Software Implementation of a Video Resampling Algorithm on the TMS320C80

by
Agnieszka Reiss

Submitted to the Department of Electrical Engineering and Computer Science in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the
Massachusetts Institute of Technology
December 1995

Signature of Author_____
Department of Electrical Engineering and Computer Science
November 15, 1995

Certified by_____
Prof. Anantha Chandrakasan
Thesis Supervisor

Certified by_____
D. Scott Silver, P.E.
Thesis Supervisor

Accepted by_____
Prof. Frederic R. Morgenthaler
Chairman, Department Committee on Graduate Theses

Software Implementation of a Video Resampling Algorithm on the TMS320C80

by
Agnieszka Reiss

Submitted to the Department of
Electrical Engineering and Computer Science
in December, 1995,
in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

Abstract

Over the past few decades Digital Video Effects (DVEs) have been implemented in hardware or, more recently, in non-real-time software on Macintosh computers or Personal Computers. As technology advanced, digital signal processing IC's appeared on the market. Texas Instruments' TMS320C80 is a powerful, programmable parallel processor which can be used to implement DVEs in software, and in real time. This thesis explores several video resampling algorithms and their applicability to the 'C80 and implements approximate polyphase filtering in order achieve a smooth shrinking effect. The implementation is done in C, keeping in mind that a highly optimized version of the same code needs to be able to run in real time.

Keywords: Video, Resampling, TMS320C80

Thesis Advisors: Prof. Anantha Chandrakasan
Department of Electrical Engineering and Computer Science

D. Scott Silver, P.E.
Tektronix, Inc.

Acknowledgments

The work of this thesis was performed at Tektronix, Inc. through MIT's 6A program. Many thanks go out to my thesis advisors, D. Scott Silver, P.E., at Tektronix and Professor Anantha Chandrakasan at MIT, for their invaluable guidance throughout the thesis process. I would also like to thank Bruce Penney of Tektronix for sharing his knowledge of video, and all the members of the Video and Networking Division Derivative Products Group at Tektronix, especially Warren Kafer and Dale Jordan, for their ready and insightful advice. Last but certainly not least, I would like to thank Dean Messing and Keith Slavin of Tektronix for their help with resampling theory.

I could not have done this work without the support of my family and friends. I would especially like to thank my parents, brother, and sister in law, as well as Amy Singer, Bridget Banas, and Evan Matteo, for bolstering my spirits over the past six months.

Finally, I would like to thank anybody and everybody I have been able to talk into going to the beach with me over the course of my undergraduate, graduate, and particularly my thesis writing careers. Their good company and the sight, sound, smell, and feel of the Atlantic and Pacific Oceans have made life better in immeasurable ways.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The goal of this thesis project is to implement the "Shrink" digital video effect (DVE) for real-time video on the TMS320C80 ('C80). The "Shrink" effect is essentially a resampling problem; for a given input video sequence the output video sequence is a smaller version of the input. The output video must be able to change size smoothly over time and be of high enough quality that any artifacts introduced by the processing are not easily noticed.

The 'C80 is a powerful programmable processor particularly well suited for video made by Texas Instruments. Its ability to efficiently move data around on and off-chip memory banks and to process large amounts of data quickly make it possible to implement a DVE such as "Shrink", in real-time software. DVE's have traditionally been implemented in hardware or in non-real-time software.

This thesis explores several different ways of implementing video resampling in C language on the 'C80 such that, if the inner loop of the algorithm were to be highly optimized, it could run in real time. Special consideration is given to the 'C80's architecture so as to use the processor most efficiently, and the special requirements of video.

A two stage algorithm based on approximate polyphase filtering for resizing video down to 25% of its original size is implemented in C and tested. The results were evaluated subjectively, and were deemed to be quite good.

1.1 'C80

The 'C80 consists of six main pieces: a floating point Master Processor (MP), four fixed point Parallel Processors (PPs), and a data movement engine called the Transfer Controller (TC). Essentially, the MP is used to calculate parameters and to direct all the data movement and processing on the 'C80. The TC actually executes all the MP's data movement commands and the PPs are used to do all the actual processing.

1.2 Video and Digital Video Effects

Processing digital video is not necessarily the same as processing other forms of digital data. Video is typically quantized to eight or sometimes ten bits whereas audio, for example, is typically quantized to sixteen bits. Video, therefore, does not have to be processed with the same precision as audio. On the other hand, any error artifacts that change from frame to frame are disturbing to a viewer. For example, alias in a piece of undersampled digital video depends on the phase of the sampling. If the sampling phase differs from frame to frame the alias jumps around on the screen from frame to frame, creating a disagreeable effect. So, video processing must be very smooth. In the case of resampling this means that as a piece of video is shrunk down at an arbitrary rate the effects of the resampling process must be similar enough for neighboring output picture sizes that any artifacts caused by the processing are as unnoticeable as possible. Also, the resampling must be done at sub-pixel resolution so that the change in size from frame to frame is uniform, regardless of the rate of shrinking. Furthermore, artifacts which appear in patterns are more noticeable than artifacts which appear to be random, so care must be taken so that the results of the resampling processing are as close to uniform across the entire picture as possible. In

many cases tradeoffs must be made between the overall quality of a processed piece of video and both temporal and spatial continuity.

1.3 The Video Resampling Problem

On the algorithm side there are two main issues in resampling video: correctly determining the source of an output pixel, and low-pass filtering to ensure that no aliasing artifacts are introduced by the reduction in signal bandwidth that accompanies a reduction in signal sampling frequency. In most cases an efficient algorithm does both of these operations simultaneously, and at the lower output sampling rate rather than at the higher input sampling rate.

This thesis explores several ways of resampling signals, evolving from the classical but impractical upsample and then downsample strategy to Crochiere and Rabiner's more practical but still inadequately flexible for this application polyphase filtering approach to the final approximate version of polyphase filtering.

Approximate polyphase filtering is implemented in two stages. If a piece of video is to be shrunk to more than 50% of its original size horizontally and vertically straight approximate polyphase filtering is used. The size and characteristics of the filters used are varied according to the desired output video size such that only as many computations as can be done in real time by the 'C80 are necessary and so that the appropriate low-pass filtering is done. For output sizes less than 50% of the input size the same approximate polyphase filtering is done after an initial stage of half-band filtering and reduction in size by 50% in each direction.

Several different design methods for the filters used are also explored, from the Remez-Exchange algorithm to various windowing methods. Designing the filters proved to be quite a challenge because in most cases the design specifications were

very overconstrained. In the end most of the filters used are designed using Hamming windows, and a few others are designed using Kaiser windows.

Chapter 2

TMS320C80 and Development Environment

This chapter provides an overview of the processor used in this thesis, the TMS320C80, or 'C80, and its components, the hardware environment within which it operates, and the software development environment.

2.1 Introduction

The 'C80, is a monolithic multi-instruction, multi-data (MIMD) parallel processor. It consists of a floating point Master Processor and four fixed point Parallel Processors, as well as 50 Kbytes of on-chip static RAM. It also has a Transfer Controller which handles data transfers and a Video Controller which handles video timing. It can run as fast as 50 MHz, but the development system runs at 30 MHz.

The Master Processor is a 32-bit general purpose RISC processor with a 4 Kbyte instruction cache and a 4 Kbyte data cache. The Parallel Processors are 32-bit Advanced Digital Signal Processors (ADSPs), each with a 2 Kbyte instruction cache and 8 Kbytes of data RAM divided into three 2 Kbyte blocks for data and one 2 Kbyte block for parameters. The on-chip memory is shared such that all of it is accessible to the MP, and all PP memory is accessible to the PPs through a crossbar structure. There are many possible ways to use the shared memory; the configuration described above, however, is standard. The Transfer controller handles data between on-chip and off-chip RAM. The Video Controller can take care of video timing issues, but is not used in this project.

The MP and the PPs can be programmed in C using compilers provided by Texas Instruments, or in assembly language. The MP and the PPs have different compilers and different assembly languages. Typically the MP is used for overhead and for setting up tasks for the PPs. The PPs are used for repeated calculations, such as convolving a line of data with an FIR filter. This makes for a very efficient scheme since the major time sink in processing video is in processing a large number of data points (1440 samples per line and 486 lines per frame for CCIR-601 video, which is a widely used broadcast studio standard) and each of the four PPs can process data nearly continuously if all the overhead is taken care of by the MP.

For the purposes of this thesis both MP and PP code is written in C. The code written relies on a backbone of previously written functions that take care of some of the Transfer Controller operations and interface with the development hardware.

2.2 'C80

A block diagram of the 'C80 and its main data paths is shown in Figure 2-1[TI95] . The MP and the four PPs each have their own memory banks which they can access locally or globally. The TC has access to all these memory banks via the crossbar structure and, along with the VC, acts as the interface to the external world.

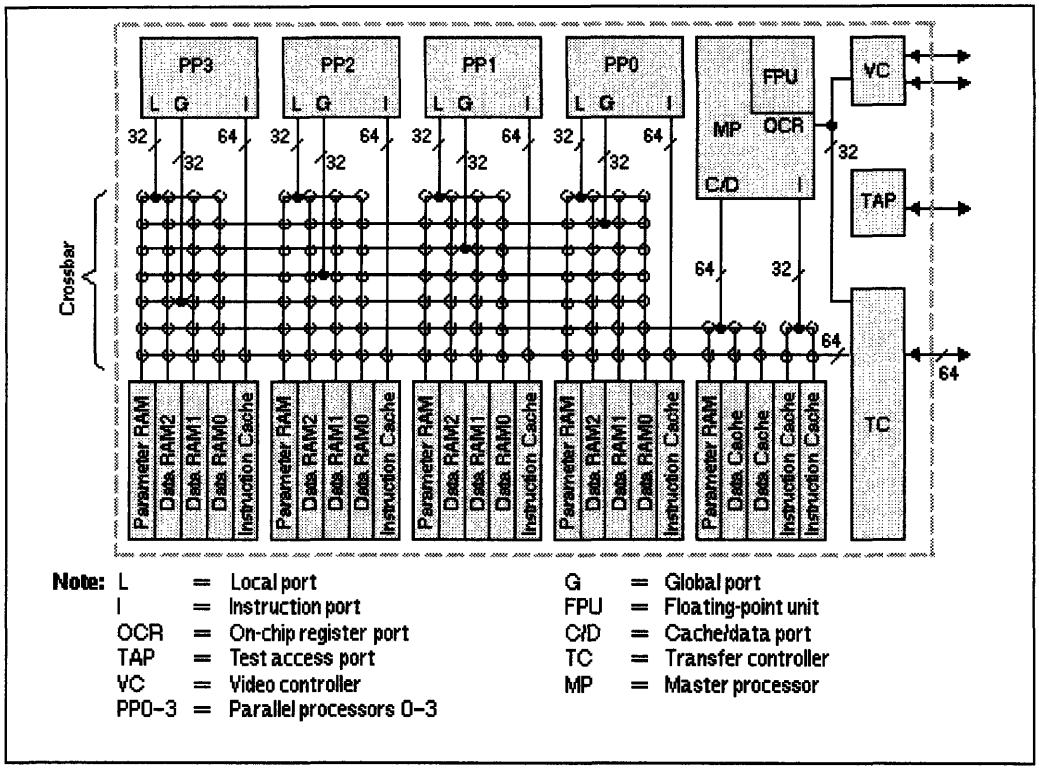

Figure 2-1: Block Diagram of The 'C80 and its Main Data Paths

2.2.1 Master Processor

The MP is a general purpose processor RISC with an integral IEEE-754 floating point unit. Its instruction set contains many of the primitives used by C compilers, and it has a zeroing register that is often used by C, so that it is especially efficient in executing C code. It has a 4 Kbyte instruction cache, a 4 Kbyte data cache, and 2 Kbytes of parameter RAM. In addition, it is able to access all of the PP RAM through the crossbar structure, and to start or halt the PPs' processing using the `cmnd` command.

The function of the MP is to coordinate all the activities of the 'C80. It interfaces with the hardware and sets up all the video timing and I/O via the Video Controller. For each frame it sets up and controls the PPs' processing: It calculates appropriate parameters and writes them to the PPs' parameter RAMs; it controls data flow through

the PPs by setting up and dispatching Transfer Controller tasks; and it tells the PPs when to process and on which of its data memory banks to operate.

2.2.2 Parallel Processors

A PP is a 32 bit DSP with a 2 Kbyte instruction cache, 2 Kbytes of parameter RAM, and three 2 Kbyte banks of data RAM. Each PP also has access to all the other PPs' data RAMs through the crossbar, but that feature is not used in this project. Among the PP's features is a 16×16 or dual 8×8 bit multiplier, a splittable three input ALU, and a 32 bit barrel rotator. It has 64-bit instruction words which support parallel operations. For example, a PP can execute one 16×16 bit or two 8×8 bit multiplies, an ALU operation such as a shift-and-add or up to four adds, two memory accesses, and the necessary pointer updates all in one cycle. Also, each PP has a three stage pipeline so that it supports a fast instruction cycle and three hardware loop controllers which allow for zero overhead looping.

Essentially, the PP is the number crunching workhorse of the 'C80. Considering the PPs' tight loop capabilities and the fact that there are four of them this is very efficient. The PPs waste very little time on overhead. They receive instructions from the MP that tell them what operation to perform on the data in its data banks, and just carry them out.

2.2.3 Transfer Controller

The Transfer Controller handles movement of data and instructions for the MP, PPs, Video Controller, and any external devices. It has a 64 bit interface to the crossbar structure which allows it to access all of the on-chip RAM, and another 64 bit interface which allows it to access off-chip memory. It also handles memory control functions such as cache fills and write backs, DRAM refreshes, and others.

The TC does moves data in a process called Packet Transfers. Packet Transfers can be initiated by the MP, the PPs, the Video Controller, or external devices, though they are initiated exclusively by the MP in this implementation. A Transfer Packet is a multi-dimensional structure containing information about source and destination addresses and how they are to be updated, how much information is to be transferred, and how the source data and address space is to be mapped to the destination data and address space.

2.3 Development Environment

This section describes the hardware for which the code of this thesis is written and on which it is tested and debugged as well as the tools and existing code base that are used in developing code.

2.3.1 Development Hardware

The development hardware used consists of the circuit board shown in Figure 2-2. The heart of the board is a 'C80. It also has five 128 Kbyte × 64 bit banks of RAM, divided into two banks of input memory, two banks of output memory, and one bank of intermediate or 'Program' memory. The 'C80 accesses the memory banks over a 32 bit address and 64 bit data bus. Two switches ping-pong the input and output memory banks so that, for example, one input memory bank is being used by the 'C80 while the other is being used to store incoming video, and one output memory bank is being used by the 'C80 while the other is being written out to a video display. The external hardware takes care of all the video timing issues, so the Video Controller on the 'C80 is not used. There is also one small bank of code memory which is used by the 'C80 via an 8 bit bus. Video data enters the board over a parallel cable in Rec. CCIR-601 (601) format. It is deformatted and de-multiplexed to form the 64 bit data that

is then stored in one of the input memory banks. After processing, the 64 bit data in one of the output memory banks is multiplexed and formatted to 601 and leaves the board on parallel cable, as it came in.

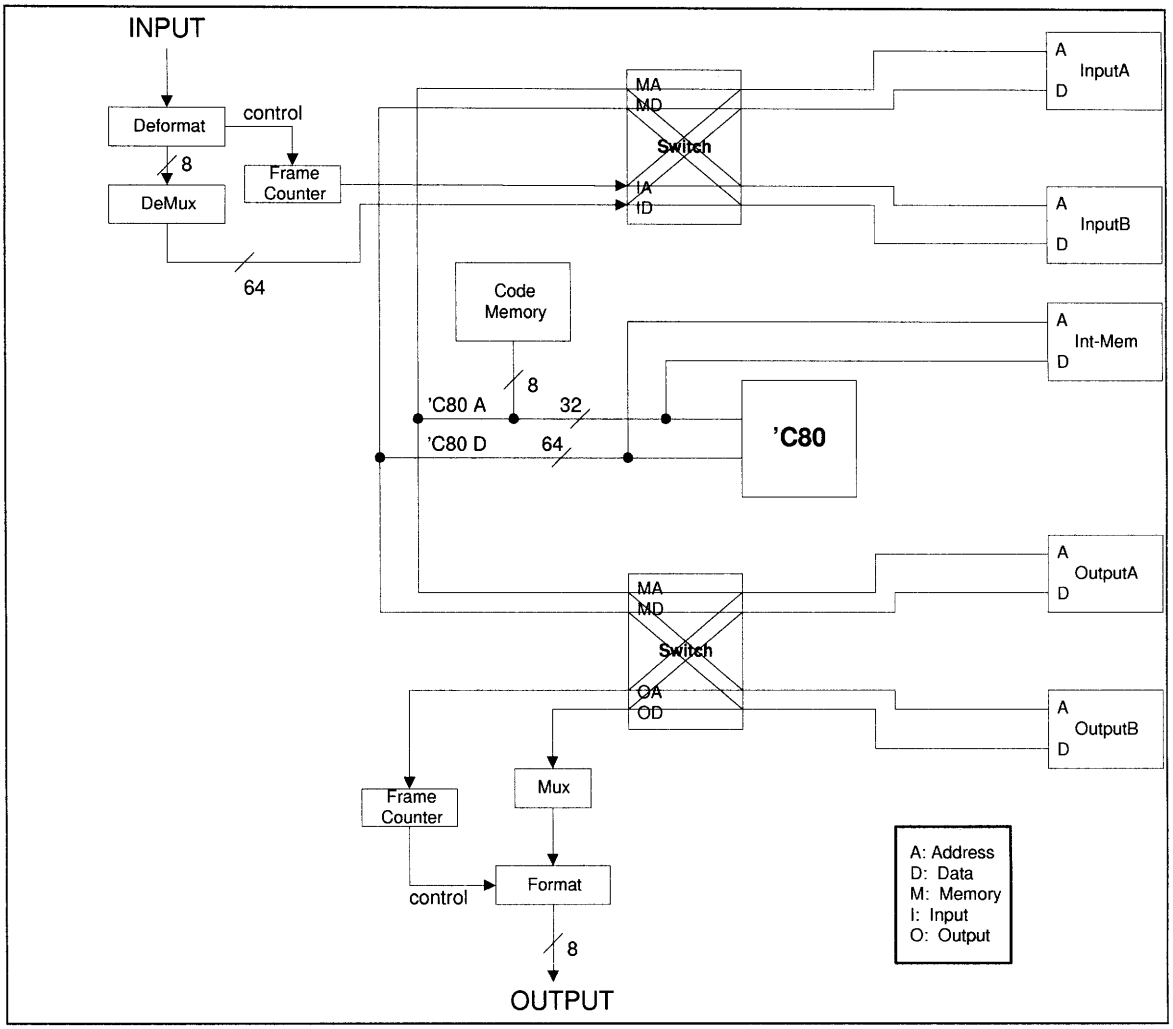

Figure 2-2: Block Diagram of the Development Hardware

2.3.2 Programming Environment

The C code written for this project uses a library of functions that facilitate Transfer Controller operations and setting up Transfer Packets. It also uses a set of functions that set up and execute hardware functions such as I/O, video timing and ping-ponging input and output memory banks, cycling LEDs, and reading information from

the development user interface, a simple joystick with one control button. This backbone of code was written by Dale Jordan and Warren Kafer.

Also, a program written by Dale Jordan which implements resampling by the 'magic' number of $\frac{1}{3}$ was used as a guide. $\frac{1}{3}$ is a 'magic' number for two reasons: First, resampling by a rational fraction with numerator one is simple downsampling, and no sophisticated methods to map points which lie in between actual input samples to output samples are necessary. Second, when resampling by $\frac{1}{3}$ input lines from one field are always mapped to that same field in the output. Jordan's program, then, is simpler than the one put together for this thesis project, but serves as an important and useful reference, especially for developing the MP's flow control strategy.

Code was compiled using the MP and PP compilers provided by Texas Instruments, and was tested and debugged directly on the development hardware. An emulator was not available.

Chapter 3

Video and Digital Video Effects

This chapter describes the type of video that this thesis deals with, known as CCIR-601 video. It then gives a brief overview of the kinds of digital video effects that an editor may wish to use when putting together a piece of video.

3.1 Video

This thesis deals with video under the standard known as CCIR-601, or just 601, as defined by the International Radio Consultative Committee (CCIR) in its *Recommendation 601-1.* Additionally, it is constrained to video which is displayed at 59.94 Hz (heretofore referred to as 60 Hz), with interlaced scanning like in the U.S. NTSC system as opposed to video which is displayed at 50Hz also with interlaced scanning as in the European PAL system. This means that a frame of digital video is defined as 486 lines of 720 pixels each where each pixel is stored in 8 bit 4-2-2 component format.

4-2-2 component format means that digital color video is stored in luminance, *Y*, and chrominance, *Cb* and *Cr*, components. This is similar to the more familiar *Y-I-Q* format, which maps to *R-G-B* through the matrix transformation in Equation 3.1, and where *Y* can be interpreted as the intensity or gray level of a black and white image and *I* and *Q* contain the color information.

$$\begin{bmatrix} Y \\ I \\ Q \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ 0.596 & -0.274 & -0.322 \\ 0.211 & 0.523 & 0.312 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \Leftrightarrow \begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1.000 & 0.956 & 0.621 \\ 1.000 & -0.273 & -0.647 \\ 1.000 & -1.104 & 1.701 \end{bmatrix} \begin{bmatrix} Y \\ I \\ Q \end{bmatrix}$$

Equation 3.1

In *Cb-Y-Cr* format *Y* is the intensity and *Cb* and *Cr* contain the color information in the form *Y* minus the blue component and *Y* minus the red component, respectively. For 8 bit video *Y* has 220 quantization levels, with black corresponding to 16 and white corresponding to 235, though the signal level may occasionally be higher than 235. *Cb* and *Cr* have 225 quantization levels centered about 128 such that if both are equal to 128 the picture appears to be monochrome. The three components are constructed from the digitized values of the primary red, green, and blue analog signals R_a, G_a, and, B_a according to Equation 3.2, where $R_d = \text{int}(219 \cdot R_a) + 16$, $G_d = \text{int}(219 \cdot G_a) + 16$, and $B_d = \text{int}(219 \cdot B_a) + 16$.

$$\begin{aligned} Y &= \frac{77}{256} R_d + \frac{150}{256} G_d + \frac{29}{256} B_d \\ Cr &= \frac{131}{256} R_d - \frac{110}{256} G_d - \frac{21}{256} B_d + 128 \\ Cb &= -\frac{44}{256} R_d - \frac{87}{256} G_d + \frac{131}{256} B_d + 128 \end{aligned}$$

Equation 3.2

Since the human eye is less sensitive to color than to intensity, color information can be sampled at a lower rate than is luminance information. *Y*, therefore, is sampled at 13.5 MHz and *Cb* and *Cr* are sampled at half that frequency, 6.75 MHz. When stored in memory two consecutive pixels of a line of digital video are represented by the 4-byte

word *Cb-Y-Cr-Y*. Vertical bandwidth is the same for both luminance and chrominance components, so a 3×6 pixel picture, for example, is stored in the format shown in Figure 3-1.

Cb	Y	Cr	Y	Cb	Y	Cr	Y	Cb	Y	Cr	Y
Cb	Y	Cr	Y	Cb	Y	Cr	Y	Cb	Y	Cr	Y
Cb	Y	Cr	Y	Cb	Y	Cr	Y	Cb	Y	Cr	Y

Figure 3-1: Representation of 3x6 Pixel Picture in Memory

Because of bandwidth considerations at the beginning of the television era, video is typically interlace scanned. Each frame is divided into two fields, where each field is illuminated on the screen on every other line, with one field filling in the lines in between the other's. The fields are alternately illuminated at a rate of 60 fields per second or 30 frames per second under the NTSC system or at 50 fields or 25 frames per second under PAL. To the human eye this appears to be a continuous picture.

3.2 Digital Video Effects

The category Digital Video Effects includes rotations, page turns, scaling, perspective projection, and other effects. These effects are used for all kinds of video production, from newscasts to sportscasts to corporate videos to film. Scaling, for example, can be used to effect bringing a picture in from infinity or simply to resize a piece of video to fit in a particular portion of the screen.

DVEs have historically been implemented in hardware, for example in the Grass Valley Group's Kaleidoscope and DPM100 instruments and the Alladin Pinnacle. Most effects can be implemented in real time in hardware, but this is expensive and often inflexible. More recently, they have been implemented in software on MACs or PCs, but

not in real time. The work done for this thesis is a step towards real-time software implementation.

Chapter 4

Shrinking

The digital video effect implemented in this thesis is shrinking a video image. This chapter describes the shrink effect in more detail and discusses the goals of an implementation of it. It also points out some of the major issues that affect the implementation. Finally, it provides a brief description of some previous implementations of the effect.

4.1 Description

Successive frames of video are to be smoothly shrunk from full size to arbitrarily small size at an arbitrary rate. This is essentially a resampling problem. The number of pixels is decreased: for an input picture of $N \times M$ pixels and a shrink factor *P*, the output picture is $N \cdot P \times M \cdot P$ pixels, where *P* ranges from 1 or 100% for large output pictures to zero for infinitely small output pictures. This thesis project is mostly confined to the case where $25\% < P \leq 100\%$. Unless *P* is a rational number with numerator one, like ½ or ⅓, output pixels do not map directly to input pixels; they generally map to points somewhere in between input pixels. So, each output pixel's value must somehow be calculated from the input pixels surrounding the point to which it maps. Also, the high frequency content of the input picture must be attenuated in order to prevent aliasing.

4.2 Goals

An acceptable implementation of picture resampling for video needs to meet a certain set of quality standards. These standards, which speak to both the quality of the filtering which is used in the resampling and to the necessary sub-pixel resolution are outlined below. Additionally, requirements for real-time operation and use of the processor are discussed.

4.2.1 Quality

The ideal shrunk picture should look exactly like the original picture when viewed from farther away. That means that it has to have exactly the same DC values as the original, as well as that it has to retain as much of the high frequency components of the original as possible while remaining non-aliased. If the input picture has frequency content up to the Nyquist frequency ϖ_N, which is equal to π for a discrete time signal, the output picture can only have frequency content up to the new Nyquist frequency $\varpi_{N'} = P \cdot \varpi_N$.

In order for the shrinking operation to be smooth at arbitrary rates the output picture size cannot be limited to whole pixel values. In fact, it must be possible to change the shrink factor by increments small enough so that when the picture is being shrunk extremely slowly the difference between successive output frames is imperceptible. As a point of reference the shrink effect developed in this thesis project should have identical or better performance than the Grass Valley Group (GVG) DPM100 DVE. In the DPM100 the shrink factor P is changeable by increments as small as 0.01%.

Also, the mapping of a desired point in the input to an output sample needs to be implemented with a certain amount of precision. It was decided that at least one tenth of a sample of resolution was necessary.

4.2.2 Processor Time

First and foremost the resampling processing must be done in real time. "Real Time" means that all processing on each frame of video must be complete in time for successive output frames to be displayed at the same rate as the input frames come in. For 60 Hz video this means that all the processing for each field must be complete inside 16.6 msec. Since video is being treated as frames, not fields, for the purposes of this thesis, all processing for each frame must be complete inside 33.3 msec.

Additionally, it is desirable to use a decreasing proportion of the available processing power as the output picture size decreases. For very small output pictures it is desirable for the processing time to be short enough so that several different small output pictures can be computed for each frame.

4.3 Major Issues

An algorithm which reaches the goals of this project must take into account several issues, such as algorithm separability, the limitations of how much a filter of a certain size can accomplish, and the fact that the video to be used is interlace scanned. These issues and their impact on this thesis project are addressed below.

4.3.1 Separability

Resampling can be viewed as a fully separable problem. This means horizontal and vertical processing on a frame of video can be done separately and independently. A frame can be resampled first vertically, then horizontally, or vice versa. The output will be the same. So long as the horizontal and vertical processing is the same the

number of computations will also be the same. It is also possible to view the problem as a non-separable one, where the horizontal and vertical resampling is done simultaneously. This, however, is prohibitively more complicated and computationally expensive to implement. It is more complicated because, if nonseparable filters are used, it is necessary to look a certain number of samples in both the horizontal and the vertical directions in order to compute a single output sample. This makes it difficult to efficiently pipeline PP operations because of the 2 Kbyte size of a PP's Parameter RAM block and the work involved in setting up the TC to move data in a way that would make it possible. Using non-separable filters is also more computationally expensive because, for an $a \times b$ point filter $h[n_1, n_1]$, the convolution sum

$y[n_1, n_2] = \sum_{k_1=-\infty}^{\infty} \sum_{k_2=-\infty}^{\infty} x[k_1, k_2] h[n_1 - k_1, n_2 - k_2]$, with *x* being the input, requires $a \cdot b$ multiplies and adds to compute a single output sample $y[n_1, n_1]$ whereas with a separable filter of the same size $h[n_1, n_1] = h_{horizontal}[n_1] h_{vertical}[n_2]$ the convolution sum can be rewritten as $y[n_1, n_2] = \sum_{k_2=-\infty}^{\infty} \left(\sum_{k_1=-\infty}^{\infty} x[k_1, k_2] h[n_1 - k_1] \right) h[n_2 - k_2]$ and only requires $a + b$ multiplies and adds to compute a single output sample. For most combinations of *a* and *b* $a \cdot b$ is significantly greater than $a + b$.

4.3.2 Filter Limitations

The effectiveness of simultaneously cutting down the bandwidth of the picture and interpolating output pixel values is constrained by the performance of the filters used to do so. There is no such thing as a finite length ideal filter. So, filters were designed to take the best possible advantage of tradeoffs between their characteristics and the amount of computation necessary to implement them.

4.3.3 Interlaced Scanning

Interlaced scanning causes some very noticeable artifacts when video is resampled and low-pass filtered vertically. If the video is treated as a series of frames information which started out in one field of a frame may very well be mapped to the other field. If that happens then the information is displayed in reverse chronological order, causing smearing in areas where there is horizontal motion. If instead the video is treated as a series of fields there is no horizontal smearing, but vertical resolution is degraded.

A good example of the problems interlaced scanning can cause in resampling video occurs when the video size and bandwidth is halved in the vertical direction. In the original full size picture, if an object is moving horizontally across the screen it will be farther along in the direction of motion in the second field of a frame than in the first. If the video is treated as a series of frames by the resampling algorithm the area of motion in the output picture will look smeared. This is because if the input is subsampled with no filtering, information from only one of the input fields is mapped to the output, and information from the other field is thrown away. The moving object is equally far along in the direction of motion in both output fields. When it is displayed in interlaced format its horizontal lines are alternately lined up and doubly spread out, with an overall effect of smearing the object horizontally. This effect is outlined in Figure 4-1. Low-pass filtering in addition to subsampling allays this effect, but does not change its nature. Since a low pass FIR filter has a peak at its center most of the information in an output pixel comes from the input pixel to which it maps, so fields can still be easily reversed.

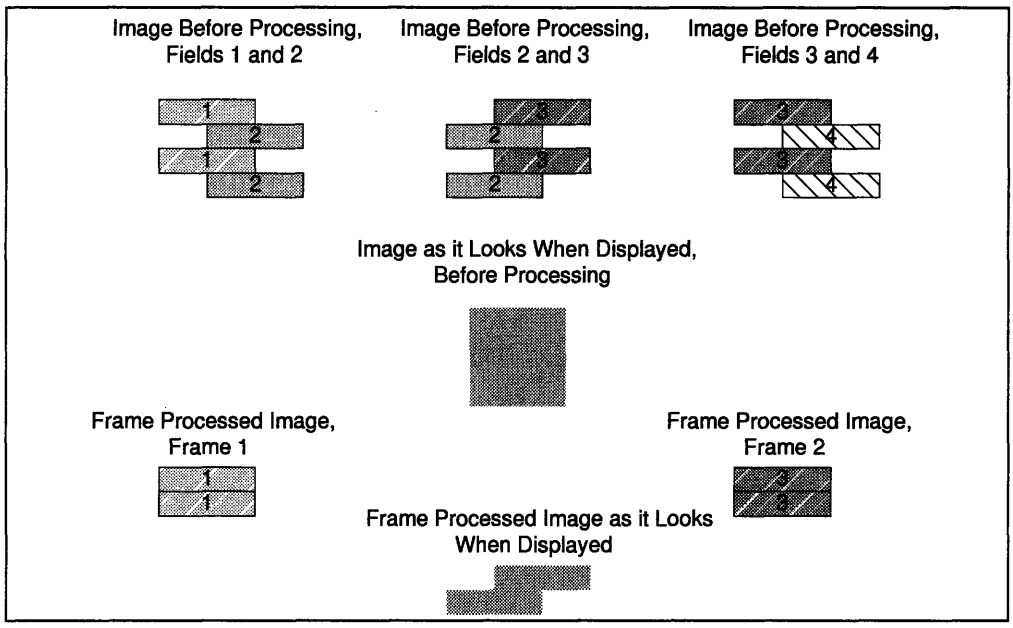

Figure 4-1: Example of the Effects of Treating Interlaced Video as a series of Frames when Decimating Vertically by a Factor of Two.

One way to fix this motion problem is to treat video as a series of fields. This, however, cuts down vertical bandwidth makes the entire output picture appear blurry. That is not a big problem in the area of motion, because human vision is less precise where motion is involved, but in stationary areas the picture appears significantly degraded.

Ideally, video would be processed as a series of fields in areas with motion, and as a series of frames in stationary areas. This requires storing frame to frame information and implementing a motion detection algorithm, however. This project has neither the memory in the development hardware nor the compute-time to do such motion detection. Consequently video is treated as frames in this project. Simple modifications to the code could be made to treat video in fields.

Chapter 5

Description of Algorithm

There are several possible ways to approach the resampling problem. This chapter describes and evaluates the classical but impractical way to do it by upsampling and downsampling, as well as alternative schemes such as classical polyphase filtering, approximate polyphase filtering, a two step filter and then interpolate scheme, and half band filters. It then discusses the collection of algorithms that were actually chosen for implementation on the 'C80 system and the reasoning behind the choices. Finally, it outlines the process for designing the filters used in the final implementation.

For the purposes of this thesis resampling video is treated as a separable problem, as explained in Section 4.3.1. Therefore the following discussion of resampling algorithms speaks in terms of *signals* rather than *pictures*, where a signal can be either a line or a column of data.

5.1 Possible Methods of Resampling

The classical way to resample a signal is to first upsample and then downsample it. Upsampling and downsampling are simple ways to increase and decrease the sampling rate by integer factors, respectively. Taken together, they can change the sampling rate by factors which are rational fractions.[OPP89]

Upsampling and downsampling, however, is not a very practical approach due to limitations in memory space and compute time. For example, a line of video upsampled by a factor of two no longer fits in a 2 Kbyte PP data RAM block. There are ways to get around this, such as having the PPs process half lines of data at a time instead of whole

lines, or maybe finding a way to do the computations in-place so that more than one RAM block is available to a PP at a time. Both of these schemes sacrifice speed and neither scales well, however.

There are still other alternatives. One is a sort of virtual upsampling, where no upsampling is done but the filter used for each output point generates the point as if it were filtering an upsampled signal. The filter used would have to depend on how the input points map to the output point. This is the method of periodic filters, also known as polyphase filters.[CRO83] [LIM88]

Another is low pass filtering combined with some kind of interpolation. A picture is low pass filtered with an appropriate cutoff frequency, then any points that translate directly to output points are transcribed, and all other output points are bilinearly or bicubically interpolated from the intermediate data. This method scales quite well, but is somewhat computationally intensive since the low pass filtering convolution must be carried out for every single input point, and then each output point has to be interpolated from the intermediate data.

5.1.1 Upsampling and Downsampling

Downsampling is a simple method of reducing the size of a picture by an integer factor. It also provides an illustrative example of the necessity for low pass filtering. If combined with upsampling, an equivalent method for increasing the size of a picture by an integer factor, downsampling can be used to change the size of a picture by a factor which is an integer fraction. Upsampling and downsampling are described in detail by Oppenheim and Schafer [OPP89], and are summarized below.

5.1.1.1 Description

Consider a line of digital video, $x[n]$ of length N. A simple minded way to reduce the length of this line to N/M would be to subsample it, so that the output $y[n]$ is $y[n] = x[nM]$. This is simple minded because it causes aliasing unless the input $x[n]$ is bandlimited such that it has no frequency components at frequencies larger than 1/M times the Nyquist frequency. $x[n]$ is a discretized version of the analog signal $x_a(t)$ evaluated at $t = nT$, where T is the sampling period, or one over the sampling frequency. So $x_a(t) = x[nT]$. In the frequency domain this means that the Fourier transform of $x[n]$, $X(e^{j\varpi})$, is equal to the periodic replication of the Fourier transform of $x_a(t)$, $X(j\Omega)$, evaluated at $\Omega = \varpi/T$, where the Fourier transform is defined as $X(j\Omega) = \int_{-\infty}^{\infty} x_a(t)e^{-j\Omega t}dt$. So $X(e^{j\varpi}) = \frac{1}{T}\sum_{k=-\infty}^{\infty} X_a\left(j\frac{\varpi}{T} - j\frac{2\pi k}{T}\right)$. This is illustrated in Figure 5-1.

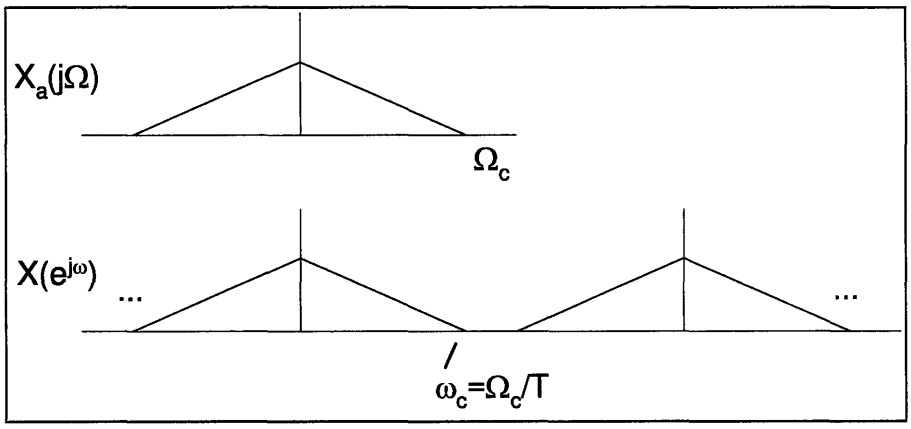

Figure 5-1: Frequency Domain View of Continuous to Discrete Time Conversion

If $y[n]=x[nM]=x_a(nMT)$ periodic replication in the frequency domain can cause aliasing. $Y(e^{j\varpi})=\frac{1}{MT}\sum_{k=-\infty}^{\infty}X_a\left(j\frac{\varpi}{MT}-j\frac{2\pi k}{MT}\right)$, which means that the frequency content of the input signal is stretched out to M times its original location. Aliasing occurs and information is lost wherever this causes overlapping.

Figure 5-2 illustrates this for the case M = 2.

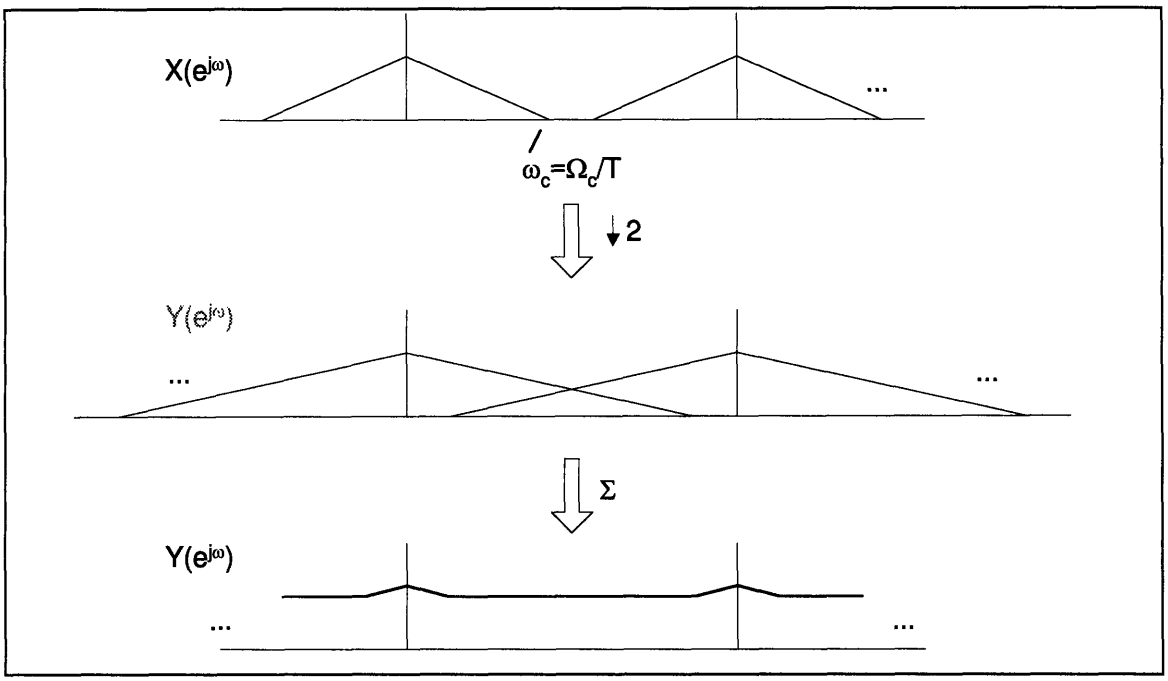

Figure 5-2: Downsampling by 2 Without Proper Low Pass Filtering

The solution to the aliasing problem is to low pass filter the input data so that it is bandlimited to 1/M times the Nyquist frequency. This process and its effects in the frequency domain are illustrated in

Figure 5-3 and

Figure 5-4, respectively.

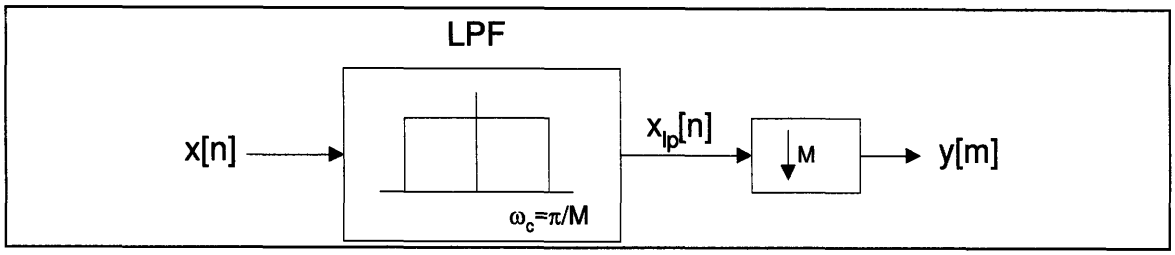

Figure 5-3: Block Diagram of Proper Downsampling

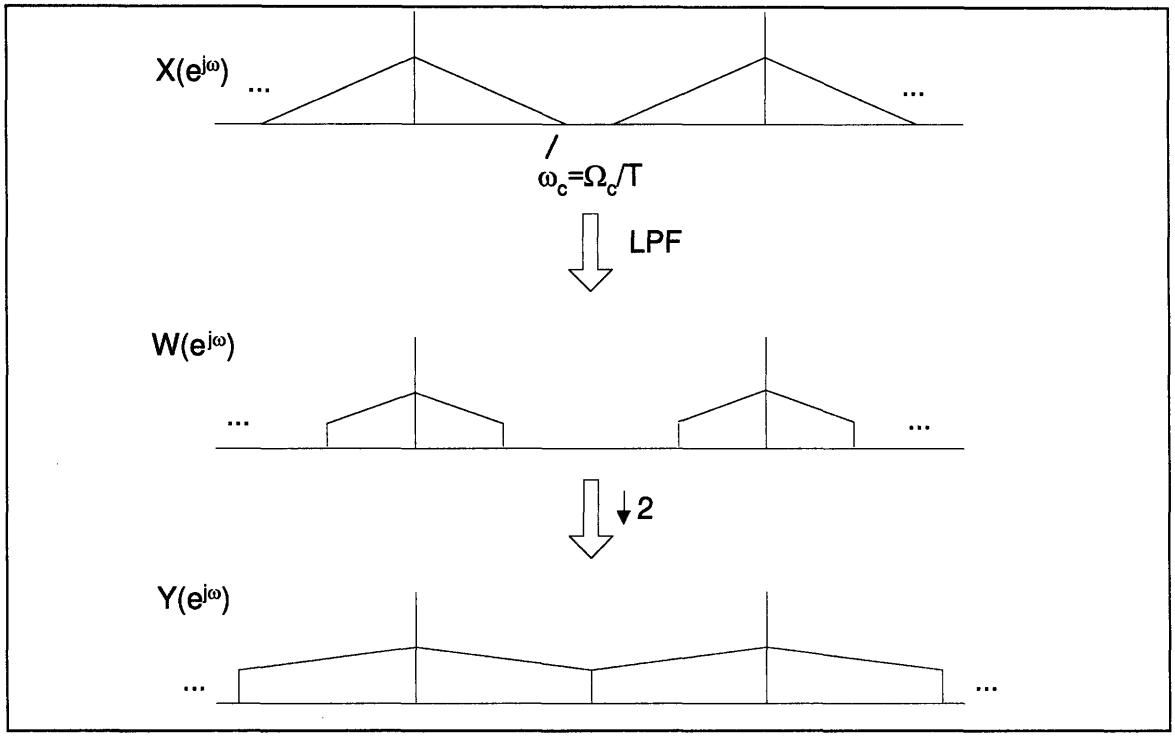

Figure 5-4: Downsampling with Filtering

Upsampling is a similar process, used to increase the sampling frequency of a signal $x[n]$ by an integer factor *M*. It is commonly done by stretching the input signal $x[n]$ in the time domain by inserting $M-1$ zeros in between each sample, with the result that $y[m]=\begin{cases} x[n/M], & (n/M)=\lfloor n/M \rfloor \\ 0, & otherwise \end{cases}$. In the frequency domain, this means

that $Y(e^{j\varpi}) = \frac{M}{T}\sum_{k=-\infty}^{\infty} X_a\left(j\frac{\varpi M}{T} - j\frac{2\pi kM}{T}\right)$, or that the frequency spectrum of the output $y[m]$ is a compacted version of the frequency spectrum of the input as shown in Figure 5-5 for the case where $M = 3$.

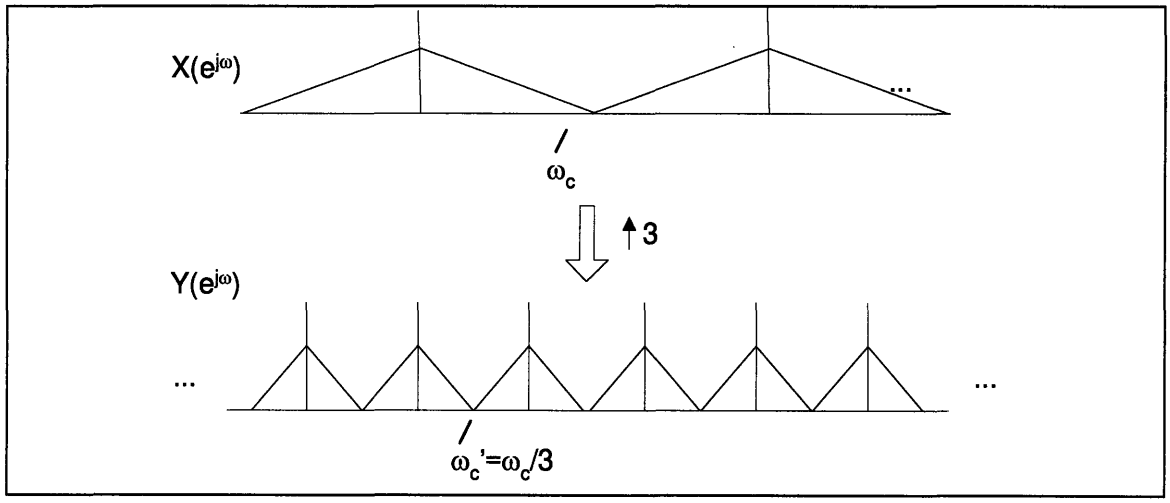

Figure 5-5: Upsampling by 3 Without Filtering

There is no aliasing so no information is lost in this procedure. There is, in fact, too much information, since the frequency spectrum which was centered around 2π is mapped to one centered around $2\pi/M$ and the one centered around 4π is mapped to one centered around $4\pi/M$, and so on. There are, in a manner of speaking, $M-1$ extra 'copies' of the frequency information within each period. These extra 'copies' must be eliminated by low-pass filtering $y[m]$ with a filter with cutoff frequency $\varpi_c = \pi/M$ to generate the desired properly upsampled version of $x[n]$, $y_{lp}[m]$. This process is outlined and its result shown in Figure 5-6 and Figure 5-7.

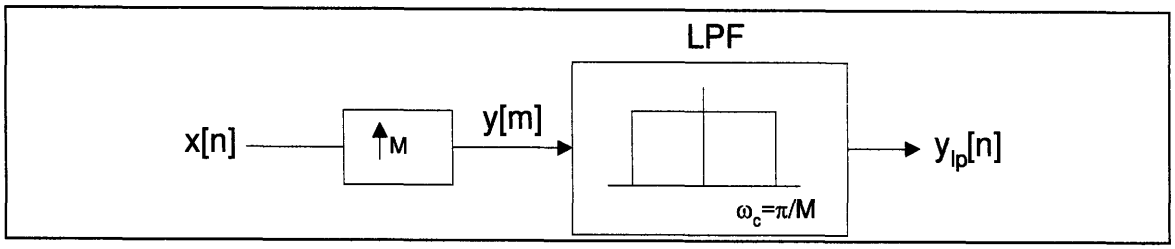

Figure 5-6: Block Diagram of Proper Upsampling

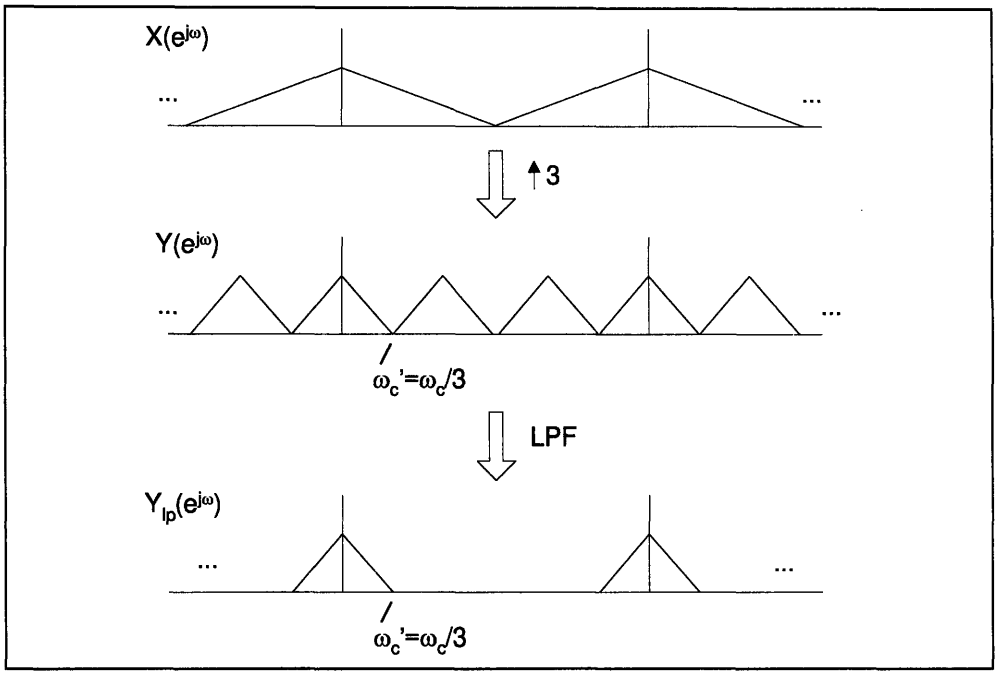

Figure 5-7: Upsampling by 3 With Proper Filtering

Taken together, upsampling and downsampling is a simple way to change the sampling frequency of a signal by rational fractions. The upsampling is done first because it is lossless; if the downsampling were done first information would be lost unnecessarily in the low-pass filtering stage preceding the downsampling. Also, this configuration allows the two low-pass filtering stages to be combined into one, with cutoff frequency $\varpi_c = \pi/\max(L, M)$.

5.1.1.2 Evaluation

While resizing pictures by upsampling and downsampling is simple, it has some serious drawbacks. The most significant of these is that, in this application at least, it is prohibitively inefficient.

The low-pass filter and downsampling stages can actually be combined if the filters used are finite impulse response (FIR) and are implemented by convolution in the time domain rather than by multiplication in the frequency domain, as they are in this case because it is significantly faster for the small filter kernel sizes used. There is no reason to compute all of the low-pass filtered points at the high sampling rate; only every *M*th point is actually used in the output, so only every *M*th point need be computed.

The upsampling part of the algorithm, however, causes a problem. Upsampling a line of video even by a small factor requires a large amount of memory; more than is available in a PP's data RAM. Upsampling by a large factor, such as 49 for an output picture size $49/50$ or 98% of the input picture size, becomes unmanageable.

Fortunately there is a way to combine the upsampling, filtering, and downsampling operations all into a single operation by using periodic filters.

5.1.2 Periodic Filters

A good way to explain periodic filters is through an example. Consider changing the sampling frequency of an input signal $x[n]$ by a factor of $2/3$, using a 5-tap FIR filter $h[n]$ with cutoff frequency $\varpi_c = \pi/3$ to do the low-pass filtering. One zero is inserted between each sample of $x[n]$ in the upsampling operation to make an intermediate signal $w[n']$. The combined low-pass filtering and downsampling operation then works

on $w[n']$ to compute the output signal, $y[m]$. When doing the convolution only half of the values of the filter $h[n]$ are used at any one time to compute an output pixel; the rest fall on the zeros inserted in the upsampling stage.

This leads to an equivalent but more efficient way to resample $x[n]$ by a factor of L/M. It is possible to skip the upsampling all together and compute the output samples by convolving the input with the values of $h[n]$ which would have fallen on the non-zero parts of $w[n']$ had $w[n']$ actually been computed. For the case where $L/M = 2/3$ this means that for alternate m's $y[m]$ is computed as a convolution of the appropriate part of $x[n]$ with either $h_1[n]$ or $h_2[n]$, where $h_1[n] = h[-2]\delta[n+1] + h[0]\delta[n] + h[2]\delta[n-1]$ and $h_2[n] = h[-1]\delta[n] + h[1]\delta[n-1]$. This analysis is shown in Figure 5-8.

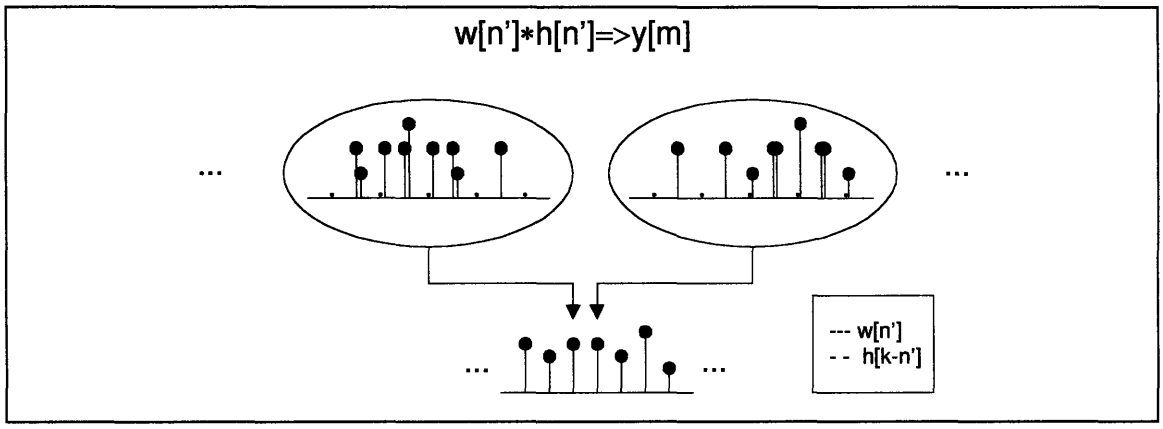

Figure 5-8: Example of Combining Resampling and Filtering Operations

Provided that the subfilters $h_i[n]$ are long enough each of them has roughly the same magnitude frequency response $|H_i(e^{j\varpi})|$. Exactly how well they are matched depends on their length and design. What is different about them, and what makes them able to interpolate between pixel values in addition to low-pass filtering them, is

their phase response $\angle H_i(e^{j\varpi})$ and group delay $grd[H_i(e^{j\varpi})]$, where group delay is defined as the negative first derivative of the unwrapped phase response. The group delay of a filter at a particular frequency ϖ can be interpreted as the number of samples the filter will delay the frequency component of the input lying at ϖ. For a filter to interpolate the value of a pixel which would be a distance *x* away from the pixel preceding it and a distance $1-x$ away from the pixel after it a filter ought to have a constant group delay of x in its pass band. The group delay in the stop band is irrelevant, since those frequencies are, by definition, attenuated.

An FIR filter $h[n]$ symmetric about $n=0$ has zero phase and, consequently, zero group delay. That same filter shifted to the right by one sample, $h'[n]-h[n-1]$, has a phase response with slope 1, and a group delay $grd[H'(e^{j\varpi})]=1$. The interpolating subfilters $h_i[n]$ are essentially shifted by fractions of a sample from each other. In the case where $L/M=2/3$ which was discussed above $h_1[n]$ has a group delay of zero and $h_2[n]$ has a group delay of one half so it can interpolate the value of a pixel exactly in the middle of two pixels.

5.1.2.1 Exact Polyphase Filter Approach

The exact polyphase filter approach is based on the periodic filtering described above. It takes advantage of the facts that the two low-pass filtering stages can be combined and that only the output samples which will actually be used need be computed. Exact polyphase filters are described in detail by Crochiere and Rabiner [CRO83] [LIM88] , and are summarized below.

Description

Changing the sampling frequency of a signal by a rational factor L/M is done by modifying the regular convolution operation, where output pixels are calculated as one filter $h[n]$ steps through the input signal $x[n]$, to an operation which computes output pixels by cycling through a succession of *L* filters $h_i[n]$, $i = 0,1,...,L-1$ as it steps through $x[n]$. *L* such filters are required because, since $L-1$ zeros would have been inserted in the upsampling stage if that stage were actually implemented there would be *L* different positions for which a unique set of points in $h[n]$ would not fall on the inserted zeros. Each of the *L* filters has roughly the same magnitude frequency response, that of a low-pass filter with cutoff frequency $\varpi_c = L/M$ if $M > L$, as it is for shrinking. Each of the L filters has a different phase response and group delay such that the group delay of each successive $h_i[n]$ has a group delay $1/L$ higher than the previous $h_i[n]$. Hence the appellation 'polyphase filters.'

Evaluation

This approach is quite effective as far as it goes. It is computationally efficient in that it only needs *Q* multiplies and adds per output sample, where *Q* is the number of taps in $h_i[n]$. Also, it skips the upsampling stage, so it does not use any more data memory than it takes to store an input signal.

The drawback is that the resampling capabilities of exact polyphase filters are limited to rational fractions. Furthermore, limitations in the size of the Parameter RAM of a PP restrict the rational fractions to be fairly simple. For each set of *L* *Q*-tap polyphase filters $Q \cdot L$ coefficients must be stored in each PP's Parameter RAM for the PP to be able to its work. For a fraction which will give the desired 0.1% or better resolution this is not possible. For example, a shrink factor of 81.1% would mean that $L = 811$ and

$M = 1000$. If Q is, say, 5, then 4 Kbytes are required to store the filter coefficients if they are 8 bits wide, and 8 Kbytes if they are 16 bits wide. A PP's Parameter RAM is 2 Kbytes, and only about 1.5 Kbytes are actually available to the user. Finally, even if the PPs' Parameter RAMs were larger the sheer number of coefficients that would have to be stored in off-chip memory in order to accommodate 0.1% or better resolution is prohibitive.

The exact polyphase filter approach, then, is one which is of limited use for all its elegance. It can only be implemented on the 'C80 for resampling video in rather coarse increments, like two or at most one percent. A similar but more attractive approach is that of using approximate polyphase filters.

5.1.2.2 Approximate Polyphase Filter Approach

The approximate polyphase filter approach is, again, an implementation of periodic polyphase filters. It differs from the exact polyphase approach in that it uses a fixed number of polyphase filters to do the low-pass filtering and resampling for a particular resampling factor.

Description

The filtering and resampling operation is again carried out using a set of periodic polyphase filters. However, the period or number of different filters used is fixed at L_f. This means that there are also a fixed number of pass band group delays, one associated with each filter. The range of the group delays is $-\left(\frac{L+1}{2L}\right),\ldots,-\frac{2}{L},-\frac{1}{L},0,\frac{1}{L},\frac{2}{L},\ldots,\frac{L-1}{2L}$, if $L = L_f$ is odd, and $-\left(\frac{1+L/2}{2L}\right),\ldots,-\frac{3}{2L},-\frac{1}{2L},\frac{1}{2L},\frac{3}{2L},\ldots,\frac{1+L/2}{2L}$, if L is even. So long as L is sufficiently large this is a good enough approximation to the fixed phase model that, to a human eye watching

video at least, the difference is indistinguishable. Regardless of its group delay each low-pass filter should have a cutoff frequency $\varpi_c = P$, where *P* is a non-rational resampling factor between zero and one.

The filtering and resampling convolution operation for this scheme is carried out in the following way: Regular convolution, $y[n] = \sum_{k=-\infty}^{\infty} x[k]h[n-k]$, can be seen as flipping a filter $h[n]$ and sliding it along the input signal $x[n]$ such that at each position the output value $y[n]$ is the sum of the dot product of $x[n]$ and the flipped $h[n]$ in the region that they overlap. In this case, the procedure can be seen as indexing through the input $x[n]$ and, at each position *k*, choosing the flipped version of a *Q*-tap filter $h_i[n]$ from the set of available filters *H* and setting the output value corresponding to that position $y[m]$ to the sum of its dot product with the part of $x[n]$ it overlaps when positioned at *k*. The position *k* is calculated from a running fraction rf_m, which is related to *P* by $rf_m = m/P$, such that $k = \lfloor rf_m \rfloor$. The fractional part of *rf*, $f = rf_m - \lfloor rf_m \rfloor$ is used to select the filter $h_i[n]$ to be used at position *k*.

Figure 5-9 shows a diagram of this process.

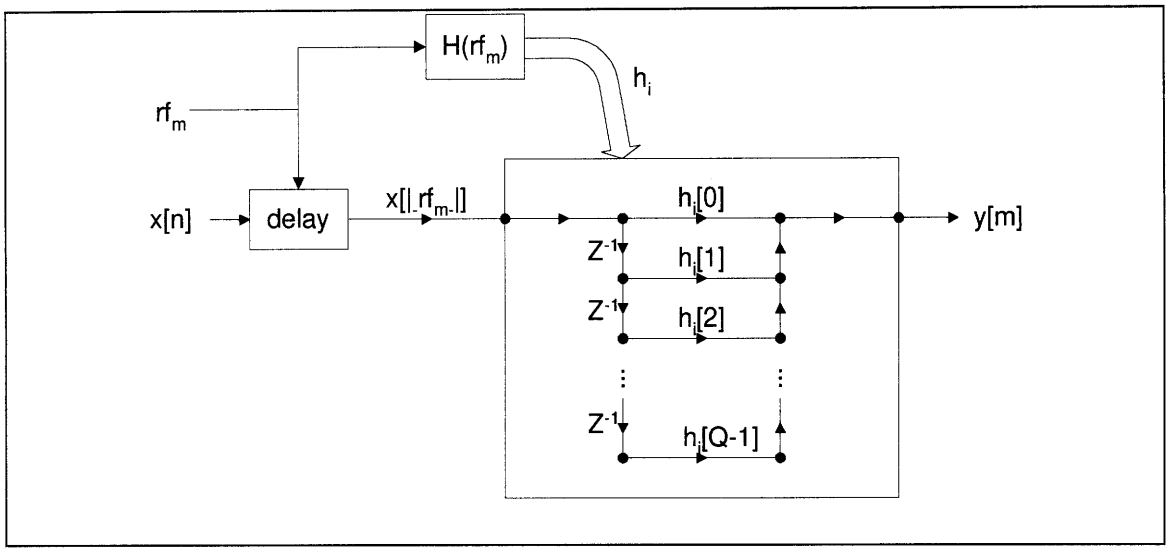

Figure 5-9: Diagram of Approximate Polyphase Filtering Process

Evaluation

The approximate polyphase filter approach is both effective and efficient. It uses a limited amount of memory space; only $L \cdot Q$ coefficients need be in a PP's Parameter RAM at a time, and *L* is a fixed number. Also, there is no need to store a separate set of filter coefficients in off-chip memory for each possible value of *P*. Each set of filter coefficients stored is good for several possible values of *P*. The set of possible values of *P*, then, is limited by neither memory space constraints nor by a need for it to be a rational fraction, only by the precision of the variables *rf* and *f*.

5.1.3 Filter then Interpolate

Another way to reduce the sampling rate of a signal $x[n]$ is to first low-pass filter it at its full sampling rate so as to prevent aliasing, and then to resample it by using some form of interpolation

To reduce the sampling rate by a factor *P* $x[n]$ is first low-pass filtered with a filter with cutoff frequency $\varpi_c = P \cdot \pi$ to make an intermediate signal $w[n]$. The

advantage here is that the filter used can be FIR or IIR, and there need be only one filter for a range of *P*'s; no need to complicate matters with a set of periodic polyphase filters. The disadvantage is that this filtering is done at the full sampling rate, which can mean that it requires a very large number of computations. The output signal $y[m]$ is interpolated from $w[n]$ using any of a number of methods, such as nearest neighbor, linear interpolation, cubic spline interpolation, or cubic convolution. Again, there is an advantage in terms of flexibility of the algorithm, and a disadvantage in terms of the computational expense of doing a second pass.

In general an FIR discrete time filter is easier to implement than an IIR filter. On the upside an IIR filter can come closer to the desired frequency response with a fewer number of coefficients than an FIR filter can. However, an FIR filter is inherently stable, whereas it is possible for an IIR filter to go unstable, even if it is designed properly, due to accumulated precision errors. Also, IIR filters require division, which can be computationally costly, while FIR filters are strictly multiplicative. Finally, it is not possible to vary the group delay of an IIR filter in a periodic fashion, so they cannot be used for simultaneous low-pass filtering and interpolation. All the low-pass filtered output samples must be computed first at the original sampling rate and then the signal must be resampled using some form of interpolation.

5.1.4 Half-Band Filters

The ideal half-band filter, or one with cutoff frequency $\varpi_c = \pi/2$,

$h_{ideal}[n] = \frac{\sin(\varpi_c n)}{\pi n} = \frac{\sin(\frac{1}{2}\pi n)}{n}$ is unique in that $h_{ideal}[n] = 0$ for all even values of *n* except $n = 0$. An FIR half-band filter based on $h_{ideal}[n]$, such as one designed by windowing $h_{ideal}[n]$, retains this property. A properly optimized convolution algorithm

can exploit the property to significantly cut down the computational expense of resampling a signal by a factor of $1/2$. A 13-tap half-band filter could be implemented using 7 multiply-adds, a 17-tap half-band filter could be implemented using 9 multiply-adds, and so on. If a half-band filter is used to resample a signal by a factor of $1/2$ there is no interpolation to be done. Each output sample maps directly to a particular input sample, so the operation can be viewed strictly as one of downsampling. This is both a benefit, because it makes for a very simple process, and a drawback, because it limits the use of half-band filters to resampling by factors of $1/2$ only.

5.1.5 Summary

Several possible methods of resampling video were presented and evaluated in this section: upsampling and downsampling, exact and approximate polyphase filtering, filtering followed by interpolation, and halfband filtering. Each of these methods has its own set of strengths and weaknesses.

The first two methods, upsampling and downsampling and exact polyphase filtering, are theoretically simple and accurate. However, the former method requires a large number of operations to perform the upsampling, and it requires extra memory space to store the upsampled intermediate signal before the downsampling operation is done. For a 1440 sample line of video data any upsampling requires more storage space than is available in a 2 Kbyte PP data RAM buffer. The extra manipulations necessary to properly manage these memory requirements would be difficult at best. Together with the large number of operations necessary the memory requirement issue makes pure upsampling and downsampling an unattractive option for resampling video on the 'C80.

The latter method requires a reasonable number of operations to implement and does not use any extra memory space. The down side is that it is limited to resampling by factors *P* which are rational fractions $P = L/M$. This poses a problem because, if *Q*-tap filters are to be used, $Q \cdot L$ coefficients need to be stored in each PP's parameter RAM in order to perform the processing efficiently. For large values of *L*, which are unavoidable if *P* is to have the desired 0.1% or better resolution, the number of coefficients needed becomes so large that it no longer fits in a PP's parameter RAM. Furthermore, a separate set of coefficients has to be kept in off chip memory for every possible value of *P*. For these reasons exact polyphase filtering also is not suitable for implementation on the 'C80.

The other three algorithms described, approximate polyphase filtering, filtering and then interpolating, and halfband filtering, are more reasonable approaches to implementing video resampling on the 'C80. The number of operations required to perform the necessary calculations is manageable. It is different for different combinations of the three algorithms and under for different values of relevant parameters, as will be discussed in the next section. Memory space is also manageable: of the three only the filter and interpolate approach requires extra memory to store intermediate results, and even that is the same amount of memory as is taken up by the input data; also the number of coefficients stored is fixed to at most $Q \cdot L_f$, where L_f is the number of filters used in approximate polyphase filtering. Finally, the resampling factor *P* is not limited to rational fractions.

The figures below show an example of why the last three algorithms are better suited to implementation than the first two. The data presented is based upon calculations done according to

Table 5-1, where $K \times N = 486 \times 1440$ is the size of an input picture, $Q = 5$ is the number of coefficients per filter, *P* is the resampling factor, defined as $P = L/M$, $L_f = 16$ is the number of filters used in approximate polyphase filtering, and $Q' = 3$ is the number of non-zero coefficients in a halfband filter of length *Q*. Four different values of *P*, of varying complexity, are used. *P* is constrained to be a rational number so that a meaningful comparison can be made between algorithms which require such a constraint and those which do not.

	# Operations, assuming no overhead	Extra Memory Required (in bytes)	Memory Required for Coefficients (in bytes)
Upsample & Downsample	LN+QNP	LN	2Q
Exact Polyphase	QNP	0	2QL
Approximate Polyphase	QNP	0	$2QL_f$
Filter & Interpolate	QN+PN	N	2Q
Halfband	Q'PN	0	2Q'

Table 5-1: Calculations for Comparing Resampling Methods

Figure 5-10 shows that upsampling and downsampling is significantly more computationally intensive than the other algorithms. Figure 5-10: Comparison of Resampling Methods in Terms of Number of Calculations

shows that while both upsampling and downsampling and filtering and interpolating require extra memory space to store intermediate values, the former algorithm does so in a dynamic way and can easily require much more space than the latter algorithm.

Figure 5-11: Comparison of Resampling Methods in Terms of Data Memory Requirements

shows that the exact polyphase filter approach can require much more memory space than is possible for storage of filter coefficients in the PP's parameter RAMs.

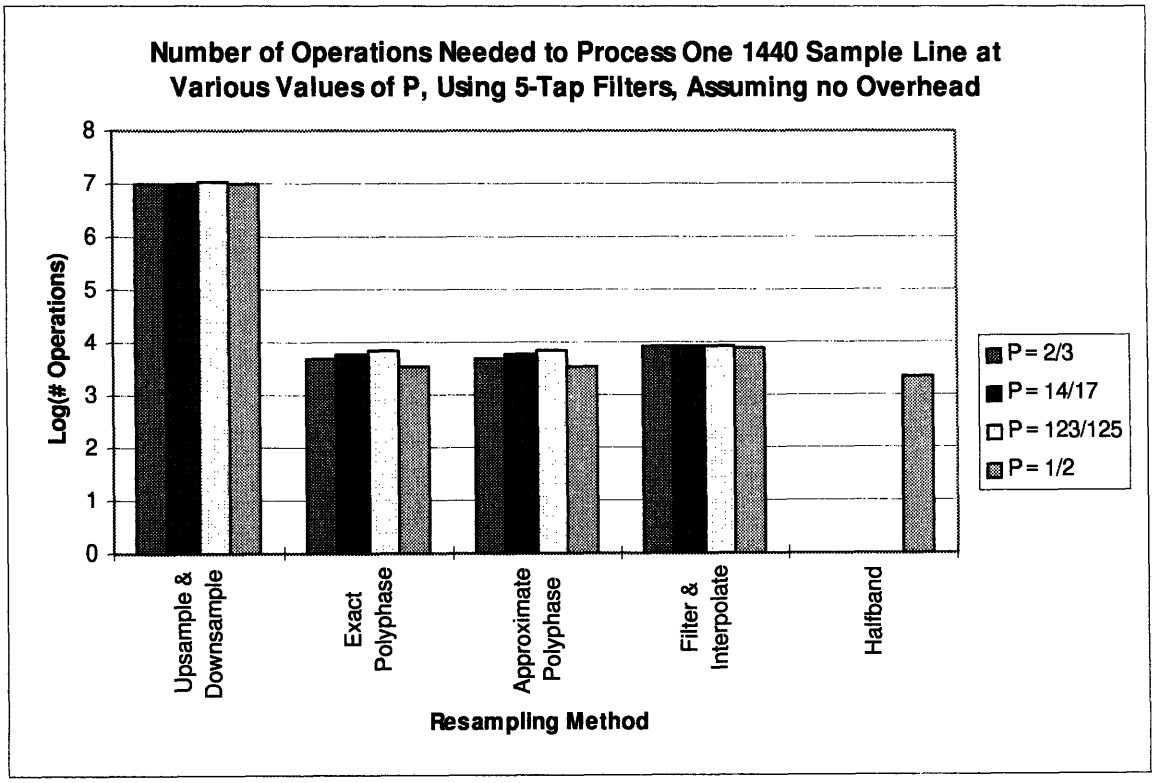

Figure 5-10: Comparison of Resampling Methods in Terms of Number of Calculations

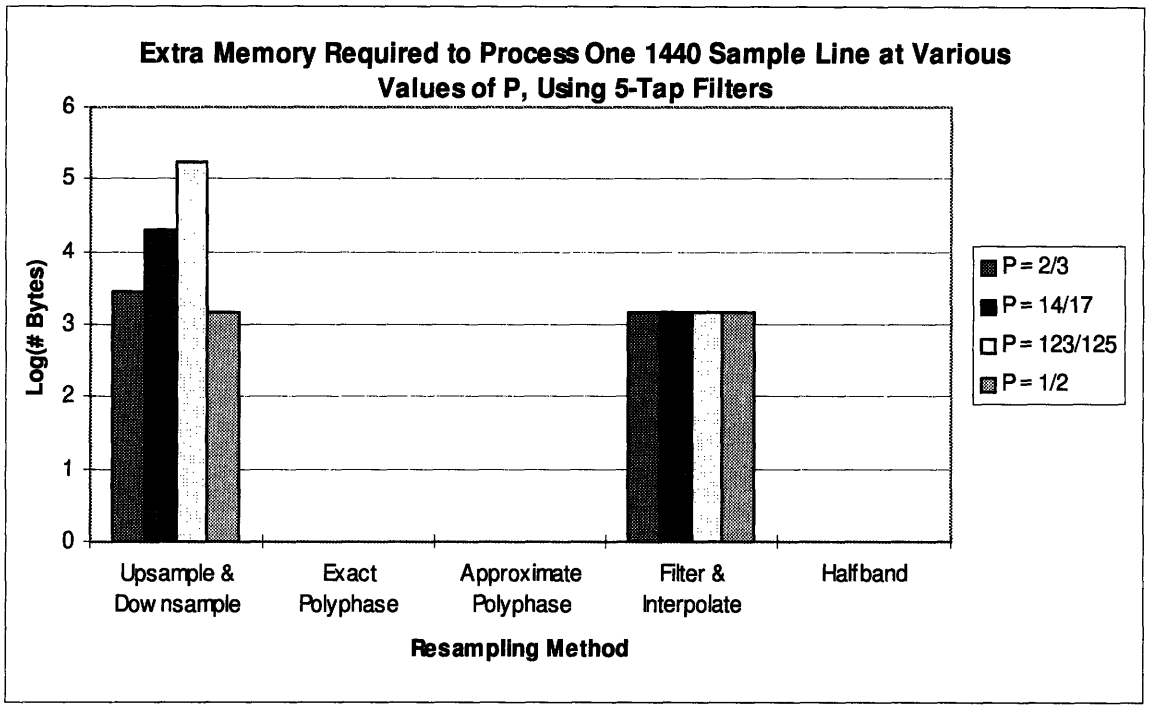

Figure 5-11: Comparison of Resampling Methods in Terms of Data Memory Requirements

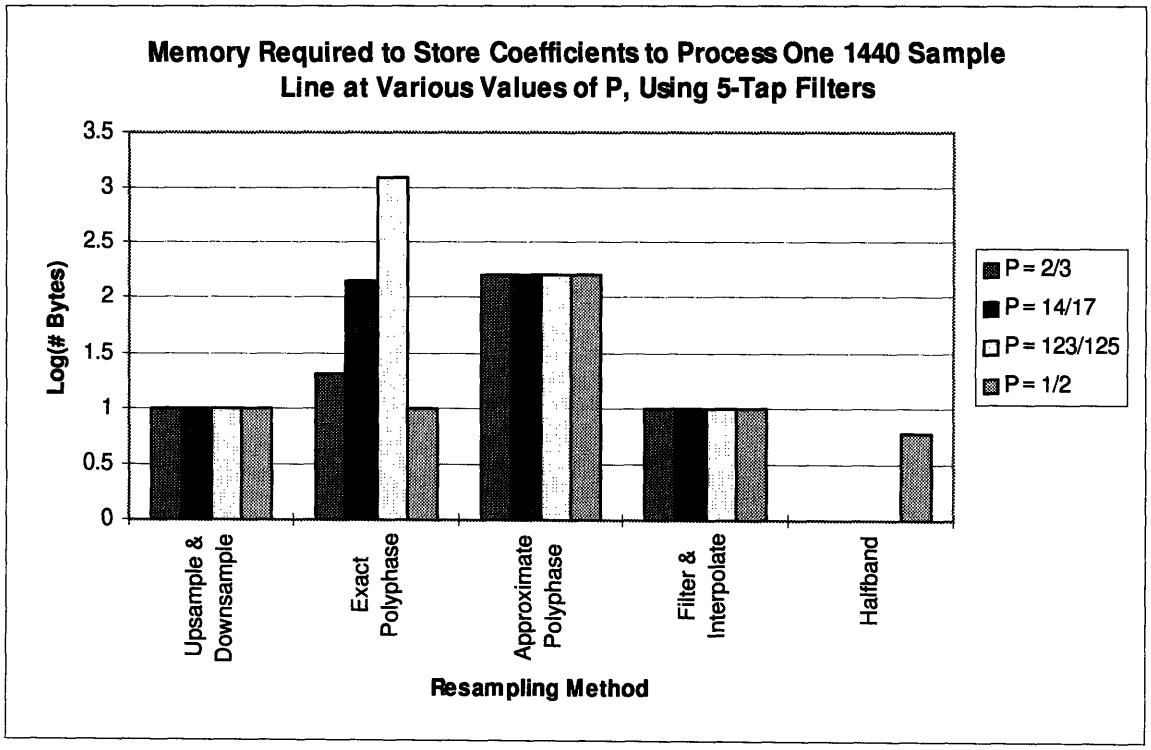

Figure 5-12: Comparison of Resampling Methods in Terms of Parameter Memory Requirements

5.2 Final Choice of Algorithms

One or a combination of the above algorithms needed to be mapped to the 'C80 for real-time operation in order to achieve the desired video shrinking effect. After examining the performance goals of the effect and the computational constraints of the 'C80 and the development hardware a multi-pass algorithm was decided upon.

5.2.1 Computational Constraints

Separating the resampling of an image into two stages, one horizontal and one vertical as shown in

Figure 5-13, offers a significant computational savings for the second stage. For an input image of $n \times m$ samples, a resampling factor *P*, and assuming that it takes *C* processor cycles to compute an output sample in each stage, the number of cycles necessary to complete the first stage is $C \cdot (P \cdot n \cdot m)$ and the number of cycles necessary to complete the second stage is $C \cdot (P \cdot n \cdot P \cdot m)$, for a total number of cycles $C_T = C \cdot n \cdot m \cdot (P + P^2)$. This means that the number of cycles available for computing an output sample in each stage in real time is

$$C_a = \left(\frac{1}{30\,\text{sec}}\right)\left(\frac{1}{n \cdot m \cdot (P + P^2)\,operations}\right)\left(\frac{X\ cycles\ /\ processor}{\text{sec}}\right)\left(N_p\ processors\right).$$

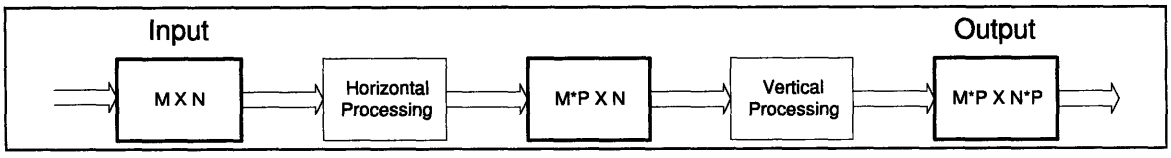

Figure 5-13: Two Stage Processing: Horizontal, Then Vertical

5.2.1.1 'C80

If a frame of video has $n = 1440$ samples per line and $m = 486$ lines and the processor used is a 'C80 with $N_p = 4$ and $X = 50\,MHz$ then $C_a = \frac{9.5260}{(P + P^2)}$. A plot of C_a versus P is shown in Figure 5-14 and Figure 5-15.

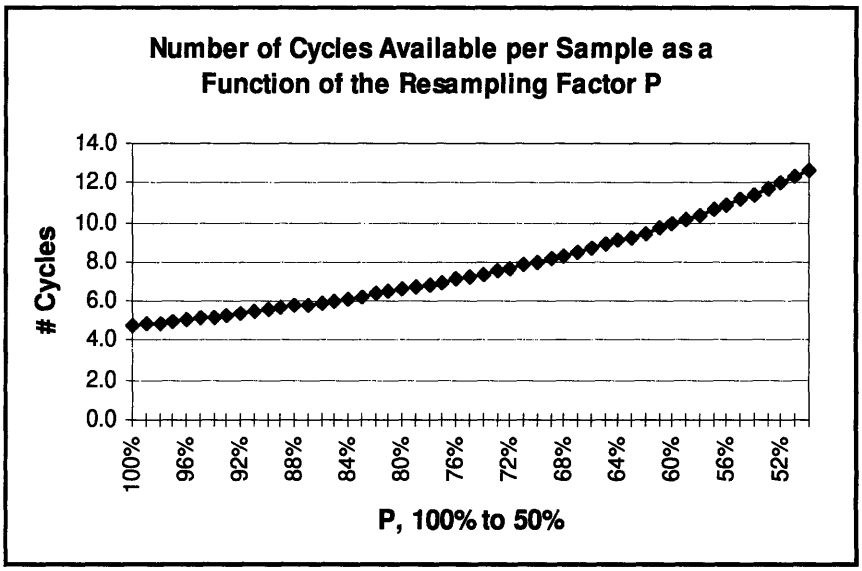

Figure 5-14: Cycles Per Output Sample

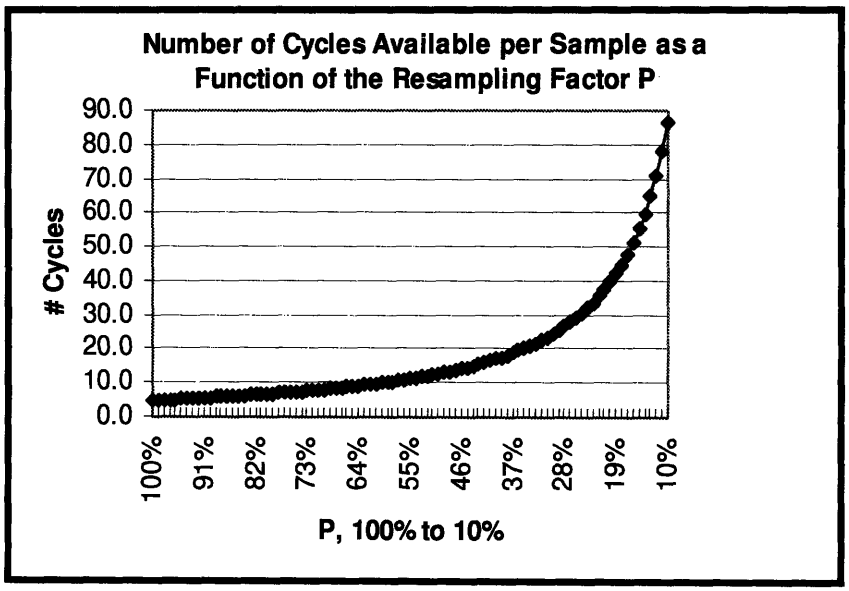

Figure 5-15: Cycles Per Output Sample (larger range of P)

As the above plots indicate the number of cycles available to calculate an output sample in each stage of processing is most limited when the resampling factor *P* is closest to 100%, to as few as 4.8 cycles per sample per stage at 100%, and increases quickly as *P* becomes very small, to as many as 86.6 cycles per sample per stage at 10%. This means that it is computationally feasible to use increasingly large filters as the value of *P* decreases. It is desirable to do so, to a certain point, because as the value of *P* decreases aliasing poses a greater problem so better, and therefore longer, filters need to be used.

The architecture of the 'C80 and the development hardware encourages resampling horizontally first and vertically second, as opposed to the other way around. The two ways are computationally equivalent so long as the same filtering technique is used processing in each direction, but the former is better suited in terms of data transfer. Video frames are stored as lines in the input memory bank. For horizontal processing all the Transfer Controller needs to do to copy a line of data to a PP's data RAM is to grab it whole and copy it over. For vertical processing the Transfer Controller must use an extra dimension and grab *k* byte blocks from each of a series of lines and copy them into a PP's data RAM as a line. An example of how a vertical line of a 4×6 pixel image is mapped to a PP data RAM with $k = 4$, is shown in

Figure 5-16. If horizontal resampling is done first there are fewer vertical lines to transfer around in the vertical resampling stage, making for a more efficient algorithm. For the program that was written vertical columns were $k = 8$ bytes wide.

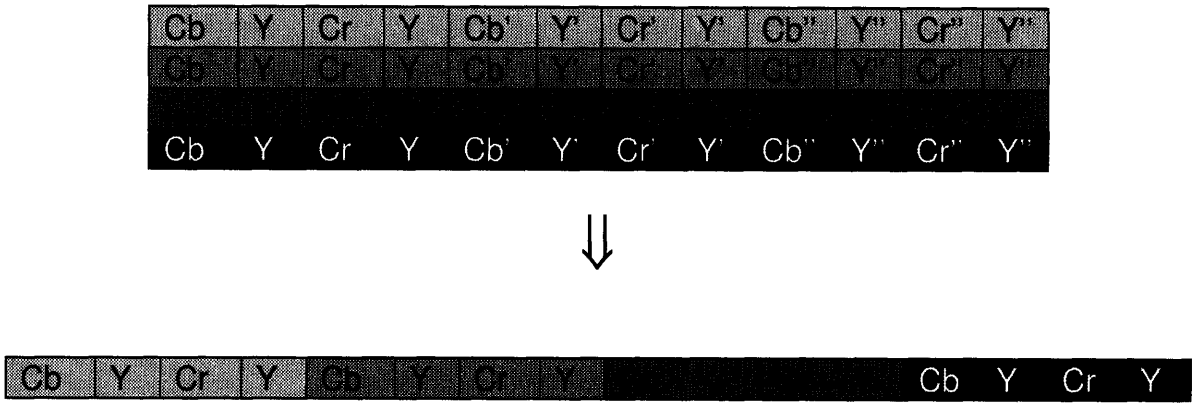

Figure 5-16: Mapping of a 4-byte Wide Column of Data to a PP's Memory

Finally there is the issue of memory space, which was touched upon in the discussion of polyphase filters. Each PP has 2 Kbytes of Parameter RAM, of which the first 0.5 Kbyte is reserved for use Transfer Controller and for handling interrupt vectors. Also, the PP stack grows from the bottom of the Parameter RAM, so only the remaining 1.5 Kbytes minus the stack size are available to actually store parameters. At the same time the PPs must have all of their parameters available locally in order to operate efficiently. Consequently it is not feasible to implement any algorithm which requires a large amount of memory to store filter coefficients, such as the exact polyphase filter algorithm.

5.2.1.2 Assumptions

Since the 'C80 PP can do a multiply, an accumulate, a data fetch and store, an instruction fetch, and pointer updates in a single cycle even 4.8 cycles per output sample per stage is enough to do a reasonable number of calculations. The remainder of this thesis project relies on the assumptions that PP code can be optimized to implement a *Q*-tap FIR filter in $Q+1$ cycles, and to implement a half-band filter with Q_{hb} non-zero coefficients in Q_{hb} cycles. These assumptions were decided to be reasonable after consulting with local 'C80 programmers.

5.2.2 Multi-Pass Algorithm

After reviewing the computational, I/O, and memory constraints of the 'C80 and the current development hardware a multi-pass algorithm was decided upon for implementing the video resizing effect. This algorithm is diagrammed in Figure 5-17.

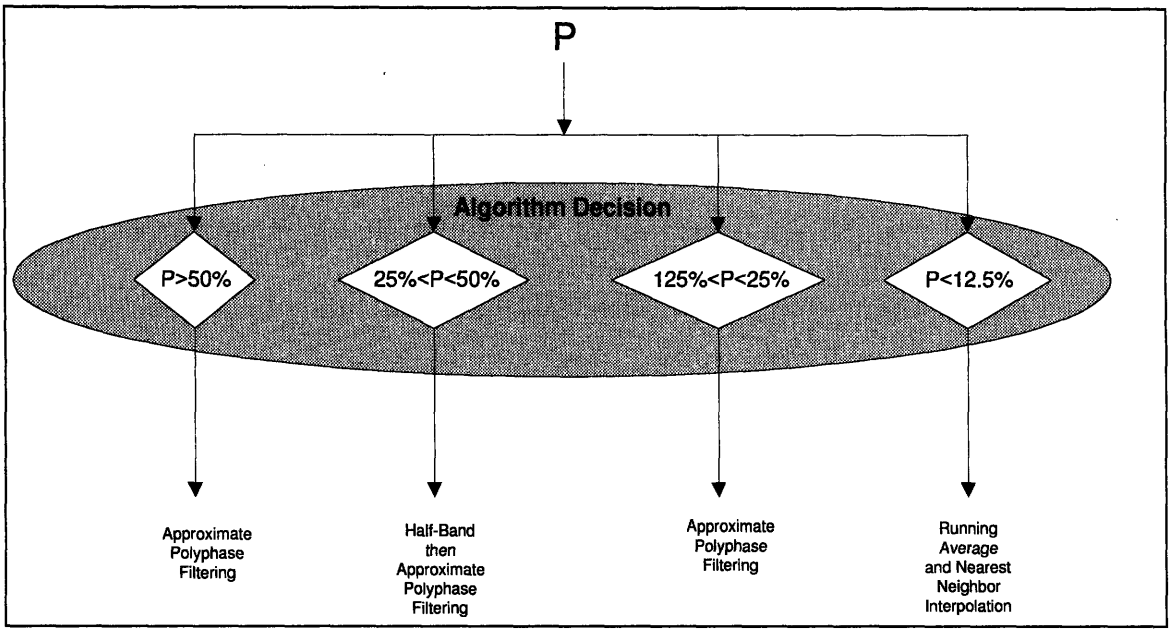

Figure 5-17: Algorithm Decision Tree

The resampling problem is treated differently depending on the value of the resampling factor *P*. The range of values of *P* that are dealt with in this thesis is $25\% < P \leq 100\%$. Methods for resampling video for smaller values of *P* are considered, but not implemented. For the range where $50\% < P < 100\%$ a single pass of approximate polyphase filtering is necessary. If $25\% < P < 50\%$ the picture is first downsampled to half its size in each direction using a half-band filter and then is resampled by a factor $P_{eff} = 2P$ using approximate polyphase filtering. Two methods of handling the case where $P < 25\%$ are proposed, going back to approximate polyphase

filtering and using a combination of running average filtering and nearest neighbor interpolation.

The reason the problem is broken up this way is that switching methods offers computational savings, as will be discussed in the sections that follow. No one resampling method is best for all values of *P*, so an efficient implementation of resampling switches between different methods as is appropriate.

5.2.2.1 P>50%: Single Pass of Approximate Polyphase Filtering

Approximate polyphase filtering, used in the first and only pass of the resampling algorithm if $P > 50\%$ and in the final pass if $12.5\% < P < 50\%$, is used in the following form: *L* *Q*-tap polyphase filters are designed and used for each of several ranges of *P*. *L* is fixed at 16 because subpixel resolution of at most one tenth of a pixel is required and because it is most convenient to implement a value of *L* which is a power of two. The number of taps *Q* in a set of polyphase filters is constant across the set of filters because it is significantly simpler to implement fixed rather than variable length filters. For each range of *P* *Q* is chosen such that $Q+1$ is less than the number of cycles available per sample per pass for calculation in the case that $P > 50\%$. The same set of filters is used for the final pass in the case that $12.5\% < P < 50\%$, so that as *P* decreases the resampling process begins to free up some of the processor. The ranges of *P* and their corresponding *Q* values are shown in Figure 5-18.

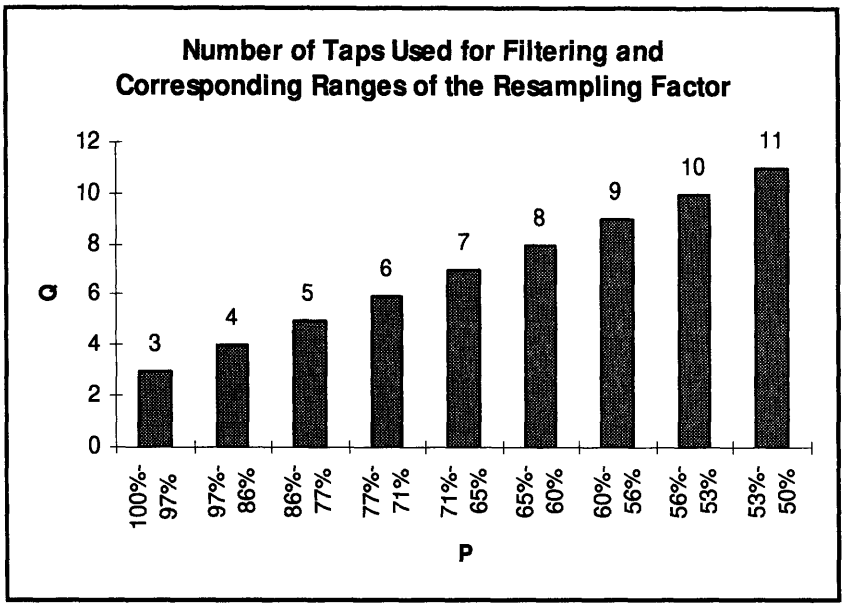

Figure 5-18: Relationship of Filter Size to Resampling Factor

5.2.2.2 25%<P<50%: Combined Half-Band and Approximate Polyphase Filtering

If $25\% < P < 50\%$ a frame of video is resampled in two steps. The first step, downsampling by a factor of $1/2$, efficiently if coarsely reduces the size of the input picture. A half-band filter with $Q = 13$ taps is used for this step. Assuming a highly optimized filter, this step should take 63% of the processor. The second step, approximate polyphase filtering with $P_{eff} = 2P$, fine tunes the results of the first step to effect resampling by the original factor *P*. Such a two-step algorithm results in a net savings in computation time over a single approximate polyphase filtering algorithm with equivalent performance. For example if $P = 45\%$ the two step algorithm, using 5-tap polyphase filters in the second step, takes 55% of the processor. A one step algorithm with equivalent performance, 15 taps, since the first and last taps of the half-band filter are zero, would take 96% of the processor. Although the difference is not extremely large it is significant because the single step algorithm is right on the edge of being able

to run real time, while the two step algorithm has a bit of a safety margin. The theoretical relative performance characteristics of the two-step and equivalent one-step processes are shown in Figure 5-19. The two traces do cross at $P \approx 30\%$, but the difference between them never becomes very large beyond that point.

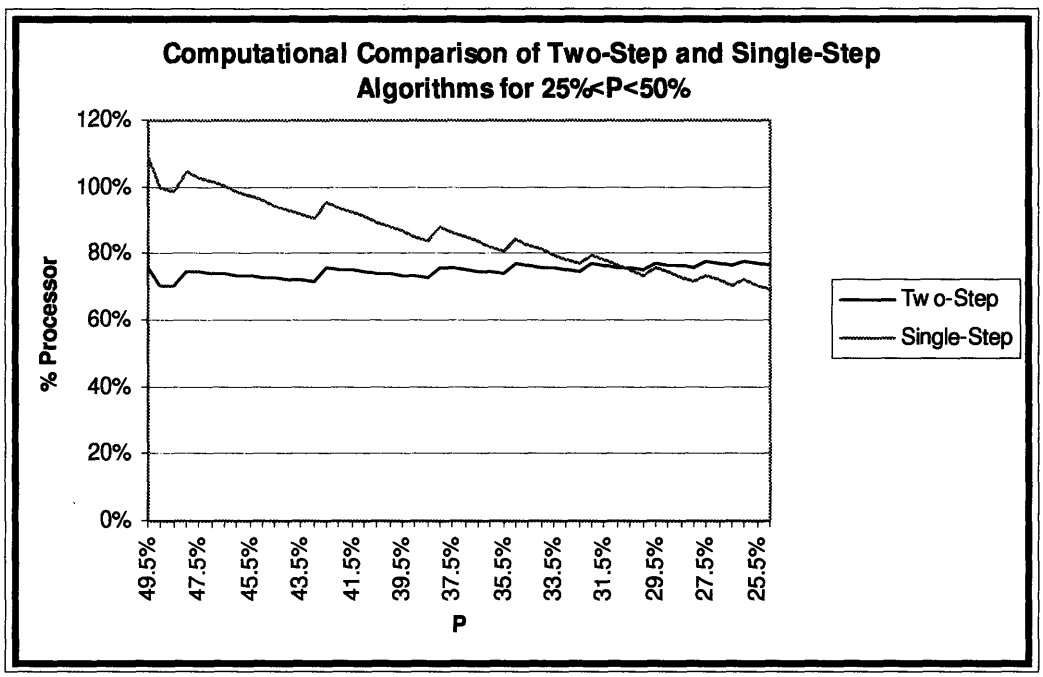

Figure 5-19: Processor Use for 12%<P<50%

5.2.2.3 12.5%<P<25%: Approximate Polyphase Filtering

One way to handle the case where $12.5\% < P < 25\%$ would be to go through the half-band filter and decimate step twice to achieve a one quarter reduction in size in each direction, and then to use approximate polyphase filtering to fine tune the resampling to the desired value of *P*. Such a three-step algorithm is a simple solution, but not necessarily the best one possible. For a 13-tap half-band filter the first two steps use up 69% of the processor. Not much additional processing time is required for the third step, but using an equivalent single-step approximate polyphase filtering algorithm quickly becomes more efficient, as shown in Figure 5-20. Furthermore, as the output

picture size becomes very small its quality becomes less important; a 31-tap filter for an output picture 12.5% the size of the input picture in each direction is overkill. Consequently, a return to the original single-step approximate polyphase filter approach is proposed for the case where $12.5\% < P < 25\%$.

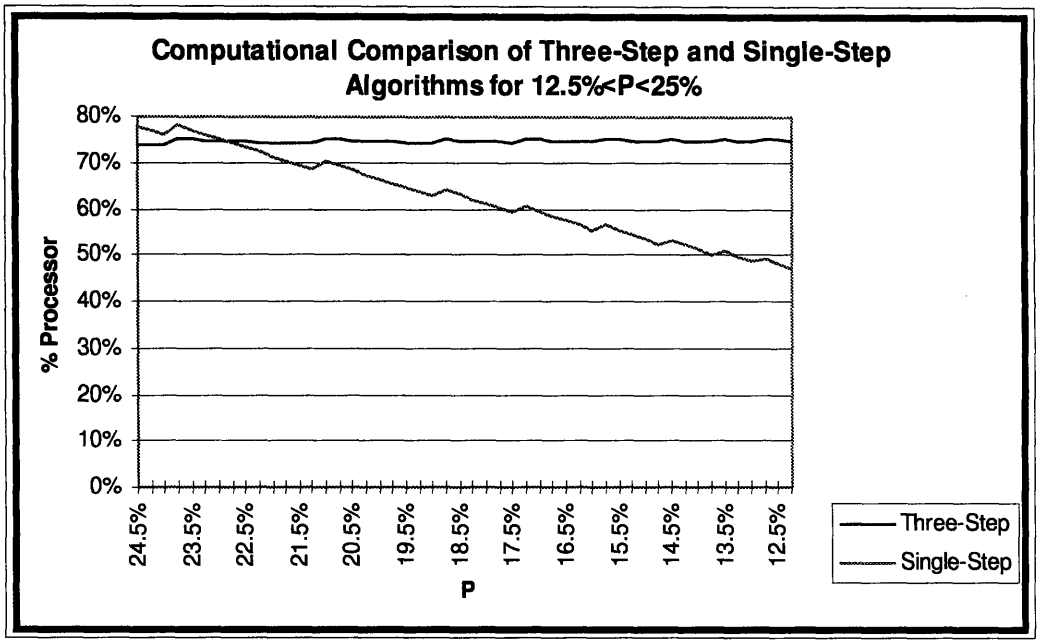

Figure 5-20: Processor Use for 12.5%<P<25%

5.2.2.4 P<12.5%: Running Average Filter then Interpolate

The filter and then interpolate algorithm is proposed for very small output pictures, such as ones below 12.5% of the original size along each axis. A simple filter, an FIR box or running average filter is proposed. Such a filter is far from ideal--the frequency response of a 10-point filter, shown in Figure 5-21, has sidelobes almost a quarter of the height of the main lobe and an ill defined pass band and wide transition region--but for an output picture of 60×90 pixels or smaller it ought to be perfectly adequate. Similarly, nearest neighbor interpolation would look terrible on a large full-resolution picture, but should be adequate for a small, low frequency picture. Taken

together running average filtering and nearest neighbor interpolation give small output pictures of reasonable quality at small computational expense.

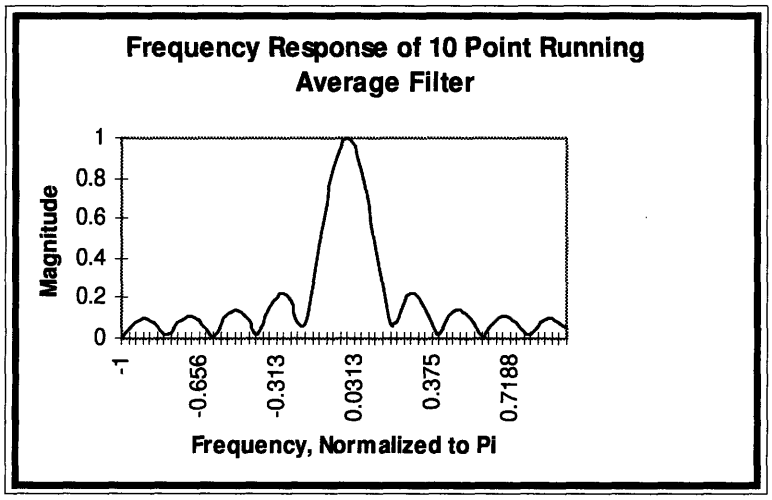

Figure 5-21: Running Average Filter

5.2.2.5 Edge Effects

Anytime a finite length signal is filtered to produce another finite length signal the question of how to handle the edges comes up. Any non-degenerate filter will introduce errors in the first and last few samples of the output. For example, if an FIR filter with an odd number Q of coefficients is convolved with an input signal $x[n]$ defined on some interval $n = 0,1,\ldots,N$ to produce a signal $y[n]$ defined on some interval $n = 0,1,\ldots,M$ the first $K = \frac{Q-1}{2}$ samples of $y[n]$ will be calculated from unknown samples of $x[n]$ where $n < 0$. The same effect occurs at the other end, where $y[n]$ depends on values of $x[n]$ for $n > N$. There are three standard techniques used to handle this problem: zero padding, repetition, and reflection. Each of these techniques assigns values to $x[n]$ in the unknown regions $n = -K,-K+1,\ldots-1$ and $n = N+1,N+2,\ldots N+K$. Zero padding sets $x[n]$ to zero in both these regions,

replication assigns $x[n] = x[0]$ in the former and $x[n] = x[N]$ in the latter region, and reflection assigns $x[n]$ in the unknown regions values which are the reflection of the values of the defined portion of $x[n]$ about $n = 0$ and about $n = N$. Of these three methods zero padding was chosen for this project because, although produces less pleasing results than the other two, it is the simplest to implement. Also, since the number of coefficients used in the filtering operations increases only as the ratio of input to output picture size increases any artifacts caused by the zero padding are guaranteed to remain in a small number of pixels at the edges of the output picture.

5.3 Filter Design

A good design makes compromises among the desirable features in such a way that the resulting set of filters produces the best looking output video. Design of both periodic and half-band filters is discussed in this section.

5.3.1 Periodic Filter Design

A set of periodic filters can be designed either by sampling a prototype filter at different offsets or by attempting to specify both the desired frequency and phase response for each of the filters in a set separately. Beyond that they can be designed using several methods including the Remez-exchange algorithm and windowing. The actual design methodology was based on how well it facilitated meeting the design criteria and on its simplicity of use.

5.3.1.1 Desirable Features of Filters

A set of polyphase filters has several desirable features, some of which are particularly important for video. Among these are DC gain, matching of magnitude frequency response, group delay, transition band, and pass-band and stop-band ripple.

Perfection, of course, is not possible. There are many trade-offs to be made; the sharper the transition band, the larger the pass-band ripple, and so on. How these trade-offs are made depends on the relative importance of the desirable features of a set of filters for a particular application. For video the first three features listed are of particular importance.

The DC gain of each filter $h_i[n]$ of a set *H* must be equal to exactly one. The case of a flat field test signal illustrates why this is such a hard requirement. If the input picture is uniformly gray the output picture should also be uniformly gray. With 8 bit video even one bit's difference between the two is noticeable, and unacceptable.

The pass-band group delays of a set of filters are important because they are the mechanism of the interpolation. The trade-off is that for a small number of taps designing a set of filters with constant and correct group delays affects the possibility of matching their magnitude frequency responses. At the same time, the magnitudes of the frequency responses of each of the $h_i[n]$s in a set *H* need to be the same. Otherwise some of the filters do a better job of low-pass filtering and interpolation than others. The result is that a high frequency object at one position on the input picture will look different on the output picture than the same object at a different position on the input picture. A high frequency object moving across the screen, for example, will appear to blink in and out as it periodically becomes more and less blurry. A sinusoidal input whose frequency falls in the pass-band of some of the $h_i[n]$s, in the transition band of some others, and in the stop-band of still others will be attenuated more and less in a periodic manner. This effect is especially problematic because the Cathode Ray Tubes (CRT's) used to display video have non-linear gain due to what is known as gamma correction. In general the higher value of a pixel the higher the gain, so a

modulated monochrome sinusoid with a DC value of gray actually appears to be a modulated sinusoid with a DC value which oscillates at the modulation frequency, as shown in Figure 5-22.

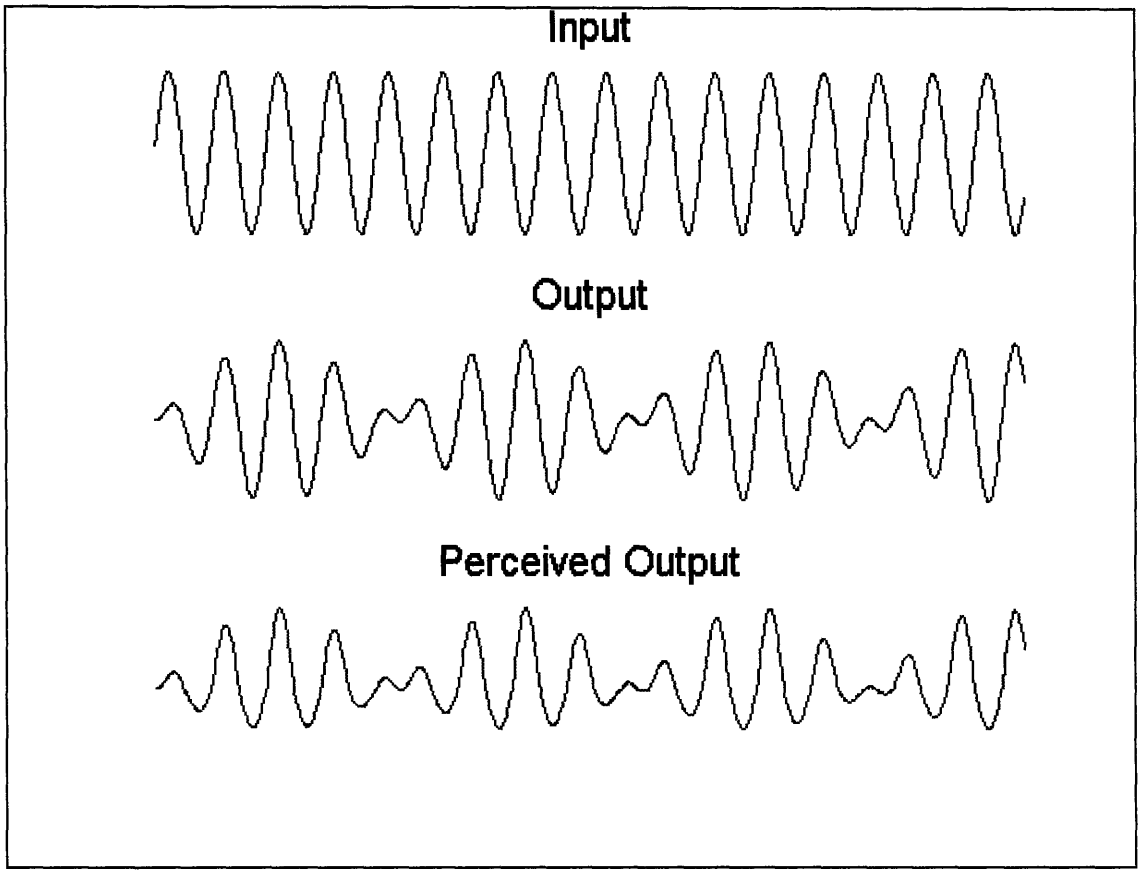

Figure 5-22: Effects of Periodic Filters With Mismatched Magnitude Frequency Responses

The magnitude of the pass-band ripples must be less than 5%. Anything larger will be noticeable to a careful observer. Also, all output values must be between 0 and 255, and ought to fit in the 16-235 for luminance and 16-240 for chrominance valid range of 8-bit video. Large overshoots will cause values which are significantly out of the valid range, which not only looks wrong but causes errors in any subsequent processing of the same video. The down side of designing for such small ripple is that it necessitates wide transition regions.

5.3.1.2 Sampled Prototype Filter

One way to design *L* *Q*-tap periodic polyphase filters with cutoff frequency $\varpi_c = P$ is to design a single $L \cdot Q$ tap prototype filter $h_p[n]$ with cutoff frequency $\varpi_c = P/L$, and to sample that filter to create the individual polyphase filters $h_i[m] = h_p[mL + i]$. This works quite well so long as the magnitude of the frequency response of $h_p[n]$ is sufficiently close to zero above $\varpi_L = 1/L$. If it is not then sampling $h_p[n]$ will in fact cause aliasing in the subfilters. This aliasing will be worse for some of the subfilters than for the others, depending on the value of *i*, causing a set of poorly matched filters.

5.3.1.3 N L Point Filters

Another approach is to design each filter of a set *H* separately. This can be done by designing a single *Q*-tap zero-phase filter with cutoff frequency $\varpi_c = P$ and interpolating the individual $h_i[n]$s from it. Ideally the interpolation could be done by convolving the impulse response of the zero-phase filter $h_0[n]$ with an ideal interpolator such that $h_i[n] = \sum_{k=-\infty}^{\infty} h_0[k] \frac{\sin[\pi(n - \alpha_i - k)]}{\pi(n - \alpha_i - k)}$, where α_i is the desired group delay of $h_i[n]$. Unfortunately the $h_i[n]$s designed in this way have infinite impulse responses. If they are truncated to *Q* coefficients their performance suffers. They become just as difficult to match as $h_i[n]$s designed by sampling a prototype $h_p[n]$.

Each *Q*-tap $h_i[n]$ could also be designed by specifying both its desired magnitude frequency response and its desired group delay in the design algorithm. No suitable design algorithm was found to do this, however.

5.3.2 Filter Design Techniques

Filters were designed using the sampled prototype filter technique because it proved to be simpler than designing N L point filters: in both cases only one filter needed to be designed, but sampling one large one was somewhat easier to do than doing a lot of fractional sample delay convolutions.

Several design methods were considered, including the Remez-exchange algorithm and windowing. After evaluating these methods windowing was used.

5.3.2.1 Remez Exchange

The Remez Exchange algorithm can be used to design optimal filters, where 'optimal' means that the maximum error from the desired filter is minimized. This algorithm works quite well, provided that the number of coefficients in a filter is sufficiently large. If, however, the design problem is as highly constrained as it is here, the Remez-Exchange algorithm breaks down. According to the alternation theorem[OPP89] a mini-max optimized filter with a small number of taps has an even smaller (roughly half) number of alternations, with all the ripples in the pass band being of equal size and all the ripples in the stop band being of equal size. The small numbers of taps being used in this implementation and the design goal of significant attenuation by $\varpi = 1/L$ in the prototype filter mean that a prototype filter designed using the Remez-Exchange algorithm typically has unacceptably large ripples. Also, as the algorithm iterates through its design procedure it ends up resorting to using coefficient values of unreasonably large magnitudes at the edges of the filter in order to meet at least some of the design constraints. Consequently the sub-filters made by sampling the prototype filter can be severely mismatched; some are actually high pass filters. Figure 5-23 shows an example of the time and frequency response of an 80 tap

prototype filter designed with $P = 80\%$ and the frequency responses of the 16 5 tap subfilters made by subsampling it. The Remez-Exchange algorithm was not used for filter design in this project because of its poor performance for the kind of overconstrained filters that are necessary.

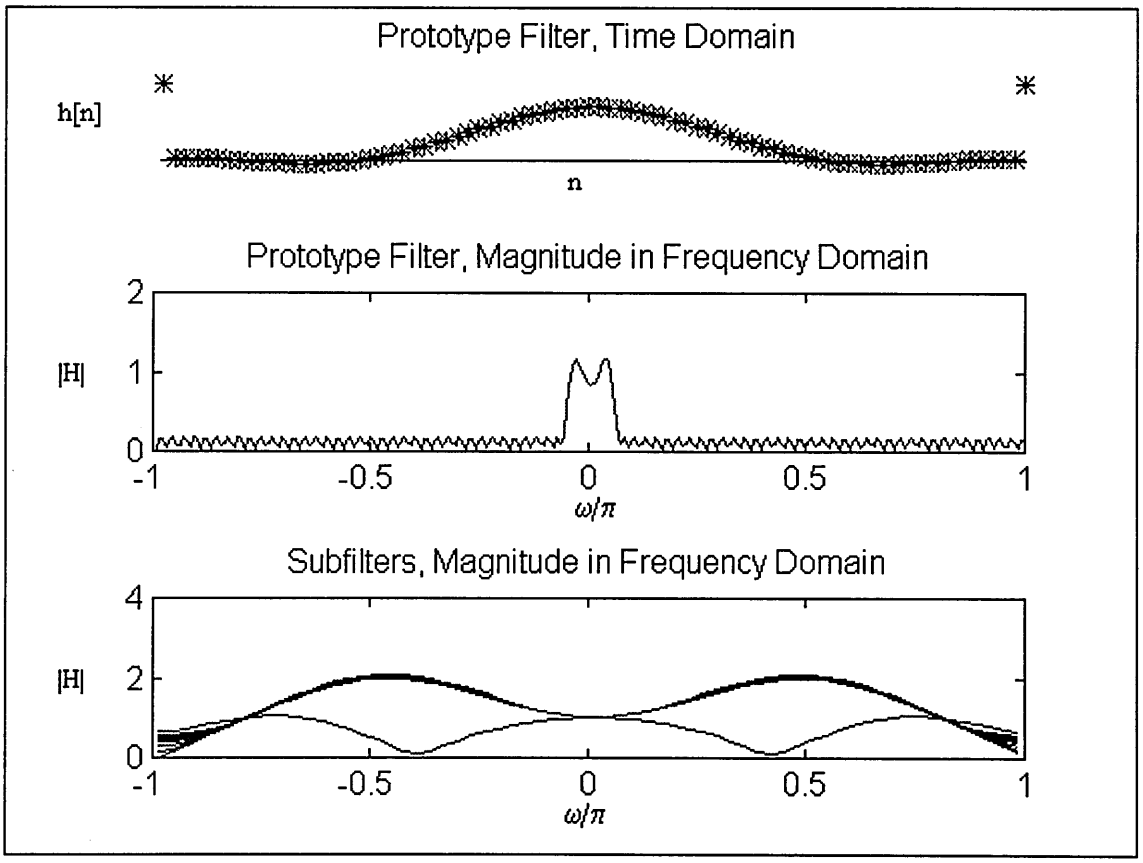

Figure 5-23: Example of Overconstrained Filter Designed Using the Remez-Exchange Algorithm

5.3.2.2 Windowing

Filters designed by windowing are ideal $\sin x/x$ filters multiplied by a window such that $h[n] = h_{ideal}[n]w[n]$. The reason for using the window $w[n]$ is to attenuate the ripples caused by the Gibbs Phenomenon. Several different types of windows can be used. Among them are Boxcar, Hamming, Hanning, Bartlett, Blackman, and Kaiser windows. Essentially the Boxcar window does nothing at all, the Hanning and Hamming

windows are cosine shaped, the Bartlett is a triangle, and the Blackman is a sum of two cosines. These windows are defined as follows for filters of length N+1 [OPP89] :

Boxcar:
$$w[n] = \begin{cases} 1, & 0 \le n \le N \\ 0, & otherwise \end{cases}$$

Hamming:
$$w[n] = \begin{cases} 0.54 - 0.46\cos(2\pi n/N), & 0 \le n \le N \\ 0, & otherwise \end{cases}$$

Hanning:
$$w[n] = \begin{cases} 0.5 - 0.5\cos(2\pi n/N), & 0 \le n \le N \\ 0, & otherwise \end{cases}$$

Bartlett:
$$w[n] = \begin{cases} 2n/N, & 0 \le n \le N/2 \\ 2 - 2n/N, & N/2 < n \le N \\ 0, & otherwise \end{cases}$$

Blackman:
$$w[n] = \begin{cases} 0.42 - 0.5\cos(2\pi n/N) + 0.08\cos(4\pi n/N), & 0 \le n \le N \\ 0, & otherwise \end{cases}$$

The Kaiser window actually meets filter specifications of desired width of the transition region $\Delta\varpi$ and the maximum allowable error δ in so far as it is possible to do so with a filter of length N+1. Knowing any two of the three parameters the designer can calculate the third such that the resulting filter will meet its specifications. With the three parameters set according to those calculations the Kaiser window can do as good or better a job as any of the other windows discussed above; in fact in some cases it turns out to be equivalent to one of those windows. If, however, all three of the parameters are set ahead of time, as they are in the case considered here, it is just another window. A Kaiser window is designed using the equation,

$$w[n] = \begin{cases} \dfrac{I_o\left[\beta\left(1-\left[(n-\alpha)/\alpha\right]^2\right)^{1/2}\right]}{I_o(\beta)}, & 0 \le n \le N, \\ 0, & otherwise \end{cases}$$

where $\alpha = N/2$ and $I_o(\cdot)$ is a zeroth-order modified Bessel function of the first kind. β is defined as,

$$\beta = \begin{cases} 0.1102(A-8.7), & A > 50 \\ 0.5842(A-21)^{0.4} + 0.07886(A-21), & 21 \le A \le 50, \\ 0.0, & A < 21 \end{cases}$$

where *A* is related to N, $\Delta\varpi$, and δ by $A = -20\log\delta$ and $N = \dfrac{A-8}{2.285\Delta\varpi}$.

Aside from the Boxcar window the various windows are quite effective at attenuating the large overshoot caused by the Gibbs phenomenon. They do so, however, at the expense of a wider transition band. In cases where only a small number of taps are used the transition band can be so wide that the design goal of significant attenuation by $\varpi = 1/L$ in the prototype filter cannot be met. This causes a problem because the subfilters made by sampling such a prototype filter are then aliased and consequently mismatched. A smooth roll-off, on the other hand, is actually somewhat advantageous for the video application addressed in this thesis. This is because it means that there are only small, incremental differences between filters that are appropriate for different but similar values of the resampling factor *P*, which makes for very smooth transitions. The human eye is quite sensitive to motion, so smooth transitions are even more important than absolute quality. Also, a single set of filters can be used for a range of values of *P*.

The frequency response characteristics of filters designed using the six different types of windows discussed above were evaluated for various combinations of resampling factors and numbers of coefficients. Special attention was paid to matching and overshoot when making comparisons. An example of such a comparison for the 16 5 tap subfilters with desired cutoff frequency $\varpi_c = 0.8\pi$ is shown in Figure 5-24. When comparing against the Kaiser window parameter values of $\delta = 0.05$ and therefore $\beta = 1.5098$ were used. The transition width $\Delta\varpi$ was left as the free parameter because, for a filter which is as heavily constrained by the number of coefficients as the filters designed here are, fixing $\Delta\varpi$ to its actual desired value leads to outrageously large values of the error δ.

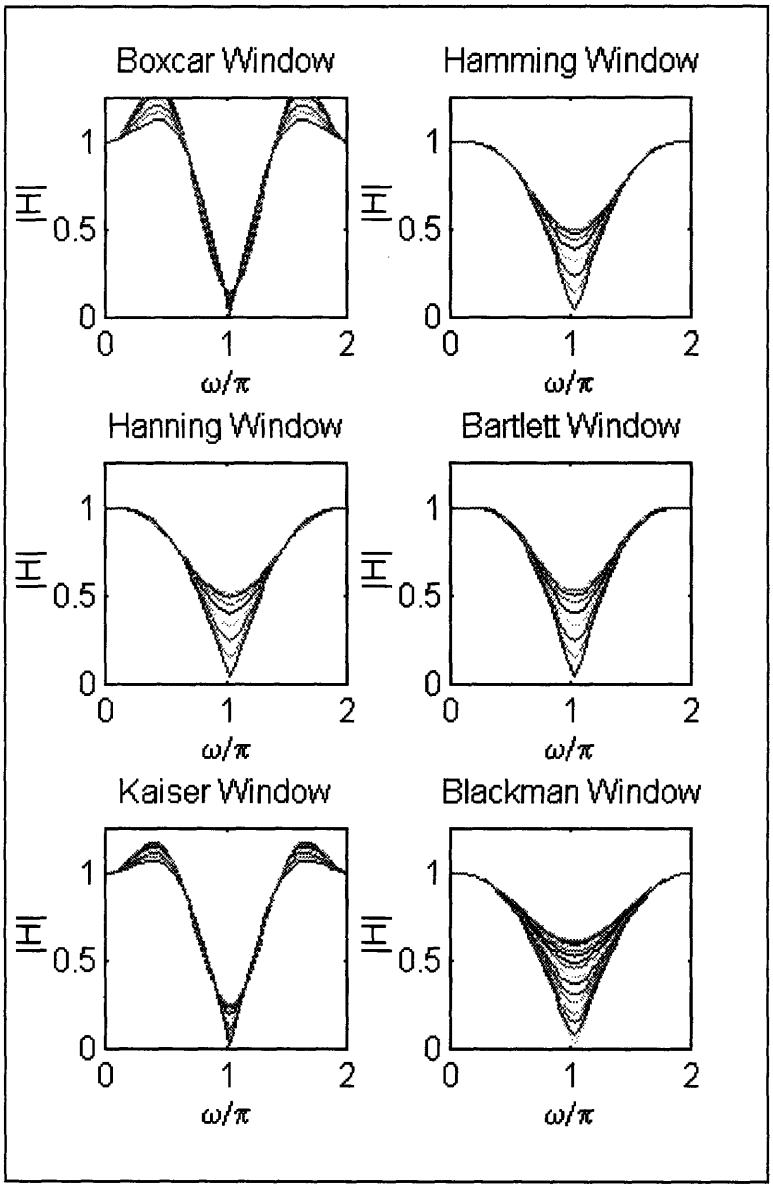

Figure 5-24: Example of Frequency Responses of Periodic Filters Designed Using Various Windows.

It was found that for most cases prototype filters designed using the Hamming window had the best matching and either the best or comparably small overshoots. Hanning, Bartlett, and Blackman windows had similar but slightly worse results. The Kaiser window was so overconstrained by the limitations on the number of coefficients a filter could have that the transition bands were so large that the sampled subfilters were aliased and so were poorly matched and had large overshoots. For the case where P is

close to one and the number of filter coefficients available per subfilter is as small as 3 or 4 a Kaiser window actually gave good results, and was used. For such a small number of taps the other windows responses of filters designed using the other windows was so poor that errors introduced by Kaiser windowing were more acceptable. For the case where *P* is close to 50% and the number of filter coefficients available per subfilter is as large as 11 both Kaiser windowing and the Remez exchange algorithm, no longer as overconstrained, actually did a good job of matching and keeping error small, but Hamming continued to be used in this region regardless in order to insure continuity. In some cases it was necessary to design filters with specified cutoff frequencies smaller than the desired cutoff frequencies so as to attenuate the effect on matching of too large a transition band.

Table 5-2 shows the breakdown of design methods and actual specified cutoff frequencies (for the subfilters) for each range of the resampling factor *P*.

Range of P	Number of Taps Per Periodic Filter	Desired Cutoff Frequency	Actual Specified Cutoff Frequency	Window Type
100% - 97%	3	0.97π	0.9π	Kaiser
97% - 86%	4	0.86π	0.8π	Kaiser
86% - 77%	5	0.78π	0.75π	Hamming
77% - 71%	6	0.71π	0.71π	Hamming
71% - 65%	7	0.65π	0.65π	Hamming
65% - 60%	8	0.6π	0.6π	Hamming
60% - 56%	9	0.56π	0.56π	Hamming
56% - 53%	10	0.53π	0.53π	Hamming
53% - 50%	11	0.5π	0.5π	Hamming

Table 5-2: Filter Design Methods and Parameters

5.3.2.3 Powers of Two Filters

One traditional way to design filters for hardware implementations is to constrain their coefficients to be powers of two. Such filters are can be efficiently implemented in hardware systems by shifting the input data rather than multiplying it. On the 'C80, however, a rotate is equally or more computationally expensive than a multiply, depending on the precision of the multiply. Powers of two filters, then, offer no advantage to compensate for imposing an extra constraint on the filter design.

5.3.3 Half-Band Filter Design

The half-band filter used was necessarily designed using windowing. This is because a window designed filter is the only one that is based directly on the ideal $\sin x/x$ function, so is the only one that keeps the special property that every other coefficient is equal to zero with the exception of the zeroth coefficient. Other design algorithms, such as the Remez-exchange, make no guarantees about zero valued coefficients. Since the extra computational efficiency provided by the regular zero-valued coefficients is the reason using a half-band filter is worthwhile in the first place this is not acceptable. So, the half-band filter was designed by windowing the ideal filter $h_{ideal}[n] = \frac{\sin(\frac{1}{2}\pi n)}{n}$ with a 13 point Hamming window.

Chapter 6

Implementation of Algorithm

The algorithm described in the previous chapter is implemented on the 'C80 in a way that makes efficient use of the processor's components and of the development hardware. After a brief introduction, this chapter describes the details of the implementation. A discussion of the MP and its role as the director of all the processing is followed by an overview of the data flow through the system and the parameter structures used to control PP operation. Finally, the details of the filtering as performed by the PPs are explained.

6.1 Introduction

The way the 'C80 is used in this project the Master Processor is, indeed, the master of all operations, the Transfer Controller is the data movement machine, and the Parallel Processors are slave processors which do all the real number crunching work. The MP takes care of interfacing with the timing hardware, directs all data movement using the TC, and sets up parameters for the PPs to use in their processing. It also decides what kind of processing is to be done, and directs the PPs to process which chunk of data and when.

The PPs are used in a round robin fashion: the MP sets up the data and parameters for PP0 and starts it. It then sets up the parameters for PP1 and starts it, and proceeds to do the same for PP2 and PP3. Eventually it comes back to PP1 where it sets up the PP to process the next set of data, waits for it to finish its processing, starts it processing the next set of data, reads the results of the previous set of

processing, and goes on to the next PP. Other schemes for setting up the PP processing are also possible. For example, the 'C80 is able to support running the PPs independently of the MP and of each other, with each of them requesting the TC to provide it with data when it needs it. However, the current scheme is the simplest (the one just described would require more sophisticated timing, bus arbitration, and TC code). It is also particularly effective in this case because the processing for each line in horizontal processing or for each column in vertical processing is the same.

When the program is first downloaded the MP takes care of a whole slew of initialization. Then it enters its main loop, which it passes through once per frame for the duration of the program's run time. There it makes decisions based on the value of the resampling factor P as to which algorithm it will use and how many passes that algorithm will take. It does whatever initialization is needed for each frame, and then, for each pass, updates the appropriate TC structures and PP parameters and handles the data movement so as to set up and implement first horizontal then vertical filtering. If necessary it makes sure that the vertical filtering is seamless. While doing all this it takes care to interleave data transfers with processing for maximum efficiency, so that most if not all of the data transfers occur while the PPs are busy processing. When all the processing is finished the MP takes care of any hardware issues such as the timing of the output frame.

Meanwhile the PPs take care of the actual filtering work. They, too, do a bit of setup and initialization the first time they are called, but carry very little overhead on subsequent calls.

6.2 Program Structure (Description of MP Program)

When the program is first loaded into the development hardware the Master Processor takes care of a set of initialization procedures. First of all it sets up the hardware/software interface. It then starts each of the Parallel Processors doing their own initialization, creates and initializes the necessary transfer controller structures, and sets up the initial states for the LED's that indicate whether or not the program is operating and waits for the first full page of data to start processing. Then it makes sure that all the PPs are finished with their initialization procedures, creates and initializes parameter structures in their Parameter RAMs, and collects pointers to those parameter structures in the array *ppParams[4]* and pointers to the filter coefficient fields in the array *CoefTable[4].* Finally it initializes a few variables and enters the endless main processing loop.

Once inside its main loop, which it steps through once per output frame, the MP directs all the processing. The structure of the main loop is shown in Figure 6-1.

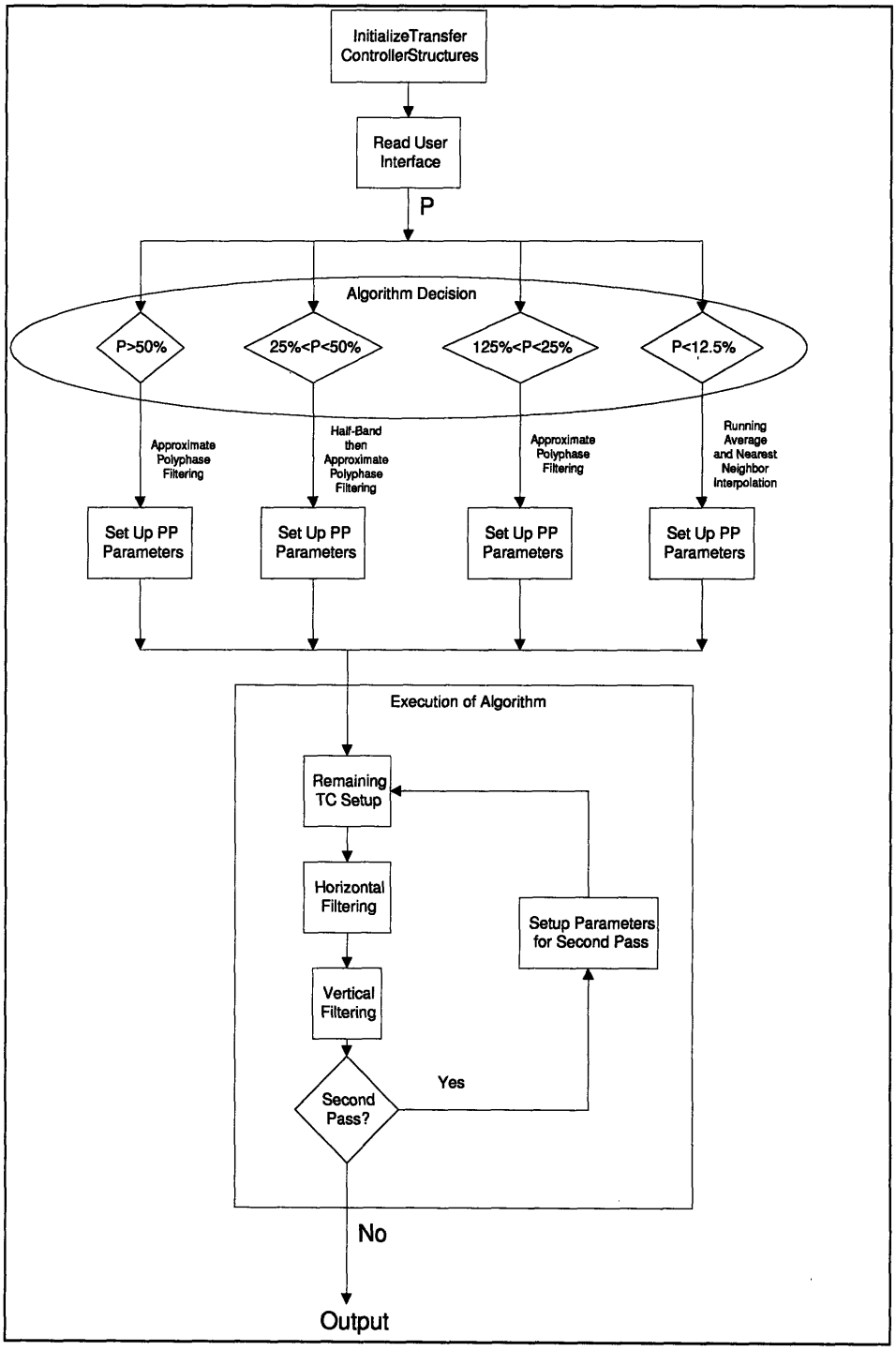

Figure 6-1: Structure of Main MP Loop

For each frame, the first thing the MP does is to reset any Transfer Controller Packet Transfer source and destination pointers that may have been corrupted by the previous frames' processing. It then reads the user interface to set its input parameter

incr, which is related to the resampling factor *P* by $P = \frac{1}{incr+1}$. Both the variables *P* and *incr* are used by the program; *incr* was chosen as the primary variable because, since it is the variable passed to the PPs and used by them to step through the input data, it is the simplest and offers the greatest resolution. *incr* is a fixed point variable with 16 bits after the decimal point, so it can be incremented by as little as 1 bit or 1.53×10^{-5}. That means that *P* has a resolution of 0.00153%. The MP then uses *incr* and *P to* select a resampling strategy to use as discussed in section 5.2.2, sets up the appropriate PP parameters to execute that strategy, including copying the necessary filter coefficients into the PP parameter RAMs, and executes it. If the chosen strategy requires two passes the MP executes the first pass, then sets up the parameters for the second pass and resets any TC structures that will be reused, and executes the second pass.

When a pass of the chosen resampling strategy is executed a frame of video is processed first horizontally and then vertically. In horizontal processing the MP interleaves several setup operations with starting the processing of the first four lines of data. This way the MP can efficiently touch all four of the PPs, and do some of the setup in the otherwise 'dead' time that the PPs are busy computing a line of output data. The first of these setup operations is initializing the PP memory banks to BLACK, where BLACK means that $Y = 16$, $Cb = 128$, and $Cr = 128$ as prescribed by the CCIR-601 signal definition. This is done to facilitate zero padding the input data. BLACK is used instead of absolute zero because it is the effective zero for color video, while absolute zero is used as a sync signal. The setup also includes setting any parameters in each PP's parameter structures that are peculiar to horizontal filtering, such as those which indicate which set of filter parameters are to be used, how many passes and how many filters in each pass are to be used, and the source and destination offset for each filter.

For each PP the MP tells the PP to begin processing as soon as it has received all the information it needs to do so.

After starting the PPs running on the first four lines of data the MP enters a loop which cycles through the PPs, transferring the next line of data to be processed to each one, waiting for it to finish its current processing, starting it on its next processing run, and transferring the line of data it has just finished processing out to the intermediate program memory. When all the processing is done the MP transfers out the last few lines of processed data. In addition, while it directs the first pass of horizontal processing the MP copies the input to the output to act as background. It does this line by line, interleaving one line of transfer during the 'dead' time while the PPs are processing.

Vertical filtering is done in much the same fashion as horizontal filtering at the MP level. The MP sets parameters unique to vertical filtering as it starts the PPs processing the first four columns, and then loops through the PPs, directing them to process all the columns while passing them new data from the previously horizontally filtered frame of video residing in the program memory and reading their processed data out to either program or output memory at opportune moments.

The major difference lies in that, for the first pass anyway, a full 8 byte wide 486 line column of data will not fit in a PP's 2 Kbyte data buffer. So, the vertical processing is done in two passes, half a column at a time, with a bit of extra work to make sure that there is continuity of output data between the two half-column blocks.

This 'seaming' issue is taken care of in the following fashion: Two additional variables, *nLinesOutPass* and *fracStart* are defined. The first half or $LASTLINE/2$ of the columns are processed without paying attention to seaming, except that only the first

nLinesOutPass output lines are computed. *nLinesOutPass* is defined as $nLinesOutPass = \left\lfloor \frac{numLinesOut+1}{2} \right\rfloor$ where $numLinesOut = numLinesIn(486) \cdot P$, and is the number of output lines that can be computed from $LASTLINE/2$ input columns using exclusively known values of the input, except for for the first few output lines. fracStart, which indicates the group delay of the first filter that should be used in processing a set of data, is set to zero for the first vertical pass. The second vertical pass uses input data which overlaps the first $LASTLINE/2$ input columns such that the first output line computed will the $nLinesOutPass + 1$st line. This overlap is achieved by using input data offset from the base of the input memory bank, *newCenter-EP_to_start*, where $newCenter = \frac{nLinesOutPass}{P}$ or $newCenter = nLinesInPass$ if a half band filter is being used, and EP_to_start is the distance in memory that is used for back tracking for zero-padding. Also, no zero-padding is used at the beginning of the second half of the columns; *srcOffset* is set to equal *EDGE_PAD* so that only output samples which are fully composed of weighted valid input samples are computed. Finally, *fracStart* is set to the fractional part of $\frac{nLinesOutPass}{P}$ so that the group delay progression from filtering the first to the second half of the columns is smooth.

6.3 Data Flow

The data memory structure of the program as it runs on one frame of video is outlined in Figure 6-2. The 'C80 has access to three of the five 128 Kbyte × 64 bit external memory banks during each frame cycle: Input Memory, intermediate Program Memory, and Output Memory. The MP uses the Transfer Controller to move data from these banks to the PPs and vice versa. As for the actual processing, the three 2 Kbyte memory banks within each PP are used as a set of rotating buffers. A variable *rrindex,*

valued 0,1, or 2, is used to index into the buffers in a round robin fashion. At any one time the data being processed by the PP is in the *src* or source buffer, where $src = rrindex$, the results of the processing are being deposited in the *dst* or destination buffer, where $dst = (rrindex + 2) \bmod 3$, and the results of processing the previous set of data are stored in the previous destination buffer $last\_dst = (rrindex + 1) \bmod 3$. Also, after the results of processing the previous line are read out the input data for the next line can be copied into the *last_dst* buffer without disturbing the current processing. An example is shown in Figure 6-3. Such a scheme means that the results of processing one set of data can be read out and copied to their destination and the next set of input data can be loaded at the same time as the results of processing the current set of data are being computed, without contention.

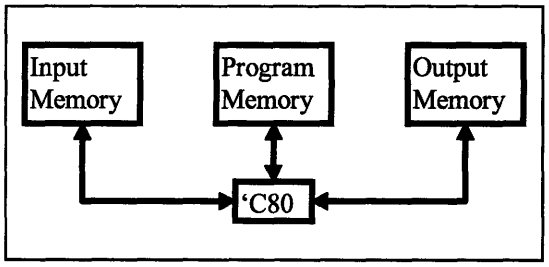

Figure 6-2: System Memory Structure

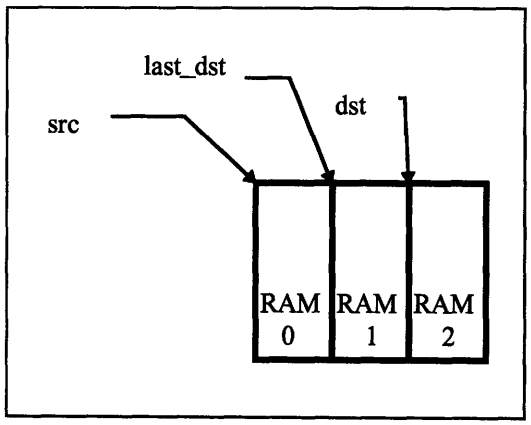

Figure 6-3: PP Memory Structure

The movement of the data around the system is handled by the Transfer Controller using dimensioned Packet Transfer structures. The Packet Transfers are either two or three dimensional. Line data is moved using two dimensional transfers where the packet's source and destination A count is equal to the number of bytes of a line to be transferred, the source and destination B counts are set to zero, and the source and destination B pitches are set to *LMEMBLOCK*, which is equal to 2048 bytes, the length of a line of memory, and is dictated by this particular hardware implementation. Column data is moved using three dimensional transfers where the packet's source and destination A count is equal to the number of bytes to be transferred per line, in this case 8, the source and destination B counts are set to the number of lines that make up a column minus one, because the B counts count up from zero, and the source and destination B pitches are again set to *LMEMBLOCK*. If data is being moved from a memory bank to a PP the Packet Transfer is set for source update mode, and if data is being moved from a PP to a memory bank the Packet Transfer is set for destination update mode.

The Packet Transfer structures used in this project are summarized in Table 6-1. In most cases their names indicate their functions. *tcInToTmp* and *tcTmpToOut* are used to transfer an image from input memory, or *InputBaseA*, to output memory, or *OutRAMBase*, to be used as background. *tcInToPp* is used to transfer an input image to the PP's for the first round of horizontal processing. *tcPpToPgm1* is used to move horizontally processed data from the PP to intermediate memory, or *ProgramRAM*, in line mode. *tcPpToPgm2* moves vertically processed data from the PP's to intermediate memory, in column mode, in the case where a multi-step algorithm is being used. Similarly, *tcPgmToPp1* moves data from intermediate memory to the

PP's for vertical processing in column mode and *tcPgmToPp2* moves data from intermediate memory to the PP's for horizontal processing in the second pass in line mode. Finally, *tcPpToOut* moves fully processed data from the PP's to output memory.

Name	Type	Mode	Source	Destination	A Count	B Count	B Pitch
tcInToTmp	Line	SUM	InputBaseA	PP	2*PIXPERLINE	0	LMEMBLOCK
tcTmpToOut	Line	DUM	PP	OutRAMBase	2*PIXPERLINE	0	LMEMBLOCK
tcInToPp	Line	SUM	InputBaseA	PP	2*PIXPERLINE	0	LMEMBLOCK
tcPpToPgm1	Line	DUM	PP	ProgramRAM	Variable	0	LMEMBLOCK
tcPpToPgm2	Column	DUM	PP	ProgramRAM	8	Variable	LMEMBLOCK
tcPgmToPp1	Column	SUM	ProgramRAM	PP	8	LASTLINE/2	LMEMBLOCK
tcPgmToPp2	Line	SUM	ProgramRAM	PP	Variable	0	LMEMBLOCK
tcPpToOut	Column	DUM	PP	OutRAMBase	8	Variable	LMEMBLOCK

Table 6-1: Packet Transfers and Their Parameters

6.4 Control of Filtering: PP Parameter Structures

The MP also sets up parameter structures in each of the PP's data RAMs and initializes them to default values. It does this by defining and initializing one such structure, *shrnkPpParams*, in the MP, copying it to each of the PP's, and making a table of pointers to the new PP structures so as to be able to access and modify them.

shrnkPpParams' structure type *shrnkMsgStruct* and its subsidiary structure types *passArgs* and *firArgList* are defined as follows:

```
struct shrnkMsgStruct {
      short           filtType;
      unsigned char*  src;
      unsigned char*  dst;
      short           passCount;
      short           numberOfFilters;
      passArgs*       fArgsPtr;
      passArgs        hFiltArgs[3];
      passArgs        vFiltArgs[1];
      short           fCoef[MAXFILTSIZE];
};
typedef struct shrnkMsgStruct shrnkMsgStruct;

struct passArgsStruct {
      firArgList      ppFirArgs;
      short           srcOffset;
      short           dstOffset;
};
typedef struct passArgsStruct passArgs;
```

```
struct firArgList {
      short        fracStart;
      short        nCoef;
      short        pixSpace;
      short        nPixels;
      unsigned short incr;
};
typedef struct firArgList firArgList;
```

The members of *shrnkMsgStruct* tell the PPs what method to use to filter which data and sets up the appropriate parameters for that filtering operation. *filtType* can be *PERIODIC*, which indicates approximate polyphase filtering, or *HALFBAND*, which indicates halfband filtering and decimating by two. *src* and *dst* are pointers to the source data and destination memory banks to be used by a particular filtering operation. *passCount* is the number of passes a particular filtering operation is to make over a set of data. It is equal to three for horizontal filtering, one pass for Y, one for Cb and one for Cr. For vertical filtering it is equal to eight, since vertical filtering is done in eight byte wide columns. *numberOfFilters* is the number of different sets of filters and filter arguments to be used for an operation. It is equal to three for horizontal filtering, since Y, Cb, and Cr must be treated separately and differently, and to one for vertical filtering, since vertical filtering is the same for all the elements in a column. The *hFiltArgs* and *vFiltArgs* arrays store the parameters for each type of filtering, and the *fCoef* array stores the filter coefficients to be used.

The structure *passArgs* describes the parameters of a filtering operation particular to filtering luminance, chrominance, or vertical data. Of its members *ppFirArgs* sets up the details of the filtering operation and *srcOffset* and *dstOffset* tell the PP exactly where in the input data memory bank to begin processing and exactly where in the output data memory bank to place the processed data, respectively. *srcOffset*, and *dstOffset,* therefore, are used to take care of zero-padding and to make sure that luminance and chrominance input values are properly mapped to the output.

Finally, the *firArgList* structure is used to set up the remaining detail of the filtering operation. Its members *fracStart* and *incr* tell the PP at what point of an approximate polyphase filtering operation to begin and how to update the running fraction counters, respectively. *nCoef* indicates how many filter coefficients are to be used, *nPixels* tells how many output pixels are to be computed, and *pixSpace* indicates how many bytes apart input and output pixels are, 2 for Y and 4 for Cb and Cr in horizontal filtering, and 8 for vertical filtering.

6.5 Filter Implementation (Description of PP Program)

Each of the four Parallel Processors performs a set of initialization tasks when it is first called by the Master Processor. It sets several variables relating to which PP it is, sets a variable *argsPtr* to be a pointer to the *shrnkMsgStruct* that resides in that PP, sets a variable *uCmnd* to be a command which halts the PP, and evaluates that command. The next time the PP is called it enters a loop which it stays in for the remainder of the program. That loop consists of reading *argsPtr* to find out which memory banks to use such that *srcBuf = argsPtr->src and dstBuf = argsPtr->dst*, calling the appropriate filtering function once for every pass and once for every separate filter specified for each pass, incrementing the *srcBuf* and *dstBuf* pointers in between passes, and halting the PP when all operations are finished. If a halfband filter is to be used the function *DecimateFunc* is called with arguments that tell the function where to begin processing, where to place computed output samples, where to find the appropriate *firArgList* parameter structure, and where to find the filter coefficients. If a set of periodic filters is to be used the function *ppQbyLTapFunc* is called with the same set of arguments.

The halfband filtering function *DecimateFunc* calculates the output samples by stepping through the input and, at every other sample, summing the dot product of the first *Q* input samples with the *Q* coefficients of the half band filter. It is assumed that the filter as stored in Parameter RAM is already flipped, so that the sum of the dot product is actually a convolution. For a symmetric filter, of course, flipping makes no difference in any case. After each output sample is computed it is placed in the destination memory buffer.

The periodic filtering function *ppQbyLTapFunc* performs its filtering in a similar but slightly more complicated fashion. In addition to stepping through the input samples using the source pointer *sPtr* it uses a 16 bit counter *fracCounter* to keep track of the point in between the sample pointed to by *sPtr* and the next one which really maps to a particular output sample. For example, to shrink a line of data by 78.1% one really wants to step through the input samples at increments of 1.2804. This is not possible; only integer size steps through a bank of memory make sense. So one steps through the input samples at integer intervals using *sPtr*, and keeps track of the fractional part by incrementing *fracCounter* by a variable *incr*, in this case 0.2804, with every step. Whenever *incr* crosses one *sPtr* is incremented by two samples and *fracCounter* is set back to its fractional part, otherwise *sPtr* is incremented by a single sample. *fracCounter* is then used to pick the appropriate filter h_i from a set of filters *H* to be used, one which has a group delay that will allow it to interpolate the value of the output sample that maps back to the input sample that would be at *sPtr* + *fracCounter*. The *H* filters, again pre-flipped, are stored in the Parameter RAM such that they are pointed to by *coefBase*. Since there are sixteen of them and they are all stored in contiguous memory the pointer to the filter to be used at a particular point, *currentCoefs*, is simply

equal to *CoefBase* plus the four highest order bits of *fracCounter* times the number of coefficients: *currnentCoefs = coefBase + (fracCounter>>12)*nCoef*. Once the position of the input samples to be used and h_i are chosen the output sample is calculated by summing the dot product of the first Q input samples with the Q coefficients of the filter h_i.

In both cases the math is done as 16×16 bit multiplies (8 bit data and 16 bit coefficients) and 32 bit adds. When the code is optimized 8 bit coefficients, and so 8×8 bit multiplies and 16 bit adds may be used for cases where the resampling factor P is large. An experiment was tried which showed that for large P video processed with 16 and 8 bit coefficients was indistinguishable.

Chapter 7

Results and Discussion

The program written for this thesis project is capable of shrinking a frame of video down to 25% of its original size both horizontally and vertically. It does not currently run in real time on the 'C80, but is designed such that a highly optimized assembly language version of the inner loop of the PP program should be able to do so on a 50 MHz part. The resampling is done with $1/16$ pixel resolution, and the resampling factor can be adjusted by increments as small as 0.00153%.

The quality of the output video can only be judged qualitatively and subjectively. Standard test signals such as color bars, flat fields, sinusoidal sweeps, and zone plates were used to test output quality. Test signal results, however, do not necessarily map to real video. For example, a process that looks terrible on a zone plate may actually look fine on real video. So, several clips of real video were also tested.

The test signals were used primarily for program development both in terms of algorithm and filter design and for debugging purposes. Final judgment on the quality of the program's output was made based on real video clips. This is because a test signal need only look so good in order for real video to look good. In the end, the output quality was deemed to be acceptably good.

7.1 Test Signals

Several of the standard video test signals were used to judge the quality of the program's output. These signals were particularly useful in the development stages of

the program because they show any mistakes very well, and because their predictability can be helpful in debugging. For example, a mistake in setting up the vertical processing to be 'seamless' even though it is done half a column at a time may be hard to find on a real picture, but shows up readily on a zone plate. Figure 7-1 shows such a seam on a zone plate with shallow diagonal lines superimposed on it.

Figure 7-1: Bad 'Seam' on a Zone Plate

Sinusoidal sweeps and a zone plate were used primarily to test filter matching and performance. Poor filter matching, for example, shows up as a low frequency modulation to the higher frequency region of a sweep. Figure 7-2 shows that effect for one line of a horizontal sweep resized by 98% using three tap filters. The filters used were matched as best as was possible considering their extremely short length. Figure 7-3 is a picture of a portion of the actual video output of the same operation on a zone plate, so that the effect of the mismatched filters is shown for both horizontal and vertical filtering. The effect is particularly noticeable here because the DC value of the output image is also modulated, even though the DC value of the signals it is made up of is not, because of Gamma correction. This effect is most visible for the cases where 3 and 4 tap filters are used. When the filters are larger they are matched well enough that there is no problem.

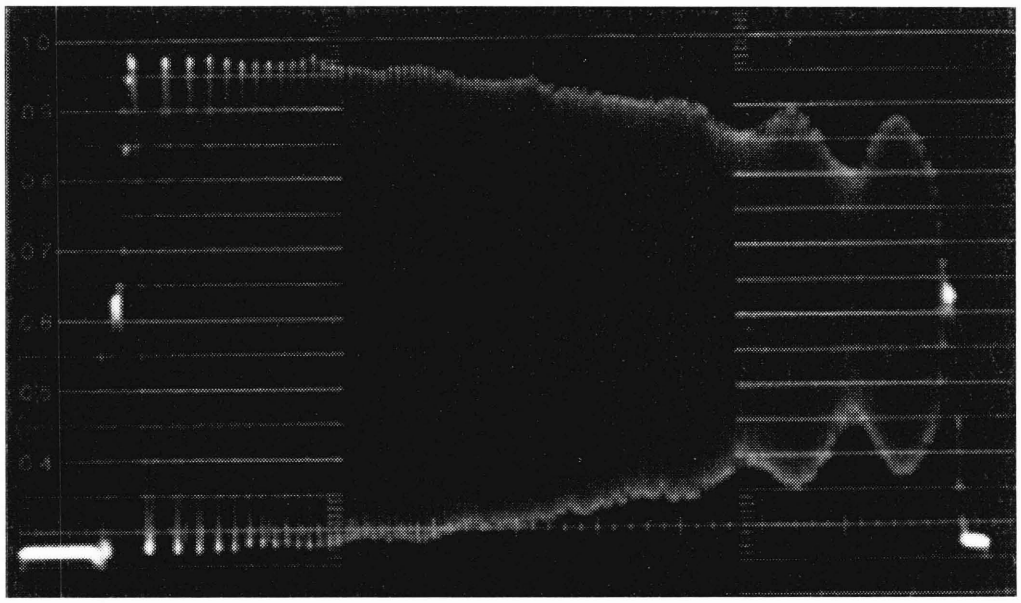

Figure 7-2: Effect of Mismatched Filters on a Single Line of a Horizontal Sweep

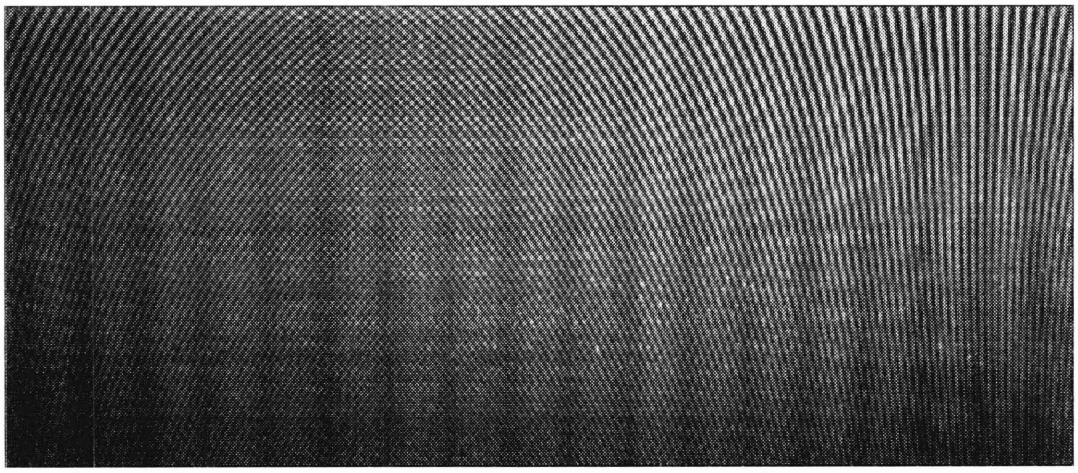

Figure 7-3: Effect of Mismatched Filters on a Zone Plate

Sweeps are also good for testing filter performance because they effectively show the frequency response of a filter and clearly display any aliasing. Figure 7-4 and Figure 7-5 show the signal of one line of output video that is a horizontal sweep resampled by 62%. In Figure 7-4 8 tap filters with cutoff frequency $\varpi_c = 0.85\pi$, which is too high, was used in doing the resampling. The output signal is attenuated very slowly and aliasing is clearly visible. In Figure 7-5 8 tap filters with the more correct cutoff

frequency $\varpi_c = 0.60\pi$ were used in doing the resampling. The output signal is attenuated with a much sharper cutoff, and there is significantly less aliasing. Most of the filters that were used performed quite well: only a small amount of aliasing was seen, and only at the very high frequencies.

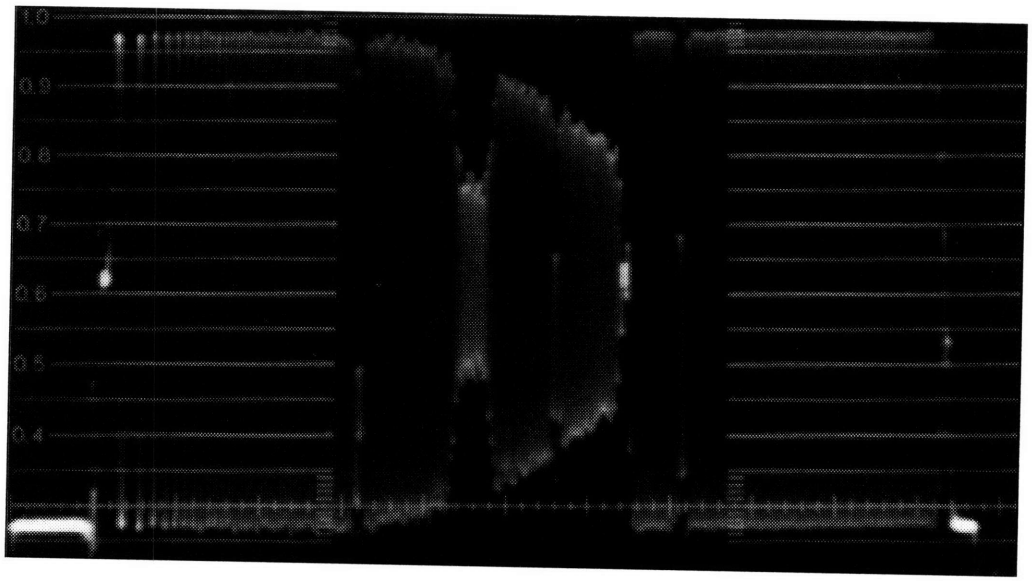

Figure 7-4: Effect of Resampling With Filters With Too High a Cutoff Frequency

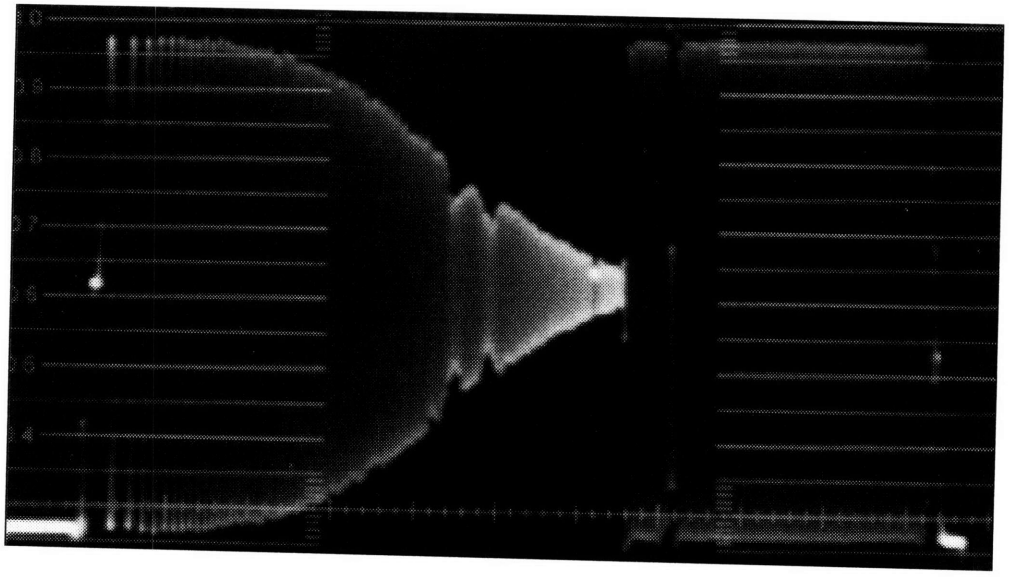

Figure 7-5: Effect of Resampling With Filters With an Appropriate Cutoff Frequency

Test signals such as color bars and flat fields were used to make sure that the resampling process has unity gain at DC and that it treats color information correctly.

7.2 Real Video

The overall quality of the processing performed by the program written was tested using real video clips. For example, a piece of video of a courtyard with trees and buildings in the background and a man running across the screen in the foreground was used. The results were quite good: The edges of the building and of the sidewalk remained sharp and the details of the trees did not show significant aliasing artifacts. The program does not run in real time so motion artifacts due to frame processing and interlaced scanning could not be evaluated, but they surely exist.

Another, harder, piece of video that was used for evaluation was a still of a dollar bill, a magazine cover, and a few other objects. This still had lots of very hard (therefore high frequency) edges and fine details. Here it was possible, though barely, to discern that the wide transition bands of the filters softened the edges somewhat and that there was, in fact, a slight alias component in the finely detailed regions of the dollar bill. Overall, however, the output quality was quite acceptable; the artifacts were only visible in side by side comparisons.

7.3 Effects of Coefficient Accuracy

The greatest effect of coefficient accuracy was that the DC gain had to be equal to exactly one. Even one bit's difference was clearly noticeable on a flat field, and would be on any piece of real video with sizable low frequency areas.

16 bit coefficients were used in the final program. However, 8 bit coefficients could be used in the optimized version for cases where the output picture size is large in

order to gain some computational efficiency. Output quality would not be significantly degraded because, since the output picture size is large the filters do not do very much.

7.4 Further Work

There is still much work that could be done to improve the performance of the program. First and most obviously, the PP part of the program needs to be optimized to run in real time. Second it needs to be extended to be able to treat shrink factors smaller than 25%.

Third, there are several small modifications that could make it run better and smoother: When the current program resamples vertically it does so in 8 byte blocks. This works perfectly fine, so long as the number of pixels in an output line is a multiple of four. Otherwise the program can either display extra columns of bogus data or it can cut down the number of columns it processes such that it processes only full columns. It does the latter, and that appears somewhat jerky at the right edge as the picture size is varied in time. The program could be modified to be able to process the last set of columns in blocks of any size smaller than or equal to eight. Also, there is a bug, a horizontal black line that appears at the top of the output picture whenever P is less than 50%, whose cause has not yet been found.

Finally, the program could be modified to treat video as fields or frames, to be insensitive to the size of the input picture, and to be able to place the output picture anywhere in the output frame at subpixel resolution.

7.5 Conclusions

This thesis explored several different ways to resample video and implemented approximate polyphase filtering in a two stage fashion on the 'C80. The 'C80 lends itself

quite well to resampling video because of its efficient data movement capabilities and the computational horsepower provided by its four parallel processors. Because it is a programmable processor it can be used to implement a dynamic algorithm which uses different resampling methods and different size filters as needed.

Resampling of acceptable quality, with $1/16$ pixel resolution in terms of interpolation and 0.00153% resolution in terms of resizing, was achieved. It can not currently be done in real time, but should be able to when properly optimized. The resampling was tested on test signals and real video. The testing showed that matching the magnitude of the frequency response of the approximate polyphase filters to each other was especially important for video, even more so than making sure that the magnitude frequency response was as close to ideal as possible, and the filters finally used were designed accordingly. The final version of the program was found to be of acceptable quality on real video.

Bibliography

CCIR (International Radio Consultative Committee). *Recommendation 601-1: Encoding Parameters of Digital Television for Studios*, Recommendations and Reports of the CCIR, 1986, Volume XI Part 1: Broadcasting Service (Television). Geneva, 1986.

Crochiere, Ronald E. and Rabiner, Lawrence A. Multirate Digital Signal Processing. Prentice Hall, Inc. New Jersey, 1983.

Lim, Jae S. and Oppenheim, Alan V. Eds. Advanced Topics in Signal Processing. Prentice Hall, Inc. New Jersey, 1988.

Lim, Jae S. Two-Dimensional Signal and Image Processing. Prentice Hall, Inc. New Jersey, 1990.

Penney, Bruce. Private Conversations June-November 1995.

Texas Instruments, Inc. TMS320C80 (MVP) Online Reference, Release 1.10. Texas, 1995.

Wolberg, George. Digital Image Warping. IEEE Computer Society Press. Los Alamitos, California, 1990.Appendices

Appendix A: MATLAB Code

```
%  File:  ARMkFlt
%  Creates L Q-tap approximate polyphase filters, each with cutoff frequency P*pi
%  designed by sampling an L*Q filter, which in turn is designed by windowing
%  an ideal low-pass filter with cutoff frequency P*pi/L with 'window'
%  Also orders the filters according to group delay.

f0 = fir1(Q*L, P/L, window);
f1 = arconv(L,Q,f0);
f1 = f1';

filt = [];

if ((Q/2) ~= floor(Q/2))
for i=L:-1:L/2+1
        filt = [filt; f1(i,:)];
end;
for i=L/2:-1:1
        filt = [filt; f1(i,:)];
end;

else
for i=1:L
       filt(i,:) = f1(L+1-i,:);
end;
end;
```

```
function out = To15(in)
%  function out = To15(in)
%
%  converts a set of filters with values ranging from -1 to 1
%  to fixed point signed 15 bit numbers such that all the coefficients
%  of each filter sum to 2^15.


[m,n] = size(in);

%  scale input
in = in*32768;

%  make sure the integer version adds up to 2^15
for i=1:m
        tmp = round(in(i,:));
        if sum(tmp) == 32768
                out(i,:) = tmp;
        else
                while sum(tmp) > 32768
                        min_frac = .99;
                        min_ind = 1;
                        for j=1:n
                                if in(i,j) > 0
                                        frac = in(i,j) - floor(in(i,j));
                                else
                                        frac = 1 - (in(i,j) - floor(in(i,j)));
                                end;
                                if ((frac < min_frac) & (frac>.5))
                                        min_frac = in(i,j)-floor(in(i,j));
                                        min_ind = j;
                                end;
                        end;
                        tmp(min_ind) = tmp(min_ind)-1;
                        in(i,min_ind) = tmp(min_ind);
                end;
                while sum(tmp) < 32768
                        max_frac = 0;
                        max_ind = 1;
                        for j=1:n
```

```
                                if in(i,j) > 0
                                        frac = in(i,j) - floor(in(i,j));
                                else
                                        frac = 1 - (in(i,j) - floor(in(i,j)));
                                end;
                                if ((frac > max_frac) & (frac<.5))
                                        max_frac = frac;
                                        max_ind = j;
                                end;
                        end;
                        tmp(max_ind) = tmp(max_ind)+1;
                        in(i,max_ind) = tmp(max_ind);
                end;
        out(i,:) = tmp;
        end;
end;
```

```
% Filename:  sim.m
% simulation of approximate polyphase filtering with sinusoidal sweep as input

pixCounter = 1;
fracCounter = 0;
incr = 3449/2^16;
coefOffset = 1;
numfilts = 16;
numpix = 800;

p=0;
for i=1:1000
        p = p+4*i;
        in(i) = sin(p/1000);
        end;
end;

n=1:1000;
in=sin(n*(1/pi));

flag = 0;

for i=1:numpix
        val = in(pixCounter:pixCounter+Q-1).*filt(coefOffset,:);
        out(i) = sum(val);

        pixCounter = pixCounter+1;
        fracCounter = fracCounter+incr;
        if fracCounter >= 1
                fracCounter = fracCounter-1
                pixCounter = pixCounter+1;
        end;
        coefOffset = floor(fracCounter*16) + 1;
end;
```

```
% Filename:  Spectra.m
% Used to compare filters designed using boxcar, hamming,
% hanning, bartlett, blackman, and kaiser filters.


echo on;
clg;

delta = [];
maxes = [];
n = (1:64)/32;

subplot(3,2,1);
window = boxcar(Q*L);
armkflt;
mag = abs(fft(filt',64));
delta = max(min(mag))-min(min(mag));
maxes = max(max(mag));
plot(n,mag);
xlabel('w/p','fontName','symbol');
```

```
ylabel('|H|');
title('Boxcar Window');
axis([0,2,0,1.25]);

subplot(3,2,2);
window = hamming(Q*L);
armkflt;
mag = abs(fft(filt',64));
maxes = [maxes;max(max(mag))];
delta = [delta;max(min(mag))-min(min(mag))];
plot(n,mag);
xlabel('w/p','fontName','symbol');
ylabel('|H|');
title('Hamming Window');
axis([0,2,0,1.25]);

subplot(3,2,3);
window = hanning(Q*L);
armkflt;
mag = abs(fft(filt',64));
maxes = [maxes;max(max(mag))];
delta = [delta;max(min(mag))-min(min(mag))];
plot(n,mag);
xlabel('w/p','fontName','symbol');
ylabel('|H|');
title('Hanning Window');
axis([0,2,0,1.25]);

subplot(3,2,4);
window = triang(Q*L);
armkflt;
mag = abs(fft(filt',64));
maxes = [maxes;max(max(mag))];
delta = [delta;max(min(mag))-min(min(mag))];
plot(n,mag);
xlabel('w/p','fontName','symbol');
ylabel('|H|');
title('Bartlett Window');
axis([0,2,0,1.25]);

subplot(3,2,5);
window = kaiser(Q*L, 1.5098);
armkflt;
mag = abs(fft(filt',64));
maxes = [maxes;max(max(mag))];
delta = [delta;max(min(mag))-min(min(mag))];
plot(n,mag);
xlabel('w/p','fontName','symbol');
ylabel('|H|');
title('Kaiser Window');
axis([0,2,0,1.25]);

subplot(3,2,6);
window = blackman(Q*L);
armkflt;
mag = abs(fft(filt',64));
maxes = [maxes;max(max(mag))];
delta = [delta;max(min(mag))-min(min(mag))];
plot(n,mag);
xlabel('w/p','fontName','symbol');
ylabel('|H|');
title('Blackman Window');
axis([0,2,0,1.25]);


delta
maxes
```

Appendix B: 'C80 Code

```
/*
 * FileName: shrnkm.c
 * MP code for shrinking
 * Copyright Tektronix, Inc.
 * Company confidential.
 */

#include "global.h"
#include "hwstuff.h"
#include "mvphw.h"
#include "video.h"
#include "mvp.h"
#include "protohw.h"
#include "string.h"
#include "shrnkMsg.h"
#include "joyStick.h"
#include "backGndGen.h"
#include "tcPacket.h"
#include "stdio.h"
#include "ppParamAlloc.h"
#include "Coef_table.h"

#define PP_ALL_ADDR    0x0000000FL                    /* All PPs Addresses combined. */
#define ALL_PP_ERR_HALT (PP_ALL_ADDR << 16)       /* All PPs halted mask. */

const short* chooseFilter(float, const short*, short*);

/*
 * Define pointers to PP RAM locations
 */
extern VOID    PP_START(VOID);       /* Starting address for PPs. */
ppParameterRam* ppParamRam[PP_COUNT] = {
  (ppParameterRam*)PP0_PARAMETER_RAM,
  (ppParameterRam*)PP1_PARAMETER_RAM,
  (ppParameterRam*)PP2_PARAMETER_RAM,
  (ppParameterRam*)PP3_PARAMETER_RAM
 };


UBIN8* ppState[PP_COUNT][3] = {
  (UBIN8 *)PP0_DATA_RAM_0, (UBIN8 *)PP0_DATA_RAM_1, (UBIN8 *)PP0_DATA_RAM_2,
  (UBIN8 *)PP1_DATA_RAM_0, (UBIN8 *)PP1_DATA_RAM_1, (UBIN8 *)PP1_DATA_RAM_2,
  (UBIN8 *)PP2_DATA_RAM_0, (UBIN8 *)PP2_DATA_RAM_1, (UBIN8 *)PP2_DATA_RAM_2,
  (UBIN8 *)PP3_DATA_RAM_0, (UBIN8 *)PP3_DATA_RAM_1, (UBIN8 *)PP3_DATA_RAM_2,
};


/*
 * Define prototype PP Parameter Structure
 */
shrnkMsgStruct shrnkPpParams = {
  PERIODIC, 0, 0, 1, 3, 0,
```

```
{                   /* hFiltArgs */
 { {0, 3, 2, PIXPERLINE,0 }, EDGE_PAD-1, EDGE_PAD+1 },
 { {0, 3, 4, PIXPERLINE/2,0 }, EDGE_PAD-4, EDGE_PAD },
 { {0, 3, 4, PIXPERLINE/2,0 }, EDGE_PAD-2, EDGE_PAD+2 },
},
{                   /* vFiltArgs */
 { {0, 3, 8, LASTLINE/2+1,0 }, EDGE_PAD-8, EDGE_PAD },
},
{ 0,0,0,0,0,0,0,0,0,0 },
};

main()
{
  /*
   * Pointers to TC Structures
   */
  tcPacket        *tcInToTmp;     /* Used to copy the full field background */
  tcPacket        *tcTmpToOut;    /* Used to copy the full field background */
  tcPacket        *tcInToPp;      /* Copy input to PP for filtering */
  tcPacket        *tcPpToPgm1;    /* Copy result of H filtering to intermediate memory */
  tcPacket        *tcPgmToPp1;    /* Copy h filtered to PP */
  tcPacket        *tcPpToPgm2;    /* Copy result of h filter to int memory, if 2 pass */
  tcPacket        *tcPgmToPp2;    /* Copy h filtered to PP, if 2 pass */
  tcPacket        *tcPpToOut;     /* Copy shrunk image to output */
  tcPacket        *tcPpToWhere;   /* Pointer to either tcPpToOut or tcPpToPgm2, depending on pass */
  tcPacket    *tcWhereToPp;       /* Pointer to either tcInToPp or tcPgmToPp2, depending on pass */

  /*
   * Pointers to PP memory locations
   */
  shrnkMsgStruct*         ppParams[PP_COUNT];  /* Pointers to Param structures in PPs */
  short*          CoefTable[PP_COUNT];         /* Pointers to Coefficient tables in PPs */

  /*
   * Hardware stuff
   */
  UBIN8                   page;                /* Current memory page. */
  UBIN8                   tmpPage;             /* Temporary memory page. */
  UBIN8                   RedLED;              /* Current Red LED state. */

  /*
   * Processing control variables
   */
  short           numPasses;                   /* Number of passes to be made */
  short           pass;                        /* Current pass */
  int              ppNum;                      /* PP index. */
  int              rrIndex;                    /* Round Robin data buffer index */
  short           numLinesOut;                 /* number of output lines */
  short           numPixOut;                   /* number of pixels per output line */
  UBIN32           ioStartAddr;                /* used to read data from PPs */
  UBIN32           ioDestAddr;                 /* used to write data to PPs */
  int              lineOut;                    /* Number of lines of data left to read out */
  int              lineCnt;                    /* Number of lines left to process */
  int              lineStart;          /* Number of lines being processed */
  int              newLines;                   /* lines the PPs are processing. */
```

```
int             colOut;                 /* Number of columns of data left to read out */
int             colCnt;                 /* Number of columns left to process */
int             colStart;           /* Number of columns being processed */
int             newCols;                /* Columns the PPs are processing. */
short           numLinesIn;             /* Number of columns in in current pass */
short           numPixIn;               /* Number of pixels in in current pass */
short           num_vert_passes;        /* Number of passes for vertical filtering */
int             vert_pass;              /* Current pass in vertical filtering */
short           nLinesInPass;           /* Number lines in in current vert. pass */
short           nLinesOutPass;          /* Number lines out in current vert. pass */
short           newCenter;              /* Where to get data from in 2nd vert. pass */

/*
 * MP Parameters
 */
int             method;             /* How to run user interface */
unsigned int    incr;               /* Related to resampling factor.  fixed point, 16.16 bits */
unsigned int    incrincr;           /* For user interface, changes in incr */
float           percent;            /* Resampling factor */
short           filtType;           /* Type of filter to be used:  periodic or halfband */
short           EP_to_start;        /* Related to number of coefs, for edge padding */
const short     *filtCoefs;         /* Pointer to filter coefs to be used */
int             filtSize;           /* number of filter coefs to be copied into PP param RAM */
/*
 * Parameters to be passed to the PPs
 */
short                   srcOffset;      /* Where to start filtering */
unsigned short          fracStart;      /* At what group delay to start filtering */
short                   coef;           /* number of coefficients */



/*
 * Initialize hardware to proper mode.
 */
VideoIntDI();           /* Disable frame interrupt. */
InDelayReset();         /* Reset the input. */
/*  OutPageAuto();*/    /* Set up the output for real-time operation */
OutPageManual();        /* Set up output for non-real-time operation */
OutPageZero();          /* Set up the output. */
InOneChMode();          /* Set input to One Ch mode. */
InSelCh2();             /* MUST be done in One Ch mode. */
InPageAuto();           /* Set Auto page select mode. */
InPageZero();           /* Set up the input. */
OutDelayReset();        /* Reset the Output. */
InDelayBypass();        /* Set filter-bypass delay. */
prmsg("TEST-AR2...\n");

/*
 * Initialize all PPs, (Halt & Flush, set Task Interrupt vector,
 * and start them up.
 */
command(PP_ALL_ADDR | CMD_D_FLUSH_OP | CMD_I_FLUSH_OP | CMD_RESET_OP);
for (ppNum = 0; ppNum < 4; ppNum++) {
```

```
  ppParamRam[ppNum]->vector.taskInterrupt = (UBIN32) (PP_START);
}
command(PP_ALL_ADDR | CMD_UNHALT_OP);

/*
 * Create and Initialize Transfer Controller structures.
 */
tcInToTmp = NewMpTcPacket();
TcLineCopy(tcInToTmp, InputBaseA, 0,
            PIXPERLINE, SUM_MODE_B);

tcTmpToOut = NewMpTcPacket();
TcLineCopy(tcTmpToOut, 0, OutRAMBase,
        PIXPERLINE, DUM_MODE_B);

tcInToPp = NewMpTcPacket();
TcLineCopy(tcInToPp, InputBaseA, 0,
        PIXPERLINE, SUM_MODE_B);

tcPpToPgm1 = NewMpTcPacket();
TcLineCopy(tcPpToPgm1, 0, ProgramRAM,
        PIXPERLINE, DUM_MODE_B);

tcPpToPgm2 = NewMpTcPacket();
TcDstVStripCopy(tcPpToPgm2, 0, ProgramRAM,
            BLOCKSIZE, LASTLINE/2+1);

tcPgmToPp1 = NewMpTcPacket();
TcSrcVStripCopy(tcPgmToPp1, ProgramRAM,
            0, BLOCKSIZE, 251);

tcPgmToPp2 = NewMpTcPacket();
TcLineCopy(tcPgmToPp2, ProgramRAM, 0,
        PIXPERLINE, SUM_MODE_B);

tcPpToOut = NewMpTcPacket();
TcDstVStripCopy(tcPpToOut, 0, OutRAMBase,
            BLOCKSIZE, LASTLINE/2+1);

/*
 * Set initial states.
 */
LEDRedOn();             /* Turn on the Red LED. */
RedLED = 1;             /* Remember state of LED. */
page = (InPage & ReadStatusP());            /* Load current Input Page number. */

while (page == (InPage & ReadStatusP())) /* Wait for new page. */
  ;

/*
 * Wait for all PPs to halt.
 */
while (ALL_PP_ERR_HALT - (ALL_PP_ERR_HALT & PPERROR)) {
  printf("\nuipp error = 0x%x\n", (unsigned int) PPERROR);
}
```

```
/*
 * Init PP data structures with default values
 * and initialize pointers to Coefficients
 */

for (ppNum = 0; ppNum < 4; ppNum++) {
  ppParams[ppNum] = NewPpParamWithCopy(ppNum, sizeof(shrnkPpParams),
                    (void*)&shrnkPpParams);
  CoefTable[ppNum] = (short *) (&ppParams[ppNum]->fCoef);
}

/*
 * Init Main loop variables
 */
method = AUTO;
incr = 0;
incrincr = 0x10;

/*
 * Endless Main loop.
 */
while (1) {                    /* Endless Loop! */
  LEDGrnOn();/* Turn on the Green LED. */


  /*
   * Init transfer structures
   */
  TcDstStart(tcTmpToOut, OutRAMBase);
  TcSrcStart(tcInToTmp, InputBaseA);
  TcSrcStart(tcInToPp, InputBaseA);
  TcDstStart(tcPpToPgm1, ProgramRAM);
  TcSrcStart(tcPgmToPp1, ProgramRAM);
  TcDstStart(tcPpToPgm2, ProgramRAM);
  TcSrcStart(tcPgmToPp2, ProgramRAM);
  TcDstStart(tcPpToOut, OutRAMBase);


  /*
   * Read user interface to get value of incr
   */

  /*
   * If Joystick is in upper quarter and button is released
   * change user interface methods
   */
  if (JoyYPos() > ((JOY_MAX-JOY_MIN)*3/4 + JOY_MIN)) {
    if (ButtonActionOnRelease()) {
      method = (method + 1) % NUM_METHODS;
      printf("method = %d\n", method);
    }
  }
```

```
/*
 * If method is AUTO, reset incr when button is released.
 * If method is MANUAL, read incr directly from h. position of joystick.
 * If method is FIXED, increment incr if button is released.
 */
if (method == AUTO) {
  if (ButtonActionOnRelease()) {
    incr = 0;
  }
  incr += incrincr;
}
else if (method == MANUAL)
  incr = ((JoyXPos() - JOY_MIN)<<17)/(JOY_MAX-JOY_MIN);
else if (method == FIXED) {
  if (ButtonActionOnRelease()) {
    incr += incrincr;
  }
}


/*
 * Decide how to process frame depending on value of incr
 */
if ((incr>>16) > 0) {
  filtType = HALFBAND;
  percent = .50;
  if (incr == (1<<16)) {
    numPasses = 1;
  }
  else
    numPasses = 2;
}
else {
  filtType = PERIODIC;
  numPasses = 1;
}

numLinesIn = LASTLINE;
numPixIn = PIXPERLINE;


/*
 * Processing loop
 */
for (pass=0;pass<numPasses;pass++) {
  TcWaitForFree();

  /*
   * Decide parameters
   */
  if (numPasses == 1) {
    tcPpToWhere = tcPpToOut;
    tcWhereToPp = tcInToPp;
    if (filtType == HALFBAND)
      percent = .50;
```

```
  else
    percent = 1.0/((((float) incr)/(65536.0))+1.0);
}
else if (pass == 0) {
  tcPpToWhere = tcPpToPgm2;
  tcWhereToPp = tcInToPp;
  percent = .50;
}
else {
      tcPpToWhere = tcPpToOut;
  tcWhereToPp = tcPgmToPp2;
  percent =  1.0/((((float) ((unsigned short) incr))/(65536.0))+1.0);
  TcDstStart(tcPpToPgm1, ProgramRAM);
  TcSrcStart(tcPgmToPp1, ProgramRAM);
  filtType = PERIODIC;
  numLinesIn = numLinesOut;
  numPixIn = numPixOut;
}


numPixOut = (short) ((float) numPixIn)*percent;
numLinesOut = (short) ((float) numLinesIn)*percent;
filtCoefs = chooseFilter(percent, filtCoefs, &coef);
EP_to_start = coef/2;

/*
 * Copy coefficient table into PP Parameter RAMs
 */
if (filtType == PERIODIC)
  filtSize = sizeof(short) * ((int) coef) * 16;
else
  filtSize = sizeof(short) * 11;
for (ppNum=0;ppNum<PP_COUNT;ppNum++) {
  memcpy(CoefTable[ppNum],(void*) filtCoefs , filtSize);
}

/*
 * Set up additional PP Params
 */
for (ppNum=0;ppNum<PP_COUNT;ppNum++) {
  ppParams[ppNum]->hFiltArgs[0].ppFirArgs.nPixels = (short) numPixOut;
  ppParams[ppNum]->hFiltArgs[0].ppFirArgs.nCoef = coef;
  ppParams[ppNum]->hFiltArgs[0].ppFirArgs.incr = (unsigned short) incr;
  ppParams[ppNum]->hFiltArgs[0].ppFirArgs.fracStart = 0;
  ppParams[ppNum]->hFiltArgs[1].ppFirArgs.nPixels = (short) numPixOut/2;
  ppParams[ppNum]->hFiltArgs[1].ppFirArgs.nCoef = coef;
  ppParams[ppNum]->hFiltArgs[1].ppFirArgs.incr = (unsigned short) incr;
  ppParams[ppNum]->hFiltArgs[1].ppFirArgs.fracStart = 0;
  ppParams[ppNum]->hFiltArgs[2].ppFirArgs.nPixels = (short) numPixOut/2;
  ppParams[ppNum]->hFiltArgs[2].ppFirArgs.nCoef = coef;
  ppParams[ppNum]->hFiltArgs[2].ppFirArgs.incr = (unsigned short) incr;
  ppParams[ppNum]->hFiltArgs[2].ppFirArgs.fracStart = 0;
  ppParams[ppNum]->vFiltArgs[0].ppFirArgs.nCoef = coef;
  ppParams[ppNum]->vFiltArgs[0].ppFirArgs.incr = (unsigned short) incr;
  ppParams[ppNum]->vFiltArgs[0].ppFirArgs.fracStart = 0;
```

```
  ppParams[ppNum]->filtType = filtType;
}

/*
 * Update TC structures
 */
tcPpToPgm1->srcAcount = numPixOut*2;
tcPpToPgm1->dstAcount = tcPpToPgm1->srcAcount;

/*
 * Initialize loop control variables
 */
rrIndex = 1;
lineCnt = numLinesIn-4;       /* Number of lines to process */
lineOut = 4;                  /* Number of lines transfered out. */
lineStart = 0;                /* Number of lines started. */


/*
 * Init PP data RAMs
 */
for (ppNum = 0; ppNum < 4; ppNum++) {
  /*
   * Color them BLACK
   */
  GenColorLine(ppState[ppNum][0], 1024, BLACK);
  GenColorLine(ppState[ppNum][1], 1024, BLACK);
  GenColorLine(ppState[ppNum][2], 1024, BLACK);

  /*
   * Setup PP memory pointers
   */
  ioStartAddr = (UBIN32)ppState[ppNum][0]+EDGE_PAD;
  ioDestAddr = (UBIN32)ppState[ppNum][2]+EDGE_PAD;

  /*
   * If first pass, copy input to output
   */
  if (pass == 0) {
    TcDstStart(tcInToTmp, ioStartAddr);
    TcStart(tcInToTmp);
    TcSrcStart(tcTmpToOut, ioStartAddr);
    TcStart(tcTmpToOut);
  }

  TcWaitForFree();

  /*
   * load PPs with data and parameters
   */
  TcDstStart(tcWhereToPp, ioStartAddr);
  TcStart(tcWhereToPp);
  ppParams[ppNum]->src = ppState[ppNum][0];
  ppParams[ppNum]->dst = ppState[ppNum][2];
  ppParams[ppNum]->fArgsPtr = ppParams[ppNum]->hFiltArgs;
```

```
  ppParams[ppNum]->passCount = 1;
  ppParams[ppNum]->numberOfFilters = 3;
  ppParams[ppNum]->hFiltArgs[0].srcOffset = EDGE_PAD-(EP_to_start*2)+1;
  ppParams[ppNum]->hFiltArgs[1].srcOffset = EDGE_PAD-(EP_to_start*4);
  ppParams[ppNum]->hFiltArgs[2].srcOffset = EDGE_PAD-(EP_to_start*4)+2;

  /*
   * Wait until the data got to the PP then start the PP
   */
  TcWaitForFree();
  command(CMD_PP_ADDR(ppNum) | CMD_UNHALT_OP);
}


/*
 * Loop to process horizontal filtering
 */
while (lineOut) {
  for (ppNum = 0; (lineCnt > 0) && (ppNum < 4); lineCnt--, ppNum++) {
    ioStartAddr = (UBIN32)ppState[ppNum][rrIndex];

    /*
     * If first pass, copy input to output
     */
    if (pass == 0) {
      TcWaitForFree();
      TcDstStart(tcInToTmp, ioStartAddr+EDGE_PAD);
      TcStart(tcInToTmp);
      TcSrcStart(tcTmpToOut, ioStartAddr+EDGE_PAD);
      TcStart(tcTmpToOut);
    }

    /*
     * Set up next data to go into PP
     */
    TcWaitForFree;
    TcDstStart(tcWhereToPp, ioStartAddr+EDGE_PAD);
    TcStart(tcWhereToPp);

    lineStart++;
  }

  /*
   * Start PP on next data, transfer out processed data
   */
  newLines = 0;

  for (ppNum = 0; ppNum < lineOut; ppNum++) {
    ioStartAddr = (UBIN32)(ppState[ppNum][rrIndex]);
    ioDestAddr = (UBIN32)(ppState[ppNum][(rrIndex+2) % 3]);

    while (!(PP_ERR_HALT(ppNum) & PPERROR))       /* Wait for PP. */
      ;

    if (lineStart > 0) {       /* Start PP on next line. */
```

```
      ppParams[ppNum]->src = (unsigned char*)ioStartAddr;
      ppParams[ppNum]->dst = (unsigned char*)ioDestAddr;
      TcWaitForFree();
      command(CMD_PP_ADDR(ppNum) | CMD_UNHALT_OP);
      newLines++;       /* Inc. Lines to be trans. out. */
      --lineStart;      /* Dec. Lines started count. */
    }
    ioStartAddr = (UBIN32)(ppState[ppNum][(rrIndex+1) % 3]);

    TcWaitForFree();
    TcSrcStart(tcPpToPgm1, ioStartAddr+EDGE_PAD);
    TcStart(tcPpToPgm1);
  }

  lineOut = newLines;       /* Update Transfer out count. */

  rrIndex = (rrIndex+1) % 3;
}



/*
 * Set up vertical filtering
 */

/*
 * Decide how many passes are necessary
 */
if (numLinesIn <= LASTLINE/2) {
  num_vert_passes = 1;
  nLinesInPass = numLinesIn;
  nLinesOutPass = numLinesOut;
}
else {
  num_vert_passes = 2;
  nLinesInPass = LASTLINE/2;
  nLinesOutPass = (numLinesOut+1)/2;
}

/*
 * Enter vertical loop
 */
for (vert_pass=0;vert_pass<num_vert_passes;vert_pass++) {
  TcWaitForFree();

  /*
   * Set up parameters to make sure there is no problem with seaming
   */
  if (vert_pass == 1) {
    fracStart = (unsigned short) (incr*nLinesOutPass);
    srcOffset = EDGE_PAD;

    if (filtType == HALFBAND) {
      newCenter = nLinesInPass;
    }
```

```
    else
      newCenter = (short) ((nLinesOutPass*((1<<16)+incr))>>16);
    TcSrcStart(tcPgmToPp1, ProgramRAM+LMEMBLOCK*(newCenter-EP_to_start));
    if ((pass == 0) && (numPasses == 2))
      TcDstStart(tcPpToWhere, ProgramRAM+LMEMBLOCK*nLinesOutPass);
    else
      TcDstStart(tcPpToWhere, OutRAMBase+LMEMBLOCK*nLinesOutPass);
    nLinesInPass = numLinesIn-nLinesInPass;
    nLinesOutPass = numLinesOut-nLinesOutPass;
  }
  else {
    fracStart = 0;
    srcOffset = EDGE_PAD-(EP_to_start*8);
    TcSrcStart(tcPgmToPp1, ProgramRAM);
    if ((pass == 0) && (numPasses == 2))
      TcDstStart(tcPpToWhere, ProgramRAM);
    else
      TcDstStart(tcPpToWhere, OutRAMBase);
  }

  TcWaitForFree();
  tcPpToWhere->srcBcount = nLinesOutPass-1;
  tcPpToWhere->dstBcount = tcPpToWhere->srcBcount;

  /*
   * Initialize loop control variables
   */
  rrIndex = 1;
  colCnt = numPixOut/4-4;          /* Number of vertical blocks to process */
  colOut = 4;                      /* Number of blocks to be transfered out. */
  colStart = 0;                    /* Number of blocks started. */

  /*
   * Send in parameters, first four blocks
   */
  for (ppNum = 0; ppNum < 4; ppNum++) {
    ioStartAddr = ((UBIN32)(ppState[ppNum][0]));
    ioDestAddr = ((UBIN32)(ppState[ppNum][2]));
    TcWaitForFree();
    TcDstStart(tcPgmToPp1, ioStartAddr+EDGE_PAD);
    TcStart(tcPgmToPp1);
    ppParams[ppNum]->src = (unsigned char*)ioStartAddr;
    ppParams[ppNum]->dst = (unsigned char*)ioDestAddr;
    ppParams[ppNum]->fArgsPtr = ppParams[ppNum]->vFiltArgs;
    ppParams[ppNum]->passCount = 8;
    ppParams[ppNum]->numberOfFilters = 1;
    ppParams[ppNum]->vFiltArgs[0].ppFirArgs.fracStart = fracStart;
    ppParams[ppNum]->vFiltArgs[0].ppFirArgs.nPixels = nLinesOutPass;
    ppParams[ppNum]->vFiltArgs[0].srcOffset = srcOffset;

    /*
     * Wait until the data got to the PP then start the PP
     */
    TcWaitForFree();
    command(CMD_PP_ADDR(ppNum) | CMD_UNHALT_OP);
```

```
    }


    /*
     * Loop to process the vertical filtering
     */
    while (colOut) {       /* Columns left to transfer out */

      /*
       * Transfer data to PPs
       */
      for (ppNum = 0; (colCnt > 0) && (ppNum < 4); colCnt--, ppNum++) {
        ioStartAddr = (UBIN32) (ppState[ppNum][rrIndex]);

        TcWaitForFree();
        TcDstStart(tcPgmToPp1, ioStartAddr+EDGE_PAD);
        TcStart(tcPgmToPp1);

        colStart++;
      }

      /*
       * Start PPs processing new data, transfer out processeddata.
       */
      newCols = 0;       /* Temp. counting variable. */

      for (ppNum = 0; ppNum < colOut; ppNum++) {
        ioStartAddr = (UBIN32)(ppState[ppNum][rrIndex]);
        ioDestAddr = (UBIN32)(ppState[ppNum][(rrIndex+2) % 3]);

        while (!(PP_ERR_HALT(ppNum) & PPERROR))       /* Wait for PP. */
          ;

        if (colStart > 0) {       /* Start PP on next line. */
          ppParams[ppNum]->src = (unsigned char*)ioStartAddr;
          ppParams[ppNum]->dst = (unsigned char*)ioDestAddr;
          TcWaitForFree();
          command(CMD_PP_ADDR(ppNum) | CMD_UNHALT_OP);
          newCols++;       /* Inc. Lines to be trans. out. */
          --colStart;       /* Dec. Lines started count. */
        }
        ioStartAddr = (UBIN32)(ppState[ppNum][(rrIndex+1) % 3]);

        TcWaitForFree();
        TcSrcStart(tcPpToWhere, ioStartAddr+EDGE_PAD);
        TcStart(tcPpToWhere);
      }

      colOut = newCols;       /* Update Transfer out count. */

      rrIndex = (rrIndex+1) % 3;
    }
    TcWaitForFree();
  }
}
```

```
    /* Hardware Misc. */
    LEDGrnOff();        /* Turn off for free time. */

    if (RedLED) {    /* Toggle the Red LED. */
      LEDRedOff();
      RedLED = 0;
      OutPageOne();
    } else {
      LEDRedOn();
      RedLED = 1;
      OutPageZero();
    }

    if (page == (InPage & ReadStatusP())) {        /* Same as when we started? */
      /*
       * Still in the same frame as when we started processing
       * video (we got done ahead of time) so we just wait for
       * the next frame and start processing over again.
       */
      while (page == (tmpPage = InPage & ReadStatusP()))
        ;
    } else {
      /*
       * We took too long to process data, so let's wait for
       * a brand new frame and start all over again.  Actually
       * we wait for two frames so that we alternately write
       * into each output frame (Things really look ugly if we don't).
       */
      while (page != (InPage & ReadStatusP()))
        ;                   /* wait for this corrupted frame to finish */
      while (page == (tmpPage = InPage & ReadStatusP()))
        ;                   /* wait again so we write into the alternate page */
    }
      page = tmpPage;       /* Save new page number. */

    }
  }

  /* End of Main. */



/*
 * Function chooseFilter, chooses the right filter depending on the shrink factor
 */

const short*
  chooseFilter(float percent, const short* filtCoefs, short* coefs) {

  int intPercent;
  intPercent = (int) (percent*1000.0);

  if (intPercent >= 970) {
```

```
    filtCoefs = coef90_3kai;
    *coefs = 3;
  }
  else if (intPercent >= 860 ) {
    filtCoefs = coef80_4kai;
    *coefs = 4;
  }
  else if (intPercent >= 780 ) {
    filtCoefs = coef75_5;
    *coefs = 5;
  }
  else if (intPercent >= 710 ) {
    filtCoefs = coef71_6;
    *coefs = 6;
  }
  else if (intPercent >= 650 ) {
    filtCoefs = coef65_7;
    *coefs = 7;
  }
  else if (intPercent >= 600 ) {
    filtCoefs = coef60_8;
    *coefs = 8;
  }
  else if (intPercent >= 560 ) {
    filtCoefs = coef56_9;
    *coefs = 9;
  }
  else if (intPercent >= 530 ) {
    filtCoefs = coef53_10;
    *coefs = 10;
  }
  else if (intPercent > 500 ) {
    filtCoefs = coef50_11;
    *coefs = 11;
  }
  else if (intPercent == 500) {
    filtCoefs = halfband;
    *coefs = 11;
  }
  else {
    filtCoefs = coef90_3kai;
    *coefs = 3;
  }
  return filtCoefs;
}
```

```
/*
   FileName:  shrnkp.cp
*/

#include "global.h"
```

```
#include "protohw.h"
#include "mvphw.h"
#include "pphwc.h"
#include "shrnkMsg.h"
#include "video.h"


extern cregister volatile unsigned int COMM;
extern void pp_cmnd(long val);


const int ppParam[PP_COUNT] = {
  PP0_PARAMETER_RAM + GENERAL_PURPOSE_OFFSET,
  PP1_PARAMETER_RAM + GENERAL_PURPOSE_OFFSET,
  PP2_PARAMETER_RAM + GENERAL_PURPOSE_OFFSET,
  PP3_PARAMETER_RAM + GENERAL_PURPOSE_OFFSET,
};

void ppInToOutFunc(unsigned char* sPtr, unsigned char* dPtr,
                   const firArgList* firArgs);
void ppQbyLTapFunc(unsigned char* sPtr, unsigned char* dPtr,
                   firArgList* firArgs, short* Coefs);
void DecimateFunc(unsigned char* sPtr, unsigned char* dPtr,
                   firArgList* firArgs, short* Coefs);

void main(void)
{
  int                 ppNum;          /* Which PP is this. */
  unsigned int        PP_Bit;         /* Bit mask for this PP. */
  unsigned int        uCmnd;          /* Command word for ip. */
  shrnkMsgStruct*     argsPtr;        /* Pointer to Parameter Structure */
  UBIN8*              dstBuf;         /* Pointer to video output buffer. */
  UBIN8*              srcBuf;         /* Pointer to video source buffer. */
  int                 passCnt;        /* number of passes */
  int                 filt;           /* Used to cycle through filters */
  passArgs*           firArgs;


  /*
   * Initialize
   */
  ppNum = COMM & 7;                   /* Read PP number. */
  PP_Bit = 0x1 << ppNum;              /* Turn number into bit mask. */
  argsPtr = (shrnkMsgStruct*)ppParam[ppNum];

 /*
  * Halt the PP to synchronize with other PPs.
  */
  uCmnd = PP_Bit | CMD_HALT;          /* Set the HALT bit. */
  pp_cmnd(uCmnd);                     /* Issue the command. */


  /*
   * Enter endless processing loop
   */
```

```
  while (1) {

    /*
     * Set Src and Dst buffers.
     */
    srcBuf = argsPtr->src;
    dstBuf = argsPtr->dst;

    /*
     * Implement filtering for right number of passes and filters
     */
    for (passCnt = 0; passCnt < argsPtr->passCount; passCnt++) {
          firArgs = argsPtr->fArgsPtr;
          for (filt = 0; filt < argsPtr->numberOfFilters;  filt++) {
            if (argsPtr->filtType == PERIODIC)
              ppQbyLTapFunc(srcBuf + firArgs->srcOffset, dstBuf + firArgs->dstOffset,
                            &firArgs->ppFirArgs, argsPtr->fCoef);
            else
              DecimateFunc(srcBuf + firArgs->srcOffset, dstBuf + firArgs->dstOffset,
                            &firArgs->ppFirArgs, argsPtr->fCoef);
            firArgs++;
          }
          srcBuf++;
          dstBuf++;
    }

    /*
     * When done, halt the PP.
     */
    pp_cmnd( uCmnd );
  }
}



/*
 * Function:  ppInToOutFunc
 * Does no processing, simply copies input data to output buffer
 * Used for debug
 */

void
ppInToOutFunc(unsigned char* sPtr, unsigned char* dPtr, const firArgList* firArgs) {

  int      i;

  for (i = 0; i < firArgs->nPixels; i++) {
    dPtr[i*firArgs->pixSpace] = sPtr[i*firArgs->pixSpace];
  }
}



/*
 * Function:  ppQbyLTapFunc
```

```
 * Implements approximate polyphase filtering using L=16 Q-tap filters
 */

void
ppQbyLTapFunc(unsigned char* sPtr, unsigned char* dPtr, firArgList* firArgs,
              short* coefBase) {
  short                 i,j;
  int                   accum;
  short                 pixSpace;
  short                 nCoef;
  unsigned short        fracCounter;
  unsigned short        incr;
  short                 dataReturn;
  short*          currentCoefs;

  /*
   * Initialize variables
   */
  nCoef = firArgs->nCoef;
  pixSpace = firArgs->pixSpace;
  incr = firArgs->incr;
  fracCounter = firArgs->fracStart;
  dataReturn = pixSpace*(nCoef-1);
  currentCoefs = coefBase + (fracCounter>>12)*nCoef;

  /*
   * Process all data
   */
  for (i = 1; i < firArgs->nPixels; i++) {
    /*
     * Apply filter at current position
     */
    accum = 0;
    for (j=0;j<nCoef;j++) {
          accum += (*sPtr)*(*currentCoefs);
          sPtr += pixSpace;
          currentCoefs++;
    }

    /*
     * Write computed data point to output memory buffer
     */
    *dPtr = (unsigned char) (accum>>15);

    /*
     * Increment data and filter indeces
     */

    sPtr -= dataReturn;
    fracCounter += incr;
    if (fracCounter < incr) {
          sPtr += pixSpace;
    }
    currentCoefs = coefBase + (fracCounter>>12)*nCoef;
    dPtr += pixSpace;
```

```
  }

  /*
   * Apply filter at current position
   */
  accum = 0;
  for (j=0;j<nCoef;j++) {
    accum += (*sPtr)*(*currentCoefs);
    sPtr += pixSpace;
    currentCoefs++;
  }

  /*
   * Write computed data point to output memory buffer
   */
  *dPtr = (unsigned char) (accum>>15);
}



/*
 * Function:  DecimateFunc
 * Implements half-band filtering
 */

void
DecimateFunc(unsigned char* sPtr, unsigned char* dPtr, firArgList* firArgs,
             short* coefBase) {
  short                 i,j;
  int                   accum;
  short                 pixSpace;
  short                 nCoef;
  short                 dataReturn;
  short*          currentCoefs;

  /*
   * Initialize variables
   */
  nCoef = firArgs->nCoef;
  pixSpace = firArgs->pixSpace;
  dataReturn = pixSpace*(nCoef-2);
  currentCoefs = coefBase;

  /*
   * Process all data
   */
  for (i = 1; i < firArgs->nPixels; i++) {
    /*
     * Apply filter at current position
     */
    accum = 0;
    for (j=0;j<nCoef;j++) {
          accum += (*sPtr)*(*currentCoefs);
          sPtr += pixSpace;
          currentCoefs++;
```

```
    }

    /*
     * Write computed data point to output memory buffer
     */
    *dPtr = (unsigned char) (accum>>15);

    /*
     * Increment data and filter indeces
     */
    currentCoefs = coefBase;
    sPtr -= dataReturn;
    dPtr += pixSpace;
  }

  /*
   * Apply filter at current position
   */
  accum = 0;
  for (j=0;j<nCoef;j++) {
    accum += (*sPtr)*(*currentCoefs);
    sPtr += pixSpace;
    currentCoefs++;
  }

  /*
   * Write computed data point to output memory buffer
   */
  *dPtr = (unsigned char) (accum>>15);
}
```

```
/*****************************************************************************/
/*                                                                           */
/*        Module: TI MVP MP/PP Message structure.                            */
/*****************************************************************************/
/* Copyright Tektronix, Inc. 1995 */
/* company confidential */


#define EDGE_PAD            40      /* number of bytes at edge of waveform
                                     * needed clean filtering (at least half
                                     * the filter length) */

#define BLOCKSIZE           8       /* Width of column blocks, in bytes */
#define MAXFILTSIZE         176

enum method{AUTO, MANUAL, FIXED, NUM_METHODS};
enum filtType{PERIODIC, HALFBAND};


struct firArgList
```

```
{
        short                   fracStart;
        short                   nCoef;
        short                   pixSpace;
        short                   nPixels;
        unsigned short          incr;
};
typedef struct firArgList firArgList;

struct passArgsStruct {
        firArgList      ppFirArgs;
        short           srcOffset;
        short           dstOffset;
};
typedef struct passArgsStruct passArgs;

struct shrnkMsgStruct {
        short           filtType;
        unsigned char*  src;
        unsigned char*  dst;
        short           passCount;
        short           numberOfFilters;
        passArgs*       fArgsPtr;
        passArgs        hFiltArgs[3];
        passArgs        vFiltArgs[1];
        short           fCoef[MAXFILTSIZE];
};
typedef struct shrnkMsgStruct shrnkMsgStruct;
```

```
/*
 * Filename:  Coef_table.h
 * Filter coefficients to be used when resampling
 */


const short coef90_3kai[48] = {
  17068, 19061, -3361,
  15021, 20971, -3224,
  12944, 22743, -2919,
  10865, 24327, -2424,
  8819, 25671, -1722,
  6842, 26728, -802,
  4972, 27457, 339,
  3246, 27829, 1693,
  1693, 27829, 3246,
  339, 27457, 4972,
  -802, 26728, 6842,
  -1722, 25671, 8819,
  -2424, 24327, 10865,
  -2919, 22743, 12944,
  -3224, 20971, 15021,
  -3361, 19061, 17068,
```

```
};

const short coef80_4kai[64] = {
  4487, 25252, 6028, -2999,
  3079, 25280, 7733, -3324,
  1763, 25063, 9531, -3589,
  557, 24594, 11391, -3774,
  -523, 23872, 13280, -3861,
  -1462, 22902, 15157, -3829,
  -2250, 21701, 16980, -3663,
  -2879, 20291, 18705, -3349,
  -3349, 18705, 20291, -2879,
  -3663, 16980, 21701, -2250,
  -3829, 15157, 22902, -1462,
  -3861, 13280, 23872, -523,
  -3774, 11391, 24594, 557,
  -3589, 9531, 25063, 1763,
  -3324, 7733, 25280, 3079,
  -2999, 6028, 25252, 4487,
};

const short coef75_5[80] = {
  -1131, 16707, 18243, -899, -152,
  -1267, 15109, 19686, -557, -203,
  -1321, 13459, 20984, -91, -263,
  -1309, 11796, 22106, 510, -335,
  -1246, 10152, 23032, 1251, -421,
  -1147, 8559, 23743, 2135, -522,
  -1027, 7044, 24225, 3163, -637,
  -895, 5628, 24467, 4331, -763,
  -763, 4331, 24467, 5628, -895,
  -637, 3163, 24225, 7044, -1027,
  -522, 2135, 23743, 8559, -1147,
  -421, 1251, 23032, 10152, -1246,
  -335, 510, 22106, 11796, -1309,
  -263, -91, 20984, 13459, -1321,
  -203, -557, 19686, 15109, -1267,
  -152, -899, 18243, 16707, -1131,
};

const short coef71_6[96] = {
  -1449, 5644, 23122, 6931, -1573, 93,
  -1307, 4444, 22944, 8296, -1666, 57,
  -1150, 3338, 22567, 9712, -1718, 19,
  -990, 2337, 22000, 11162, -1715, -26,
  -833, 1449, 21257, 12623, -1648, -80,
  -683, 676, 20351, 14073, -1505, -144,
  -545, 21, 19301, 15489, -1276, -222,
  -422, -520, 18126, 16848, -950, -314,
  -314, -950, 16848, 18126, -520, -422,
  -222, -1276, 15489, 19301, 21, -545,
  -144, -1505, 14073, 20351, 676, -683,
  -80, -1648, 12623, 21257, 1449, -833,
  -26, -1715, 11162, 22000, 2337, -990,
  19, -1718, 9712, 22567, 3338, -1150,
```

```
  57, -1666, 8295, 22944, 4444, -1306,
  93, -1573, 6931, 23122, 5644, -1449,
};

const short coef65_7[112] = {
  -870, 62, 16390, 17382, 652, -1020, 172,
  -726, -440, 15330, 18293, 1330, -1173, 154,
  -590, -856, 14208, 19099, 2095, -1322, 134,
  -465, -1190, 13039, 19791, 2945, -1463, 111,
  -353, -1446, 11841, 20357, 3873, -1589, 85,
  -254, -1629, 10629, 20789, 4875, -1694, 52,
  -169, -1746, 9418, 21081, 5941, -1769, 12,
  -97, -1803, 8224, 21228, 7061, -1808, -37,
  -37, -1808, 7061, 21228, 8224, -1803, -97,
  12, -1769, 5941, 21081, 9418, -1746, -169,
  52, -1694, 4875, 20789, 10629, -1629, -254,
  85, -1589, 3873, 20357, 11841, -1446, -353,
  111, -1463, 2945, 19791, 13039, -1190, -465,
  134, -1322, 2095, 19099, 14208, -856, -590,
  154, -1173, 1330, 18293, 15330, -440, -726,
  172, -1020, 652, 17382, 16390, 62, -870,
};

const short coef60_8[128] = {
  -372, -1699, 8032, 19595, 9053, -1555, -482, 196,
  -273, -1792, 7031, 19486, 10083, -1354, -602, 189,
  -186, -1838, 6057, 19264, 11112, -1093, -731, 183,
  -110, -1842, 5119, 18934, 12127, -768, -867, 175,
  -46, -1809, 4225, 18499, 13119, -377, -1008, 165,
  8, -1746, 3381, 17966, 14075, 81, -1150, 153,
  52, -1657, 2593, 17339, 14986, 608, -1290, 137,
  88, -1549, 1867, 16628, 15840, 1204, -1425, 115,
  115, -1425, 1204, 15840, 16628, 1867, -1549, 88,
  137, -1290, 608, 14986, 17339, 2593, -1657, 52,
  153, -1150, 81, 14075, 17966, 3381, -1746, 8,
  165, -1008, -377, 13119, 18499, 4225, -1809, -46,
  175, -867, -768, 12127, 18934, 5119, -1842, -110,
  183, -731, -1093, 11112, 19264, 6057, -1838, -186,
  189, -602, -1354, 10083, 19486, 7031, -1792, -273,
  196, -482, -1555, 9053, 19595, 8032, -1699, -372,
};

const short coef56_9[144] = {
  -37, -1768, 2232, 15266, 15908, 2910, -1827, -102, 186,
  19, -1687, 1602, 14564, 16484, 3635, -1861, -177, 189,
  65, -1588, 1024, 13812, 16990, 4400, -1864, -263, 192,
  102, -1477, 501, 13016, 17420, 5203, -1833, -358, 194,
  132, -1353, 31, 12185, 17769, 6037, -1765, -463, 195,
  155, -1224, -383, 11326, 18035, 6896, -1656, -577, 196,
  171, -1091, -742, 10448, 18214, 7774, -1501, -699, 194,
  182, -957, -1047, 9558, 18303, 8664, -1299, -826, 190,
  190, -826, -1299, 8664, 18303, 9558, -1047, -957, 182,
  194, -699, -1501, 7774, 18214, 10448, -742, -1091, 171,
  196, -577, -1656, 6896, 18035, 11326, -383, -1224, 155,
  195, -463, -1765, 6037, 17769, 12185, 31, -1353, 132,
```

```
  218, -99, -1752, 601, 10668, 16280, 8586, -515, -1483, 81, 183,
  207, -31, -1675, 191, 9986, 16341, 9290, -181, -1584, 29, 195,
  195, 29, -1584, -181, 9290, 16341, 9986, 191, -1675, -31, 207,
  183, 81, -1483, -515, 8586, 16280, 10668, 601, -1752, -99, 218,
  171, 124, -1374, -809, 7878, 16157, 11333, 1050, -1815, -175, 228,
  159, 161, -1261, -1066, 7172, 15974, 11975, 1535, -1859, -259, 237,
  147, 190, -1141, -1285, 6471, 15733, 12589, 2055, -1884, -351, 244,
  136, 213, -1021, -1467, 5780, 15435, 13171, 2608, -1885, -450, 248,
  124, 230, -901, -1615, 5104, 15083, 13717, 3194, -1862, -556, 250,
  113, 241, -783, -1729, 4446, 14680, 14223, 3807, -1810, -667, 247,
};

const short halfband[11] = {
  295,    0, -1876,    0,  9780, 16370,  9780,    0, -1876,    0,  295,
};
```