair 0. This writes an interleaved frame of video to SDRAM bank 0. The fact that the frame is stored as interleaved data does not change the operation of the VEP because the video encoder expects data in interleaved order (thus the address counters for the uncompressed local video FIFOs on the VDP and VEP side are identical). The Sub-sampler address counter writes to bank 1 and counts pairs 0-79 (inclusive) and lines 0-119 (inclusive) using the same scheme.

SDRAM Service Decisions Once the SDRAM decides to service the VDP, it will perform an 8-word burst write operation on the local video or SS FIFO. Which FIFO gets serviced is determined by the VDP Local Arbitration system. Priority is given first to local video, as its buffer fills faster, and then to SS. If the SS is full, however, the VDP will grant service to the SS FIFO. After a decision is made, dedicated muxes transfer the `DataOut` and `rd_en` signals to and from the correct FIFO, and increment one of the two address counters.

ITU 601 to ITU 656 mapping The data written to RAM undergoes additional combinational processing so that it can be correctly interpreted by the VEP. This process occurs regardless of which FIFO is being serviced and is a mapping where $\{Y, Cr, Y, Cb\} \longrightarrow \{Cb, Y, Cr, Y\}$.

2.2.2 Video Encoder Processor

The Video Encoder Processor's (VEP) job is to collect pixel pair data from various sources and present the correct pair to the video encoder when required. Data sources include video data from SDRAM (namely uncompressed local video, sub-sampled video (known as Picture-In-Picture, or PIP, to the VEP), and remote sub-sampled data) as well as GUI-generated pixel pairs from the various GUI processors.

The VEP data flow is the inverse of the VDP data flow. When data is read from SDRAM, it is buffered in FIFOs[4] before being presented to the video encoder. This is because 8 words are read from memory in a single read and the video encoder, like the decoder, only processes 1 pixel pair every 4 clock cycles. Address counters are used to keep track of where each FIFO is currently reading from SDRAM, and the Arbiter's service decisions dictate when each address counter increments. There are 3 FIFOs to arbitrate between; namely the uncompressed local video FIFO, the FIFO that receives the data written by the Sub-sampler (PIP), and the FIFO that receives the wireless sub-sampled data. These FIFOs send service requests to the Arbiter whenever **at least one** of them is empty. If service is granted to the VEP, the data is given to the uncompressed local video FIFO first, and then the Sub-sampler FIFOs (where priority between PIP and the remote video FIFO is arbitrary).

Similar to the data mux scheme in the VDP, the VEP muxes the data out of each of the FIFOs and the GUI-processor to determine which pixel pair is actually received by the video encoder. Since the video encoder outputs a requested pair and line, which FIFO/GUI-processor actually sends off its data is determined by where the video encoder is on the screen, giving the viewer a discernible image. A dedicated RangeDetector thus translates a given requested line/pair into a particular select to be interpreted by the data mux.

[4]VEP FIFOs are synchronous because the video encoder operates on the same clock as the main design.

2.2.3 User Interface

The User Interface is is the product of the dedicated GUI Processors in the VEP. The data output from each of these GUI Processors is routed through the VEP data mux, given that the RangeDetector built into the VEP is servicing the GUI as opposed to a video window.

GUI Processors A different GUI processor is responsible for every separate rectangular partition of the screen where a widget (text, a keyboard, IM chat-box, etc) is needed. All GUI processors are given a requested pair and line (from the video encoder) and are committed to output a valid pixel pair, given the request, at the next rising edge. If the requested line/pair is outside the range in which a GUI processor is meant to function, its output is potentially garbage, but is irrelevant because the data mux will not select it. Dedicated GUI Processors include all of the text generation modules, the on-screen keyboard, the IM chat-boxes, and the GUI data display panel (showing channel, source/destination address, and mode).

Local Video Mode Users can choose whether to display uncompressed local video to the entire screen or the wireless video conferencing system (with the rest of the GUI). When the user toggles between these two modes, a dedicated state machine will flush all video FIFOs, reset the corresponding address counters, and reset both the video encoder and decoder. Once the reset sequence is complete, only those FIFOs that are needed for the desired mode (the uncompressed local video FIFOs versus the SS/PIP FIFOs) are enabled.

Text Display Processors A large number of the GUI Processors are dedicated text generation processors. These processors, given an `InRequest` line and pair, will determine where that pixel is, relative to their defined range[5], and output a character pixel (one of two colors, depending on the text and its background). This process is split into two steps. First, the pixel pair is mapped to 8×16 pair/line boxes in the region (defining which character region the pixel pair falls in). Second, the pixel pair is mapped to one of the 8×16 pixels within the given 8×16 block. Given the exact pixel in the region, the text processor will read out a desired character ASCII code from a static LUT and pass that ASCII through another LUT that will produce a 64-bit bitmap which is used to determine whether the character itself or the character background color should be sent to the output of the text processor.

2.2.4 Graphics Transport Engine

The Graphics Transport Engine (GTE, see figure 5) is a two-way pipeline that forms the interface between SDRAM and the Communications Engine. Only sub-sampled video data is processed by the GTE, due to its initial compression factor. This data is further compressed by a Discrete Fourier Transform (DCT), translated into a Huffman bit stream, and packaged into wireless payload-size 248-bit packets for transport to a remote implementation of the design. On the receiver's side, the packets are fed through an inverse Huffman processor, converting the bit stream into valid DCT output values, which are then presented to an inverse DCT (iDCT), at which point they are translated back into valid sub-sampled video pixel pairs.

Arbiter-side FIFOs Interfacing directly with the Arbiter are two synchronous FIFOS that are 8-bits wide and 128 deep. Each is wide enough to store one sub-sampled pixel pair (given the VDP's grayscale and luma averaging scheme). They must be 128 deep to accommodate the DCT units, which take 64 pixel

[5] A GUI Processor's range is the rectangular region in which is designed to operate.

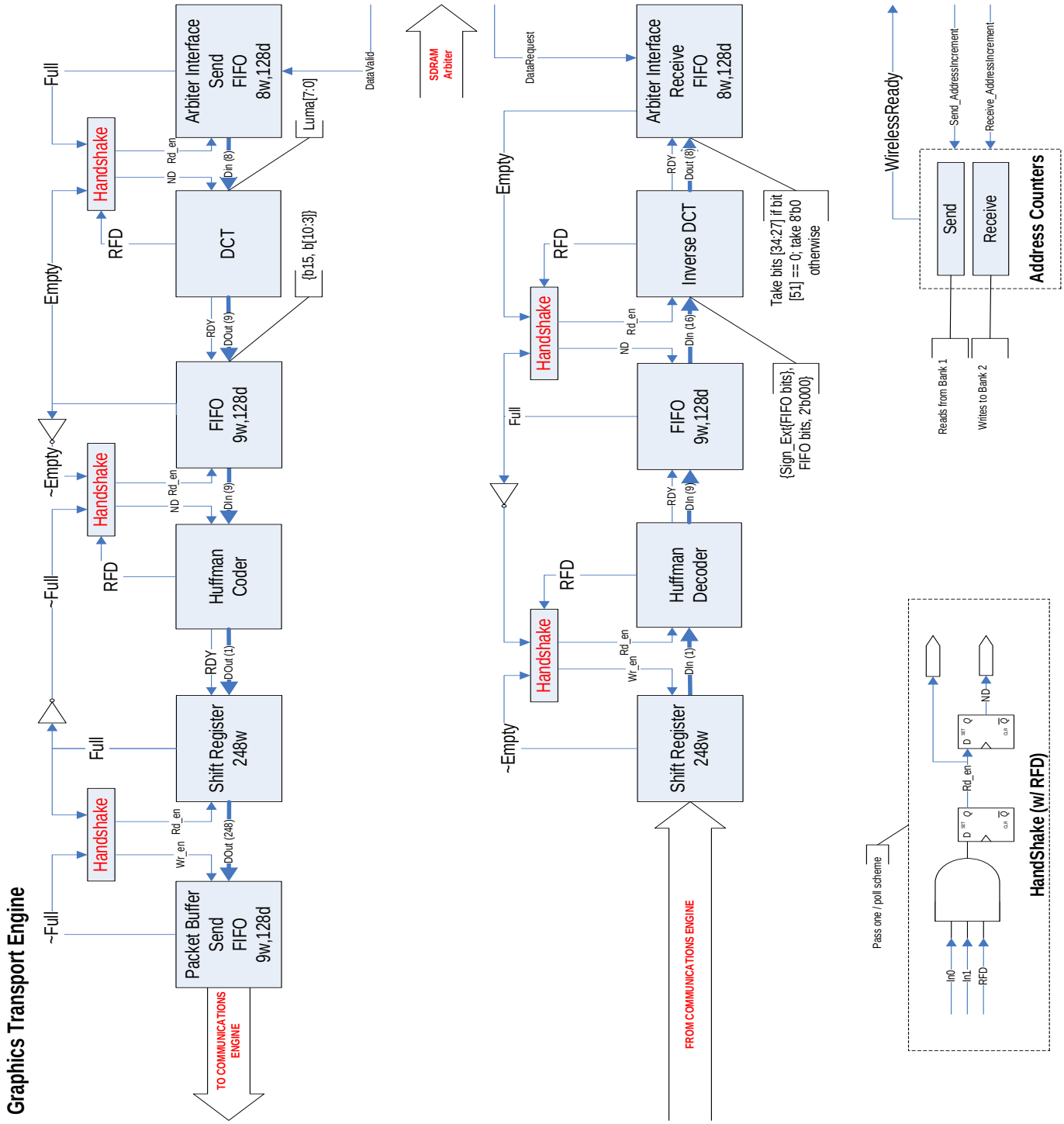


Figure 5: The Graphics Transport Engine.

pairs (in/out for DCT/iDCT, respectively). Address counting is done by column instead of by row. In other words, the line address increases from 0-119 for each address increment, and then resets to 0, at which point the column increases by 8 (one SDRAM burst) until it reaches 72 (inclusive). This presents the DCT with 150 8x8 pixel pair blocks as opposed to interleaved lines, where each 150-block set corresponds to exactly one sub-sampled frame.

DCT / iDCT The DCT will convert an 8x8 block in the spatial domain to values in the frequency domain. When 64 pixel pairs are received by the DCT, it will process the block and place 16-bit frequency values into a buffer that is emptied by the Huffman Coder. Since the DCT output values never exceed a certain range, the FIFO that feeds the Huffman Coder actually produces a limited range of DCT values, divided by 8 (resulting in a 9-bit two's complement value). This makes the compression lossy but worth the loss in terms of the increased compression factor. The iDCT, given the exact same 9-bit values (sign-extended without restoring the bottom 3 bits) will reproduce a spatial domain 8x8 block that can be stored back into SDRAM on the receive side.

Huffman Coder / Huffman Decoder The Huffman Coder/Decoder (see figure 6) is a one-to-one mapping of 9-bit values to a bit stream and back. This is accomplished by a prefix-free variable-length encoding scheme designed to encode the most frequent values out of DCT in the smallest bit-strings. Unlike the DCT, Huffman is lossless in that the two's complement values input to the Huffman Encoder will appear exactly as they were presented at the output of the Huffman Decoder. When generated on the send side, the Huffman bitstream is stored in a shift register. The Huffman Decoder is provided a Huffman-from-wireless bitstream by the receive side shift register.

In figure 6, timer assignment is based on the magnitude of the input two's complement value (see table 2). This guarantees that the timer ticks when exactly the right number of $0s$, and afterwards when the correct number of magnitude bits, have been output. In the Huffman Decoder, the counter will count however many $0s$ appear consecutively at the input, and then use this value to count how many magnitude bits should then follow to reform a two's complement value.

|DIn|	**Set timer and initially shift data by ...**
$255 - 128$	8
$127 - 64$	7
$63 - 32$	6
$31 - 16$	5
$15 - 8$	4
$7 - 4$	3
$3 - 2$	2
1	1

Table 2: Huffman Encoder timer scheme.

Packet Generation The interface between the GTE and the communications engine is a pair of 248 wide packet buffers. On the send side, the packet buffer is a 248-wide and 16-deep FIFO. Once the shift register on the output of the Huffman Coder is filled to 248 bits, it is emptied as 248-bit parallel output to the packet

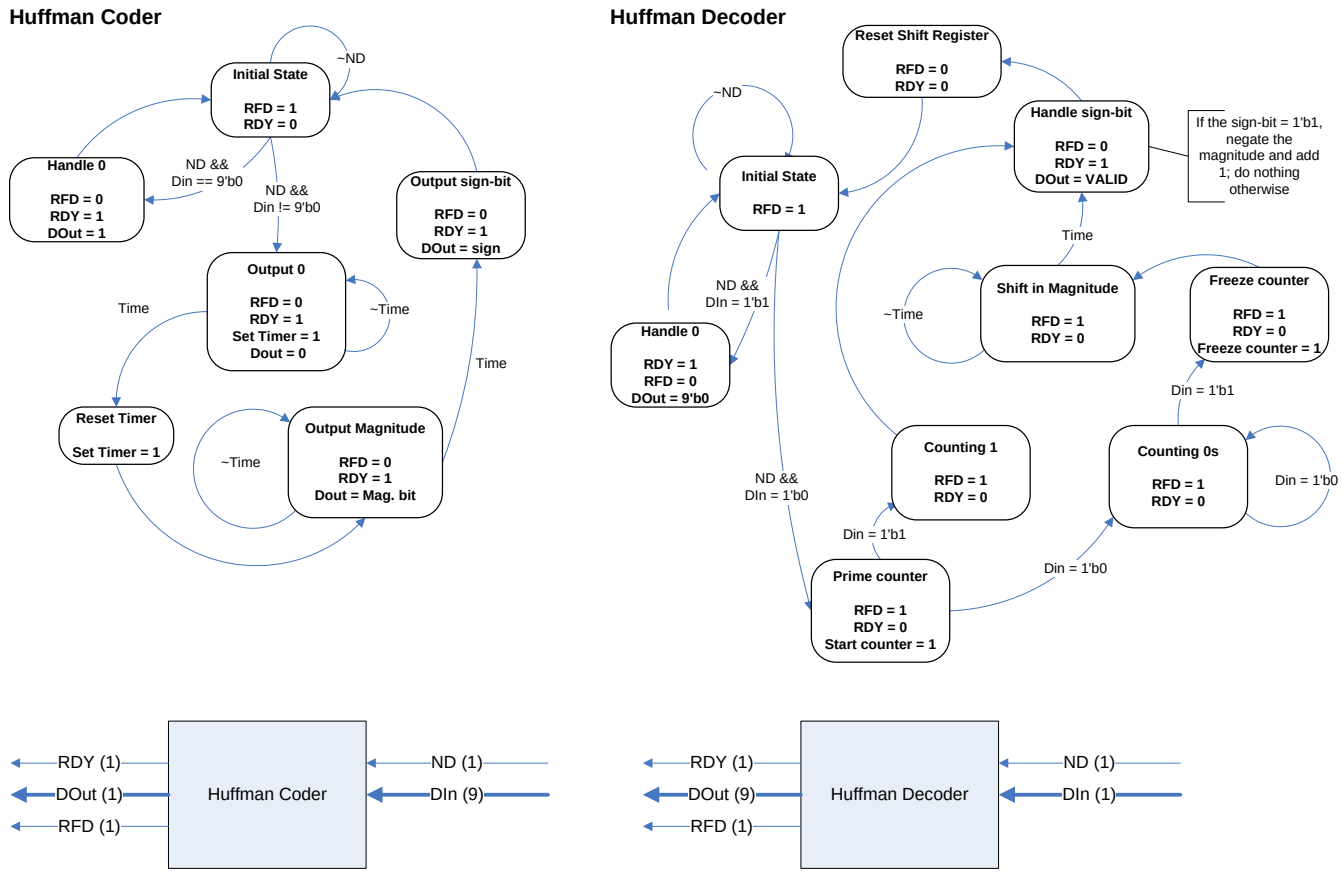


Figure 6: The Huffman Encoder and Decoder.

buffer, which interacts directly with wireless. On the receive side, the packet buffer is only a 248-wide shift register. When a packet is received, it is immediately placed in this shift register and shifted out into the Huffman Decoder. No dedicated FIFO is needed on the receive side because the shift register can empty out into the Huffman Decoder in several hundred cycles, whereas wireless only receives a new packet once on the order of tens of thousands of cycles.

Handshaking The handshaking scheme for each module (non-buffer) checks to see if the buffer feeding it has data, the buffer being fed by it has room, and the module itself is ready (given by RFD for the DCT/Huffman units). Specifically, the handshakes occur between each module and the buffer that feeds it. Assuming that the handshake is valid, the outbound buffer is guaranteed to have space, which is why no handshakes are needed between each module and the buffer they feed into. For the iDCT, the handshake will only pass one piece of data at a time, poll the RFD line for one cycle, and repeat, to ensure that the iDCT doesn't lose the first pair of each block.

2.3 Communications System

The Wireless Video Conferencing project depends directly on the availability of a lossless, reliable wireless link. The Communications Engine provides a means to transport packets in order, resending the lost or damaged ones. The Communications system is also responsible for creation and interpretation of the packets it transports, and utilizes a FIFO-like interface exposed by the graphics engine and the instant messaging engine. The Communications system includes a network controller responsible for establishing and maintaining a reliable connection with a single peer, and a radio interface that encapsulates the details of initializing and using the Chipcon CC2420 radio. Packet construction and interpretation happens at the interface with the packet buffers exposed by the Graphics engine, and the FIFO-like IM send and receive interface.

2.3.1 RF Transceiver Interface

Overview Interfacing with the Chipcon CC2420 proved much more troublesome than we had anticipated. Due to the fact that the SPI controller we use to communicate with the radio is blackboxed, and due to the incomplete documentation of the blackbox, debugging the RF Transceiver Interface was a nightmare. The interface communicates to the radio through a combination of SPI commands and control wires. The interface requires a stateful controller to interact with the SPI and to initialize the radio. This is a very extensive state machine, and in combination with the fact the controller must balance efficiency and reliability (a cause for complex state transitions), the FSM becomes very convoluted.

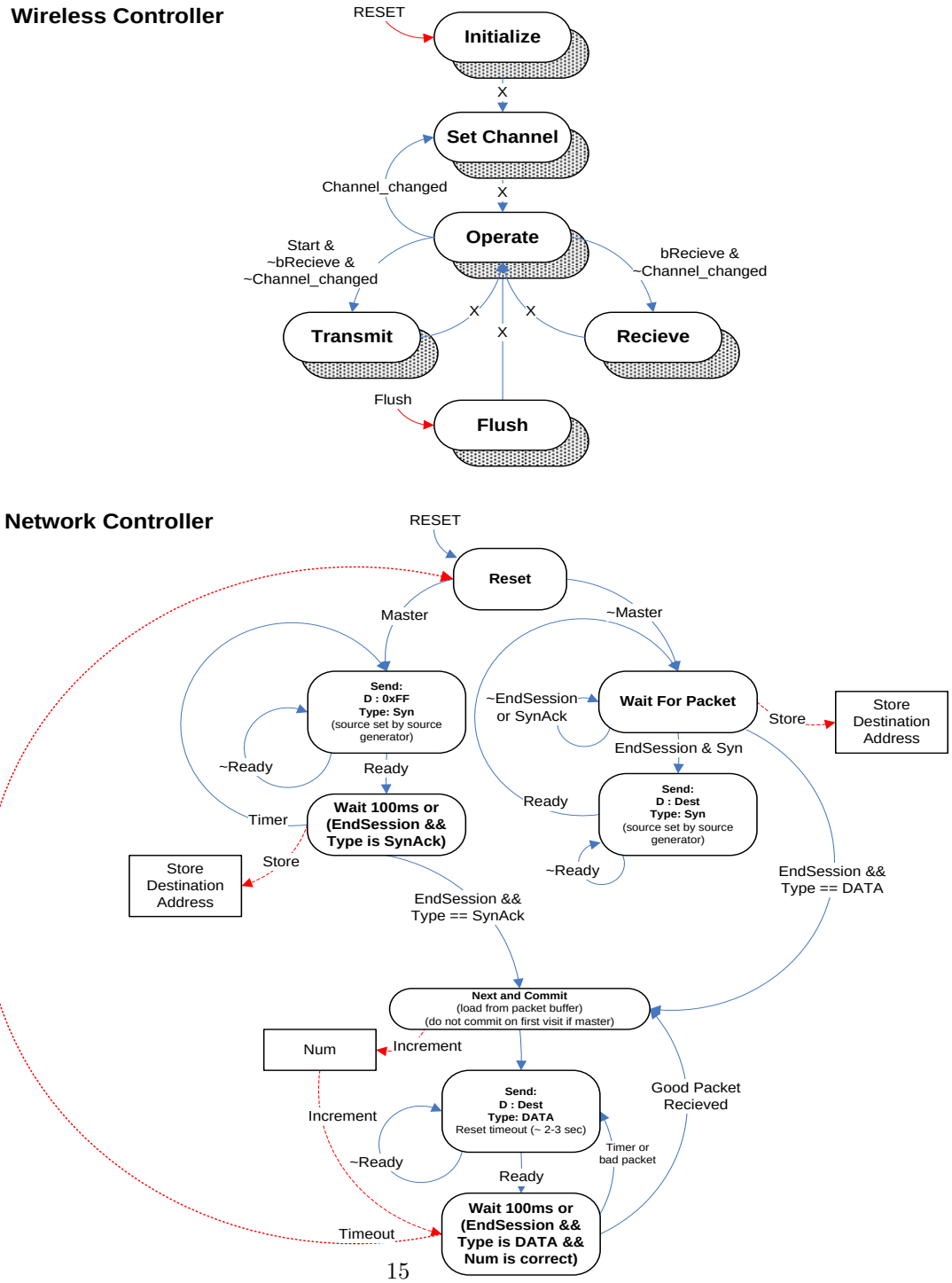
SPI Interface The interface controller is blackboxed, and exposes two FIFO-like interfaces: A ready - start outbound channel and a data-dataValid inbound channel. The timing of the Chip Select signal originating in this module is the cause for much of the confusion associated with this part of the project.

Initialization, Channel Select, Transmit, and Receive The operation of the controller for the RF Transceiver Interface is very complex and is best conveyed through a simple state diagram. Please refer to the state diagram for the RF Transceiver, and the Chipcon datasheet. On a high level, after the radio has been initialized, a channel is selected and the internal oscillator must be restarted. After that, the radio is receiving at all times other than during transmission. Any time a full packet has been received, it must be read out of the radio's FIFO. Since the radio channel is prone to interference, erroneous packets are possible. If the length field of the packet is not the expected value, the entire FIFO must be flushed. In all other cases, if the packet is not valid, or has an incorrect address, it is read out of the FIFO and discarded. When transmitting, the data is written to a FIFO on the radio chip and the controller is ready to receive while waiting for the channel to become available. When the radio can send, the controller initiates transmission and ensures it proceeds to completion. After a required delay, the controller is ready to send again.

2.3.2 Network Controller

Overview The Network Controller is a finite state machine responsible for lossless transport of packets across the radio channel. The controller establishes and maintains a connection with a single host, and delivers packets in order, resending lost/damaged ones if needed. The Network controller does not distinguish the content of transported packets, but does specify the type and ordering number field on outbound packets, and use these fields on incoming packets to guarantee reliable transport. Due to the nature of our video compression engine, reliability of the wireless network is required: if a packet containing a part of the video stream was lost or damaged, the video stream would be corrupt and the system may not be able to recover.

Figure 7: The Wireless Transceiver and Communications Engine.



If a packet containing a character of an IM message, the character would not show on the recipient's screen. Since radio is an inherently unreliable medium, a reliable protocol layer is introduced over the physical layer provided by the wireless chip (Chipcon CC2420). This is the Network Controller.

Connection and Communication The Network Controller introduces a master-slave protocol detailed below. During initialization, the controller enters either the Master or Slave mode, depending on the user's setting. When establishing a connection, a master sends packets looking for a slave. A slave responds, and the two begin exchanging data. To ensure reliable, ordered delivery, after each data packet is sent, the controller waits for a data response from the other peer. Due to the limited use of the ordering numbers, the process of ensuring ordered delivery of packets can be optimized somewhat: we only need to ensure the fact that any packet received is not a retransmission of the previous packet. In order to do this, the controller stores the ordering number of each packet it receives, and only accepts packets with a new number.

Master Mode When establishing a connection in master mode, the network controller sends out "SYN" packets at regular intervals of about 15ms. If the controller receives a "SYNACK" response, it saves the sender's address as the destination address and begins exchanging data with the slave. The first packet of data sent has an ordering number of zero.

Slave Mode When establishing a connection in slave mode, the network controller waits for any incoming packets. If the controller receives a "SYN" packet, it responds with a "SYNACK" to the sender's address. If the controller receives a data packet with an ordering number of zero, it saves the received address and data and begins exchanging data with the master. The first packet of data sent is the response to the master's data packet and has an ordering number of zero.

Transfer of Data Due to the optimization of ordering number checking outlined earlier, the procedure for transfer of data is identical in master and slave modes. Therefore, we can use the same state machine for both modes. The trick here is the fact that the slaves receives data before the master does, so after a connection has been established, the slave must immediately save data to the packet buffer, while the master needs to pull the next packet out of the packet buffer immediately, but does not save until later, when a response arrives. In order to accommodate this, the state machine for the transfer of data is cyclical, and is as follows: The next packet is created, sent, retransmitted if needed, the ordering number is incremented, the response is then saved, and the state machine loops around. When initialized in master mode, the controller will traverse the states in this order, but a slave will start at the last step. As a result, both master and slave mode can transfer data using the same state machine.

2.3.3 Packets: Video and Instant Messaging.

Because the network controller does not distinguish the contents of the packets it sends, we use the same methodology to transport video and ASCII for our instant messaging engine. We exploit the fact that our Huffman implementation is unable to produce a zero stream, we use a string of 240 zeros to mark the packet as non-video, and fill the remaining byte with a single ASCII character. Logic on the recipient's implementation will recognize the packet as non-video, and will send the data to the IM engine, rather than the graphics engine.

3 Design Metrics

Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	14,935	38,400	38%
Number of 4 input LUTs	14,699	38,400	38%
Total Number of 4 input LUTs	15,402	38,400	40%
Number of 4 input LUTs	14,699		
Number of Block RAMs	55	160	34%
Timing Constraints	Requested	Actual	Absolute Slack
TS_Y5_CLK	37.037 ns	36.035 ns	1.002 ns

Table 3: Device Utilization Summary.

Milestone	**Design**	**Coding**	**Debugging**
SDRAM Controller	2	2	0
SDRAM Arbiter	1	1	0
Local Video	2	2	0
Sub-sampled Video / PIP	2	1	12
Wireless Tranceiver	4	5	50
Graphics Transport Engine	5	10	15
Communications Engine	2	2	15
User Interface	10	15	2
Instant Messaging	5	4	10
PS/2 and Ethernet[6]	8	8	40
Sub-Totals	41	50	144
Total	235		

Table 4: Man hours summary.

4 Conclusion

Looking back, there was nothing particularly difficult about the project. Although the process of implementation and testing was extremely time-consuming, due to the tight requirements, and detailed descriptions of the checkpoints, there was nothing particularly complex about the design process. The only aspect of design truly open to us was the optimization of the system, but due to the low clock frequencies and loose timing constraints, and vast amounts of FPGA fabric available, optimization served an aesthetic, rather than any practical reason. Difficulty was not an issue with this project. Nevertheless, we feel compelled to mention the fact that wireless communication proved to be very unpleasant to design for: due to the small size of the CRC and the congestion of the radio medium in the lab, packets would often arrive garbled, resulting in infuriating hours of debugging only to find out that our design works perfectly when the wireless traffic dies down a little. Time management and regulation of the wireless channels were key to timely completion of the project.

A word of advice to those considering this class: the project is extremely time-consuming. Although we believe it is possible, with proper time management, to finish it in its entirety without pulling all-nighters, this class will interfere with your personal schedule and your other coursework. Time management is the key to success in this class. Although it is most unfair, but success in this course also depends greatly on the partner one works with. If both partners are well-equipped for digital design, the project will be a rewarding experience. If, however one of the partners is lacking, the entire team will find it difficult to complete the work by due date.

Having finished the required functionality of the Wireless Video Conferencing System, we were excited to work on the extra credit. We believe that this portion of the project, and the idea of implementing something we had designed from scratch, was most enjoyable. We have implemented an on-screen keyboard, instant messaging, an N64 interface to use the on-screen keyboard. We have also implemented scrolling text, and have expanded the UI required for the project greatly. We have implemented a PS/2 interface as an alternative to the on-screen keyboard, and have designed a quad-channel ethernet interface to allow the system to send uncompressed video at a respectable FPS. The PS/2 and ethernet interfaces, however were not included in the final submission due to the rising PAR times, and the lack of time to debug these large additions to the project's feature set. In the end, we feel we have learned much and matured greatly as a result of this project. This has been most students' largest implementation project yet, and we feel that the experience gained is most invaluable.

5 Suggestions

Given the obvious difficulty associated with managing a class of this caliber, we refrain from excessive criticism. Nevertheless, we feel it is important to remark on some of the most unpleasant shortcomings. First, the TAs should not assume the students' theoretical knowledge of the material associated with the project. For example, we found ourselves explaining what a FIFO is to more than a quarter of the class. Also, we feel that it is inappropriate to assume knowledge of concepts like DCT. EE20 is not a prerequisite for this course, and frequency decomposition is outside of the scope of the class. New concepts must be properly introduced. Also, we felt that the delays in checkpoint assignments were unfair to those in the early lab sections. For example, those Tuesday lab had only one full day to work on designs posted late the previous weekend. Otherwise, the course was not below expectations. We would like to thank the TAs for guiding the class through this long process of design and implementation.

We would like to communicate several ideas we feel would improve the CS150 project experience. First, If the project in its entirety would be available early in the semester, the completion rate would increase significantly: many students expressed a desire to have started on the bulky checkpoints earlier. Second, fewer blackboxes would greatly improve the project. Although we can understand the necessity of blackboxing the test suites for early checkpoints, there were a number of modules that were blackboxed for no reason: SPI and the Video Encoder, just to name a few. When we encountered bugs, we were never able to rule out the blackboxes. At one point, we noticed the fact that the text ROM blackbox was adding over 10 levels of logic to our critical path. We would have felt far more comfortable if we had access to all of the verilog in the implementation.

6 Acknowledgments and References

The Wireless Video Conferencing System may have used ideas and debugging tactics shared among the CS150 lab. This is expected due to the increasingly communal nature of the lab as the deadline approached. We would like to recognize those in the CS150 community who worked alongside us to complete the project by early deadline - this has been a long project and we feel we have all learned and matured from it, as well as from each other. Also, we would like to note that this project takes advantage of the FPGATOP, Register, Counter, ShiftRegister, and other verilog made available by Greg Gibeling for CS150 use. These files can be found in the CS150 labs, and in the RAMP project repository. Finally, we would like to recognize Greg Gibeling - an excellent mentor and resource without whom this class would not have been a tenth as resourceful.