Demonstration System for an Ultra-Low-Power Video Coder and Decoder

by

Rex K. Min

Bachelor of Science in Electrical Engineering and Computer Science
Massachusetts Institute of Technology, 1998

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of

Master of Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 21, 1999
[June 1999]

Signature of Author ....
Department of Electrical Engineering and Computer Science
May 21, 1999

Certified by ....
Anantha Chandrakasan, Ph.D.
~~Associate~~ Professor of Electrical Engineering
Thesis Supervisor

Accepted by ....
Arthur Clarke Smith, Ph.D.

Demonstration System for a Low-Power Video Coder and Decoder

by

Rex K. Min

Submitted to the Department of Electrical Engineering and Computer Science on 21 May 1999 in partial fulfillment of the requirements for the degree of Master of Engineering in Electrical Engineering and Computer Science

Abstract

A digital system showcasing two custom video processing chips was designed and implemented. The custom chips perform a forward and inverse discrete cosine transform (DCT) operation using only milliwatts of power. This work presents a hardware-based demonstration system for this chipset that captures and buffers full-motion video data, routes the data through the DCT and inverse DCT devices, and displays the resulting video stream on an LCD in real-time. This demonstration system is implemented with off-the-shelf components, including an NTSC video decoder, RAM and ROM memories, programmable logic devices, and a LCD. Control logic written in VHDL handles the flow of real-time video data through the system, coefficient quantization, synchronization signals for the LCD, and an I^2C serial bus interface. The system is contained on a single printed circuit board for simple, portable demonstrations. This system not only demonstrates the functionality and low power consumption of the DCT chipset with arbitrary real-time video data, but it also demonstrates direct, hardware-based applications of the most popular design concepts from undergraduate Electrical Engineering and Compute Science courses, including modularity and abstraction, top-down design, and facilitated testing and debugging.

Thesis Supervisor: Anantha Chandrakasan
Title: Associate Professor of Electrical Engineering and Computer Science

Acknowledgments

First and foremost, I give my sincerest thanks to Prof. Anantha Chandrakasan, my advisor for this work. In addition to his technical expertise, he offered the support, patience, and encouragement that made him a true mentor. I am grateful for his offer of this project and his guidance through this wonderful learning experience. I also thank Prof. Charles Sodini for evaluating and suggesting improvements to my system.

I thank everyone in Prof. Chandrakasan's and Prof. Sodini's research group. Thucydides Xanthopoulos was the architect behind the ultra-low-power DCT and IDCT processors. While going from graduate student to Ph.D. right before my eyes, he has supported his chipset through invaluable documentation, data, and discussion. Keith Fife was the master of Cypress programmable logic and surface-mount parts; I couldn't have soldered the fine-pitch, surface-mount CPLDs without him. Tom Simon provided a wide variety of technical advice and kept me sane with his unique brand of insanity, especially during those particularly late-night coding and debugging sessions. Dan McMahill provided guidance with the Accel design tools and all the resources available in the lab.

Finally, I thank my family, both at home and abroad, for their unconditional support throughout this project and my life. I am forever indebted to my mother and father, who instilled in me the worth of full effort and a positive attitude, and spent plenty of time and money on the phone sharing their compassion and confidence whenever I reached big stumbling blocks in this work. I love and thank them immensely. I also send special thanks to my grandfather Dong-Shik Min, who has been my strongest and most loyal supporter from Korea. This thesis is dedicated to them.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation for Low-Power Video Processing

Throughout the 1990s, multimedia and wireless communication have been two major driving forces behind technology. As computing power and storage have kept pace with Moore's Law, multimedia computers supporting high-quality digital video and sound have become a reality. In response to the increasing popular demand for "anytime, anywhere" communication, great advances have been made in wireless telephony and networking.

With these technologies firmly in place, the next step appears to be a combination of wireless and multimedia technology. Potential applications could include the replacement of the cellular telephone with a wireless video communicator, perhaps a product closely resembling Dick Tracy's fanciful "two-way wrist communicator" which, until recently, has been limited to the realm of science fiction.

The technology necessary for such a device is already in place. PCS cellular networks currently implement digital communication on support small, lightweight handsets. Mobile laptop computers can play movies and support videoconferences. However, the biggest challenge, power consumption, is yet to be conquered. For battery-powered devices, energy consumption can be as significant an issue as performance. Any cellular telephone or laptop computer user will attest to this.

How difficult is the task of engineering a low-power "wireless multimedia communicator"? Suppose that we were to implement such a communicator with a 1.8-volt, 233 MHz Pentium MMX processor, a common processor for laptop computers in early 1999. Suppose further that size constraints required that the entire device be powered by a single

"AA" battery, much like a pager. A high-capacity NiMH "AA" cell has a capacity of 1.25V * 1.45 AH * 3600 sec/H = 6500 Joules [1]. The processor, when fully utilized, dissipates 5.3 Watts of power [2][3], and would run for about 20 minutes before requiring a new battery. This estimate does not account for the power consumption of the display, as well as the wireless transceiver, which is typically the largest power consumer in wireless devices, would further reduce battery life.

Even if we were to replace the Pentium processor with a processor specially designed for low power consumption, such as a StrongARM microprocessor, we would still not be able to achieve days or weeks of continuous operation from a small, lightweight source of power. Any general-purpose processor, even one designed for low power consumption, is not sufficient. The ideal solution is a custom processor designed from scratch and optimized specifically for low-power manipulation of video data. In general, whenever low power consumption is a primary design goal, every component must be custom-designed to minimize the energy used. After all, a system consumes as much power as all of its individual components. A single, inefficient component will dominate the power consumption of the whole system.

1.2 The Wireless Camera Project

The ultra-low power wireless camera project is a demonstration of recent advances in low-power circuitry applied to a complete wireless video system [4]. The project is being undertaken by the Microsystems Technology Laboratory (MTL) in the Department of Electrical Engineering and Computer Science.

The low-power wireless camera collects video data and transmits the data to a base station for further processing and display. The system can be conceptualized by the block diagram in Figure 1.1. The transmitting section consists of the sensor, an analog-to-digital (A/D) converter, a compression and coding section, and a transmitter. The image sensor captures an image in analog form. The A/D digitizes the analog image. Next, compression logic reduces the data rate to satisfy the transmitter's requirements, and coding logic encodes the data with an error correction scheme. Finally, the transmitter sends this binary stream to a base station (a personal computer with additional hardware on an expansion card), where the image data is decoded, decompressed, and displayed. Students and faculty within MTL are optimizing each "block" of the wireless camera for minimal power consumption.

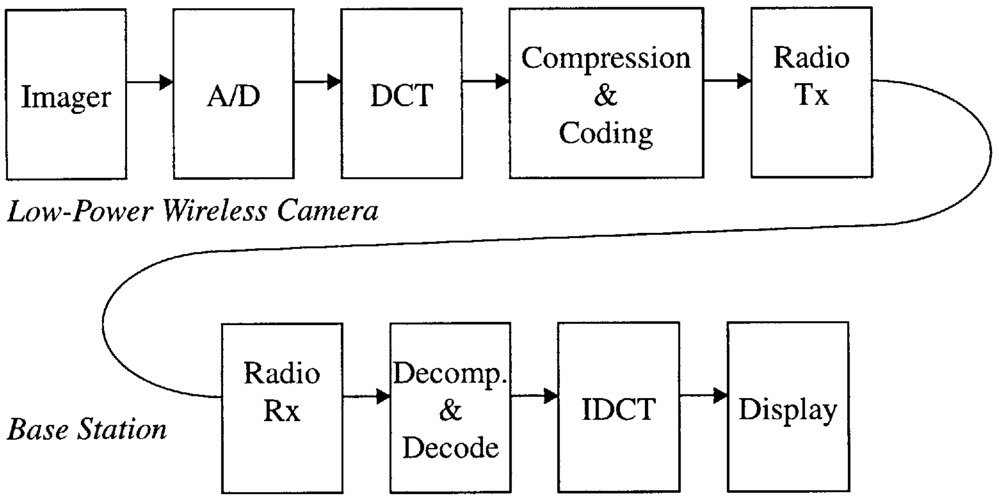


Figure 1.1: Block diagrams for the wireless camera project.

1.3 The Discrete-Cosine Transform and the DCT Chipset

Among the first devices to emerge from the wireless camera project are an ultra-low-power discrete-cosine transform chipset. One device performs a forward discrete-cosine transform; another performs the inverse operation. These devices can fill the DCT and IDCT blocks in Figure 1.1. This chipset was designed by Thucydides Xanthopoulos.

The discrete-cosine transform operation performed by this chipset is one of many frequency-domain transforms and resembles the well-known Fourier transform in this respect. The DCT, however, uses a purely real cosine as its basis function rather than the complex $e^{j\omega t}$. In video applications, the data is two-dimensional, and a two-dimensional DCT is required. The forward DCT for an 8x8 block of pixels is defined as

$$X[u, v] = \frac{c[u]c[v]}{4}\sum_{i=0}^{7}\sum_{j=0}^{7} x[i, j]\cos\left(\frac{(2i+1)u\pi}{16}\right)\cos\left(\frac{(2j+1)v\pi}{16}\right) \tag{1.1}$$

with $c[u] = 2^{-1/2}$ for $u = 0$ and $c[u] = 1$ for all other u.

The corresponding inverse is defined as

$$x[i, j] = \frac{1}{4}\sum_{u=0}^{7}\sum_{v=0}^{7} c[u]c[v]X[u, v]\cos\left(\frac{(2i+1)u\pi}{16}\right)\cos\left(\frac{(2j+1)v\pi}{16}\right) \tag{1.2}$$

with $c[u]$ defined in the same way as the forward DCT.

The DCT facilitates the compression of image data. When applied to a two-dimensional block of pixels, the transform removes spatial redundancy by resolving the image data into an orthogonal basis of cosines. The resulting block of decorrelated spatial coefficients is easier to quantize and compress, as seen in Figure 1.2. This property of the DCT is the motivation for its use in the MPEG video compression standard [5][6], as well as the data compression and decompression paths for the wireless camera project.

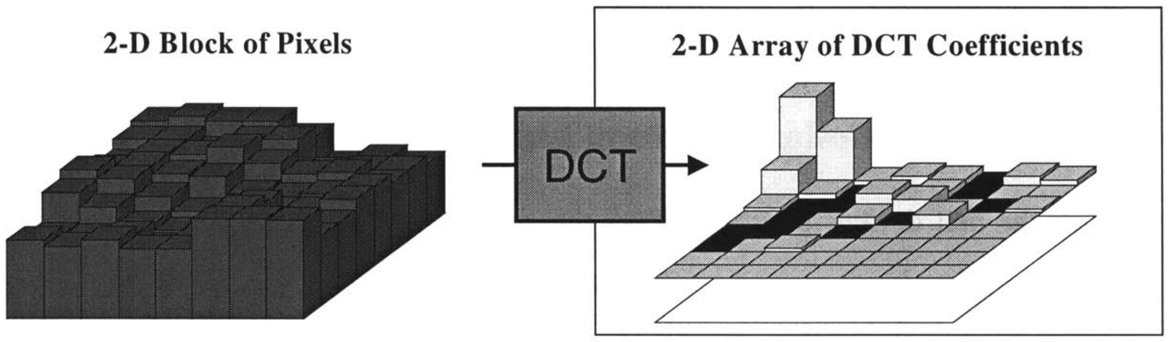


Figure 1.2: Block of pixels and its discrete-cosine transform.

This DCT chipset uses two general techniques to minimize power consumption [3]. The first is a set of novel data-dependent, power-aware algorithms. For instance, the algorithm performs varying amounts of computation for each pixel instead of "butterfly" multiplications on all the data. The processor analyzes the input data and dynamically reduces the precision of computations on low-magnitude or highly correlated pixels to conserve power. Furthermore, when power is limited, the processor limits the precision of the computation so that image quality degrades gracefully as available power is reduced.

The second technique for minimizing power dissipation is voltage scaling. The chipset is designed to run on the lowest voltage possible for the amount of computation required. Since the energy consumed by a device is proportional to the square of the input voltage, designing the chip to operate with lower voltage can provide a significant reduction in the energy consumed.

The DCT processor consumes an average of 4.38 mW at 1.56V when run at 14 MHz, and the IDCT consumes 4.65 mW at 1.32V and 14 MHz. A 233 MHz, 1.8 Volt Pentium MMX processor running a DCT algorithm in software would consume at least two orders of magnitude more power than this low-power processor.

1.4 A Demonstration System

This work is the first system-level proof-of-concept of the wireless camera project's core datapath. Up to the present, the focus the project had been the design of the individual low-power components. With the focus now moving toward system-level integration, a complete demonstration is necessary.

This system is also the first real-world evaluation and demonstration of the low-power DCT and IDCT processors, the first complete products of the wireless camera project. Before this demonstration system was complete, the functionality and power consumption levels of these processors had been evaluated through software simulations and with test vectors downloaded to the chips from a PC. These tests used still-image data and were performed separately on the DCT and IDCT devices, rather than as a complete system.

This system demonstrates the transform, quantization, inverse transform, and display of image data all at once, and in real-time. Furthermore, this system displays full-motion video limited only by the images on a videotape, camera lens, or television broadcast, so that the performance of the chipset can be evaluated in truly "real-world" situations.

Above all, this work is a challenge in real-time system design. Real-time data processing requirements of this system adds a substantial degree of complexity to a design. The successful design and implementation of this demonstration system requires careful design and engineering methodologies to promote simplicity and robustness.

1.5 Implementation Overview

This demonstration system is intended to model the video processing datapath of the wireless camera as closely as possible. The focus of the demonstration is the DCT chipset, the most recently completed components of the wireless camera project. The remaining blocks in the diagram are replaced one-for-one with off-the-shelf devices. For this demonstration, a versatile and compelling demonstration of the completed low-power stages is more important than the optimization of the entire system for power consumption. Rather than focusing on the individual components, as others within MTL already are, this work focuses on system integration.

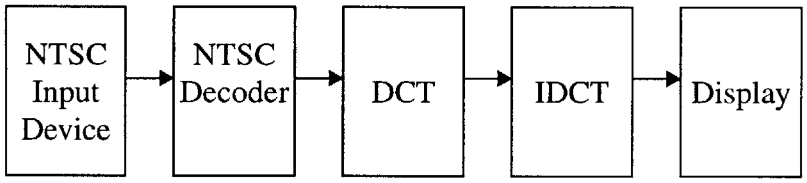


Figure 1.3: Adaptation of LPE for this work.

The block diagram for the wireless camera can be adapted for this demonstration system per Figure 1.3. In place of the sensor/camera, this system is designed for an arbitrary NTSC video input. Adoption of the NTSC standard allows a variety of video input devices to serve as the image source. Any device capable of generating an NTSC signal—most every TV, VCR, and video camera used in the United States—can provide a video input to this demonstration. In place of the A/D converter is an off-the-shelf NTSC video decoder. Video decoders are readily available and are common in devices such as multimedia cards for personal computers. Compression requires two steps: a frequency-domain transform followed by frequency-domain quantization and compression logic. Both are available on the low-power DCT chip, so there is no need to add an off-the-shelf solution.

Radio transmission and reception are not implemented in this demonstration. This system simply passes the compressed output of the DCT coder directly to the IDCT device. This not only eliminates a tangential source of complexity but also allows the entire system to be implemented on a single, portable printed circuit board.

Without a radio communication link, there is no need for a complete base station to receive and process transmitted data. A video display is sufficient, and a commercial flat-panel LCD display suits this purpose. As the display provides digital video inputs, it can be driven directly by the system without any additional D/A conversion.

1.6 Design Requirements and Goals

Specifications for data rate and format are taken directly from the specifications for wireless camera project. This system processes thirty non-interlaced frames of video per second. Each frame consists of either 128x128 pixels or 256x256 pixels, with eight bits of grayscale per pixel. Only the higher resolution is supported in this demonstration.

Because this system is a testbed for the LPE datapath and the DCT/IDCT chipset, the usual goals of test and demonstration systems apply. Flexibility and versatility are essential; the system should be able to simulate different operating scenarios with a minimum of effort and intervention. Portability is also a worthwhile goal. For portable demonstrations, a small, stand-alone circuit board is preferred to a large board that requires a connection to a computer. Since a testing environment can place strenuous demands on the components under evaluation, the design should be able to isolate and expose bugs and problems. After all, debugging is a primary use of test systems.

As the demonstration system is not intended for large-scale production, common commercial issues such as cost, the large-scale availability of components, and user interface are secondary. The primary goal is a successful, working design that offers a compelling demonstration of the DCT and IDCT devices in real-world operation.

Chapter 2

The Video Data Path

This chapter introduces the components that form the main video datapath of the demonstration system. Beginning with a conceptual block diagram, off-the-shelf devices are combined with the DCT chipset to realize each of the blocks. These devices are documented in this chapter. The clocking requirements for the datapath are also considered. Reflecting on the chosen design, it becomes clear that modularity and abstraction are essential for a successful, robust datapath.

2.1 High-Level Block Diagram

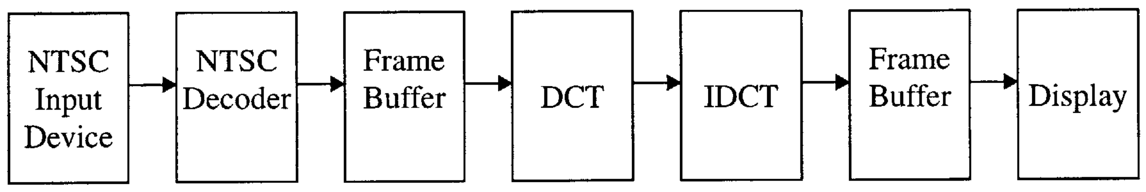


Figure 2.1: Block diagram of the real-time video datapath.

The full video data path consists of seven stages. An NTSC source generates an analog video signal. A video digitizer converts the analog signal into a series of digital still images, or frames. A ping-pong frame buffer stores and formats each frame for further processing. The DCT coder transforms each frame into the frequency domain and optionally performs rudimentary quantization. The IDCT decoder then decodes the coefficients and generates pixels. A second frame buffer re-assembles the frames, and a video display presents the final frames in real time. All stages operate at 30 frames per second.

Each of the "blocks" of the above diagram are implemented with one or two devices. The NTSC decoder is realized by a Brooktree Bt829A video decoder, a single-chip device that digitizes an NTSC video signal and provides a convenient digital output. Each frame buffer is implemented by a pair of IDT7008 static RAM devices. The entire frame buffer is capable of storing up to two full frames, one frame per IDT7008. The DCT and IDCT stages are the ultra-low-power processors to be tested. The video display is a Sharp active-matrix LCD. Figure 2.2 shows the new block diagram with the "blocks" replaced by the selected components.

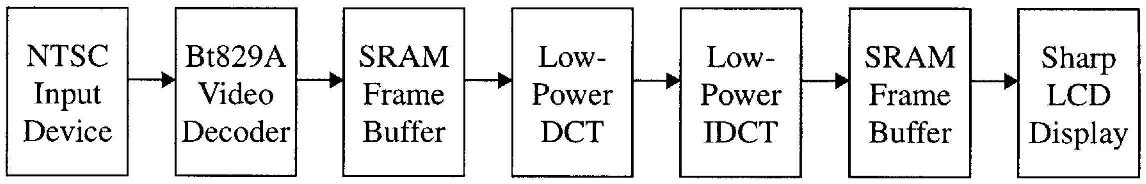


Figure 2.2: Block diagram with the chosen components.

2.2 The NTSC Decoder

The NTSC decoder receives an analog NTSC video signal and produces a digitized eight-bit grayscale image that can be passed through the video datapath. As NTSC is an ubiquitous standard for analog video in the United States, an NTSC interface allows for versatile demonstrations with a variety of video input devices.

The NTSC analog signal is digitized with the Bt829A video decoder. The Bt829A is a single-chip video digitizer produced by Rockwell Semiconductor [7]. The device supports NTSC, PAL, and SECAM formats as input and provides a digital YCrCb output with eight bits of luminance (brightness) and eight bits of chrominance (color) data. Image parame-

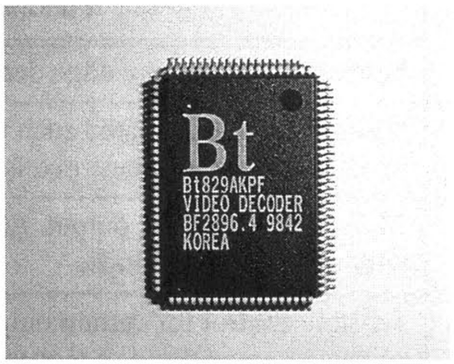


Figure 2.3: The Bt829A video decoder.

ters such as scaling, frame rate, and contrast can be adjusted by programming the device on startup. Table 2.1 describes the pins most relevant to this work.

Pin	Mode	Function
YIN	analog	analog composite input for NTSC
SCL, SDA	I/O wired AND	Clock and data lines for I^{2}C serial bus for device programming.
I2CCS	I	LSB of 8-bit I^{2}C device address for this chip.
$\overline{\text{RST}}$	I	Resets device to default state.
XT0I	I	NTSC clock oscillator signal (28.636 MHz).
VD[15..8]	O	Digitized luminance output in 16-bit mode.
VD[7..0]	O	Digitized chrominance output in 16-bit mode.
DVALID	O	Data Valid. High when a valid data (image pixel *or* blanking) data is being output. Low during blanking intervals or when no pixel is output due to scaling.
ACTIVE	O	Active Video. High when an the active image area (as opposed to a blanking interval) is being digitized.
VACTIVE	O	Vertical blanking. Low during active vertical lines.
FIELD	O	Odd/even field indicator. By default, "1" denotes that an odd field is being digitized.
$\overline{\text{HRESET}}$	O	Horizontal Reset. Falling edge denotes new horizontal scan line.

Table 2.1: Bt829A pin descriptions

Pin	Mode	Function
$\overline{\text{VRESET}}$	O	Vertical Reset. Falling edge denotes a new field.
QCLK	O	"Qualified Clock." Gated such that edges occur only when valid, active image pixels are being output.
CLKx1	O	14.31818 MHz clock output. All output signals are timed with respect to this clock.
$\overline{\text{OE}}$	I	Tri-state control for certain outputs. Effect of the pin can be programmed through register OFORM.

Table 2.1: Bt829A pin descriptions

To decode an NTSC video signal, the Bt829A requires only two active inputs: a composite analog NTSC signal at pin YIN, and a 28.636 MHz clock at XT0I. Using the clock as a reference, the Bt829A samples the analog signal at YIN, recovers line, field, and frame boundaries, and distinguishes between valid image data and supplementary data (such as closed-caption information) included in the blanking intervals between each line and field. Digitized pixels from each field are output on the 16-bit output VD. The eight most significant bits represent luminance information. The eight least significant bits represent chrominance and can be ignored by this demonstration system.

The VD output is clocked at 14.318 MHz, half the input clock frequency. The Bt829A provides a 14.318 MHz clock output on CLKx1 for synchronization with this output. Due to blanking intervals, decimation, and other processes that reduce data throughput, a valid pixel is not output at every clock. Hence, the Bt829A provides a variety of status signals, including DVALID, ACTIVE, VACTIVE, and FIELD, to indicate what portion of the NTSC video stream is currently being digitized. These signals are described fully in table 2.1. Figure 2.4 illustrates the output interface discussed here; the white VD blocks indicate valid data

When the system is powered on, the Bt829A must be reset by asserting the RST line low for at least four cycles. After reset, the Bt829A must be programmed with the desired

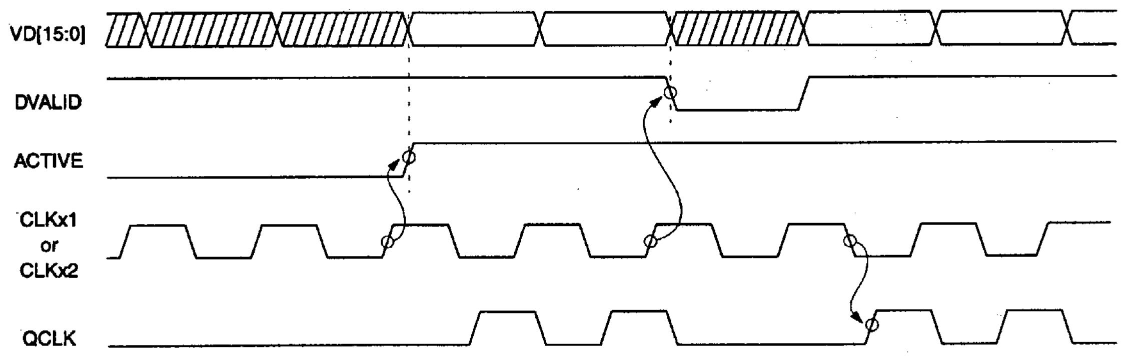


Figure 2.4: Output timing for the Bt829A, reproduced from [7].

parameters for digitization, such as the output frame rate, size of the digitized image, brightness and contrast, and impulse response lengths for the decimation filters. These parameters are loaded into the device through the SCK and SDA lines, which operate under the serial I^2C (Inter-Integrated Circuit) bus protocol [8].

For this system, the Bt829A was initially programmed to produce a 256x256 image with no temporal decimation and no image quality adjustments. The Bt829A would be programmed to digitize both fields of every frame to form a 640x480 image at 30 frames per second. The image would then be downsampled to 256x256 through decimation filters in both dimensions. Unfortunately, using both the even and odd fields in this way caused jagged horizontal lines with animated images.

An alternative approach, dropping the entire odd field in each frame, produced better results. Dropping one field halves the number of vertical lines from the NTSC standard of 480 to 240, resulting in sixteen fewer lines than the 256 lines required by wireless camera. It was decided that the better quality of the field-dropped 256x240 image was preferable to the 256x256 specification, which would require data from both fields.

2.3 The “Ping-Pong” Frame Buffer

Frame buffers provide flow control and data reordering between the video and coding stages in the system. The video stages (the NTSC decoder and LCD screen) require raster-ordered data: pixels are generated or received in left-to-right, top-to-bottom order. Furthermore, these devices are bursty; an NTSC signal, for instance, contains retrace periods where no image data is being sent. As we will discuss below, the DCT and IDCT processors require a completely different pixel format, and they are best suited for processing data at a constant throughput. Buffering pixels between such stages provides the system with the space and time to perform these format conversions.

2.3.1 Design of a Dual-Port Frame Buffer

A buffer receives data from one stage of the system, stores it, and sends the data to the following stage. Since the buffer is being read and written to simultaneously, it must function as a dual-port device. Since data must be reordered and moved at potentially different rates, the two ports should be as independent as possible.

The most robust design is a frame buffer that is large enough to store two full 256x256 frames, or $2*2^8*2^8 = 2^{17}$ bytes = 128 kilobytes. With two full frames of storage, the stage that writes the buffer is guaranteed enough storage for a full frame of data, while the stage that reads the buffer is guaranteed a full frame of valid data. The buffer is partitioned into two halves of one frame each, and the reading and writing stages read alternate halves of the buffer and swap in a ping-pong fashion at every frame. While more complicated schemes can perform a similar function with less memory, they reduce the timing flexibility of the reading and writing stages.

This type of ping-pong buffer is readily implemented with off-the-shelf memory devices. Two prominent issues arise in such a design. The first is the choice of static versus dynamic RAM. A second is the choice of a dual-port memory versus a single-port memory with additional control circuitry to simulate a second port. This design chooses dual-port static RAMs, but we first discuss the alternatives and the reasons for their dismissal.

One possible implementation for the frame buffers is a dynamic RAM (DRAM) instead of a static RAM. As DRAM is available in larger sizes than SRAM, a DRAM-based frame buffer can easily be implemented by a single chip. DRAM is also less expensive than SRAM. Unfortunately, DRAM requires additional refresh and addressing circuitry that would increase the complexity of the control logic. Additional logic requires additional macrocells on a programmable logic device. The task of refreshing the DRAM also adds more constraints on the timing requirements of incoming and outgoing data because data can not be read or written while the memory is being refreshed. In a commercial, high-volume product, the lower cost of DRAM would justify the additional control circuitry required. For a prototype system such as this one, where logic complexity is a greater enemy than cost, static RAM is the better choice.

Another design issue is single-port versus dual-port SRAM. If a single-port SRAM is clocked at twice the frequency of the incoming or outgoing data, the memory can be read and written on alternate cycles. The cost in additional control complexity is seemingly small, and like DRAMs, single-port SRAMs are available in larger sizes and lower cost than their dual-port cousins. However, two design issues make dual-port SRAM the more favorable option.

First, the use of a single-port SRAM would require tri-state outputs on the device driving the SRAM, as well as an additional eight-bit register to latch the outgoing data from the SRAM for the reading device. This circuit is illustrated in Figure 2.5. The IDCT device, which writes the second frame buffer in this system, was not equipped with tri-state outputs. Hence, a frame buffer based on single-port SRAM would require an additional tri-state driver for the IDCT, as well as the eight-bit latch. Compared to the single-

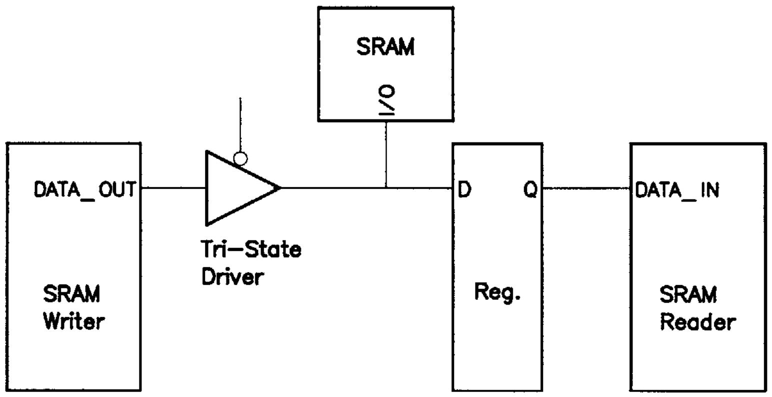


Figure 2.5: Alternative buffer design with a single-port SRAM.

chip implementation of NTSC decoding, this heterogeneous three-chip solution seems significantly less elegant.

Second, the clock rates of the two devices writing to and reading from the frame buffer are likely to be different. A true dual-port static memory can support fully asynchronous access to both ports. A scheme that simulates two ports through control logic would require additional synchronization logic.

After considering the drawbacks of single-port memories or DRAM, dual-port static RAM was deemed the best choice for implementing a frame buffer. The IDT7008 memory is chosen for this design.

2.3.2 Application of the IDT7008 Static Ram

The IDT7008 was the largest single-chip, 8-bit wide, dual-port static RAM that could be located at the time of this work [9]. The IDT7008 is a 2^{16} byte memory, enough for a single frame of 256x256, eight-bit video. Each frame buffer is implemented as two IDT7008 devices operating in parallel to simulate 2^{17} bytes of dual-port RAM.

The IDT7008 is an asynchronous device. Each port is equipped with a $R/\overline{W}$ line to select whether the port is reading and writing data, an $\overline{OE}$ line for tri-stating the port, and

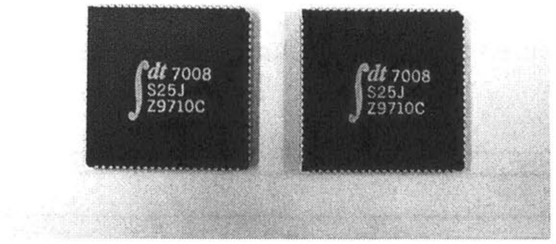


Figure 2.6: A pair of IDT7008 static RAM devices.

two chip enables, $\overline{\mathrm{CE}}_0$ and CE_1, that are internally ANDed together into a single enable signal. The two chip enable lines allow several IDT7008 devices to act as a single, deeper SRAM. Semaphore signalling between ports and arbitration logic are also provided, but they are not necessary for this design.

Pin	Mode	Function
A[15..0]	I	16-bit address
R/W	I	High to read SRAM, low to write SRAM
$\overline{\mathrm{CE}}_0$, CE_1	I	Chip selects. CE = ($\overline{\mathrm{CE}}_0$ AND CE_1)
$\overline{\mathrm{OE}}$	I	Output enable. High to tri-state the port.
I/O[7..0]	I/O	8-bit port. For SRAM reads, data appears here. For SRAM writes, data should be driven here.

Table 2.2: IDT7008 pin descriptions for each of two ports.

Figures 2.7 and 2.8 illustrate the reading and writing interfaces for each port. A port is read by driving both CE lines active, driving R/$\overline{\mathrm{W}}$ high, and presenting a valid address on the address lines. After the last of these conditions has been fulfilled, valid data appears on the port after a brief delay. The memory is written by driving both CE lines active and driving R/$\overline{\mathrm{W}}$ low. After a brief delay, the data at the port is written into the desired address in memory. The port's outputs are automatically disabled during a write cycle, but it is always a good idea to drive $\overline{OE}$ high any time a write is about to occur. For write operations, it is essential that the controller set-up and hold a valid address on the SRAM around a write cycle so that spurious writes to unintended portions of the memory do not occur.

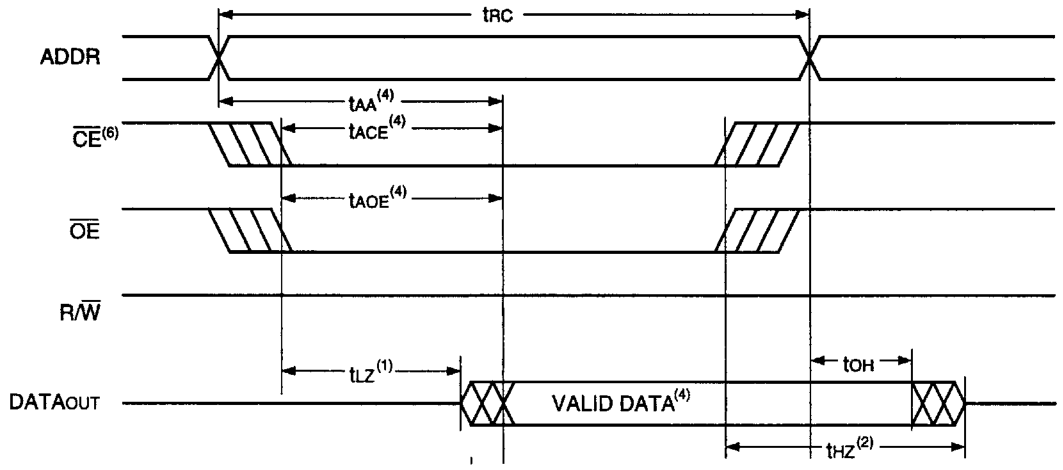


Figure 2.7: Read interface for the IDT7008, reproduced from [9].

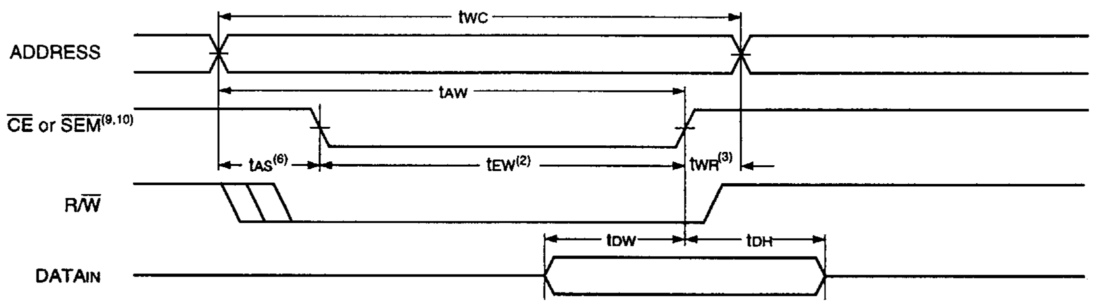


Figure 2.8: Write interface for the IDT7008, reproduced from [9].

With the buffer realized by two distinct SRAM devices, a natural implementation of the ping-pong scheme is for one SRAM to receive data from the previous stage while the other SRAM sends data to the next stage. Each memory stores a full frame of data. When both reading and writing transactions have completed, the two devices switch roles. The presence of two chip select lines on each port greatly assists this procedure; the additional chip selects on each port can act as the high address bit and select which SRAM is active for each stage.

In retrospect, the timing requirements for reading and writing an asynchronous SRAM could have been avoided with a synchronous SRAM. The IDT70908, a 2^{16} byte dual-port synchronous memory, provides a clocked control interface than the IDT7008.

2.4 The DCT Processor

The low-power DCT processor receives two-dimensional blocks of pixels and outputs the corresponding block of DCT spectral coefficients. Table 2.3 describes the functions of the DCT's pins.

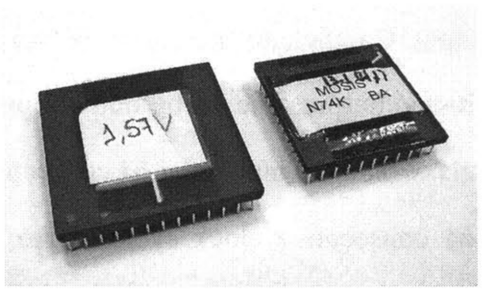


Figure 2.9: The DCT processor (left) and IDCT processor (right).

Pin	Mode	Function
DIN[7..0]	I	8-bit pixel input
DOUT[11..0]	O	12-bit DCT spectral coefficient output
LPEout	O	Serial output for selected DCT coefficients. Output represents 8:1 compression of initial pixel data.
SBIN	I	Start Block In. Denotes that the input pixel is the first pixel in a block of 64.
SBOUT	O	Start Block Out. Denotes that the output DCT coefficient is the first in a series of 64.

Table 2.3: DCT coder pin descriptions.

Pin	Mode	Function
RESET	I	Resets device to default, unprogrammed state.
TDI, TMS, TCK, TRST	I	JTAG-compliant testing and programming pins.

Table 2.3: DCT coder pin descriptions.

The DCT processor receives blocks of 64 eight-bit pixels, one pixel per clock cycle. The SBIN input must be pulsed high during the first pixel in each block to signify the start of a new block. No pause is necessary between blocks; the first pixel of a new block may immediately follow the last pixel of the previous block. Incoming pixel values are latched on the falling edge of the clock.

After a latency of 97 cycles from the first pixel, sixty-four DCT spectral coefficients begin to appear at the output. Coefficients are output in two ways, as a 12-bit wide output as well as a serial compressed output. The 12-bit output operates analogously to the pixel input. Like the input pixels, which are input over 64 consecutive clock cycles, DCT coefficients are output over 64 consecutive clock cycles. Like SBIN, the SBOUT signal is pulsed high during the first coefficient to signify the start of a new group of coefficients.

A serial output is produced on the LPE pin. The purpose of this pin is to provide a rudimentary form of 8:1 data compression. This output provides eight DCT coefficients per pixel block, at a precision of eight bits per coefficient. Only coefficients 0, 1, 2, 8, 9, 10, 16 and 17 are output. Hence, over 64 clock cycles, the LPE pin delivers eight coefficients of eight bits each, for a total of 64 bits—a reduction of 8:1 from the 64 *bytes* of input pixels.

On power-up, the DCT must be reset by asserting its RESET line high for several clock cycles. Six registers on-board the DCT must also be programmed through a JTAG-compliant interface. These registers determine parameters for the transform computation.

2.5 The Inverse Discrete-Cosine Transform Processor

Since the DCT and IDCT processors were designed by the same student, it is not surprising that they have nearly identical interfaces. Table 2.4 summarizes the functions of the relevant pins. Coefficients are received by the IDCT processor in groups of 64. The coefficient input is 12 bits wide; there is no serial input analogous to the DCT's serial output. The SBIN signal must be pulsed high during the first coefficient. After a latency of 90 cycles from the first coefficient, a block of 64 pixels is produced on an 8-bit output. The SBOUT signal is pulsed high on the first pixel of each block. The SBIN and DIN lines of the IDCT are directly compatible with the corresponding outputs of the DCT. These two devices may simply be connected together.

The IDCT processor must be reset by holding its RESET line high. No device programming is required.

Pin	Mode	Function
DIN[11..0]	I	12-bit DCT coefficient input
DOUT[9..0]	O	10-bit pixel output
SBIN	I	Start Block In. Denotes that the input coefficient is the first in a series of 64.
SBOUT	O	Start Block Out. Denotes that the output pixel is the first in a block of 64.
RESET	I	Resets device to default, unprogrammed state.

Table 2.4: IDCT decoder pin descriptions

2.6 The Video Display

The Sharp LQ14D412 is an active-matrix TFT liquid crystal display that supports 640x480 pixels in 24-bit color. Unlike conventional computer monitors which require a partly analog interface, the synchronization, clock, and data signals for the Sharp LCD are all received in the digital domain. A digital controller on-board the LCD processes these signals and drives the actual display. This level of separation between the display's interface and its actual pixels results in a remarkably flexible interface. For instance, the number of pixels in a line or lines in a frame may vary without introducing jitter in the display, and arbitrarily low refresh rates will not cause flicker. The latter property is excellent for a 30 frame per second video system, as computer monitors are driven well above 60 Hertz for flicker-free images.

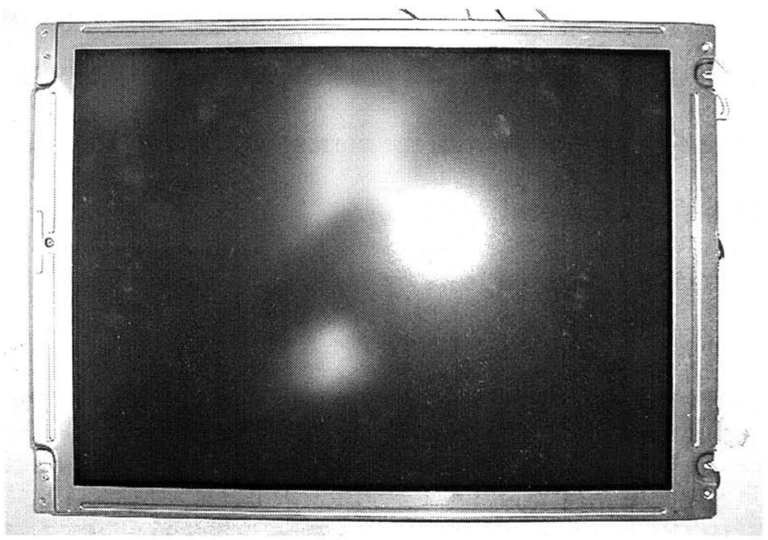


Figure 2.10: The Sharp LCD.

As summarized by table 2.5, the control interface is similar to standard VGA. Edges on HSYNC and VSYNC delineate the start of a new line or frame of video. For its 640x480 display mode, in which every pixel on the LCD is being utilized, the LCD requires roughly 800x550 clock cycles, maintaining a sort of backward compatibility with analog displays which required a retrace interval between each line and frame. The data enable signal ENAB indicates where valid pixel data begin on each horizontal line.

Pin	Mode	Function
HSYNC	I	falling edge indicates start of horizontal scan line
VSYNC	I	falling edge indicates start of new 640x480 frame
ENAB	I	rising edge indicates start of valid pixel data in a line
RED[7:0] GREEN[7:0] BLUE[7:0]	I	eight-bit RGB pixel values

Table 2.5: Sharp LCD pin descriptions (signal polarities for 640x480 mode).

Since the frames passing through the system are 256x240 grayscale, this demonstration system does not require the full display area of this LCD. Additional control logic is responsible for displaying a 256x240 grayscale image attractively on the LCD display.

The choice of the Sharp LCD was not without consideration. In its favor are a simple, abstract digital interface with no minimum refresh rate, a crisp, clear image, and immediate availability at the time of design. However, there are two significant disadvantages—this particular LCD screen is both large and somewhat non-standard. Its large size makes portable demonstrations more cumbersome. The 640x480 pixel display is unnecessarily large for display of a 256x240 pixel image; pixel doubling would only make the image more grainy and detract from the demonstration. A non-standard interface port presents two additional problems. First, it requires a custom adapter, which was fabricated as a PCB for this work. Second, it precludes the use of other output devices, such as a computer or television monitor, in contrast to the versatile NTSC video input.

There are several alternative display options. An NTSC encoder chip, analogous to the Bt829A decoder, could drive any standard NTSC video display. Although the versatility of this solution is appealing, the quality of NTSC displays, however, is generally lower than their non-interlaced counterparts. The interlacing itself may also introduce undesirable effects. A rather complex option is a VGA output. Such an interface opens the door to high-quality, non-interlaced displays, but adding a VGA driver interface would require at least one additional block to the system—we are basically adding a video card to the system!

2.7 Clocking Requirements

As suggested by Figure 2.11, the system can be divided into three distinctly different clocking regimes, one each for the Bt829A, the DCT/IDCT chipset, and the LCD display.

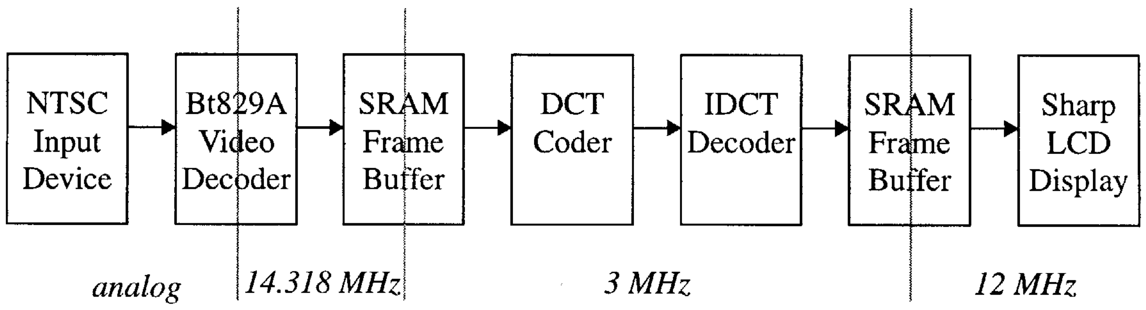


Figure 2.11: Timing requirements for the real-time video datapath.

The Bt829A requires a 28.636 MHz clock to decode an NTSC video signal. Pixel data are output at half this frequency, 14.318 MHz. These clocking requirements are fixed by the Bt829A.

The LCD offers more clocking flexibility than the Bt829A. Documentation for the LCD suggests a 25 MHz clock for a 60 frames/second rate, assuming about 750x520 clock cycles per frame. (Recall that the number of clocks per frame is higher than the number of pixels on the screen to simulate some horizontal and vertical blanking time.) Because this system is running at 30 frames/second, a clock rate close to 12.5 MHz would be ideal. A rate of 12 MHz was chosen both for the ready availability of clock oscillators and the ease of numeric computation.

The DCT and IDCT support a wide range of clock frequencies; they can operate at tens of megahertz, or tens of kilohertz. Their clock should be chosen fast enough so that the incoming pixel data can be processed in real-time without frame drops. Since the frame rate is 30 frames per second at 256x256 = 2^{16} pixels per frame, the chipset must process 1,966,080 pixels per second. This is the minimum allowable clock rate for the chipset. While a 2 MHz clock is sufficient, it would leave little idle time between frames, imposing strict timing requirements on the surrounding stages. A more reasonable choice is a 3 MHz clock, which provides a generous 50% margin of safety. Another compelling reason for choosing a 3 MHz clock is that it is exactly one-fourth of 12 MHz, the clock rate chosen for the LCD. The DCT chipset can now operate synchronously with the LCD section and the intervening frame buffer.

2.8 A Design Lesson: Modularity and Abstraction

A modular design divides its functionality among a number of distinct units, or "modules," allowing each module to be designed and tested independently. Modular design is evidenced by block diagrams, in which designers categorize functionality into distinct, independent blocks. In a well-designed modular system, each module presents its functionality through a simple input-output interface, and hides the low-level details of its operation in a metaphorical "black box" [10]. Functionality is *abstracted* into modular units.

As each block of this system is implemented by a single device (or pair of devices), the actual hardware maintains the level of abstraction offered by a block diagram. The video input and output, in particular, provide a tremendous amount of functionality through a simple digital interface.

The Bt829A, for instance, carries out all NTSC decoding and digitizing on a single device. Using the single-chip Bt829A eliminates the need to work with the lower-level elements that are required for video digitization, such as A/D converters, phase-locked loops, and digital and analog filters. Most of these components are on-board the Bt829A and are hidden from the system; the five discrete analog components connected to the Bt829A occupy little more than a square inch on the circuit board. Since the goal of this project is to design a real-time video processing system, and not to design a better NTSC decoder, the Bt829A provides a valuable abstraction of digital video input.

The Sharp LCD screen provides a similar abstraction for video output. Its versatile, all-digital interface conceals the details of drawing and refreshing the display. Furthermore, as discussed in section 2.6, the LCD's built-in controller can withstand variations in line length, number of vertical lines, and refresh rate. The LCD can be treated as a black box for digital video output.

The clocking discussion of section 2.7 suggests that an even higher-level abstraction is present in this design. There is a fundamental reason for the three different clocking regimes: different sections of the system process data in different ways. The video input and output stages require relatively high clock rates, since each of these stages process 480-line frames with long retrace intervals between frames. The DCT and IDCT processors process a smaller, 256x256 frame, and can operate on a slower clock. Furthermore, these stages require different pixel formats. The video input and output stages work with pixels in a raster-scan (left to right, top to bottom) order, while the DCT chipset works with pixels in two-dimensional, 8x8 blocks.

It is evident that there are three high-level video functions in this system: digitization, processing, and display. Furthermore, much of the complexity in this system arises from the interconnection of these three elements. The solution employed in this system, the dual-port frame buffer, isolates each of the three high-level functions. These buffers introduce the necessary latency and memory to reorder pixels and alter their rate of flow, and to allow each of these three major sections to operate asynchronously. In other words, the frame buffers are physical abstraction barriers between each of the three high-level functions in this system.

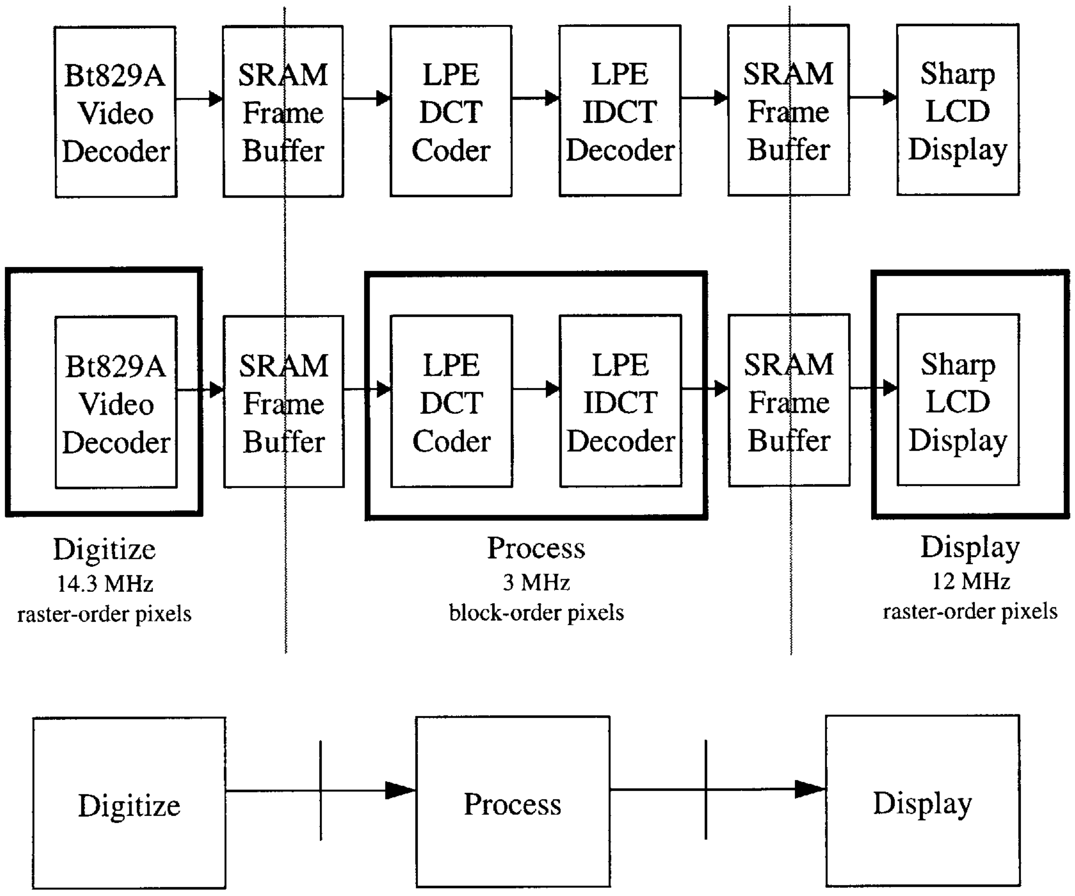


Figure 2.12: Derivation of a higher-level, three-block abstraction.

Chapter 3

The Control Subsystem

This chapter discusses the control logic for the demonstration system. Control logic is implemented on Cypress programmable logic devices using VHDL. The devices realize a series of small finite-state machines that control small segments of the system. The FSMs control the interfaces of the video data path and device initialization. The wise use of programmable logic yields a design that is resilient to growth and change.

3.1 The Cypress CPLD

Control logic is stored in complex programmable logic devices (CPLDs). The particular device chosen for this work is the Ultra37256, manufactured by Cypress Semiconductor Corporation [11]. Ultra37256 devices consist of 256 macrocells arranged into 16 logic blocks. The devices are "in-system reprogrammable," meaning that they can be reprogrammed even after placement onto the circuit board. Logic is programmed in VHDL, a high-level logic description language, and is compiled and simulated in Warp, a proprietary integrated development environment for Cypress programmable logic devices.

Figure 3.1: The Cypress Ultra37256 CPLD.

The CPLD chosen for this work is a CY37256P160-154AC. The 160 denotes its 160-pin package, and the 154 denotes its stated maximum operating frequency of 154 MHz. Out of its 160 pins, 128 pins are available for use as input or output signals. 128 macrocells are associated with these 128 pins, one pin per macrocell. The remaining 128 macrocells are "buried," for their values are not accessible from a pin. These macrocells can be used to hold state, intermediate logic, or other logic values that need not appear at the outputs.

Prior successes of other groups in MTL with this device family [12][13] greatly influenced its selection. The 154 MHz version of the Ultra37256 was the fastest available when the system was designed. Despite its higher cost over slower models, it was chosen to minimize propagation delay through the control logic.

As such a high-speed CPLD may seem excessive for a system whose fastest oscillator is 28.6 MHz, a detailed discussion of CPLD speed ratings is worthwhile. The maximum operating frequencies for CPLDs can be somewhat misleading; the actual maximum clock speed for a CPLD in a real-world system is dependent on the complexity of the logic realized within the system. The CY37256P160-154AC, though rated for 154 MHz, has a maximum propagation delay of $t_{PD} = 7.5$ ns and a setup time of $t_S = 4.5$ ns. Hence, for a simple circuit that requires a CPLD output to be fed back into an input, we should operate the device at no higher than 1/(7.5 ns + 4.5 ns) = 77 MHz, half the rated maximum frequency. Perhaps the 154 MHz rating indicates that computation can be performed on both phases of a 77 MHz clock with careful pipelining of the logic within the device.

Second, signals that pass through the CPLD's interconnect matrix *n* times incur *n* times the propagation delay t_{PD}. Signals that arise from particularly complex VHDL structures, such as highly-nested IF statements and asynchronous overrides of clocked logic, tend to require more than one macrocell on the CPLD. Every time a signal moves from one macrocell to another for further processing, it incurs an additional propagation delay. Hence, propagation delays for some signals can be double or triple the rated t_{PD}.

Third, even if the device could support computation at a 154-MHz clock frequency, it would be unwise to use such a CPLD in a system with a clock frequency anywhere near 154 MHz. Doing so would imply that the other devices that interact with the CPLD have zero delay! The actual maximum operating frequency depends on the sums of *all* the setup times, hold times, and propagation delays in any path. Suppose that n devices in a critical path with the CPLD, and for simplicity assume that all n devices have similar timing requirements as the CPLD. For this path, the actual maximum operating frequency is the CPLD's rated maximum operating frequency divided by n. With devices operating on clocks as high as 28.6 MHz, choosing the fastest available Ultra37256 device no longer seems like an extravagant decision.

3.2 Real-Time Control for the Video Datapath

Control logic is essential for managing the interface between devices in the video datapath. When data is produced by one device, control logic ensures that the next device is prepared to receive and process it.

3.2.1 Overview

Each stage, be it a frame buffer, coding chip, or display, operates in basically the same way. A stage remains idle until signalled by the previous stage that a new frame is ready for output. The stage then reads the video data and process it appropriately. When the stage is ready to pass the frame's data on to the next stage, it produces a "frame ready" signal of its own, and begins sending data. Figure 3.2 depicts this conceptual view.

The first stage, the video decoder, produces the initial "frame ready" signal as it digitizes the NTSC signal. This signal propagates down the entire datapath. Hence, the system

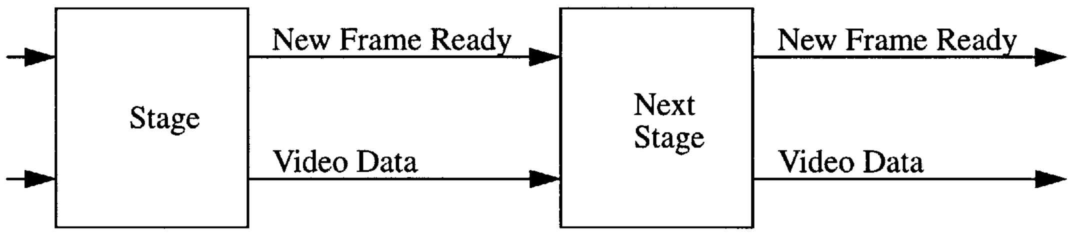


Figure 3.2: Conceptual view of stage interactions.

depends upon the frame boundaries of the analog NTSC signal to control the flow of frames through the entire system.

Note that the propagation of control signals is entirely one-way. When a stage sends data, it need only signal the next stage that data will be on the way shortly. The sender does not wait for any acknowledgment. The receiver is completely responsible for being prepared for each new incoming frame. Assuming that this condition can be met, this one-way flow of control and data greatly simplifies the interfaces between stages of the video datapath.

Table 3.1 introduces the five FSMs that control the video datapath. Each is responsible for controlling data flow at the interface between two adjacent devices in the datapath. The NTSC, DCT, IDCT, and LCD state machines each manage the transfer of data to or from a frame buffer. The DCT and IDCT FSMs are further responsible for converting the raster-order pixel data to and from the 8x8 pixel block format required by the DCT and IDCT devices. The NTSC and IDCT FSMs, which operate the write (left) port of a dual-port frame buffer, generate a "next frame ready" signal for the next FSM in the chain, for the frame buffers, unlike the other stages in the datapath, are not capable of generating this signal by themselves. The COMPRESS state machine binds the DCT output with the IDCT input. It is responsible for converting the DCT's serial output, when active, to a parallel input for the IDCT.

FSM	Duties
NTSC	Manage data transfer between video decoder and first frame buffer Signal DCT state machine when a full frame has been loaded
DCT	Manage data transfer between first frame buffer and DCT coder Reorder pixels in frame buffer into 8x8 block format
COMPRESS	Manage data transfer between DCT and IDCT devices Perform serial-to-parallel conversion on DCT serial output
IDCT	Manage data transfer between IDCT and second frame buffer Reorder pixels in frame buffer into raster format Signal LCD state machine when a full frame has been loaded
LCD	Manage data transfer between second frame buffer and LCD Generate LCD control signals

Table 3.1: FSMs that control the video datapath.

3.2.2 The NTSC State Machine

The NTSC state machine manages the flow of data between the Bt829A video decoder and the write port of the frame buffer. This FSM toggles between one of two states. It is *waiting* when idle, and *loading* when the Bt829A is digitizing an even field.

The *waiting* state is the FSM's idle state. The SRAM's port is disabled, and the address counter is reset to zero. The FSM remains in the *waiting* state until the Bt829A signals the arrival of an even field by driving both VRESET and ACTIVE to a logical zero. On the clock edge following this condition, the FSM advances into the *loading* state and also toggles WHICHRAM, the signal that selects which of the two SRAMs is being loaded by the Bt829A. Toggling WHICHRAM also notifies the state machine that controls the DCT that a new frame of data is available.

In the *loading* state, the FSM checks whether a valid pixel is being produced by the Bt829A at every clock. If both DVALID and ACTIVE are high, the data byte that is output by the Bt829A is a valid pixel of the digitized 256x240 image. In this case, the SRAM's port is enabled so that the pixel can be written, and the SRAM address counter is incremented. If either DVALID or ACTIVE is low, the Bt829A data lines contain invalid data. Hence, the port is disabled, and its address is held at its current value. Once 240 lines of the image have been read, the FSM returns to the *waiting* state, where it remains until a new even field is digitized.

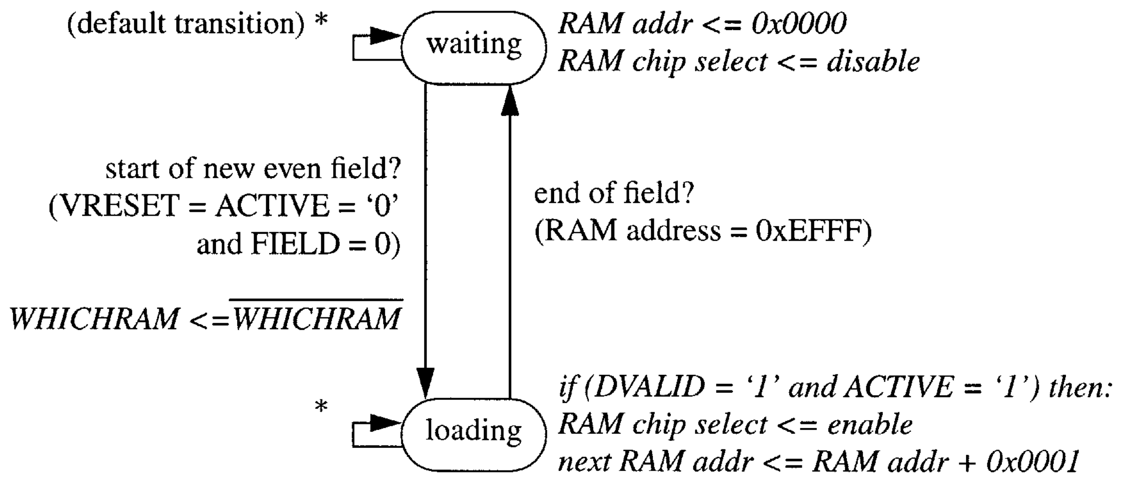

(default transition) *
waiting
RAM addr <= 0x0000
RAM chip select <= disable
start of new even field? (VRESET = ACTIVE = '0' and FIELD = 0)
WHICHRAM <= $\overline{WHICHRAM}$
end of field? (RAM address = 0xEFFF)
*
loading
if (DVALID = '1' and ACTIVE = '1') then:
RAM chip select <= enable
next RAM addr <= RAM addr + 0x0001

Figure 3.3: State transitions for the NTSC FSM.

3.2.3 The DCT State Machine

The DCT state machine manages the flow of data between the first frame buffer and the DCT processor. Much like the NTSC state machine, it is implemented in two states, *waiting* and *loading*. These states also serve analogous functions, except that data is read *from* the frame buffer and *to* the DCT coder.

Figure 3.3 illustrates the state transitions for the DCT FSM. The FSM remains idle in the *waiting* state and polls the WHICHRAM signal from the NTSC state machine. Recall that this signal is toggled by the NTSC FSM to signify that a complete frame has been written by the video digitizer. The DCT FSM synchronizes the WHICHRAM signal to its

3 MHz clock and advances to the *loading* state once it detects that the synchronized signal has changed polarity since the last clock.

In the *loading* state, the FSM sends data from the SRAM's right (read) port to the DCT coder. Instead of counting sequentially through the address space, however, the FSM counts the port address in such a way that the pixels are sent in 8x8 square blocks. (The pixels within each block, as well as the blocks themselves, are sent in raster order.) The addressing scheme can be conceptualized as the "crossing" of outputs of a sequential 16-bit counter as shown in Figure 3.4. This is basically how the counter is implemented in VHDL.

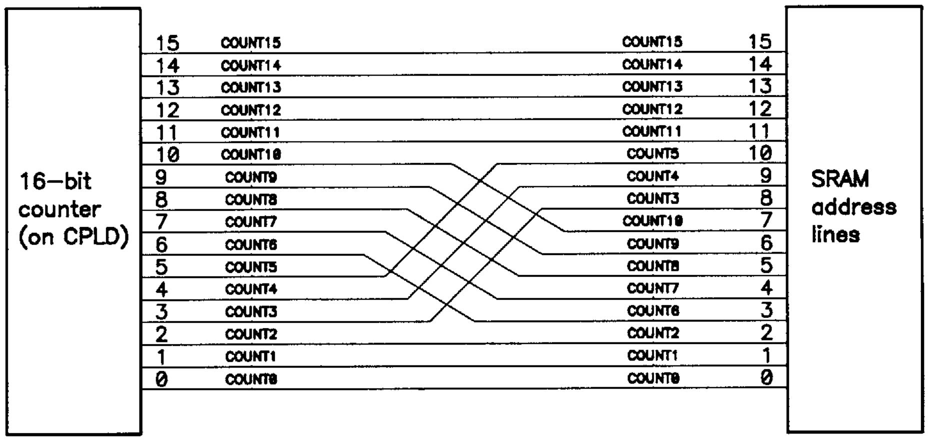
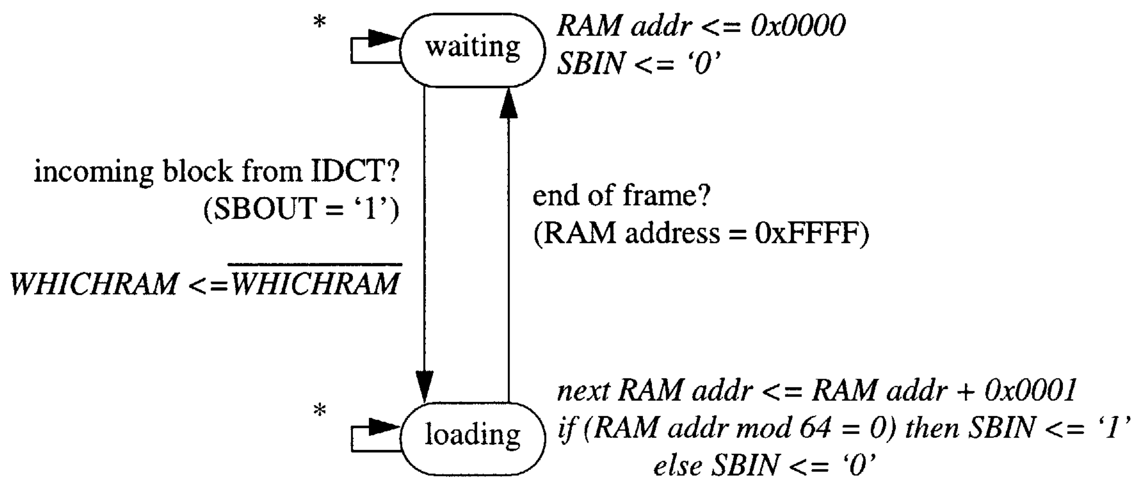


Figure 3.4: Addressing the frame buffer for raster-to-block conversion.

As the DCT coder requires a start-of-block signal at the beginning of every 64-pixel block, the FSM generates such a signal in combinational logic. A pulse is sent to the DCT's SBIN input whenever the SRAM's port address, modulo 64, is exactly zero.

The FSM returns to the idle *waiting* state once it has finished sending all 256x256 pixels in the frame to the DCT. Although the video decoder has actually provided only 256x240 pixels, this stage assumes a full 256x256 frame for maximum flexibility and pro-



*
waiting
RAM addr <= 0x0000
SBIN <= '0'
incoming block from IDCT?
(SBOUT = '1')
WHICHRAM <= $\overline{\text{WHICHRAM}}$
end of frame?
(RAM address = 0xFFFF)
*
loading
next RAM addr <= RAM addr + 0x0001
if (RAM addr mod 64 = 0) then SBIN <= '1'
else SBIN <= '0'

Figure 3.5: State transitions for the DCT FSM.

cesses the sixteen extra lines as if they were a part of the image. The last sixteen lines are cropped on the LCD.

3.2.4 The IDCT State Machine

The IDCT state machine connects the IDCT processor with the second frame buffer. As seen in Figure 3.5, the structure and function are identical to the NTSC FSM. A SBOUT (start of block) signal from the IDCT causes the FSM to transition from the *waiting* state to the *loading* state. Like the NTSC FSM, this state machine also toggles a WHICHRAM signal to access the other SRAM in the frame buffer and to notify the next FSM in the chain that the previously loaded frame is ready. Since the DCT FSM guarantees that each frame contains exactly 2^{16} pixels, the IDCT state machine simply reverts to the *waiting* state as soon as it completes writing the last pixel. This pixel should correspond to the last pixel of the last block of each frame produced by the IDCT processor.

The IDCT state machine performs two additional functions. First, the FSM actually buffers each incoming pixel from the IDCT chip for one clock cycle, effectively introducing a two-stage pipeline. This delay is introduced in case future expansion calls for high-latency processing of the data at this stage, before it is displayed on the LCD. Second, the FSM hard-limits the 10-bit, two's complement pixel values into an unsigned 8-bit range. Due to roundoff effects within the DCT and IDCT devices, the unsigned 8-bit input to the DCT can result in a value slightly less than zero or slightly more than 255. Since only the least significant eight bits are used by the LCD, the FSM's hard-limiting ensures that such overflows and underflows will not cause, say, a black pixel to appear bright white on the LCD display.

3.2.5 The LCD State Machine

The LCD state machine manages the flow of data between the second frame buffer and the LCD screen. The LCD screen offers a display area of 640x480 pixels, but the displayed frames occupy only 256x240 pixels. In collaboration with a timing signal generator, the LCD FSM ensures that the 256x240 pixel frame appears in the center of the display.

A timing signal generator, also in programmable logic, synthesizes HSYNC and VSYNC synchronization signals for the display, based on a 12 MHz clock and counters for each dimension. HSYNC and VSYNC scan the display in raster order, from left to right and top to bottom. The LCD FSM reads these counters to track the raster scanning of the LCD display.

This FSM operates similarly to the prior FSMs, with the addition of one additional state. The FSM begins in the *idle* state, when the inactive top rows of the display are being scanned. As soon as the timing signal generator reaches the intended location of the upper-left hand corner of the image, the LCD FSM enters the *lineDrive* state. Data is driven from the frame buffer's read port onto the LCD data lines. The port's address counter is also incremented. 256 pixels later, the FSM enters the *lineWait* state, where it remains inactive until the scanner counts to the intended left edge of the image on the next line, or returns to the *idle* state if the last line in the frame was sent to the display. Figure 3.6 tracks the state transitions of the LCD FSM as a function of the screen location.

Blanking of the inactive regions of the LCD is achieved by setting the frame buffer address to 0x0000 while the FSM is the *idle* or *lineWait* states. This location is always loaded with the zero byte 0x00, which displays as black on the LCD.

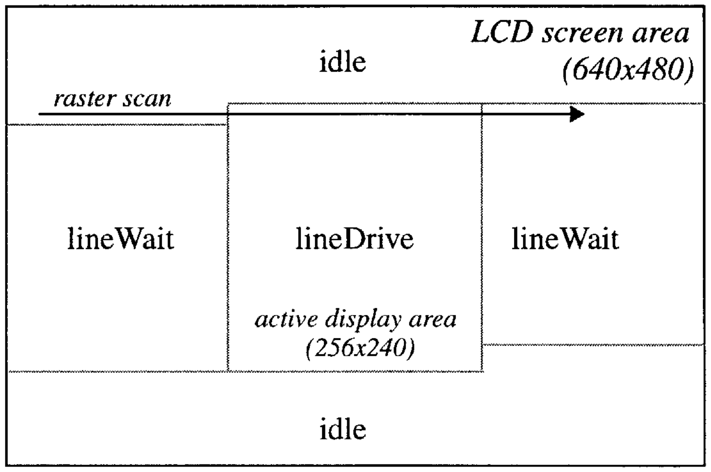


Figure 3.6: States for the LCD FSM for each pixel on the LCD

3.2.6 The Compress State Machine

The Compress FSM controls the transfer of data between the DCT and IDCT devices. It is so named because a quantization and compression unit typically follows the DCT in a compression scheme such as MPEG.

As discussed in section 2.4, the DCT processor outputs data in both a 12-bit parallel output and a compressed serial output. The parallel output is directly compatible with the 12-bit parallel input of the IDCT. However, no corresponding serial input exists for the IDCT. The serial data must be converted into parallel form. This is the primary function of the Compress FSM.

Based on a user input, either the serial or parallel DCT coefficient values are routed to the IDCT. If the parallel source is selected, the FSM is inactive, and the DCT's SBOUT block synchronization signal and the parallel coefficient outputs are routed directly to the corresponding IDCT inputs with a one-cycle delay. This delay reduces the critical path length in case further combinational processing is added at a later time

If the serial source is selected, the Compress FSM works to convert and expand the serial data stream to a parallel form. As the serial data arrive continuously, the solution is conceptually similar to the "ping-pong" buffering scheme used for data format conversion throughout this work. However, since valuable Ultra37256 macrocells are being used for storage and no data rate conversion is necessary, this section is optimized for minimal memory use.

Rather than a state diagram, the operation of the serial-to-parallel converter is shown in a timeline form in table 3.2. Over the course of 64 clock cycles, t = 0 to t = 63, eight 8-bit coefficients are read from the DCT's serial output. They are read in the order D7-D0. The IDCT, however, requires that they appear in the order D0-D7. Furthermore, the coefficients must be placed in certain positions in a 64-coefficient block.

Time (t = 0 is start of block)	Coefficients received from DCT (serial bit stream)	Coefficients sent to IDCT (parallel output)
t = 0...7	D7	D0, D1, D2 (prev. block)
t = 8...15	D6	D3, D4, D5 (prev. block)
t = 16...23	D5	D6, D7 (prev. block)
t = 24...31	D4	
t = 32...39	D3	
t = 40...47	D2	
t = 48...55	D1	
t = 56...63	D0	
t = 64...71	D7 (next block)	D0, D1, D2
t = 72...79	D6 (next block)	D3, D4, D5
t = 80...87	D5 (next block)	D6, D7

Table 3.2: Timetable for serial-to-parallel conversion.

From the table, it is clear that more than 64 bits of storage are necessary. Storing D7-D0 from one block is not sufficient, for the next block immediately follows the current block. Without additional memory, D7 and D6 from the next block would overwrite the current D7 and D6 before they could be sent to the IDCT. D5-D0, however, could be overwritten safely by the next block, for they will have been sent to the IDCT before the next D5-D0 arrive from the DCT. Hence, 80 bits is the minimum amount of storage necessary for this serial-to-parallel converter.

3.3 System Initialization

3.3.1 The Reset State Machine

The Reset state machine is responsible for coordinating a complete system reset. The FSM is activated when the user depresses a button on the circuit board. The Reset FSM then asserts the proper RESET signals to other components and FSMs, manage the initialization of the DCT and Bt829A devices, and signal the other FSMs once the system reset is complete.

When the system's reset button is depressed, the Reset state machine transitions from an *idle* state to a *pause* state, where it remains until the reset button is released. During this and all following states, RESET signals are asserted for the Bt829A, DCT, IDCT, and all state machines.

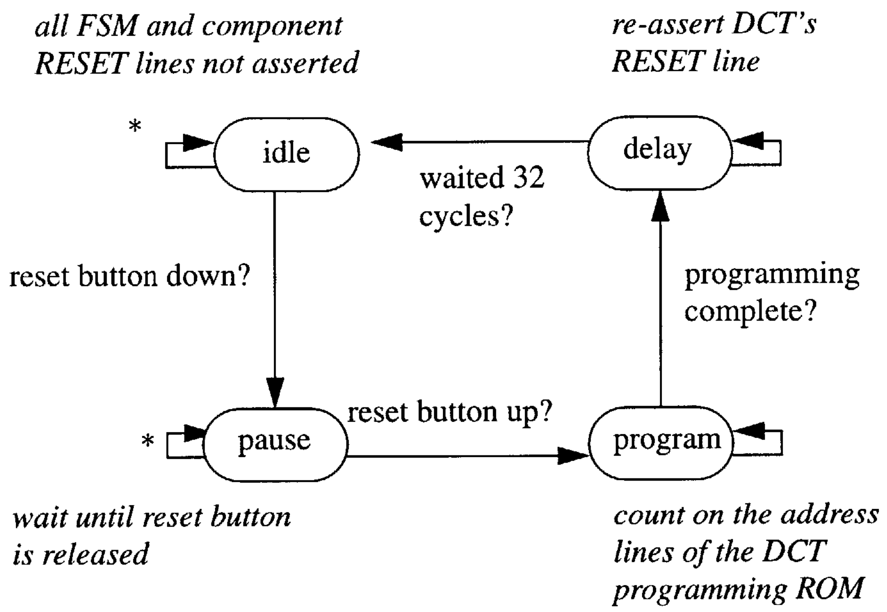

all FSM and component RESET lines not asserted
re-assert DCT's RESET line
*
idle
delay
waited 32 cycles?
reset button down?
programming complete?
reset button up?
*
pause
program
wait until reset button is released
count on the address lines of the DCT programming ROM

Figure 3.7: State transitions for the Reset FSM.

Once the reset button is released by the user, the Reset FSM enters a *program* state. The RESET signal for the DCT (and only the DCT) is released while the DCT is programmed. Once programming is complete, the FSM enters a second *delay* state, where the RESET signal for the DCT is re-asserted. After 32 cycles, the Reset FSM reverts to its *idle* state and all RESET signals are released. The components and FSMs that control the system are now in a known state and are prepared for operation.

During the *program* state, the Reset FSM interacts with an EEPROM which stores an initialization program for the DCT. The Reset FSM addresses the EEPROM from address 0 and increments the address once per clock cycle. Since some of the EEPROM output pins are connected to the DCT's programming lines, this process sends the stored program to the DCT. At the end of the program stored in EEPROM, the high output bit changes state, notifying the Reset FSM that it can leave the *program* state and proceed to the *delay* state.

3.3.2 The I2C State Machine

The I2C state machine is responsible for programming the Bt829A after a system reset has completed. Unlike the DCT, which can be programmed through an EEPROM, the Bt829A requires the programming unit to process acknowledgments. The Bt829A is programmed through a serial I^2C bus interface, which uses a two-way, send-and-acknowledge protocol. A full FSM is required to implement the I^2C bus protocol.

When the I2C FSM is reset, it delivers a program on the I^2C bus connecting the FSM to the Bt829A programming lines. The FSM is capable of handling re-send requests from the Bt829A. As the I^2C bus protocol is fairly complex and the I2C FSM does not interact with any other elements in the video datapath, its detailed behavior is not discussed here. Code for the I2C FSM, as well as the state machines above, is available in appendix B. The I^2C bus protocol is discussed in depth in [8].

3.4 Summary

With the addition of control logic to the video datapath, the complete system can now be conceptualized as in Figure 3.8. This more detailed diagram shows the explicit flow of both control and data through the system.

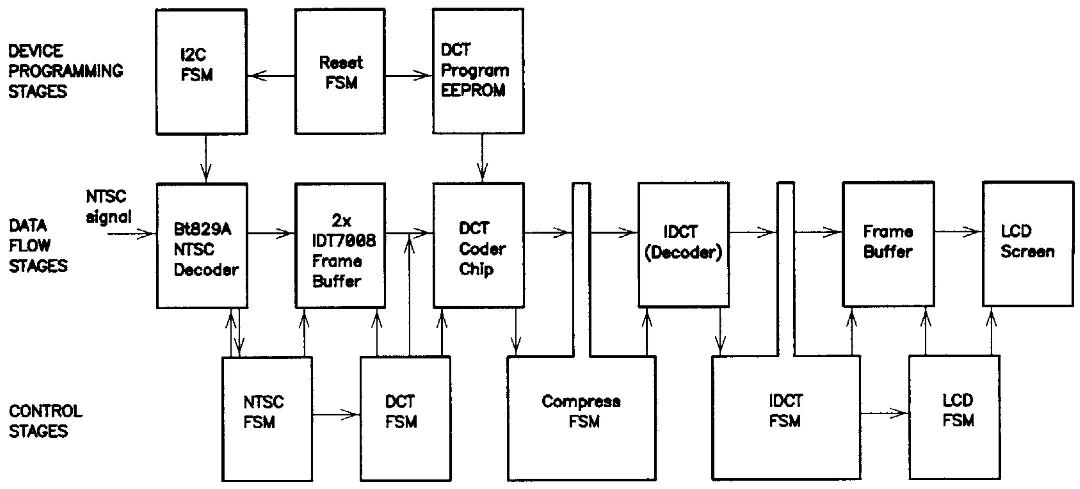


Figure 3.8: Summary of control and data flow through the system.

3.5 Design Lessons: Growth and Change

Systems should be designed to accommodate growth and change. A system that supports change can be debugged and tweaked quickly. A system that supports growth can withstand the addition of significant new features without the need for a complete redesign—in this case, the redesign and re-fabrication of the printed circuit board. It is especially critical that the control logic, which underlies most of the system's functionality, be designed for growth and change.

The Ultra37256 devices were chosen especially for their support of growth and change. On the software side, the Warp development environment supports VHDL, a high-level logic description language. Major changes to algorithms and state machines could be made by rewriting just a few lines of code and recompiling the VHDL files. Just as computer programs written in a high-level language are easier to modify than assembly-language programs, logic described by VHDL is easier to modify than lower-level descriptions or hardware.

Hardware features of the Ultra37256 also support a flexible, expandable design. The in-system reprogramming feature of the Ultra37256 devices allows recompiled logic to be downloaded to the CPLDs in under a minute. Pin assignments for signals can be changed even after the devices have been soldered onto the board, a boon for debugging...or for miswired signals.

The use of two Ultra37256 devices instead of one is a deliberate design decision, rather than a necessity. With some optimization and off-chip, hardware realization of simple logic functions such as counters, one highly-utilized CPLD would have been sufficient. Instead, two under-utilized CPLDs were chosen for the design. There were two reasons for adding the complexity of a second programmable logic device. First, logic realized on a CPLD is more flexible than logic realized by, say, a group of 7400-series devices. Second, leaving the devices under-utilized allows room for the addition of significant amounts of new logic. In this design, the two CPLDs were 46% and 57% utilized.

Implementing the control logic with a careful eye toward growth and change proved invaluable on one particular occasion. After the entire system had been almost completely designed, it was realized that one component—an I^2C bus driver chip—was unavailable at the last minute. The solution was an emulation of this device in programmable logic through the I2C state machine discussed in section 3.3.2 above. To the author's surprise, this reasonably sophisticated state machine was coded in VHDL in a single day, and within a minute of compilation, it was downloaded onto the board. This last-minute modification could not have been possible without high-level logic descriptions, rapid device reprogrammability, and enough free space on a device for major additions.

Chapter 4

Implementation

After the video datapath was designed and the control logic described, the system was realized on a printed circuit board. This board was designed with careful attention to debugging and testing, and the board was functional soon after fabrication. This system successfully verified the functionality of the DCT chipset.

4.1 Printed Circuit Board

The video data path and control logic (with the exception of the LCD screen itself) were placed on a printed circuit board. The PCB was designed with a suite of tools from Accel Technologies. A full circuit schematic was entered graphically into Accel Schematic, and components were placed and signal lines routed with Accel PCB. Since portability is a worthwhile goal for this demonstration system, the board is designed for minimum size. The result is a 6.5" x 7" circuit board with a dense arrangement of components. An unpopulated board is shown in Figure 4.1.

The board is composed of six layers. The four outermost layers are signal layers, and the two innermost layers are the power and ground planes. With the sheer number of power and ground connections required, the two power planes are a necessity. The planes also provide some degree of bypassing. Since the Bt829A requires clean analog power and ground connections for its analog section, the planes are notched at one corner of the board to provide some isolation from digital noise. The analog section of the decoder, which is not coincidentally concentrated on one corner of the chip, is placed within the notched "sub-plane." The DCT and IDCT processors require a separate low-voltage power

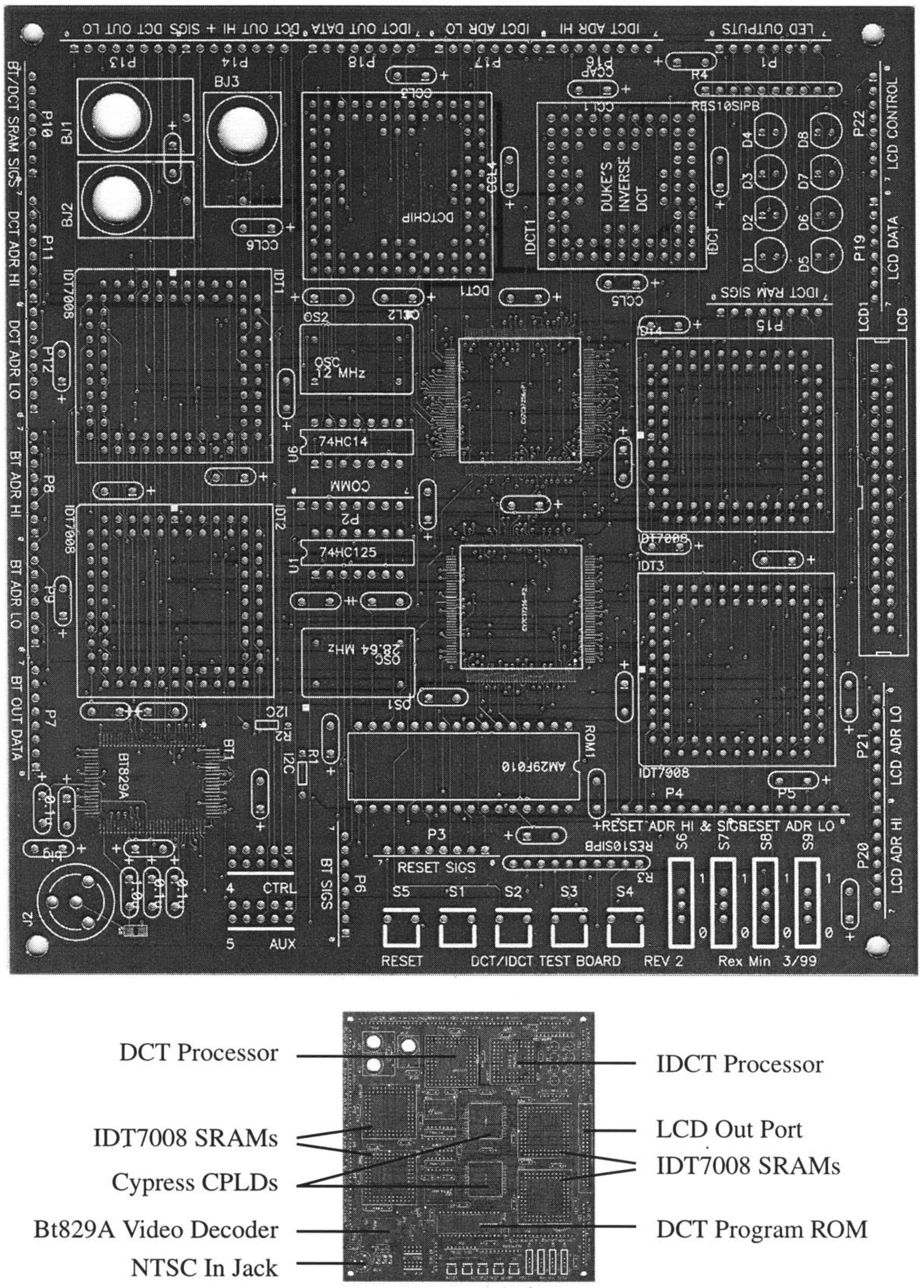


Figure 4.1: The printed circuit board.

supply for their DCT and IDCT cores. This power is provided on wide traces on an internal signal layer. These features can be seen 4.2 and 4.3.

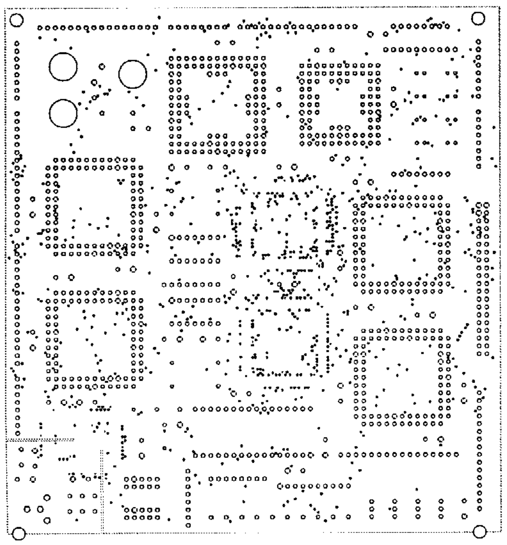


Figure 4.2: Power plane with isolation for the analog section of the Bt829A.

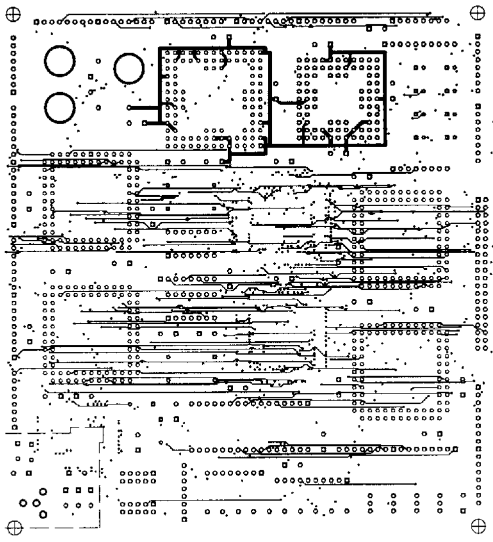


Figure 4.3: Signal layer 2. Wide trace is the low-voltage power supply.

Signal traces are 6 mils in width. This is the widest trace supported by PCB's automatic router for a 12.5 mil routing grid. A 12.5 mil or narrower grid is required to route signals from the closely-spaced surface mount pads of the CPLDs. A narrower routing grid or narrower traces would decrease clearances on the board, resulting in potentially lower reliability and more challenging fabrication.

4.2 Debugging

Facilitating testing and debugging is a goal that was kept in mind throughout the design of both the PCB and the control logic. The result was well worth the design effort; the entire system was debugged and functional in only three days.

The PCB is designed with several on-board debugging aids. Most information-bearing signals, such as RAM addresses, control signals, and video data between each stage of the system, are made available on header pins located along the outer edges of the PCB. This arrangement of over 150 pins greatly expedited the observation of signals on an oscilloscope. With so many signals available on the headers, probing individual pins or vias was almost never necessary. Probing vias is impractical on a board with a 6 mil trace width.

A bank of eight switches and eight light-emitting diodes on the board facilitate debugging. Four of each are connected to each CPLD. The switches allow a variety of debugging routines and test vectors to be stored in the VHDL code and to be called at the touch of a button. The LEDs are convenient for observing the system's general condition and state.

The control logic is also designed with debugging in mind. For example, hidden state, such as FSM states and inter-FSM communication signals within a single CPLD, are routed onto I/O pins on the CPLDs. Information such as FSM state transitions are very useful for locating problems; exposing them reduces debugging time. All control logic was simulated extensively using the CPLD simulator in the Warp development environment. Knowing the control logic's simulated response to test vectors reduced the types of bugs that could be expected in the actual system.

4.3 Performance

The system was operational soon after fabrication. Though the system worked reliably, frame latencies were higher than expected. The DCT chipset performed flawlessly with real-world video input.

4.3.1 The System

Figure 4.4 shows the system in operation. Image quality is good; noise on the power supplies did not seem to hinder the Bt829A's digitization circuitry. Every NTSC device that has been with the system—three different cameras and a VCR—yielded a clean

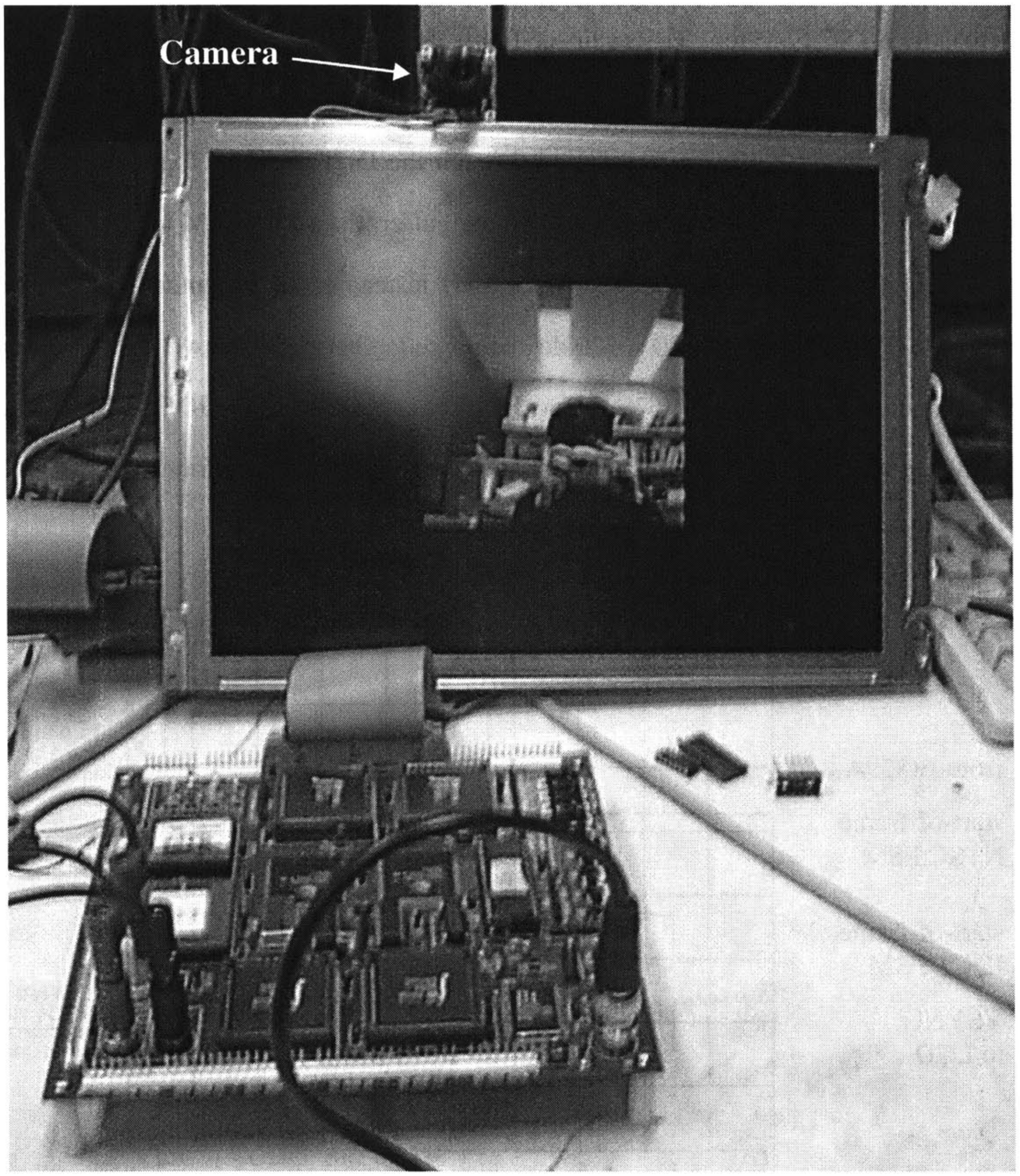


Figure 4.4: The working system. The photographer is displayed.

image. The datapath functions as designed; a full 30 frames per second of video are supported with no dropped frames.

Figure 4.5 shows the flow of the frame-start signal through the video datapath as observed by a digitizing oscilloscope. The Bt829A generates a 30 Hz waveform to signal the beginnings of each even and odd field. On every even field (FIELD driven to logical zero), the NTSC FSM toggles its start-of-frame line to effect the "ping-pong" swapping of static RAM memories and to notify the next FSM in the datapath that a new frame can be read. Once the entire frame has passed through the DCT and IDCT processors and has been completely loaded into the second frame buffer, the IDCT FSM toggles its start-of-frame signal. Since processing occurs at 3 MHz instead of the minimum 2 MHz, there is indeed a (2 MHz/3 MHz)*180 = 120 degree phase lag between the two start-of-block signals. Finally, generation of VSYNC for the LCD is coupled to the IDCT FSM's start-of-block signal. Hence, about 1+2/3 frames of latency separate a new frame (VSYNC) at the Bt829A and the corresponding VSYNC at the LCD.

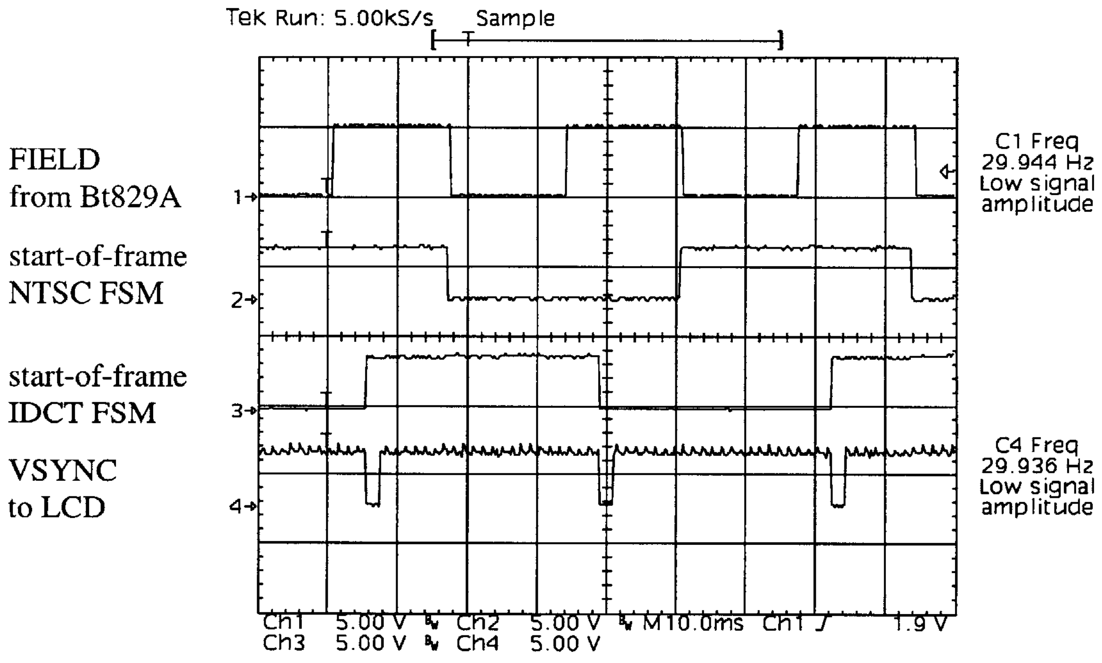


Figure 4.5: Control signal propagation through the video datapath.

Figure 4.6 provides a closer look at control signal propagation for the DCT and IDCT processors. As soon as a frame-start signal is received from the NTSC FSM, the DCT FSM begins to send SBIN block-start pulses to the DCT processor every 64 cycles, indi-

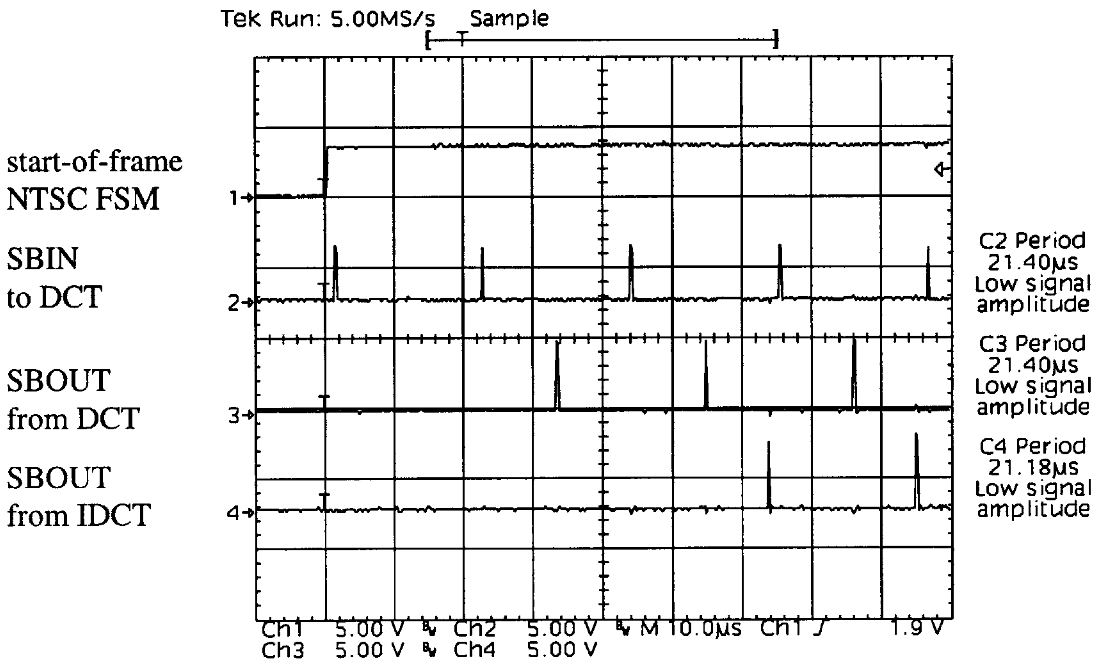


Figure 4.6: Control signal propagation through the DCT chipset.

cating the block boundaries for incoming data. These start-of-block pulses propagate through the DCT processor to its output, continue via control logic to the IDCT's input (not shown), and finally arrive at the output of the IDCT. This final SBOUT signal is used by the IDCT FSM to monitor data flow between the IDCT processor and the second frame buffer.

The latency between input and output was higher than expected. The system is basically designed with a two-frame latency. Each frame buffer introduces about a full frame delay, but the remaining stages contribute almost no latency. The NTSC decoder might store a few lines for linear filtering, and the DCT and IDCT have latencies of under 100 pixels. In a demonstration, however, one spectator estimated a 1/8 second delay—about 4 frames—by oscillating his hands in front of the camera and approximately noting the frequency of oscillation at which the on-screen display was 180 degrees out-of-phase with his hand. This was an unexpected result.

The most likely reason for this delay is additional latency in the camera and the LCD. The camera used for that particular demonstration was CCD-based; these cameras typically sample and buffer a full frame in the digital domain before transmitting the frames in NTSC. The LCD is also likely to contain a frame buffer to support its flexible clocking requirements, bringing the total latency for the system to four frames. Latencies for the camera and LCD could not be confirmed in the documentation, but they are reasonable hypotheses.

4.3.2 The DCT Chipset

The DCT and IDCT processing cores in this system operate at voltages down to 1.2 V, below which increasing numbers of blocks appeared corrupted on the display. At 1.2 V, the two cores require 2.2 mA, for a total power consumption of 2.64 mW. From section 1.3, these devices together consume 4.38 mW + 4.65 mW = 9.03 mW at 14 MHz. Interpolating this figure to 3 MHz yields 1.94 mW, which is not far from the actual power that these processing cores consume in this system.

For comparison, the 5-volt components in the system draw 0.75 A of current for a net power dissipation of 3.75 Watts. The CPLDs, operating in a low-power mode, will consume 120 mA at 3.3 V [11], for 400 mW of power. The IDT7008 static memories used in this system consume 5 mW on standby, but 750 mW when active [9]. With these figures in mind, the low-power performance of the computation-intensive DCT chipset is impressive indeed. One can only imagine the reduction in power consumption that will be brought about by the low-power components of the wireless camera project!

Chapter 5

Conclusion and Future Work

A number of suggestions for future work were offered and discussed during demonstrations of this system. Three of these ideas are presented here. They promise more functionality, better performance, and an enhanced user interface.

Currently, no data quantization or compression is performed between the DCT and IDCT processors. However, data is passed through a CPLD between these stages, and there are many macrocells available in the CPLD further computation. These macrocells can realize a quantizer for the DCT coefficients. The DCT's quantization scheme for its serial output is a simple one that could certainly be improved upon. Potentially better schemes can be tested in programmable logic.

The latency of the system can be reduced. Latency, while not a major concern in the design process, turned out to be quite noticeable when a video camera was used for video input and the observer could see himself on the LCD. Furthermore, the camera and LCD both turned out to have higher latencies than expected (see section 4.3). The latency of the system could be reduced by buffering less than a full frame of data at each buffer while ensuring that no frames are dropped due to the stricter timing requirements.

User interface enhancements are another possibility. Currently, only 20% of the LCD's screen area is utilized. Additional information could be displayed on the LCD alongside the image: a title banner, the system's status and mode of operation, on-screen controls, and so forth. The unused switches and buttons on the board could provide access to parameters that are currently fixed, such as on-screen brightness and contrast.

It is worth noting that most of these improvements can be accommodated without redesign and refabrication of the circuit board. As this system was designed from the start for expandability and growth, significant revisions can be made with cleverness and code, rather than a soldering iron and PCB design tool.

This work is a demonstration of a low-power video coding chipset which requires a challenging, real-time system design with video input, processing, and output capabilities. These capabilities are attained by combining the low-power chipset with carefully chosen, off-the-shelf parts, and arranging them into a seven-stage video datapath. The real-time constraints are met with programmable control logic, and the constraints themselves are relaxed through the judicious use of data buffering.

Good engineering design methodologies contributed heavily to the success of this design. Modularity and abstraction are two fundamental principles. Examples of a modular design include components which serve as "black-boxes" of functionality, the isolation of conceptually different parts of the system by buffering, and the use of many small FSMs in the control logic. Successful designs also facilitate debugging, growth, and change. These ideas are evident in the implementation of in-system reprogrammable control logic, larger-than-necessary CPLDs, and header pins placed around the board.

As the first system-level demonstration of the low-power DCT chipset and the wireless camera's datapath, this system has provided a great deal of practical data and feedback. The functionality of the DCT and IDCT processors, which had not previously been tested with full-motion video, were thoroughly verified. As additional components for the wireless camera project are completed, the promise of a low-power wireless video system draws nearer.

Most of all, the greatest beneficiary of this work is the author. This work encompassed video processing, real-time control, VHDL logic synthesis, and printed circuit board design in a full-year design challenge. The background and methodologies gained through the design and implementation of a complete system are valuable experiences for every engineer.

References

[1] Kodak Corporation, Product label for Kodak HR6 NiMH rechargeable battery.

[2] Intel Corporation, Mobile Pentium Processor with MMX^{TM} Technology on .25 micron. http://developer.intel.com/design/mobile/datashts/243468.htm, January 1998.

[3] T. Xanthopoulos, *Low Power Data-Dependent Transform Video and Still Image Coding,* PhD thesis, Massachusetts Institute of Technology, May 1999.

[4] A. Chandrakasan, H.-S. Lee, C.G.Sodini, and T.Barber, Ultra Low Power Wireless Sensor Project." *Microsystems Technology Laboratories Annual Report*, p.29, May 1999.

[5] International Organization for Standardisation ISO, "Generic coding of moving pictures and associated audio, Reccomendation H.262," ISO/IEC 13818-2 Draft International Standard, 1994.

[6] Didier LeGall, "MPEG: a video compression standard for multimedia applications," *Communications of the ACM*, 34(4):46-58, April 1991.

[7] Rockwell Semiconductor Systems, "Bt829A/827A VideoStream II Decoders," Preliminary product datasheet, November, 1997.

[8] Rockwell Semiconductor Systems, *I^2C-Bus Reference Guide*. June 1997.

[9] Integrated Device Technology, "IDT7008S/L High-Speed Dual-Port Static RAM," Preliminary product datasheet, June 1998.

[10] H. Abelson, G. Sussman, and J. Sussman, *Structure and Interpretation of Computer Programs,* MIT Press, 1996.

[11] Cypress Semiconductor Corporation, "Ultra37256 UltraLogic^{TM} 256-Macrocell ISR^{TM} CPLD," Preliminary product datasheet, revised March 1998.

[12] Zubair Talib, "A Real-Time 256x256 Pixel Parallel Image Processing System," Master's thesis, Massachusetts Institute of Technology, December 1998.

[13] Keith Fife, "A Stereo Vision System with Automatic Brightness Adaptation," Master's thesis, Massachusetts Institute of Technology, May 1999.

Appendix A

Circuit Schematics

This appendix consists of the complete schematics of the printed circuit board and its components. All graphics were generated with Accel Schematic.

The remainder of this page is intentionally left blank. The first schematic plate begins on the following page.

A.1 The Video Decoder

Figure A.1 below shows the Bt829A video digitizer, along with supporting analog circuitry, the 28.636 MHz clock oscillator, and a section of programmable logic which corresponds to a portion of the NTSC state machine.

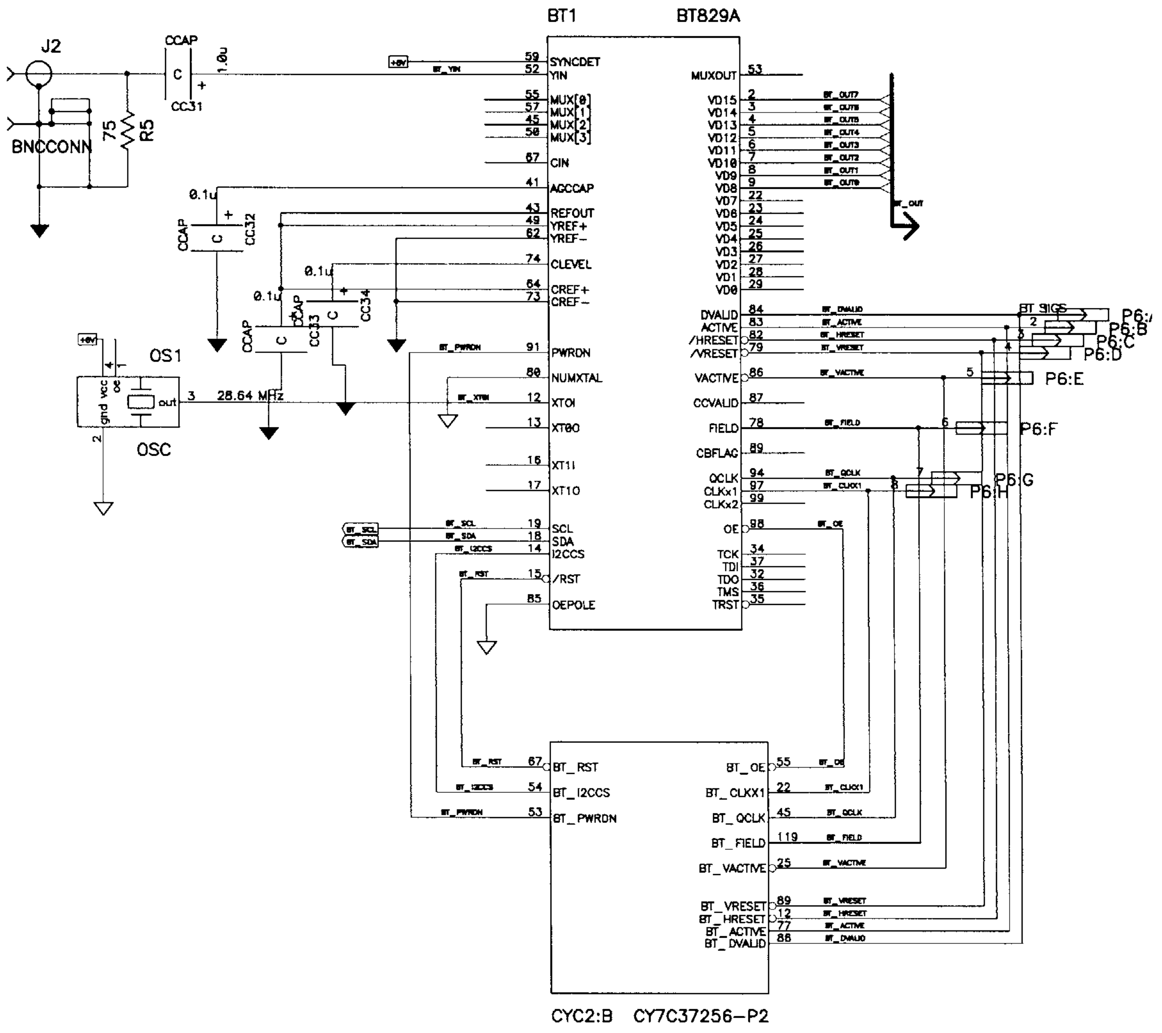


Figure A.1: Video datapath, Bt829A video decoder.

A.2 The First Frame Buffer

Figure A.2 below shows the pair of IDT7008 devices that make up the first frame buffer, and sections of programmable logic which correspond to portions of the NTSC and DCT state machines. The blocks to the upper-left and upper-right are header pins which carry signals to the edges of the board for easy observation.

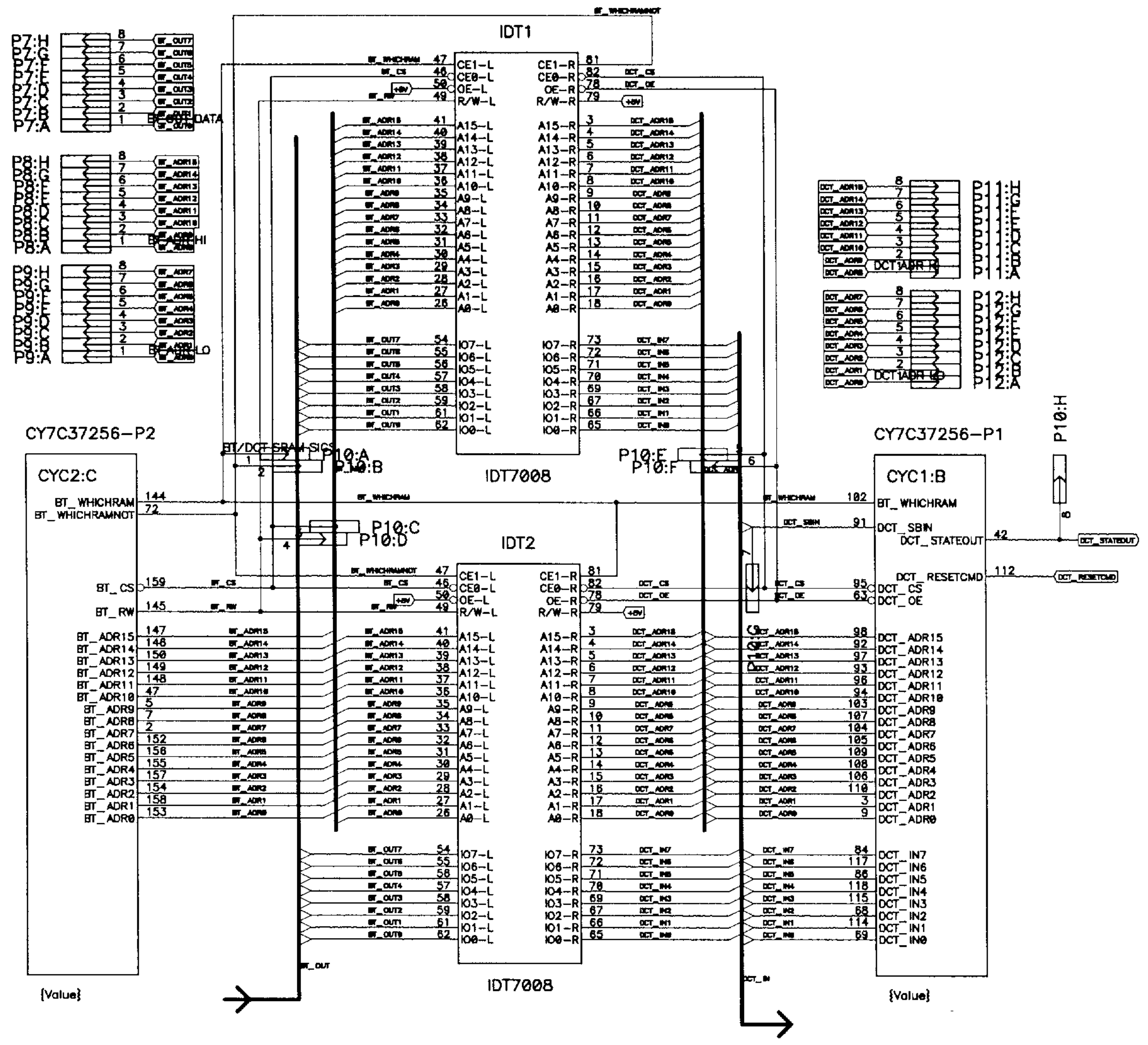


Figure A.2: Video datapath, first frame buffer.

A.3 The DCT and IDCT Processors

Figure A.3 below shows the low-power DCT and IDCT processors, and a section of programmable logic which corresponds to the COMPRESS state machine.

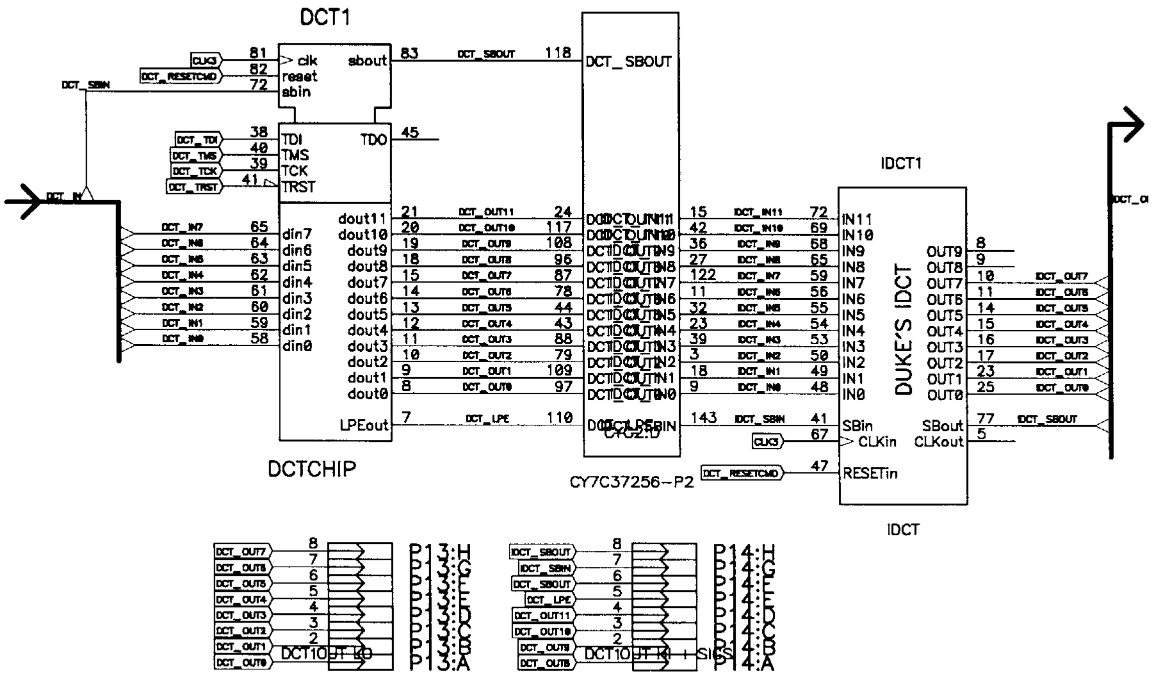


Figure A.3: Video datapath, DCT and IDCT processors.

A.4 The Second Frame Buffer

Figure A.4 below shows the pair of IDT7008 devices that make up the second frame buffer, and sections of programmable logic which correspond to portions of the IDCT and LCD state machines.

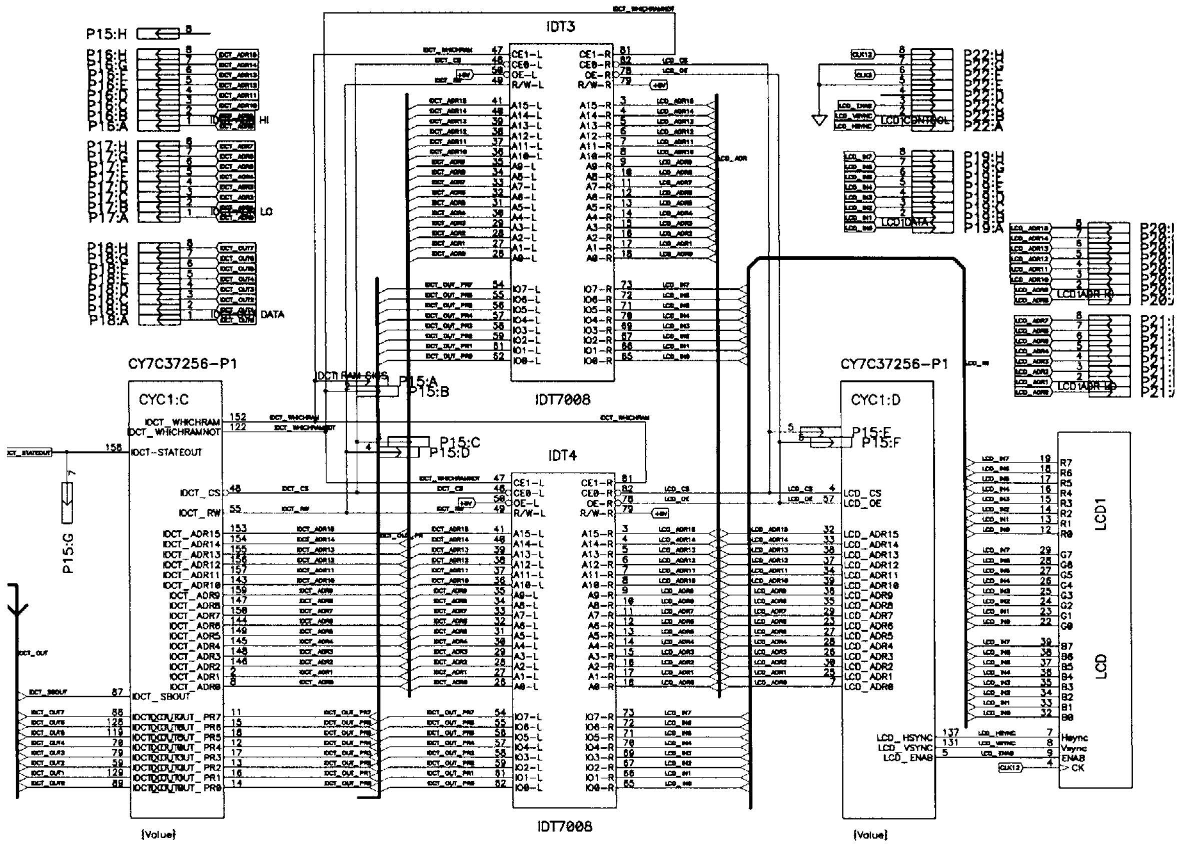


Figure A.4: Video datapath, second frame buffer and LCD inerface.

A.5 Miscellaneous Control

Figure A.5 below shows a variety of supporting structures. At the upper-right is the ROM which programs the DCT on startup, along with a section of programmable logic. Below are the two top-level interfaces of each CPLD on the board, with switch inputs and LED outputs. Clock generation and buffering are represented along the bottom.

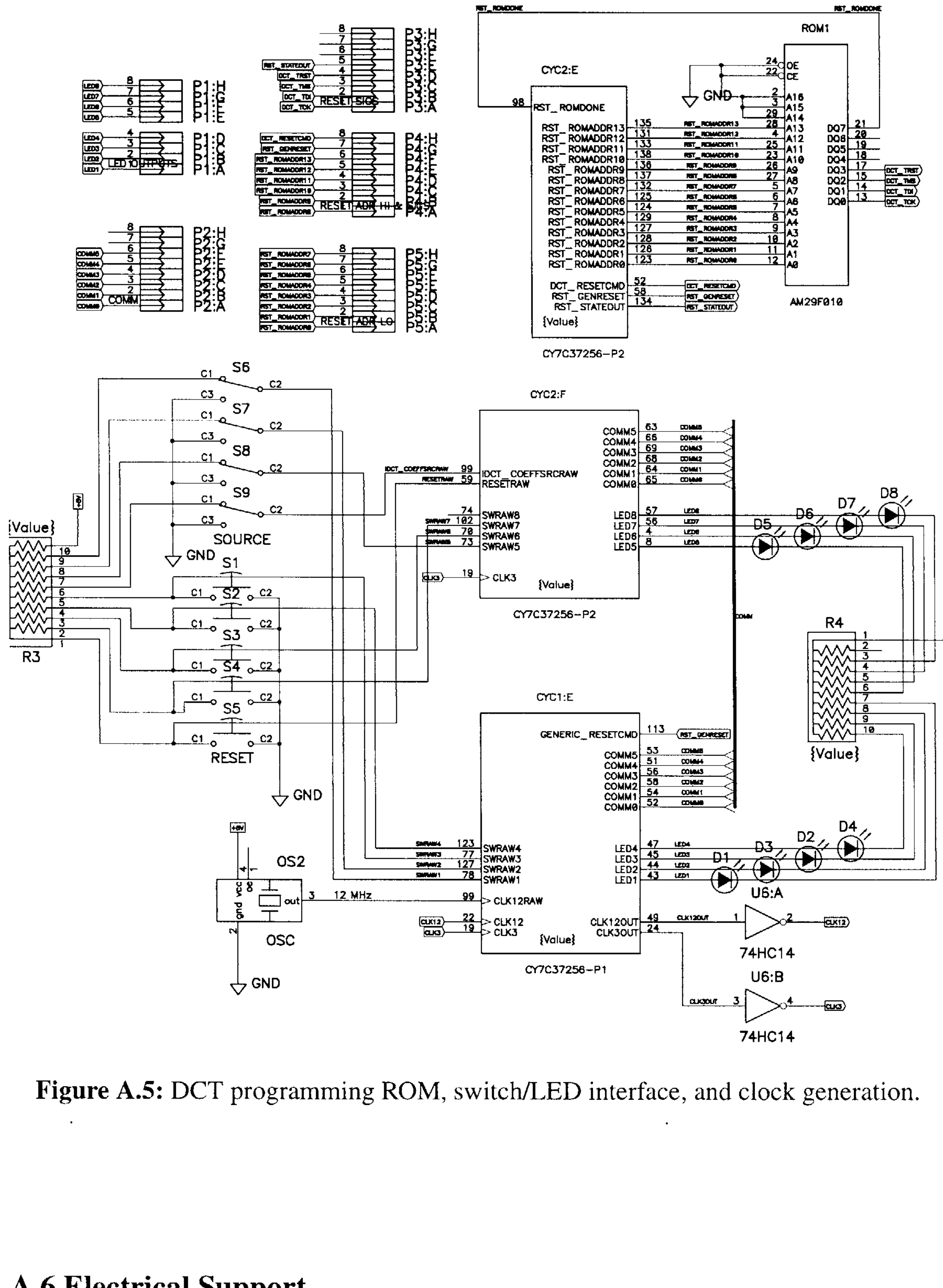


Figure A.5: DCT programming ROM, switch/LED interface, and clock generation.

A.6 Electrical Support

Figure A.6 below shows the remaining components. On the upper-right are the connections for the in-system reprogramming ports of the CPLDs, where new logic is downloaded to the devices. To their left is the wired-AND structure for the I2C bus which connects to the Bt829A. Bypass capacitors, banana jacks for power, and mounting holes complete the board.

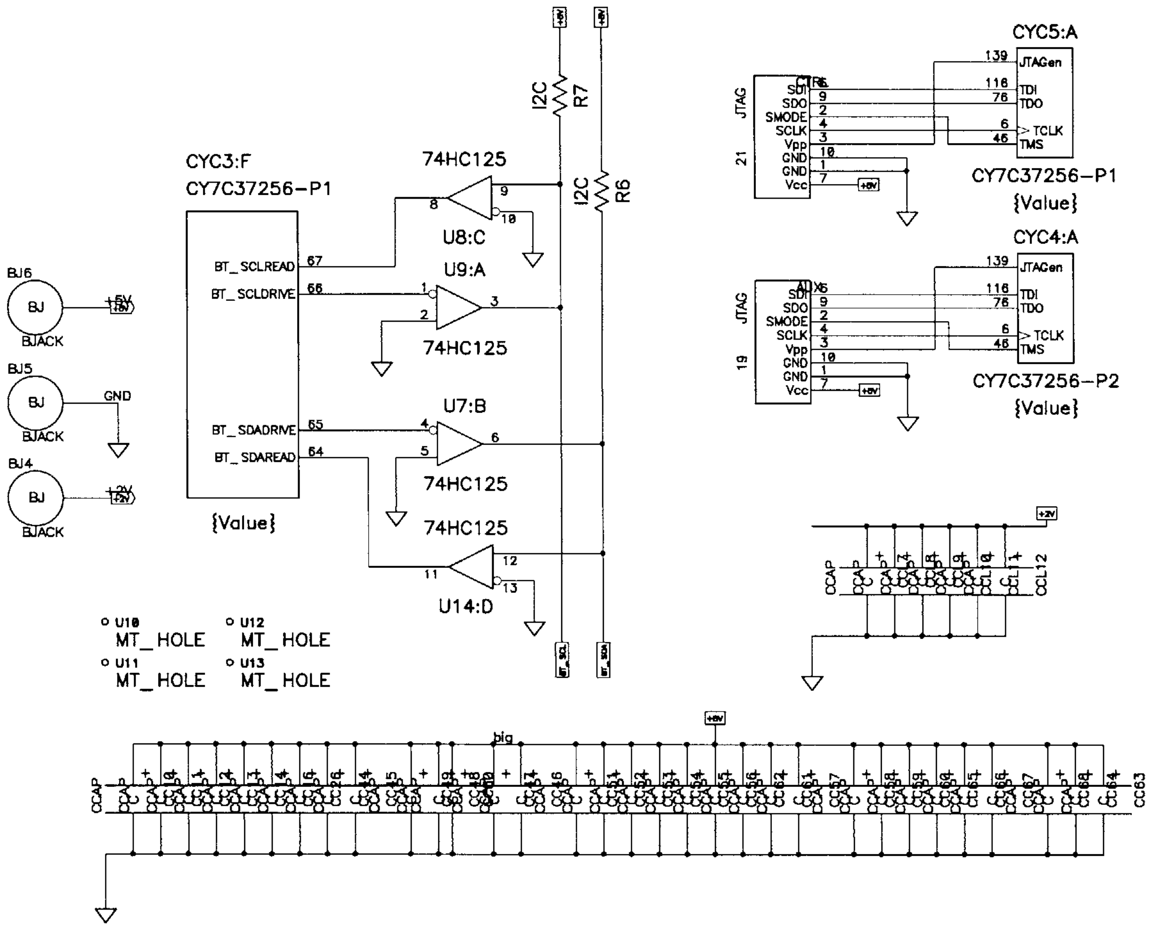


Figure A.6: I2C wired-AND, CPLD ISR pins, and circuit board miscellany.

Appendix B

VHDL Implementation of the Control Logic

Control logic for this system is written in VHDL. This work requires nine VHDL files whose code is distributed over two Cypress Ultra37256 programmable logic devices.

B.1 The NTSC State Machine

The following file *ntsc.vhd* generates the NTSC state machine, which controls the interface between the Bt829A video decoder and the first frame buffer.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE work.std_arith.ALL;


entity ntsc_sram_drive is port (
      -- SRAM left side addresses
      ntscAdr: buffer std_logic_vector(15 downto 0);
      whichRam: buffer std_ulogic;
      whichRamNot: buffer std_ulogic;
      cs: out std_ulogic;
      rw: out std_ulogic;

      -- Brooktree 829A
      dvalid, active, hreset, vreset, vactive: in std_ulogic;
      field, qclk, clkx1: in std_ulogic;
      oe, rst, i2ccs, pwrdn: out std_ulogic;

      -- misc. inputs
      reset: in std_ulogic
      );
end entity ntsc_sram_drive;


architecture ntsc_arch of ntsc_sram_drive is
type States is (waiting, loading);
signal state, nextState: States;
signal ntscAdrCount: std_logic_vector(15 downto 0);
signal aboutToLoad: std_ulogic; -- pulses high on wait->load state transition

begin

-- placeholder for unused input signals
```

```
stateMachine: process (state, nextState, vreset, field, ntscAdrCount)
begin
      case state is
      when waiting =>
            -- wait for start of new (even) field
            if (vReset = '0') and (active = '0') then
                  nextState <= loading;
                  aboutToLoad <= '1';
            else
                  nextState <= waiting;
                  aboutToLoad <= '0';
            end if;

      when loading =>
            aboutToLoad <= '0';
            -- reset after receiving 240 lines
            if (ntscAdrCount = "1110111111111111") then
                  nextState <= waiting;
            else
                  nextState <= loading;
            end if;
      end case;

end process stateMachine;


cs <= '0' when (dvalid = '1') and (active = '1') and (clkx1 = '0')
      else '1';
ntscAdr <= ntscAdrCount;-- no fancy SRAM addressing required here
whichRamNot <= not whichRam;

i2ccs <= '0'; -- misc. Bt829A control signals
pwrdn <= '0';
rw <= '0';
oe <= '0';

clockUpdate: process (clkx1, reset)
begin
      if (reset = '1') then
            state <= waiting;
            rst <= '0';
            whichRam <= '0';-- not necessary; next FSM only reads transitions
      elsif (clkx1'event) and (clkx1 = '1') then
            state <= nextState;
            rst <= '1';

            if (aboutToLoad = '1') then
                  whichRam <= not whichRam;
            end if;

            if (state = waiting) then
                  ntscAdrCount <= (others => '0');
            elsif (dvalid = '1') and (active = '1') then
                  ntscAdrCount <= ntscAdrCount + 1;
            end if;
      end if;
end process clockUpdate;

end architecture ntsc_arch;


LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE work.std_arith.ALL;
```

```
package ntsc_pack is
component ntsc_sram_drive port (
      -- SRAM left side addresses
      ntscAdr: buffer std_logic_vector(15 downto 0);
      whichRam: buffer std_ulogic;
      whichRamNot: buffer std_ulogic;
      cs: out std_ulogic;
      rw: out std_ulogic;

      -- Brooktree 829A
      dvalid, active, hreset, vreset, vactive: in std_ulogic;
      field, qclk, clkx1: in std_ulogic;
      oe, rst, i2ccs, pwrdn: out std_ulogic;

      -- misc. inputs
      reset: in std_ulogic
      );
end component;
end package ntsc_pack;
```

B.2 The DCT State Machine

The following file *dct.vhd* generates the DCT state machine, which controls the interface between the first frame buffer and the DCT processor.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE work.std_arith.ALL;


entity dct_sram_drive is port (
      -- SRAM right side
      dctAdr: buffer std_logic_vector(15 downto 0);
      whichRam: in std_ulogic;
      data: in std_logic_vector(7 downto 0);
      cs, oe: out std_ulogic;

      -- DCT chip
      blockStart: out std_ulogic;

      -- misc. inputs
      clk3: in std_ulogic;
      reset: in std_ulogic;

      -- misc. outputs
      stateOut: buffer std_ulogic
      );
end entity dct_sram_drive;


architecture dct_arch of dct_sram_drive is
type States is (waiting, loading);
signal state, nextState: States;
signal dctAdrCount: std_logic_vector(15 downto 0);
signal syncReg, whichRamSync, whichRamOld: std_ulogic;
signal aboutToLoad, aboutToWait: std_ulogic;
```

```
begin

stateMachine: process (state, nextState, whichRamSync, whichRamOld, dctAdrCount)
begin
      case state is
      when waiting =>
            -- wait for transition on synchronized WHICHRAM from NTSC FSM
            aboutToWait <= '0';
            if (whichRamSync = not whichRamOld) then
                  aboutToLoad <= '1';
                  nextState <= loading;
            else
                  aboutToLoad <= '0';
                  nextState <= waiting;
            end if;

      when loading =>
            -- send 256x256 pixels, then go back to waiting state
            aboutToLoad <= '0';
            if (dctAdrCount = "1111111111111111") then
                  aboutToWait <= '1';
                  nextState <= waiting;
            else
                  aboutToWait <= '0';
                  nextState <= loading;
            end if;
      end case;

end process stateMachine;


cs <= '0' when (state = loading)
      else '1';
dctAdr <=dctAdrCount(15 downto 11) &-- perform raster-to-block mapping
            dctAdrCount(5 downto 3) &
            dctAdrCount(10 downto 6) &
            dctAdrCount(2 downto 0);



stateOut <= '1' when (state = loading) else '0';
oe <= '0';

clockUpdate: process (clk3, whichRamSync, reset)
begin
      if (reset = '1') then
            state <= waiting;
            whichRamOld <= whichRamSync;
      elsif (clk3'event) and (clk3 = '1') then
            -- synchronize incoming whichRam signal (two registers)
            syncReg <= whichRam;
            whichRamSync <= syncReg;

            state <= nextState;

            -- blockStart assigned synchronously to prevent glitches
            if ((state = waiting) and (aboutToLoad = '1')) or
               ((state = loading) and (aboutToWait = '0') and (dctAdrCount(5 downto 0) =
"111111")) then
                  blockStart <= '1';
            else
                  blockStart <= '0';
            end if;

            if (state = loading) then
```

```
                whichRamOld <= whichRamSync;
            end if;

            if (state = waiting) then
                dctAdrCount <= (others => '0');
            else
                dctAdrCount <= dctAdrCount + 1;
            end if;
        end if;
end process clockUpdate;

end architecture dct_arch;


LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE work.std_arith.ALL;


package dct_pack is
component dct_sram_drive port (
        -- SRAM right side
        dctAdr: buffer std_logic_vector(15 downto 0);
        whichRam: in std_ulogic;
        data: in std_logic_vector(7 downto 0);
        cs, oe: out std_ulogic;

        -- DCT chip
        blockStart: out std_ulogic;

        -- misc. inputs
        clk3: in std_ulogic;
        reset: in std_ulogic;

        -- misc. outputs
        stateOut: buffer std_ulogic
        );
end component;
end package dct_pack;
```

B.3 The IDCT State Machine

The following file *idct.vhd* generates the IDCT state machine, which controls the interface between the IDCT processor and the second frame buffer.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE work.std_arith.ALL;


entity idct_sram_drive is port (
        -- SRAM left side
            idctAdr: buffer std_logic_vector(15 downto 0);
            whichRam: buffer std_ulogic;
            whichRamNot: buffer std_ulogic;
            cs, rw: out std_ulogic;
            dataLatch: buffer std_logic_vector(7 downto 0);

            -- IDCT chip
            blockStart: in std_ulogic;
            dataIn: in std_logic_vector(9 downto 0);
            -- misc. inputs
            clk3: in std_ulogic;
            reset: in std_ulogic;

            -- misc. outputs
            stateOut: buffer std_ulogic
      );
end entity idct_sram_drive;




architecture idct_arch of idct_sram_drive is

constant lastIdctAdr: std_logic_vector(15 downto 0) := "1111111111111111";

type States is (waiting, loading);
signal state, nextState: States;
signal idctAdrCount: std_logic_vector(15 downto 0);
signal aboutToLoad, aboutToWait: std_ulogic;
begin

stateMachine: process (state, nextState, idctAdrCount, blockStart)
begin
      case state is
      when waiting =>
            aboutToWait <= '0';
            if (blockStart = '1') then
                  aboutToLoad <= '1';
                  nextState <= loading;
            else
                  aboutToLoad <= '0';
                  nextState <= waiting;
            end if;

      when loading =>
            aboutToLoad <= '0';
            if (idctAdrCount = lastIdctAdr) then
                  aboutToWait <= '1';
                  nextState <= waiting;
            else
                  aboutToWait <= '0';
                  nextState <= loading;
            end if;
      end case;

end process stateMachine;


cs <= clk3 when (state = loading)
      else '1';
rw <= '0';
idctAdr <=idctAdrCount(15 downto 11) &-- perform block-to-raster mapping
            idctAdrCount(5 downto 3) &
            idctAdrCount(10 downto 6) &
            idctAdrCount(2 downto 0);
stateOut <= '1' when (state = loading) else '0';
whichRamNot <= not whichRam;

clockUpdate: process (clk3, reset)
begin
      if (reset = '1') then
            dataLatch <= (others => '0');
            state <= waiting;
      elsif (clk3'event) and (clk3 = '1') then

            state <= nextState;

            -- always assign zero to address zero (for blanking)
            if (idctAdrCount = "0000000000000000") then
                  dataLatch <= (others => '0');
            else

            -- on 4/8 we discover why white pixels turn black and vice versa.
            -- over/under flow!  we implement a hard-limiter here to fix it.

                  if (dataIn(9) = '1') then-- 1xxxxxxxxx = negative number.
                        dataLatch <= "00000000";
                  elsif (dataIn(8) = '1') then-- 01xxxxxxxx = overflow.
                        dataLatch <= "11111111";
                  else                    -- 00xxxxxxxx = within range.
                        dataLatch <= dataIn(7 downto 0);
                  end if;
            end if;

            if (idctAdrCount = lastIdctAdr) then
                  whichRam <= not whichRam;
            end if;

            -- outputBlockStart assigned synchronously to prevent glitches

            if (state = waiting) then
                  idctAdrCount <= (others => '0');
            else
                  idctAdrCount <= idctAdrCount + 1;
            end if;
      end if;
end process clockUpdate;

end architecture idct_arch;


LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE work.std_arith.ALL;


package idct_pack is
component idct_sram_drive port (
      -- SRAM left side
      idctAdr: buffer std_logic_vector(15 downto 0);
      whichRam: buffer std_ulogic;
      whichRamNot: buffer std_ulogic;
      cs, rw: out std_ulogic;
      dataLatch: buffer std_logic_vector(7 downto 0);

      -- IDCT chip
      blockStart: in std_ulogic;
      dataIn: in std_logic_vector(9 downto 0);

      -- misc. inputs
      clk3: in std_ulogic;
      reset: in std_ulogic;

      -- misc. outputs
      stateOut: buffer std_ulogic
      );
end component;
end package idct_pack;
```

B.4 The LCD State Machine

The following file *lcd.vhd* generates the LCD state machine, which controls the interface between the second frame buffer and the LCD and generates timing signals for the LCD.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE work.std_arith.ALL;


entity lcd_sram_drive is port (
      -- ce, rw always enabled in hardware
      -- ENAB on LCD always held low; data starts at C48
      clk12: in std_ulogic;
      reset: in std_ulogic;

      -- SRAM right side
      lcdAdr: buffer std_logic_vector(15 downto 0);-- LCD "column" data address, SRAM A
      whichRam: in std_ulogic;
      cs, oe: out std_ulogic;

      -- LCD screen
      enab: out std_logic;
      hsync, vsync: buffer std_ulogic  -- active low for 640x480; register these to
prevent glitches )
);
end entity lcd_sram_drive;


architecture lcd_arch of lcd_sram_drive is
signal hCount: std_logic_vector(9 downto 0);
signal vCount: std_logic_vector(8 downto 0);
signal next_lcdAdrCount: std_logic_vector(15 downto 0);


constant hDataStart: integer := 320;-- LCD pixel position for first pixel. MUST BE 2 OR
HIGHER.
constant hDataEnd: integer := 575; -- LCD pixel position for last pixel
constant vDataStart: integer := 150;
constant vDataEnd: integer := 389;
constant hMax: integer := 762;  -- number of horizontal pixels, plus 48, plus hsync width
constant vMax: integer := 525;
constant hsyncEnd: integer := 80;-- must be at least 1 less than hMax
constant vsyncEnd: integer := 30;-- must be at least 1 less than vMax
.                                                       .

type vd_states is (idle, lineWait, lineDrive, advanceRow);
signal vd_state, vd_nextState: vd_states;
signal lcdAdrCount: std_logic_vector(15 downto 0);
signal oldWhichRam: std_ulogic;
begin


vid_drive: process (hCount, vCount, vd_state)
      begin
```

```
            case vd_state is
                  when idle =>
                        if (vCount = vDataStart) then
                              vd_nextstate <= lineWait;
                        else
                              vd_nextstate <= idle;
                        end if;

                  when lineWait =>-- count off hDataStart pixels before driving LCD on
this line
                        if (hCount = hDataStart - 1) then-- start blasting data from next
clock!
                              vd_nextstate <= lineDrive;
                        else
                              vd_nextstate <= lineWait;
                        end if;

                  when lineDrive =>
                        if (hCount = hDataEnd) or (hCount = hMax) then-- no more data for
this line?
                              if (vCount = vDataEnd) then-- no more data for this FRAME?
                                    vd_nextstate <= idle;-- immediately jump to reset
state!
                              else              -- (end of LINE but not end of FRAME)
                                    vd_nextstate <= advanceRow;-- done with this row; move
to next one.
                              end if;  -- (vCount = vDataEnd)
                        else  -- more data in this line (default case), so stay here.
                              vd_nextstate <= lineDrive;
                        end if;

                  when advanceRow =>-- identical to lineWait, except this state triggers
an increment
                                                            -- in the high address. Hence, we
stay here only once cycle....
                        if (hCount = hDataStart - 1) then
                              vd_nextstate <= lineDrive;
                        else
                              vd_nextstate <= lineWait;--- ...and then fall back to state
lineWait.
                        end if;

                  when others =>
                              vd_nextstate <= idle;
            end case;

      end process vid_drive;


      cs <= '0';-- for now, always address device (raising CS causes floated pins)
      enab <= '1';-- fix high so that LCD takes clk 48 as pixel 0
      oe <= '0';

reader_addrCount: process (lcdAdrCount, vd_state)
      begin
            case vd_state is
                  when idle =>-- reset counter
                        next_lcdAdrCount <= (others => '0');

                  when lineWait =>-- new scan line. reset low byte address only
                        next_lcdAdrCount(7 downto 0) <= (others => '0');
                        next_lcdAdrCount(15 downto 8) <= lcdAdrCount(15 downto 8);

                  when lineDrive =>-- advance 1 pixel. increment low byte of address
                        next_lcdAdrCount(7 downto 0) <= lcdAdrCount(7 downto 0) + 1;
                        next_lcdAdrCount(15 downto 8) <= lcdAdrCount(15 downto 8);

                  when advanceRow =>-- advance 1 line. increment high byte; reset low byte
of addr
                        next_lcdAdrCount(15 downto 8) <= lcdAdrCount(15 downto 8) + 1;
                        next_lcdAdrCount(7 downto 0) <= (others => '0');

                  when others =>-- should never happen. reset all.
                        next_lcdAdrCount <= (others => '0');
            end case;
      end process reader_addrCount;

-- blank screen when address is 0. also blank last 16 lines.
lcdAdr <= lcdAdrCount when (vd_state = lineDrive)
          else (others => '0');



reader_clock_update: process (clk12, reset)
      begin
            if (reset = '1') then
                  vd_state <= idle;

            elsif (clk12'event and clk12 = '1') then
                  lcdAdrCount <= next_lcdAdrCount;
                  vd_state <= vd_nextstate;
            end if;

      end process reader_clock_update;



-- TIMING SIGNAL GENERATOR for LCD. Generates HSYNC and VSYNC scanning signals.

sig_gen: process (clk12, reset, whichRam)
      begin
            if (reset = '1') then
                  hCount <= (others => '0');
                  vCount <= (others => '0');
                  vsync <= '1';
                  hsync <= '1';
                  oldWhichRam <= whichRam;

            elsif (clk12'event and clk12 = '1') then
                  if (hCount = hMax) then    -- END OF LINE; assert hSync & reset hCount
                        hsync <= '0';-- assert...
                        hCount <= (others => '0');-- reset.

                        if (oldWhichRam = not whichRam) then
                              oldWhichRam <= whichRam;
                              vsync <= '0'; -- assert...
                              vCount <= (others => '0');-- reset.
                        else                -- END OF LINE BUT NOT END OF FRAME
                              if (vCount = vSyncEnd) then
                                    vsync <= '1';  -- deassert vSync after vSyncEnd lines
                              end if;
                              vCount <= vCount + 1;
                        end if;
                  else  -- not end of line, not end of frame ("USUAL" CASE); increment
hCount
                        if (hCount = hsyncEnd) then
                              hsync <= '1';    -- deassert hSync after hSyncEnd pixels
                        end if;
                        hCount <= hCount + 1;
```

```
                end if;    -- [if hCount = hMax]
            end if; -- [if clk12 has rising edge]
      end process sig_gen;

end architecture lcd_arch;


LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE work.std_arith.ALL;


package lcd_pack is
component lcd_sram_drive port (
      -- ce, rw always enabled in hardware
      -- ENAB on LCD always held low; data starts at clock 48 after HSYNC
      clk12: in std_ulogic;
      reset: in std_ulogic;

      -- SRAM right side
      lcdAdr: buffer std_logic_vector(15 downto 0);-- LCD "column" data address, SRAM A
      whichRam: in std_ulogic;
      cs, oe: out std_ulogic;

      -- LCD screen
      enab: out std_logic;
      hsync, vsync: buffer std_ulogic  -- active low for 640x480; register these to
prevent glitches (better technique?)
);
end component;
end package lcd_Pack;
```

B.5 The Compress State Machine

The following file *compress.vhd* generates the Compress state machine, which controls the interface between the DCT and IDCT processors.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE work.std_arith.ALL;

entity compressor is port (
      --- DCT chip
      lpe: in std_ulogic;
      coeffIn: in std_logic_vector(11 downto 0);
      blockStartIn: in std_ulogic;

      --- IDCT chip
      coeffOut: buffer std_logic_vector(11 downto 0);
      blockStartOut: buffer std_ulogic;

      --- other i/o
      reset: in std_logic;
      clk3: in std_ulogic;
      source: in std_ulogic
);
end entity compressor;



architecture comp_arch of compressor is
```

```
constant lowFourBytes: std_logic_vector(3 downto 0) := "1010";

signal intPhase: integer(0 to 63);
signal blockStartBuf1, blockStartBuf2: std_ulogic;
signal serialData: std_logic_vector(63 downto 0);
signal serialOverflow: std_logic_vector(63 downto 48);
signal lpeCoeff, parallelCoeff: std_logic_vector(11 downto 0);
signal sourceSync: std_ulogic;

begin


loadSerialData:process
begin
      wait until (clk3'event) and (clk3 = '1');   -- act on rising edge of clock
      -- the following events happen AFTER the clock rising edge, to be ready
      -- in time for the FOLLOWING edge:

      serialData(intPhase) <= lpe;-- capture incoming bit from DCT
                              -- (if we're waiting in 63, we continuously overwrite)
      if (reset = '1') then-- reset (active-low) asserted?
            intPhase <= 63;-- reset counters to 63. we count DOWN 63 to 0
      elsif (intPhase = 63) and (blockStartIn = '0') and (blockStartBuf1 = '0') then
            -- keep on waiting for a blockStart.
            sourceSync <= source;-- update coef. source only when idle
      else

            intPhase <= intPhase - 1;-- update counters: we count DOWN 63 to 0

            if (intPhase = 32) then-- copy D6..D7 to serialOverflow buffer, or they'd get
smashed
                  serialOverflow(63 downto 48) <= serialData(63 downto 48);
            end if;
      end if;

end process loadSerialData;


outputserialData: process(intPhase, serialOverflow, serialData)
begin
      case intPhase is
      when 63 =>-- send data bit D0 in block 1x1 of 8x8
            lpeCoeff <= serialData(7 downto 0) & lowFourBytes;
      when 62 =>-- send data bit D1 in block 2x1 of 8x8
            lpeCoeff <= serialData(15 downto 8) & lowFourBytes;
      when 61 =>-- send data bit D2 in block 3x1 of 8x8
            lpeCoeff <= serialData(23 downto 16) & lowFourBytes;
      when 55 =>-- send data bit D3 in block 1x2 of 8x8
            lpeCoeff <= serialData(31 downto 24) & lowFourBytes;
      when 54 =>-- send data bit D4 in block 2x2 of 8x8
            lpeCoeff <= serialData(39 downto 32) & lowFourBytes;
      when 53 =>-- send data bit D5 in block 2x3 of 8x8
            lpeCoeff <= serialData(47 downto 40) & lowFourBytes;
      when 47 =>-- send data bit D6 in block 3x1 of 8x8
            lpeCoeff <= serialOverflow(55 downto 48) & lowFourBytes;
      when 46 =>-- send data bit D7 in block 3x2 of 8x8
            lpeCoeff <= serialOverflow(63 downto 56) & lowFourBytes;
      when others =>
            lpeCoeff <= (others => '0');
      end case;

end process outputserialData;
```

```
router: process(clk3, sourceSync)
begin

      if (clk3'event) and (clk3 = '1') then
            if (sourceSync = '0') then
                  coeffOut <= coeffIn;
                  blockStartOut <= blockStartIn;-- load block start delay register

            else
                  coeffOut <= lpeCoeff;
                  if (tempPhase = 63) then
                        blockStartBuf1 <= blockStartIn;
                        blockStartOut <= blockStartBuf1;
                  else
                        blockStartOut <= '0';
                  end if;
            end if;
      end if;
end process router;

end architecture comp_arch;



LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE work.std_arith.ALL;

package comp_pack is
component compressor port (
      --- DCT chip
      lpe: in std_ulogic;
      coeffIn: in std_logic_vector(11 downto 0);
      blockStartIn: in std_ulogic;

      --- IDCT chip
      coeffOut: buffer std_logic_vector(11 downto 0);
      blockStartOut: buffer std_ulogic;

      --- other i/o
      reset: in std_logic;
      clk3: in std_ulogic;
      source: in std_ulogic
);
end component;
end package comp_pack;
```

B.6 The Reset State Machine

The following file *reset2.vhd* generates the Reset state machine, which coordinates a complete system reset and drives the ROM that initializes the DCT processor.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE work.std_arith.ALL;


entity reset2 is port (
      clk3: in std_ulogic;
      resetButton: in std_ulogic;
      romDone: in std_ulogic;

      ntsc_resetCmd: out std_ulogic;
      dct_resetCmd: out std_ulogic;
      masterReset: buffer std_ulogic;

      romAddr: buffer std_logic_vector(13 downto 0);
      resetStateOut: buffer std_ulogic
);
end entity reset2;


architecture reset2_arch of reset2 is

type RunStateType is (run, waitForRelease, programRom, resetHold);
signal resetState, nextState: RunStateType;
signal nextRomAddr: std_logic_vector(13 downto 0);
signal resetHoldCount: std_logic_vector(4 downto 0);
signal nextResetHoldCount: std_logic_vector(4 downto 0);

begin
      resetStateOut <= '0' when (resetState = run) or (resetState = programRom) else
                  '1' when (resetState = waitForRelease) or (resetState = resetHold);

      fsm: process (resetState, romAddr, romDone, resetButton, resetHoldCount)
      begin
            case resetState is
                  when run =>-- wait for reset button to go down
                        if (resetButton = '0') then
                              nextState <= waitForRelease;
                        else
                              nextState <= run;
                        end if;

                  when waitForRelease =>-- wait for reset button to be released
                        if (resetButton = '1') then
                              nextState <= programRom;
                        else
                              nextState <= waitForRelease;
                        end if;

                  when programRom =>-- wait until DCT programming is done
                        if (romDone = '1') and (romAddr(13) = '0') then
                              nextState <= resetHold;
                        else
                              nextState <= programRom;
                        end if;

                  when resetHold =>-- stall a bit to allow DCT to reset again
                        if (resetHoldCount = "11111") then
                              nextState <= run;
                        else
                              nextState <= resetHold;
                        end if;
            end case;
      end process fsm;


      nextResetHoldCount <= (resetHoldCount + 1) when (resetState = resetHold)
                        else (others => '0');
      nextRomAddr <= (romAddr + 1) when (resetState = programRom)
                  else (others => '1');  -- default state is all-high!

      -- for most sections, assert reset signal throughout the resetting procedure
      ntsc_resetCmd <= '0' when (resetState = run)
                  else '1';
      masterReset <= '0' when (resetState = run)
                  else '1';
      -- DCT reset is asserted before/after programming routine
      dct_resetCmd <= '1' when (resetState = resetHold) or (resetState = waitForRelease)
                  else '0';

      clock_update: process(clk3)
      begin
            if (clk3'event) and (clk3 = '1') then
                  resetHoldCount <= nextResetHoldCount;
                  romAddr <= nextRomAddr;
                  resetState <= nextState;
            end if;

      end process clock_update;
end architecture reset2_arch;


LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE work.std_arith.ALL;



package reset2_pack is
component reset2 port (
      clk3: in std_ulogic;
      resetButton: in std_ulogic;
      romDone: in std_ulogic;

      ntsc_resetCmd: out std_ulogic;
      dct_resetCmd: out std_ulogic;
      masterReset: buffer std_ulogic;

      romAddr: buffer std_logic_vector(13 downto 0);
      resetStateOut: buffer std_ulogic
);
end component;
end package reset2_pack;
```

B.7 The I2C State Machine

The following file *i2c.vhd* generates the I2C state machine, which initializes the Bt829A decoder using the I2C serial bus protocol.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE work.std_arith.ALL;

entity i2c_drive is port (
      sclW, sdaW: out std_ulogic;
      sclR_raw, sdaR: in std_ulogic;
      clk3: in std_ulogic;
      reset: in std_ulogic
);
end entity i2c_drive;


architecture i2c_arch of i2c_drive is

signal sclR: std_ulogic;-- synchronized version of sclR_raw

type sclkStates is (low, high);
signal sclkState, next_sclkState: sclkStates;


constant maxPhase: integer := 31;-- SCK changes phase every "maxPhase" cycles of clk3
constant midPhase: integer := 15;-- phase where data changes, start/stop conditions are
sent, etc.
signal phase, next_phase: std_logic_vector (4 downto 0);

signal sendGo: std_ulogic;
signal bitNum, next_bitNum: integer (0 to 7);-- 0 to 7 addresses bits in a byte
signal byteToSend: std_logic_vector(7 downto 0);

type sdaStates is (idle, startCond, sendByte0, getAck0, ack0Passed, sendByte1, getAck1,
ack1Passed,
               sendByte2, getAck2, success, ackFailed, stopCond);
signal sdaState, next_sdaState: sdaStates;

type sendStates is (sendCmd, waitForReply, done);
signal sendState, next_sendState: sendStates;

signal cmdNum, next_cmdNum: integer (0 to 15);
constant lastCmdNum: integer := 11;

signal oldReset: std_ulogic;

begin
      sclkFsm: process(sclkState, phase, sclR)
      begin

            case sclkState is
            when low =>-- drive SCL low for "phase" cycles of clk3
                  sclW <= '0';
                  if (phase = maxPhase) then-- time to flip SCL?
                        next_sclkState <= high;
                        next_phase <= (others => '0');
                  else
                        next_sclkState <= low;
                        next_phase <= phase + 1;
                  end if;
            when high =>-- drive SCL high for "phase" cycles of clk3
                  sclW <= '1';
                  if (phase = maxPhase) then-- time to flip SCL?
                        next_sclkState <= low;
                        next_phase <= (others => '0');
                  else
                        next_sclkState <= high;
                        if (sclR = '0') then-- is Brooktree holding SCL line LOW?
                              next_phase <= (others => '0');-- wait for BT to release SCL
                        else
                              next_Phase <= phase + 1;
                        end if;
                  end if;
            end case;
      end process;
```

```
	sdaFsm: process (sdaState, phase, sclR, sdaR, byteToSend, sendGo, bitNum)
	begin
		case sdaState is
		when idle =>-- wait until we are ready to generate a start condition
			sdaW <= '1';
			next_bitNum <= 7;
			if (sendGo = '1') and (sclR = '1') and (phase = midPhase) then
				next_sdaState <= startCond;-- drop SDA to signal a start condition
			else
				next_sdaState <= idle;
			end if;

		when startCond =>-- drop SDA now to signal start condition.
			sdaW <= '0';
			next_bitNum <= 7;
			if (sclR = '0') and (phase = midPhase) then
				next_sdaState <= sendByte0;
			else
				next_sdaState <= startCond;
			end if;

		when sendByte0 =>
			if (bitNum = 7) or (bitNum = 3) then-- send "10001000" to device
				sdaW <= '1';		-- this initiates a write to the BT
			else
				sdaW <= '0';
			end if;

			if (sclR = '0') and (phase = midPhase) then
				next_bitNum <= bitNum - 1;
				if (bitNum = 0) then
					next_sdaState <= getAck0;-- if that was the last bit, switch
states
				else
					next_sdaState <= sendByte0;
				end if;
			else
				next_sdaState <= sendByte0;
				next_bitNum <= bitNum;
			end if;

		when getAck0 =>
			sdaW <= '1';-- drive high onto bus, so that BT can pull the line low to
acknowledge
			next_bitNum <= 7;-- set to 8 so that sendByte1 will
			if (sclR = '1') and (phase = midPhase) then
				if (sdaR = '0') then
					next_sdaState <= ack0Passed;-- successful ACK, send another
byte
				else
					next_sdaState <= ackFailed;-- no ACK. give up.
				end if;
			else
				next_sdaState <= getAck0;
			end if;

		when ack0Passed =>-- wait for middle of SCK low before sending next bit
			sdaW <= '1';
			next_bitNum <= 7;
			if (sclR = '0') and (phase = midPhase) then
				next_sdaState <= sendByte1;
			else
				next_sdaState <= ack0Passed;
			end if;
```

```
            when sendByte1 =>
                  sdaW <= byteToSend(bitNum);-- drive bit "bitNum" onto the bus
                  if (sclR = '0') and (phase = midPhase) then
                        next_bitNum <= bitNum - 1;
                        if (bitNum = 0) then
                              next_sdaState <= getAck1;-- if that was the last bit, switch
states
                        else
                              next_sdaState <= sendByte1;
                        end if;
                  else
                        next_sdaState <= sendByte1;
                        next_bitNum <= bitNum;
                  end if;

            when getAck1 =>
                  sdaW <= '1';-- drive high onto bus, so that BT can pull the line low to
acknowledge
                  next_bitNum <= 7;
                  if (sclR = '1') and (phase = midPhase) then
                        if (sdaR = '0') then
                              next_sdaState <= ack1Passed;-- successful ACK, send another
byte
                        else
                              next_sdaState <= ackFailed;-- no ACK. give up.
                        end if;
                  else
                        next_sdaState <= getAck1;
                  end if;

            when ack1Passed =>-- wait for middle of SCK low before sending next bit
                  sdaW <= '1';
                  next_bitNum <= 7;
                  if (sclR = '0') and (phase = midPhase) then
                        next_sdaState <= sendByte2;
                  else
                        next_sdaState <= ack1Passed;
                  end if;

            when sendByte2 =>
                  sdaW <= byteToSend(bitNum);-- drive bit "bitNum" onto the bus
                  if (sclR = '0') and (phase = midPhase) then
                        next_bitNum <= bitNum - 1;
                        if (bitNum = 0) then
                              next_sdaState <= getAck2;-- if that was the last bit, switch
states
                        else
                              next_sdaState <= sendByte2;
                        end if;
                  else
                        next_sdaState <= sendByte2;
                        next_bitNum <= bitNum;
                  end if;

            when getAck2 =>
                  sdaW <= '1';-- drive high onto bus, so that BT can pull the line low to
acknowledge
                  next_bitNum <= 7;
                  if (sclR = '1') and (phase = midPhase) then
                        if (sdaR = '0') then
                              next_sdaState <= success;
                        else
                              next_sdaState <= ackFailed;
                        end if;
```

```
                else
                      next_sdaState <= getAck2;
                end if;

          when success =>-- signal that we successfully completed a transaction
                sdaW <= '1';
                next_bitNum <= 7;
                if (sclR = '0') and (phase = midPhase) then
                      next_sdaState <= stopCond;
                else
                      next_sdaState <= success;
                end if;

          when ackFailed =>-- signal that we could not complete the transaction
                sdaW <= '1';
                next_bitNum <= 7;
                if (sclR = '0') and (phase = midPhase) then
                      next_sdaState <= stopCond;
                else
                      next_sdaState <= ackFailed;
                end if;

          when stopCond =>
                sdaW <= '0';-- pull line down during low SCK. returning to idle state
during
                            -- middle of SCK high pulls SDA up and generates stop
condition.
                next_bitNum <= 7;
                if (sclR = '1') and (phase = midPhase) then
                      next_sdaState <= idle;
                else
                      next_sdaState <= stopCond;
                end if;

          end case;
    end process;


    sendCommands: process(sendState, sdaState, cmdNum)
    begin

          case sendState is
          when sendCmd =>-- tell process above that we wish to send data
                if (sdaState = startCond) then-- has it kicked off yet?
                      next_sendState <= waitForReply;
                      next_cmdNum <= cmdNum;
                else        -- keep asserting our send request
                      next_sendState <= sendCmd;
                      next_cmdNum <= cmdNum;
                end if;

          when waitForReply =>
                if (sdaState = success) then
                      if (cmdNum = lastCmdNum) then
                            next_sendState <= done;
                            next_cmdNum <= cmdNum;
                      else
                            next_sendState <= sendCmd;
                            next_cmdNum <= cmdNum + 1;
                      end if;
                elsif (sdaState = ackFailed) then
                      next_sendState <= sendCmd;
                      next_cmdNum <= cmdNum;
                else
                      next_sendState <= waitForReply;
```

```
                    next_cmdNum <= cmdNum;
                end if;

            when done =>-- all done! don't ever execute this code again (until next reset)
                next_sendState <= done;
                next_cmdNum <= cmdNum;
            end case;
        end process;

        sendGo <= '1' when (sendState = sendCmd) else '0';


        btProgram: process(sdaState, cmdNum)
        begin
            if (sdaState = sendByte1) then-- BT register addresses
                case cmdNum is
                when 0 =>  byteToSend <= "00011111";-- software reset (SRESET)
                when 1 =>  byteToSend <= "00000001";-- input format (IFORM)
                when 2 =>  byteToSend <= "00000010";-- temporal decimation (TDEC)
                when 3 =>  byteToSend <= "00000011";-- MSB cropping (CROP)
                when 4 =>  byteToSend <= "00000100";-- vertical delay (VDELAY_LO)
                when 5 =>  byteToSend <= "00000101";-- vertical active (VACTIVE_LO)
                when 6 =>  byteToSend <= "00000110";-- horizontal delay (HDELAY_LO)
                when 7 =>  byteToSend <= "00000111";-- horizontal active (HACTIVE_LO)
                when 8 =>  byteToSend <= "00001000";-- horizontal scale hi (HSCALE_HI)
                when 9 =>  byteToSend <= "00001001";-- horizontal scale lo (HSCALE_LO)
                when 10 => byteToSend <= "00010011";-- vertical scale hi (VSCALE_HI)
                when 11 => byteToSend <= "00010100";-- vertical scale lo (VSCALE_LO)
                when others => byteToSend <= "11110000";
                end case;

            elsif (sdaState = sendByte2) then  -- command arguments
                case cmdNum is
                when 0 =>  byteToSend <= "00000000";-- don't care
                when 1 =>  byteToSend <= "01001001";-- force NTSC mode
                --when 2 =>  byteToSend <= "00000000";-- no decimation (both flds)
                when 2 =>  byteToSend <= "10011110";-- TDEC = 9E (odd field only)
                when 3 =>  byteToSend <= "00100001";-- VD VA HD HA
                when 4 =>  byteToSend <= "00010110";-- VDELAY = 22 (Default)
                when 5 =>  byteToSend <= "00000000"; -- VACTIVE = 512 (high bit in R3)
                when 6 =>  byteToSend <= "01011000"; -- HDELAY = 32+24+32
                when 7 =>  byteToSend <= "00000000"; -- HACTIVE = 256 (high bit in R3)
                when 8 =>  byteToSend <= "00010101";    -- HSCALE HI = 0x15
                when 9 =>  byteToSend <= "01010101";    -- HSCALE LO = 0x55
                when 10 =>  byteToSend <= "00000000";    -- VSCALE HI = 0x00  (odd fld
only)
                --    when 10 =>  byteToSend <= "01111111";    -- VSCALE HI = 0x1F (both
flds)
                when 11 =>  byteToSend <= "00000000";    -- VSCALE LO = 0x00
                when others => byteToSend <= "11110000";
                end case;
            else
                byteToSend <= (others => '0');
            end if;
        end process;



--      sclR <= sclW;-- enabled for simulation only

        clockUpdate: process(clk3, sclR_raw)
        begin
            sclR <= sclR_raw;-- synchronize the incoming SCLK signal

            if (clk3'event) and (clk3 = '1') then
```

```
                if (reset = '1') then
                      phase <= (others => '0');
                      sclkState <= low;
                      sdaState <= idle;
                      bitNum <= 7;
                      sendState <= sendCmd;
                      cmdNum <= 0;
                else
                      phase <= next_phase;
                      sclkState <= next_sclkState;
                      sdaState <= next_sdaState;
                      bitNum <= next_bitNum;
                      sendState <= next_sendState;
                      cmdNum <= next_cmdNum;
                end if;
          end if;
    end process;
end;


LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE work.std_arith.ALL;


package i2c_pack is
component i2c_drive port (
      sclW, sdaW: out std_ulogic;
      sclR_raw, sdaR: in std_ulogic;
      clk3: in std_ulogic;
      reset: in std_ulogic
      );
end component;
end package i2c_pack;
```

B.8 CPLD Top-Level File #1

The following file *top2.vhd* assembles the code for the DCT, IDCT, LCD, and I2C state machines for placement in a single CPLD. This file also contains logic for receiving and dividing the signal from the 12 MHz clock oscillator.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE work.std_arith.ALL;
USE work.dct_pack.ALL;
USE work.idct_pack.ALL;
USE work.lcd_pack.ALL;
USE work.i2c_pack.ALL;


entity toplev2 is port (

-- DCT DECLARATIONS
```

```
        -- control
        bt_whichRam: buffer std_ulogic;

        -- SRAM right side
        dct_adr: buffer std_logic_vector(15 downto 0);
        dct_cs, dct_oe: out std_ulogic;
        dct_in: in std_logic_vector(7 downto 0);-- access to raw video data

        -- DCT chip
        dct_sbin: out std_ulogic;

        -- DCT misc
        dct_stateout: buffer std_ulogic;


-- IDCT DECLARATIONS
        -- SRAM left side
        idct_adr: buffer std_logic_vector(15 downto 0);
        idct_whichram: buffer std_ulogic;
        idct_whichramnot: buffer std_ulogic;
        idct_cs, idct_rw: out std_ulogic;
        idct_out_pr: buffer std_logic_vector(7 downto 0);

        -- IDCT chip
        idct_sbout: in std_ulogic;
        idct_out: in std_logic_vector(9 downto 0);

        -- misc. outputs
        idct_stateout: buffer std_ulogic;


-- LCD DECLARATIONS
        -- SRAM right side
        lcd_adr: buffer std_logic_vector(15 downto 0);-- LCD "column" data address, SRAM A
        lcd_cs, lcd_oe: out std_ulogic;

        -- LCD screen
        lcd_enab: out std_ulogic;
        lcd_hsync, lcd_vsync: buffer std_ulogic;  -- active low for 640x480; registered to
prevent glitches

-- I2C DECLARATIONS

        i2c_scl_drive,  i2c_sda_drive: out std_ulogic;
        i2c_scl_read, i2c_sda_read: in std_ulogic;

-- STUFF FROM OTHER CPLD

        dct_resetCmd: in std_logic;
        generic_resetcmd: in std_logic;
        comm: in std_logic_vector(3 downto 0);  --reduced from 5 to extend the IDCT output
4/8

--- other signals handled here

        clk12Raw: in std_ulogic;-- clock signal from crystal oscillator
        clk12Out: out std_ulogic;-- all clocks exit before re-entering to minimize slew
        clk3Out: buffer std_ulogic;
        clk12, clk3: in std_ulogic;-- incoming clock signals

-- debugging switches
        swRaw: in std_logic_vector(4 downto 1);
        led: out std_logic_vector(4 downto 1)
);
```

```
end entity toplev2;

architecture top2_arch of toplev2 is



-- Signals used solely by modules here
      constant clockDivideFactor: integer := 1;-- clk3 is clk12 divided by 4
      signal clockPhase: std_logic_vector (2 downto 0);
      signal sw: std_logic_vector(4 downto 1);

begin

      -- placeholder for currently unused i/o pins
      led <= idct_out_pr(7 downto 6) & idct_out_pr(1 downto 0);
      --comm(5 downto 0) <= "010101";

      -- pass the 12 MHz oscillator signal through

      clk12Out <= clk12Raw;

      -- divide 12 MHz clock down to 3 MHz


      clockDivider: process(clk12Raw, clockPhase, clk3Out)
      begin
            if (clk12Raw'event and clk12Raw = '1') then-- use raw clock to min. skew?
                  if (clockPhase < clockDivideFactor) then
                        clockPhase <= clockPhase + 1;
                  else
                        clk3Out <= not clk3Out;
                        clockPhase <= (others => '0');
                  end if;
            end if;
      end process clockDivider;

      -- synchronize raw input signals. coming soon...debouncing!

      syncTime:  process (swRaw, clk3)
            begin
                  if (clk3'event) and (clk3 = '1') then
                        sw <= swRaw;
                  end if;
            end process syncTime;


      -- VHDL convention is:   partSignal => toplevelSignal

      dctPart: dct_sram_drive port map (
            -- SRAM right side
            dctAdr => dct_adr,
            whichRam => bt_whichram,
            data => dct_in,
            cs => dct_cs,
            oe => dct_oe,

            -- DCT chip
            blockStart => dct_sbin,

            -- misc. inputs
            clk3 => clk3,
            reset => dct_resetCmd,

            -- misc. outputs
            stateOut => dct_stateout
      );


      idctPart: idct_sram_drive port map (
            idctAdr => idct_adr,
            whichRam => idct_whichram,
            whichRamNot => idct_whichramnot,
            cs => idct_cs,
            rw => idct_rw,
            dataLatch => idct_out_pr,

            -- IDCT chip
            blockStart => idct_sbout,
            dataIn => idct_out,

            -- misc. inputs
            clk3 => clk3,
            reset => generic_resetcmd,

            -- misc. outputs
            stateOut => idct_stateout
      );


      lcdPart: lcd_sram_drive port map (
            -- ce, rw always enabled in hardware
            -- ENAB on LCD always held low, data starts at C48
            clk12 => clk12,
            reset => generic_resetcmd,

            -- SRAM right side
            lcdadr => lcd_adr,
            whichRam => idct_whichram,
            cs => lcd_cs,
            oe => lcd_oe,

            -- LCD screen
            enab => lcd_enab,
            hsync => lcd_hsync,
            vsync => lcd_vsync  -- active low for 640x480, registered to prevent glitches
      );

      i2cPart: i2c_drive port map (
      sclW => i2c_scl_drive,
      sdaW => i2c_sda_drive,
      sclR_raw => i2c_scl_read,
      sdaR => i2c_sda_read,
      clk3 => clk3,
      reset => generic_resetcmd
      );
end architecture top2_arch;
```

B.9 CPLD Top-Level File #2

The following file *topaux.vhd* assembles the VHDL files for the Reset, Compress, and NTSC state machines for placement in a single CPLD.

```
LIBRARY ieee;
```

```
USE ieee.std_logic_1164.ALL;
USE work.std_arith.ALL;
USE work.reset2_pack.ALL;
USE work.comp_pack.ALL;
USE work.ntsc_pack.ALL;


entity toplev_aux is port (
      --- DCT chip
      dct_lpe: in std_ulogic;
      dct_out: in std_logic_vector(11 downto 0);
      dct_sbout: in std_ulogic;

      --- IDCT chip
      idct_in: buffer std_logic_vector(11 downto 0);
      idct_sbin: buffer std_ulogic;

      --- other i/o
      comp_reset: in std_logic;
      clk3: in std_ulogic;
      idct_coeffsrcraw: in std_ulogic;


-- NTSC DECLARATIONS (39 bits)

      -- sram left side
      bt_adr: buffer std_logic_vector(15 downto 0);
      bt_whichram: buffer std_ulogic;-- to other CPLD
      bt_whichramnot: buffer std_ulogic;
      bt_cs: out std_ulogic;
      bt_rw: out std_ulogic;

      -- Brooktree 829A
      bt_dvalid, bt_active, bt_hreset, bt_vreset, bt_vactive: in std_ulogic;
      bt_field, bt_qclk, bt_clkx1: in std_ulogic;
      bt_oe, bt_rst, bt_i2ccs, bt_pwrdn: out std_ulogic;


--- RESET UNIT DECLARATIONS  (15 bits)

      rst_romdone: in std_ulogic;
      rst_romaddr: buffer std_logic_vector(13 downto 0);
      rst_stateout: buffer std_ulogic;
      rst_genreset: buffer std_ulogic;
      dct_resetcmd: out std_ulogic;

--- other signals handled here
      resetRaw: in std_ulogic;-- unsynchronized reset button
      swRaw: in std_logic_vector(8 downto 5);
      led: out std_logic_vector(8 downto 5);
      comm: in std_logic_vector(5 downto 0)

);
end entity toplev_aux;


architecture topaux_arch of toplev_aux is

signal ntsc_resetcmd: std_ulogic;
signal rst_buttonsync: std_ulogic;
signal sw: std_logic_vector(8 downto 5);
signal idct_coeffsrc: std_ulogic;   -- synchronized version
begin

-- placeholders
```

```
      led <= sw or (sw(7) & sw(8) & sw(5) & sw(6)) or (bt_hreset & bt_vactive & bt_qclk &
sw(5));
--    comm(5 downto 0) <= idct_in(11 downto 9) & idct_in(2 downto 0);



      syncTime:  process (resetRaw, clk3)
      begin
            if (clk3'event) and (clk3 = '1') then
                  idct_coeffsrc <= idct_coeffsrcraw;
                  sw <= swRaw;
                  rst_buttonsync <= resetRaw;
            end if;
      end process syncTime;


      compPart: compressor port map (
            --- DCT chip
            lpe => dct_lpe,
            coeffIn => dct_out,
            blockStartIn => dct_sbout,

            --- IDCT chip
            coeffOut => idct_in,
            blockStartOut => idct_sbin,

            --- other i/o
            reset => rst_genreset,
            clk3 => clk3,
            source => idct_coeffsrc
      );


      ntscPart: ntsc_sram_drive port map (
            ntscAdr => bt_adr,
            whichRam => bt_whichram,
            whichRamnot => bt_whichramnot,
            cs => bt_cs,
            rw => bt_rw,

            -- Brooktree 829A
            dvalid => bt_dvalid,
            active => bt_active,
            hreset => bt_hreset,
            vreset => bt_vreset,
            vactive => bt_vactive,
            field => bt_field,
            qclk => bt_qclk,
            clkx1 => bt_clkx1,
            oe => bt_oe,
            rst => bt_rst,
            i2ccs => bt_i2ccs,
            pwrdn => bt_pwrdn,

            -- misc. inputs
            reset => ntsc_resetcmd
      );


      resetPart: reset2 port map (
            clk3 => clk3,
            resetButton => rst_buttonsync,
            romDone => rst_romdone,

            ntsc_resetCmd => ntsc_resetCmd,
            dct_resetCmd => dct_resetCmd,
            masterReset => rst_genreset,

            romAddr => rst_romaddr,
            resetStateOut => rst_stateout
      );

end architecture;
```