A Low Power Video Compression Chip for Portable Applications

by

Thomas Simon

Bachelor of Science in Electrical Engineering
Massachusetts Institute of Technology, June 1990

Master of Science in Electrical Engineering and Computer Science
Massachusetts Institute of Technology, September 1994

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

Doctor of Philosophy in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

June 1999

Signature of Author _____
Department of Electrical Engineering and Computer Science
April 30, 1999

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

Accepted by _____
Arthur Clarke Smith, Ph.D.
Professor of Electrical Engineering
Graduate Officer

A Low Power Video Compression Chip for Portable Applications

by

Thomas Simon

Submitted to the Department of Electrical Engineering and Computer Science
on April 30, 1999 in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Electrical Engineering and Computer Science

Abstract

This thesis describes the development of a low power, single chip video encoder intended for battery operated portable applications. Such an encoder is intended to serve as part of the DSP in a portable device which might image, data convert, compress, and transmit video signals. The encoder described in this thesis is designed with the goal of minimizing system power, minimizing utilized bandwidth, and maximizing system integration. The encoder achieves a peak power dissipation of several hundred μW while scalably compressing a video stream of 8 bit gray scale, 30 frame/sec, and 128x128 demonstration resolution. The encoder scales up for greater resolutions at mostly linear cost. The compression is performed using wavelet filtering and a combination of zero-tree and arithmetic coding of filter coefficients, all integrated on a single demonstration chip. The compression results achieved (a tradeoff curve of compression factor versus PSNR) are on par with the best available based on wavelet filters. The above results do not include the use of motion compensation, however, hooks are implemented at the algorithmic and architectural levels to add motion compensation at the cost of power dissipation a few times higher. These results are obtained by the careful coordination of design in a deep vertical manner, ranging from system, algorithmic, and architectural to circuit, floor planning, and layout. This thesis describes the motivation of the design goals, the interlinking vertical design choices, and the results achieved.

Thesis Supervisor: Anantha Chandrakasan
Title: Associate Professor of Electrical Engineering

Acknowledgments

I would like to thank Prof. Anantha Chandrakasan for agreeing to supervise this thesis and for displaying great patience throughout. I would also like to apologize for so much patience having been required of him. Anantha managed to improve the quality of my thesis despite my stubbornness and impatience.

I thank Dr. Ichiro Masaki for generously agreeing to read my thesis.

I owe unending gratitude and credit to my family. Without the support and understanding of my parents Marta and George, and my sister Rita, it is unclear that I could have finished this thesis at all.

I also thank Prof. Arthur Smith for his advice and kindly understanding. It is clear that I would not have finished were it not for 1/2 an hour of his time at a key moment. Of course, the same goes for Marilyn Pierce, except for the better part of a decade.

I would like to thank all my friends and colleagues for support and contributions great and small. Thanks to: Duke Xanthopoulos, Raj Amirtharajah, Vadim Gutnik, Jim Goodman, Scott Meninger, Matt Frank, Dan McMahill, Don Hitko, Rex Min, Kush Gulati, Jim MacArthur, Wendi Rabiner, Mike Bolotski, Oscar Mur-Miranda, Amit Sinha, Nate Shnidman, Alice Wang, Gangs Konduri, SeongHwan Cho, and Josh Bretz. If you should be on this list and are not, forgive me and stop by my office and smack me (no hockey sticks).

I thank Profs. Charlie Sodini and Harry Lee for contributing to my non-technical education.

Finally, very special thanks to Dr. Thomas Knight for reading my thesis and more importantly teaching me wonderful bits from his large bag of engineering tricks. TK is one of the most creative people I have worked with at MIT, as well as a great educator and a jolly good fellow. I'm not sure which we should be most thankful for, fortunately in TK's case we do not have to choose.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Greater demand from today's portable computation and communication systems is progressively increasing the requirements on long battery lifetime, efficient bandwidth use, and integration. On the other hand, the desirability of information sources such as real time video introduce huge amounts of data for Digital Signal Processing systems. The goal of this thesis is to develop a DSP framework to process real time video data for wireless portable applications. The desired solution should be highly integrated, exhibit very low system power across relevant modes of operation, as well as minimize RF bandwidth use for the transmission of the video data. The solution proposed in this thesis addresses the problem broadly at the system, algorithmic, architectural, and circuit levels.

An example context for the encoder is shown in Figure 1.1. An imager senses the video stream which is then digitized and compressed. Additional signal processing (DSP) may be performed before RF transmission. This may include encryption, error-correction

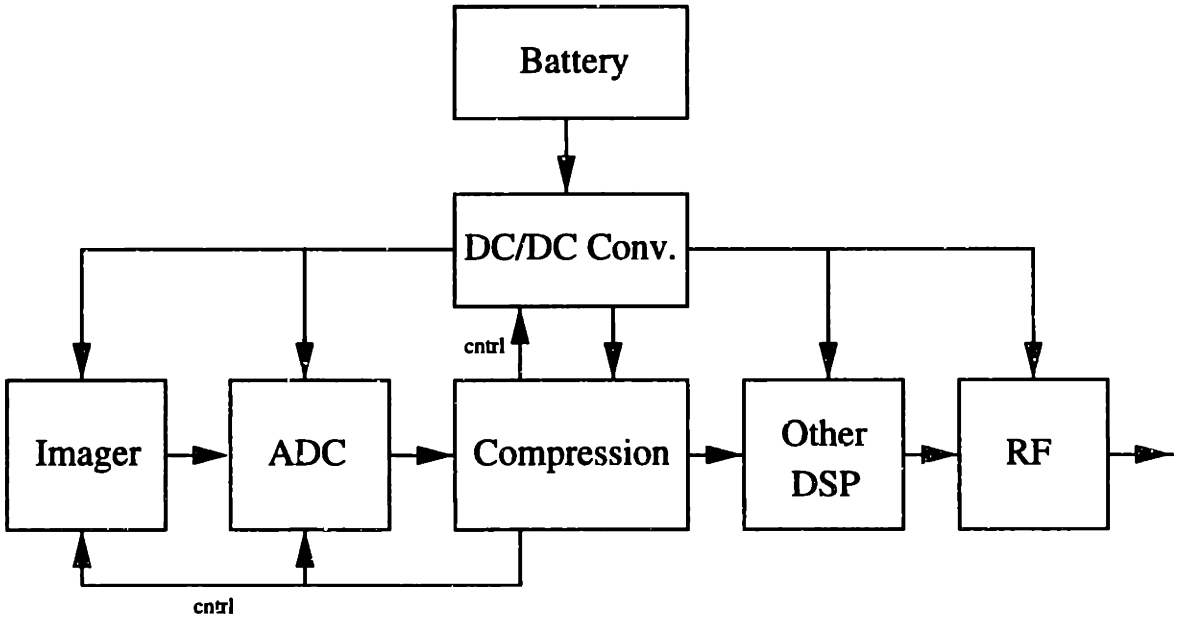


Figure 1.1: Block diagram of a portable video sensor.

coding, framing, etc ... The RF transmitter communicates the data to some remote base station which may be receiving and transmitting to multiple sensors and other base stations. Figure 1.1 only shows the sensing and outgoing half of the device. There may also be a receiving and display half. The two halves are not completely independent. A back channel may also be used by the base station to deliver on the fly control information to any part of the sensor.

The design objectives are low system power for long battery lifetime, and closely related, low transmission bandwidth. Meeting these objectives will depend on the performance of the solution in two distinct modes of operation: in the presence of significant motion in the video data source, and the absence of such motion, which for some applications such as remote surveillance may represent a large duty cycle.

1.2 Overview

During periods of motion in the video stream, the transmission of data is expected to dominate system power. The transmission power required is limited by fundamental noise sources, as a function of distance and environment. As such, technology scaling cannot significantly lower this power in the same way as the local computational power. For very short distances, the transceiver circuit power may become the dominant factor. In this case, system power, and obviously bandwidth, are roughly linear in the amount of transmitted data.

System performance in this mode is optimized by achieving good data compression. This means compressing the video stream to the smallest possible size for some desired visual quality level, which may be set on the fly by the receiving base station, for example. The requested visual quality may be a function of a-priori application requirements, feedback on the quality of the communication channel, or feedback on the available bandwidth due to other users. Chapter 3 describes the compression algorithm used to achieve this goal. The algorithm is based on wavelet filtering and uses a combination of zero-tree and arithmetic coding to compress the filter coefficients.

In addition, in this operating mode, system power is also optimized by performing the imaging, data conversion, and DSP with negligible dissipation relative to the transmission power, if possible. Chapters 4 through 6 describe how this goal is achieved for the video encoder. The architectural basis of the implementation is an efficient mapping of most of the computation onto a very fine granularity SIMD array of simple 12 bit wide processing elements (PE). The partitioning assigns the computation associated with a small rectan-

gular block of pixels in each frame to one PE. For the demonstration implementation this mapping is a 4x4 pixel block per PE.

This fine partitioning, which is enabled by the compression algorithm, provides the parallelism needed to operate both memory and logic circuits slowly, at low voltage, while still meeting the required computational throughput. In fact, in the fabricated test chip, supply scaling is limited by the technology's device thresholds. The low power dissipation of the encoder is due in large measure to the energy efficiency of operating at low voltage, as well as highly localized patterns of memory access and communication.

In the absence of motion in the data stream, the result of compression should yield almost no output data. With a largely inactive transmitter, system power is reduced to the computation power used by the DSP, imager, data converter, and power supply overhead. This power, despite being possibly much lower than the active transmission power, can be very important if a device spends most of the time detecting little motion. Chapter 2 describes techniques applicable to the video encoder which can minimize the standby power for itself and the other system components.

1.3 Demonstration Implementation

The testing and performance of a demonstration encoder are reported in Chapter 7. The demonstration scalably compresses 8 bit gray scale, 128x128 resolution, 30 frames/sec video data, with dissipation of a few hundred μW. The compression rate can be scaled on the fly between factors close to 10, resulting in visually perfect reproduction, to highly lossy factors above 100.

In order to ward off skepticism, it is worth commenting early on the fairly small pixel resolution of the implementation. The SIMD array of PEs, each consisting of a small register file and a simple ALU, essentially constitute a large memory and some embedded logic. The total memory capacity distributed in the array is several 12 bit wide frame buffers (3 for the test chip) required to store and perform intermediate computations for each video frame. As such, the chip can be thought of as a memory with very finely divided subarrays and very large bandwidth utilized locally near the memory elements.

This application is a perfect candidate for mixed DRAM/logic CMOS process technology. However, due to availability in the academic environment, not only does the test chip not take advantage of mixed memory/logic, it is fabricated in a fairly unaggressive 1.8 micron minimum metal pitch technology. Therefore, the resolution of the test chip is limited by area. With 128x128 resolution the test chip occupies roughly $1cm^2$.

However, the dissipation of the demonstration chip is still indicative for a number of reasons. First, given the area, if the resolution were hypothetically scaled up with the same technology, most of the power dissipation would only scale up linearly with total resolution. Only a small proportion would grow quadratically (see Section 5.7).

Second, technology benefits to both memory and logic would greatly reduce the dissipation for a fixed resolution because of reduced area (shorter wires) and lower device thresholds enabling more supply voltage scaling. It should also be noted here that the test chip fabrication technology has fairly high device thresholds (.8V to .9V) so there is room to go before hitting subthreshold leakage bottlenecks. In addition, the large parallelism obtained from the fine granularity SIMD array architecture enables aggressive supply scaling.

Even in the crufty implementation technology, the test chip circuitry does not greatly exercise device performance. This allows the test chip to be operated with supply close on the heels of device thresholds (chip functionality was verified as low as 1.3V). This means that device threshold reductions due to process scaling would readily translate into substantial supply scaling and power savings.

To a large extent, it is expected that a resolution scaled up version implemented in a modern mixed memory/logic process would experience similar power levels, with some complications. Total area would remain the same since technology benefits would be spent on resolution. Most of the system power is in decoded instruction distribution to the SIMD array, in the form of global densely loaded wiring. This would suffer from greater per length wiring parasitics, but lower device loading. Local wiring within and between PEs would also suffer greater parasitics, but for shorter distances.

The greatest complication arises from the uneven scaling of memory and logic. A migration to a mixed DRAM/logic technology would benefit memory elements more than logic. This would dictate a different choice of logic to memory ratio, which is principally embodied in the size of the pixel blocks assigned to each PE. That choice is made by subtracting the total required memory area for a given resolution from the available reticle area. The remainder may be divided into as many logic elements as possible. With uneven scaling of memory and logic, it is expected that the work load per PE will go up. However, this relative loss of parallelism (higher required throughput per PE) would not translate into higher operating voltage and therefore energy per operation, due to the offsetting effect of lower device thresholds.

To summarize, there are a number of competing and offsetting effects involved in scaling up resolution with a modern process technology. There would be more circuitry, but with finer feature sizes, especially for memory elements. There would be more operations, but operations would be performed with shorter local wiring and smaller devices. There would be less parallelism per image pixel, but the computational throughput would be met by faster devices with lower thresholds. The detailed effects of scaling on power dissipation depend on the technology parameters of a prospective process.

1.4 Previous Work

Efforts at addressing video compression for low power have so far resulted in piecemeal implementations of algorithmic subcomponents or have yielded unsatisfactory power reduction. As described, the aim of this thesis is to deliver very high compression for a given video quality, with bit-rate scalability, including possible use of motion compensation, fully integrated, and at power levels modest even for short range RF transmission. No previous solution is known which delivers these goals.

For example, the matching encoder and decoder designs in [1] [2] use power levels comparable to our entire power target to perform a quantization compression step utilizing a Vector Quantization algorithm. This quantization is performed for filtered coefficients whose computation is not included in the designs. Further, their encoder delivers a fixed compression rate output which is not suitable for wireless applications in many cases of interest. The decoder in [3] also uses the Vector Quantization method to perform the same computation step with the same power levels.

The design reported in [4] computes the Discrete Cosine Transform of image frames to produce coefficients which could be used as inputs to the Vector Quantizers described above. The reported power dissipation at video frame rates is 10mW. A much more aggressive DCT implementation is presented in [5]. Distributed arithmetic is used to scale the complexity of the operations performed on-the-fly, and therefore the power, taking hints from the quantization applied and the spatial content of the image data. The implementation transforms 640X480, 30 frames/sec, dissipating just over 4mW.

[6] performs the computations required for motion estimation and compensation. The reported dissipation is 40mW. However, this does not include the contribution of required off chip memory which would likely be more than 40mW itself.

[7] computes a more complete compression algorithm (though not including motion compensation) for both encoder and decoder, side by side. Compression is performed using 3 dimensional wavelet filtering, scalar quantization, and run-length Huffman coding. The power dissipation for both encoding and decoding is 100-200mW including separate memory. The storage requirements are aggravated due to the choice of 3D filtering instead of frame to frame motion compensation.

[8] gives a design for both encoding and decoding, including full search motion estimation and compensation. At 144x176 resolution, 15 frames/sec, color images, they report a power of 1.2W. This again does not include separate memory or a coprocessor which performs the late quantization and rate control functions on the output bit stream.

By contrast, the total functionality of the encoder reported in this thesis is shown in Figure 1.2 which is a superset of the works listed above. The single chip with measured

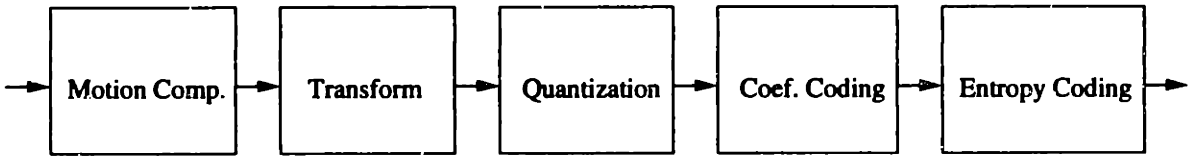


Figure 1.2: Total functionality of proposed encoder.

dissipation less than 1mW, performs all the algorithmic components except motion compensation. That can be incorporated at the expense of silicon area and several times more power, but with no significant changes to architecture or circuits. In all cases, the proposed encoder does not rely on any external memory or coprocessor.

Chapter 2

System Considerations

2.1 Introduction

The goal of this work is not to design a video encoder with low power dissipation per se, but to optimize for the system power of a portable device that might include the encoder. This requires optimizing in two different regimes of operation, with and without the presence of motion in the video stream.

Operation in the motion regime requires good compression performance because the amount of transmitted data determines the required data rate and duty cycle of the radio. It is expected that communication power will dominate local computational power due to fundamental physical range and noise constraints on wireless transmission, or transceiver circuit power in the case of very short ranges. In this regime, the burden on the encoder and other system components is to meet peak sensing and computational loads with dissipation levels that do not compete with the RF transmitter.

Operation in the motionless regime requires the encoder and other system components

to operate with as little dissipation as possible. This is because, with a largely inactive transmitter, the local sensing and computational power becomes the entire system power. However, for some applications, a sensor may spend a large fraction of its battery lifetime not sensing any motion in its field of view. In this case, this quiescent power may largely determine battery lifetime.

For example, for a video format of 350x240 resolution, 30 frames/sec, and 8 bits/pixel, a modest compression factor of 20 during periods of activity results in a transmission bit rate of 1 Mb/sec. On the other hand, in the absence of motion, compression can result in as little data as a few bits per frame, or about 200 bits/sec.

Note that determining whether there is motion or not requires the use of some kind of thresholding to flag the no motion case. Appropriate motion thresholds would prevent noise in the visual scene or the imager from triggering frivolous and wasteful use of energy and bandwidth. No attempt is made in this thesis to explore what motion thresholds should be. That would be determined by considering application specific requirements and the noise performance of the employed imager or even the data converter.

Note also that the preceding discussion makes clear the need for buffering of outgoing data and efficient transmission of small packets. Therefore, it is incumbent on the transmitter to avoid lengthy, expensive turn on transients. Buffering and packetization is required even during peak operation, since compressed data is expected to be quite bursty. This becomes critical during quiescent periods, as the token 200 bit/sec data rate cannot be allowed to trigger repeated radio turn on transients and packetization overheads.

However, buffering will result in greatly varying latency to the base station, as a func-

tion of data rate. For example, if the choice during no motion periods is to send 1 second of data in 200 bit packets, the worst case latency increases from about 1 frame plus transmission time to 30 frames plus transmission time. For applications which cannot tolerate the worst case latency, the base station may have to interpret large delays in packet arrival as a sign of no motion in the video data (in which case the actual delivery of the token packet serves merely as a "still alive" signal).

2.2 System Optimization with Motion

The following chapters describe how peak power is minimized in the encoder. Here, a brief survey is taken of optimistic (lowest) dissipation levels of other likely system components, as a gauge to justify power goals for the encoder.

The most stringent power case for RF transmission is small range and relatively predictable environment, such as indoors. No ultra low power solutions to this problem have been found in the literature. However, cues for low power, short range transmitters are taken from low power receiver designs [9] which share symmetry with circuit structures relevant to transmission, and from reported developments of underlying transmitter circuit structures [10] [11]. From these, it is not unreasonable to conclude that under the most favorable circumstances 1 Mb/sec data rates can be supported with less than 20 mW power levels.

Literature surveys for other system components reveal the following peak dissipations: CMOS imagers [12] [13] [14] - less than 1 mW, video rate and precision data conversion [15] - less than $\frac{1}{2}$ mW, other DSP functions such as stream cipher encryption [16] - less than

Component	Peak Power Cost
RF	< 20mW
Imager	< 1mW
ADC	$< \frac{1}{2}$mW
Encryption	$< 100\mu$W
Supply Overhead	$< 10\%$

Table 2.1
Peak power of other system components.

100 μW, and power supply overhead for low current, low voltage, and wide load range [17] [18] [19] - less than 10%.

Table 2.1 summarizes the components of peak system power which compete with the compression function in the energy budget. Excluding the components which process compressed data (radio, ECC, encryption), these numbers also give an indication of what can be expected during quiescent periods of little motion if no special steps are taken for that case. In order to keep the impact of the encoder negligible during peak periods, it is not unreasonable to target a dissipation of less than 1 mW.

It should be noted that a relative comparison such as this also determines the efficacy of including the use of motion compensation in the algorithm presented in the next chapter. The several fold multiplication of the compression power must be weighed against the expected additional compression of the output data and the accompanying reduction in the transmission cost. The expected benefit to compression performance can be a strong function of the visual content in the video source [20] [21].

2.3 System Optimization with no Motion

For quiescent periods with no motion, the power dissipation of the encoder benefits from all the architectural and circuit techniques brought to bear to reduce peak power, but with the added advantage that many fewer operations need be performed. The operations required to compute a frame difference and some motion thresholding are a small subset of the compression computation. The motion compensation, wavelet filtering, zero-tree coefficient encoding, and arithmetic coding are all omitted if the difference of the current and previous frames falls bellow the motion threshold.

In addition, the video encoder can feedback information about the presence or absence of motion to the system in order to further lower its own quiescent power and that of the imager, data converter, and power supply overhead. For example, after an extended period with no motion, the frame rate throughout the system can be lowered until new motion is detected. Of course, excessive decimation of the frame rate may allow quick bursts of motion to go undetected. The slower frame rate also determines the latency with which new motion is detected and transmitted. The severity of both these constraints may be strong functions of the specific application. No attempt is made in this thesis to explore these limits.

It should be noted that even with generous allowances for frame rate decimation, substantial power saving may be an elusive goal. Ideally, the lower frame rate should enable slower operation of all circuits, and therefore greater energy efficiency per operation, as well as fewer operations. Any number of obstacles, such as poor power supply efficiency at greatly lower average current levels, or device threshold limits to supply scaling, may

make slower operation difficult. Even the non-ideal case of same speed but bursty operation at the lower frame rate requires at least that circuitry can be turned off (with no DC power dissipation) and back on without long startup transients. The problem of pursuing energy efficiency past voltage scaling limits may be addressed by adiabatic circuit techniques [22] [23] [24].

With the same caveats concerning the translation of fewer operations into lower dissipation, two other techniques are suggested for optimizing quiescent system power during long motionless periods - spatial decimation and reduced precision. The idea of spatial decimation would be to reduce the resolution of the image which is being sensed, converted, and differenced while the system is searching for new motion. The full resolution would be restored as soon as new motion was detected. The lower resolution might be arrived at by averaging pixels, or true decimation. Similarly, the search for new motion might be carried out with lower bit precision than that required to visually represent the image. The applicability of these techniques is not studied in this thesis, and is left for future work.

2.4 Error Correction and Synchronization Recovery

Because of the wireless transmission of the compressed data, it is assumed that the communication cannel is subject to fairly high bit error rates (BER). No study of optimal error correction or detection coding is made here. However, the compression algorithm presented in the next chapter provides two mechanisms needed to operate in a high BER environment.

The compression algorithm is easily scaled on the fly. The compression rate can be traded off for visual quality over a wide range. This tradeoff is carried out with good compression performance and energy efficiency at all points along the tradeoff curve. This convenient on the fly scalability is useful, among other things, for responding to a time varying channel quality by changing the output data rate.

In addition, the compression algorithm provides periodic checkpoints at which the encoder and base station decoder can become resynchronized after uncorrectable errors. This limits the maximum loss of visual data to the time between synchronization points. As will be shown in the following chapter, there is a tradeoff between this time and compression since the synchronization points cost bits. The choice of this time period is application specific. The relevant factors are the probability of uncorrectable errors and the maximum allowable loss of data.

Chapter 3

Algorithm

3.1 Introduction

The system considerations outlined in the previous chapter present several goals for the compression algorithm:

The resulting compression factor versus video quality must be good. This is important because, during peak periods of motion in the video stream, the amount of compressed data to be transmitted essentially determines system power. Video data rates, even after compression, can be fairly high (perhaps 1 Mb/sec), so that wireless transmission will dwarf the power dissipation of local computation.

Further, the power cost of transmission is lower bounded by fundamental physical constraints such as noise and range. While current technology has not reached these bounds, further improvements would depend on clever use of technology and would only benefit from simple fabrication process scaling to some extent. On the other hand, the bound on local computation is much lower [25] [26] [27], and computational power benefits vigorously

from process scaling as well as cleverness. The cost of large amounts of local computation, as shown in Table 2.1, is already smaller than transmission, and this disparity is expected to grow.

Therefore, a tradeoff of increased local computation for reduced data rate is pursued throughout this work, where such a tradeoff is lucrative. Some opportunities for this tradeoff are suggested even if they do not payoff in current technology. In such cases, it is expected that those techniques would become relevant as the cost of computation and communication diverge. This tradeoff influences major architectural choices as described in the following chapter. Algorithmically, the tradeoff motivates the inclusion of motion compensation despite its relatively heavy computational burden.

On the other hand, a second competing algorithmic goal is to keep the power dissipation of the local computation as small as possible. This is important for two reasons. During peak motion periods, the cost of local computation should not compete with communication in the power budget. During possibly long quiescent periods with no motion, the wireless link remains largely unused, and the computational cost of detecting new motion determines the system power.

Methods by which quiescent power can be minimized in the encoder and rest of the system were outlined in the previous chapter. Here, the techniques used to keep the peak computational power cost relatively small are presented. This optimization often entails the opposite tradeoff than the one discussed above. Several examples are presented throughout this chapter of marginal data rate reductions being discarded because of their large computational cost.

There are several connected properties of an algorithm which further the goal of low computational cost:

- The most basic power efficiency is to minimize the total number of operations performed.

- Not all operations can be performed at the same cost. So equally important, arithmetic and logical operations should be kept as simple as possible. Where the opportunity arises, addition and shifting are preferred to multiplication and division. Similarly, integer operations of as little precision as required are preferred to floating point.

- Even on chip, communication can account for as much, or more, power as computation. The most effective algorithmic way to control the cost of communication is to minimize mutual interactions between large amounts of data, or movement of data over large distances. Some optimizations are evident from first principles, while others require a model of the architectural and circuit implementation. While these are described in detail in succeeding chapters, minimal overviews are used to justify algorithmic choices. The vertical, coupled design approach taken in this thesis requires this mingling of subject material.

- It is important for the algorithm to be conveniently parallelizable. Parallelism enables voltage scaling, which is a principle architectural and circuit level energy efficiency. Parallel computation allows the use of many slower, and lower energy per operation, circuits to meet the total throughput needs of the application.

A third principle algorithmic goal is data rate versus video quality scalability. On-the-fly scalability can be used to tailor data rate to user demand. In addition, scalability can also be used by the system in feedback to respond to changes in communication channel quality or crowding due to other users.

The remainder of this chapter describes how these goals are met, by first describing a two dimensional image coding algorithm, and then the extension to time for sequences of video frames.

3.2 Basic EZW Image Coding Algorithm

3.2.1 Advantages

The basis of the image compression algorithm is the EZW algorithm reported in [28]. The EZW algorithm is chosen as a starting point for 3 reasons:

First, as shown below, this algorithm achieves very good compression, being competitive with the best know results at the time this work was begun. Further, because this algorithm is based on a hierarchical, multi-resolution wavelet transform as described below, not only are the compression resluts good in a numerical sense (according to distortion metrics such as PSNR), but highly compressed images avoid some of the most bothersome visual artifacts associated with blocked transform schemes [29] [20], such as the block DCT used in the JPEG standard [30].

Second, like other basic numerical transform algorithms, including DCT, the EZW algorithm is conveniently parallelizable and amenable to circuit implementation. EZW has

been found to be computationally efficient in the number of operations per pixel. In addition, the most burdensome costs associated with the complexity of operations and especially with the locality of communication have been found to contribute little to compression performance. Modifications and omissions from the algorithm reported in [28] are described below.

Finally, the EZW algorithm is inherently scalable. In fact, the algorithm produces an embedded output bit stream. Embeddedness is a more rigorous superset of the scalability property needed for this application. An embedded output stream contains all lower accuracy encodings as its prefix. That is, the encoded bits come out in order of importance. Each output bit produced is the next most useful one according to some numerical distortion metric (such as PSNR for the algorithmic parameters suggested in [28] and adopted here). If the bit stream is terminated at any point, for example at some target bit rate per frame, the resulting output is the best one the algorithm is capable of producing.

This is the most rigorous form of scalability, being applicable at the bit level. For this application, it is satisfactory to obtain scalability at coarser granularities. Some algorithmic modifications motivated by computational and communication cost are described which sacrifice bit level embeddedness, but preserve it coarsely. Scalability is achieved, without sacrificing compression performance, by limiting compression choices to the granularities which preserve embeddedness. These choices are known a priori and are convenient to dial up on-the-fly.

3.2.2 Overview

Before describing the detailed operation of the EZW algorithm, the main ingredients of its success are summarized.

Wavelet Transform

The use of discrete wavelet transforms to represent images provides multiresolution decompositions which compactly represent the image content at all scales, and without the imposition of arbitrary block boundaries. A wavelet transform is recursively applied to obtain a hierarchical subband decomposition with octave spaced bands. This multiresolution decomposition can represent low frequency narrow band information, such as textures and background, and visually important wide band information, such as edges, equally well. An excellent qualitative overview of the power of hierarchical wavelet decompositions can be found in [28] itself. For rigorous mathematical treatments, the reader is referred to [31] [32] [33] [34].

Successive Approximation Quantization

The embedded property is derived from the use of successive approximation quantization. While this is a fairly general notion, the realization in the EZW algorithm essentially boils down to bit plane encoding. The computed wavelet coefficients are encoded one bit at a time, from MSB to LSB. A key to producing an embedded output is that a bit plane is fully encoded for all coefficients before the next bit plane is begun for any coefficient. This ordering is motivated by the relative importance of MSBs of other coefficients versus

further resolution of any single coefficient regardless of its magnitude or spatial frequency.

Within a given bit plane, the EZW algorithm imposes other prioritizations on the encoding of coefficients. All these encoding priorities contribute to embeddedness by targeting the most significant distortion reductions first. These intra bit plane priorities are: new significants second - identification of new non-zero coefficients is encoded after further resolution of those determined to have non-zero magnitudes in previous bit planes, scale - new non-zero coefficients are encoded from low to high frequency, magnitude - coefficients already encoded with a non-zero value from preceding bit planes are further resolved in descending magnitude order, spatial location - this is the default catch all, if nothing else distinguishes coefficients, they are encoded in some a priori scanning order agreed on by encoder and base station decoder.

The successive bit plane quantization, scale, and spatial location prioritizations are key, and fortunately not computationally burdensome, and are adhered to. The new significants second and magnitude prioritizations, however, are omitted to reduce complexity and therefore power, as described later in this chapter. The new significants second ordering is incompatible with architectural choices, while the magnitude prioritization is rejected from first principles, with an eye toward the localization of communication patterns. The omission of these two priorities results in the loss of a completely embedded output encoding within bit planes. In order to achieve scalability without compromising compression performance, compression choices are limited to the bit plane boundaries.

Zero-Tree Significance Maps

A large part of the good compression results obtained with the EZW algorithm can be attributed to the use of zero-trees. These tree data structures are imposed on the wavelet coefficients according to frequency and location. The trees organize coefficients corresponding to the same area of the image from low frequency at the root, to high frequencies at the leaves, and are intimately tied to scale prioritization.

During encoding of new non-zero coefficients, a progression is made from tree roots (low frequency) to leaves (high frequency). The ordering of low frequency components before high ones at the same location not only contributes to embedding, but also allows an important conditioning of high frequency components with the low frequency coded magnitudes. The zero-tree coding scheme, described below, takes advantage of statistics found in natural images [20] [29] [28] by predicting the insignificance of high frequency components from the insignificance of low frequency ones. That is, the use of zero-trees effectively embodies into the algorithm a statistical image model which predicts that small amplitude low frequencies are accompanied by small magnitude high frequencies in any given area of a frame. By incorporating this model into the coding of new non-zero coefficients, the EZW algorithm greatly reduces the cost of the significance maps of those coefficients.

Adaptive Arithmetic Coding

An adaptive arithmetic coder is used as the final lossless entropy coding compression step in the EZW algorithm. The adaptivity is important to optimize compression performance

in the face of non-stationary image statistics, without requiring training of the algorithm on sample images beforehand. Training is a viable strategy for database applications which compress libraries of images to reduce storage size and transmission bandwidth. However, a priori training for a portable sensor is unworkable.

The use of an arithmetic coder is important because the EZW algorithm can produce symbols with probabilities close to 0 or 1, especially for high compression ratios. Arithmetic coders can approach bit rates close to symbol entropies for these cases easily and elegantly. Adaptive Huffman and run-length codes are very awkward when faced with wide dynamic ranges in symbol probabilities. Compound Huffman codes may have to be used for groups of symbols. The result for both Huffman and run-length codes is large alphabets.

By contrast, the EZW algorithm itself outputs symbols from a fairly small alphabet, as described below. The preservation of small alphabet sizes by the arithmetic coder is an advantage for adaptivity, since it takes less symbols to acquire good probability estimates. The use of small alphabet sizes also turns out to be an enabler for a modification to the arithmetic coder, described below. The modification, motivated by power efficiency, approximates symbol probabilities to reduce the complexity of mathematical operations in the coder. These approximations are more accurate for small alphabets.

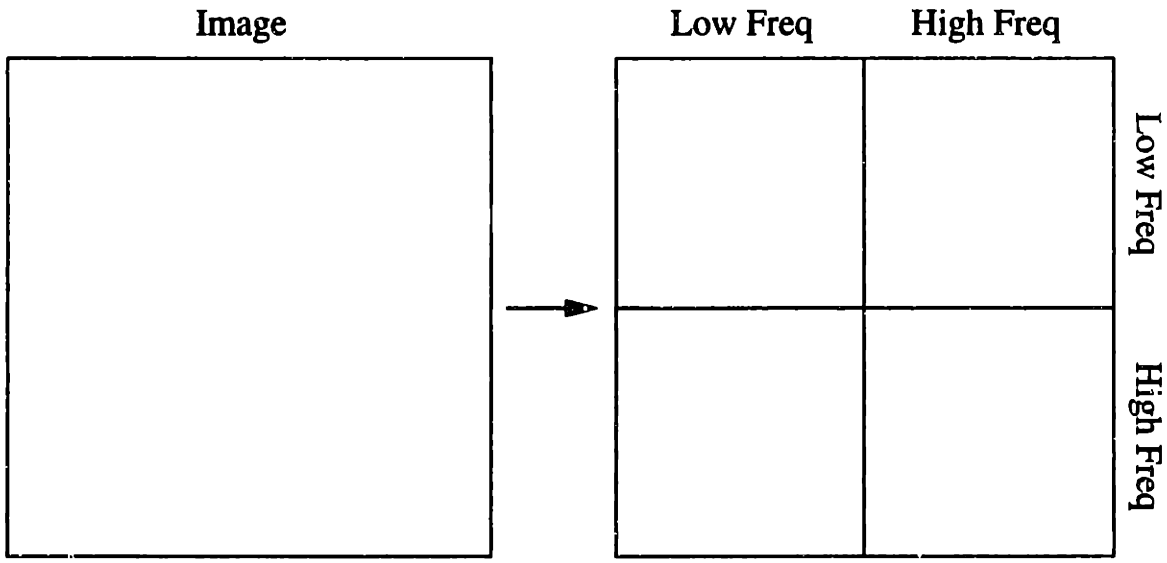


Figure 3.1: Result of 1 level of wavelet filtering. Separable horizontal and vertical filters are used to compute critically subsampled low and high frequency subbands.

3.2.3 Operation

Wavelet Transform

Before filtering, the image mean is computed, subtracted off, and coded separately. The hierarchical 2D wavelet representation is arrived at by recursive application of separable horizontal and vertical filters. For each recursion, low pass and high pass filters are used to divide the image into equal sized subbands, which are critically subsampled. The result of the first level of filtering is shown in Figure 3.1. At each level, the low frequency quadrant is recursively filtered. The result of 3 levels of filtering is shown in Figure 3.2.

No attempt is made in this thesis to explore the optimization of filter design. The reader is referred to [35] [31] [29] [36]. Following [28] the filters in [35] are used here, examples of which are shown in table 3.1. Note that the symmetry of the filters allows simple reflection

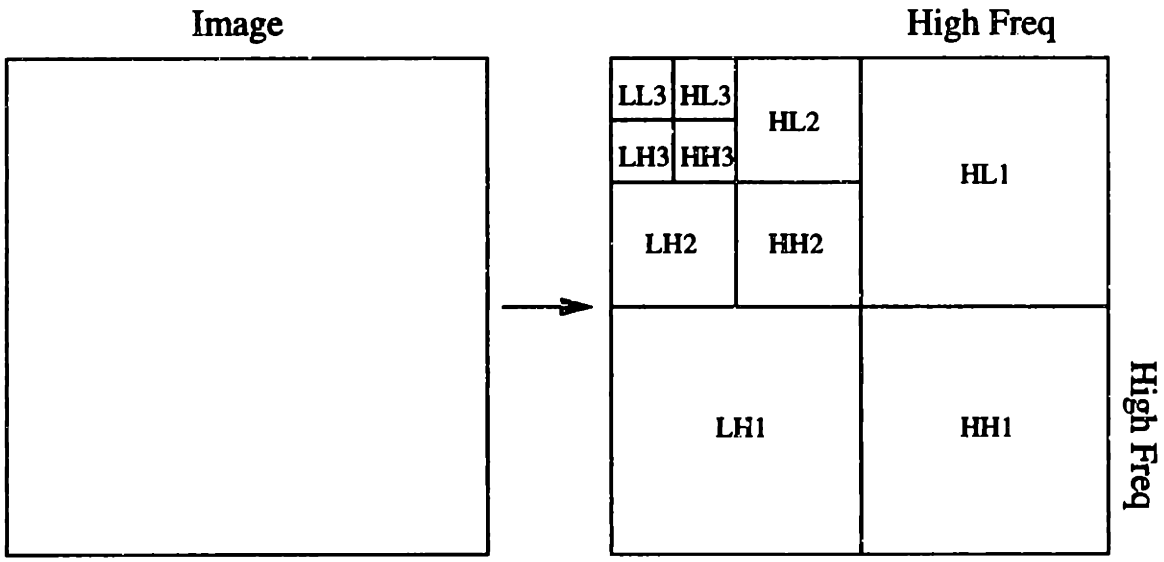


Figure 3.2: Result of 3 levels of filtering. At each level, the low-X, low-Y (like LL at level 3) frequency subband is recursively filtered and critically subsampled to obtain the next filter level. Labels used to refer to the subbands are shown inside.

to be used to treat image edges.

Bit Planes

The coefficients resulting from the hierarchical filtering are coded one bit plane at a time from MSB to the desired accuracy level. The coding of a bit plane is separated into two phases, termed subordinate and dominant in [28]. First, during the subordinate phase,

Length	Coefficients								
5			-0.05381	0.25	0.60762	0.25	-0.05381		
7		0.00525	-0.05178	0.25525	0.60355	0.25525	-0.05178	0.00525	
9	0.01995	-0.04271	-0.05224	0.29271	0.56458	0.29271	-0.05224	-0.04271	0.01995

Table 3.1
Example wavelet filters. Lowpass filters are shown. High-pass filters are derived by multiplying by -1^{tap} assuming the center tap is numbered 0.

all coefficients already found to have non-zero magnitudes (significant coefficients) from previous bit planes are refined by coding their bit in the current plane (termed subordinate bit). Note that the first plane is exceptional in that there are no previous significant coefficients, and the first subordinate phase is skipped. After a subordinate phase is complete, the significant coefficients are sorted in decreasing magnitude order. This ordering is used in succeeding subordinate phases to deliver LSBs. The relative ordering of subordinate and dominant phases within a bit plane, and the sorting of significants by magnitude, are two of the prioritizations introduced above, which are used to achieve embedded encoding.

During dominant phases, new significant coefficients are mapped with the aid of the zero-tree structures. That is, all coefficients are identified which have the leading 1 of their magnitude in the current bit plane. Zero-trees and the coding rules described below are used to aid compression by predicting the insignificance of high frequencies from low ones in the same spatial area, as discussed above. As new significants are identified, they are appended to the list for use in succeeding subordinate phases. (They cannot immediately be ordered by magnitude since, until further refinement, they all look the same.)

Zero-Trees and Dominant Phases

The purpose of zero-trees is to establish an ordered dependency graph between coefficients in the same spatial area, from low to high frequency. A small example is shown in Figure 3.3. The example is for 3 levels of filtering and a 16x16 image size. The bold outlines indicate frequency bands while the light ones show all the coefficients which belong to the example tree. For the 16x16 pixel example, four such trees would make up the whole image.

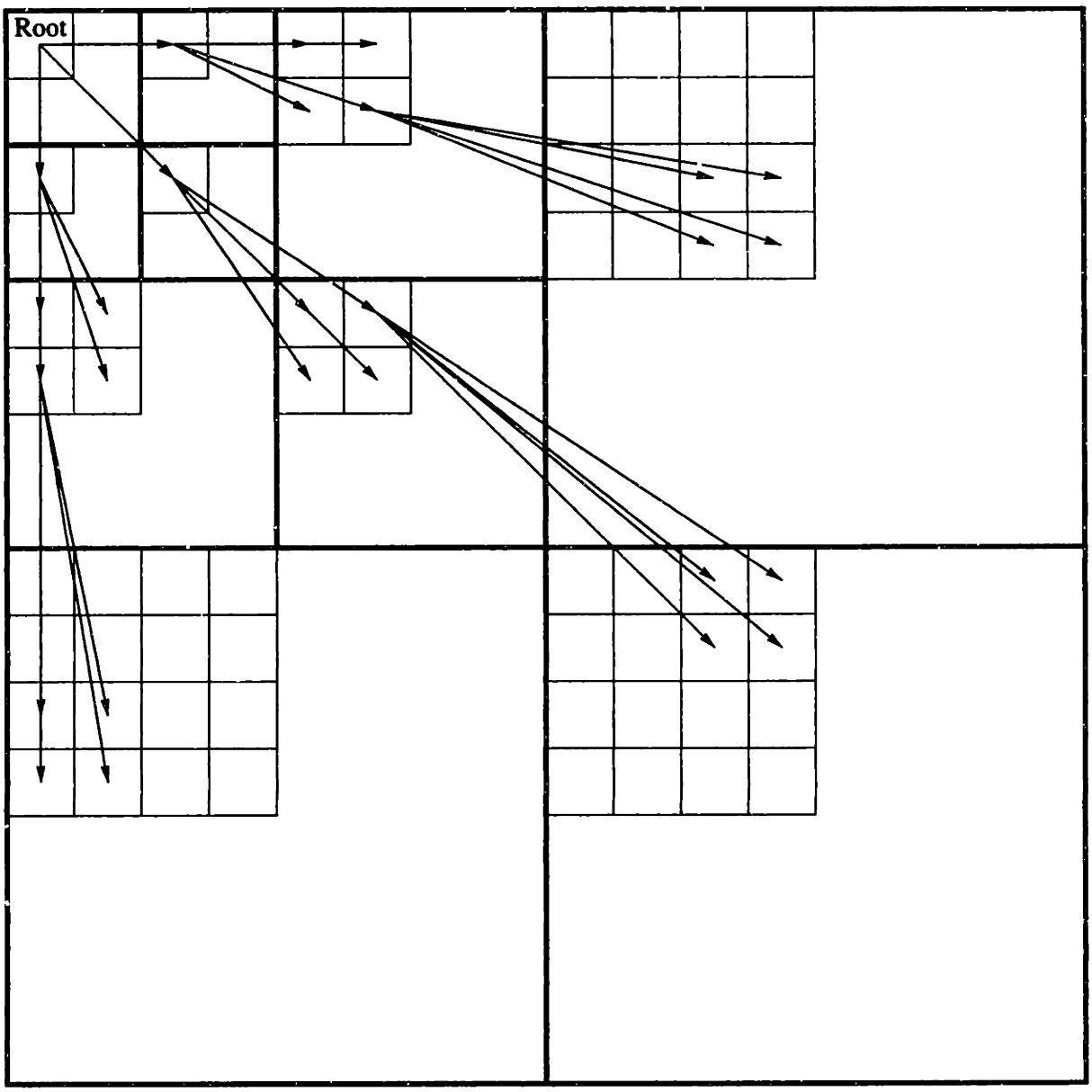


Figure 3.3: Example of a zero-tree for a 16x16 image with 3 levels of filtering. The tree is a dependency graph for coefficients belonging to the same image area, from lowest frequency (upper left) to highest (down and right). Bold outlines show subbands. Light outlines show all coefficients belonging to this tree. Not all dependencies are drawn. 4 such trees are needed to fully represent the 16x16 image.

The root of the tree is the low frequency coefficient at the third filter level. It's 3 children are the higher frequency components at the same level. From here, until the tree leaves, each node has 4 children which are the high frequency coefficients at the next lower filter level, corresponding to the same spatial area, and the same X-Y, high-low frequency orientation. For clarity, not all the dependencies are represented by arrows in Figure 3.3. The total number of coefficients belonging to the example tree is 64.

For purposes of dominant phases, some a priori scanning order is agreed on which ensures that low frequency coefficients are encoded before higher ones. For the 16x16 example, this would mean the 4 lowest frequency ones (in some order), followed by the 12 others at filter level 3, then the 48 at level 2, and finally the 192 at level 1. The following simple set of cases is used to encode the coefficients:

1. If a coefficient has been found significant in a previous bit plane, the decoder already knows its status and it can be completely ignored.

2. If a coefficient is being identified as a new significant this plane, a symbol is sent indicating the sign.

3. If a coefficient is still insignificant, and all its descendants are either also still insignificant or have been found significant during previous bit planes, the coefficient is encoded as the root of a zero-tree. This is expected to be the common case. The descendants are effectively already encoded and are marked as ignorable for the duration of this bit plane.

4. If a coefficient is still insignificant, but any descendant is a new significant this bit

plane, the coefficient is encoded with an isolated zero symbol. This is expected to be the uncommon case. In this case, the algorithm is forced to separately recurse on all the children of the affected coefficient, which may result in more isolated zero symbols until the new significant is reached and encoded.

5. As indicated above, if a coefficient is included in a zero-tree reported by an ancestor, the decoder already knows its status and it can be completely ignored.

Note that, in the general case, there are 4 symbols in the EZW dominant phase alphabet: positive/negative new significant, zero-tree root, and isolated zero. As an exception, since the highest frequency level 1 coefficients have no children, the root and isolated zero symbols are merged into a still zero symbol, which reduces the alphabet size to 3. Of course, subordinate phases use a 2 symbol alphabet, 1 and 0 to indicate LSBs.

These outgoing symbols are entropy coded by the adaptive arithmetic coder described below. Here, some optional conditioning possibilities are explained:

The goal of the wavelet, or any other, filtering is to remove all dependencies between the resulting coefficients and to concentrate the signal energy into as few coefficients as possible. If this were perfectly true, coefficients would be statistically independent of each other and could be optimally coded individually. This ideal condition is clearly not met. In addition to the zero-tree scheme, some conditioning of outgoing symbols by nearby states can also be used to take advantage of residual correlations. In [28] it was found marginally useful to condition outgoing EZW symbols with the significance of a coefficient's zero-tree parent and a nearest neighbor at the same filter level.

Finally, in order to further understanding of the statistical model implicitly incorporated into the algorithm by the use of zero-trees, it is worth briefly commenting on a choice made in the third of the coding rules listed above. Coefficients already found to be significant do not block the formation of zero-trees in succeeding bit planes. That is, after a parent has been an isolated zero because a child violates the statistical assumption of decaying distribution with frequency, it can later go back to being part of a zero-tree if no further significant energies are present in the area.

The point of zero-trees is to code significance maps cheaply by taking advantage of the decaying spectral energy distributions commonly found in real visual images. However, the algorithm does not ignore, obscure, or statistically swamp out higher frequency anomalies which violate this assumption, but which may represent visually important information such as edges. The isolated zero case is the mechanism which allows the anomalies to be recognized, with some bit cost per anomaly, as expected. An anomaly being what it is however, it is expected that much of the surrounding areas follow the common case. Therefore, after the anomaly has been recognized, its presence is masked for future bit planes so neighboring coefficients can be coded normally, without contamination from the statistical outlier.

Arithmetic Coding

The job of the entropy coder is to squeeze the last bits of compression out of the EZW symbol stream losslessly. All quantization is performed by choosing how many bit planes to encode. The goal is to approach as closely as possible the actual entropies of symbols, for both dominant and subordinate phases. This must be done in the face of non-stationary

statistics and some probabilities close to 0 or 1.

Probabilities close to 0 may result from infrequent occurrences of isolated zero symbols. Probabilities close to 1 may result from long runs of zero root symbols. This kind of pattern is alternately compressed with run-length coding, however, as discussed above, the choice of an adaptive arithmetic coder is more efficient and elegant.

Following the suggestion in [28], the arithmetic coder used is based on the one found in [37], which contains an excellent description of the mechanics of arithmetic coding, explanations of the advantages, and examples of working code. Arithmetic coding is as old as the hills [38], despite possibly being underappreciated and underutilized, and a mathematical development is not duplicated here.

The arithmetic coder uses estimates of symbol probabilities to make optimal partitions of its code space. In this case, local probability estimates are learned on-the-fly from symbols recently encoded by keeping a histogram of the symbols. The relative frequencies recorded in the histograms serve as probability estimates. Different histograms are kept for the respective alphabets used (dominant and subordinate) and conditioning values (parent and neighbor significance). The tallies are kept fresh (local) by periodically derating the contribution of older samples. When the total histogram count reaches a preset maximum, all current counts are divided by 2, and this discounted starting point is used to continue further encoding.

The choice of maximum histogram count involves a tradeoff, albeit a fairly shallow one. The maximum determines the smallest probability which can be represented and, therefore, how closely the entropy for very infrequent symbols can be reached. The small-

est probability is related to the reciprocal of the maximum count. On the other hand, the maximum also controls the learning rate. A smaller maximum adapts the probability estimates to more recent and local samples. As in [28], a maximum histogram count of 255 was found to be a good tradeoff between granularity and learning rate. All the histograms are reinitialized every bit plane to allow the acquisition of estimates without contamination from previous phases.

3.2.4 Image Compression Results

Table 3.2 shows compression vs. PSNR results from [28]. The data is taken from compressing images with working code, not simulations. The test images are the well known "Lena" and a lesser known "Barbara". These compression results were among the best available when this work began. The author is not aware of recent changes in the state-of-the-art for compression algorithms.

It should be noted that the results shown in Table 3.2 are for the exact algorithm reported in [28]. Specifically, they include the use of 6 filter levels and the 9 tap filter shown in Table 3.1. These results do not include the effects of any of the modifications described below. The effects of those modifications are stated using these original results as a reference point. Compression results for the final algorithm including all modifications are shown at the end of the chapter as a summary.

Compression Ratio	PSNR	
	Lena	Barbara
8	39.55	35.14
16	36.28	30.53
32	33.17	26.77
64	30.23	24.03
128	27.54	23.10
256	25.38	21.94
512	23.63	20.75

Table 3.2
Image compression results for unmodified algorithm. Results obtained with 6 filter levels and 9 tap wavelet filter.

3.3 Modifications to EZW Algorithm

3.3.1 Wavelet Transform

Filter Choice

No modifications were actually made to the wavelet filters used. However, while [28] uses the 9 tap filter shown in Table 3.1 to report compression results, the 5 tap filter is used throughout the examples in this thesis. There is nothing algorithmic or architectural preventing the use of longer filters here. The use of the shorter filter was motivated by the author's desire to cut corners in writing microcode to control the test chip (Appendix C). With the proper microcode, longer filters can be used to achieve slightly better compression performance at the expense of more power.

While no careful studies were done, it should be noted that compression performance saturates quickly with the filter length used. Results using a 3 tap filter not shown were observed to be quite bad. The change from 5 tap to 7 tap is marginal, and 7 to 9 negligible.

Meanwhile, the use of longer filters increases power cost in two ways. There are more operations, for example the 7 tap filter requires 2 more additions and multiplications. Further, some multiplications are more complex. As discussed in the next chapter, multiplications are not performed by dedicated circuitry, but pieced together with shifts and additions. Complex coefficients require more microcode operations to perform each multiply. These additional costs are not a bottleneck, however, since filtering represents a tiny fraction of the encoder's power consumption and computational throughput. Even without the use of motion compensation, filtering amounts to less than 10% of the power. With motion compensation, this would be reduced to noise.

Filter Levels

The number of filter levels used is reduced for reasons of power and area. The choice of 6 levels used in [28] is changed to 3 in the examples here. Power savings is obtained from two sources. A negligible gain comes from the omitted computations. This does not amount to much because of the exponentially decreasing amount of computation with each succeeding filter level.

A more substantial gain in power, and area, comes from the required arithmetic precision used to compute filter coefficients, as discussed below. The precision grows with the number of filter levels. Lower frequency coefficients require greater dynamic range to accurately represent large image areas. This translates into wider bit representations of data and arithmetic circuitry, at the rate of 1 bit per extra filter level.

On the other hand, filter levels greater than 3 have little impact on compression perfor-

mance, except for very high compression factors (a few hundred) for which visual quality becomes quite poor. Extra filter levels result in coefficients which represent exponentially smaller subband width and greater image area. For reasonable compression ratios and visual quality, the encoding reaches significant higher frequency coefficients at early filter levels. These completely dominate the resulting bit rate. Any gains achieved during early bit planes, where only very low frequency coefficients have any chance of being significant and where the extra filter levels help, are quickly swamped out. The extra filter levels only translate into compression for encodings which stop at those early bit planes, where only a few low frequency coefficients are transmitted.

Precision

No mention of arithmetic precision is made in [28], but it is assumed that floating point representations and operations are used. The circuits required to perform such operations are very expensive. Here, fixed point operations are used instead, with the lowest possible precision.

The minimum precision is determined by the number of filter levels, the highest desired visual quality, and rounding/truncation errors. Extra filter levels add 1 bit of required precision each, to contain larger magnitude low frequency coefficients, as discussed above. Higher visual quality requires extra precision in order to encode extra bit planes. Finally, some extra bits are required beyond the last encoded bit plane to keep the rounding and truncation errors of intermediate computations from reaching the last bit plane. The minimum precision for 3 filter levels and 8 maximum encoded bit planes (visually perfect reconstruction) is 12 bits.

3.3.2 Subordinate Phase

Prioritization by Magnitude

The sorting step after subordinate phases which orders significant coefficients according to descending magnitude is discarded. This is motivated from first principles. Sorting large numbers of items is an inherently costly operation and involves widespread communication patterns.

As discussed above, the omission of this, or other, encoding prioritization does not affect compression. It does spoil the embedded output stream property for small granularities (within bit planes). Limiting compression scalability choices to bit plane boundaries makes this effect irrelevant.

Subordinate Bit Encoding

As encodings progress through bit planes, subordinate bits quickly become decorrelated from each other, so the probabilities of 0s and 1s are always very close to $\frac{1}{2}$ and there is no way to avoid paying 1 output bit per subordinate bit. There is no point in the arithmetic coder learning the statistics of subordinate bits, so the subordinate alphabet histogram is discarded. Instead, all subordinate bits are forcibly encoded with probability $\frac{1}{2}$. Except for very high compression factors, there is no effect on compression performance.

3.3.3 Dominant Phase

Sign Bit Encoding

As with subordinate bits, sign bits are largely uncorrelated, except for very high compression ratios. It must be remembered that the image mean is subtracted off before filtering, so the resulting coefficients have a globally zero mean distribution. In addition, the sign bits do not have much local correlation in real images. Therefore, sign bits are also encoded with constant $\frac{1}{2}$ probabilities, and the arithmetic coder histograms associated with them are discarded. The special treatment of sign bits within the dominant alphabet is facilitated by the breakup of the 4 and 3 symbol alphabets into a succession of multiple smaller alphabets. This is coincidental, due to changes to the arithmetic coding itself, described below.

Conditioning

As in [28], it was found that conditioning EZW dominant phase symbols with parent and neighbor significance has marginal effect on compression. For substantial tests on sample images, the contribution of the conditioning was never observed to be greater than 5%. More precisely, that was the maximum loss of compression due to discarding the conditioning and using one histogram instead of 4. This net loss is not solely attributed to the conditioning since there is an incidental counteracting effect to the reduced number of histograms. With all dominant symbols being counted in the same histogram, the probability estimates developed in it form more quickly and track changing statistics faster. That is, the learning rate of the coder is enhanced by not diluting the samples among 4 separate

histograms and forcing the coder to learn 4 separate sets of probabilities.

Extended 3 Symbol Regions

In [28] the EZW coder switches from the 4 to the 3 symbol alphabet at the first level of filter coefficients (the highest frequency). This is because the lowest level coefficients have no children in the zero-trees, so there is no need to distinguish between zero root and isolated zero for insignificant coefficients. This switch is also applicable in a different region of operation.

The maximum magnitude of coefficients grows with each filter level, 1 bit per level as discussed above. For example, it is known that at the highest bit plane, where level 3 coefficients might be significant, level 2 and 1 coefficients must all be 0. Similarly, the second bit plane might contain significant level 3 and 2 coefficients, but not level 1. Therefore, there is no point during these early bit planes to even consider these high frequency coefficients (which also happen to be the most numerous). For these early phases, the zero-trees are effectively truncated at the higher levels and the newly exposed leaves are coded with the 3 symbol alphabet.

This modification is not motivated by a tradeoff between compression and computational complexity. There is, in fact, a benefit to compression, albeit a negligible one except for very high compression ratios. The real benefit is reduced computation in the form of large numbers of ignored coefficients. For example, out of the 8 total possible bit planes available for coding with the chosen 12 bit precision, the top bit plane need only consider $\frac{1}{16}$ of the coefficients and the second plane $\frac{1}{4}$. (Note that this benefit also applies to the num-

ber of coefficients that must be examined during the immediately succeeding subordinate phases).

3.3.4 Arithmetic Coder

Initilization of Histograms

In [28] the histograms which implement the adaptivity of the arithmetic coder are reinitialized each bit plane. The assumption is that symbols in different bit planes are not well correlated to each other, and the coder is better off learning new statistics from scratch.

In fact, it was observed that it is slightly beneficial to reinitialize more frequently than that. Symbols at different filter levels within the same bit plane do not share the same distributions, so reinitialization is performed at each level. The compression benefit of this modification was never observed to be greater than 3%, so this is not being touted as an algorithmic improvement. However, the modification is free, and this discussion serves to explain the implementation difference which is evident in Appendix A.

Probability Approximation

The arithmetic coder acts on an already compressed data stream. However, adaptive arithmetic coding is inherently a serialized set of operations. Preceding symbol encodings are used to estimate probabilities of later ones. It is very difficult to parallelize the arithmetic coder, especially in light of the fact that individual symbol encodings take an indeterminate number of computations and generate an indeterminate number of output bits [37]. Note that counting symbol frequencies in some average haphazard manner does not suffice.

The decoder must have exact a priori knowledge of ordering in order to remain properly synchronized and decode the data stream.

The difficulties of parallelizing the arithmetic coder mean that architectural and circuit techniques which achieve energy efficiency by using slower circuits (adiabatic, voltage scaling, etc ...) are ineffective. Therefore, despite the partially compressed data stream the coder acts on, there is strong motivation to keep the complexity of the coder as small as possible. Power is directly benefited by any reductions in number of operations or their complexity, and indirectly by reductions in required throughput and critical path delays, which alleviate the speed burden on circuitry.

The principal computational cost in the arithmetic coder stems from the probability estimates and their conversion into appropriate fractions of the code space. A symbol's probability estimate is the ratio of its current histogram count to the sum of the counts in the alphabet. This estimate is used to partition the coder's space in the fashion described in detail in [37]. Both of these steps involve division, which is a very expensive arithmetic operation, as well as multiplication by non-constant values (in contrast to the wavelet filtering which involves multiplication by constant filter coefficients which can be trivially converted to small numbers of shifts and additions at compile time).

This cost is eliminated by making a tradeoff against a modest compression degradation. In the arithmetic coder used here, all probability estimates are approximated as fractions which can be represented by inverse powers of 2. This converts all divisions and multiplications into the much cheaper operations of shifting and addition (required to make the approximations). Note that while the shifts are by non-constant values and are

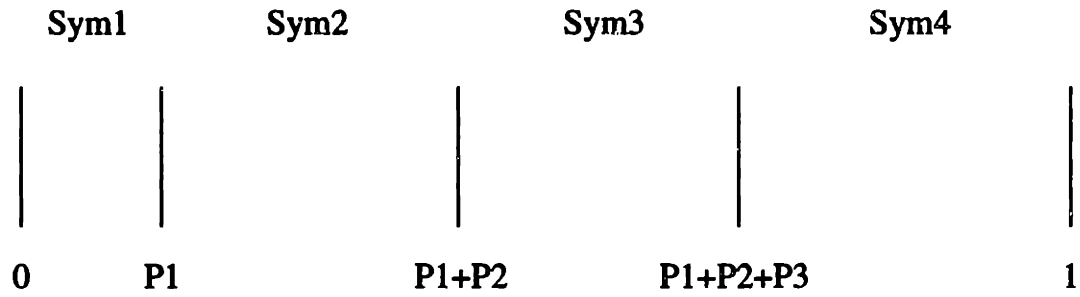


Figure 3.4: The arithmetic coder segments the encoding space optimally by allotting each symbol in the alphabet a segment proportional to its probability. The symbol probabilities are the ratio of symbol counts to total histogram count. The values shown (probability sums) must be approximated as inverse powers of 2.

therefore not quite as cheap as the compile time shifts used in the filtering, the number of shifting choices is quite small due to the limited dynamic range of the probabilities being estimated. For example, the largest shift necessary is 7, so a logarithmic shifter needs only 3 levels of 2-to-1 muxes (shift by 4, 2, and 1) to implement the entire operation.

While this strategy is conceptually simple, the implementation is not straightforward. There are a number of impediments to accurately approximating relevant fractional values as inverse powers of 2. Figure 3.4 shows the values used by the coder which are subject to approximation for the 4 symbol alphabet case. The arithmetic coder segments the encoding space optimally by allotting each symbol in the alphabet a segment proportional to its probability. The symbol probabilities are the ratio of symbol counts to total histogram count. The fractional values which divide the space, (P1, P1+P2, P1+P2+P3) must be approximated as inverse powers of 2.

In addition, to ensure functional correctness, no probability can be set to 0, even if a symbol has not yet been encountered in a large sample set. That is, no matter how small

the likelihood of a symbol, a finite width segment of the code space must be allocated to enable coding an unpredictable occurrence.

These two requirements make acceptable approximations difficult. For example, if the 4 histogram counts were C1=100. C2=80, C3=75, C4=2, the smallest space that could be allocated to symbol 4 is $\frac{1}{2}$, equivalent to a 50% probability of occurrence. This kind of gross inequity could be rectified by reassigning symbols to a different ordering for purposes of segments in the encoding space. After each new histogram entry, the symbols could be sorted in increasing probability order, with a mapping recording the assignment of symbol number to segment number. The decoder would have all the same histogram values from previous decoded symbols and could duplicate the mapping applied to the next one. However, the quality of the resulting approximations would still be poor. Consider the simple example of all equal probabilities. The closest match for segment boundaries would be 0 - $\frac{1}{8}$ - $\frac{1}{4}$ - $\frac{1}{2}$ - 1. The disparity between segments 1 and 4 is a factor of 4.

The solution to these problems is to break down the EZW alphabets into hierarchies of 2 symbol alphabets. Instead of transmitting a single symbol from several choices, multiple consecutive symbols are sent with only 2 choices each. Figures 3.5 and 3.6 show how this is done for the 4 and 3 symbol cases, respectively. There is no compression penalty for doing this, since entropies of independent events add. In addition, after each transmitted symbol, each 2 symbol sub-alphabet is sorted by probability, as proposed above for the 4 symbol case. Of course, with only 2 symbols per sort, this reduces to a very cheap toggling operation. With these changes, the actual approximation becomes straightforward. For example, for any 2 symbol sub-alphabet, a probability of $\frac{1}{7}$ is approximated as $\frac{1}{8}$, or $\frac{1}{160}$ as

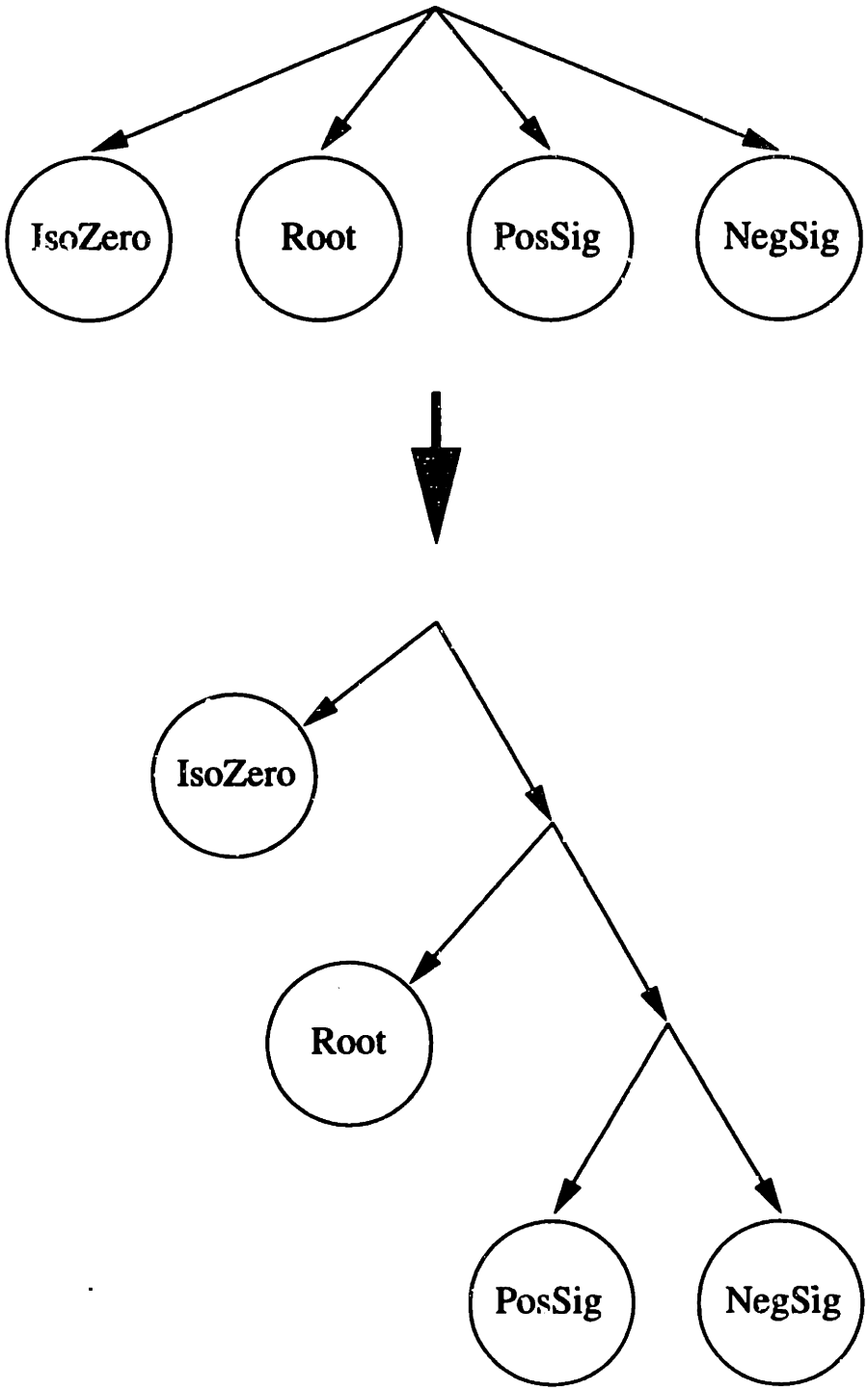


Figure 3.5: Hierarchical alphabet break down for 4 symbol EZW alphabet.

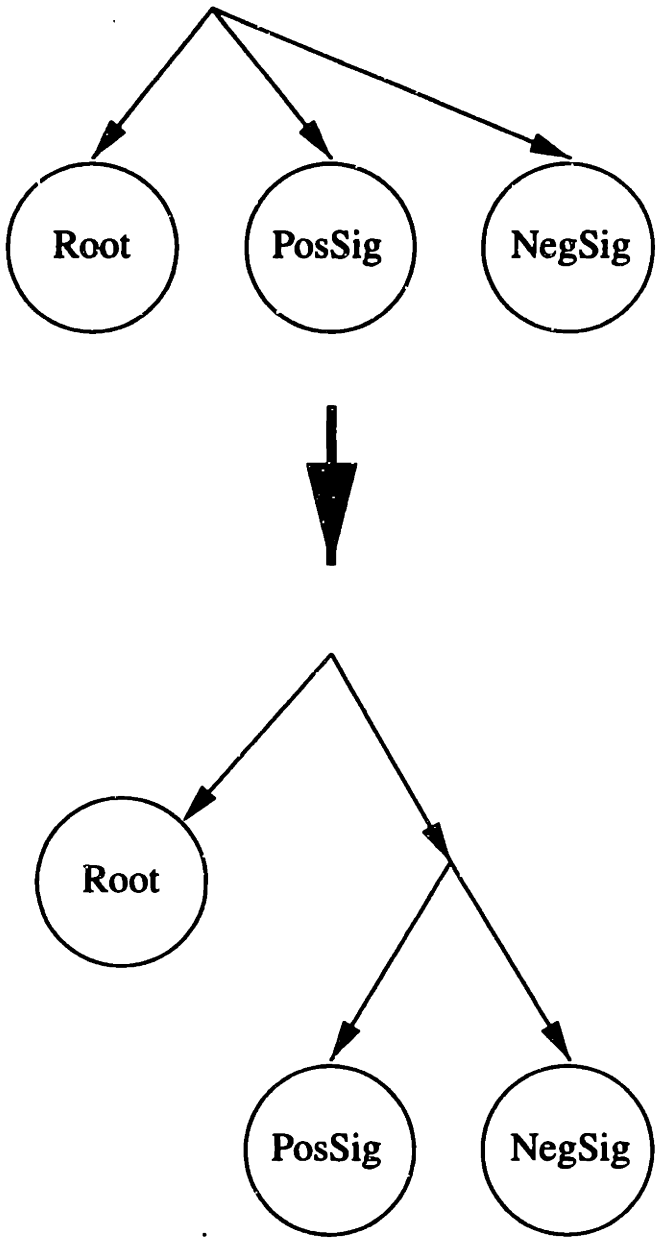


Figure 3.6: Hierarchical alphabet break down for 3 symbol EZW alphabet.

$\frac{1}{128}$.

Note that separate histograms are kept for the 2 symbol alphabets. On the other hand, the convenient isolation of the sign decision for significants makes it easy to discard that particular histogram, as discussed above. In total, 2 histograms of 2 entries each are kept. One for identifying isolated zeroes, the second for distinguishing zero-tree roots from significants. The first histogram is unused in 3 EZW symbol regions.

The penalty in lost compression due to these modifications is quite small. In extensive trials on varied test images, the loss in compression was never observed to be greater than 2%. It may seem surprising that an approximation as gross as substituting nearest powers of 2 for values should have so little effect. However, it should be kept in mind that the probabilities being approximated are only gross estimates themselves. Estimates of non-stationary statistics which are tracked with finite speed and sample sets (despite possibly small symbol probabilities in some cases). In many cases, approximations both smaller or larger than the original probability estimates will actually yield better results than the use of the unadulterated values.

Appendix A contains code for 2 versions of the probability approximation. The unused version (commented out) includes a slightly better rounding scheme. The simpler version, which is implemented on the test chip, losses an additional 1.5% in compression. There is no reason not to use the better rounding scheme. The additional operations involved are quite cheap. The simpler version used was another corner cutting opportunity for the author.

3.3.5 Symbol Scanning Order

The modifications described in this section involve the reordering of symbols encoded in both dominant and subordinate phases. These ordering changes are motivated by architectural issues, and it is necessary to take a small detour to preview what those issues are. The architectural choices stated below are motivated and described in detail in the next chapter.

The overriding architectural factor is the SIMD arrangement of processing elements (PEs). The implication is that operations which keep the array of PEs working together in lock step are efficient. Operations which serialize computation by making exceptions are inefficient.

Mapping Data onto SIMD Array

With an eye on efficiency and parallelism, every effort is made to equalize the work load and memory requirements of all PEs. Toward that end, each PE in the array is assigned a 4x4 pixel area of an image, and all the associated computation. Figure 3.7 shows how an 8x8 part of an image is mapped onto 4 neighboring PEs. Each pixel is numbered with the index of the PE it is assigned to.

Figure 3.8 shows how the wavelet coefficients corresponding to the same image area and resulting from 3 levels of filtering are mapped onto the 4 PEs. The bold outlines separate coefficients of different filter levels. Note that the 48 coefficients of level 1 and the 12 of level 2 map onto the respective PEs uniformly. The 4 coefficients of level 3, however, do not. (Note that this is an artifact of the 4x4 pixel per PE granularity. A different choice,

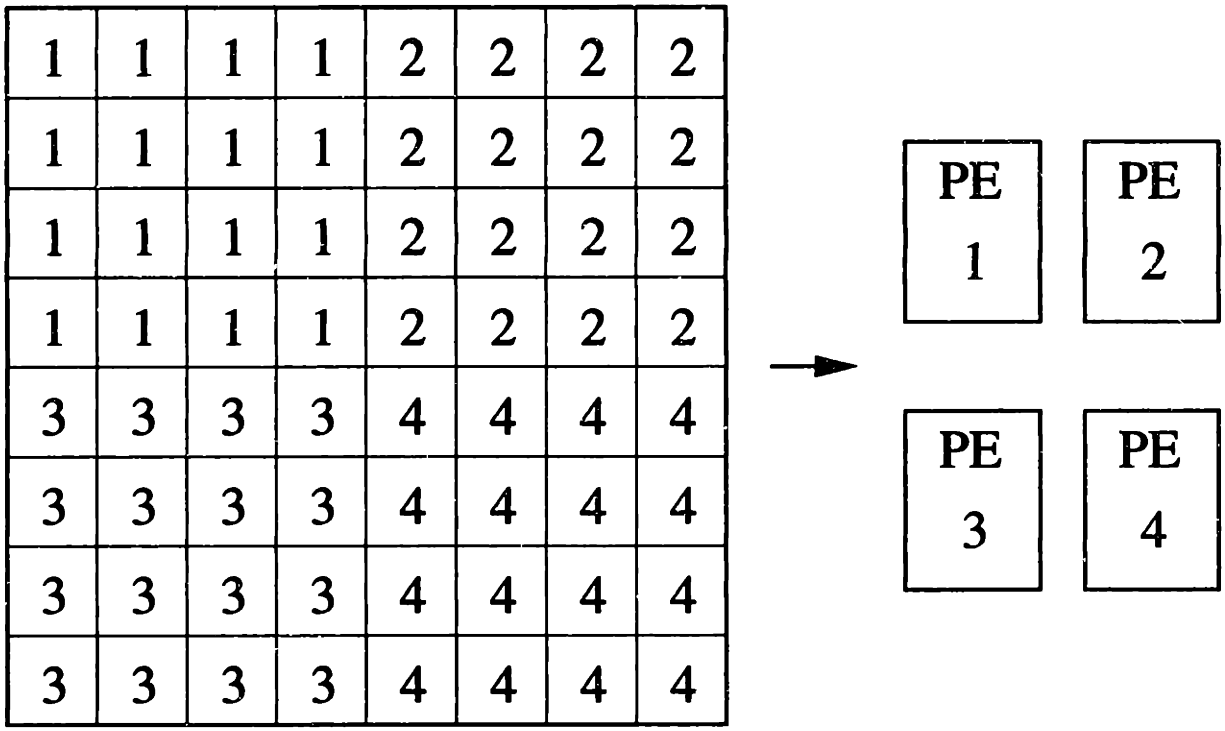


Figure 3.7: Mapping of image pixels onto SIMD array. Pixels are numbered with the index of the PE they are assigned to.

such as 8x8 pixels/PE, resulting from a different process technology, as discussed in the introductory chapter, would not experience this uneveness.)

Scanning Order of Level 3 Coefficients

While no explicit scanning order is specified in [28] other than that imposed by zero-trees, the most obvious would be a raster scan of one subband at a time. For example, after encoding all the level 3 low-X, low-Y frequency (LL) coefficients, the ordering might continue through all the level 3 HL, LH, and HH coefficients, and then on to filter levels 2 and 1.

Due to the level 3 subband mapping onto PEs, as shown in Figure 3.8, the encoding order chosen involves an interleaving of subbands HL, LH, and HH in level 3. That is,

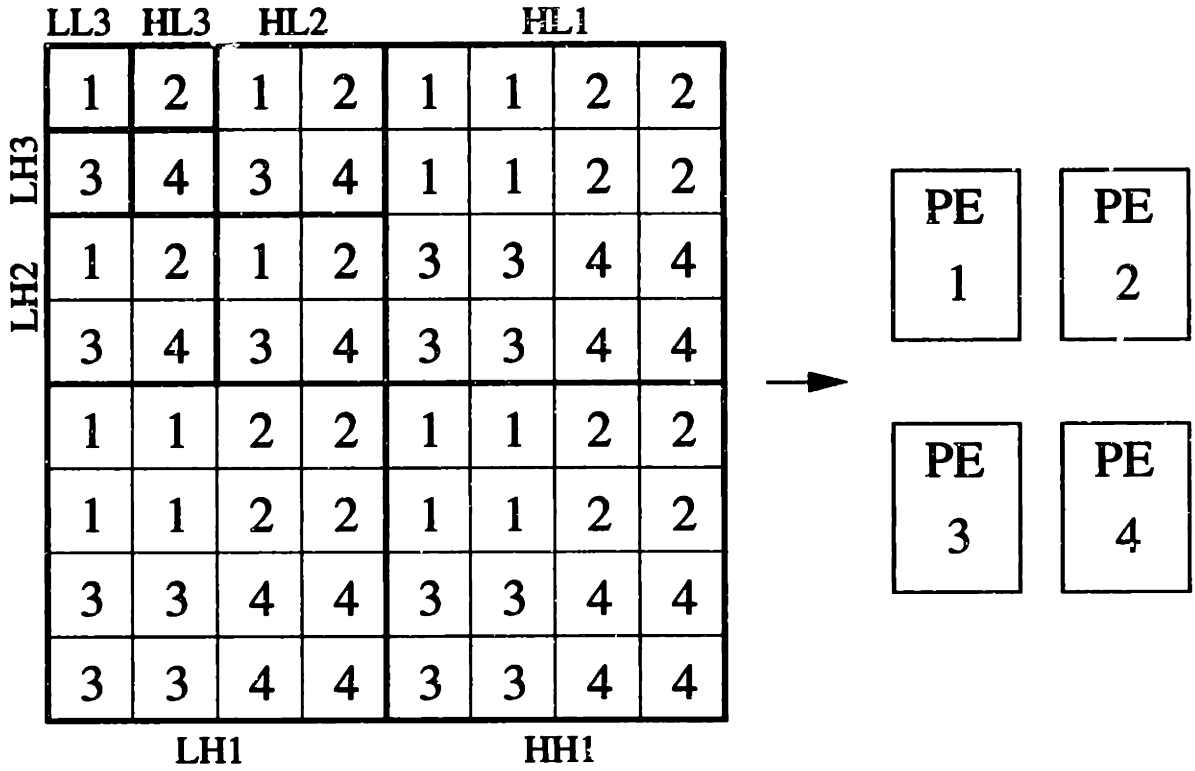


Figure 3.8: Mapping of subband coefficients onto SIMD array. Coefficients are numbered with the index of the PE they are assigned to. Bold outlines separate coefficients at different filter levels.

instead of having the SIMD array execute 3 separate sets of instructions which picked off the respective coefficients of those subbands continuously, before moving on to the next subband, only 1 set of instructions is used to encode all 3 bands in 1 pass. Thus, starting a raster scan of the PEs, say from the upper left corner, and scanning PEs across one row at a time, the encoding encounters an entire row of HL coefficients, followed by a row of interleaved LH and HH coefficients, and then another row of HLs, etc ... The raster scanning across PEs is converted to a serialized stream of symbols for use in the arithmetic coder by parallel to serial conversion circuitry described in the next chapter.

The price for this interleaving is a loss of the embedded property at small granularities.

Again, however, macroscopic scalability is not disturbed. Note that compression is technically not identical in this case, due to a second order effect in the arithmetic coder. If the coder was presented symbols in separated bands, it would learn and use their statistics in order. The interleaving causes the coder to learn and use joint statistics for the conglomerate. As it happens, this is suboptimal since coefficients have more correlation within their own subband. However, this effect was found to be completely negligible.

Note that the level 3 LL subband is kept separate from the other 3. This is not because of any violation of zero-tree ordering that might occur. Notice that a left to right and up to down scanning of PEs always places an LL coefficient before its own 3 higher frequency children (though not necessarily before the higher frequency coefficients in neighboring zero-trees). The separation is maintained because the statistics of the LL subband is different enough from the other 3 to impact the performance of the arithmetic coder.

Scanning Order of Level 1 Coefficients

The mapping of level 1 coefficients onto the PE array causes a lesser disturbance to the scanning order compared to level 3. While all PEs contain coefficients in each of the 3 level 1 subbands, so a PE scan can be filled with coefficients of the same band, it is evident that a raster scanning across PEs does not result in a raster scanning across coefficients. In fact, 4 passes are required to complete each of the 3 level 1 subbands, with each pass skipping every other column and every other row. Again, there is a negligible second order effect on the learning pattern of the arithmetic coder, this time due to the less spatially localized (as opposed to subband localized) delivery of symbols.

Interleaving of Dominant and Subordinate Phases

The last, and most dramatic, ordering change involves the interleaving of symbols from adjacent dominant and subordinate phases. Because the interleaving is between symbols of a dominant phase of one bit plane and the subordinate phase of the next lower bit plane, a subtlety of the embedding/scaling scheme is elaborated which has so far been omitted for clarity.

So far, subordinate and dominant phases have been grouped in bit plane pairs, with the exception of the absent top plane subordinate phase. Further, modifications affecting the embedded output property at various granularities have constrained scaling choices to bit plane boundaries. However, the modifications described so far only spoil the embedded property within a single dominant phase or a single subordinate phase. Nothing prevents choosing scaling points at bit plane midpoints, between a subordinate phase and its matching dominant phase. This is in fact the choice made here. From the standpoint of embedding/scaling, this frees up the possibility of mixing up the order of symbols across back to back dominant and subordinate phase pairs.

This is done by only scanning across PEs in the SIMD array once for each group or subband of coefficients instead of twice. For each coefficient, any necessary dominant symbol is encoded, followed immediately by a subordinate bit, if necessary. Note that there are 4 possibilities of what categories of symbols are encoded for each coefficient: none - the coefficient is still insignificant and part of a zero-tree, just dominant - the coefficient is insignificant but is a root or isolated zero, just subordinate - the coefficient was found significant in a preceding plane, both - the coefficient is a new significant. Also note that,

at this point, the symbols being referred to are not the final ones represented in the output bit stream, but instructions to the arithmetic coder which are translated to the output.

This ordering modification is motivated by the power used in the parallel to serial conversion circuitry, described in detail in the next chapter. The serialization involves a lot of clocking which would be duplicated if dominant and subordinate symbols are sent in separate PE scanning passes. In addition, some power would also be wasted on the data wires, because both dominant and subordinate symbols alone do not pack nicely into compact representations. For example, separately there would be 5 dominant symbols: root, isolated zero, positive significant, negative significant, none (merely a NOP instruction to the arithmetic coder for this slot). The 5 choices would pack inefficiently onto 3 data wires. Meanwhile, the subordinate list is an awkward 3 long: 1, 0, none. Sent together, the two sets can be compressed to 9 choices on 4 wires: none, root, isolated zero, 2 kinds of subordinate only, 4 kinds of new significant (2 signs, 2 subordinate bits).

This time, there is no second order effect on the performance of the arithmetic coder due to learning pattern. The relative order of the dominant symbols is undisturbed, while the interlaced subordinate bits do not contaminate the histograms which record dominant symbol statistics.

3.3.6 Miscellaneous

Rounding vs. Truncation

With the fully embedded output of [28], the encoding may be suddenly truncated, say when an exact target bit rate has been reached. With controlled video quality scaling deci-

sions made either at the encoder or fed back from a base station, the precision with which wavelet coefficients are quantized is known on a per image (or frame) basis. This presents the opportunity to round coefficients before quantization, instead of just truncating them. While the gains here are small and the optimum is shallow, compression of test images has been used to pick $\frac{1}{4}$ the value of the LSB as a rounding value added to coefficients before quantization.

3.4 Extension to Time - Frame Differencing

3.4.1 Introduction

The algorithm presented so far is capable of compressing single images. While it could be used to compress one frame of a video sequence at a time, this would not take advantage of the substantial correlation between frames.

The first choice made toward extending the single frame algorithm to the time dimension is to rule out any form of 3D transform. That is, just as a 2D wavelet transform is used to concentrate signal energy into a few coefficients, a 3D transform, separable or otherwise, could be constructed to do the same for groups of consecutive frames. This possibility is ruled out due to the prohibitive memory burden imposed by the need to collect several frames at a time prior to the 3D filtering. Instead, the strategy is to remove correlations between 2 frames at a time.

The following section discusses the possibility of using motion compensation. Here, the simpler option of frame differencing without motion compensation is discussed. This is the option implemented on the test chip.

3.4.2 Computational Cost

Simple frame differencing is computationally trivial. Before quantizing and encoding a frame, a representation equivalent to the previous frame received by the decoder is subtracted off. Only the difference is encoded. Note that, as convenience dictates, the subtraction may be performed in either the pixel or coefficient domain, since the wavelet filtering is a linear transformation. The choice here is to subtract the wavelet representation since the quantized and transmitted versions of frames in this representation are readily available to the encoder. The only requirement is one frame buffer to store the encoded coefficients transmitted in the previous frame. Performing the differencing in the pixel domain would require the encoder to perform an extra inverse transform of the quantized coefficients.

3.4.3 Compression

Frame differencing is also remarkably effective at removing correlations between frames. Any static object or background in a visual scene is subtracted off. Most of the residual energy is due to motion, in which case there will certainly be content around the object edges, and some in the interior. However, some energy from the interior of moving objects can still be removed without the use of motion compensation (eg. smooth object textures). Of course, other residual energy may be due to lighting intensity changes or just noise.

Note that the author cannot confidently account for the EZW algorithm's success at compressing frame differences as well as frames. An underlying statistical assumption of the EZW algorithm is decaying signal energy with spatial frequency in natural images. It

seems obvious that this is less so for frame differences, where a lot of DC energy may be removed, but high frequency energy around the edges of moving objects remains. It is left for future mathematical and algorithmic work to determine whether the assumption is still valid or if there is some other property which explains the algorithm's success. Nevertheless, the algorithm's compression of frame differences is spectacular, as demonstrated below.

3.4.4 Synchronization and Error Recovery

As described so far, the algorithm might begin encoding at some first frame after a system reset and continue with frame differencing indefinitely. There is no problem with regard to integrating numerical errors due to limited precision and truncation. This is prevented by the feedback of subtracting previous frames just as the decoder is receiving them, after the effects of rounding, truncation, etc ...

However, as described in the previous chapter, there is a problem with high bit error rates expected in wireless communication and synchronization loss. The decoder needs to know exactly what symbols have preceded any point of an encoding to know how to proceed. This is true because of the state of the arithmetic coding [37], the state of the symbol histograms, the place in the PE scanning order, etc ... Clearly, an error burst which the degree of error correction coding used cannot recover from will result in a complete loss of data afterward.

To limit the losses due to uncorrectable bit errors, the encoding process is periodically reset. At the reset points, frame differencing is abandoned for one frame. The first frame

of each group is encoded in its entirety. In addition, both the arithmetic coder state and its histograms are reinitialized.

These actions enable the decoder to acquire fresh data uncontaminated by errors in the previous group, so long as the decoder can tell where the new data stream begins. To assist with this, the encoder chip provides an extra signal to the sensor's transmitter. The signal marks the reset point between the last frame of one group and the first frame of the new group. The transmitter can use this event to trigger the flushing of any buffering used, including the possible use of a last padded packet. In addition, it is assumed that the next packet header is marked with a flag informing the decoder at the base station that a synchronization reset point has been reached. The decoder can then also flush all of its state and begin decoding from scratch at the first bit of the flagged data packet.

For example, for the test algorithm and chip the choice was arbitrarily made to reset the encoding every 16 frames. With a frame rate of 30/sec this results in a maximum loss of about $\frac{1}{2}$ sec of data. There is a tradeoff here. More frequent synchronization points reduce the maximum data loss. However, each synchronization point involves the coding of an entire first frame, which is more expensive than the coding of succeeding frame differences. Therefore, there is an associated loss of compression. An indication of the relative cost of the lead and differential frames can be seen in the example bit rates of Table 3.3. (The bit rates are taken from a test example given below.) This tradeoff is subject to purely application specific considerations. Note, however, that there is no reason this choice cannot be made on-the-fly with possible feedback from the base station, just as the scaling of quantization and bit rate.

Frame #	Output Bits
1	6574
2	1631
3	2450
4	1656
5	1232
6	1752
7	1299
8	1534
9	1717
10	1575
11	1443
12	1542
13	401
14	543
15	1408
16	1277

Table 3.3
Number of output bits for frames from a 16 long synchronization group shows the relative cost of the lead frame and differential frames. These output bit rates are for a test example whose PSNR performance and visual quality are given later.

3.4.5 Miscellaneous Modifications

Two changes are made to the frame compression algorithm specifically to accommodate the encoding of frame differences.

Empty Bit Plane Flags

The removal of a lot of low frequency signal energy by the use of frame differencing results in a high probability that several top bit planes will be completely insignificant. These are bit planes for which, even in a whole image, only low frequency coefficients have much chance of being significant. The empty bit planes would consist only of zero-tree root

symbols for each of the LL subband coefficients at the top filter level.

This is not a large cost because the arithmetic coder would quickly learn the high probability of the root symbol and encode a succession of them fairly cheaply. Nevertheless, it is a trivial matter to short circuit this cost by preceding bit planes with a 1 bit flag indicating whether the plane is empty and can be omitted. In fact, the algorithm generates empty flags until the first significant bit plane, and one non-empty flag for that first plane. After the first significant plane, flags are not included for the remainder of the frame, and all succeeding planes are encoded normally.

The most important benefit of this addition is in the case of empty frame differences resulting from no motion in the video stream. The previous chapter addressed techniques which can be employed to optimize system performance during prolonged periods of no motion. These techniques are not implemented because they relate to other system components and involve application specific tradeoffs.

However, the empty bit plane flags do address the case of isolated or small numbers of empty frame differences during short periods of no motion. Note that the cost of encoding an empty frame has been reduced to 1 bit per plane (with a maximum of 8 bits). Of course, the same mechanism marginally improves the compression of mostly empty frames.

Image Mean

In the same vein as above, the image mean is not coded for empty frame differences. The image mean is always coded for the lead frame of each synchronization group, and for succeeding frame differences which contain a significant plane.

3.5 Video Compression Results

Results for the entire video compression algorithm are shown below. These results are for the real algorithm implemented on the test chip. C code which emulates the encoder, and a matching software decoder, is given in Appendix A. (The C code generates output 8 times bigger because data is saved in ASCII. This is a debugging aid leftover). The algorithm includes all the modifications described above including the use of the 5 tap filter, 3 subband levels, 12 bit integer precision, and 16 frames per synchronization group.

Figures 3.9 and 3.10 show compression results for different scaling choices. The two columns of images are the lead and end frame from the same synchronization group for corresponding scaling choices. The video sequence is a head and shoulders sequence with little motion. Table 3.4 gives PSNR results for the compressed images shown. Figures 3.11 and 3.12 and Table 3.5 show similar results for an outdoors scene with a great deal of motion, including camera pan, rotation, and zoom. The relative results for the two sequences underline the need for motion compensation.

Compression Ratio	PSNR	
	Frame 1	Frame 16
7	44.326	44.626
12.8	43.264	43.594
23.8	40.900	41.359
49	37.502	37.885
115	33.494	33.773
300	29.831	29.851
967	26.120	26.169

Table 3.4
PSNR results for head and shoulders sequence. Frames 1 and 16 are the first and last frames of one synchronization group.

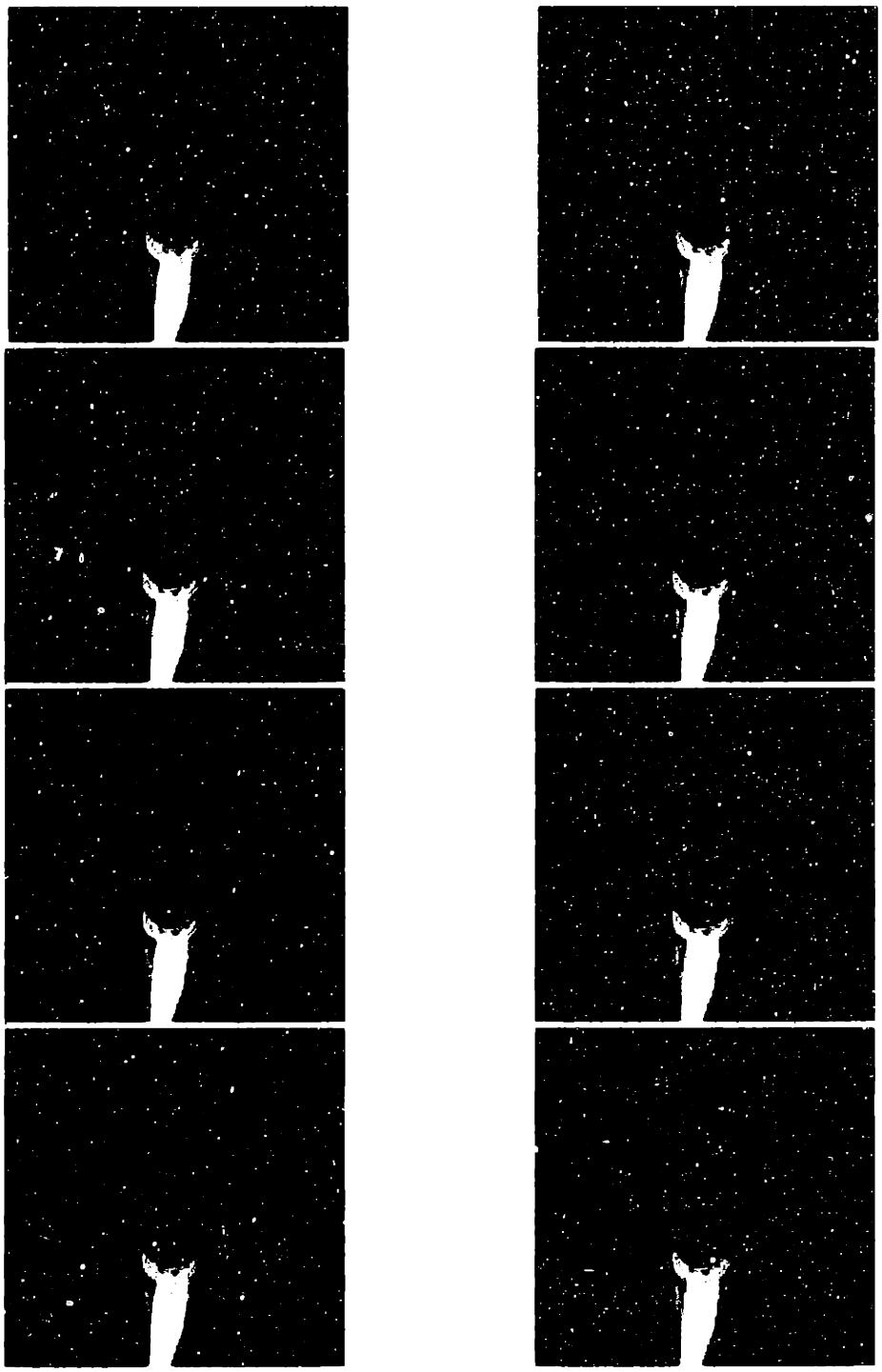


Figure 3.9: Compressed frames of head and shoulders sequence. Left and right columns are first and last frames of one synchronization group. From top to bottom, the rows are: original images, compression factor 7, compression factor 12.8, and 23.8 .

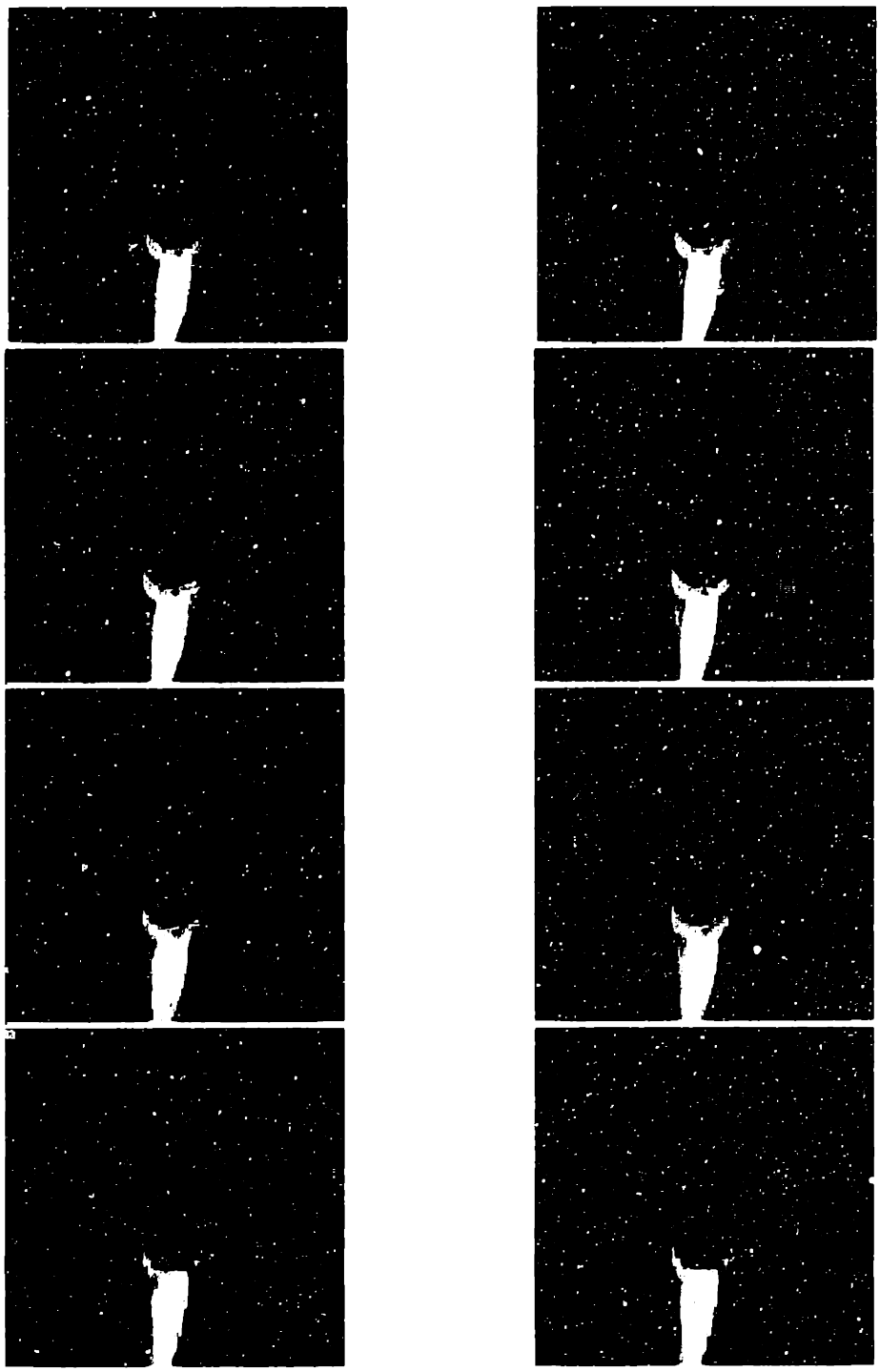


Figure 3.10: More compressed frames of head and shoulders sequence. Compression factors: 49, 115, 300, and 967 .

Figure 3.11: Compressed frames of outdoors sequence. From top to bottom: original images, compression factor 2.6, 6.3, 12.2 .

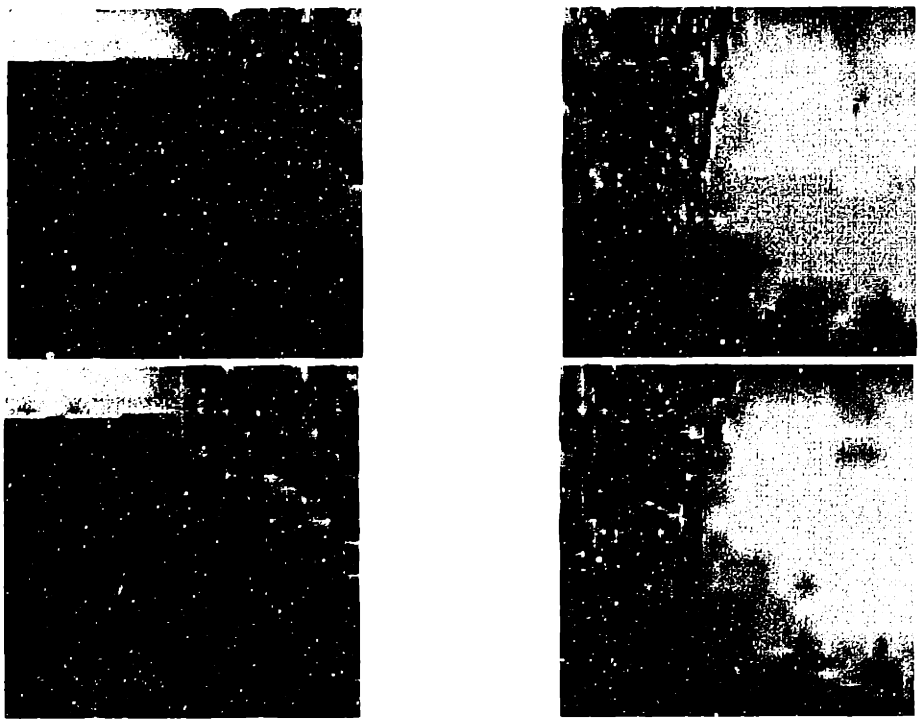


Figure 3.12: More compressed frames of outdoors sequence. Compression factors 26.7 and 67 .

Compression Ratio	PSNR	
	Frame 1	Frame 16
2.6	40.496	40.176
6.3	35.156	35.194
12.2	31.412	31.613
26.7	28.059	28.067
67	24.574	25.045

Table 3.5
PSNR results for outdoors sequence.

3.6 Motion Compensation

3.6.1 Introduction

Motion compensation before frame differencing is expected to aid compression performance, especially for motion which can be well modeled as translational. This is ideally the case for objects moving parallel to the field of view, and for camera pan. The benefits are the removal of more signal energy of moving objects, both high frequency around edges and all scales in the object interior.

Use of motion compensation is not included in the test chip. Further, motion compensation itself is a substantial subject of study. No attempt is made in this thesis to optimize this technique for low power or to describe it in detail. The reader is referred to [20] [39].

3.6.2 Memory Requirements

Inclusion of motion compensation on the test chip was ruled out due to memory area costs already exacerbated by the coarse implementation process technology. Without motion compensation, the test chip requires 3 separate frame buffers: previous frame - used for frame differencing, current frame - used as workspace for intermediate results and buffer for coefficients being sent to the arithmetic coder, next frame - used to load pixels of the next frame in parallel with current frame encoding. The inclusion of motion compensation would raise this to 5: next frame - same function, current frame - current frame and workspace, previous frame coefficients - buffer holding outgoing coefficients for the arithmetic coder, previous frame quantized pixels - used as a reference for motion compensation, previous frame shifted - used as workspace to try varying motion vectors.

The increase in memory capacity requires some explanation. First, note that with motion compensation, the current frame workspace can no longer double as the outgoing coefficient buffer. With simple frame differencing, the algorithm has almost no work to do. In fact, more than 90% of the computational throughput in the test chip is spent idle. This throughput would normally be reserved for the considerable computational needs of motion compensation. Therefore, in the test implementation, the computation of the current frame and its processing through the arithmetic coder can be serialized. After filtering, frame differencing, etc ... the resulting coefficients remain in the current frame buffer, from which they are shipped to the arithmetic coder over the duration of the remainder of the frame. With motion compensation, the throughput of the entire frame is required to process the motion compensation step, so the coding must be parallelized from a separate buffer. Note that, because of energy efficiency issues as discussed above, it would not be acceptable to speed up the arithmetic coder so it could process a frame in much less than a frame period.

Second, motion compensation requires that the previous frame information be kept in both wavelet coefficient and pixel format. The wavelet domain is required for parallel coding, as just discussed. The pixel domain is required because motion compensation is fundamentally a pixel domain operation. Note therefore, that the inverse transform optimized out with simple frame differencing is no longer avoidable. Motion compensation must also be with respect to the previous frame as the decoder has access to it. The quantized coefficients must be inverse transformed to obtain this representation. However, this extra cost is swamped out by the motion compensation itself.

Finally, a separate copy of the previous frame must be kept, as it is shifted around to identify optimal motion vectors, because shifting operations are irreversible (with just one buffer, data can be lost at the image edges after a shift by a trial motion vector).

3.6.3 Future Work

However, motion compensation cannot be completely ignored. As discussed above in this chapter and in the previous chapter, expected trends in computational and communication costs make techniques which trade local resources for communication bandwidth desirable. Motion compensation is an important such technique for video compression.

Therefore, high level architectural decisions are made in the next chapter, and architectural hooks are included in the design, so that motion compensation can easily be added, as process technology and memory area allow. In addition, two avenues for future work are suggested below which may play an important part in optimizing the use of motion compensation for low power.

Temporal Motion Vector Prediction

Motion vector estimation is computationally very expensive. For each frame, image data is partitioned into small blocks of pixels. For each such block, an attempt is made to identify an optimal motion vector which predicts the visual content of the block from another block known from the previous frame. Optimality is measured by the amount of residual signal energy remaining from the difference between the translated blocks. The resulting coding cost is for the motion vector itself and a much reduced block difference from the previous frame (at least if there was significant motion).

Recent studies [21] [40] [41] [42] suggest that a significant way to reduce the cost of identifying motion vectors is to predict them from previous frames. The studies indicate that motion vectors have good correlation from frame to frame. This may seem obvious by simply considering the continuity of the motion of real objects in a scene. However, there are complications from local minima, motion vectors identified by the residual energy metric which are fleeting coincidences but not real motion vectors of objects. These kinds of minima are not expected to have good correlation across frames.

Nevertheless, the studies indicate that motion vectors from previous frames can be good starting hints for new searches. In this case, computational cost could be reduced to identifying motion vector corrections in small windows around the values of the starting points. Presumably, a compression benefit is also obtainable from coding vector corrections instead of full vectors.

Note that, again, there is a tradeoff involving the frequency of synchronization resets since, at reset points, full motion vectors must be sent to the decoder. On the other hand, there is no reason why the encoder cannot still reduce its computational costs by taking hints from vectors used before reset points.

Global Motion Vector Sets

Typically, full blown motion compensation attempts to assign individual motion vectors to each small image block before comparison to the previous frame. This is fundamentally incompatible with the SIMD architectural model which serves the rest of the algorithmic needs so well. Efficiency is maximized for a SIMD architecture if all elements are doing

the same thing. However, independent block motion vectors require pervasive serialized exceptions to be made as each block vector is corrected from frame to frame.

Instead, a study is proposed to ascertain the utility of using a small number of global motion vectors. For each frame, a few possible motion vectors would be identified. Each pixel block would have the choice of using one of the vectors or opt out entirely and default to the (0, 0) vector (simple frame differencing). This would greatly reduce computational costs since all elements would shift and compare all blocks in the same way, for each global vector.

In addition, there could also be compression benefits from using a limited set of motion vectors. Instead of coding vectors for each block, the global vectors would be encoded once, along with a block to vector choice map. Further, the coding of the vector choice maps may benefit from the same scale-hierarchical structure used in the rest of the algorithm. That is, one might expect that if the real motion vector of an object was identified, there would be good correlation of the vector choices of nearby image blocks which cover that object.

There is good reason to expect a payoff from such a scheme. A few global motion vectors cover a great deal of the benefit one might expect from motion compensation for natural video sequences. Clearly, camera pan is covered supremely well, in addition to the motion of small numbers of large objects in a scene. Not covered would be a lot of haphazard motion of many real objects, or incidental benefits due to block matches which do not represent real motion. On the other hand, a few globally optimal motion vectors may be much more likely to identify real motion, as opposed to accidental local minima.

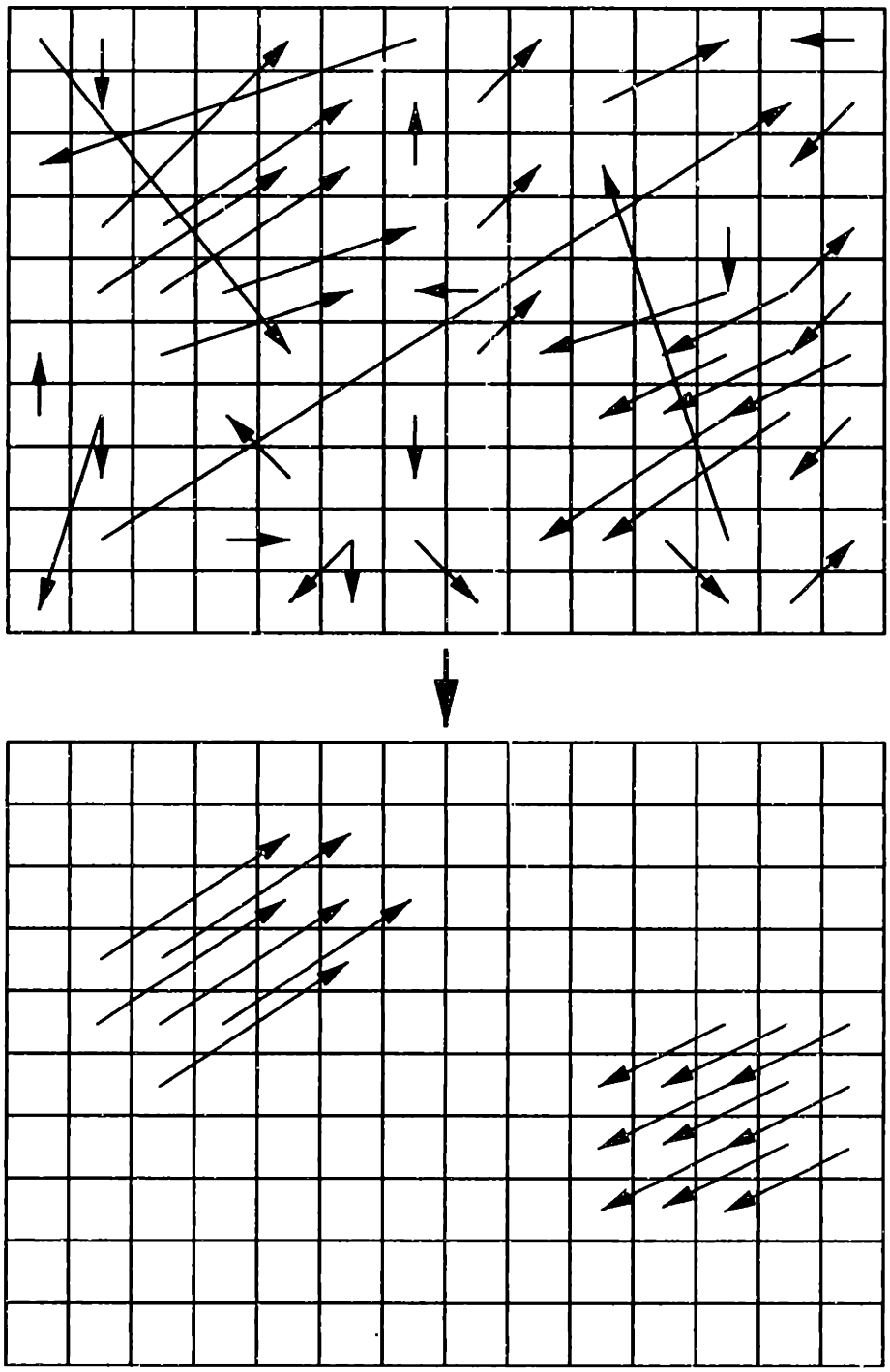


Figure 3.13: SIMDification of motion vectors with 2 global vector choices. Upper figure is a hodge-podge of independent block vectors. Below is the result of using just 2 globally optimal vectors. (For clarity, vectors are drawn with block sized magnitudes. Actual motion vectors have at least pixel resolution.

This would enhance the predictability of these vectors in succeeding frames, as suggested above. Figure 3.13 shows an example of how motion vectors might be constrained by the use of global vectors, in a manner compatible with a SIMD architecture. Of course, a minimum benefit threshold, or some other heuristic, would be required to weed out stale vectors and trigger a full search for new motion.

3.6.4 Relative Power Estimate

In order to get a qualitative feel for the extra power cost of using motion compensation, a rough, back of the envelope estimation is used to compare the cycle counts for motion compensation and the rest of the algorithm, whose cost is included in the test chip's measured dissipation. Note that the total power with motion compensation would not be the chip power multiplied by the factor estimated here. The chip power includes components which are unaffected by the use of motion compensation, such as: new pixel loading, EZW symbol parallel to serial conversion (actual power is reduced by the extra compression of the motion compensation!), arithmetic coding (also reduced!), SIMD controller and peripheral circuitry, etc ...

A qualitative estimate for the instruction count of motion compensation is based on the structure of the SIMD array, described in the next chapter, and the following list of assumptions and parameters:

- A scheme like the global motion vector sets suggested above is used, with 2 such vectors for this example.
- Motion vector magnitudes are limited to 32x32 pixels.

- Frame to frame motion vector prediction is used. The cost estimated here is that of searching for motion vector corrections in a 7x7 pixel window (chosen as an arbitrary example) centered around each previous frame vector. Figure 3.14 shows an efficient pattern of image shifts which can be used to cover the search window. The arrows indicate the shifts performed by the entire SIMD array. At each vector, all blocks are compared to the current frame and residual energy is measured to find the globally optimal motion vector. Only 7 shifts by the magnitude of the entire vector need to be performed. Most of the shifts are incremental single row or column hops. Further improvement could be obtained by executing U incremental turns. This could be at the cost of some accuracy loss by dropping pixels at image edges, or adding gutted dummy PEs at the SIMD array edges with enough memory to cache the shifting spillover.

The estimate for this computation is 25K SIMD instructions per frame. This includes the cost of the inverse transform, getting residual energy measures out of the array to the controller, and all shifting and arithmetic operations for both motion estimation and compensation. This is compared to about 3.3K instructions per frame for the rest of the algorithm. The factor is less than 8.

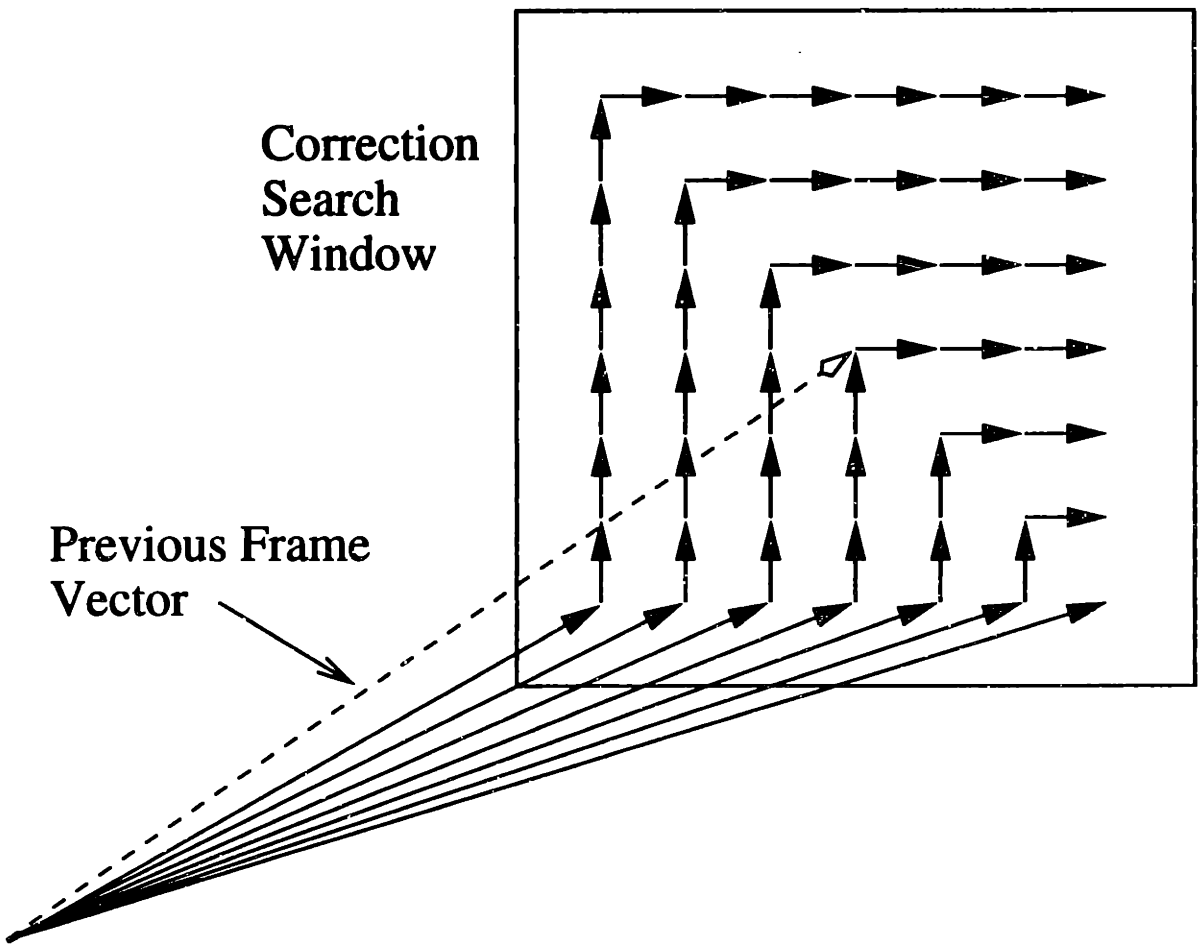


Figure 3.14: Solid arrows show combinations of long and short image shifts which cover a 7x7 pixel window. The window is used to search for the optimal motion vector around the previous frame's (dashed arrow).

Chapter 4

Architecture

4.1 Introduction

If not using adiabatic charging, the central mantra of low power architecture design is equation 4.1.

$$Power = \sum_{all\ circuits} \frac{1}{2} C V^2 f \alpha \tag{4.1}$$

where C is the capacitance of any circuit element, V is operating voltage, f is clock frequency, and α is the probability that a circuit node will change value in any given clock cycle. V^2 becomes $V\Delta v$ for low swing signals.

The most important architectural contribution to low power operation is the minimization of physical capacitance, ΣC. Not only does this linearly reduce power, it increases circuit speed and typically reduces area. In contrast, techniques which lower power by reducing α sometimes use more area [43] (raising Cs, but reducing the amount of capacitance actually switched per cycle).

Finally, reducing the V^2 term (voltage scaling) involves a tradeoff between power and speed. The quadratic reduction in power due to lower V is offset by, ideally, a linear reduction in the speed of circuits. To maintain some desired throughput, more circuitry, or C, must be used in parallel, which costs area.

Note that parallel operation of slower circuitry is subject to the algorithmic parallelism available. The constraints are not only functional correctness, but the circuit overhead imposed by parallelism, which detracts from the maximum quadratic gains of voltage scaling. Overhead may take the form of distributing or merging input and output data, suboptimal functional granularities, etc ... [43].

In addition, it is important to note the effects of device threshold voltages, the principal physical limit to voltage scaling. As operating voltage approaches device thresholds, the speed of circuitry falls off more rapidly than the linear ideal, again reducing the gains of voltage scaling. Device modeling and behavior is extensively studied in the literature, and the reader is referred to [44] [45] [46], however, a very simplistic relationship is given in equation 4.2. In the test chip, voltage scaling limits are arbitrarily set by the implementation technology's device thresholds. More generally, threshold voltage reduction is itself subject to limits imposed by subthreshold leakage, which increases exponentially with lower thresholds. Loss due to leakage must be kept small for equation 4.1 to remain valid.

$$Circuit\ Speed \propto \frac{(V - V_{Th})^2}{V} \tag{4.2}$$

The literature derives detailed power minima in the face of known thresholds. This is the crossover point between reduced V^2 and the larger than quadratic offsetting increase

in Cs associated with parallelism required to keep up with the slower circuits. The important realization is that, as operating voltage approaches device thresholds, voltage scaling bottoms out. If more parallelism is algorithmically available, other means must be found to exploit it. In this vein, unusual circuit examples are given in the next chapter which reduce Cs at the expense of slower circuits!

4.1.1 SIMD

Fortunately, the compression algorithm, especially after the modifications described in the previous chapter, falls into a category known in the algorithm community as "embarrassingly parallel". In addition, the copious parallelism exhibits a lot of data parallelism, with very localized communication patterns, if data is correctly organized. A fine granularity SIMD array is a natural architectural choice. No overhead is paid for the independent instruction execution, interprocessor synchronization or handshake, addressable communication networks, etc ..., of more complex parallel architectures, such as MIMD.

As already shown in Figure 3.7, the compression algorithm is mapped onto a SIMD array of processing elements, such that small blocks of neighboring image pixels are assigned to one PE. Each PE contains both the logic and memory required for its image block. One effect of this partitioning is to split up the total memory of the architecture into very small subarrays. This fine granularity benefits the bandwidth out of the total memory and the energy cost per load or store. However, the small subarray sizes also impose some circuit design challenges, especially with regard to area, since bit line peripheral circuitry is amortized over few memory elements. These issues are discussed in the next chapter.

4.1.2 SIMD vs. Dataflow

While the fine granularity SIMD array is one obvious choice, a dataflow-like architecture, which is in widespread use in modern dedicated DSP systems, must be ruled out. Here, dataflow means that the graph of operations which make up the algorithm are flattened out in space, with data flowing through a network of circuits which implement all the required computations. Typically, heavy use of pipelining raises throughput, which is desirable to enable voltage scaling, just as many parallel PEs in the SIMD array do.

The attraction of a dataflow type architecture is the inherent, hard wired instruction sequence which is embodied in the nodes and interconnections of the circuit network. This saves the considerable cost of distributing instructions, which is required in one form or another for any architecture that reuses circuits to unfold operations in time rather than space. Although, it should be noted that unfolding complex graphs in space can be difficult and result in inefficiencies due to merging data streams widely separated in area.

The use of any dataflow architecture is ruled out for the simple reason that it is incompatible with the algorithmic demands of motion compensation. Motion compensation is a global optimization, which means that all elements of an image frame must be present and involved in the operation at the same time. Further, motion compensation is the first algorithmic step. The filtering, quantization, and coding of any image element cannot start until the entire motion compensation operation is complete. This is incompatible with a dataflow structure which pipelines the data in sequence through a one way network of circuits.

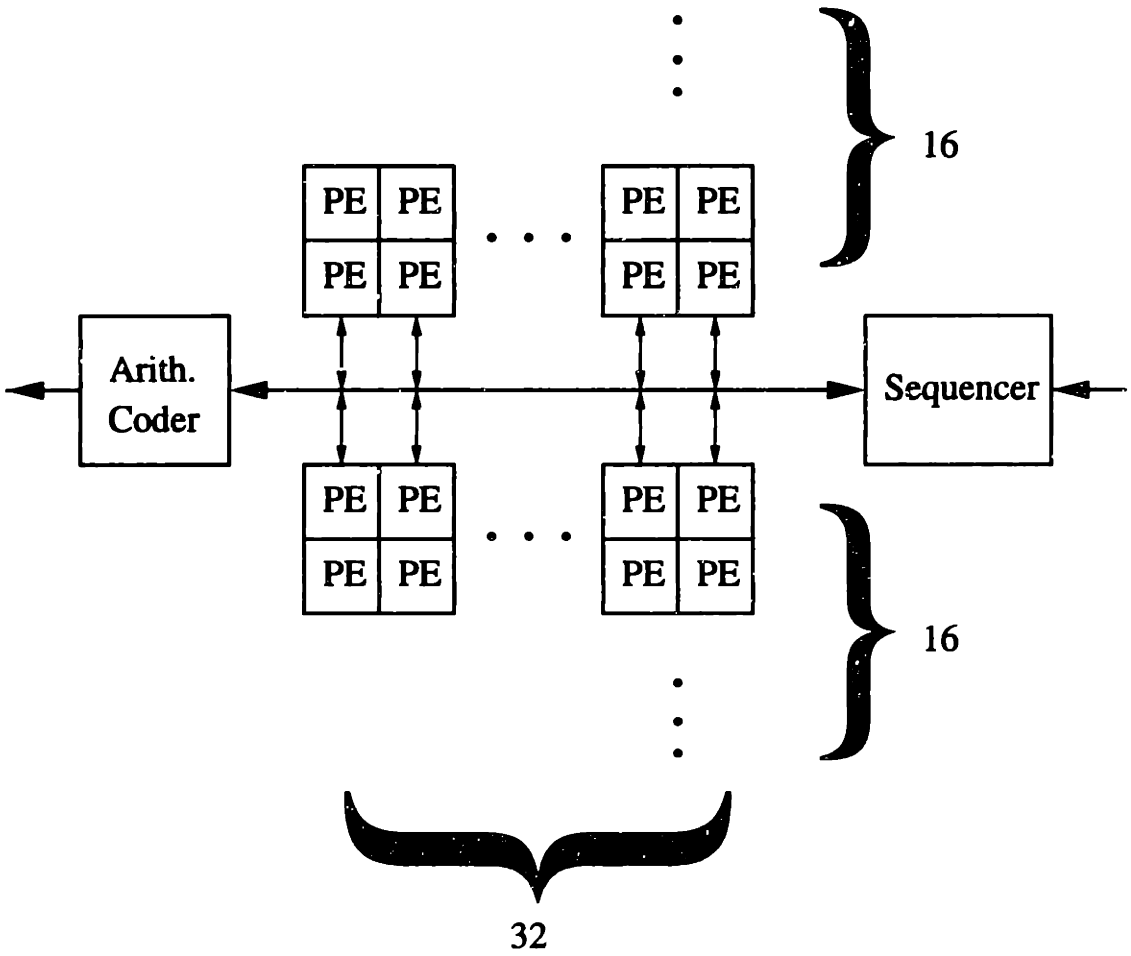


Figure 4.1: Block diagram of architecture shows 32x32 PE array and peripheral circuitry, including arithmetic coder and sequencer.

4.2 Array Implementation

4.2.1 Overview

Figure 4.1 shows a block diagram of the entire architecture. The SIMD array consists of a square arrangement of 32x32 PEs. A granularity of 4x4 pixels per PE results in the 128x128 resolution of the test chip. The SIMD array is responsible for the computation, communication, and memory requirements for most of the algorithm, including motion compensa-

tion, frame differencing, wavelet filtering, quantization, and zero-tree coding. Peripheral functions are handled by the separate arithmetic coder and the controller/sequencer.

The controller decodes a microcode instruction stream for the PE array, generates PE timing signals, controls loading of incoming pixels and unloading of outgoing symbols to the arithmetic coder, and keeps track of global state used to sequence through the microcode.

Communication is served by 4 separate and independent networks distributed throughout the PE array. These are described in detail below. They are: a one hop north-east-west-south (NEWS) network, a serial to parallel converting pixel load network, a parallel to serial converting unload network for coded EZW symbols going to the arithmetic coder, and a one bit global wired or used to return control information to the sequencer.

The SIMD array is split into two equal sections of 32x16 PEs. The trench separating the two halves carries the backbones for the pixel, EZW, and global OR networks, as well as control and handshake signals between the controller and arithmetic coder. The gap separating the two array sections is a bit awkward from a layout/floor-planning standpoint since PE boundaries are designed to allow simple tiling with no additional glue. The alternative would be to leave the array unbroken and place the communication network backbones along one side of the array. However, the special layout treatment at the trench is warranted by the power efficiency in the communication networks, as described below.

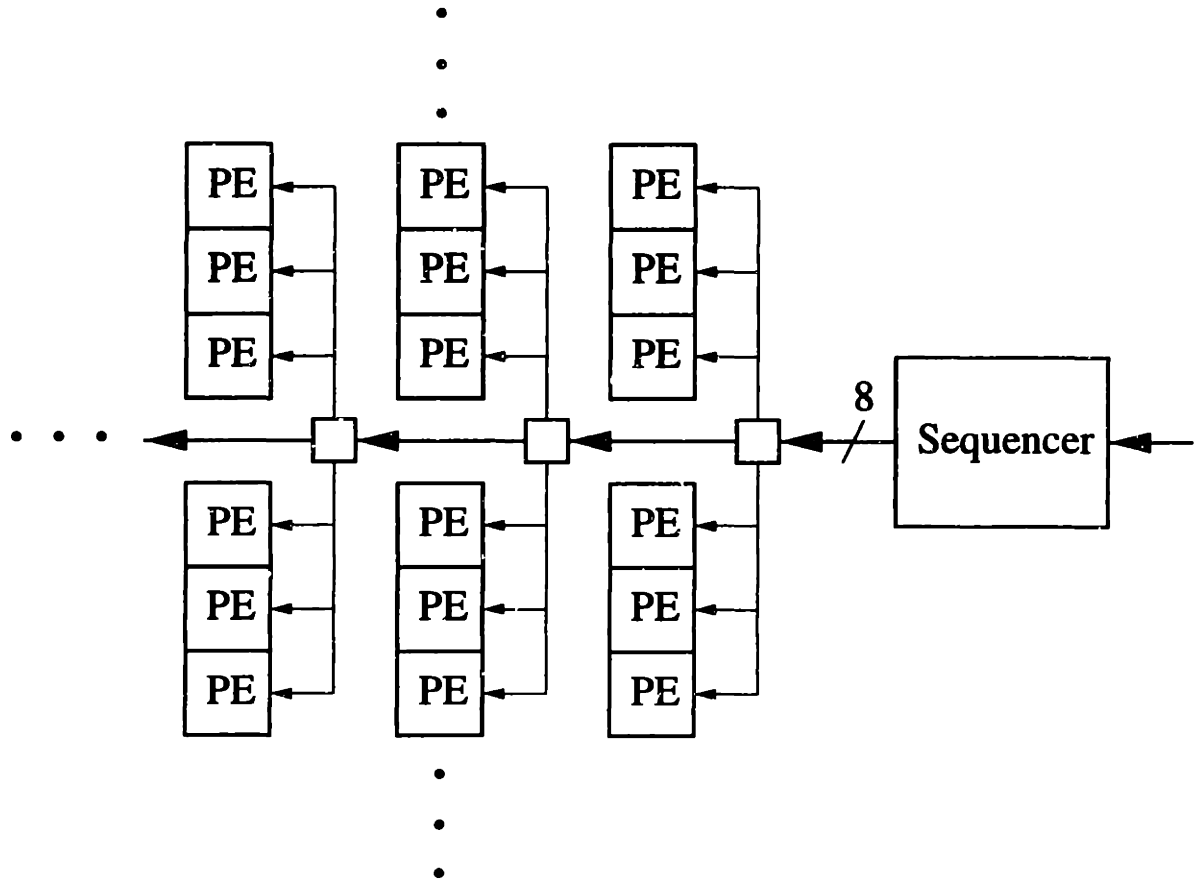


Figure 4.2: Block diagram of pixel load network.

4.2.2 Pixel Load Network

The pixel load network, shown in Figure 4.2, must distribute serially incoming 8 bit pixels to the PE array. A separate communication network is required because the pixels arrive continuously over an entire frame time (to allow the imager and ADC to operate as slowly and energy efficiently as possible).

The data stream is parallelized in 2 dimensions for delivery to the array. The 1st dimension conversion is performed in the trench backbone. One row of pixels at a time is collected in the backbone and delivered to the appropriate row of PEs. The PEs hold the

incoming pixels in special registers until all PE rows have received data, which completes the 2nd dimension of parallel conversion. Once complete, a single SIMD instruction loads the holding registers into the appropriate PE memory locations. Note that the loading instructions are not entered in the microcode, the controller generates them automatically and transparently to the microcode. This is desirable because the loads are required at precise times, while the microcode may be executing branch instructions that make synchronization uncertain. Circuitry which enables trivial implementation of the transparent pixel loading is described in the next chapter.

Note that each load of pixels into the array memories delivers 1 of 16 pixels of the image block each PE is assigned. 16 such passes per frame are required to load an entire image. Clearly, the scanning order of incoming pixels is not a simple rastering. The combination of 16 passes and uniform SIMD loading results in the scanning order shown in Figure 4.3, which must be adhered to by the imager.

Serial to parallel conversion is performed with busses and gated clocks rather than shift registers, in order to save power. As each row is delivered from the backbone to a PE row, the controller provides the specific PE row one clock edge to load the holding registers. Clock power is saved because clocks are not delivered all the time and not delivered to all rows along with row enables. While a shifting scheme would limit the distance traveled by any datum to just the right number of PE rows, and a wire in a simple bus implementation might span all the rows, some data power is also saved with the use of busses. First, the considerable extra capacitance of shift register internal nodes is avoided. Second, busses can be segmented to limit the needless wire lengths exposed to switching data. Circuits

1	2	3	4	1	2	3	4	1	2	3	4
5	6	7	8	5	6	7	8	5	6	7	8
9	10	11	12	9	10	11	12	9	10	11	12
13	14	15	16	13	14	15	16	13	14	15	16
1	2	3	4	1	2	3	4	1	2	3	4
5	6	7	8	5	6	7	8	5	6	7	8
9	10	11	12	9	10	11	12	9	10	11	12
13	14	15	16	13	14	15	16	13	14	15	16
1	2	3	4	1	2	3	4	1	2	3	4
5	6	7	8	5	6	7	8	5	6	7	8
9	10	11	12	9	10	11	12	9	10	11	12
13	14	15	16	13	14	15	16	13	14	15	16

Figure 4.3: 16 pass pixel scanning order to deliver 1 image frame to SIMD array. Pixels are labeled with the pass they are delivered in. All pixels in pass 1 are raster scanned, followed by pass 2, etc ...

which perform this segmenting are described in the next chapter. However, the placement of the network backbones in the center trench is a coarse example of this concept. Busses to the upper and lower halves are not connected. The controller routes rows of pixels to one side or the other as appropriate. This is a simple way of cutting the capacitance seen by data on these busses in half.

4.2.3 EZW Unload Network

The EZW unload network, shown in Figure 4.4, serially delivers 4 bit EZW symbols from the PE array to the arithmetic coder. Again, a separate network is used in order to spread the activity of the coder over the entire frame time. The structures and circuits used for the unload network are the exact counterpart to those used in the pixel load network, but in reverse. All the same comments apply with regard to use of busses, gated clocks, segmented wires, and the central backbone.

The EZW symbol ordering described in the previous chapter is the counterpart to the dicing of the pixel scanning order into 16 passes. In total, when taking into account the ordering just referred to, the 8 bit plane maximum chosen, and the special treatment of 3 dominant symbol regions also described in Chapter 3, the maximum number of EZW unload passes per frame is 109.

It may seem puzzling that the incoming pixels are delivered in the course of 16 passes, while 109 are used to extract an already highly compressed output stream. The discrepancy is that an overwhelming fraction of the EZW symbols are NOPs. This cannot really be helped since, the arithmetic coder does not have information to predict where the NOPs

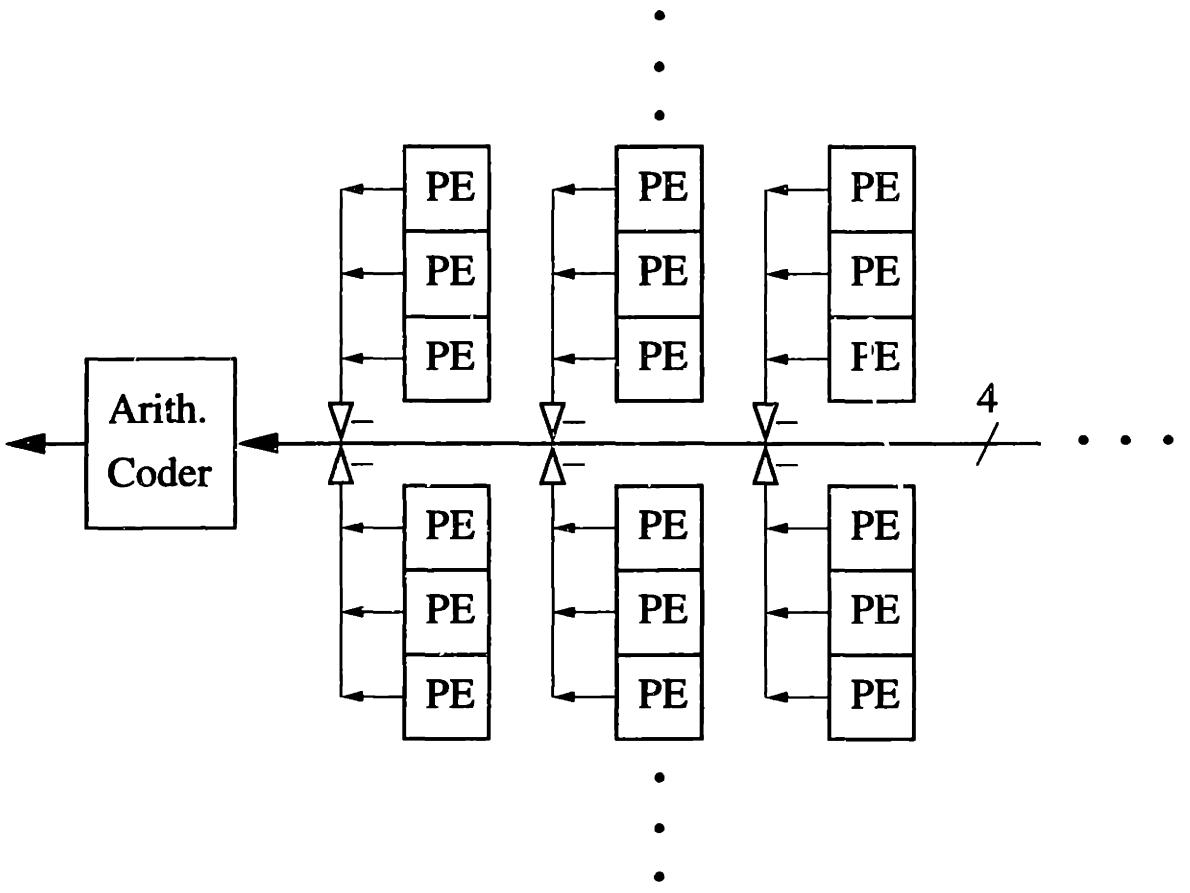


Figure 4.4: Block diagram of EZW unload network.

will be. Fortunately, this also comes at a fairly small price. The throughput of the arithmetic coder is minimally burdened by the volume of NOPs (more on this below), and a large succession of identical symbols does not cause any power consuming switching on the data busses. Only the associated clocking power is still spent.

Table 4.1 shows the encoding of the 9 symbols used onto the 4 wires. These symbols are decoded, in some cases split into multiple sub symbols, recorded in histograms, and entropy coded by the arithmetic coder.

Code	Meaning
0000	NOP (part of a zero-tree)
001B	already significant sub bit = B
1S1B	new significant sign = S, sub bit = B
0100	zero-tree root
0110	isolated zero

Table 4.1
EZW symbol encoding.

4.2.4 Global OR Network

The global Or network, shown in Figure 4.5, is a one bit wired Or used to return control information from the array to the sequencer. For example, the global Or is used by the sequencer to poll the PEs for significant wavelet coefficients in order to identify empty bit planes, as described in the previous chapter. The global Or network could also be used to extract more localized information if combined with the PEs' ability to determine their own addresses, described below.

The labeling of this network as a global wired Or indicates functionality, not implementation. In fact, for latency reasons, only small segments are actually wired together, and the entire network is precharged. There are a total of 65 wired segments, 32 upper and 32 lower half columns, and the backbone.

Each use of the network occurs over two microcode cycles, precharge and evaluate. During the first, the 65 segments are precharged in parallel. In the second, PEs leave off or turn on evaluate devices attached to the column wires. Each half column is buffered before triggering an evaluate device on the backbone, in a Domino fashion. Note that

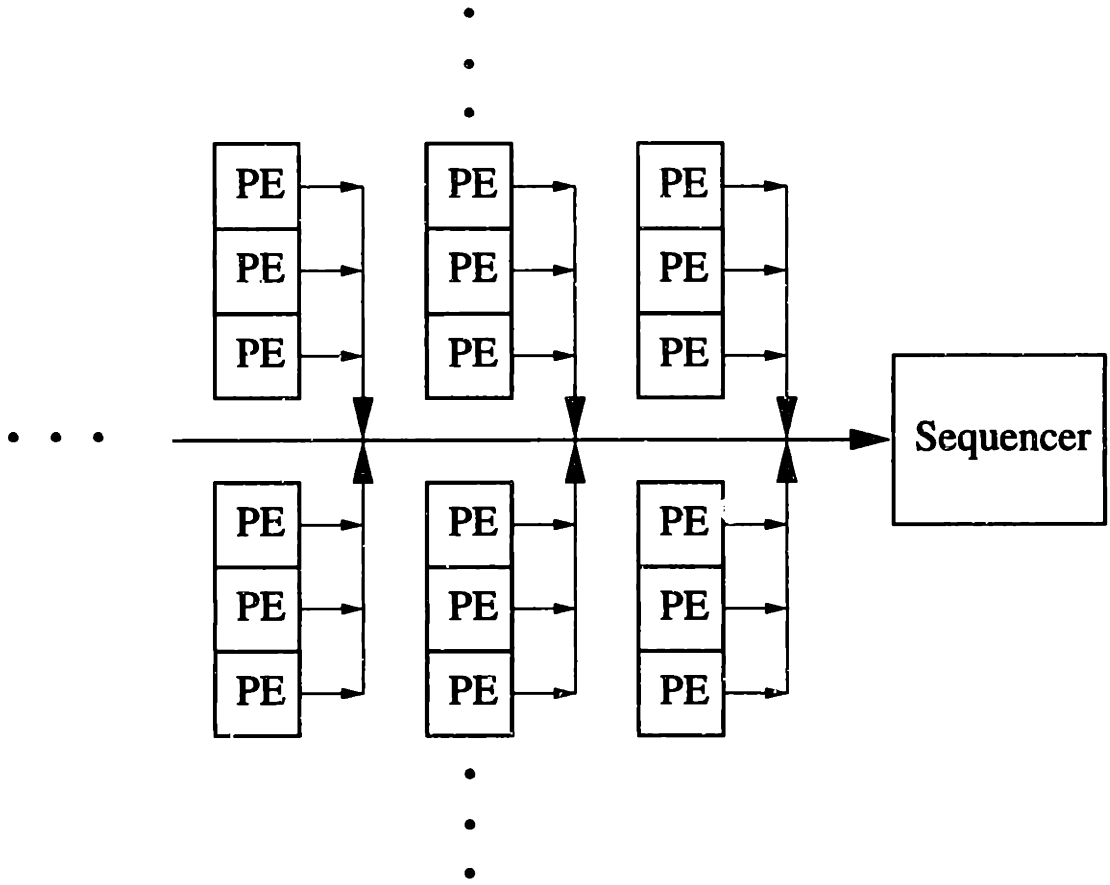


Figure 4.5: Block diagram of wired OR network.

separate cutoff devices in each PE are dispensed with. Before precharge, all PEs turn off their evaluate devices, under microcode control.

In addition to doubling as cutoffs, the evaluate devices can also act as keepers, if needed. During prolonged periods of disuse, evaluate devices could be forced on under microcode control. This would prevent high impedance column wires from leaking charge, experiencing degraded voltage levels, and triggering massive dissipation losses in the form of overlap currents in the buffers to the backbone. This usage is only necessary if the sum of the device thresholds, $|V_{TP}|$ plus V_{TN}, is smaller than the operating voltage. If

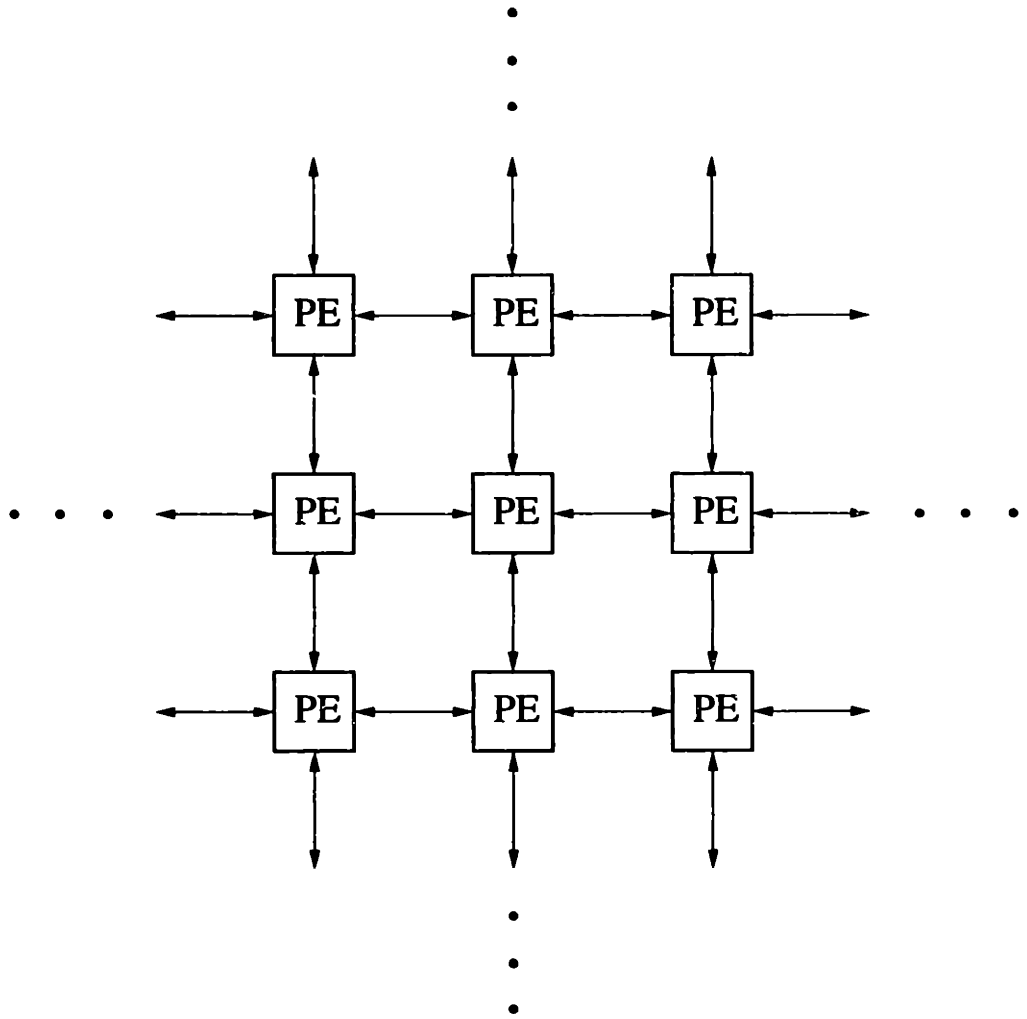


Figure 4.6: Block diagram of NEWS network.

the sum is larger, the buffers effectively become Schmitt triggers, and no overlap currents are possible.

4.2.5 NEWS Network

The one hop NEWS network, shown in Figure 4.6, carries all inter PE communication. This includes pixels (during filtering or motion compensation), wavelet coefficients, and EZW symbols or state information. The NEWS network also operates in a SIMD fashion. All

PEs transmit in the same direction at the same time. At the array edges, known values (all 0s) are supplied as inputs to the NEWS network to prevent floating voltages and spurious overlap currents.

The NEWS network is capable of carrying full 12 bit values, though a very small proportion of the traffic (wavelet coefficients) is the full data width of the PEs. (Pixels are 8 bits, and most zero-tree state information is 1 bit. No serious penalty is paid for this mismatch. The NEWS clock wire must span all the bits anyway, and unused bits will typically be zeroed out, avoiding data switching at those positions.) Normally, values would be transmitted one hop per cycle. However, because of the omission of motion compensation from the test chip, and therefore most of the NEWS traffic, the opportunity was taken to save some area. The test chip time multiplexes the 12 bit data onto 3 wires per direction, taking 4 cycles per hop. This is a fairly small savings. The full 12 wires would be used with inclusion of motion compensation. The cost estimate given in the previous chapter for motion compensation assumes one cycle per hop. (Not only would the throughput be increased to meet the demands of motion compensation, power per transmitted word would be reduced. Clock power would be reduced through fewer edges per word, and data power through the bypassing of extra internal capacitances of shift stages.)

4.3 PE Implementation

4.3.1 Overview

A block diagram of a single PE is shown in Figure 4.7. Both memory and datapath are 12 bits wide, following the algorithmic considerations discussed in the preceding chapter.

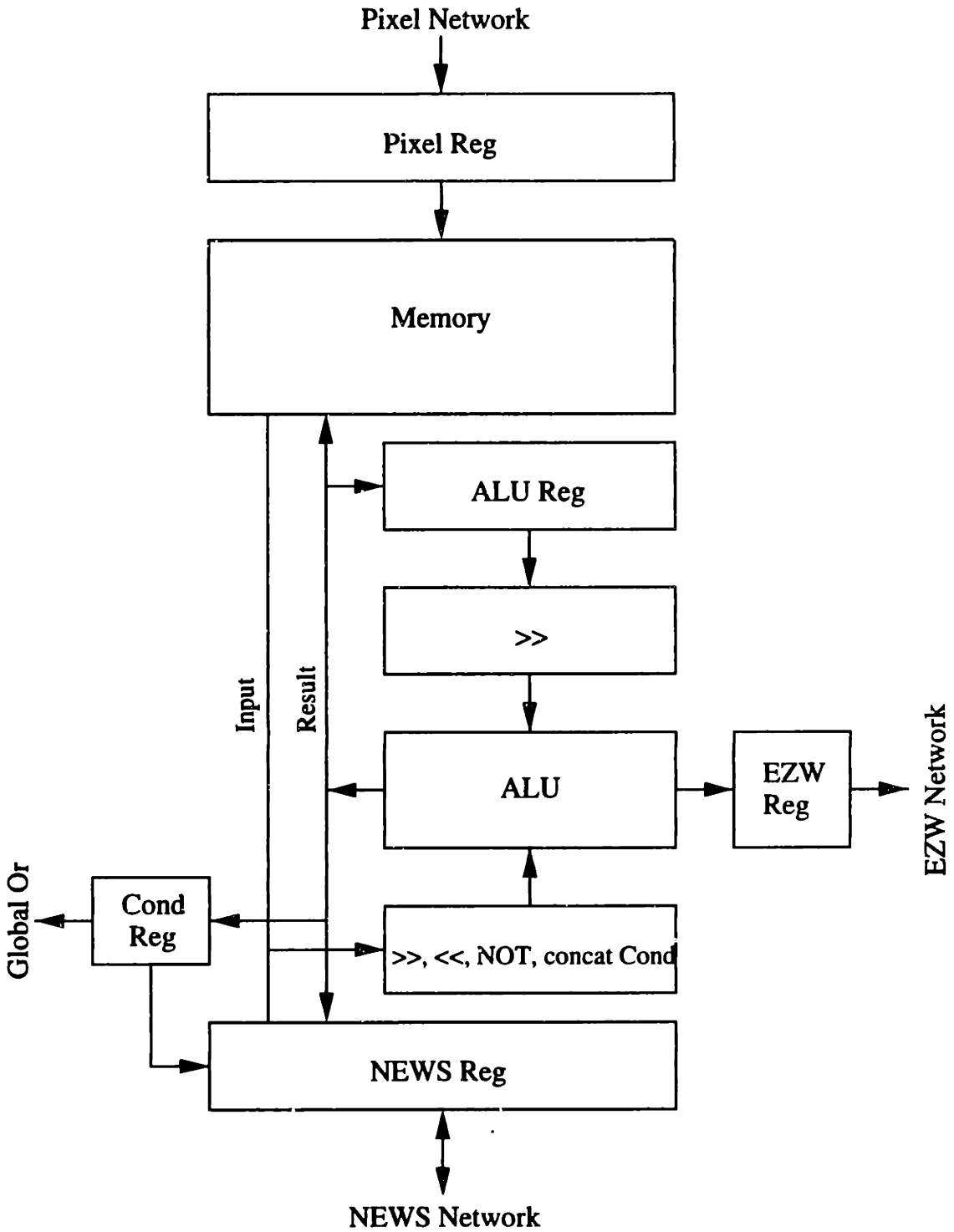


Figure 4.7: Block diagram of PE. Memory and datapath are 12 bits wide.

Each of the blocks shown is described in detail below. The pervading emphasis throughout the PE design is on simplicity (in fact, minimalism), both for power and area reasons. Section 5.3.6 shows the timing for PE control signals which are identically distributed to the entire array from the sequencer. PE timing is postponed until after the description of circuits which use those control signals. Some issues relating to instruction distribution are discussed later.

4.3.2 Memory and Pixel Load Register

The PE memories are hardly more than register files. In the absence of motion compensation, 3 frame buffers are required to execute the algorithm, as developed in the previous chapter. With 4x4 pixels per PE, this amounts to 48 words, 12 bits wide. The memory serves to store the previous, current, and next frames, as well as all required temporary workspace. A SIMD array cycle is split into read, modify, and write portions, so the PE memories perform 2 memory operations per cycle (1 read, 1 write).

It should be noted that the wavelet filtering operation requires more workspace than the current frame allotment. This is handled in a somewhat esoteric way. The start of the computation for each frame is synchronized to the arrival and delivery of the last pixel raster pass to the PEs. These are placed into the next frame buffer. By convention, this data should be transfered to the current frame buffer for encoding since the next frame buffer must be made available to the pixel loading process which continues at the first pass of the succeeding frame.

However, because the pixel load holding registers collect rows of data until an en-

tire pass is completed, the next frame memory locations, inside the PE register files, are not needed for $\frac{1}{16}$ of the frame time. Fortunately, the wavelet filtering operation is small enough to fit comfortably in this window. Therefore, the next frame buffer is transparently pressed into service, without interfering with the background loading process. Note that, transparently means it is up to the microcode to guarantee this condition is satisfied. No special circuitry is added to ensure against data collisions.

This situation would be modified in the presence of motion compensation. The filtering step occurs after motion estimation and compensation, which takes up most of the frame time. Clearly, the next frame buffer is no longer available. However, after motion estimation is complete, the extra shifted, previous frame buffer is no longer needed for its main purpose, and can be used for extra workspace.

The memory is constructed from pseudo DRAM cells in the test chip (the process used is a vanilla, logic only one). Circuit and implementation details of the DRAM elements and operation of bit and word lines is presented in the following chapters. The issue of DRAM refresh is addressed below with the array controller.

Of architectural interest in the memory design is the transparent loading of incoming pixels. As with the dual use of the next frame buffer, no special circuitry is used to prevent collisions between pixel loads from the holding register and normal microcode triggered memory accesses. This is accomplished through the use of a pseudo second memory port. The DRAM is not really dual ported, however, the bit line power optimization technique described in the next chapter incidentally results in the formation of a back door into the register file. The operation of this feature is described then.

4.3.3 ALU Register

The ALU register is used to preload the second operand required for some ALU operations. This is necessary because there is only one read phase per cycle. The ALU register is often used as an accumulate register. Many sequences of arithmetic operations can keep reading fresh data, combining it, and storing the result directly back to the ALU register with no intermediate fuss. To aid the efficient repeated use of the ALU register, its contents can be optionally right shifted one bit before delivery to the ALU. Sign extension is used to perform the right shift. This shifting is often used in sequences which perform multiplication, such as required by the wavelet filtering.

4.3.4 ALU and EZW Unload Register

The ALU can perform 3 different kinds of operations: logical, arithmetic, and EZW specific. Multilexers are used to select the desired output, as shown in Figure 4.8.

Logical

As with the ALU register's contents, the input provided by the read bus can be optionally manipulated before use by the ALU, however with considerably more choices. The input can be: right shifted one bit (sign extended), left shifted one bit (LSB is zero filled), have the LSB replaced by the contents of the 1 bit conditional register (described below), and be inverted.

Inversion is orthogonal to the other 3 choices and is performed last. That is, the input can be shifted and then optionally also inverted.

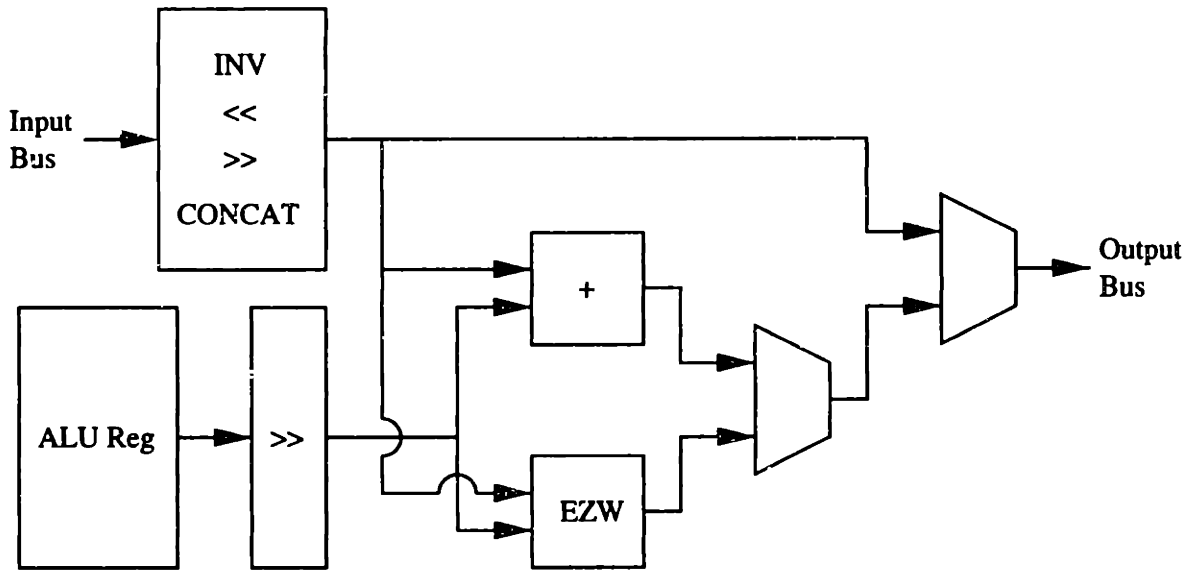


Figure 4.8: ALU block diagram shows the 3 modes of operation.

The other 3 choices are mutually exclusive. This decision was made because the same cycle use of more than one of the 3 options was considered rare. The benefit is the saving of an instruction signal (the concatenation of the conditional register to the bottom of the input word is signaled by the assertion of both shift down and shift up signals).

The logical mode of ALU operation simply routes the shifted and inverted input to the result bus, bypassing the rest of the ALU.

Adder

The 12 bit adder is the only arithmetic unit in the PE. Its implementation is covered in the next chapter. Multiplications are unfolded into sequences of shift and add operations, aided by the one bit shift options on both inputs. Reliance on the limited shifters to sequentially perform multiplication is a great power and area benefit. Multiplication is only

performed in the wavelet filtering step, which is a small part of the computation, even without motion compensation. On the other hand, barrel shifters, and especially multipliers, are large, complex circuits, which would needlessly occupy area and load ALU input and output nodes most of the time.

Subtraction is performed in conjunction with the input inversion. Note that only the input can be subtracted since the ALU register cannot be inverted. As can be seen in the microcode in Appendix C, this restriction does not result in any inefficiencies. A study of microcode examples was used to optimize out a possible ALU register inversion. Several cases are used to compute the adder's carry in. Normally, the carry in is 0. If both operands are being right shifted (multiplications), the carry in is set to the AND of the two discarded LSBs. This action aids rounding. For subtraction or increment operations, the sequencer can force all PE carry ins high.

EZW Specific

After wavelet filtering and frame differencing, coefficients are prepared for quantization and coding. Two's complement coefficients are separated into sign and magnitude, magnitudes are rounded as appropriate for the number of bit planes chosen, and sign and magnitude are packaged together into one 12 bit word along with state information such as bits which flag coefficients already found significant.

The coding of each bit plane is split into two portions. The first collates zero-tree root information by working from high to low frequencies. That is, the state and significance of coefficients is propagated up from leaves to zero-trees roots to find the highest level at

which insignificant zero-trees can be represented and coded. The second portion uses the state of each coefficient and its possible inclusion in zero-trees to code instructions for the arithmetic coder. This second stage progresses from low to high frequencies, as dictated by the algorithm. In the course of the second stage, results of the first stage are propagated back down the trees to inform coefficients about their inclusion in insignificant zero-trees at higher levels.

The propagation of zero-tree information both up and down the tree involves one bit logical operations and is handled by the conditional register, as described below. The coding of symbols during the downward stage involves more substantial operations, and these are hardwired in specialized logic which constitutes the ALU's 3rd distinct mode. In this mode, the ALU performs all relevant computations in a single cycle, at the cost of a modest amount of special logic.

The EZW logic takes coefficient magnitude and state information, assembled in the ALU register and the input bus, and produces two results. The 4 bit arithmetic coder instruction, shown in Table 4.1, is routed to the EZW holding register. Once loaded with a pass worth of symbols, those holding registers throughout the SIMD array are subject to control by the arithmetic coder, which serially raster scans the PEs and processes the symbol stream, as described previously. The holding registers have tristate outputs which are sequenced one row at a time to perform the first dimension of parallel to serial conversion (the second occurs in the trench backbone).

The second ALU result, routed to the result bus, is the new state for the coefficient. This is written back to the current frame buffer and will be used again in succeeding bit planes.

Table 4.2 shows the format of inputs to the EZW logic. Table 4.3 gives equations for the outputs computed.

Source	Bit Field	Name	Function
ALU Reg	11	FND	already found significant
	10:3	$BITS[7:0]$	current bit plane is at MSB
	2		0
	1	SGN	coefficient sign
	0	$\overline{ROOT}$	could not be a zero-tree root
Input Bus	0	POT	already part of a zero-tree

Table 4.2
ALU input sources and values for EZW logic mode.

Output	Equation
NOP	$POT + FND$
$NEWFND$	$FND + BITS[7]$
$ezw2a$	$\overline{SGN * EZWReg[3]}$
$EZWReg[3]$	$\overline{\overline{BITS[7]} + FND}$
$EZWReg[2]$	$\overline{NOP} * ezw2a$
$EZWReg[1]$	$FND + \overline{ROOT}$
$EZWReg[0]$	$NEWFND * BITS[6]$
$result[11]$	$NEWFND$
$result[10:3]$	$BITS[6:0]$, 0
$result[2]$	0
$result[1]$	SGN
$result[0]$	POT

Table 4.3
ALU output equations for EZW logic mode. Includes intermediate shared terms in top lines.

4.3.5 NEWS Register

Despite its name, the NEWS register serves 3 purposes. The obvious one is to interface to the NEWS network. A straightforward assortment of muxes and tristate buffers is used to route data to and from the appropriate direction.

The second function is to enable PEs to execute conditional branches of the instruction sequence. This is accomplished in conjunction with the conditional register, which is used to compute a condition flag based on appropriate local conditions for each PE. The NEWS register can then conditionally load or ignore a particular result based on that value (ie. the conditional register acts as a load enable for the NEWS register). Note that two distinct modes are available for loading the NEWS register, the one just described, and an override mode in which the sequencer forces all PEs to ignore their conditional registers and load NEWS regardless.

The final utility of the NEWS register is as a generic, 1 word, low power cache. That is, intermediate values to be used in the near future, or used frequently, can be stored in the NEWS register instead of a workspace location in memory. Access of values stored this way save the power of traversing bit lines and bit line peripheral circuitry.

4.3.6 Conditional Register

The conditional register also serves 3 purposes. It complements the NEWS register for conditional instruction execution by acting as its (overrideable) load enable. Secondly, the conditional register is used to compute arbitrary 1 bit logical operations, for example, zero-tree significance information. Finally, the register controls the evaluate/cutoff device

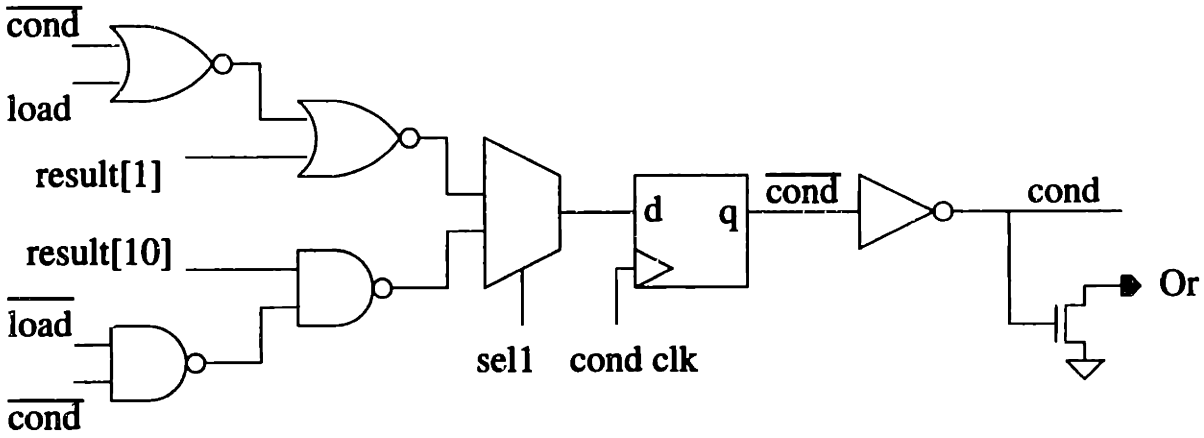


Figure 4.9: Functionality of conditional register.

Possible conditional register functions
cond = result[10]
cond = result[1]
cond = cond * result[10]
cond = cond + result[1]

Table 4.4
Functionality of conditional register.

wired to the global Or network.

Besides being wired to the NEWS load enable, functionality is facilitated by making the conditional register available for write back into an input word. As described above, the 1 bit flag can be concatenated to the bottom of the input word to the ALU. In addition, the conditional register can be loaded in a number of ways, as shown in Figure 4.9 and Table 4.4. With a minimum of logic and instruction signals, a great deal of flexibility is provided to load and compute conditional values.

New inputs can be taken from two different bit positions of the ALU result. The result bus is sourced so that incoming data can benefit from optional shifting and inversion. Bit

positions 10 and 1 (out of 11 to 0) are chosen for two reasons. The separation of the two choices makes 6 of the 12 bits available in a single cycle (in combination with input up or down shifting). Second, the two choices are strategically placed near positions of interest: sign bit, MSB, LSB, etc ... New inputs from either location can either load fresh data, or be logically combined with the current value. Note that for flexibility, both AND and OR functions are provided as operators. However, for simplicity, only one or the other is available at each bit position. The coupling of AND with bit 10, and OR with bit 1, is combined with careful placement of relevant bit operands inside words to achieve very efficient use of the resources. This optimization is due to the moderate cleverness of the author (enough to carry it out, not enough to have found a better use of his time).

4.3.7 Address/Position ROM

Algorithmic requirements make various operations dependent on the location of a PE in the array. For example, filtering treats image edges specially, and mapping subband coefficients onto groups of 2x2 PEs, as shown in Figure 3.8, depends on the location of a PE in its group.

To satisfy this requirement, a 49th read only memory location is included in the register file. The contents of the ROM word, whose value varies from PE to PE, are shown in Table 4.5. Following the discussion above, the flags are positioned near bits 10 and 1, and it should be remembered that their use can include convenient inversion. (Note that access of bit positions 10 and 1 themselves is slightly more power efficient than their neighbors, since the shift up or down control signals are not switched. The row and column position information is accessed more often than the edge positions, and are placed in the sweet

Bit	11	10	9	8	7	6	5	4	3	2	1	0
Flag	right edge	odd col	left edge	X	X	X	X	X	X	up edge	odd row	down edge

Table 4.5
PE position/address ROM.

spots. This is an indicator of the author's compulsiveness.)

4.4 Controller

The controller serves as microcode sequencer and glue logic between the SIMD array, the arithmetic coder, and the pixel and global Or communication networks. (The NEWS network is wholly contained in the array, and the arithmetic coder controls the EZW network directly.) The controller contains instruction decode logic, generates PE timing signals, controls pixel loading, keeps a program counter and other state to sequence instructions, interacts with the arithmetic coder, and schedules PE DRAM cell refreshing. These functions are discussed below.

4.4.1 Instruction Decode

There are two major categories of microcode instructions: array instructions, listed in Table 4.6, which cause the array PEs to take some action, and control instructions, listed in Table 4.7, which leave the SIMD array idle. A C program which assembles microcode into the opcodes listed in the two Tables is given in Appendix B. Microcode based on this instruction set which implements the algorithm of Chapter 3 is given in Appendix C.

Bit Field	Value	Function
19:18	01	array inst., ALU input shift down
	10	array inst., ALU input shift up
	11	array inst., no ALU input shift
17	0	no ALU input inversion
	1	ALU input inversion
16:11	0-47	ALU input from mem. address 0-47
	48	ALU input from ROM position flags
	49	NOP, ALU unused this cycle
	51	ALU input from NEWS register
10:5	0-47	store ALU output to mem. address 0-47
	48	store bit 1 to cond. register
	49	store ((bit 1) or cond.) to cond.
	50	store bit 10 to cond. register
	51	store ((bit 10) and cond.) to cond.
	52	store to NEWS unconditionally
	53	store to NEWS conditional on cond.
	54	store to ALU register
	55	NOP, no store this turn
4:3	00	ALU output = input with shifts and inversion
	01	ALU output = EZW state update
	10	ALU output = adder
	11	ALU output = adder, with (ALU reg. >> 1)
2:0	000	NOP, no special action
	001	concatenate cond. reg. to bottom of input input = input[11:1], cond. reg.
	100	NEWS transmit South
	101	NEWS transmit West
	110	NEWS transmit North
	111	NEWS transmit East

Table 4.6

SIMD array instruction opcodes. Table shows bit fields, values, and effect for array microcode instructions.

Bit Field	Value	Function
19:18	00	control instruction
17:14	0000	decrement global counter
	0010	global counter = 11 - (EZW bit planes)
	0011	global counter = (EZW bit planes) - 2
	0110	start arith. coder pass and stall until done
	0111	stall until new frame loaded
	1000	NOP, just a jump instruction
	1001	precharge global Or
	1010	set subordinate possible bit
	1011	clr subordinate possible bit (last plane)
	1100	set 4 dominant symbol alphabet
	1101	clr 4 dominant symbol alphabet
	1110	reset coder histograms
	1111	reset coder encoding state
13:11	000	no jump
	100	jump if global counter not 0
	101	jump if image_mean[7] = 0, and rotate image_mean (ROL)
	110	jump if last frame in synch group
	111	jump if empty plane
10:0	addr	jump address

Table 4.7

Sequencer instruction opcodes. Table shows bit fields, values, and effect for controller instructions.

4.4.2 Clocking

For the test chip, in the absence of motion compensation, it was found convenient to clock the controller and SIMD array at the incoming pixel frequency (128 x 128 x 30), about 500KHz. This clock is externally provided, and assumed to be 50% duty cycle. Both edges are used as seeds for PE timing generation. Various timing points, resulting in signals shown in Section 5.3.6, are derived by delaying either clock edge by some amount. In a commercial product, the desired delays would be ensured against process variations with an automated feedback scheme. For simplicity, the test chip uses variable delay elements which are controlled from external static chip inputs.

4.4.3 Pixel Loading

Serial to parallel conversion of the pixel stream is controlled with a straightforward assortment of shift registers, clock gates and buffers, etc ... Note that, full 8 bit wide busses are used to collect rows of pixels in the trench backbone. However, the distribution to PE rows is 4 to 1 time multiplexed over 2 wires per PE column. This trades a negligible amount of row clocking power, for a small area benefit.

In addition, the controller counts pixels for start of frame synchronization, and accumulates the image mean each frame for subtraction from the image and transmission to the decoder.

4.4.4 Arithmetic Coder Control

The controller has various small hooks to affect the arithmetic coder's behavior, some automatic, others under microcode control.

- The microcode determines if a 3 or 4 symbol dominant phase alphabet is in effect.
- The microcode signals the last bit plane, in which case no subordinate bits are used.
- The microcode signals that EZW holding registers are prepared with another symbol pass. In the test chip, this causes the controller to start the arithmetic coder and idle itself and the PE array until the pass is complete. With motion compensation, this throughput cannot be lost and an interrupt or polling scheme would have to be used to allow the array to continue to process the next frame's motion estimation step.
- The controller automatically causes the coder to include the image mean into the output stream at the appropriate time (beginning or end of frame, depending on whether it is a lead synchronization group frame or differential frame). A special mechanism causes the coder to spit out individual bits without triggering a symbol pass encoding.
- Using the same single bit output mechanism, the controller automatically inserts bit plane significance flags when the microcode computes such information.
- The microcode orders symbol frequency histograms to be reset.
- The controller orders the coder to flush remaining state to the output stream and reset when a 16 frame synchronization group is complete.

4.4.5 Microcode Sequencing and Synchronization

Sequencing is accomplished with an 11 bit program counter and other assorted state. The microcode in Appendix C fits comfortably in 2K instruction words.

The program counter operates in 3 different modes: it can increment during normal operation, it can be halted during idle periods, and various jump instructions can load new values from the microcode. The PC is halted under two conditions, while the arithmetic coder is processing an EZW symbol pass, and while the sequencer waits for a new frame of pixels to finish loading into the PE array. Jump instructions can act on the following 4 conditions:

- A 3 bit counter which can be loaded and decremented is used to count encoded bit planes. The sequencer can branch on a test of the counter's value (0 or not). This counter is sometimes loaded with a known nonzero value and immediately branched on to simulate an unconditional jump instruction.

- A 4 bit counter tracks the number of frames encoded in a 16 frame synchronization group. The value of this counter can also be branched on.

- The sequencer can branch on the value of the significant bit plane flag, which is collected from the PEs via the global Or network.

- The pixel mean for each frame is collected into an 8 bit accumulator. The sequencer can branch on the MSB of the image mean (the mean automatically executes a logical rotate left operation whenever this branch instruction is encountered).

4.4.6 DRAM Refresh

Refresh of PE DRAM cells is handled in a variety of ways. A single, general method is avoided to save power. Different methods are tailored to the needs of different frame buffers at different times.

- Refresh operations are omitted in some cases, and DRAM cells allowed to lose their state to leakage currents. This is the case for memory elements belonging to the current frame buffer, after the encoding of a frame is complete and the array idles until the next frame. At that point, all required state is contained in the next and previous frame buffers. The current frame buffer will be initialized with new data after another frame of pixels is loaded.

- Some refresh operations are automatically scheduled by the sequencer. For these cases, the sequencer contains special logic which schedules the refreshes at an adequate rate to avoid unacceptable leakage levels, and to avoid collisions in the PE memories between the refresh operations and normal microcode memory accesses. This is done for the next and previous frame buffers, during array idle periods, such as arithmetic coder passes and next frame wait. Restricting automatic refreshing to the idle periods simplifies the collision avoidance circuitry, since no normal memory accesses are occurring (only need to check for idling, instead of checking for memory use or interrupting the instruction sequence).

- Refresh operations may be explicitly included in the microcode. This is periodically done for the current frame buffer before the frame encoding is complete, and for the

next and previous buffers during long continuous periods of instructions with no idle breaks (arithmetic coder passes). This last method determines the worst case time DRAM cells must keep their state without refresh. The current frame buffer is not automatically refreshed by the sequencer during the idle arithmetic coder passes (after which the current frame buffer is still needed, except for after the last pass of the frame). Therefore, the cells must survive the longest possible single arithmetic coder pass duration, which is about 4m sec. The DRAM cells, discussed in the next chapter, are sized accordingly.

4.4.7 Chip Input Timing

The test chip takes 4 sets of input signals: a 50% duty cycle 500KHz clock, an 8 bit pixel, a reset, and a start signal. The separate reset need only be pulsed for one clock cycle at any arbitrary point before the start sequence shown in Figure 4.10. The Figure shows the relationship between the start signal and the first delivered pixels. It is assumed that the preceding ADC switches pixel data on positive clock edges. The test chip samples the

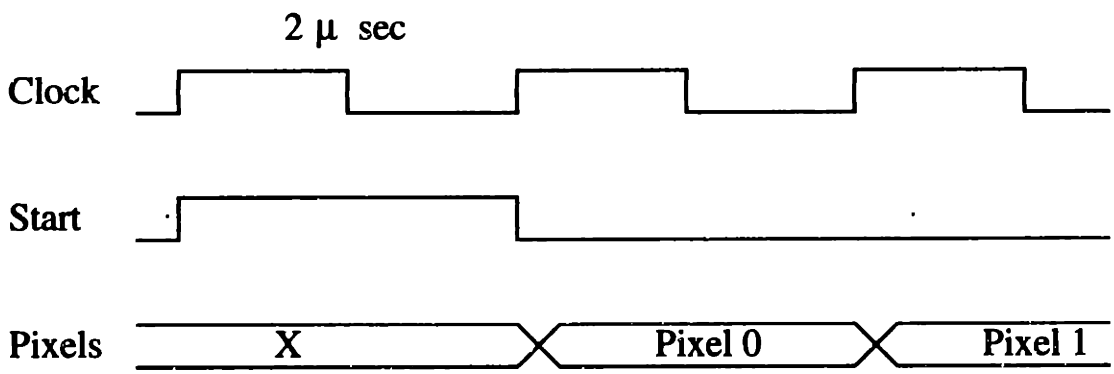


Figure 4.10: Timing of input clock, start signal, and pixels.

pixels on negative edges.

4.5 Arithmetic Coder

The arithmetic coder is largely a straightforward and verbatim implementation of decoding, histogram tables (registers), and data path logic (adders, barrel shifters), following the C code which appears in Appendix A. Two aspects of the coder are architecturally interesting.

4.5.1 Clocking

First, the coder operates off a self generated, and gated clock, which is asynchronous with the sequencer and PE array. The coder operates at a much higher frequency than the sequencer and array because it processes the EZW symbols serially. In addition, the number of cycles needed per pass is unknown because of the unpredictable nature of arithmetic coding and the variability of the kind of EZW symbols arriving. Therefore, the coder and sequencer signal the start and end of EZW symbol passes with a simple two wire asynchronous handshake protocol.

During operation, the coder clock is generated with a simple ring oscillator, which is much easier than a frequency multiplier synchronized with the array clock. After the completion of each pass, the sequencer gates off the coder's clock while the PE array prepares the next pass of symbols (or until the next frame after the last pass). The coder may experience considerable off periods, so the clock gating saves significant power (the coder is small, but the frequency is high). Note that using a frequency divider from the fast

coder clock to the slower array clock is out of the question since the slow clock is needed continuously, while the fast one is not.

The frequency of the coder is set to cover the worst case, maximum number of cycles required per frame. For example, the test chip uses a 5Mhz ring oscillator (compare that to the 500KHz main array clock). As with the PE timing generation, externally controlled variable delay elements in the ring oscillator produce the desired frequency. Note that, for obvious reasons, the EZW network is timed by the coder's clock. The array clock is used up to the EZW holding registers inside the PEs. Past that point, everything leading up to the coder is synchronized to its clock.

4.5.2 NOP Lookahead

The second interesting architectural feature is a decoding lookahead for NOP symbols. Many symbols delivered to the arithmetic coder through the EZW network are NOPs. These are decoded and immediately discarded without triggering any activity inside the coder. On the other hand, other symbols typically require several coder cycles to process. Consecutive substantial symbols cannot be decoded and processed without interruption. While the coder processes a preceding symbol, the EZW network is halted. Thus, the parallel to serial conversion is executed in fits and starts. An exception is made if succeeding symbols are NOPs. Thus, even during the processing of a previous symbol, the EZW network is allowed to continue so long as only NOP symbols are decoded. (The decoding front end can operate independently from the rest of the coder.) While this scheme cannot reduce the worst case number of cycles and therefore reduce the operating frequency, the average number of cycles is substantially reduced, lowering the coder's clocking power.

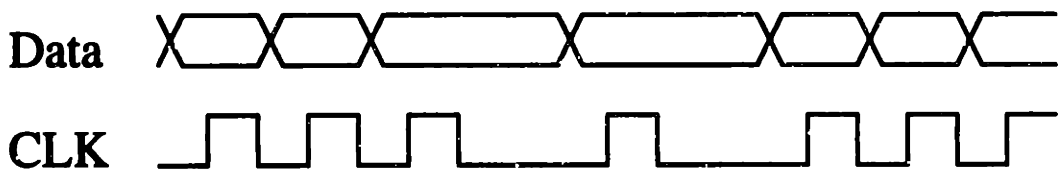


Figure 4.11: Timing of output data and clock.

4.5.3 Chip Output Timing

The test chip outputs 3 signals, a bursty serial data out, a sampling clock which accompanies the data out, and an asynchronous end of 16 frame group signal.

The output sampling clock is a gated version of the arithmetic coder's 5MHz internal clock. The output clock fires whenever a valid data bit is transmitted. Data is switched by the arithmetic coder on negative edges, and it is assumed that external circuitry samples the data on positive edges. The generated clock has a 50% duty cycle. Both data and clock are bursty for algorithmic reasons already discussed. The output clock remains low when idle. Figure 4.11 shows an example of output timing.

The end of 16 frame group signal is asynchronous with the serial output data. It pulses high for 2μ sec (one array clock period) to signal the end of a frame group. Both encoder and decoder are expected to reset their arithmetic coders at this point. The end of group signal is given well away from any output data activity.

Chapter 5

Circuits

5.1 Introduction

The most basic goal of low power circuit design is to ensure that power dissipation is $\frac{1}{2}CV^2f\alpha$ dominated. (Again, this is in the absence of adiabatic circuit techniques.) Toward this end, circuit forms which experience static dissipation, or prolonged overlap currents, are avoided. This includes ratioed and positive feedback circuits in which a fight between competing devices must settle before the circuit action completes. Care must also be taken to avoid triggering inadvertent overlap currents due to wandering input voltages. This may happen due to leakage currents at nodes left at high impedance for long periods (gated clocks), or charge sharing between suddenly connected high impedance nodes. Note that these kinds of overlap current are principally of concern if the operating voltage, V, is comparable to, or greater than, the sum of device thresholds, $V_{TN} + |V_{TP}|$.

In addition to avoiding current losses, circuit techniques may also be employed to reduce dynamic dissipation, by lowering Cs and αs. Clock gating, the architectural equiv-

alent of this, was already discussed in the previous chapter, as applied to both the SIMD array and arithmetic coder. Clock gating can be thought of as lowering $C\alpha$ in time. Segmenting both data and clock wires, described in examples below, is the equivalent benefit in space. The sizing of devices is covered in detail in the following section. Layout and floor planning techniques used to reduce Cs in the test chip are discussed in the next chapter.

5.2 Device Sizing

In marked contrast to typical standard cell design, device sizing used throughout the test chip leans heavily toward small or minimum widths. This is true for both the all custom circuits used in the PE design, and the cell library used for the sequencer, arithmetic coder, and communication network backbones. Two principal considerations motivate this trend.

The combination of a fixed computational throughput (fixed maximum, but not always constant), copious available parallelism, and process thresholds pushes voltage scaling near its limit. As discussed in the previous chapter, reducing capacitance is another way of trading circuit speed for further power reduction.

The second motivation stems from the generally short wires which load the circuitry. Relatively short wires permeate the design for a number of reasons. Specific algorithmic concessions were described in Chapter 3 which ensure communication locality in the SIMD array. Second, the PE is a hand crafted, all custom layout because it is duplicated over 1000 times and occupies most of the chip area. Third, despite the use of a fairly generic cell library (except sized down in device widths) for the peripheral circuitry, the placement was

done by hand (no auto place and route was used anywhere on the test chip), for various reasons.

It is desirable to keep the trench separating the two array halves small and tidy in order to minimize the impact of the breach between PEs, which are designed to abut. The total area of the arithmetic coder and sequencer are not of great concern, but their aspect ratio is. Because both are pressed up against the massive and very smooth edged SIMD array, the coder and sequencer can impact the area of the whole chip out of proportion with their own individual areas. For this reason, both the coder and sequencer are laid out to be very narrow and tall, in order to efficiently press up against the SIMD array. This kind of awkward aspect ratio is very difficult to obtain from CAD tools, so hand placement was used instead. A side effect of this tender care is relatively short wires compared to what might be expected from a typical standard cell implementation.

Examples of the resulting device sizes will follow throughout the remainder of this chapter.

5.3 PE

5.3.1 DRAM Cells

The use of real DRAM cells was not an option on the test chip due to process restrictions (logic only, no trench capacitors). However, since memory constitutes so much of the architecture, even the marginal area benefit of pseudo DRAM is desirable. Rather than choosing the classic 2T cell, the verbatim translation of DRAM to a logic only technology, the 3T DRAM cell shown in Figure 5.1 is used. This trades a small area penalty for some

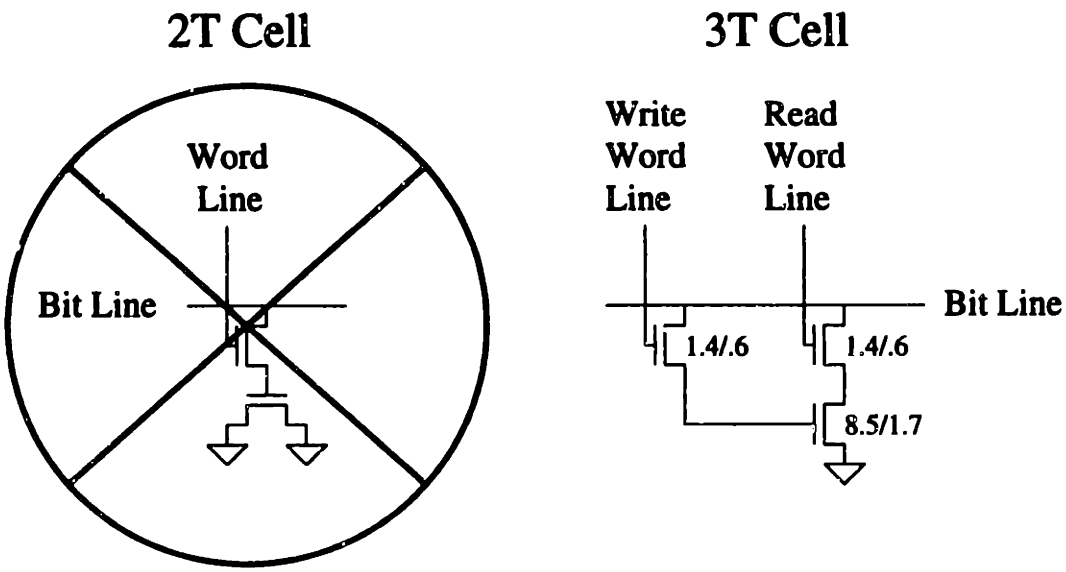


Figure 5.1: 2T versus 3T DRAM cell. Devices sizes are in microns. The storage device is sized to obtain a 4m sec refresh time. (The cell leaks away 10% of its charge in 4m sec.)

power benefit.

Because of separate read and write access points, the 3T cell is not subject to destructive reads, as the 2T cell is. This obviates the need for extra bit line peripheral circuitry and the switching power of timing signals required to control that circuitry. Such circuitry would be needed to perform a write back of the read value (in addition to the real write of the ALU output to the write address).

In addition, reading through an active pulldown device makes it possible to develop the read value on the bit line as a macroscopic voltage change. Use of the 2T cell would result in small voltage changes, as limited by charge sharing between the bit line and cell storage capacitance. Unfortunately, this is exacerbated by cell capacitance values obtainable in reasonable area in logic processes (compared to DRAM process trench capacitors). As in real DRAMs, the bit line read voltage would have to be greatly amplified by sense

circuitry, which costs power. Keep in mind that the sensing power is being amortized over a very small bit line with few cells. This is a different design point than optimally sized subarrays in large volume commercial DRAMs.

The area cost of using the 3T cell is due to the extra access device and word line. However, this cost is not as large as it might seem. The extra device is dwarfed by the large storage device. While the critical dimension of the cell is determined by the minimum metal pitch of word lines, as described in the next chapter, the cell area could not easily benefit from the elimination of the second word line. The layout of the cell is restricted by pitch matching considerations and aspect ratio limits.

The storage device is sized to accommodate the 4m sec refresh time requirement. Simulation of worst case subthreshold leakage was used to size the storage device so that 90% of the cell charge is retained after 4m sec. The next section describes a technique used to reduce the worst case leakage.

5.3.2 Bit Line Operation and Peripheral Circuits

Operation

Figure 5.2 shows a bit line and its peripheral circuitry. A SIMD array cycle consists of read, modify, and write phases. The read or write operations may not access the memory each cycle. Other sources and destinations may be used by the ALU, for example the NEWS, ALU, or conditional registers. To satisfy the maximum possible throughput, the memory is operated as shown in the simplified timing diagram of Figure 5.3. The real, more complex timing is shown in Figure 5.10.

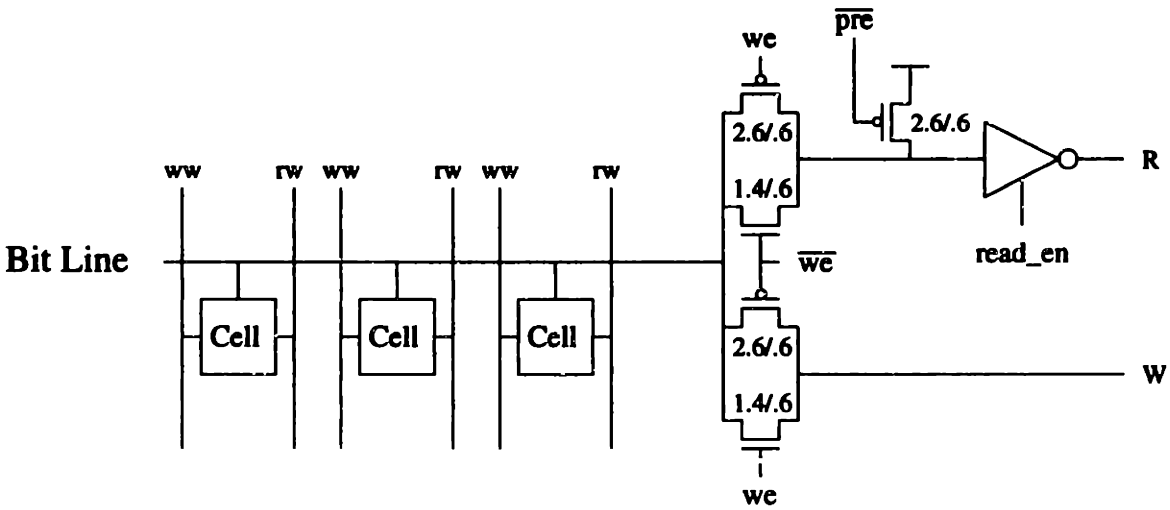


Figure 5.2: PE bit line and peripheral circuits, including sense, latch, and write.

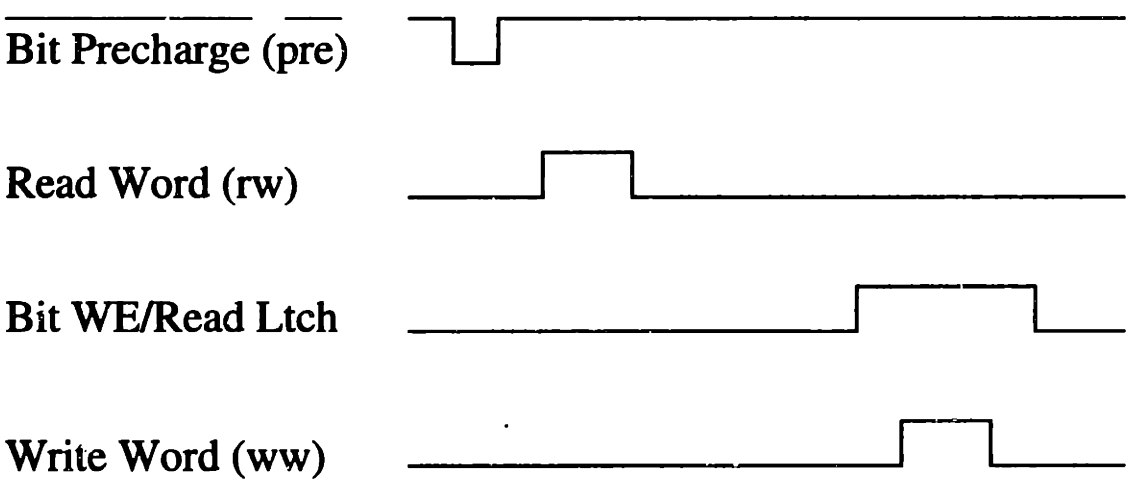


Figure 5.3: Simplified memory timing diagram.

A bit line precharge pulse precedes the read operation. Read word lines are not activated until after the end of the precharge pulse to avoid overlap losses. The read data propagates through the read latch to the input bus and ALU. The read latch requires a tri state output to allow the input bus to be alternately driven from the NEWS register.

If a write is required, the read value is latched and the write pass gate is enabled. Activation of write word lines is slightly delayed to avoid contamination of the read value. This may happen due to charge sharing between the high impedance bit line and write cell storage node. The deliberate delay avoids a race condition between signals triggered from the same sequencer timing event, but which pass through different circuitry. The release of the write enable is similarly delayed until after word lines are turned off. This eliminates possible contamination of the write value by charge sharing with the read latch internal node.

Note that write word lines must be raised to a higher ON voltage than the normal supply. This is required to be able to pass V_{dd} to cells through the N-channel only write access device. This is especially important with low operating supply, since V_{dd} may only be a few hundred mV higher than V_{TN}. Charging the cell storage node all the way to V_{dd} is critical to turning on the storage device for reads, as well as providing charge to hold the state for the required refresh time. To accomplish this, the ON voltage of write word lines must be at least $V_{dd} + V_{TN}$, including any body effects on the threshold voltage. The higher write word line voltage is provided by the controller, and is discussed in Section 5.6.1. The higher voltage used for the write word lines does not represent a great power burden. The word lines have small activity factors and account for a small part of chip power.

Peripheral Circuitry

The bit line peripheral circuitry is very light on both area and power. This frugality is due to several contributions:

- No buffering is needed between the result bus and bit line write port. A small, simple pass gate is adequate because the 48 cell bit line is so small a load.
- The sense amp is a simple inverter because the read value is developed as a large swing signal, as enabled by the 3T DRAM cell.
- The 3T cell does not require special refresh circuitry or timing to compensate for destructive reads.
- Instruction distribution power is saved by reusing the write enable signal for the read latch. In this case, there is no possible race condition with the write enable which might open the read latch too early or close it too late, because the same signal is used to synchronize the two events. In addition, any momentary overlap of an open latch and write pass gate due to finite transition time of the enable is too small to disturb any charges or voltages.

Note that, many of these circuit choices are an artifact of the logic only technology used for the test chip. With a mixed memory process, the operation of the bit and word lines would have to be reworked to gain the area benefit of real DRAM cells. However, the author does not necessarily believe that the correct solution would consist of typical DRAM circuit techniques. Again, the small granularity of the subarrays makes this an interesting, and different design point.

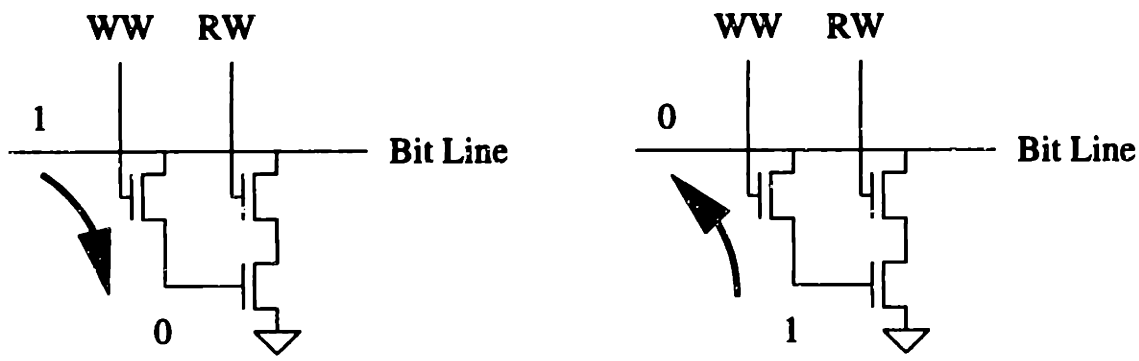


Figure 5.4: Two possible cases of cell state and bit line voltage which experience subthreshold leakage.

Cell Leakage Reduction

The most significant cell charge loss mechanism is subthreshold leakage (across the write access device). Diffusion to substrate leakage tends to be much lower. Subthreshold leakage occurs when a device source and drain are at different potentials, despite the device being cut off. Clearly, if a cell storage node happens to be at the same potential as the bit line, storage charge will only be lost to diffusion leakage (and that only if the cell state is high - only N-diffusion to substrate touches the storage node).

However, there are two cases in which subthreshold leakage does occur, as shown in Figure 5.4, and they exhibit very different behavior. A high cell state leaks away charge to a low bit line at a fairly uniform rate. However, a high bit line potential leaking charge into a low cell state experiences a very strong negative feedback. Waveforms demonstrating the two cases are shown in Figure 5.5. As small amounts of leakage charge accumulate inside the cell, the write access device's gate to source voltage is lowered (in fact, become negative). Since this exponentially reduces subthreshold current, the process quickly be-

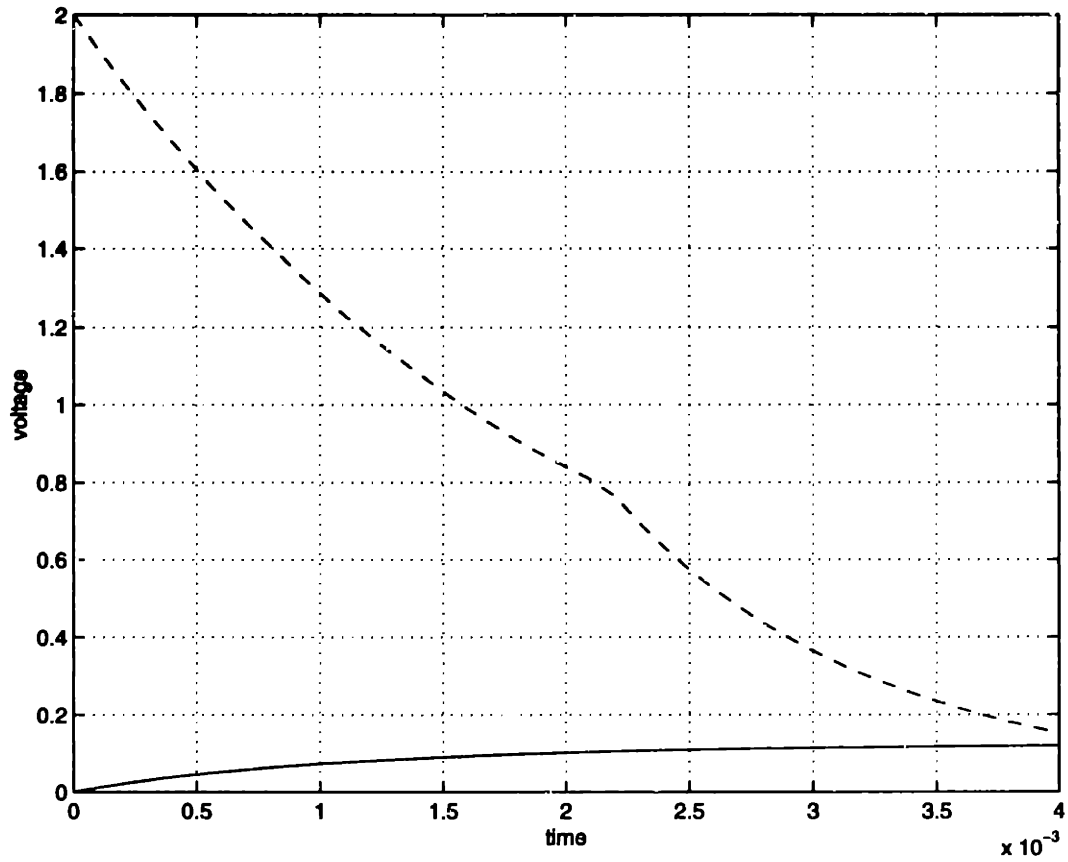


Figure 5.5: DRAM leakage waveforms. The two voltage waveforms show cell leakage to opposite state bit lines.

comes self limited. This presents an interesting opportunity to affect DRAM leakage rates, other than paying area for larger storage capacitance.

Note that, during periods of SIMD array activity, bit line potential is dictated by the statistics of the data that happens to move across it. The resulting, worst case refresh rate may be strongly influenced by the worse, low bit line potential case identified above. This is of little concern, however. Refresh during active periods is either not necessary because of assured regular access patterns (current frame/workspace buffer), or accomplished with explicit microcode instructions (next and previous frames) which account for

a tiny fraction of the active computation cycles.

Refresh requirements during idle periods are of great concern however, because refresh of previous and next frame buffers are required even during periods of no motion in the video stream. In this case, with little computation being performed and a largely inactive radio, refresh cycles may significantly contribute to the lower bound on system power. (Note that techniques discussed in Chapter 2, such as frame rate decimation, do not help the cost of DRAM refresh.) For example, with a 4m sec refresh time, 30 frames/sec, and 32 cells requiring refresh, 256 refresh cycles are needed per frame. This is already about 8% of the peak SIMD computational throughput of the test chip. Note that this refresh lower bound on idle power would not be swamped out with the inclusion of motion compensation. A simple frame differencing and threshold check would be performed before motion estimation. The considerable cost of motion estimation is avoided if no motion is detected.

Fortunately, during idle periods, bit lines are not occupied with data accesses, so bit line potential can be controlled to benefit cell leakage rates. This is done by keeping bit lines continuously precharged during idle periods. This restricts cell subthreshold leakage to the self limited case identified above. Note that no power is wasted switching precharge clocks. For this purpose, precharge signals are not pulsed. The sequencer enters a special mode, continuously forcing precharge clocks ON.

This circuit technique is of more general interest than some of the logic only process artifacts discussed above. It can be applied equally well to bit lines with real DRAM cells. Further, this represents a specifically low power circuit technique, in that the refresh rate benefit relies on the use of gated clocks and idle periods.

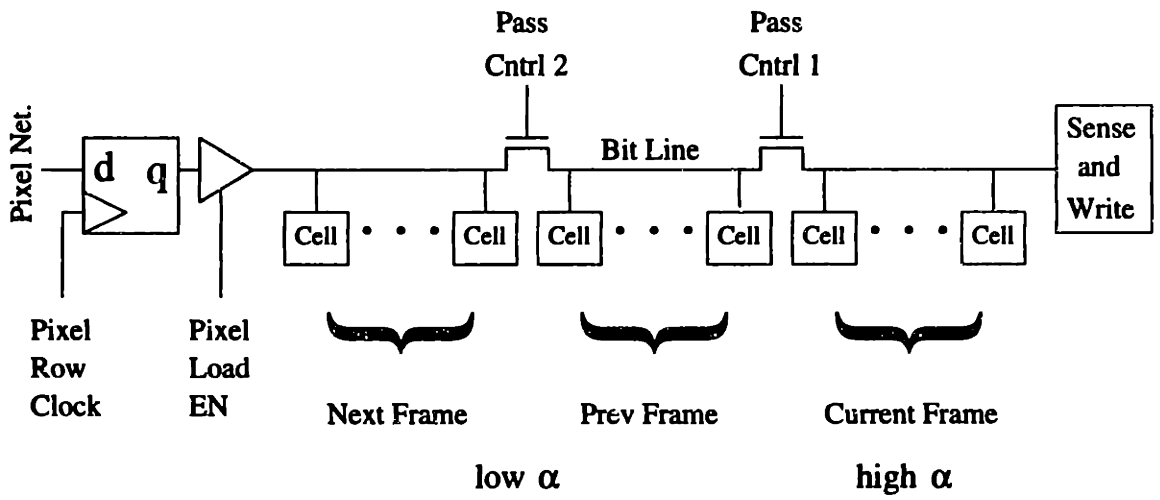


Figure 5.6: PE bit line segmentation and pixel load port.

Finally, note that if the read latches are forced open during idle periods, the continuous bit line precharging also coincidentally serves to keep the latch node at low impedance, preventing wandering potentials and unintentional overlap current in the sense inverter.

5.3.3 Bit Line Segmentation

Memory elements belonging to the different frame buffers experience widely varying access rates. The current frame buffer is accessed continuously throughout the frame computation. Previous and next frame buffers are accessed essentially once each. Similar variations would exist among the 5 frame buffers required with motion compensation.

To take advantage of this, the PE bit lines are segmented with pass gates, as shown in Figure 5.6. The frequently used current frame buffer is positioned in the first segment, close to the business end of the bit line. This results in lower average bit line power. When a cell on the first segment is accessed, the pass gate control signal remains off. The capacitance switched during precharge, read evaluate, and write is cut to one third of the entire bit

line. Only accesses to the last bit line segment switch the entire capacitance (in fact, a tiny bit more because of the extra pass gates themselves). This is negligible, however, due to the low access frequencies to the back of the bit lines. This technique effectively cuts bit line power by a factor of 3. While the full bit line power does not represent a dominating fraction of the encoder power (see Section 5.7), this optimization comes at practically no cost.

The inter segment pass gates are implemented with only N-channel devices. It is important for area reasons to avoid N-wells and routing V_{dd} inside the memory cell arrays. To allow full write voltage levels, the pass gates must be over driven, just as the write word lines are. Again, the higher switching voltage of the pass controls are not a power burden because of their low α. In addition, the pass gates add extra series resistance to the far segments of the bit lines, but the small loads involved makes this tolerable.

Bit line segmentation by access frequency is also of more general interest as a low power circuit technique, and could be used with real DRAM technology. It can be thought of as a cheap way of mimicking subarray division - cheap in that there is no need for duplicate bit line peripheral circuitry or muxing. The price of an occasional N-channel pass gate and (effectively) extra word line is quite small. This is important in real DRAM processes, especially with already small subarray sizes like the PE register files, because the cost of any extra substantial bit line circuits is amortized over a small number of memory cells.

Note that the access frequency argument presented so far only justifies the first pass gate at the $\frac{1}{3}$ point. The second pass gate at the $\frac{2}{3}$ point is added to provide the cheap, pseudo, second memory port hinted at previously. Because bit line segments are discon-

nected so long as the pass gates are off, new pixels can be loaded from the back of the bit line, without interfering with normal microcode accesses at the front, and without requiring special collision avoidance circuitry in the sequencer. To ensure the required conditions, microcode access to the next frame buffer is forbidden after $\frac{1}{16}$ of the frame time after the start of the frame (the time needed to collect the first pass of new pixels), as discussed before. The second pass gate is required because access to the previous frame buffer could not be prevented later in the frame. Finally, of course, the next frame is placed on the last bit line segment, with the previous frame occupying the middle position.

5.3.4 Adder

The adder cells used in the PE are shown in Figure 5.8. The two cells are alternated as explained below, in a ripple carry fashion. The XOR used in the adder, shown in Figure 5.7, is a standard transmission gate XOR.

These adder cells are an effort to exploit the long, 2μ sec cycle time, the small complexity of the PE, and the relatively narrow 12 bit word, in some additional way to voltage

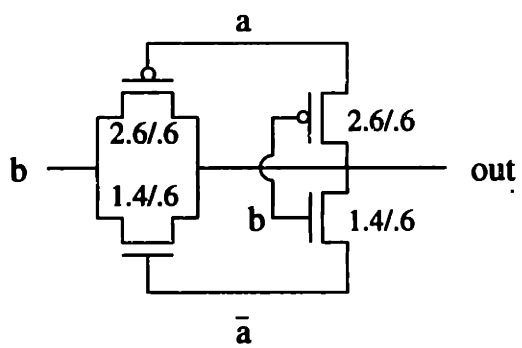


Figure 5.7: XOR gate used in adder cells.

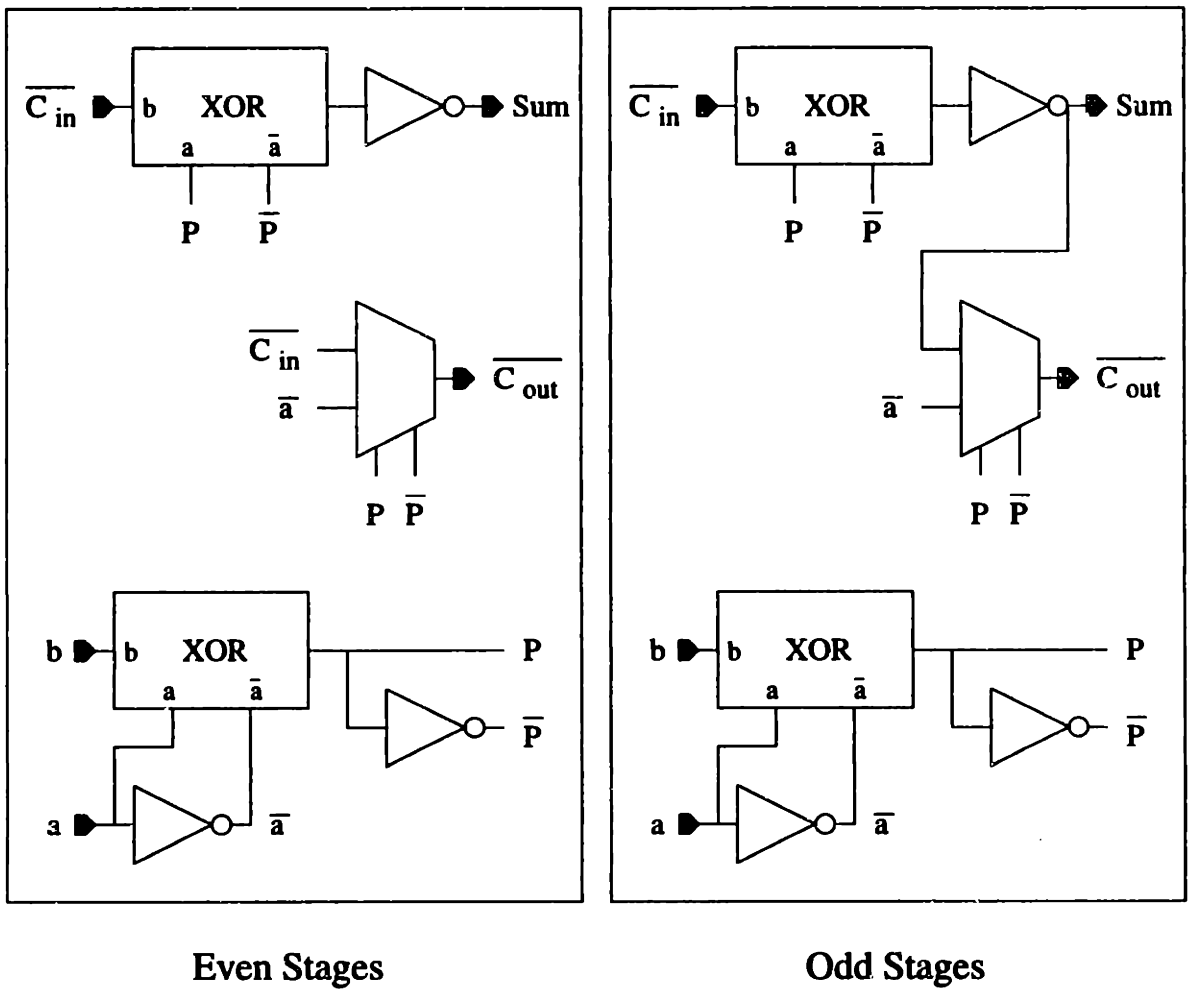


Figure 5.8: Static PE adder cells with minimized area and total capacitance.

scaling. Toward that end, the cells were designed to minimize area and switched capacitance. These adders are the smallest static adder cells the author is aware of, given that the SUM output inverters are needed for ALU output buffering anyway, and hence represent no additional cost (they are shown in Figure 5.8 to explain functionality).

Note that a precharged adder (especially precharged carry chain) was not used in order to avoid the significant power cost of another timing signal. It must be remembered that the switching cost of a carry chain through static circuits is limited by the average number

of bit positions carries propagate, which is typically between 2 and 3. On the other hand, a precharge clock switches at two edges per cycle, each cycle of ALU use, and spanning all 12 bits of the word.

The even stage cell shown on the left is a fairly standard transmission gate adder. Typically, ripple carry configurations of such cells use occasional (every 2 or 3 stages) buffers to interrupt the carry chain. This prevents very poor carry propagation through an unbroken, and quadratically degrading *RC* chain. The area and capacitance benefit comes from the use of the already computed SUM output in the odd stages as a way of introducing regeneration into the carry chain. This has the dual advantage of eliminating the extra, inter bit carry buffers, and of making the layout of different bit positions practically identical, as well as smaller.

Note that this is a very quirky adder, and the author does not necessarily recommend its use heartily. In particular, the reader should be aware that for the odd stages, the carry computation is dependent on, and slower than the SUM output!

5.3.5 Flip-Flops

Gated clocks are used for all PE registers: ALU, NEWS, conditional, EZW, and pixel. Each register is provided a separate clock which is pulsed on the register's active cycles. However, the registers must hold their state for possibly long idle periods. For example, the pixel holding registers are clocked every $\frac{1}{16}$ of the frame (about 2m sec), while the others must hold their state across arithmetic coder passes (maximum 4m sec). In addition, high impedance nodes, susceptible to leakage and drifting voltages, are avoided to ensure

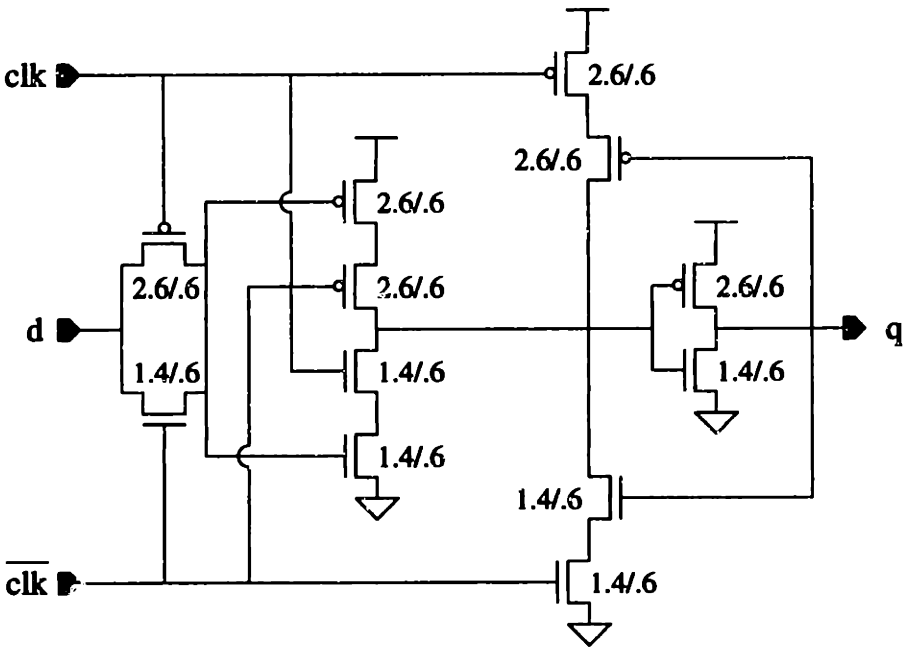


Figure 5.9: PE Flip-Flop with low idle state clocks.

against overlap currents, as discussed above.

For these reasons, static flip-flops like the one shown in Figure 5.9 are used. Since it is known that clocks are gated low during idle periods, only the second latch employs feedback. The first latch remains open during idle periods, and its storage node is not high impedance. The feedback path of the second latch is cut off by the series, clock controlled devices. This prevents a current wasting fight when the second latch is opened, which would alternately be the case if the feedback devices were merely made weak. Note that the lost current could be minimized by making the feedback devices very weak, for example by using minimum channel width and considerable length. However, this is not power efficient. Despite the two switching events per clock cycle, the series cut off devices are small loads on the clock. Very long channel feedback devices, however, would be a

sizeable load on the latch output.

In addition to having the required static property with minimum power impact, these flip-flops are also inherently very robust with respect to clock timing. This is taken advantage of in two ways.

The cost of producing the clock inverse is efficiently amortized over an entire row of PEs (32 of them) with one big row buffer, rather than the awkward layout of many small localized inverters. Besides the robustness of the flip-flops to tolerate some skew between the two clock polarities, this is further enabled by the known, uniform, and highly symmetric loading of the clock consumers in the SIMD environment. (Actually, as is evident from Figure 5.9, the two polarities are predictably not identically loaded. The positive clock drives two of the larger P-channel devices. This is taken advantage of by initially producing that polarity, and making the lighter loaded negative one suffer the extra inverter delay.)

Second, the flip-flops are fairly robust to poor clock rise and fall times. A small amount of power is saved by using modest row buffer sizes. Reasonable limits on these were determined with circuit simulation. For example, on the test chip, the capacitance ratio of a row clock to its buffer is 40. The resulting worst case rise/fall times, with $V_{dd} = 2V$, are about 15n sec. The flip-flops experience no erroneous input to output transparent flow through, even in shift registers with no inter bit logic.

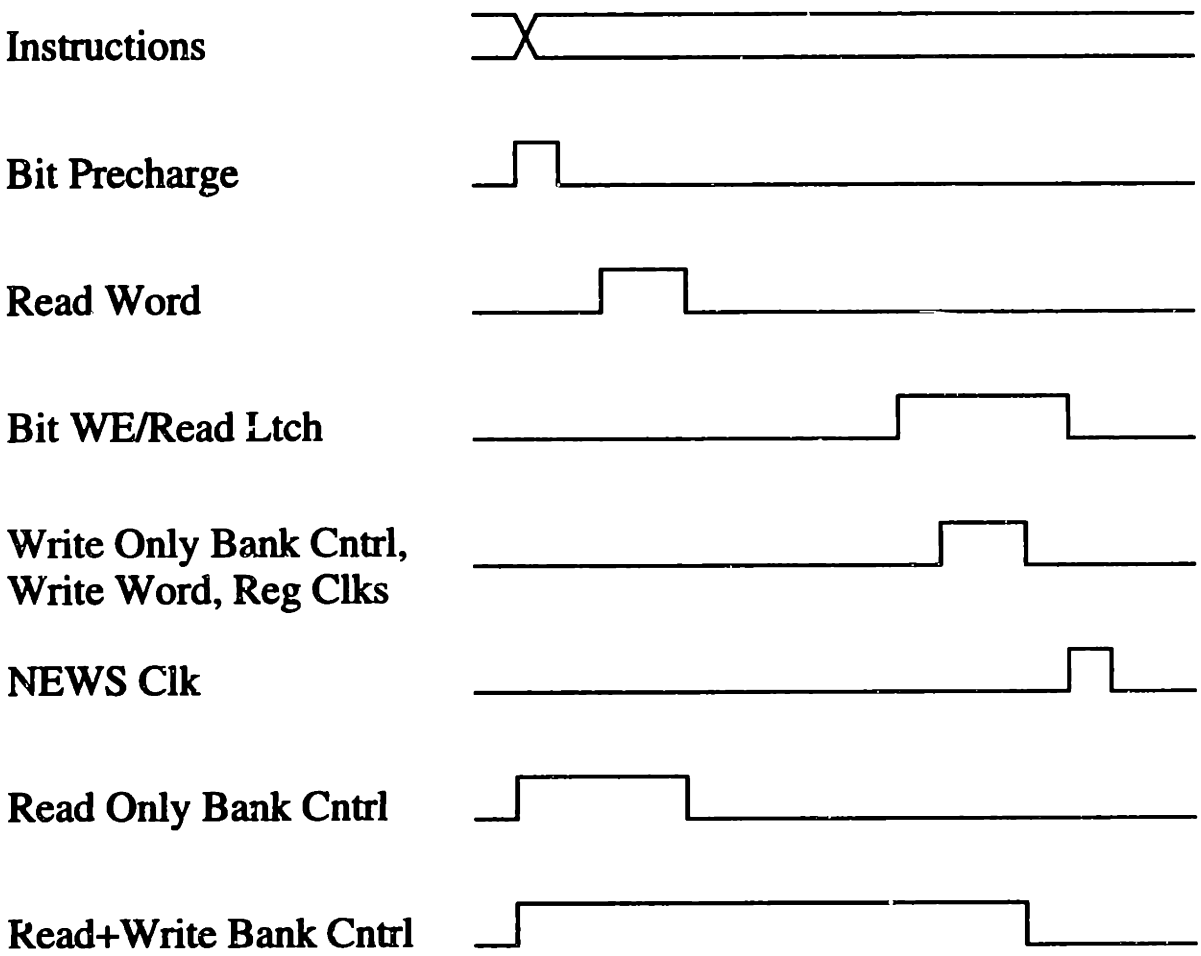


Figure 5.10: PE timing diagram.

5.3.6 PE Timing

The full PE timing diagram is shown in Figure 5.10. The non-timing instruction signals, such as shift, inversion, and ALU controls, are switched at the beginning of a cycle. Most of the memory timing controls, precharge, read, write, and read latch/write enable, have already been discussed. Figure 5.10 also shows the bit line segment pass controls. One of 3 possible timings are used, depending on whether a read and/or write access is occurring past the segmentation point. The read only pulse spans both the precharge and read phases

of the cycle. The write only pulse encloses the write word line timing. The purpose of the joint read and write option is to avoid 2 unnecessary edges when both memory accesses reach the back bit line segments. This optimization is a trivial matter for the sequencer, although, because of the low αs, it is a minor benefit.

Most of the edge triggered PE registers, ALU, conditional, etc ..., use the same timing as the write word lines. That is, registers are loaded at the start of the write pulse. To limit the complexity and width of microcode instructions, the ALU result can only be written to one destination per cycle, which can be one of the registers, or memory.

The NEWS register is an exception because it can be independently loaded from the NEWS network on the same cycle the ALU result is written to a different destination. The delayed timing of the NEWS clock is necessary to avoid a race condition. The danger would be a slightly fast loading and overwriting of the NEWS register from the network, coupled with a slow loading of the ALU destination with a result which was a function of the previous NEWS register contents. Note that, no possible race condition exists between the NEWS clock and the release of the bit line read latch. If the NEWS register is loading from the network, the new value cannot depend on the ALU or the memory read value. If it is loading the ALU result (possibly a function of a memory read), the memory cannot also be executing a write operation on the same cycle, and the write enable/read latch would not be switched that cycle in the first place.

The special treatment of the NEWS timing allows parallel PE computation and data movement, with efficient, same cycle unload and compute of the arriving values. This is important for throughput, especially with motion compensation, when a lot of data is

being moved, but not involved in extensive computations. (In the test chip, this feature eliminates the cycle cost of time multiplexing the NEWS network wires.)

5.4 Bus Segmentation

PE bit line segmentation is enabled by the access frequencies to different memory elements. A similar power optimization, implemented for the pixel and EZW networks, is enabled by the sequential access patterns. Figure 5.11 and the following description is for the pixel load network. The EZW network uses similar structures.

The pixel network backbone shown in Figure 5.11 performs the 1st dimension of serial to parallel conversion. Without the data bus and clock segmentation, the backbone would consist of a 1 bit shift register for load enables. Each enable gates its respective 8 bit pixel latch. Effectively, each cycle, a token progresses around the enable shift register to mark which column position is latching the current pixel. After an entire row is collected, the sequencer clocks the row into the next PE row holding registers, and the enable token rotates around to the beginning. (Reset logic which synchronizes the starting position of the token to the frame start signal is not shown.)

Note the power and area advantages of the pixel data bus, instead of a row long, 8 bit wide shift register:

- The 8 bit data are loaded into latches instead of full, edge triggered flip-flops.
- The clock, with 2 edges/cycle, only drives 1 register per column, instead of 8. Enables to the 8 data latches are only toggled once per pixel row (128 cycles for the test chip).

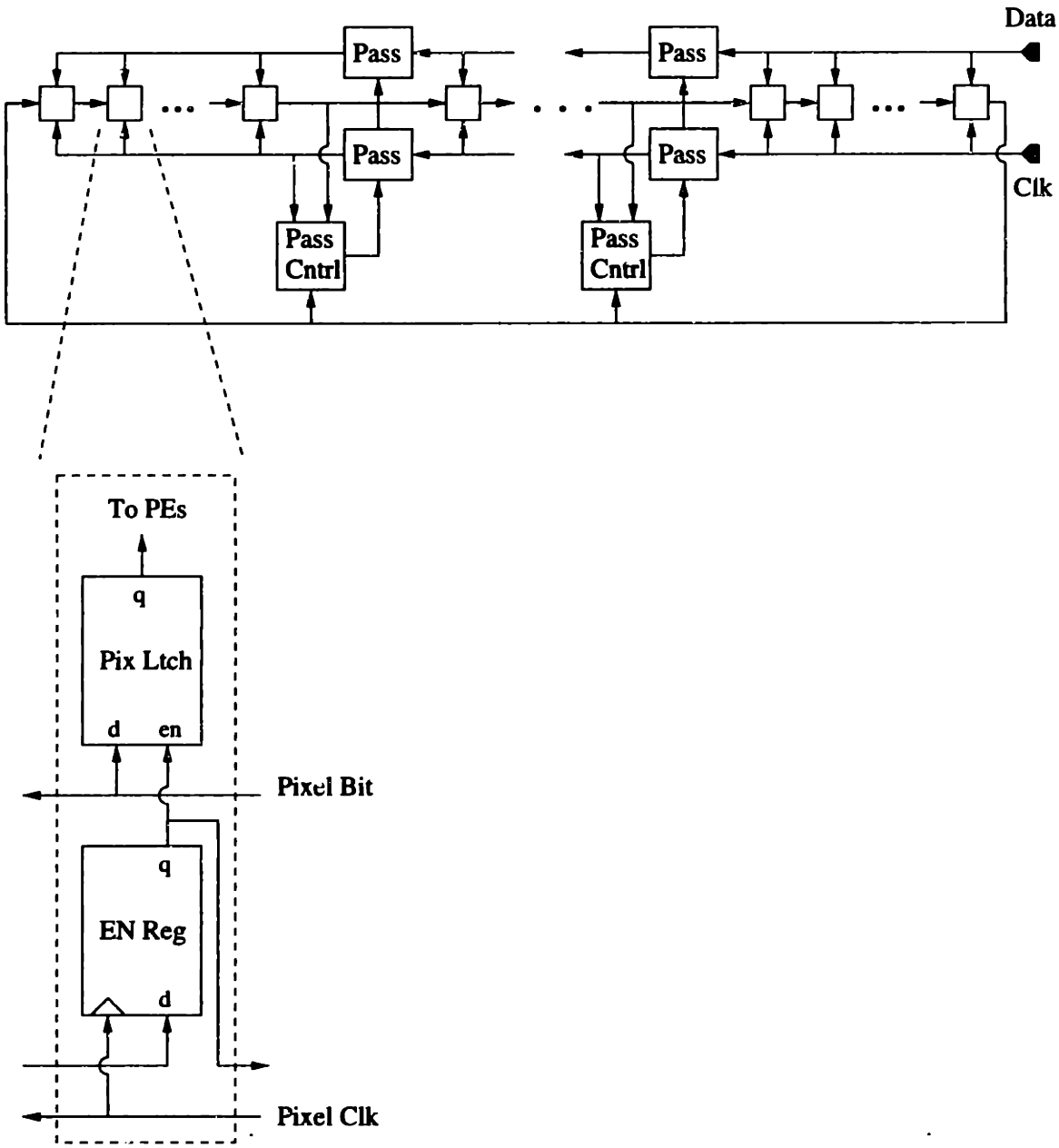


Figure 5.11: Segmentation of pixel load network. One of the 8 data wires is shown.

- Pixels are switched across wire length spanning a row, lightly loaded with diffusion capacitance from latch inputs. This is compared to the same total wire length, plus all the internal flip-flop node capacitances.

The data and clock wire transmission gates (could also be implemented as AND gates) take advantage of the sequential access pattern in order to save power. Segmenting the wires limits the spatial extent to which each wire is exposed to signal switching, thereby lowering switched capacitance. After pixels to the left of a segmentation point have been loaded, no accesses are made past that point until the next row. Neither pixel data or clock need to be delivered for the rest of the current row, and the pass gates block that switching. The controlling FSMs simply watch for the column load enable token to pass their position, and then turn off. Because the control FSMs take their own clock from the local surroundings (which are isolated by the end of a row), the returning token from the last column must be used as an asynchronous reset to force all the segmentation points back on.

In the test chip, both pixel and EZW network backbones are segmented this way into 4 pieces. This reduces both data and clock power to $\frac{5}{8}$. As shown in equation 5.1, the benefit quickly saturates with the number of segments. For example, 8 segments would give $\frac{9}{16}$ of the original power. The limit is a factor of 2 in savings (not really, since equation 5.1 does not count the cost of the segmentation circuits, which is no longer negligible for large numbers of them).

$$\frac{Power}{Unsegmented\ Power} = \frac{n+1}{2n} \tag{5.1}$$

where n is the number of segments.

As already described, the vertical dimension of both networks is cut in two by the backbone trench. This gives a full factor of 2 savings for the vertical wires. (The topology is different, since the neither half is after the other, and equation 5.1 is not followed in this case.) The vertical wires are not segmented further in the same way as the backbones. The presence of the pass gates in the PE array would result in somewhat awkward layout, and the gain would be smaller. For example, clock power per pixel is 128th that in the backbone because of the 1st dimension of parallel conversion already performed.

5.5 Arithmetic Coder

No interesting circuits are used in the arithmetic coder. It is entirely constructed from vanilla standard cells, albeit with near minimum device sizing, as described above.

5.6 Controller

The bulk of the controller and communication networks are also implemented with the scaled down cell library, except the segmentation of the data busses and clocks, as described above.

5.6.1 Word Line Up Conversion

One exception in the controller is the voltage up converter used to produce and buffer PE write word lines and the bit line segment controls. Rather than using a charge pump to produce a higher voltage, a separate supply, V_{ww}, is brought on chip for this purpose. It is

safe to assume that such a supply is available without further system cost, because the exact voltage is not critical, and the supply imposes no special filtering requirements or noise tolerances. It is only important that the voltage is V_{TN} higher than V_{dd}. The αs of both word lines and pass controls are low enough to make the power inefficiency of too high a V_{ww} unimportant. (It should be remembered that many intermediate PE results are shuffled to and from the NEWS register cache, not memory.) Meanwhile, V_{dd} would typically be the smallest voltage produced by DC/DC conversion, as enabled by computational parallelism and the modest requirements of digital circuits. Therefore, it is likely that some other system supply already exists which is adequately higher. If the only higher voltage supplies used are noise sensitive, clean analog ones, for which bursty, high frequency injection from word line switching would be bothersome, the plain battery voltage can be used.

The up converter used is shown in Figure 5.12. This up converter has two small advantages over the common differential amplifier design. No process sensitive ratioed circuits are used, and no overlap current is lost from input switching to output settling. This is achieved straightforwardly by the precharged, self resetting action of the 3 inverter feedback loop. After any input and output transition, the converter precharges itself in preparation for the opposite transition. The lower swing input and its complement are only responsible for controlling N-channel only pulldown devices. (The common reference to V_{dd} and V_{ww} is ground.) The self timed action is demonstrated with the simulated waveforms in Figure 5.13. The converter requires no special circuitry to impose a reset state, since the feedback will always take the converter to a state consistent with its current inputs.

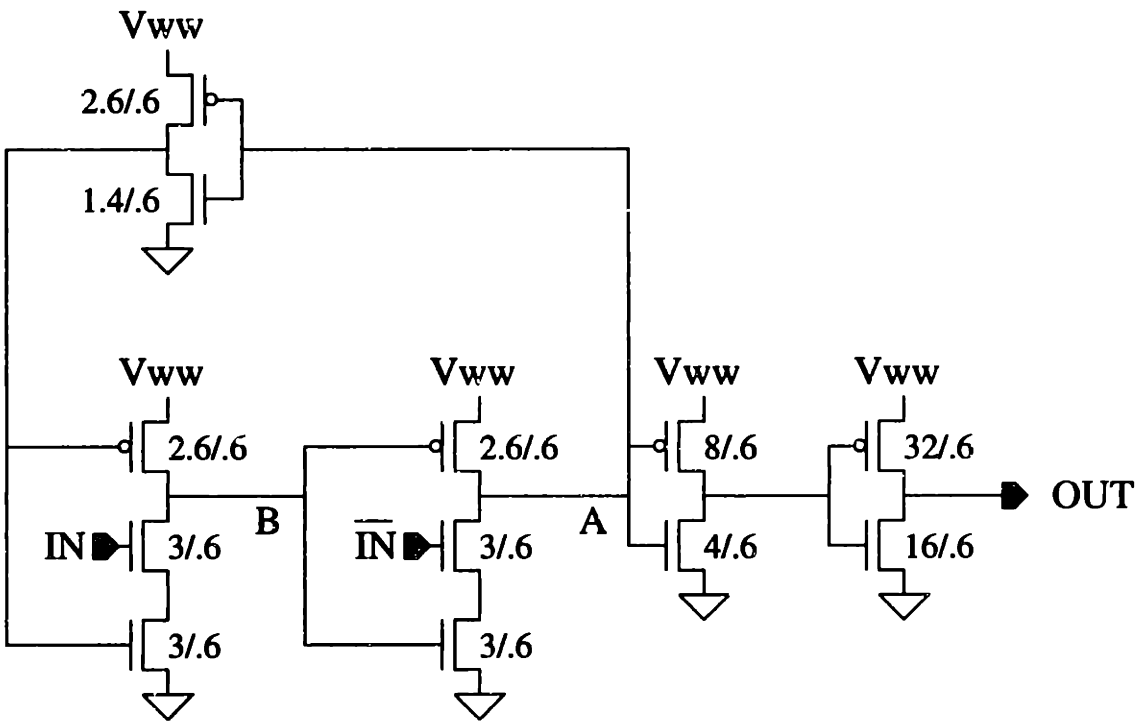


Figure 5.12: Low power word line voltage up converter and buffer. Self resetting up converter avoids ratioed circuits and overlap current. *IN* and $\overline{IN}$ are smaller swing signals, from V_{dd}. *OUT* is driven from V_{ww}, the separate, higher voltage word line supply. (Some weak anti-leakage keeper device are not shown.)

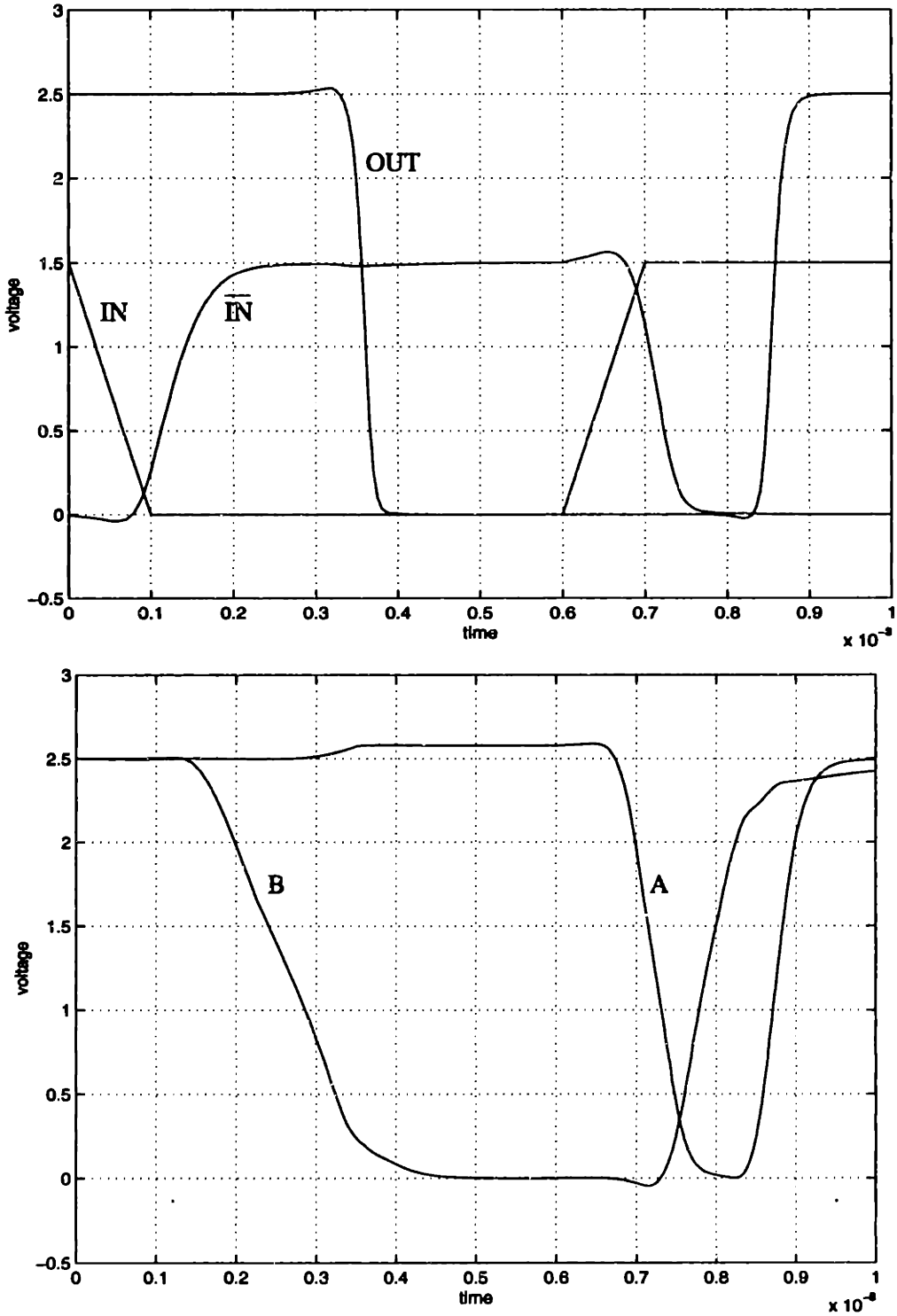


Figure 5.13: Voltage up converter waveforms.

5.7 Power Estimation

The power dissipation of the test chip is estimated in a strictly back of the envelope fashion, although the backs of several sheets of paper are required to do the accounting. The information used in the estimation is the dimensions of the PE, device sizing, process worst case wire and device parasitics, and hand tabulated statistics of microcode operations. As shown in Chapter 7, these estimates are surprisingly accurate. Such simple and effective modeling is possible because of the uniformity of the SIMD array, and the short microcode used. As such, this architecture represents a fairly unusual case. This was taken advantage of throughout the design to estimate where to invest attention, before the completion of simulation ready designs.

Table 5.1 gives the chip peak power estimates by category, assuming V_{dd} = 1.5V and V_{ww} = 2.5V. In addition, an indication of the scaling of each component with image resolution is given. This assumes that the process technology is not improved, but more area is added by multiplying the number of PEs in the SIMD array.

The PE bit line and datapath power scale linearly with the total resolution (and number of PEs) for obvious reasons. Somewhat less obvious is the linear scaling of the instruction distribution power. It is clear that the amount of wiring and device loads grows linearly, but it must also be kept in mind that the cost per wire doesn't really change. The wire *RC* time constants are more than 1 order of magnitude lower than the chip rise/fall times, so there is no bottle neck there. Further, with load to row buffer capacitance ratios of 40, even for PE clocks as described above, there is no imminent explosion of buffering power to keep up with load increases.

Component	Power (μW)	% of Total	Scaling
SIMD instruction distribution	220	38	linear
PE bit line	30	5	linear
PE datapath and NEWS	180	32	linear
Pixel load backbone	12	2	quadratic
Pixel load vertical	10	2	quadratic
EZW unload backbone	60	10	quadratic
EZW unload vertical	30	5	quadratic
Arithmetic coder	0.1	0	quadratic
Misc. Sequencer	30	5	constant
Total	572		

Table 5.1
Chip peak power estimate, broken down by category. V_{dd} = 1.5V The scaling column indicates the hypothetical growth rate with increased video resolution (spatial), assuming unlimited area in the same process technology.

The pixel load and EZW unload networks scale quadratically for obvious reasons (linearly more data, linearly longer distances). The miscellaneous sequencer circuitry includes items such as program counter, DRAM refresh counters, microcode decode logic, etc ..., which do not change with the number of PEs. Nor does the sequencer power change with the clock frequency, which is most conveniently set to the pixel load rate. Because of the pervasive use of gated clocks, the sequencer power is a function of the number of microcode instructions executed (plus the number of DRAM refreshes scheduled), not the incoming system clock cycles.

The arithmetic coder power also scales quadratically. There are linearly more data, and

the processing of that data would have to be carried out at higher operating voltage, to keep up with throughput requirements. On the test chip, the arithmetic coder is operated from V_{dd}. An obvious step up is V_{ww}, which does not require another output from the power supply. The choice of V_{dd} on the test chip was a mistake on the part of the author. The power of the coder is negligible because it is tiny, and almost no capacitance is switched. (The coder consists of a few 11 bit registers, an adder, incrementer, and a few muxes.) However, because the coder operates serially at a frequency 10 times higher than the controller and SIMD array, it turns out to be the bottleneck on voltage scaling. The lowest operating voltage that chip functionality was verified at (1.5V, see Section 7.1) is an artifact of this mistake.

Chapter 6

Implementation

6.1 Process Technology

The vital statistics of the test chip process technology are given in Table 6.1.

V_T	0.7 - 0.9 V
Min. ch. len.	0.6μ
Min. metal pitch	1.8μ
Min. contact pitch	2.4μ
Min. via pitch	2.5μ
Well/diff. spacing	1.8μ
Metal layers	3
Silicide	YES
Local interconnect	NO
Butting contacts	NO
Stacked vias	NO

Table 6.1
Process technology parameters.

6.2 PE

The name of the game inside the PE is bit slice pitch matching. All efforts are made to keep both logic and memory pitch matched. The PE architecture largely avoids inter bit communication. Most wires travel strictly within a bit slice, or to the neighboring bit position (adder and shifters). The exceptions are the special EZW logic and the conditional register, which account for a small part of the PE. Pitch matching therefore benefits layout convenience, area efficiency, and power (no extra wiring to line things up).

6.2.1 Bit Slice Pitch

The PE bit slice pitch is strategically chosen to fit a NEWS register flip-flop and an associated 2:1 mux (used to choose input between NEWS network and ALU result). This choice is a careful balance between the different cells used in the PE, with the dual goals of not awkwardly breaking up local circuit blocks and not leaving sparsely utilized spaces elsewhere. The resulting layout for the NEWS flip-flop and mux is shown in Figure 6.1. The horizontal pitch determined from this cell is 22.4μ. All other cells, some examples of which are shown below, are forced to conform to this pitch. In the case of slightly smaller bit slices, an effort is made to absorb convenient chunks from larger neighbors. In the case of irregular circuitry, such as the special EZW state logic, EZW holding register, conditional register, and global OR network, a large hammer was used to pound the layout until it fit somewhere.

Figure 6.1: Layout of the NEWS register and an associated 2:1 mux determine the horizontal bit slice pitch used throughout the PE (22.4μ).

6.2.2 DRAM

Pitch matching the DRAM cells to the logic bit slices is very difficult. The DRAM cell area is unfortunately large because of the logic only process used, however, spreading out that area over the 22.4μ pitch still results in an absurd aspect ratio. To alleviate that problem, cells are laid out side by side in pairs, straddling the bit line, as shown in Figure 6.2 (on its side). The 4 word lines required for the pair are interleaved as shown, to ease layout constraints. Pairs abut both vertically (after flipping) and horizontally, so that the 2 ground

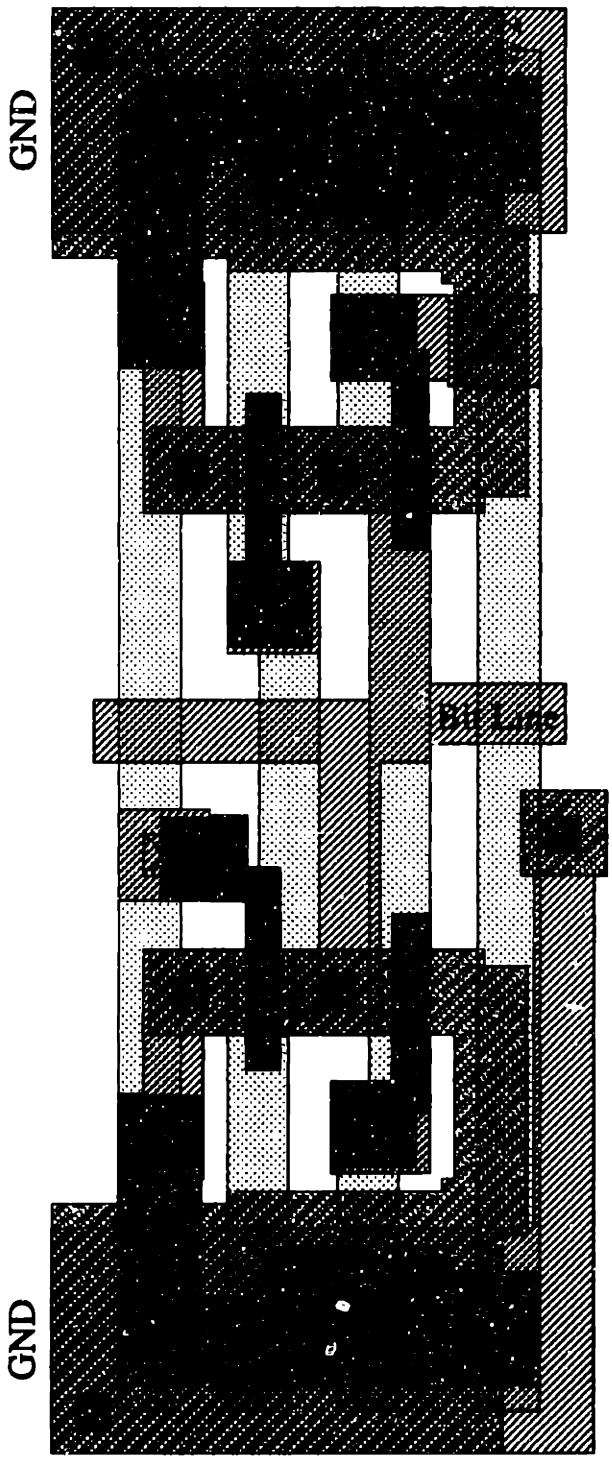


Figure 6.2: Layout of DRAM cell pair, shown on its side. Total area of 2 cells is 22.4μ by 7.7μ. (Note the large area compared to real DRAM cells, which might typically be 1μ^2 each.)

contacts and the substrate plug are shared with neighbors.

This results in reasonable cell aspect ratios, and allows optimal area efficiency. The horizontal dimension for the pair is matched to the logic pitch, while the vertical dimension is limited by the minimum metal pitch of the 4 word lines. Note that only 1 via spacing is used for each 4 word line group. The 4 vias required per cell pair are horizontally separated enough to allow the river routing of word lines around them.

Note that, no electrical problems result from the close spacing of read and write word lines, which are operated from different supplies (V_{dd} and V_{ww}, respectively). First, there is no forward biased diode/latchup danger, both because there are no N-wells in sight, and the word lines touch only device gates, not diffusion. Second, parasitic estimates and circuit simulation were used to verify that the lower swing read word lines cannot be dangerously coupled into from the neighboring higher swing write word lines.

6.2.3 Bit Line Interleaving

An obvious alternative to the cell pairing scheme used to ease pitch matching is to interleave bit lines from vertically adjacent PEs, as shown in Figure 6.3. In fact, the more compelling advantage of bit line interleaving, compared to just a pitch matching aid, is the possibility of reducing the number of word line wires. If cells from the two bit lines which have the same address can be placed next to each other, their word lines wires can be merged. This might save area and some power (half the wiring capacitance, but the same device loading).

This scheme is not used on the test chip because the pairing scheme already reduces cell

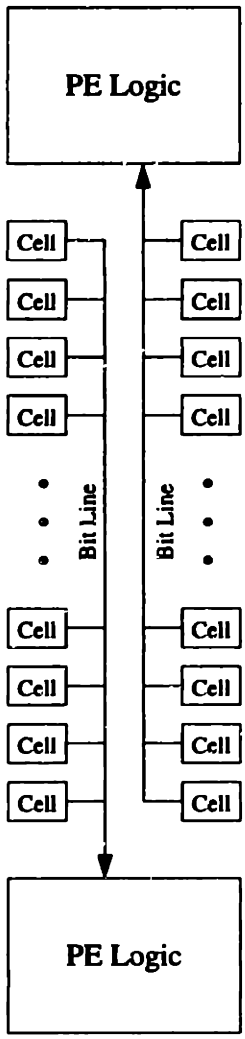


Figure 6.3: Bit line interleaving to aid memory/logic pitch matching.

aspect ratios enough to enable efficient packing of total cell areas into the layout. Further, and unfortunately, this scheme is fundamentally incompatible with bit line segmentation, as shown in Figure 6.4. Corresponding address cells cannot be lined up because of the preferred position of frame buffers with respect to their PEs. For example, each current frame buffer should be placed near its own PE, so same address cells end up on opposite vertical ends of the layout. This presents problems for an encoder implementation in a

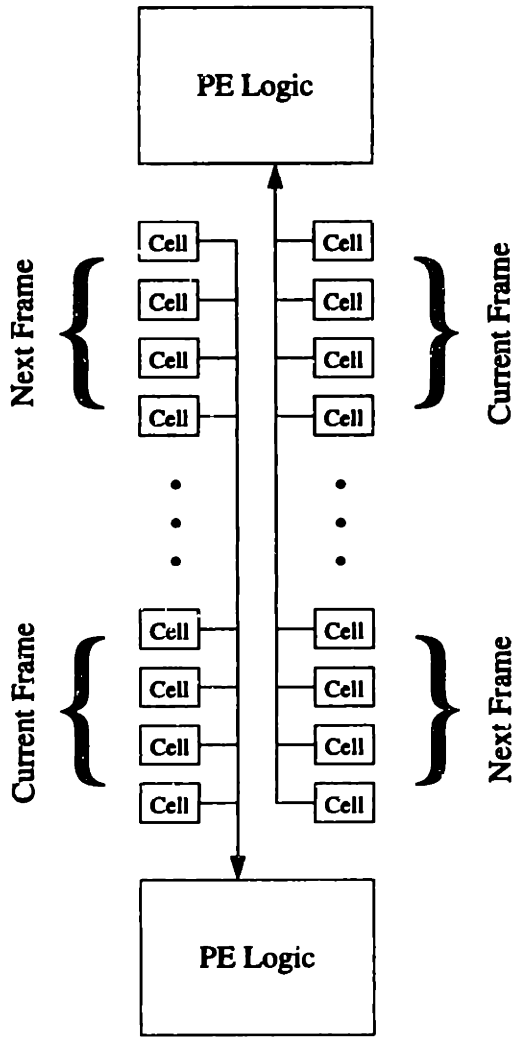


Figure 6.4: Mismatch in word line address positions due to bit line segmentation.

real DRAM process, which would decimate cell area and place much more pressure on the reduction of word line wires. In that case, bit line interleaving and segmentation would be in contention, and some tradeoff would have to be worked out. It should be kept in mind that bit line segmentation represents more power savings than half the word line wiring, and that the segmentation provides the convenient pixel load port into the PE memories.

6.2.4 Bit Line Segmentation

Figure 6.5 shows the small layout impact of bit line segmentation. A pass gate and its control wire is shown sandwiched between two adjacent DRAM cell pairs. The total layout cost is one extra via pitch along the length of the bit line.

6.2.5 Adder

The layout of a PE adder bit slice is shown in Figure 6.6. The adder is too large to fit in the prescribed pitch, so it is broken up into 2 rows of logic (each row is a strip of N and P-channel devices). After the division, some extra space is available. This is used to absorb 2 muxes which select the ALU output.

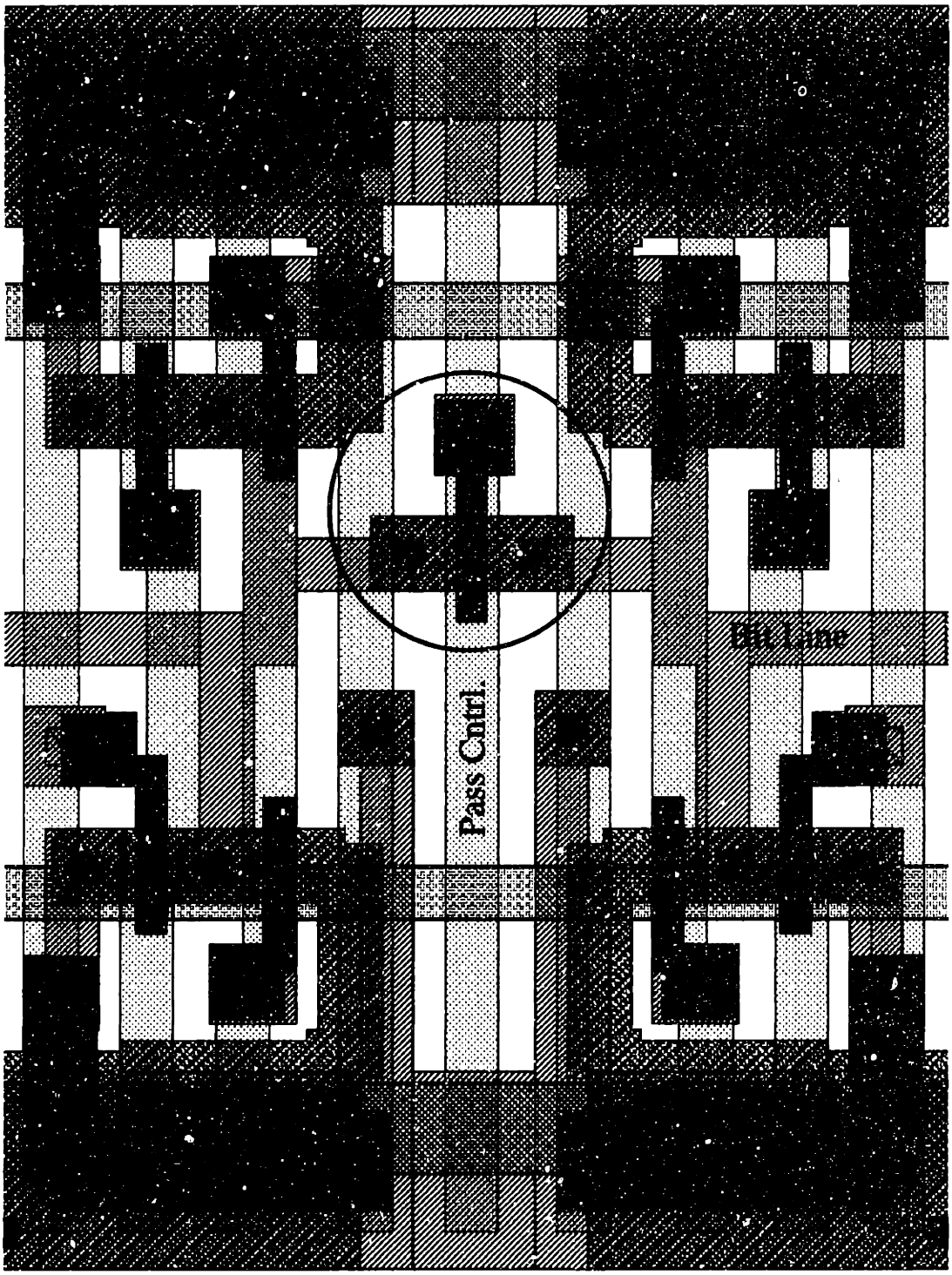


Figure 6.5: Bit line segment pass gate and control, shown on its side. Pass gate is circled, and the labeled control wire fits between neighboring word lines at minimum via pitch.

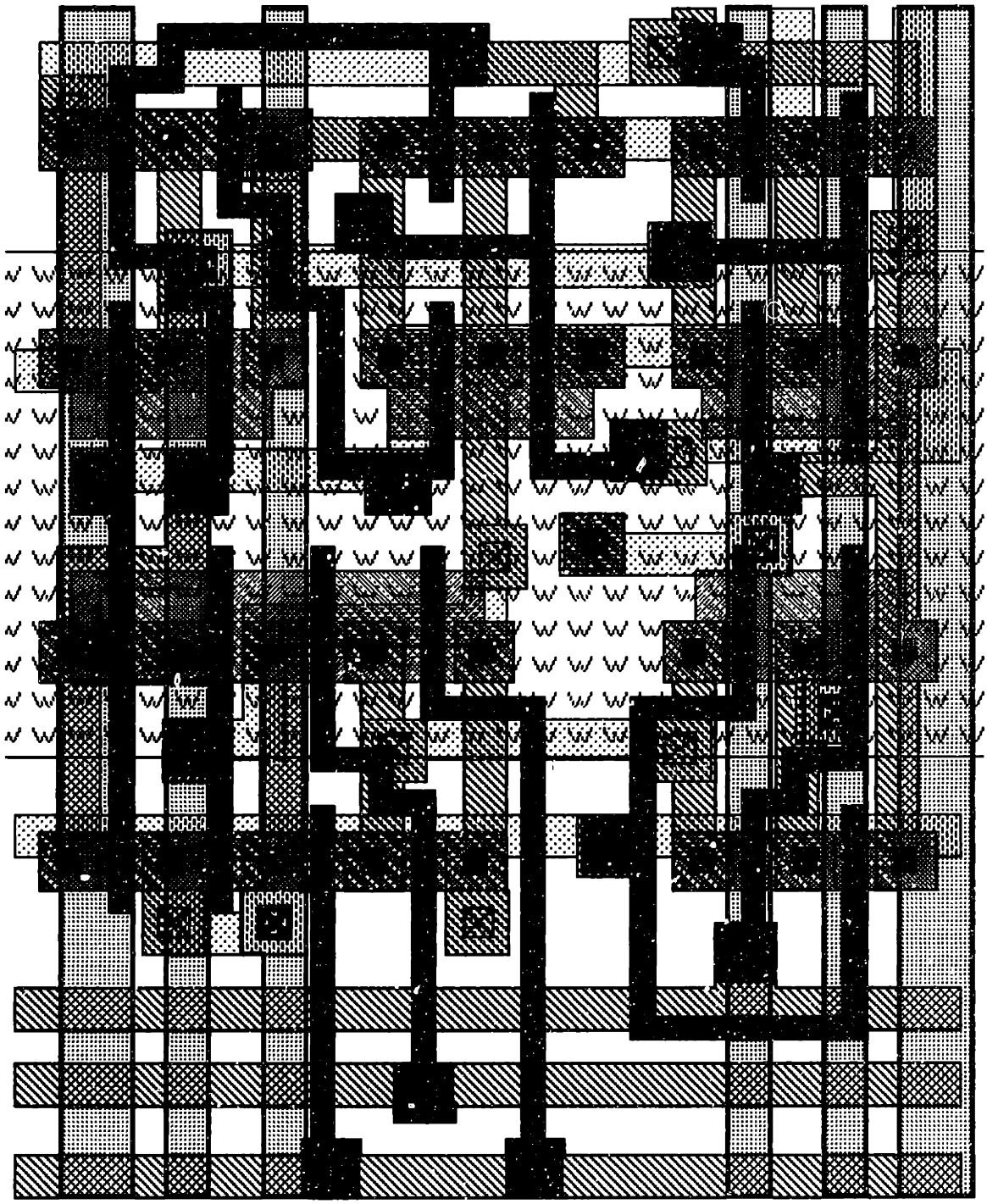


Figure 6.6: Layout of PE adder including 2 muxes which select the ALU output.

Function	Height (μ)	% Area
Memory	190	57
Pixel Load	13	4
Bit Line Peripherals and Addr ROM	15	4.5
NEWS Reg + Net	37	11
ALU Input Inv <<, >>, concat	15	4.5
Adder	30	9
EZW Reg, Logic Conditional Reg Global OR Net	17	5
ALU Reg, >>	16	5

Table 6.2
PE area usage by category. (All widths are the PE bit slice pitch - 22.4μ.)

6.2.6 Area Usage

The PE occupies an area of 268.8μ by 330.9μ, and includes almost 3000 transistors (1752 memory, 1234 logic). Figure 6.7 shows the layout of 1 PE (just metal 1 and metal 3). Table 6.2 gives statistics of how the vertical height is apportioned among the PE functions. The PEs are designed to be directly abutted both vertically (after flipping) and horizontally. The intervening trench in the middle of the chip compensates for the interruption with feed through wiring that completes appropriate connections.

The coupling of this tiny granularity and the global nature of the SIMD array (instruction distribution), may seem like an awkward combination of the worst of two domains. On one hand, the price is being paid for global distribution of centrally decoded instructions. On the other, the tiny PEs seem like an inefficient payoff (compare the register files to large commercial DRAM subarrays). Typically, low power design techniques try to min-

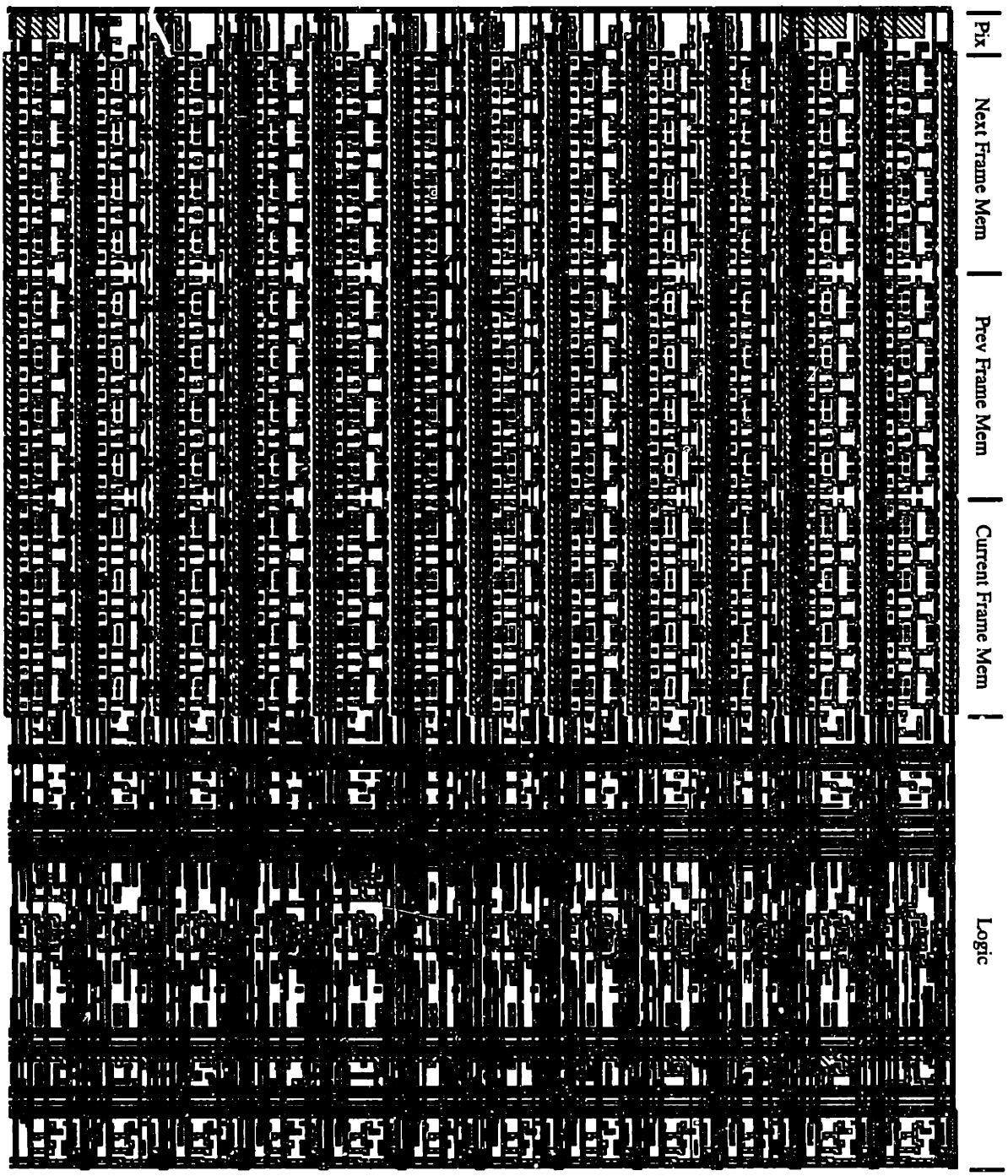


Figure 6.7: PE layout (just metal 1 and metal 3). Area is 268.8μ X 330.9μ.

imize global wiring in favor of local control and computation.

However, the layout of the PEs makes this a very different case. As already discussed, aggressive custom layout makes individual bit slice cells very efficient in terms of wiring per useful logic area. Within each PE, bit slices efficiently abut to form the 12 bit wide datapaths, and this abutment carries over to the inter PE boundaries. In fact, it becomes very difficult to distinguish PE boundaries in the assembled arrays. The memories effectively look (and act) like one wide subarray (384 bits) with large parallel throughput, all of which is used locally to each bit line. The logic elements follow the same pattern of efficiency. Macroscopically, each PE row looks as though it contains a 384 bit ALU. One has to look carefully to notice that the carry chain happens to be interrupted every 12 bits. The row instruction wires and word lines are thus very densely loaded and experience no wasted overhead. The central instruction decoding, sequencing, and timing, is therefore efficiently amortized over a large number of small consumers.

6.3 Sequencer and Arithmetic Coder

Both the arithmetic coder and sequencer contain little logic. However, as hinted at in the previous chapter, the layout of both is of some concern. Each component is placed next to the uniform SIMD array, which leaves no nooks or crannies to absorb the peripheral circuitry. To minimize die area, an effort was made to flatten out the layouts against the sides of the PE array. This was done by hand placement of the library cells used. The resulting dimensions are 160μ by 4170μ for the sequencer, and 140μ by 2070μ for the arithmetic coder.

The area of the sequencer does not include the vertical bus which distributes decoded

instructions and word lines to PE rows, or the row buffers which drive the horizontal distribution. The buffering of instructions is split into 2 stages (as opposed to just shorting the row wires to the vertical bus) to avoid unacceptable *RC* time constants on the vertical wires. The row buffers are staggered 4 deep to pitch match their outputs to the row instruction wires, and are placed underneath the vertical distribution bus. Both row wires (metal 2) and the vertical bus (metal 3) are at minimum via pitch (some care was required to avoid collisions routing both buffer inputs and outputs, given the overhead bus). The row buffers and vertical bus occupy an additional 185μ wide strip, running the height of the SIMD array.

6.4 Microcode Memory

For actual use, the microcode memory could be implemented as a small ROM in the sequencer. For debugging purposes, the test chip relies on external EPROMs. An 11 bit program address is output, and a 20 bit instruction word is input. This external memory is not counted in the chip power dissipation. However, due to the small size, 20 bits by 1.5K words, and low throughput, 3.3K instructions per frame, the unaccounted power of the missing on chip ROM would be negligible (a fraction of 1%).

6.5 I/O

Similarly, the chip I/O is designed with debugging in mind. No attempt is made to predict system integration or the form of inter component communication. Instead, the chip I/Os are designed to interface to 5V off the shelf parts used for testing. Voltage up conversion

Function	Transistors	% Area
SIMD Array	3.06M	79
Trench	6.2K	1.2
Inst Row Buffers and Vert Wires	3.7K	1.7
Arithmetic Coder	3.5K	0.25
Sequencer	6.4K	0.58
IO	1K	15.5
Total	3.08M	98.23

Table 6.3
Test chip transistor and area statistics by category. Transistor numbers are for logical devices. Parallel layout fingers are not counted separately. Total chip area is 9980μ by 11580μ.

is performed in output pads with the same circuit used on chip for DRAM word lines. A third, separate, 5V supply, V_{hh}, is used to drive external pins. Power drawn from V_{hh} is not counted in the chip dissipation. (Not only would a real implementation not use 5V, and possibly be integrated with other components, most of the test chip I/Os, such as external EPROM address and data, would not even exist.)

6.6 Chip Statistics

Table 6.3 gives some overall chip statistics. Note that between the uniformity of the SIMD array and the flattened coder and sequencer layouts, useful area utilization is higher than 98%. The remaining 1.7% (above and below the coder and sequencer, inside the corners of the pad ring) is filled with V_{dd} and V_{ww} to *GND* bypass capacitance, which cannot hurt, given the bursty nature of the currents drawn from these supplies.

The bypass capacitance is made from N-channel device gate oxide. An example cell is shown in Figure 6.8. Channel length is kept short to minimize the series resistance. On

the other hand, the cross hatched poly pattern is used to maximize the fill ratio, which determines the density of capacitance obtained. Sources, drains, and substrate contacts are shorted to *GND* in continuous metal 1, and gates are regularly strapped to V_{dd} or V_{ww} with metal 2 running overhead.

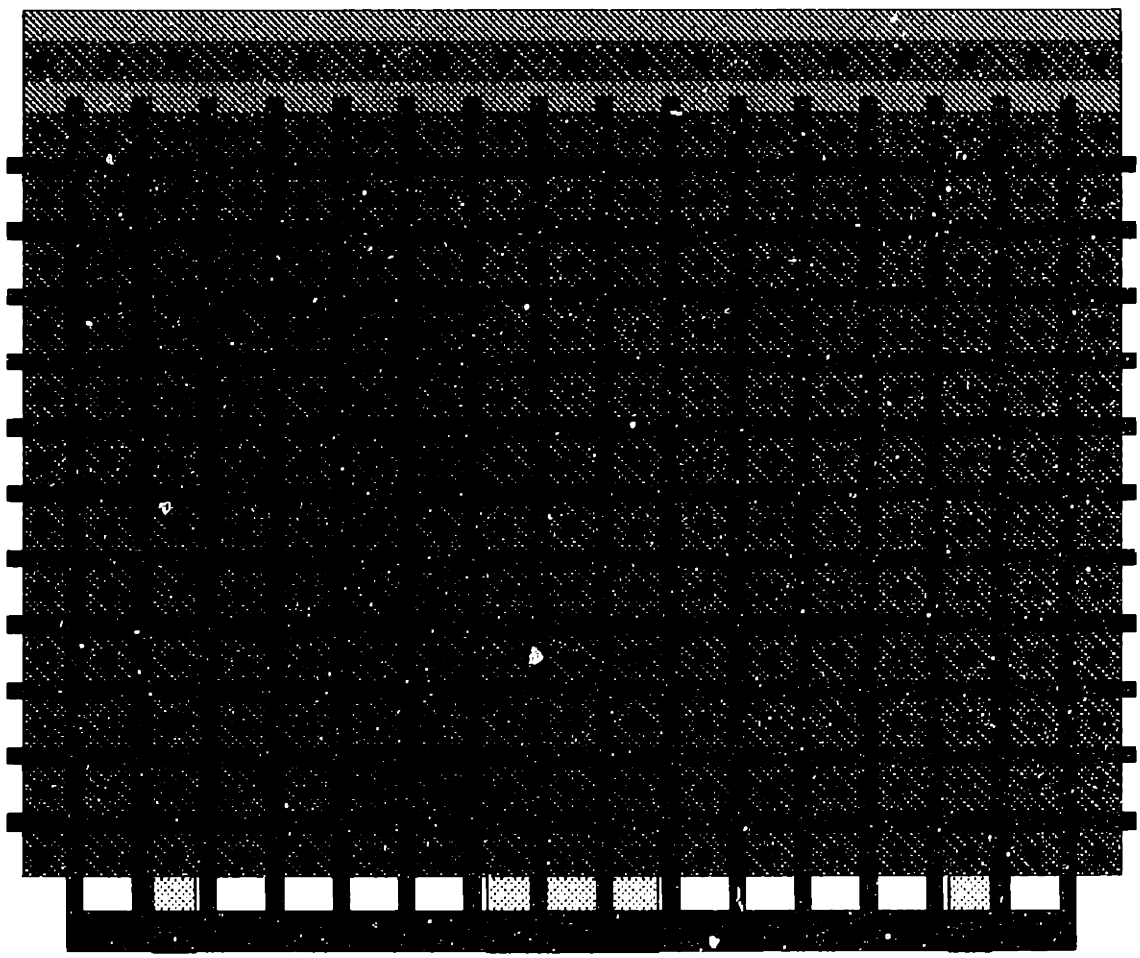


Figure 6.8: Layout of bypass capacitor cell used to fill up unused area.

Chapter 7

Testing and Results

7.1 Functional Verification

Test chip functionality was verified by performing a binary comparison of the output stream with that of the software emulator. Binary comparison was considered more reliable than mere visual inspection of the decoded output, since subtle errors cannot be obscured by small or imperceptible visual effects.

Chip and emulator output streams were compared for entire 16 frame groups spanning lead and differential frames, after reset, across 16 frame group boundaries, and for both motion and no motion cases.

Note that, the no motion test cases do not refer to the kind of system wide power optimizations discussed in Chapter 2 for prolonged idle periods. In these tests, the encoder is operated in isolation, without interactions with matching data converter, imager, or controlling base station. Therefore, no image difference and threshold is performed at the beginning of frames, no record is kept of the number of consecutive idle frames, and no

change is made to frame rates, image resolution, or pixel precision.

Rather, the same code is executed unconditionally each frame, including wavelet filtering, frame differencing, EZW coding, and previous frame updating. In this case, the effect of no motion in the video stream is that the arithmetic coder is exercised less, and some portion of microcode instructions are bypassed each bit plane, after the PEs are polled regarding the presence of significant coefficients. It is evident from the microcode found in Appendix C that a substantial portion of the instructions are still executed each bit plane, especially considering that the executed instructions are more power intensive than those bypassed, on average. Differences in the power cost of instructions are related to the number of instruction wires exercised, memory accesses made, and datapath nodes switched.

In order to keep the test setup simple, the video data is not taken from a camera. Instead, an EPROM is programmed with data from the test images used to report compression and PSNR performance in Chapter 3. The EPROM capacity fits 2 frames which are repeatedly fed to the encoder. Tests with motion content consist of alternating between 2 frames of the sequence, while no motion tests use 2 copies of the same frame.

Chip functionality was verified down to V_{dd} of 1.5V. As noted before, circuit simulation already indicated that the arithmetic coder is the bottleneck to further voltage scaling. This is due to the arithmetic coder's high operating frequency, which is required by its serialized processing. However, the coder's tiny physical capacitance makes the efficiency of its operation largely irrelevant. Unfortunately, the opportunity to take advantage of this, by operating the coder from V_{ww} for example, was missed due to the author's mistake. Note that, the minimum corresponding operating value of V_{ww} was not explored. As expected,

power drawn from V_{ww} is a small fraction of that from V_{dd}, and the word line voltage was simply kept about 1V higher for all tests.

7.2 Power Dissipation

Chip power dissipation was measured with a lot of low pass filtering provided by the measuring apparatus (a Keithley 2000 Multimeter). Low pass filtering is required because of the bursty current spikes caused by suddenly active gated clocks used throughout the chip. The meter was set to perform analog integration of supply currents over $\frac{1}{60}$ of a second (the line frequency), and to report averages digitally computed over 10 sample windows. This gives averaging windows of $\frac{1}{6}$ of a second, or 5 frame periods. This is more than enough to average the bursty current spikes which occur over fractions of 1 frame period.

Table 7.1 shows measured power numbers with V_{dd} = 1.5V, as a function of the number of bit planes encoded. The power drawn from V_{ww} always hovers around 4% of the V_{dd} power, and is not shown separately. This ratio is in line with estimates based on the relative frequency of memory writes in microcode instructions (vs. the sum of other instruction signal activity). Table 7.1 shows power numbers for both motion and no motion cases, with no motion as described above. The normalized bit rates correspond to the motion case (the output bit rate for the no motion case is n bits/frame, n being the number of planes, plus 8 extra image mean bits for the lead frame in every group of 16).

Note the following observations regarding the measured dissipations:

- The bit rate grows faster than linear with the number of planes encoded. However, as

Bit Planes	Normalized Bit Rate	PSNR	Power(μW)	
			Motion	No Motion
4	.0087	33.5	492	488
5	.0204	37.5	540	512
6	.0420	40.9	590	561
7	.0781	43.3	639	615
8	.1429	44.3	675	648

Table 7.1

Power drawn from V_{dd} of 1.5V as a function of the number of encoded bit planes. Both motion and no motion cases are taken from the head and shoulders sequence shown in Figures 3.9 and 3.10. The normalized bit rate (output/input bit rate) and PSNR is for the motion case. The power drawn from V_{ww} of 2.2V always hovers around 4% of the V_{dd} power.

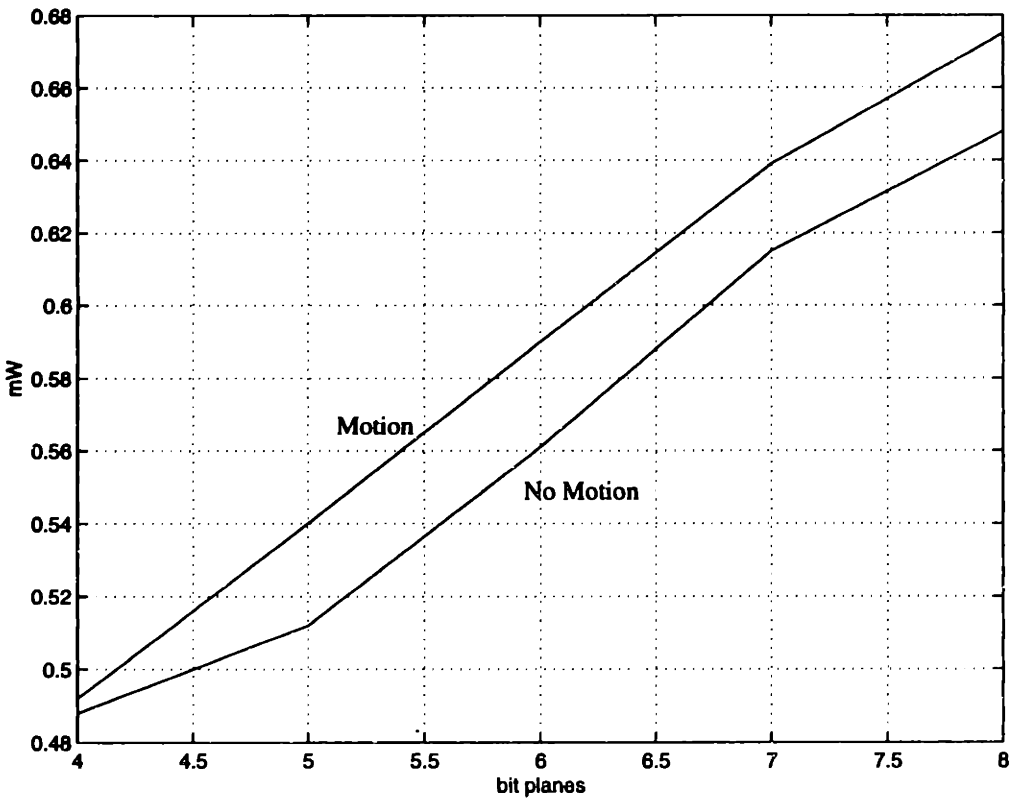


Figure 7.1: Plot of power vs. bit planes for both motion and no motion cases. Both relationships are roughly linear (within measurement accuracy), showing that power is a weak function of bit rate and arithmetic coder activity.

seen clearly in Figure 7.1, the power dissipation does not. This shows that dissipation is a very weak function of the arithmetic coder activity, as predicted from its small physical capacitance. The roughly linear dissipation growth follows the number of SIMD instructions executed, which is a constant for bit planes after the first 2 (the first 2 benefit from extended 3 symbol alphabet regions, as described in Chapter 3).

- The no motion case exhibits somewhat lower dissipation. The difference is in line with estimates based on the number and flavor of bypassed microcode instructions.

- Extrapolating toward no bit planes, the dissipation does not fall off to 0. This is in line with estimates of the constant power costs/frame, which include wavelet filtering, frame differencing, EZW symbol state setup, previous frame updating, DRAM refresh, and pixel loading.

- The peak dissipation of 675μW, for 8 bit planes, compares very well with the 572μW estimate given in Table 5.1.

7.3 Chip Die Photo

The test chip die photo is shown in Figure 7.2. The chip is packaged in a 208 pin PGA. The package was chosen for its cavity size. Few of the pins are used for I/O. The I/O includes: 20 instruction bits, 11 address, 3 serial output, 8 pixel input, 1 clock, 1 reset, 1 start frame signal, 1 arithmetic coder clock speed selector, 1 sequencer clock speed selector. The remaining pins are gratuitously divided among the 4 supply nets (GND, V_{dd}, V_{ww}, V_{hh}).

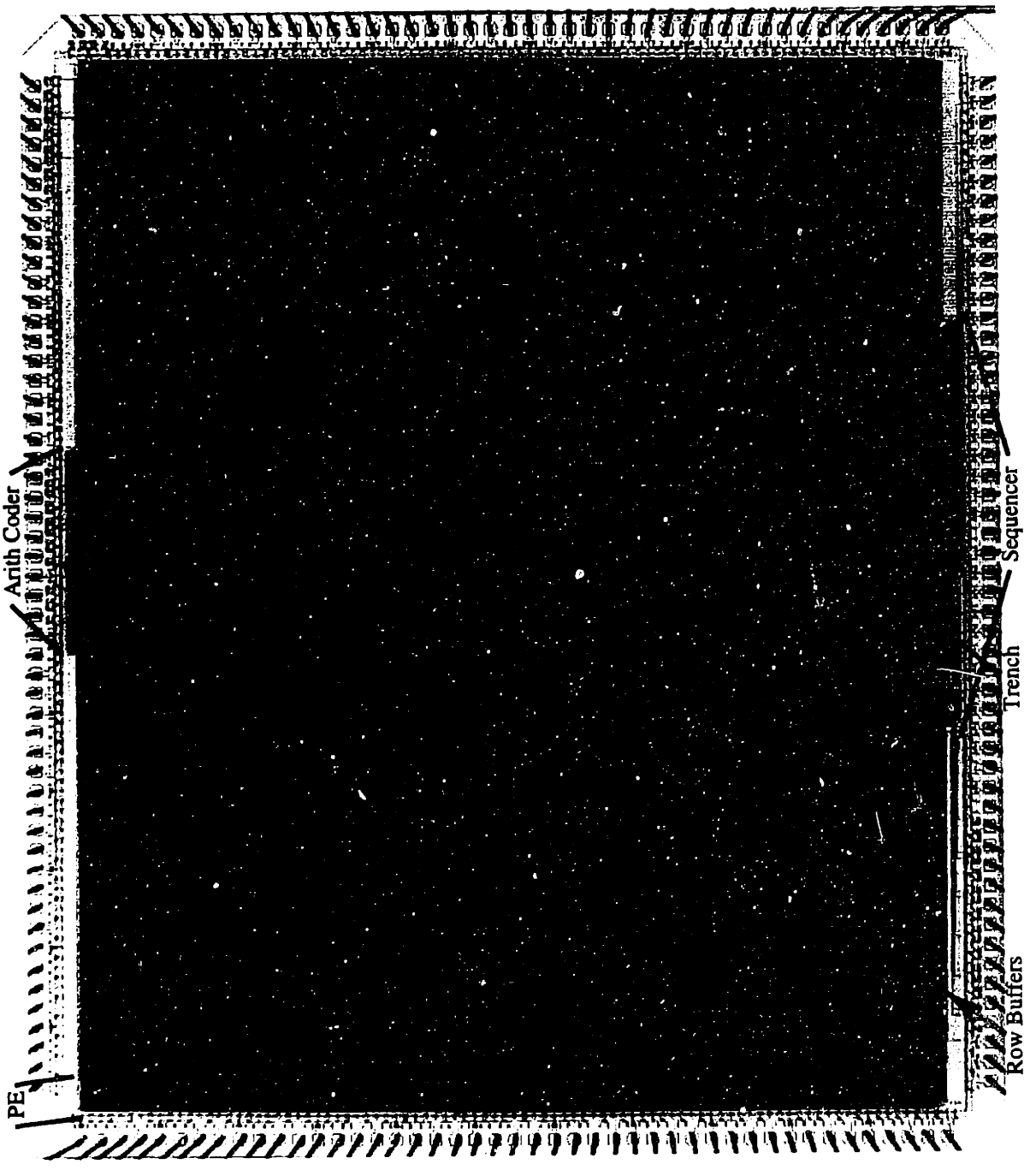


Figure 7.2: Test chip die photo.

Chapter 8

Conclusions

Compared to the known previous work on low powe video compression, the results achieved in this work are fairly surprising. While roughly maintaining the compression performance of a known, competitive algorithm, this work has resulted in a test implementation which integrates the functionality of several other efforts, with total power dissipation levels at least an order of magnitude lower than any single one of them.

This can be attributed to two principal factors. The design approach taken is very broad, meaning that from the outset, the requirements of all necessary components are considered together. This includes: memory capacity, data distribution/loading, motion compensation, filtering/transformation, quantization, symbol coding, and entropy coding.

Second, the design approach taken from the outset was a very deep, vertical one, meaning that considerations at all levels of design were taken into account and allowed to influence each other. These include: system, algorithmic, architectural, circuit, and layout. As such, the design was much more an annealing process than a flow chart. For example, the

author was as likely to make an algorithmic change after being frightened by a transistor schematic, as to devise an efficient circuit for a given job. (Note that the author scares easy - a long wire is usually enough to do the trick.)

The following list summarizes the design ingredients which are used or developed in this work to achieve low power operation:

- The underlying compression algorithm is picked with a laundry list of considerations from other design levels in mind: compression performance and scalability for system level; local communication patterns and modest computational throughput for architectural; simple computational operators and easy parallelization for circuit.

- The algorithm is modified to further localize or reduce communication costs. For example, the elimination of expensive sorting operations cater to architectural demands for local, uniform communication. A myriad of both input and output data reorderings either conform to the architectural SIMD model, or reduce circuit switching activity.

- Algorithmic tradeoffs are made to reduce computational complexity while minimally impacting compression performance. Examples can be seen in the choice of wavelet filters, number of subband decomposition levels used, and the approximation of symbol probabilities used for entropy coding.

- The algorithm is extended to the time dimension without incurring excessive memory requirements. Resynchronization hooks are inserted to deal with high bit error rates at the system level.

- Proposals are made for future work to tailor motion compensation to the SIMD architectural model with the use of limited, global motion vector sets.

- A fine granularity SIMD architectural model is adopted which matches most of the algorithmic and circuit opportunities and needs very well: the data parallelism of the algorithm is utilized with efficient, well amortized instruction distribution costs; total memory bandwidth is huge, with the memory broken into small, power efficient subarrays, whose outputs are always localized to the consumers.

- A trivial one hop NEWS network meets the intra array communication needs, both for the basic frame differencing algorithm and possible extension to motion compensation. However, extra communication networks are superposed to meet algorithmic requirements (serially operating arithmetic coder), and system requirements (evenly spread out activity of imager and data converter).

- PE design takes a minimalist approach which takes into account operation frequencies. For example, rare multiplies used in wavelet filtering are alloted few resources, just the option of 1 position shifting, as opposed to a full barrel shifter or multiplier. The precision of the entire PE is set to barely cover the upper bound algorithmic requirements. On the other hand, much more frequent logical operations involved in symbol coding are hardwired in specialized logic (though the complexity is fairly modest).

- Gated clocks are used extensively throughout the design to take advantage of bursty, unpredictable, and highly data dependent activity. Such bursty activity may be experienced in the SIMD array during arithmetic coder passes, or start of frame syn-

chronization waiting. The arithmetic coder may experience large idle periods due to low output bit rates, especially during periods of no motion.

- The small PE memory size determined at the architectural level presents an unusual circuit design point. This opportunity is taken advantage of by employing large voltage swing on bit lines, which emphasizes bit line peripheral circuit power (like that of bit line sense) over the bit line power.

- The operating efficiency provided by parallelism, locality, and layout efficiency is used to reduce power by several techniques other than the obvious voltage scaling one. These include: pervasive use of small device sizes; minimization of buffer strength used to drive SIMD instruction wires; use of circuits, such as the PE adder, which emphasize low physical capacitance over circuit speed.

- Communication and memory access patterns established by the algorithm and architecture are used at the circuit level in a variety of ways: bit lines are segmented according to access frequency; a free second memory port is made available to pixel loading; DRAM refresh times are extended for given storage node sizes by special treatment of bit lines during extended idle periods; data and clock wires in the load and unload communication networks are segmented to reduce switched capacitance.

- Aggressive custom layout of PEs is used to fulfill the efficiency promise of the fine granularity SIMD architecture.

Bibliography

[1] W. Namgoong, M. Devenport, and T. Meng. A low-power encoder architecture for pyramid vector quantization of 2D subband coefficients. In *VLSI Signal Processing VIII*, pages 391–400, 1995.

[2] B. Gordon and T. Meng. A 1.2 mW video-rate 2D color subband decoder. *IEEE Journal of Solid State Circuits*, 30:1510–1516, December 1995.

[3] A. Chandrakasan et al. A low power chipset for portable multimedia applications. In *IEEE International Solid-State Circuits Conference*, pages 82–83, 1994.

[4] T. Kuroda et al. A .9V 150MHz 10mW 4mm2 2D discrete cosine transform core ... In *IEEE International Solid-State Circuits Conference*, pages 166–167, 1996.

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

[6] G. Yeh, Y. Lu, and J. Burr. A low-power video motion estimation array processor. In *1996 Symposium on VLSI Circuits*, pages 162–163, 1996.

[7] S. Molloy, R. Jain, and K. Nishibori. A video CODEC chipset for wireless multimedia networking. In *VLSI Signal Processing VIII*, pages 381–390, 1995.

[8] M. Harrand et al. A single chip videophone video encoder/decoder. In *IEEE International Solid-State Circuits Conference*, pages 292–293, 1995.

[9] M. Bopp et. al. A DECT transceiver chip set using SiGe technology. In *IEEE Int. Solid-State Circuits Conf., vol 42*, pages 68–69, 1999.

[10] A. Abidi et. al. The future of CMOS wireless transceivers. In *IEEE Int. Solid-State Circuits Conf., vol 40*, pages 118–119, 1997.

[11] M. Perrott, T. Tewksbury, and C. Sodini. A 27 mW CMOS fractional-N synthesizer using digital compensation for 2.5 Mb/s GFSK modulation. *IEEE Journal of Solid State Circuits*, 32:2048–2060, December 1997.

[12] I. Fujimori. A differential passive pixel image sensor. Master's thesis, Massachusetts Institute of Technology, 1997.

[13] O. Schrey et. al. A locally adaptive CMOS image sensor with 90dB dynamic range. In *IEEE Int. Solid-State Circuits Conf., vol 42*, pages 310–311, 1999.

[14] T. Simon. Self communings in the bathroom. Unpublished, Massachusetts Institute of Technology, 1999.

[15] T. Simon. Low power, high speed analog-to-digital converters. Area Examination, Massachusetts Institute of Technology, 1999.

[16] J. Goodman and A. Chandrakasan. A 1Mb/s energy/security scalable encryption processor using adaptive width and supply. In *IEEE Int. Solid-State Circuits Conf., vol 41*, pages 110–111, 1998.

[17] A. Dancy and A. Chandrakasan. A reconfigurable dual output low power digital PWM power converter. In *IEEE/ACM International Symposium on Low-Power Electronics and Design*, pages 191–196, 1998.

[18] A. Dancy and A. Chandrakasan. Ultra low power control circuits for PWM converters. In *IEEE Power Electronics Specialists Conference*, pages 21–27, 1997.

[19] J. Bretz. DC-DC converters with high efficiency over wide load ranges. Master's thesis, Massachusetts Institute of Technology, 1999.

[20] A.N. Netravali and B.G. Haskell. *Digital Pictures: Representation, Compression and Standards, Sec. Ed.* Plenum, 1995.

[21] W. B. Rabiner and A. P. Chandrakasan. Network-driven motion estimation for wireless video terminals. *IEEE Transactions on Circuits and Systems for Video Technology*, 7:644–653, August 1997.

[22] S. Younis. *Asymptotically Zero Energy Computing Using Split-Level Charge Recovery Logic.* PhD thesis, Massachusetts Institute of Technology, 1994.

[23] S. Younis and T. Knight. Practical implementation of charge recovering asymptotically zero power CMOS. In *Symposium on Integrated Systems, MIT Press*, pages 234–250, 1993.

[24] W. Athas, J. Koller, and L. Svensson. An energy efficient CMOS line driver using adiabatic switching. In *IEEE Great Lakes Symposium on VLSI*, 1994.

[25] R. Landauer. Uncertainty principle and minimal energy dissipation in a computer. *International Journal of Theoretical Physics*, 21:283–297, 1982.

[26] C. Bennett. The thermodynamics of computation - a review. *International Journal of Theoretical Physics*, 21:905–940, 1982.

[27] E. Fredkin and T. Toffoli. Conservative logic. *International Journal of Theoretical Physics*, 21:219–253, 1982.

[28] Jerome M. Shapiro. Embedded image coding using zerotrees of wavelet coefficients. *IEEE Transactions on Signal Processing*, 41:3445–3462, December 1993.

[29] J. W. Woods Ed. *Subband Image Coding*. Kluwer, 1991.

[30] G.K. Wallace. The JPEG still picture compression standard. *Communications of the ACM*, 34(4):30–44, April 1991.

[31] I. Daubechies. Orthonormal bases of compactly supported wavelets. *Communications Pure Applied Math*, 41:909–996, 1988.

[32] I. Daubechies. The wavelet transform, time-frequency localization and signal analysis. *IEEE Transactions on Information Theory*, 36:961–1005, Sept 1990.

[33] S. Mallat. Multifrequency channel decompositions of images and wavelet models. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 37:2091–2110, Dec 1990.

[34] O. Rioul and M. Vetterli. Wavelets and signal processing. *IEEE Signal Processing Mag.*, 8:14–38, Oct 1991.

[35] E. Adelson, E. Simoncelli, and R. Hingorani. Orthogonal pyramid transforms for image coding. In *SPIE Visual Communications and Image Processing II*, pages 50–58, 1987.

[36] W. Zettler, J. Huffman, and D. Linden. Applications of compactly supported wavelets to image compression. In *SPIE Image Processing Algorithms*, 1990.

[37] I. Witten, R. Neal, and J. Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30:520–540, June 1987.

[38] N. Abramson. *Information Theory and Coding*. McGraw-Hill, 1963.

[39] M. Sezan and R. Lagendijk Eds. *Motion Analysis and Image Sequence Processing*. Kluwer, 1993.

[40] H. M. Hang, A. Puri, and D. Scilling. Motion-compensated transform coding based on block motion-tracking algorithm. In *IEEE Int. Conf. Communications'87, vol 1*, pages 136–140, 1987.

[41] R. M. Armitano and R. W. Schafer. Motion vector estimation using spatio-temporal prediction and its application to video coding. In *SPIE Proceedings, vol 2668*, pages 290–301, 1996.

[42] S. Kim and C. C. Kuo. A stochastic approach for motion vector estimation in video coding. In *SPIE Proceedings, vol 2304*, pages 111–122, 1994.

[43] A. P. Chandrakasan and R. W. Brodersen. *Low Power Digital CMOS Design*. Kluwer Academic Publishers, Norwell, Massachusetts, 1995.

[44] A. Bhavnagarwala, B. Austin, and J. Meindl. Minimum supply voltage for bulk Si CMOS GSI. In *International Symposium on Low Power Electronics and Design*, 1998.

[45] N. Weste and K. Eshraghian. *Principles of CMOS VLSI Design, Sec. Ed.* Addison-Wesley, 1993.

[46] G. Neudeck and R. Pierret Eds. *Modular Series on Solid State Devices, 2nd ed.* Addison-Wesley, 1989.

Appendix A

Compression Algorithm C Code

A.1 Encoder

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ezw_common.c>

int ***image_anl;
int bits_to_follow;
int sub_poss;

void output_bit(int bit) {
  if (bit) fputc('1', outFile);
  else fputc('0', outFile);
}

void bit_plus_follow(int bit) {
  output_bit(bit);
  while (bits_to_follow) {output_bit(!bit); bits_to_follow--; }
}

void flush_arith_encode() {
  bits_to_follow++;
  if (low < first_qtr) bit_plus_follow(0);
  else bit_plus_follow(1);
}

void encode_sym(int sym,int table) {
  int neg_size = (~high + low) >> code[table];
  if (places[table] ^ sym) high = low + ~neg_size;
  else low = low + ~neg_size + 1;
  for (; 1;) {
    if (high < half) bit_plus_follow(0);
    else if (low >= half) {
      bit_plus_follow(1);
      low -= half;
      high -= half;
```

```
    }
    else if ((low >= first_qtr) && (high < third_qtr)) {
      bits_to_follow++;
      low -= first_qtr;
      high -= first_qtr;
    }
    else break;
    low = 2 * low;
    high = (2 * high) + 1;
  }
  if (table) update_model(sym, table);
}

void encode_sig(int sign) {
  if (zero_poss) encode_sym(0, 1);
  encode_sym(1, 2);
  encode_sym(sign, 0);
}

void encode_root() {
  if (zero_poss) encode_sym(0, 1);
  encode_sym(0, 2);
}

void encode_zero() {
  encode_sym(1, 1);
}

void encode_sub_bit(int bit) {
  if (bit) bit = 1;
  encode_sym(bit, 0);
}

void code_pass(int bit) {
  int row, col, lvl, rows, cols, x, y, rowy, colx;
  int **part_of_tree = make2dintarray(imagecols,imagerows);
  int **zero_root = make2dintarray(imagecols,imagerows);
  int sub_bit = bit >> 1;

  sub_poss = sub_bit;

  rows = imagerows >> 1;
  cols = imagecols >> 1;
  for (row = 0; row < rows; row++)
    for (col = cols; col < imagecols; col++)
      zero_root[col][row] = ((mags[col][row] & bit) == 0) ||
                            found[col][row];
  for (row = rows; row < imagerows; row++)
    for (col = 0; col < cols; col++)
      zero_root[col][row] = ((mags[col][row] & bit) == 0) ||
                            found[col][row];
  for (row = rows; row < imagerows; row++)
    for (col = cols; col < imagecols; col++)
      zero_root[col][row] = ((mags[col][row] & bit) == 0) ||
                            found[col][row];
```

```
   for (lvl = 2; lvl <= sblvls; lvl++) {
     rows = imagerows >> lvl;
     cols = imagecols >> lvl;
     for (row = 0; row < rows; row++)
       for (col = cols; col < (cols << 1); col++)
zero_root[col][row] = (((mags[col][row] & bit) == 0) ||
                                found[col][row]) &&
                       zero_root[col << 1][row << 1] &&
                       zero_root[(col << 1) + 1][row << 1] &&
                       zero_root[col << 1][(row << 1) + 1] &&
                       zero_root[(col << 1) + 1][(row << 1) + 1];
     for (row = rows; row < (rows << 1); row++)
       for (col = 0; col < cols; col++)
zero_root[col][row] = (((mags[col][row] & bit) == 0) ||
                                found[col][row]) &&
                       zero_root[col << 1][row << 1] &&
                       zero_root[(col << 1) + 1][row << 1] &&
                       zero_root[col << 1][(row << 1) + 1] &&
                       zero_root[(col << 1) + 1][(row << 1) + 1];
     for (row = rows; row < (rows << 1); row++)
       for (col = cols; col < (cols << 1); col++)
zero_root[col][row] = (((mags[col][row] & bit) == 0) ||
                                found[col][row]) &&
                       zero_root[col << 1][row << 1] &&
                       zero_root[(col << 1) + 1][row << 1] &&
                       zero_root[col << 1][(row << 1) + 1] &&
                       zero_root[(col << 1) + 1][(row << 1) + 1];
   }
   rows = imagerows >> sblvls;
   cols = imagecols >> sblvls;
   init_arith_model();
   zero_poss = 1;
   for (row = 0; row < rows; row++)
     for (col = 0; col < cols; col++) {
       if (found[col][row] == 0)
if (mags[col][row] & bit) {
  found[col][row] = 1; encode_sig(signs[col][row]); }
else if (zero_root[col + cols][row] &&
 zero_root[col][row + rows] &&
 zero_root[col + cols][row + rows]) {
  encode_root();
  flag_dscndnts(col + cols,row       ,sblvls,part_of_tree);
  flag_dscndnts(col       ,row + rows,sblvls,part_of_tree);
  flag_dscndnts(col + cols,row + rows,sblvls,part_of_tree); }
else encode_zero();
       if (sub_bit && found[col][row])
         encode_sub_bit(mags[col][row] & sub_bit);
     }
   init_arith_model();
   if (bit == (1 << (alu_bits - 2 - enc_renorm[2] - bottom_bit)))
     zero_poss = 0;
   for (row = 0; row < rows; row++) {
     for (col = 0; col < cols; col++) {
       if ((found[col + cols][row] == 0) &&
           (part_of_tree[col + cols][row] == 0)) {
```

```
if (mags[col + cols][row] & bit) {
  found[col + cols][row] = 1; encode_sig(signs[col + cols][row]); }
else if (zero_root[col + cols][row]) {
  encode_root(); flag_dscndnts(col + cols,row,sblvls,part_of_tree); }
else encode_zero(); }
      if (sub_bit && found[col + cols][row])
         encode_sub_bit(mags[col + cols][row] & sub_bit);
    }
    for (col = 0; col < cols; col++) {
      if ((found[col][row + rows] == 0) &&
          (part_of_tree[col][row + rows] == 0)) {
if (mags[col][row + rows] & bit) {
  found[col][row + rows] = 1; encode_sig(signs[col][row + rows]); }
else if (zero_root[col][row + rows]) {
  encode_root(); flag_dscndnts(col,row + rows,sblvls,part_of_tree); }
else encode_zero(); }
      if (sub_bit && found[col][row + rows])
         encode_sub_bit(mags[col][row + rows] & sub_bit);
      if ((found[col + cols][row + rows] == 0) &&
          (part_of_tree[col + cols][row + rows] == 0)) {
if (mags[col + cols][row + rows] & bit) {
  found[col + cols][row + rows] = 1;
          encode_sig(signs[col + cols][row + rows]); }
else if (zero_root[col + cols][row + rows]) {
  encode_root();
          flag_dscndnts(col + cols,row + rows,sblvls,part_of_tree); }
else encode_zero(); }
      if (sub_bit && found[col + cols][row + rows])
         encode_sub_bit(mags[col + cols][row + rows] & sub_bit);
    }
  }
  init_arith_model();
  if (bit == (1 << (alu_bits - 2 - enc_renorm[1] - bottom_bit)))
    zero_poss = 0;
  if (bit < (1 << (alu_bits - 1 - enc_renorm[1] - bottom_bit))) {
    rows = imagerows >> 2;
    cols = imagecols >> 2;
    for (row = 0; row < rows; row++)
      for (col = cols; col < (cols << 1); col++) {
if ((found[col][row] == 0) && (part_of_tree[col][row] == 0)) {
  if (mags[col][row] & bit) {
    found[col][row] = 1; encode_sig(signs[col][row]); }
  else if (zero_root[col][row]) {
    encode_root(); flag_dscndnts(col,row,2,part_of_tree); }
  else encode_zero(); }
if (sub_bit && found[col][row])
          encode_sub_bit(mags[col][row] & sub_bit);
      }
    for (row = rows; row < (rows << 1); row++)
      for (col = 0; col < cols; col++) {
if ((found[col][row] == 0) && (part_of_tree[col][row] == 0)) {
  if (mags[col][row] & bit) {
    found[col][row] = 1; encode_sig(signs[col][row]); }
  else if (zero_root[col][row]) {
    encode_root(); flag_dscndnts(col,row,2,part_of_tree); }
```

```
  else encode_zero(); }
if (sub_bit && found[col][row])
          encode_sub_bit(mags[col][row] & sub_bit);
      }
    for (row = rows; row < (rows << 1); row++)
      for (col = cols; col < (cols << 1); col++) {
if ((found[col][row] == 0) && (part_of_tree[col][row] == 0)) {
  if (mags[col][row] & bit) {
    found[col][row] = 1; encode_sig(signs[col][row]); }
  else if (zero_root[col][row]) {
    encode_root(); flag_dscndnts(col,row,2,part_of_tree); }
  else encode_zero(); }
if (sub_bit && found[col][row])
          encode_sub_bit(mags[col][row] & sub_bit);
      }
  }
  init_arith_model();
  zero_poss = 0;
  if (bit < (1 << (alu_bits - 1 - enc_renorm[0] - bottom_bit))) {
    rows = imagerows >> 1;
    cols = imagecols >> 1;
    for (y = 0; y < 2; y++)
    for (x = 0; x < 2; x++)
    for (rowy = 0; rowy < rows; rowy += 2)
      for (colx = cols; colx < (cols << 1); colx += 2) {
row = rowy + y;
col = colx + x;
if ((found[col][row] == 0) && (part_of_tree[col][row] == 0)) {
  if (mags[col][row] & bit) {
    found[col][row] = 1; encode_sig(signs[col][row]); }
  else if (zero_root[col][row]) {
    encode_root(); flag_dscndnts(col,row,1,part_of_tree); }
  else encode_zero(); }
if (sub_bit && found[col][row])
          encode_sub_bit(mags[col][row] & sub_bit);
      }
    for (y = 0; y < 2; y++)
    for (x = 0; x < 2; x++)
    for (rowy = rows; rowy < (rows << 1); rowy += 2)
      for (colx = 0; colx < cols; colx += 2) {
row = rowy + y;
col = colx + x;
if ((found[col][row] == 0) && (part_of_tree[col][row] == 0)) {
  if (mags[col][row] & bit) {
    found[col][row] = 1; encode_sig(signs[col][row]); }
  else if (zero_root[col][row]) {
    encode_root(); flag_dscndnts(col,row,1,part_of_tree); }
  else encode_zero(); }
if (sub_bit && found[col][row])
          encode_sub_bit(mags[col][row] & sub_bit);
      }
    for (y = 0; y < 2; y++)
    for (x = 0; x < 2; x++)
    for (rowy = rows; rowy < (rows << 1); rowy += 2)
      for (colx = cols; colx < (cols << 1); colx += 2) {
```

```
row = rowy + y;
col = colx + x;
if ((found[col][row] == 0) && (part_of_tree[col][row] == 0)) {
  if (mags[col][row] & bit) {
    found[col][row] = 1; encode_sig(signs[col][row]); }
  else if (zero_root[col][row]) {
    encode_root(); flag_dscndnts(col,row,1,part_of_tree); }
  else encode_zero(); }
if (sub_bit && found[col][row])
          encode_sub_bit(mags[col][row] & sub_bit);
      }
  }
}

void loadimage() {
  unsigned char temp_char;
  int row, col;
  image_mean = 0;
  fscanf(inFile, "P5 %d %d 255", &imagecols, &imagerows);
  fread(&temp_char,1,1,inFile);
  for (row = 0; row < imagerows; row++)
    for (col = 0; col < imagecols; col++) {
      fread(&temp_char,1,1,inFile);
      if (feof(inFile)) {printf("ERROR: too few pixels\n"); exit(0); }
      image_anl[col][row][0] = (int) temp_char;
      image_mean += image_anl[col][row][0];
    }
  fread(&temp_char,1,1,inFile);
  if (feof(inFile) == 0) {printf("ERROR: too many pixels\n"); exit(0); }
  image_mean = image_mean / imagerows;
  image_mean = image_mean / imagecols;
  for (row = 0; row < imagerows; row++)
    for (col = 0; col < imagecols; col++)
      image_anl[col][row][0] = image_anl[col][row][0] - image_mean;
}

void analyze_image(int lvl) {
  int row, col, coef, index, cols, rows;
  cols = imagecols >> (lvl - 1);
  rows = imagerows >> (lvl - 1);
  for (row = 0; row < rows; row++)
    for (col = 0; col < (cols / 2); col++) {
      for (coef = 0; coef < enc_filter_len; coef++) {
index = (col * 2) + (enc_filter_len / 2) - coef;
filter_args[coef] = image_anl[reflect(index, cols)][row][0];
      }
      image_anl[col][row][1] = (*(enc_lo_pass))();
      for (coef = 0; coef < enc_filter_len; coef++) {
index = (col * 2) + (enc_filter_len / 2) - coef + 1;
filter_args[coef] = image_anl[reflect(index, cols)][row][0];
      }
      image_anl[col + (cols / 2)][row][1] = (*(enc_hi_pass))();
    }
  for (row = 0; row < (rows / 2); row++)
    for (col = 0; col < cols; col++) {
```

```
      for (coef = 0; coef < enc_filter_len; coef++) {
index = (row * 2) + (enc_filter_len / 2) - coef;
filter_args[coef] = image_anl[col][reflect(index, rows)][1];
      }
      image_anl[col][row][0] = (*(enc_lo_pass))();
      for (coef = 0; coef < enc_filter_len; coef++) {
index = (row * 2) + (enc_filter_len / 2) - coef + 1;
filter_args[coef] = image_anl[col][reflect(index, rows)][1];
      }
      image_anl[col][row + (rows / 2)][0] = (*(enc_hi_pass))();
    }
}

main (int argc,char *argv[]) {
  int row, col, lvl, bit, frame, grp, rows, cols, renorm, tmp;
  argc--;
  argv++;
  if(argc != 1) {printf("Invalid number of arguments.\n"); exit(0); }
  imagecols = 128;
  imagerows = 128;
  alu_bits = 12;
  sblvls = 3;
  group_size = 16;
  groups = 1;
  sscanf(*argv++, "%d", &ezw_passes);
  bottom_bit = 8 - ezw_passes;
  sign_bits_mask = 0 - (1 << (alu_bits - 1));
  filename_in = "infrm";
  filename_out = "rf";

  image_anl = make3dintarray(imagecols,imagerows,2);
  signs = make2dintarray(imagecols,imagerows);
  mags = make2dintarray(imagecols,imagerows);
  found = make2dintarray(imagecols,imagerows);
  prev_frame = make2dintarray(imagecols,imagerows);

  for (grp = 0; grp < groups; grp++) {
    outFile = fopen(make_frm_name(filename_out, grp), "w");
    bits_to_follow = 0;
    low = 0;
    high = top_value;
    init_arith_model();
    for (row = 0; row < imagerows; row++)
      for (col = 0; col < imagecols; col++)
        prev_frame[col][row] = 0;
    for (frame = 0; frame < group_size; frame++) {
      inFile = fopen(make_frm_name(filename_in,
                                   (grp * group_size) + frame),
                     "r");
      loadimage();
      fclose(inFile);
      if (frame == 0)
        for (bit = 7; bit >= 0; bit--)
          encode_sym((image_mean >> bit) & 1, 0);
      for (row = 0; row < imagerows; row++)
```

```
for (col = 0; col < imagecols; col++)
  image_anl[col][row][0] = image_anl[col][row][0] << 3;
      for (lvl = 1; lvl <= sblvls; lvl++) analyze_image(lvl);
      for (lvl = 1; lvl <= sblvls; lvl++) {
rows = imagerows >> lvl;
cols = imagecols >> lvl;
renorm = enc_renorm[lvl - 1];
bit = 0;
if ((bottom_bit - 1 + renorm) > 0)
          bit = 1 << (bottom_bit - 2 + renorm);
if (lvl == sblvls)
  for (row = 0; row < rows; row++)
    for (col = 0; col < cols; col++) {
      tmp = prev_frame[col][row] << renorm;
      tmp = image_anl[col][row][0] - tmp;
      signs[col][row] = tmp < 0;
      mags[col][row] = (mag(tmp) + bit) >> (bottom_bit + renorm);
    }
for (row = 0; row < rows; row++)
  for (col = cols; col < (cols << 1); col++) {
    tmp = prev_frame[col][row] << renorm;
    tmp = image_anl[col][row][0] - tmp;
    signs[col][row] = tmp < 0;
    mags[col][row] = (mag(tmp) + bit) >> (bottom_bit + renorm);
  }
for (row = rows; row < (rows << 1); row++)
  for (col = 0; col < cols; col++) {
    tmp = prev_frame[col][row] << renorm;
    tmp = image_anl[col][row][0] - tmp;
    signs[col][row] = tmp < 0;
    mags[col][row] = (mag(tmp) + bit) >> (bottom_bit + renorm);
  }
for (row = rows; row < (rows << 1); row++)
  for (col = cols; col < (cols << 1); col++) {
    tmp = prev_frame[col][row] << renorm;
    tmp = image_anl[col][row][0] - tmp;
    signs[col][row] = tmp < 0;
    mags[col][row] = (mag(tmp) + bit) >> (bottom_bit + renorm);
  }
      }
      bit = 0;
      for (row = 0; row < imagerows; row++)
for (col = 0; col < imagecols; col++) {
  bit |= mags[col][row];
  found[col][row] = 0;
  tmp = mags[col][row] << bottom_bit;
  if (signs[col][row])
          prev_frame[col][row] = prev_frame[col][row] - tmp;
  else prev_frame[col][row] = prev_frame[col][row] + tmp;
}
      for (top_bit = 1 << (9 - bottom_bit);
           top_bit && !(top_bit & bit);
           top_bit = (top_bit >> 1))
encode_sym(0, 0);
      if (top_bit) encode_sym(1, 0);
```

```
      for (bit = top_bit; bit; bit = (bit >> 1)) code_pass(bit);
      if (top_bit && frame)
        for (bit = 7; bit >= 0; bit--)
          encode_sym((image_mean >> bit) & 1, 0);
    }
    flush_arith_encode();
    fclose(outFile);
  }
}
```

A.2 Decoder

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ezw_common.c>

double ***image_syn;
int value;

#define root_sym 0
#define pos_sym 1
#define neg_sym 2
#define zero_sym 3

int input_bit() {
  int next_bit = fgetc(inFile);
  if (next_bit == '1') return 1;
  else return 0;
}

void init_arith_decode() {
  int i;
  value = 0;
  for (i = 0; i < value_bits; i++)
    value = (value << 1) + input_bit();
}

int decode_sym(int table) {
  int range = high - low + 1;
  int neg_size = (- range) >> code[table];
  int trunc = ((- range) & ((1 << code[table]) - 1)) > 0;
  int place = (1 << (8 - code[table])) >
              ((((value - low + !trunc) << 8) - 1) / range);
  int sym = places[table] ^ place;
  if (place) high = low + ~neg_size;
  else low = low + ~neg_size + 1;
  for (; 1;) {
    if (high < half) ;
    else if (low >= half) {
      value -= half;
      low   -= half;
      high  -= half;
    }
    else if ((low >= first_qtr) && (high < third_qtr)) {
      value -= first_qtr;
      low   -= first_qtr;
      high  -= first_qtr;
    }
    else break;
    low = 2 * low;
    high = (2 * high) + 1;
    value = (2 * value) + input_bit();
  }
  if (table) update_model(sym, table);
```

```
  return sym;
}

int decode_dom_sym() {
  int sym1 = 0;
  if (zero_poss) sym1 = decode_sym(1);
  if (sym1) return zero_sym;
  if (decode_sym(2))
    if (decode_sym(0)) return neg_sym;
    else return pos_sym;
  return root_sym;
}

int decode_sub_bit() {
  return decode_sym(0);
}

void decode_pass(int bit) {
  int row, col, lvl, sym, x, y, rowy, colx;
  int **part_of_tree = make2dintarray(imagecols,imagerows);
  int rows = imagerows >> sblvls;
  int cols = imagecols >> sblvls;
  int sub_bit = bit >> 1;

  init_arith_model();
  zero_poss = 1;
  for (row = 0; row < rows; row++)
    for (col = 0; col < cols; col++) {
      if (found[col][row] == 0) {
sym = decode_dom_sym();
if (sym == pos_sym) {found[col][row] = 1; mags[col][row] = bit; }
else if (sym == neg_sym) {
          found[col][row] = 1;
          signs[col][row] = 1;
          mags[col][row] = bit; }
else if (sym == root_sym) {
  flag_dscndnts(col + cols,row       ,sblvls,part_of_tree);
  flag_dscndnts(col       ,row + rows,sblvls,part_of_tree);
  flag_dscndnts(col + cols,row + rows,sblvls,part_of_tree); }
      }
      if (sub_bit && found[col][row])
if (decode_sub_bit()) mags[col][row] = mags[col][row] | sub_bit;
    }

  init_arith_model();
  if (bit == (1 << (alu_bits - 2 - enc_renorm[2] - bottom_bit)))
    zero_poss = 0;
  for (row = 0; row < rows; row++) {
    for (col = 0; col < cols; col++) {
      if ((found[col + cols][row] == 0) &&
          (part_of_tree[col + cols][row] == 0)) {
sym = decode_dom_sym();
if (sym == pos_sym) {
          found[col + cols][row] = 1;
          mags[col + cols][row] = bit; }
```

```
else if (sym == neg_sym) {
          found[col + cols][row] = 1;
          signs[col + cols][row] = 1;
          mags[col + cols][row] = bit; }
else if (sym == root_sym)
          flag_dscndnts(col + cols,row,sblvls,part_of_tree);
      }
      if (sub_bit && found[col + cols][row])
if (decode_sub_bit())
          mags[col + cols][row] = mags[col + cols][row] | sub_bit;
    }
    for (col = 0; col < cols; col++) {
      if ((found[col][row + rows] == 0) &&
          (part_of_tree[col][row + rows] == 0)) {
sym = decode_dom_sym();
if (sym == pos_sym) {
          found[col][row + rows] = 1;
          mags[col][row + rows] = bit; }
else if (sym == neg_sym) {
          found[col][row + rows] = 1;
          signs[col][row + rows] = 1;
          mags[col][row + rows] = bit; }
else if (sym == root_sym)
          flag_dscndnts(col,row + rows,sblvls,part_of_tree);
      }
      if (sub_bit && found[col][row + rows])
if (decode_sub_bit())
          mags[col][row + rows] = mags[col][row + rows] | sub_bit;
      if ((found[col + cols][row + rows] == 0) &&
          (part_of_tree[col + cols][row + rows] == 0)) {
sym = decode_dom_sym();
if (sym == pos_sym) {
          found[col + cols][row + rows] = 1;
          mags[col + cols][row + rows] = bit; }
else if (sym == neg_sym) {
          found[col + cols][row + rows] = 1;
          signs[col + cols][row + rows] = 1;
          mags[col + cols][row + rows] = bit; }
else if (sym == root_sym)
          flag_dscndnts(col + cols,row + rows,sblvls,part_of_tree);
      }
      if (sub_bit && found[col + cols][row + rows])
if (decode_sub_bit())
          mags[col + cols][row + rows] =
            mags[col + cols][row + rows] | sub_bit;
    }
  }
  init_arith_model();
  if (bit == (1 << (alu_bits - 2 - enc_renorm[1] - bottom_bit)))
    zero_poss = 0;
  if (bit < (1 << (alu_bits - 1 - enc_renorm[1] - bottom_bit))) {
    int rows = imagerows >> 2;
    int cols = imagecols >> 2;
    for (row = 0; row < rows; row++)
      for (col = cols; col < (cols << 1); col++) {
```

```
if ((found[col][row] == 0) && (part_of_tree[col][row] == 0)) {
  sym = decode_dom_sym();
  if (sym == pos_sym) {found[col][row] = 1; mags[col][row] = bit; }
  else if (sym == neg_sym) {
            found[col][row] = 1;
            signs[col][row] = 1;
            mags[col][row] = bit; }
  else if (sym == root_sym) flag_dscndnts(col,row,2,part_of_tree);
}
if (sub_bit && found[col][row])
  if (decode_sub_bit()) mags[col][row] = mags[col][row] | sub_bit;
      }
    for (row = rows; row < (rows << 1); row++)
      for (col = 0; col < cols; col++) {
if ((found[col][row] == 0) && (part_of_tree[col][row] == 0)) {
  sym = decode_dom_sym();
  if (sym == pos_sym) {found[col][row] = 1; mags[col][row] = bit; }
  else if (sym == neg_sym) {
            found[col][row] = 1;
            signs[col][row] = 1;
            mags[col][row] = bit; }
  else if (sym == root_sym) flag_dscndnts(col,row,2,part_of_tree);
}
if (sub_bit && found[col][row])
  if (decode_sub_bit()) mags[col][row] = mags[col][row] | sub_bit;
      }
    for (row = rows; row < (rows << 1); row++)
      for (col = cols; col < (cols << 1); col++) {
if ((found[col][row] == 0) && (part_of_tree[col][row] == 0)) {
  sym = decode_dom_sym();
  if (sym == pos_sym) {found[col][row] = 1; mags[col][row] = bit; }
  else if (sym == neg_sym) {
            found[col][row] = 1;
            signs[col][row] = 1;
            mags[col][row] = bit; }
  else if (sym == root_sym) flag_dscndnts(col,row,2,part_of_tree);
}
if (sub_bit && found[col][row])
  if (decode_sub_bit()) mags[col][row] = mags[col][row] | sub_bit;
      }
  }
  init_arith_model();
  zero_poss = 0;
  if (bit < (1 << (alu_bits - 1 - enc_renorm[0] - bottom_bit))) {
    int rows = imagerows >> 1;
    int cols = imagecols >> 1;
    for (y = 0; y < 2; y++)
    for (x = 0; x < 2; x++)
    for (rowy = 0; rowy < rows; rowy += 2)
      for (colx = cols; colx < (cols << 1); colx += 2) {
row = rowy + y;
col = colx + x;
if ((found[col][row] == 0) && (part_of_tree[col][row] == 0)) {
  sym = decode_dom_sym();
  if (sym == pos_sym) {found[col][row] = 1; mags[col][row] = bit; }
```

```
  else if (sym == neg_sym) {
            found[col][row] = 1;
            signs[col][row] = 1;
            mags[col][row] = bit; }
  else if (sym == root_sym) flag_dscndnts(col,row,1,part_of_tree);
}
if (sub_bit && found[col][row])
  if (decode_sub_bit()) mags[col][row] = mags[col][row] | sub_bit;
      }
    for (y = 0; y < 2; y++)
    for (x = 0; x < 2; x++)
    for (rowy = rows; rowy < (rows << 1); rowy += 2)
      for (colx = 0; colx < cols; colx += 2) {
row = rowy + y;
col = colx + x;
if ((found[col][row] == 0) && (part_of_tree[col][row] == 0)) {
  sym = decode_dom_sym();
  if (sym == pos_sym) {found[col][row] = 1; mags[col][row] = bit; }
  else if (sym == neg_sym) {
            found[col][row] = 1;
            signs[col][row] = 1;
            mags[col][row] = bit; }
  else if (sym == root_sym) flag_dscndnts(col,row,1,part_of_tree);
}
if (sub_bit && found[col][row])
  if (decode_sub_bit()) mags[col][row] = mags[col][row] | sub_bit;
      }
    for (y = 0; y < 2; y++)
    for (x = 0; x < 2; x++)
    for (rowy = rows; rowy < (rows << 1); rowy += 2)
      for (colx = cols; colx < (cols << 1); colx += 2) {
row = rowy + y;
col = colx + x;
if ((found[col][row] == 0) && (part_of_tree[col][row] == 0)) {
  sym = decode_dom_sym();
  if (sym == pos_sym) {found[col][row] = 1; mags[col][row] = bit; }
  else if (sym == neg_sym) {
            found[col][row] = 1;
            signs[col][row] = 1;
            mags[col][row] = bit; }
  else if (sym == root_sym) flag_dscndnts(col,row,1,part_of_tree);
}
if (sub_bit && found[col][row])
  if (decode_sub_bit()) mags[col][row] = mags[col][row] | sub_bit;
      }
  }
}

void dumpimage() {
  unsigned char temp_char;
  int row, col;
  fprintf(outFile,"P5 %d %d 255\n",imagecols,imagerows);
  for (row = 0; row < imagerows; row++)
    for (col = 0; col < imagecols; col++) {
      image_syn[col][row][0] += image_mean;
```

```
      temp_char = (char) image_syn[col][row][0];
      if (image_syn[col][row][0] < 0) temp_char = 0;
      if (image_syn[col][row][0] > 255) temp_char = 255;
      fwrite(&temp_char,1,1,outFile);
    }
}

int even(int arg) {
  return 1 - (arg & 1);
}

void synthesize_image(int lvl) {
  int row, col, coef, index, cols, rows;
  cols = imagecols >> (lvl - 1);
  rows = imagerows >> (lvl - 1);
  for (row = 0; row < rows; row++)
    for (col = 0; col < cols; col++) {
      image_syn[col][row][1] = 0;
      for (coef = 0; coef < dec_filter_len; coef++) {
index = reflect(row + (dec_filter_len / 2) - coef, rows);
if (even(index))
  image_syn[col][row][1] +=
    dec_lo_pass[coef] * image_syn[col][index / 2][0];
index = reflect(row + (dec_filter_len / 2) - coef - 1, rows);
if (even(index))
  image_syn[col][row][1] +=
    dec_hi_pass[coef] * image_syn[col][(index / 2) + (rows / 2)][0];
      }
    }
  for (row = 0; row < rows; row++)
    for (col = 0; col < cols; col++) {
      image_syn[col][row][0] = 0;
      for (coef = 0; coef < dec_filter_len; coef++) {
index = reflect(col + (dec_filter_len / 2) - coef, cols);
if (even(index))
  image_syn[col][row][0] +=
    dec_lo_pass[coef] * image_syn[index / 2][row][1];
index = reflect(col + (dec_filter_len / 2) - coef - 1, cols);
if (even(index))
  image_syn[col][row][0] +=
    dec_hi_pass[coef] * image_syn[(index / 2) + (cols / 2)][row][1];
      }
    }
}

main (int argc,char *argv[]) {
  int row, col, lvl, bit, frame, grp;
  argc--;
  argv++;
  if(argc != 1) {printf("Invalid number of arguments.\n"); exit(0); }
  filename_in = "rf";
  filename_out = "outfrm";
  imagecols = 128;
  imagerows = 128;
  alu_bits = 12;
```

```
    sblvls = 3;
    group_size = 16;
    groups = 1;
    sscanf(*argv++, "%d", &ezw_passes);
    bottom_bit = 8 - ezw_passes;
    image_syn = make3ddblarray(imagecols,imagerows,2);
    signs = make2dintarray(imagecols,imagerows);
    mags = make2dintarray(imagecols,imagerows);
    found = make2dintarray(imagecols,imagerows);
    prev_frame = make2dintarray(imagecols,imagerows);

    for (grp = 0; grp < groups; grp++) {
      inFile = fopen(make_frm_name(filename_in, grp), "r");
      low = 0;
      high = top_value;
      init_arith_decode();
      init_arith_model();
      for (row = 0; row < imagerows; row++)
        for (col = 0; col < imagecols; col++)
prev_frame[col][row] = 0;
      image_mean = 0;
      for (bit = 0; bit < 8; bit++)
        image_mean = (image_mean << 1) + decode_sym(0);

      for (frame = 0; frame < group_size; frame++) {
        for (row = 0; row < imagerows; row++)
for (col = 0; col < imagecols; col++) {
  mags[col][row] = 0;
  signs[col][row] = 0;
  found[col][row] = 0;
}
        for (top_bit = 1 << (9 - bottom_bit);
             top_bit && !decode_sym(0);
             top_bit = top_bit >> 1);
        for (bit = top_bit; bit; bit = (bit >> 1)) decode_pass(bit);
        if (top_bit && frame) {
image_mean = 0;
for (bit = 0; bit < 8; bit++)
  image_mean = (image_mean << 1) + decode_sym(0);
        }
        for (row = 0; row < imagerows; row++)
for (col = 0; col < imagecols; col++) {
  if (found[col][row]) {
    if (signs[col][row])
      prev_frame[col][row] -= mags[col][row] << (bottom_bit + 1);
    else prev_frame[col][row] += mags[col][row] << (bottom_bit + 1);
  }
  image_syn[col][row][0] = (double) prev_frame[col][row];
}
        for (lvl = sblvls; lvl > 0; lvl--) synthesize_image(lvl);
        outFile = fopen(make_frm_name(filename_out,
                                      (grp * group_size) + frame),
                        "w");
        dumpimage();
        fclose(outFile);
```

```
    }
    fclose(inFile);
  }
}
```

A.3 Common Subroutines

```
FILE *inFile;
FILE *outFile;
char *filename_in;
char *filename_out;
int **signs;
int **mags;
int **found;
int **prev_frame;
int imagecols, imagerows, sblvls, top_bit, image_mean;
int group_size, groups, bottom_bit, ezw_passes;
int alu_bits, sign_bits_mask, zero_poss;
int filter_args[5] = {0, 0, 0, 0, 0};

/* arithmetic coder defs */
#define value_bits 11
#define top_value ((1 << value_bits) - 1)
#define first_qtr ((top_value >> 2) + 1)
#define half      (first_qtr << 1)
#define third_qtr (first_qtr + half)
#define max_freq 255

int low, high;

#define tbls 3
int freq[tbls][2];
int places[tbls];
int code[tbls];

/* 5 x 5 wavelet filter */
double dec_lo_pass5[] = {-0.0761, 0.3536, 0.8593, 0.3536, -0.0761};
double dec_hi_pass5[] = {-0.0761, -0.3536, 0.8593, -0.3536, -0.0761};

int enc_lo5() {
  int a;
  a = (filter_args[0] + filter_args[4]) >> 1;
  a = a - filter_args[2];
  a = (a >> 3) - a;
  a = (a >> 1) + filter_args[1];
  a = (a + filter_args[3]) >> 1;
  a = (a + filter_args[2]) >> 1;
  return a;
}

int enc_hi5() {
  int a, b;
  a = (filter_args[0] + filter_args[4]) >> 1;
  a = (a >> 1) - (filter_args[2] >> 1);
  a = (a >> 3) - a;
  a = (a >> 1) - (filter_args[1] >> 1);
  a = a - (filter_args[3] >> 1);
  a = (a + filter_args[2]) >> 1;
  return a;
}
```

```
int  dec_filter_len = 5;
double *dec_lo_pass = dec_lo_pass5;
double *dec_hi_pass = dec_hi_pass5;

typedef int (*PROC)();

int enc_filter_len = 5;
int   enc_renorm[] = {3, 2, 1, 0};
PROC enc_lo_pass   = enc_lo5;
PROC enc_hi_pass   = enc_hi5;

char *strappend(char *str1,char *str2) {
  char *result;
  result = calloc((strlen(str1) + strlen(str2) + 1), sizeof(char));
  strcpy(result, str1);
  strcat(result, str2);
  return result;
}

char *make_frm_name(char *name, int frnum) {
  char *ans;
  ans = calloc(5, sizeof(char));
  if (frnum < 10) sprintf(ans, ".00%d", frnum);
  else if (frnum < 100) sprintf(ans, ".0%d", frnum);
  else sprintf(ans, ".%d", frnum);
  return strappend(name, ans);
}

int **make2dintarray(int d1, int d2) {
  int **array;
  int index;
  array = calloc(d1, sizeof(int *));
  for (index = 0; index < d1; index++)
    array[index] = (int *) calloc(d2,sizeof(int));
  return array;
}

double **make2ddblarray(int d1, int d2) {
  double **array;
  int index;
  array = calloc(d1, sizeof(double *));
  for (index = 0; index < d1; index++)
    array[index] = (double *) calloc(d2,sizeof(double));
  return array;
}

int ***make3dintarray(int d1, int d2, int d3) {
  int ***array;
  int index;
  array = calloc(d1, sizeof(int **));
  for (index = 0; index < d1; index++)
    array[index] = make2dintarray(d2,d3);
  return array;
}
```

```
double ***make3ddblarray(int d1, int d2, int d3) {
  double ***array;
  int index;
  array = calloc(d1, sizeof(double **));
  for (index = 0; index < d1; index++)
    array[index] = make2ddblarray(d2,d3);
  return array;
}

int mag(int arg) {
  if (arg < 0) arg = 0 - arg;
  return arg;
}

int reflect(int arg,int bound) {
  arg = mag(arg);
  if (arg >= bound) arg = (2 * (bound - 1)) - arg;
  return arg;
}

void flag_dscndnts(int col,int row,int lvl,int **part_of_tree) {
  if (lvl > 0) {
    part_of_tree[col][row] = 1;
    flag_dscndnts((col << 1)    ,(row << 1)    ,lvl - 1,part_of_tree);
    flag_dscndnts((col << 1) + 1,(row << 1)    ,lvl - 1,part_of_tree);
    flag_dscndnts((col << 1)    ,(row << 1) + 1,lvl - 1,part_of_tree);
    flag_dscndnts((col << 1) + 1,(row << 1) + 1,lvl - 1,part_of_tree);
  }
}

void update_model(int sym,int table) {
  int cums0, cums1, total;
  freq[table][sym]++;
  places[table] = freq[table][1] > freq[table][0];
  total = freq[table][0] + freq[table][1];
  for (cums0 = value_bits - 4; ((1 << cums0) & total) == 0; cums0--);
  for (cums1 = value_bits - 4;
       ((1 << cums1) & freq[table][1 - places[table]]) == 0;
       cums1--);
  code[table] = cums0 - cums1;
  if (code[table] < 1) code[table] = 1;
  if (total == max_freq) {
    freq[table][0] = (freq[table][0] >> 1) | 1;
    freq[table][1] = (freq[table][1] >> 1) | 1;
  }
}

void init_arith_model() {
  int j;
  for (j = 0; j < tbls; j++) {
    code[j] = 1;
    places[j] = 0;
    freq[j][0] = 1;
    freq[j][1] = 1;
```

```
  }
}

/*
;Alternate probability approximation with better
;rounding (worth < 1.5%).

int cum_freq[tbls][3];

void update_model(int sym,int table) {
  int i;
  int cums[2];
  int rounding_bits[2];
  int place = places[table] ^ sym;
  if (cum_freq[table][0] == max_freq)
    for (i = 1; i >= 0; i--) {
      freq[table][i] = (freq[table][i] >> 1) | 1;
      cum_freq[table][i] = cum_freq[table][i + 1] + freq[table][i];
    }
  freq[table][place]++;
  for (i = 0; i <= place; i++) cum_freq[table][i]++;
  if (freq[table][1] > freq[table][0]) {
    int tmp = freq[table][0];
    freq[table][0] = freq[table][1];
    freq[table][1] = tmp;
    cum_freq[table][1] = tmp;
    places[table] = 1 - places[table];
  }
  for (i = 0; i < 2; i++) {
    for (cums[i] = value_bits - 4;
         ((1 << cums[i]) & cum_freq[table][i]) == 0;
         cums[i]--);
    rounding_bits[i] = 0;
    if (cums[i])
      if ((1 << (cums[i] - 1)) & cum_freq[table][i])
rounding_bits[i] = 2;
    if (cums[i] > 1)
      if ((1 << (cums[i] - 2)) & cum_freq[table][i])
rounding_bits[i] += 1;
  }
  if ((rounding_bits[0] + (rounding_bits[1] > 1)) > 1) cums[0]++;
  if ((rounding_bits[1] + (rounding_bits[0] > 1)) > 1) cums[1]++;
  code[table] = cums[0] - cums[1];
  if (code[table] < 1) code[table] = 1;
}

void init_arith_model() {
  int i, j;
  for (j = 0; j < tbls; j++) {
    cum_freq[j][2] = 0;
    code[j] = 1;
    places[j] = 0;
    for (i = 0; i < 2; i++) {
      freq[j][i] = 1; cum_freq[j][i] = 2 - i; }
  }
```

```
}
*/
```

Appendix B

Microcode Assembler C Code

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

FILE *inFile;
FILE *outFile;
FILE *epromFileH;
FILE *epromFileM;
FILE *epromFileL;
int ezw_passes, eprom_block;

#define labels 40

char *lbl_names[labels];
int lbl_addr[labels];
char *line, *line_ptr;
int pc, next_lbl;
int inst_bit[20];
int src_addr, ld_addr;

#define cntrl_num 12

char *cntrl_name[cntrl_num] = {
  "dec_counter\n",
  "load_counter(bottom_bit)\n",
  "load_counter(ezw_passes)\n",
  "do_ezw_slot\n",
  "halt_till_new_frame\n",
  "pre_top_bit\n",
  "clr_sub_OK_bit\n",
  "set_sub_OK_bit\n",
  "clr_zero_possible\n",
  "set_zero_possible\n",
  "init_arith_tables\n",
  "init_arith_coder\n"
  };

char *cntrl_bits[cntrl_num] = {
```

```
  "0000",
  "0010",
  "0011",
  "0110",
  "0111",
  "1001",
  "1010",
  "1011",
  "1100",
  "1101",
  "1110",
  "1111"
  };

int cntrl_vals[cntrl_num] = {
  0,
  2,
  3,
  6,
  7,
  9,
  10,
  11,
  12,
  13,
  14,
  15,
  };

void first_pass() {
  fgets(line, 80, inFile);
  if (!feof(inFile)) {
    if ((strlen(line) == 1) || (strncmp(line, ";", 1) == 0))
      first_pass();
    else if (sscanf(line, "label %s", lbl_names[next_lbl]) == 1) {
      lbl_addr[next_lbl] = pc;
      next_lbl++;
      first_pass();
      }
    else {pc++; first_pass();}
  }
}

int cntrl_inst(int index) {
  if (index == cntrl_num) return 0;
  else if (strcmp(line, cntrl_name[index]))
    return cntrl_inst(index + 1);
  else {
    fprintf(outFile, "00%s00000000000%d%d%d\n", cntrl_bits[index],
    ezw_passes >> 2, (ezw_passes >> 1) & 1, ezw_passes & 1);
    fprintf(epromFileH, ":01%.4X00%.2X%.2X\n", pc,
            (cntrl_vals[index] >> 2),
            ((- ((cntrl_vals[index] >> 2) + 1 + (pc >> 8) + (pc & 255)))
             & 255));
    fprintf(epromFileM, ":01%.4X00%.2X%.2X\n", pc,
```

```
            ((cntrl_vals[index] & 3) << 6),
            ((- (((cntrl_vals[index] & 3) << 6) +
                 1 +
                 (pc >> 8) +
                 (pc & 255)))
             & 255));
    fprintf(epromFileL, ":01%.4X00%.2X%.2X\n", pc, ezw_passes,
            ((- (ezw_passes + 1 + (pc >> 8) + (pc & 255))) & 255));
    pc++;
    return 1;
  }
}

int find_lbl(int index) {
  if (index == next_lbl)
    {printf("Label not found: %s\n", lbl_names[next_lbl]); exit(0);}
  if (strcmp(lbl_names[index], lbl_names[next_lbl]))
    return find_lbl(index + 1);
  else return lbl_addr[index];
}

int jmp_inst() {
  int i, addr;

  if (sscanf(line, "jmp_if_counter %s", lbl_names[next_lbl]) == 1) {
    addr = find_lbl(0);
    fprintf(outFile, "001000100");
    for (i = 10; i >= 0; i--) fprintf(outFile, "%d", (addr >> i) & 1);
    fprintf(outFile, "\n");
    fprintf(epromFileH, ":01%.4X0002%.2X\n", pc,
            ((- (3 + (pc >> 8) + (pc & 255))) & 255));
    fprintf(epromFileM, ":01%.4X00%.2X%.2X\n", pc, (32 + (addr >> 8)),
            ((- (32 + (addr >> 8) + 1 + (pc >> 8) + (pc & 255)))
             & 255));
    fprintf(epromFileL, ":01%.4X00%.2X%.2X\n", pc, (addr & 255),
            ((- ((addr & 255) + 1 + (pc >> 8) + (pc & 255))) & 255));
    pc++;
    return 1;
    }
  if (sscanf(line, "jmp_if_not_mean_msb %s", lbl_names[next_lbl]) == 1)
    {
    addr = find_lbl(0);
    fprintf(outFile, "001000101");
    for (i = 10; i >= 0; i--) fprintf(outFile, "%d", (addr >> i) & 1);
    fprintf(outFile, "\n");
    fprintf(epromFileH, ":01%.4X0002%.2X\n", pc,
            ((- (3 + (pc >> 8) + (pc & 255))) & 255));
    fprintf(epromFileM, ":01%.4X00%.2X%.2X\n", pc, (40 + (addr >> 8)),
            ((- (40 + (addr >> 8) + 1 + (pc >> 8) + (pc & 255)))
             & 255));
    fprintf(epromFileL, ":01%.4X00%.2X%.2X\n", pc, (addr & 255),
            ((- ((addr & 255) + 1 + (pc >> 8) + (pc & 255))) & 255));
    pc++;
    return 1;
    }
```

```
  if (sscanf(line, "jmp_if_new_group %s", lbl_names[next_lbl]) == 1) {
    addr = find_lbl(0);
    fprintf(outFile, "001000110");
    for (i = 10; i >= 0; i--) fprintf(outFile, "%d", (addr >> i) & 1);
    fprintf(outFile, "\n");
    fprintf(epromFileH, ":01%.4X0002%.2X\n", pc,
            ((- (3 + (pc >> 8) + (pc & 255))) & 255));
    fprintf(epromFileM, ":01%.4X00%.2X%.2X\n", pc, (48 + (addr >> 8)),
            ((- (48 + (addr >> 8) + 1 + (pc >> 8) + (pc & 255)))
             & 255));
    fprintf(epromFileL, ":01%.4X00%.2X%.2X\n", pc, (addr & 255),
            ((- ((addr & 255) + 1 + (pc >> 8) + (pc & 255))) & 255));
    pc++;
    return 1;
    }
  if (sscanf(line, "jmp_if_not_top_bit %s", lbl_names[next_lbl]) == 1) {
    addr = find_lbl(0);
    fprintf(outFile, "001000111");
    for (i = 10; i >= 0; i--) fprintf(outFile, "%d", (addr >> i) & 1);
    fprintf(outFile, "\n");
    fprintf(epromFileH, ":01%.4X0002%.2X\n", pc,
            ((- (3 + (pc >> 8) + (pc & 255))) & 255));
    fprintf(epromFileM, ":01%.4X00%.2X%.2X\n", pc, (56 + (addr >> 8)),
            ((- (56 + (addr >> 8) + 1 + (pc >> 8) + (pc & 255)))
             & 255));
    fprintf(epromFileL, ":01%.4X00%.2X%.2X\n", pc, (addr & 255),
            ((- ((addr & 255) + 1 + (pc >> 8) + (pc & 255))) & 255));
    pc++;
    return 1;
    }
  if (strncmp(line, "debug", 5) == 0) {
    fprintf(outFile, "001C0000100000000000\n");
    return 1;
  }
  return 0;
}

void array_inst() {
  int value = 0;
  int index;

  inst_bit[19] = 1;
  inst_bit[18] = 1;
  inst_bit[17] = 0;
  src_addr = 49;
  ld_addr = 55;
  inst_bit[4] = 0;
  inst_bit[3] = 0;
  inst_bit[2] = 0;
  inst_bit[1] = 0;
  inst_bit[0] = 0;
  line_ptr = line;
  if (check_and_adv("mem[")) {
    sscanf(line_ptr, "%d]", &ld_addr);
    line_ptr = strchr(line_ptr, ']');
```

```
  check_and_adv("] = ");
  }
if (check_and_adv("cond_load_1 = ")) ld_addr = 48;
if (check_and_adv("cond_or_1 = ")) ld_addr = 49;
if (check_and_adv("cond_load_10 = ")) ld_addr = 50;
if (check_and_adv("cond_and_10 = ")) ld_addr = 51;
if (check_and_adv("news = ")) ld_addr = 52;
if (check_and_adv("news_if = ")) ld_addr = 53;
if (check_and_adv("reg = ")) ld_addr = 54;

if (check_and_adv("add(")) {
  inst_bit[4] = 1;
  }
if (check_and_adv("addh(")) {
  inst_bit[19] = 0;
  inst_bit[4] = 1;
  inst_bit[3] = 1;
  }
if (check_and_adv("sub(")) {
  inst_bit[17] = 1;
  inst_bit[4] = 1;
  }
if (check_and_adv("subh(")) {
  inst_bit[19] = 0;
  inst_bit[17] = 1;
  inst_bit[4] = 1;
  inst_bit[3] = 1;
  }
if (check_and_adv("inv(")) {
  inst_bit[17] = 1;
  }
if (check_and_adv("ezw(")) {
  inst_bit[3] = 1;
  }
if (check_and_adv("concat_cond(")) {
  inst_bit[0] = 1;
  }

if (check_and_adv("reg >> 1, ")) {
  inst_bit[3] = 1;
  }

if (check_and_adv("news")) {
  src_addr = 51;
  }
if (check_and_adv("mem[")) {
  sscanf(line_ptr, "%d]", &src_addr);
  line_ptr = strchr(line_ptr, ']');
  line_ptr++;
  }

if (check_and_adv(" >> 1")) {
  inst_bit[19] = 0;
  }
if (check_and_adv(" << 1")) {
```

```
    inst_bit[18] = 0;
    }

  while (check_and_adv(" ") ||
         check_and_adv(",") ||
         check_and_adv(")")) ;

  if (check_and_adv("news_up")) {inst_bit[2] = 1; inst_bit[1] = 1;}
  if (check_and_adv("news_down")) inst_bit[2] = 1;
  if (check_and_adv("news_left")) {inst_bit[2] = 1; inst_bit[0] = 1;}
  if (check_and_adv("news_right"))
    {inst_bit[2] = 1; inst_bit[1] = 1; inst_bit[0] = 1;}

  if (*line_ptr != '\n')
    {printf("ERROR: bogus inst: %s", line); exit(0);}

  for (index = 5; index >= 0; index--) {
    inst_bit[index + 11] = (src_addr >> index) & 1;
    inst_bit[index + 5] = (ld_addr >> index) & 1;
  }
  for (index = 19; index >= 0; index--) {
    value = value + (inst_bit[index] << index);
    fprintf(outFile, "%d", inst_bit[index]);
  }
  fprintf(outFile, "\n");
  fprintf(epromFileH, ":01%.4X00%.2X%.2X\n", pc, (value >> 16),
  ((- ((value >> 16) + 1 + (pc >> 8) + (pc & 255))) & 255));
  fprintf(epromFileM, ":01%.4X00%.2X%.2X\n", pc, ((value >> 8) & 255),
  ((- (((value >> 8) & 255) + 1 + (pc >> 8) + (pc & 255))) & 255));
  fprintf(epromFileL, ":01%.4X00%.2X%.2X\n", pc, (value & 255),
  ((- ((value & 255) + 1 + (pc >> 8) + (pc & 255))) & 255));
  pc++;
}

int check_and_adv(char *str) {
  if (strncmp(line_ptr, str, strlen(str))) return 0;
  line_ptr += strlen(str);
  return 1;
}

void second_pass() {
  fgets(line, 80, inFile);
  if (!feof(inFile)) {
    if ((strlen(line) == 1) ||
        (strncmp(line, ";", 1) == 0) ||
        (strncmp(line, "label", 5) == 0) ||
      . cntrl_inst(0) ||                                    .
jmp_inst()) second_pass();
    else {array_inst(); second_pass();}
  }
  else {
    fprintf(epromFileH, ":00000001FF\n");
    fprintf(epromFileM, ":00000001FF\n");
    fprintf(epromFileL, ":00000001FF\n");
  }
```

```
}

main (int argc,char *argv[]) {
  int index;
  argc--;
  argv++;
  if(argc != 2) {printf("Invalid number of arguments.\n"); exit(0);}
  sscanf(*argv++, "%d", &ezw_passes);
  sscanf(*argv++, "%d", &eprom_block);

  for (index = 0; index < labels; index++)
    lbl_names[index] = calloc(30, sizeof(char));
  line = calloc(80, sizeof(char));
  next_lbl = 0;

  pc = 0;
  inFile = fopen("micro.c", "r");
  first_pass();
  fclose(inFile);
  printf("%d instructions\n", pc);
  pc = eprom_block << 11;
  inFile = fopen("micro.c", "r");
  outFile = fopen("micro.dat", "w");
  epromFileH = fopen("microH.83", "w");
  epromFileM = fopen("microM.83", "w");
  epromFileL = fopen("microL.83", "w");
  second_pass();
  fclose(epromFileH);
  fclose(epromFileM);
  fclose(epromFileL);
  fclose(inFile);
  fclose(outFile);
}
```

Appendix C

Example Microcode

C.1 5 Tap Filter Microcode

```
;;; 4x4 PE, 5 tap, sblvls = 3
;;; mem[0:15]  = new frame
;;; mem[16:31] = work space, ezw flags
;;; mem[32:47] = prev frame

;;; mem[48] read only = flags:
;;;   bit   11   10    9   8   7   6   5   4   3   2   1    0
;;;       right x-odd left ?   ?   ?   ?   ?   ?  up y-odd down

;;; from reset

news = mem[48]

label new_group

init_arith_coder

reg = news
news = sub(news)
mem[32] = news
mem[33] = news
mem[34] = news
mem[35] = news
mem[36] = news
mem[37] = news
mem[38] = news
mem[39] = news
mem[40] = news
mem[41] = news
mem[42] = news
mem[43] = news
mem[44] = news
mem[45] = news
mem[46] = news
mem[47] = news
```

```
label start_frame

;;; cond = 1 during halt to keep global or low z = 0

news = news << 1
cond_load_1 = inv(news << 1)
halt_till_new_frame

;;; assemble image mean with MSB at bit 10

news = news << 1
jmp_if_not_mean_msb mean_bit2
news = concat_cond(news)

label mean_bit2
news = news << 1
jmp_if_not_mean_msb mean_bit3
news = concat_cond(news)

label mean_bit3
news = news << 1
jmp_if_not_mean_msb mean_bit4
news = concat_cond(news)

label mean_bit4
news = news << 1
jmp_if_not_mean_msb mean_bit5
news = concat_cond(news)

label mean_bit5
news = news << 1
jmp_if_not_mean_msb mean_bit6
news = concat_cond(news)

label mean_bit6
news = news << 1
jmp_if_not_mean_msb mean_bit7
news = concat_cond(news)

label mean_bit7
news = news << 1
jmp_if_not_mean_msb mean_bit8
news = concat_cond(news)

label mean_bit8
news = news << 1
jmp_if_not_mean_msb mean_done
news = concat_cond(news)

label mean_done
news = news << 1
news = news << 1
reg = news
reg = sub(news)
```

```
reg = sub(news << 1)

mem[16] = add(mem[0])
mem[17] = add(mem[1])
news = add(mem[2])
mem[18] = news
mem[19] = add(mem[3]), news_right
mem[20] = add(mem[4]), news_right
mem[21] = add(mem[5]), news_right
mem[22] = add(mem[6]), news_right
mem[23] = add(mem[7])
mem[24] = add(mem[8])
mem[25] = add(mem[9])
mem[26] = add(mem[10])
mem[27] = add(mem[11])
mem[28] = add(mem[12])
mem[29] = add(mem[13])
mem[30] = add(mem[14])
mem[31] = add(mem[15])

;;; filter lvl 0

;;; filter x

;;; left edge
cond_load_10 = mem[48] << 1

;;; row 0

;;; low pass left edge
news_if = mem[18]
reg = news
news = mem[19]
reg = addh(mem[18]),    news_right
mem[14] = sub(mem[16]), news_right
mem[15] = mem[14] >> 1, news_right
reg = mem[15] >> 1,     news_right
reg = sub(reg >> 1, mem[14])
news_if = mem[17]
reg = add(reg >> 1, news)
reg = addh(mem[17])
mem[0] = addh(mem[16])

;;; high pass left edge
reg = news
reg = addh(mem[19])
mem[14] = subh(mem[17])
news = mem[14] >> 1
reg = news >> 1
news = mem[22]
reg = sub(reg >> 1, mem[14]), news_right
reg = subh(mem[16]),          news_right
reg = sub(mem[18] >> 1),      news_right
mem[2] = addh(mem[17]),       news_right
```

```
;;; row 1

;;; low pass left edge
news_if = mem[22]
reg = news
news = mem[23]
reg = addh(mem[22]),     news_right
mem[14] = sub(mem[20]), news_right
mem[15] = mem[14] >> 1, news_right
reg = mem[15] >> 1,      news_right
reg = sub(reg >> 1, mem[14])
news_if = mem[21]
reg = add(reg >> 1, news)
reg = addh(mem[21])
mem[4] = addh(mem[20])

;;; high pass left edge
reg = news
reg = addh(mem[23])
mem[14] = subh(mem[21])
news = mem[14] >> 1
reg = news >> 1
news = mem[26]
reg = sub(reg >> 1, mem[14]), news_right
reg = subh(mem[20]),          news_right
reg = sub(mem[22] >> 1),      news_right
mem[6] = addh(mem[21]),       news_right

;;; row 2

;;; low pass left edge
news_if = mem[26]
reg = news
news = mem[27]
reg = addh(mem[26]),     news_right
mem[14] = sub(mem[24]), news_right
mem[15] = mem[14] >> 1, news_right
reg = mem[15] >> 1,      news_right
reg = sub(reg >> 1, mem[14])
news_if = mem[25]
reg = add(reg >> 1, news)
reg = addh(mem[25])
mem[8] = addh(mem[24])

;;; high pass left edge
reg = news
reg = addh(mem[27])
mem[14] = subh(mem[25])
news = mem[14] >> 1
reg = news >> 1
news = mem[30]
reg = sub(reg >> 1, mem[14]), news_right
reg = subh(mem[24]),          news_right
reg = sub(mem[26] >> 1),      news_right
mem[10] = addh(mem[25]),      news_right
```

```
;;; row 3

;;; low pass left edge
news_if = mem[30]
reg = news
news = mem[31]
reg = addh(mem[30]),     news_right
mem[14] = sub(mem[28]), news_right
mem[15] = mem[14] >> 1, news_right
reg = mem[15] >> 1,      news_right
reg = sub(reg >> 1, mem[14])
news_if = mem[29]
reg = add(reg >> 1, news)
reg = addh(mem[29])
mem[12] = addh(mem[28])

;;; high pass left edge
reg = news
reg = addh(mem[31])
mem[14] = subh(mem[29])
news = mem[14] >> 1
reg = news >> 1
news = mem[17]
reg = sub(reg >> 1, mem[14]), news_left
reg = subh(mem[28]),          news_left
reg = sub(mem[30] >> 1),      news_left
mem[14] = addh(mem[29]),      news_left

;;; right edge
cond_load_10 = mem[48] >> 1

;;; row 0

;;; high pass right edge
news_if = mem[17]
reg = news
news = mem[16]
reg = addh(mem[17]),     news_left
mem[11] = subh(mem[19]), news_left
mem[15] = mem[11] >> 1,  news_left
reg = mem[15] >> 1,      news_left
reg = sub(reg >> 1, mem[11])
reg = subh(mem[18])
news_if = mem[18]
reg = sub(news >> 1)
mem[3] = addh(mem[19])

;;; low pass right edge
reg = news
reg = addh(mem[16])
mem[11] = sub(mem[18])
news = mem[11] >> 1
reg = news >> 1
news = mem[21]
```

```
reg = sub(reg >> 1, mem[11]), news_left
reg = add(reg >> 1, mem[17]), news_left
reg = addh(mem[19]),          news_left
mem[1] = addh(mem[18]),       news_left

;;; row 1

;;; high pass right edge
news_if = mem[21]
reg = news
news = mem[20]
reg = addh(mem[21]),     news_left
mem[11] = subh(mem[23]), news_left
mem[15] = mem[11] >> 1,  news_left
reg = mem[15] >> 1,      news_left
reg = sub(reg >> 1, mem[11])
reg = subh(mem[22])
news_if = mem[22]
reg = sub(news >> 1)
mem[7] = addh(mem[23])

;;; low pass right edge
reg = news
reg = addh(mem[20])
mem[11] = sub(mem[22])
news = mem[11] >> 1
reg = news >> 1
news = mem[25]
reg = sub(reg >> 1, mem[11]), news_left
reg = add(reg >> 1, mem[21]), news_left
reg = addh(mem[23]),          news_left
mem[5] = addh(mem[22]),       news_left

;;; row 2

;;; high pass right edge
news_if = mem[25]
reg = news
news = mem[24]
reg = addh(mem[25]),     news_left
mem[18] = subh(mem[27]), news_left
mem[19] = mem[18] >> 1,  news_left
reg = mem[19] >> 1,      news_left
reg = sub(reg >> 1, mem[18])
reg = subh(mem[26])
news_if = mem[26]
reg = sub(news >> 1)
mem[11] = addh(mem[27])

;;; low pass right edge
reg = news
reg = addh(mem[24])
mem[18] = sub(mem[26])
news = mem[18] >> 1
reg = news >> 1
```

```
news = mem[29]
reg = sub(reg >> 1, mem[18]), news_left
reg = add(reg >> 1, mem[25]), news_left
reg = addh(mem[27]),          news_left
mem[9] = addh(mem[26]),       news_left

;;; row 3

;;; high pass right edge
news_if = mem[29]
reg = news
news = mem[28]
reg = addh(mem[29]),     news_left
mem[18] = subh(mem[31]), news_left
mem[19] = mem[18] >> 1,  news_left
reg = mem[19] >> 1,      news_left
reg = sub(reg >> 1, mem[18])
reg = subh(mem[30])
news_if = mem[30]
reg = sub(news >> 1)
mem[15] = addh(mem[31])

;;; low pass right edge
reg = news
reg = addh(mem[28])
mem[18] = sub(mem[30])
news = mem[18] >> 1
reg = news >> 1
news = mem[8]
reg = sub(reg >> 1, mem[18]), news_down
reg = add(reg >> 1, mem[29]), news_down
reg = addh(mem[31]),          news_down
mem[13] = addh(mem[30]),      news_down


;;; filter y

;;; up edge
cond_load_1 = mem[48] >> 1

;;; col 0

;;; low pass up edge
news_if = mem[8]
reg = news
news = mem[12]
reg = addh(mem[8]),     news_down
mem[27] = sub(mem[0]),  news_down
mem[31] = mem[27] >> 1, news_down
reg = mem[31] >> 1,     news_down
reg = sub(reg >> 1, mem[27])
news_if = mem[4]
reg = add(reg >> 1, news)
reg = addh(mem[4])
mem[16] = addh(mem[0])
```

```
;;; high pass up edge
reg = news
reg = addh(mem[12])
mem[27] = subh(mem[4])
news = mem[27] >> 1
reg = news >> 1
news = mem[9]
reg = sub(reg >> 1, mem[27]), news_down
reg = subh(mem[0]),           news_down
reg = sub(mem[8] >> 1),       news_down
mem[24] = addh(mem[4]),       news_down

;;; col 1

;;; low pass up edge
news_if = mem[9]
reg = news
news = mem[13]
reg = addh(mem[9]),     news_down
mem[27] = sub(mem[1]),  news_down
mem[31] = mem[27] >> 1, news_down
reg = mem[31] >> 1,     news_down
reg = sub(reg >> 1, mem[27])
news_if = mem[5]
reg = add(reg >> 1, news)
reg = addh(mem[5])
mem[17] = addh(mem[1])

;;; high pass up edge
reg = news
reg = addh(mem[13])
mem[27] = subh(mem[5])
news = mem[27] >> 1
reg = news >> 1
news = mem[10]
reg = sub(reg >> 1, mem[27]), news_down
reg = subh(mem[1]),           news_down
reg = sub(mem[9] >> 1),       news_down
mem[25] = addh(mem[5]),       news_down

;;; col 2

;;; low pass up edge
news_if = mem[10]
reg = news
news = mem[14]
reg = addh(mem[10]),    news_down
mem[27] = sub(mem[2]),  news_down
mem[31] = mem[27] >> 1, news_down
reg = mem[31] >> 1,     news_down
reg = sub(reg >> 1, mem[27])
news_if = mem[6]
reg = add(reg >> 1, news)
reg = addh(mem[6])
```

```
mem[18] = addh(mem[2])

;;; high pass up edge
reg = news
reg = addh(mem[14])
mem[27] = subh(mem[6])
news = mem[27] >> 1
reg = news >> 1
news = mem[11]
reg = sub(reg >> 1, mem[27]), news_down
reg = subh(mem[2]),           news_down
reg = sub(mem[10] >> 1),      news_down
mem[26] = addh(mem[6]),       news_down

;;; col 3

;;; low pass up edge
news_if = mem[11]
reg = news
news = mem[15]
reg = addh(mem[11]),     news_down
mem[27] = sub(mem[3]),   news_down
mem[31] = mem[27] >> 1,  news_down
reg = mem[31] >> 1,      news_down
reg = sub(reg >> 1, mem[27])
news_if = mem[7]
reg = add(reg >> 1, news)
reg = addh(mem[7])
mem[19] = addh(mem[3])

;;; high pass up edge
reg = news
reg = addh(mem[15])
mem[27] = subh(mem[7])
news = mem[27] >> 1
reg = news >> 1
news = mem[4]
reg = sub(reg >> 1, mem[27]), news_up
reg = subh(mem[3]),           news_up
reg = sub(mem[11] >> 1),      news_up
mem[27] = addh(mem[7]),       news_up

;;; down edge
cond_load_1 = mem[48] << 1

;;; col 0

;;; high pass down edge
news_if = mem[4]
reg = news
news = mem[0]
reg = addh(mem[4]),       news_up
mem[30] = subh(mem[12]),  news_up
mem[31] = mem[30] >> 1,   news_up
reg = mem[31] >> 1,       news_up
```

```
reg = sub(reg >> 1, mem[30])
reg = subh(mem[8])
news_if = mem[8]
reg = sub(news >> 1)
mem[28] = addh(mem[12])

;;; low pass down edge
reg = news
reg = addh(mem[0])
mem[30] = sub(mem[8])
news = mem[30] >> 1
reg = news >> 1
news = mem[5]
reg = sub(reg >> 1, mem[30]), news_up
reg = add(reg >> 1, mem[4]),  news_up
reg = addh(mem[12]),          news_up
mem[20] = addh(mem[8]),       news_up

;;; col 1

;;; high pass down edge
news_if = mem[5]
reg = news
news = mem[1]
reg = addh(mem[5]),       news_up
mem[30] = subh(mem[13]), news_up
mem[31] = mem[30] >> 1,  news_up
reg = mem[31] >> 1,       news_up
reg = sub(reg >> 1, mem[30])
reg = subh(mem[9])
news_if = mem[9]
reg = sub(news >> 1)
mem[29] = addh(mem[13])

;;; low pass down edge
reg = news
reg = addh(mem[1])
mem[30] = sub(mem[9])
news = mem[30] >> 1
reg = news >> 1
news = mem[6]
reg = sub(reg >> 1, mem[30]), news_up
reg = add(reg >> 1, mem[5]),  news_up
reg = addh(mem[13]),          news_up
mem[21] = addh(mem[9]),       news_up

;;; col 2

;;; high pass down edge
news_if = mem[6]
reg = news
news = mem[2]
reg = addh(mem[6]),      news_up
mem[8] = subh(mem[14]), news_up
mem[12] = mem[8] >> 1,  news_up
```

```
reg = mem[12] >> 1,     news_up
reg = sub(reg >> 1, mem[8])
reg = subh(mem[10])
news_if = mem[10]
reg = sub(news >> 1)
mem[30] = addh(mem[14])

;;; low pass down edge
reg = news
reg = addh(mem[2])
mem[8] = sub(mem[10])
news = mem[8] >> 1
reg = news >> 1
news = mem[7]
reg = sub(reg >> 1, mem[8]), news_up
reg = add(reg >> 1, mem[6]), news_up
reg = addh(mem[14]),         news_up
mem[22] = addh(mem[10]),     news_up

;;; col 3

;;; high pass down edge
news_if = mem[7]
reg = news
news = mem[3]
reg = addh(mem[7]),     news_up
mem[8] = subh(mem[15]), news_up
mem[12] = mem[8] >> 1,  news_up
reg = mem[12] >> 1,     news_up
reg = sub(reg >> 1, mem[8])
reg = subh(mem[11])
news_if = mem[11]
reg = sub(news >> 1)
mem[31] = addh(mem[15])

;;; low pass down edge
reg = news
reg = addh(mem[3])
mem[8] = sub(mem[11])
news = mem[8] >> 1
reg = news >> 1
news = mem[16]
reg = sub(reg >> 1, mem[8]), news_left
reg = add(reg >> 1, mem[7]), news_left
reg = addh(mem[15]),         news_left
mem[23] = addh(mem[11]),     news_left

;;; filter lvl 1

;;; filter x

;;; right edge
cond_load_10 = mem[48] >> 1
```

```
news_if = mem[16]
mem[11] = news
news = mem[20]
                    news_left
                    news_left
                    news_left
                    news_left
news_if = mem[20]
mem[15] = news
news = mem[17]
                               news_right
                               news_right
                               news_right
cond_load_10 = mem[48] << 1, news_right
;;; left edge

news_if = mem[17]
mem[10] = news
news = mem[21]
                    news_right
                    news_right
                    news_right
                    news_right
news_if = mem[21]
mem[14] = news
news = mem[16]
                    news_right
                    news_right
                    news_right
                    news_right
news_if = mem[11]
reg = news
reg = addh(mem[11])
mem[7] = sub(mem[16])
news = mem[7] >> 1
reg = news >> 1
news = mem[20]
reg = sub(reg >> 1, mem[7]),  news_right
reg = add(reg >> 1, mem[10]), news_right
reg = addh(mem[17]),          news_right
mem[8] = addh(mem[16]),       news_right

news_if = mem[15]
reg = news
reg = addh(mem[15])
mem[7] = sub(mem[20])
news = mem[7] >> 1
reg = news >> 1
news = mem[17]
reg = sub(reg >> 1, mem[7]),  news_left
reg = add(reg >> 1, mem[14]), news_left
reg = addh(mem[21]),          news_left
mem[12] = addh(mem[20]),      news_left

;;; right edge
```

```
cond_load_10 = mem[48] >> 1

news_if = mem[10]
reg = news
reg = addh(mem[10])
mem[7] = subh(mem[17])
news = mem[7] >> 1
reg = news >> 1
news = mem[21]
reg = sub(reg >> 1, mem[7]), news_left
reg = subh(mem[16]),         news_left
reg = sub(mem[11] >> 1),     news_left
mem[9] = addh(mem[17]),      news_left

news_if = mem[14]
reg = news
reg = addh(mem[14])
mem[7] = subh(mem[21])
news = mem[7] >> 1
reg = news >> 1
news = mem[8]
reg = sub(reg >> 1, mem[7]), news_up
reg = subh(mem[20]),         news_up
reg = sub(mem[15] >> 1),     news_up
mem[13] = addh(mem[21]),     news_up


;;; filter y

;;; down edge
cond_load_1 = mem[48] << 1

news_if = mem[8]
mem[14] = news
news = mem[9]
                    news_up
                    news_up
                    news_up
                    news_up
news_if = mem[9]
mem[15] = news
news = mem[12]
                              news_down
                              news_down
                              news_down
cond_load_1 = mem[48] >> 1, news_down
;;; up edge
news_if = mem[12]
mem[10] = news
news = mem[13]
                   news_down
                   news_down
                   news_down
                   news_down
news_if = mem[13]
```

```
mem[11] = news
news = mem[8]
                    news_down
                    news_down
                    news_down
                    news_down
news_if = mem[14]
reg = news
reg = addh(mem[14])
mem[7] = sub(mem[8])
news = mem[7] >> 1
reg = news >> 1
news = mem[9]
reg = sub(reg >> 1, mem[7]),  news_down
reg = add(reg >> 1, mem[10]), news_down
reg = addh(mem[12]),          news_down
mem[16] = addh(mem[8]),       news_down

news_if = mem[15]
reg = news
reg = addh(mem[15])
mem[7] = sub(mem[9])
news = mem[7] >> 1
reg = news >> 1
news = mem[12]
reg = sub(reg >> 1, mem[7]),  news_up
reg = add(reg >> 1, mem[11]), news_up
reg = addh(mem[13]),          news_up
mem[17] = addh(mem[9]),       news_up

;;; down edge
cond_load_1 = mem[48] << 1

news_if = mem[10]
reg = news
reg = addh(mem[10])
mem[7] = subh(mem[12])
news = mem[7] >> 1
reg = news >> 1
news = mem[13]
reg = sub(reg >> 1, mem[7]), news_up
reg = subh(mem[8]),          news_up
reg = sub(mem[14] >> 1),     news_up
mem[20] = addh(mem[12]),     news_up

news_if = mem[11]
reg = news
reg = addh(mem[11])
mem[7] = subh(mem[13])
news = mem[7] >> 1
reg = news >> 1
news = mem[16]
reg = sub(reg >> 1, mem[7]), news_right
reg = subh(mem[9]),          news_right
reg = sub(mem[15] >> 1),     news_right
```

```
mem[21] = addh(mem[13]),      news_right


;;; filter lvl 2

;;; filter x

mem[13] = news
news = mem[16]
                              news_left
                              news_left
                              news_left
cond_load_10 = mem[48] >> 1, news_left
;;; right edge
news_if = mem[13]
mem[14] = news,               news_left
                              news_left
                              news_left
cond_load_10 = mem[48] << 1, news_left
;;; left edge
mem[15] = news

news = mem[13]
news_if = mem[14]
mem[13] = news,     news_right
                    news_right
                    news_right
                    news_right
news_if = mem[15]
mem[12] = news
news = mem[15]

;;; right edge
cond_load_10 = mem[48] >> 1

news_if = mem[12]
mem[15] = news
reg = news

reg = addh(mem[12])
news = sub(mem[16])
mem[11] = news >> 1
reg = mem[11] >> 1
reg = sub(reg >> 1, news)
reg = add(reg >> 1, mem[13])
reg = addh(mem[14])
news = addh(mem[16])

;;; x-odd
cond_load_10 = mem[48]

reg = mem[15]
reg = addh(mem[12])
mem[10] = subh(mem[16])
mem[11] = mem[10] >> 1
```

```
reg = mem[11] >> 1
reg = sub(reg >> 1, mem[10])
reg = subh(mem[13])
reg = sub(mem[14] >> 1)
news_if = addh(mem[16])

mem[16] = news,     news_down
                    news_down
                    news_down
                    news_down

;;; filter y

mem[13] = news
news = mem[16]
                                news_up
                                news_up
                                news_up
cond_load_1 = mem[48] << 1,     news_up
;;; down edge
news_if = mem[13]
mem[14] = news,                 news_up
                                news_up
                                news_up
cond_load_1 = mem[48] >> 1,     news_up
;;; up edge
mem[15] = news
news = mem[13]
news_if = mem[14]
mem[13] = news,     news_down
                    news_down
                    news_down
                    news_down
news_if = mem[15]
mem[12] = news
news = mem[15]

;;; down edge
cond_load_1 = mem[48] << 1

news_if = mem[12]
mem[15] = news
reg = news

reg = addh(mem[12])
news = sub(mem[16])
mem[11] = news >> 1
reg = mem[11] >> 1
reg = sub(reg >> 1, news)
reg = add(reg >> 1, mem[13])
reg = addh(mem[14])
news = addh(mem[16])

;;; y-odd
cond_load_1 = mem[48]
```

```
reg = mem[15]
reg = addh(mem[12])
mem[10] = subh(mem[16])
mem[11] = mem[10] >> 1
reg = mem[11] >> 1
reg = sub(reg >> 1, mem[10])
reg = subh(mem[13])
reg = sub(mem[14] >> 1)
news_if = addh(mem[16])

mem[16] = news
reg = news

;;; compare to previous frame

;;; use mem[15] to store 0

mem[15] = sub(news)

;;; use mem[14] to store rounding bit

cond_load_1 = inv(mem[15])
mem[14] = concat_cond(mem[15])
load_counter(bottom_bit)
dec_counter
dec_counter

label rounding_bit_loop

mem[14] = mem[14] << 1
dec_counter
jmp_if_counter rounding_bit_loop

;;; lvl 3

reg = mem[16]
news = sub(mem[32])
cond_load_10 = news >> 1
reg = mem[15]
news_if = sub(news)
mem[32] = concat_cond(mem[32])
reg = mem[14]
mem[16] = add(news)

;;; lvl 2
 .                                                    .
reg = mem[17]
news = sub(mem[33])
cond_load_10 = news >> 1
reg = mem[15]
news_if = sub(news)
mem[33] = concat_cond(mem[33])
reg = mem[14]
mem[17] = add(news >> 1)
```

```
reg = mem[20]
news = sub(mem[36])
cond_load_10 = news >> 1
reg = mem[15]
news_if = sub(news)
mem[36] = concat_cond(mem[36])
reg = mem[14]
mem[20] = add(news >> 1)

reg = mem[21]
news = sub(mem[37])
cond_load_10 = news >> 1
reg = mem[15]
news_if = sub(news)
mem[37] = concat_cond(mem[37])
reg = mem[14]
mem[21] = add(news >> 1)

;;; lvl 1

reg = mem[18]
news = sub(mem[34])
cond_load_10 = news >> 1
reg = mem[15]
news_if = sub(news)
mem[34] = concat_cond(mem[34])
reg = news >> 1
mem[18] = add(reg >> 1, mem[14])

reg = mem[19]
news = sub(mem[35])
cond_load_10 = news >> 1
reg = mem[15]
news_if = sub(news)
mem[35] = concat_cond(mem[35])
reg = news >> 1
mem[19] = add(reg >> 1, mem[14])

reg = mem[22]
news = sub(mem[38])
cond_load_10 = news >> 1
reg = mem[15]
news_if = sub(news)
mem[38] = concat_cond(mem[38])
reg = news >> 1
mem[22] = add(reg >> 1, mem[14])

reg = mem[23]
news = sub(mem[39])
cond_load_10 = news >> 1
reg = mem[15]
news_if = sub(news)
mem[39] = concat_cond(mem[39])
reg = news >> 1
```

```
mem[23] = add(reg >> 1, mem[14])

reg = mem[24]
news = sub(mem[40])
cond_load_10 = news >> 1
reg = mem[15]
news_if = sub(news)
mem[40] = concat_cond(mem[40])
reg = news >> 1
mem[24] = add(reg >> 1, mem[14])

reg = mem[25]
news = sub(mem[41])
cond_load_10 = news >> 1
reg = mem[15]
news_if = sub(news)
mem[41] = concat_cond(mem[41])
reg = news >> 1
mem[25] = add(reg >> 1, mem[14])

reg = mem[26]
news = sub(mem[42])
cond_load_10 = news >> 1
reg = mem[15]
news_if = sub(news)
mem[42] = concat_cond(mem[42])
reg = news >> 1
mem[26] = add(reg >> 1, mem[14])

reg = mem[27]
news = sub(mem[43])
cond_load_10 = news >> 1
reg = mem[15]
news_if = sub(news)
mem[43] = concat_cond(mem[43])
reg = news >> 1
mem[27] = add(reg >> 1, mem[14])

reg = mem[28]
news = sub(mem[44])
cond_load_10 = news >> 1
reg = mem[15]
news_if = sub(news)
mem[44] = concat_cond(mem[44])
reg = news >> 1
mem[28] = add(reg >> 1, mem[14])

reg = mem[29]
news = sub(mem[45])
cond_load_10 = news >> 1
reg = mem[15]
news_if = sub(news)
mem[45] = concat_cond(mem[45])
reg = news >> 1
mem[29] = add(reg >> 1, mem[14])
```

```
reg = mem[30]
news = sub(mem[46])
cond_load_10 = news >> 1
reg = mem[15]
news_if = sub(news)
mem[46] = concat_cond(mem[46])
reg = news >> 1
mem[30] = add(reg >> 1, mem[14])

reg = mem[31]
news = sub(mem[47])
cond_load_10 = news >> 1
reg = mem[15]
news_if = sub(news)
mem[47] = concat_cond(mem[47])
reg = news >> 1
mem[31] = add(reg >> 1, mem[14])

;;; shift down

load_counter(bottom_bit)

label shift_down_loop

mem[16] = mem[16] >> 1
mem[17] = mem[17] >> 1
mem[18] = mem[18] >> 1
mem[19] = mem[19] >> 1
mem[20] = mem[20] >> 1
mem[21] = mem[21] >> 1
mem[22] = mem[22] >> 1
mem[23] = mem[23] >> 1
mem[24] = mem[24] >> 1
mem[25] = mem[25] >> 1
mem[26] = mem[26] >> 1
mem[27] = mem[27] >> 1
mem[28] = mem[28] >> 1
mem[29] = mem[29] >> 1
mem[30] = mem[30] >> 1
mem[31] = mem[31] >> 1
dec_counter
jmp_if_counter shift_down_loop

;;; shift up

load_counter(bottom_bit)

label shift_up_loop

mem[16] = mem[16] << 1
mem[17] = mem[17] << 1
mem[18] = mem[18] << 1
mem[19] = mem[19] << 1
mem[20] = mem[20] << 1
```

```
mem[21] = mem[21] << 1
mem[22] = mem[22] << 1
mem[23] = mem[23] << 1
mem[24] = mem[24] << 1
mem[25] = mem[25] << 1
mem[26] = mem[26] << 1
mem[27] = mem[27] << 1
mem[28] = mem[28] << 1
mem[29] = mem[29] << 1
mem[30] = mem[30] << 1
mem[31] = mem[31] << 1
dec_counter
jmp_if_counter shift_up_loop

;;; new prev frame and prep for ezw

;;; lvl 3

reg = mem[32] >> 1
cond_load_1 = mem[32] << 1
news = add(mem[16] >> 1)
news_if = sub(mem[16] >> 1)
mem[32] = news << 1
news = mem[16] >> 1
news = concat_cond(news)
mem[16] = news << 1

;;; lvl 2

reg = mem[33] >> 1
cond_load_1 = mem[33] << 1
news = add(mem[17])
news_if = sub(mem[17])
mem[33] = news << 1
news = concat_cond(mem[17])
mem[17] = news << 1

reg = mem[36] >> 1
cond_load_1 = mem[36] << 1
news = add(mem[20])
news_if = sub(mem[20])
mem[36] = news << 1
news = concat_cond(mem[20])
mem[20] = news << 1

reg = mem[37] >> 1
cond_load_1 = mem[37] << 1
news = add(mem[21])
news_if = sub(mem[21])
mem[37] = news << 1
news = concat_cond(mem[21])
mem[21] = news << 1

;;; lvl 1
```

```
reg = mem[34] >> 1
cond_load_1 = mem[34] << 1
news = add(mem[18] << 1)
news_if = sub(mem[18] << 1)
mem[34] = news << 1
news = mem[18] << 1
news = concat_cond(news)
mem[18] = news << 1

reg = mem[35] >> 1
cond_load_1 = mem[35] << 1
news = add(mem[19] << 1)
news_if = sub(mem[19] << 1)
mem[35] = news << 1
news = mem[19] << 1
news = concat_cond(news)
mem[19] = news << 1

reg = mem[38] >> 1
cond_load_1 = mem[38] << 1
news = add(mem[22] << 1)
news_if = sub(mem[22] << 1)
mem[38] = news << 1
news = mem[22] << 1
news = concat_cond(news)
mem[22] = news << 1

reg = mem[39] >> 1
cond_load_1 = mem[39] << 1
news = add(mem[23] << 1)
news_if = sub(mem[23] << 1)
mem[39] = news << 1
news = mem[23] << 1
news = concat_cond(news)
mem[23] = news << 1

reg = mem[40] >> 1
cond_load_1 = mem[40] << 1
news = add(mem[24] << 1)
news_if = sub(mem[24] << 1)
mem[40] = news << 1
news = mem[24] << 1
news = concat_cond(news)
mem[24] = news << 1

reg = mem[41] >> 1
cond_load_1 = mem[41] << 1
news = add(mem[25] << 1)
news_if = sub(mem[25] << 1)
mem[41] = news << 1
news = mem[25] << 1
news = concat_cond(news)
mem[25] = news << 1

reg = mem[42] >> 1
```

```
cond_load_1 = mem[42] << 1
news = add(mem[26] << 1)
news_if = sub(mem[26] << 1)
mem[42] = news << 1
news = mem[26] << 1
news = concat_cond(news)
mem[26] = news << 1

reg = mem[43] >> 1
cond_load_1 = mem[43] << 1
news = add(mem[27] << 1)
news_if = sub(mem[27] << 1)
mem[43] = news << 1
news = mem[27] << 1
news = concat_cond(news)
mem[27] = news << 1

reg = mem[44] >> 1
cond_load_1 = mem[44] << 1
news = add(mem[28] << 1)
news_if = sub(mem[28] << 1)
mem[44] = news << 1
news = mem[28] << 1
news = concat_cond(news)
mem[28] = news << 1

reg = mem[45] >> 1
cond_load_1 = mem[45] << 1
news = add(mem[29] << 1)
news_if = sub(mem[29] << 1)
mem[45] = news << 1
news = mem[29] << 1
news = concat_cond(news)
mem[29] = news << 1

reg = mem[46] >> 1
cond_load_1 = mem[46] << 1
news = add(mem[30] << 1)
news_if = sub(mem[30] << 1)
mem[46] = news << 1
news = mem[30] << 1
news = concat_cond(news)
mem[30] = news << 1

reg = mem[47] >> 1
cond_load_1 = mem[47] << 1
news = add(mem[31]. << 1)
news_if = sub(mem[31] << 1)
mem[47] = news << 1
news = mem[31] << 1
news = concat_cond(news)
mem[31] = news << 1

;;; EZW
```

```
;;; check for top_bit at pass1

cond_load_1 = mem[16] >> 1
pre_top_bit
cond_load_10 = mem[16]
jmp_if_not_top_bit top_bit_pass2

;;; pass 1 - sblvl 3 only

;;; up pass

cond_load_10 = mem[16]
cond_and_10 = inv(mem[16] >> 1)
news = concat_cond(mem[16])
mem[16] = news, news_left
                news_left
                news_left
                news_left
cond_or_1 = news << 1
news = concat_cond(news)
                news_up
                news_up
                news_up
                news_up
cond_or_1 = news << 1
news = concat_cond(mem[16])

;;; down pass

;;; send lvl 3 LL

init_arith_tables
set_zero_possible

;;; not x-even * y-even
cond_load_10 = mem[48]
cond_or_1 = mem[48]

reg = news
news_if = sub(news)
reg = news
news = concat_cond(news)
news = ezw(news)
news_if = mem[16]
mem[16] = news
do_ezw_slot

;;; prep for the rest of lvl 3 and distribute pot from lvl 3 LL

init_arith_tables
clr_zero_possible

;;; x-even * y-even
cond_load_1 = inv(mem[48])
cond_and_10 = inv(mem[48])
```

```
reg = news
news_if = sub(news)
reg = news
news = mem[16]
                news_down
                news_down
                news_down
                news_down
news_if = mem[16]
mem[16] = news
                              news_right
                              news_right
                              news_right
cond_load_10 = inv(mem[48]), news_right
;;; x-even

news_if = mem[16]

;;; x-even * y-even
cond_load_1 = inv(mem[48])
cond_and_10 = inv(mem[48])

news_if = concat_cond(news)
news = ezw(news)
news_if = mem[16]
mem[16] = news
do_ezw_slot

load_counter(ezw_passes)
jmp_if_counter ezw_pass2

;;; pass 2 - sblvl 3 and 2 only

label top_bit_pass2

cond_load_1 = mem[16] >> 1
pre_top_bit
reg = mem[16]
mem[16] = ezw(news)
cond_load_10 = mem[16]
cond_load_10 = mem[17]
cond_load_10 = mem[20]
cond_load_10 = mem[21]
jmp_if_not_top_bit top_bit_pass3

;;; up pass

label ezw_pass2

cond_load_10 = mem[16]
cond_and_10 = inv(mem[16] >> 1)
mem[16] = concat_cond(mem[16])

;;; lvl 2 HL and up to lvl 3
```

```
cond_load_10 = mem[17]
cond_and_10 = inv(mem[17] >> 1)
news = concat_cond(mem[17])
mem[17] = news, news_right
                news_right
                news_right
                news_right
cond_or_1 = news << 1
news = concat_cond(news)
                          news_up
                          news_up
                          news_up
cond_or_1 = mem[16] << 1, news_up
cond_or_1 = news << 1
news = concat_cond(mem[16])

;;; not x-odd * y-even
cond_load_10 = inv(mem[48])
cond_or_1 = mem[48]

news_if = mem[16]
mem[16] = news

;;; lvl 2 LH and up to lvl 3

cond_load_10 = mem[20]
cond_and_10 = inv(mem[20] >> 1)
news = concat_cond(mem[20])
mem[20] = news, news_left
                news_left
                news_left
                news_left
cond_or_1 = news << 1
news = concat_cond(news)
                          news_down
                          news_down
                          news_down
cond_or_1 = mem[16] << 1, news_down
cond_or_1 = news << 1
news = concat_cond(mem[16])

;;; not x-even * y-odd
cond_load_10 = mem[48]
cond_or_1 = inv(mem[48])

news_if = mem[16]
mem[16] = news

;;; lvl 2 HH and up to lvl 3

cond_load_10 = mem[21]
cond_and_10 = inv(mem[21] >> 1)
news = concat_cond(mem[21])
mem[21] = news, news_right
```

```
                news_right
                news_right
                news_right
cond_or_1 = news << 1
news = concat_cond(news)
                          news_down
                          news_down
                          news_down
cond_or_1 = mem[16] << 1, news_down
cond_or_1 = news << 1
news = concat_cond(mem[16])

;;; not x-odd * y-odd
cond_load_10 = inv(mem[48])
cond_or_1 = inv(mem[48])

news_if = mem[16]

;;; over to lvl 3 LL

mem[16] = news,     news_left
cond_load_1 = news, news_left
                    news_left
                    news_left
cond_or_1 = news << 1
news = concat_cond(news)
                news_up
                news_up
                news_up
                news_up
cond_or_1 = news << 1
news = concat_cond(mem[16])

;;; down pass

;;; send lvl 3 LL

init_arith_tables
set_zero_possible

;;; not x-even * y-even
cond_load_10 = mem[48]
cond_or_1 = mem[48]

reg = news
news_if = sub(news)
reg = news
news = concat_cond(news)
news = ezw(news)
news_if = mem[16]
mem[16] = news
do_ezw_slot

;;; prep for the rest of lvl 3 and distribute pot from lvl 3 LL
```

```
init_arith_tables

;;; x-even * y-even
cond_load_1 = inv(mem[48])
cond_and_10 = inv(mem[48])

reg = news
news_if = sub(news)
reg = news
news = mem[16]
                news_down
                news_down
                news_down
                news_down
news_if = mem[16]
mem[16] = news
                              news_right
                              news_right
                              news_right
cond_load_10 = inv(mem[48]), news_right
;;; x-even

news_if = mem[16]

;;; x-even * y-even
cond_load_1 = inv(mem[48])
cond_and_10 = inv(mem[48])

news_if = concat_cond(news)
news = ezw(news)
news_if = mem[16]
do_ezw_slot

init_arith_tables
clr_zero_possible

;;; lvl 2 HL

mem[16] = news,              news_left
reg = mem[17],               news_left
cond_load_1 = inv(mem[48]), news_left
cond_and_10 = mem[48],       news_left
;;; x-odd * y-even

news_if = mem[16]
mem[17] = news,              news_down
                             news_down
                             news_down
cond_load_1 = inv(mem[48]), news_down
;;; y-even

news_if = mem[17]
mem[17] = ezw(news)
do_ezw_slot
```

```
;;; lvl 2 LH

news = mem[16]
reg = mem[20],              news_right
                            news_right
cond_load_1 = mem[48],      news_right
cond_and_10 = inv(mem[48]), news_right
;;; x-even * y-odd

news_if = mem[16]
mem[20] = news,         news_up
                        news_up
                        news_up
cond_load_1 = mem[48], news_up
;;; y-odd

news_if = mem[20]
mem[20] = ezw(news)
do_ezw_slot

;;; lvl 2 HH

news = mem[16]
reg = mem[21],          news_left
                        news_left
cond_load_1 = mem[48], news_left
cond_and_10 = mem[48], news_left
;;; x-odd * y-odd

news_if = mem[16]
mem[21] = news,         news_up
                        news_up
                        news_up
cond_load_1 = mem[48], news_up
;;; y-odd

news_if = mem[21]
mem[21] = ezw(news)
do_ezw_slot

;;; All other passes

load_counter(ezw_passes)
jmp_if_counter ezw_loop

label top_bit_pass3

load_counter(ezw_passes)

label top_bit_loop

cond_load_1 = mem[16] >> 1
pre_top_bit
reg = mem[16]
mem[16] = ezw(news)
```

```
cond_load_10 = mem[16]
reg = mem[17]
mem[17] = ezw(news)
cond_load_10 = mem[17]
reg = mem[20]
mem[20] = ezw(news)
cond_load_10 = mem[20]
reg = mem[21]
mem[21] = ezw(news)
cond_load_10 = mem[21]
cond_load_10 = mem[18]
cond_load_10 = mem[19]
cond_load_10 = mem[22]
cond_load_10 = mem[23]
cond_load_10 = mem[24]
cond_load_10 = mem[25]
cond_load_10 = mem[28]
cond_load_10 = mem[29]
cond_load_10 = mem[26]
cond_load_10 = mem[27]
cond_load_10 = mem[30]
cond_load_10 = mem[31]
jmp_if_not_top_bit top_bit_pass_n

label ezw_loop

dec_counter

jmp_if_counter not_last_pass
clr_sub_OK_bit
label not_last_pass

;;; up pass

cond_load_10 = mem[16]
cond_and_10 = inv(mem[16] >> 1)
mem[16] = concat_cond(mem[16])

;;; HL and up to lvl 3

cond_load_10 = mem[18]
cond_and_10 = inv(mem[18] >> 1)
mem[18] = concat_cond(mem[18])

cond_load_10 = mem[19]
cond_and_10 = inv(mem[19] >> 1)
mem[19] = concat_cond(mem[19])

cond_load_10 = mem[22]
cond_and_10 = inv(mem[22] >> 1)
mem[22] = concat_cond(mem[22])

cond_load_10 = mem[23]
cond_and_10 = inv(mem[23] >> 1)
mem[23] = concat_cond(mem[23])
```

```
cond_load_10 = mem[17]
cond_and_10 = inv(mem[17] >> 1)
cond_or_1 = mem[18] << 1
cond_or_1 = mem[19] << 1
cond_or_1 = mem[22] << 1
cond_or_1 = mem[23] << 1
news = concat_cond(mem[17])
mem[17] = news, news_right
                news_right
                news_right
                news_right
cond_or_1 = news << 1
news = concat_cond(news)
                           news_up
                           news_up
                           news_up
cond_or_1 = mem[16] << 1, news_up
cond_or_1 = news << 1
news = concat_cond(mem[16])

;;; not x-odd * y-even
cond_load_10 = inv(mem[48])
cond_or_1 = mem[48]

news_if = mem[16]
mem[16] = news

;;; LH and up to lvl 3

cond_load_10 = mem[24]
cond_and_10 = inv(mem[24] >> 1)
mem[24] = concat_cond(mem[24])

cond_load_10 = mem[25]
cond_and_10 = inv(mem[25] >> 1)
mem[25] = concat_cond(mem[25])

cond_load_10 = mem[28]
cond_and_10 = inv(mem[28] >> 1)
mem[28] = concat_cond(mem[28])

cond_load_10 = mem[29]
cond_and_10 = inv(mem[29] >> 1)
mem[29] = concat_cond(mem[29])

cond_load_10 = mem[20]
cond_and_10 = inv(mem[20] >> 1)
cond_or_1 = mem[24] << 1
cond_or_1 = mem[25] << 1
cond_or_1 = mem[28] << 1
cond_or_1 = mem[29] << 1
news = concat_cond(mem[20])
mem[20] = news, news_left
                news_left
```

```
                news_left
                news_left
cond_or_1 = news << 1
news = concat_cond(news)
                           news_down
                           news_down
                           news_down
cond_or_1 = mem[16] << 1, news_down
cond_or_1 = news << 1
news = concat_cond(mem[16])

;;; not x-even * y-odd
cond_load_10 = mem[48]
cond_or_1 = inv(mem[48])

news_if = mem[16]
mem[16] = news

;;; HH and up to lvl 3

cond_load_10 = mem[26]
cond_and_10 = inv(mem[26] >> 1)
mem[26] = concat_cond(mem[26])

cond_load_10 = mem[27]
cond_and_10 = inv(mem[27] >> 1)
mem[27] = concat_cond(mem[27])

cond_load_10 = mem[30]
cond_and_10 = inv(mem[30] >> 1)
mem[30] = concat_cond(mem[30])

cond_load_10 = mem[31]
cond_and_10 = inv(mem[31] >> 1)
mem[31] = concat_cond(mem[31])

cond_load_10 = mem[21]
cond_and_10 = inv(mem[21] >> 1)
cond_or_1 = mem[26] << 1
cond_or_1 = mem[27] << 1
cond_or_1 = mem[30] << 1
cond_or_1 = mem[31] << 1
news = concat_cond(mem[21])
mem[21] = news, news_right
                news_right
                news_right
               .news_right
cond_or_1 = news << 1
news = concat_cond(news)
                           news_down
                           news_down
                           news_down
cond_or_1 = mem[16] << 1, news_down
cond_or_1 = news << 1
news = concat_cond(mem[16])
```

```
;;; not x-odd * y-odd
cond_load_10 = inv(mem[48])
cond_or_1 = inv(mem[48])

news_if = mem[16]

;;; over to lvl 3 LL

mem[16] = news,      news_left
cond_load_1 = news, news_left
                     news_left
                     news_left
cond_or_1 = news << 1
news = concat_cond(news)
                 news_up
                 news_up
                 news_up
                 news_up
cond_or_1 = news << 1
news = concat_cond(mem[16])

;;; down pass

;;; send lvl 3 LL

init_arith_tables
set_zero_possible

;;; not x-even * y-even
cond_load_10 = mem[48]
cond_or_1 = mem[48]

reg = news
news_if = sub(news)
reg = news
news = concat_cond(news)
news = ezw(news)
news_if = mem[16]
mem[16] = news
do_ezw_slot

;;; prep for the rest of lvl 3 and distribute pot from lvl 3 LL

init_arith_tables

;;; x-even * y-even      .
cond_load_1 = inv(mem[48])
cond_and_10 = inv(mem[48])

reg = news
news_if = sub(news)
reg = news
news = mem[16]
                 news_down
```

```
                news_down
                news_down
                news_down
news_if = mem[16]
mem[16] = news
                              news_right
                              news_right
                              news_right
cond_load_10 = inv(mem[48]), news_right
;;; x-even

news_if = mem[16]

;;; x-even * y-even
cond_load_1 = inv(mem[48])
cond_and_10 = inv(mem[48])

news_if = concat_cond(news)
news = ezw(news)
news_if = mem[16]
do_ezw_slot

init_arith_tables

;;; lvl 2 HL

mem[16] = news,             news_left
reg = mem[17],              news_left
cond_load_1 = inv(mem[48]), news_left
cond_and_10 = mem[48],      news_left
;;; x-odd * y-even

news_if = mem[16]
mem[17] = news,             news_down
                            news_down
                            news_down
cond_load_1 = inv(mem[48]), news_down
;;; y-even

news_if = mem[17]
mem[17] = ezw(news)
do_ezw_slot

;;; lvl 2 LH

news = mem[16]
reg = mem[20],              news_right
                            news_right
cond_load_1 = mem[48],      news_right
cond_and_10 = inv(mem[48]), news_right
;;; x-even * y-odd

news_if = mem[16]
mem[20] = news,        news_up
                       news_up
```

```
                        news_up
cond_load_1 = mem[48], news_up
;;; y-odd

news_if = mem[20]
mem[20] = ezw(news)
do_ezw_slot

;;; lvl 2 HH

news = mem[16]
reg = mem[21],          news_left
                        news_left
cond_load_1 = mem[48], news_left
cond_and_10 = mem[48], news_left
;;; x-odd * y-odd

news_if = mem[16]
mem[21] = news,         news_up
                        news_up
                        news_up
cond_load_1 = mem[48], news_up
;;; y-odd

news_if = mem[21]
mem[21] = ezw(news)
do_ezw_slot

init_arith_tables
clr_zero_possible

;;; lvl 3 and some bank0 dram refresh

mem[24] = mem[24]
mem[25] = mem[25]
mem[28] = mem[28]
mem[29] = mem[29]

;;; lvl 3 HL

reg = mem[18]
mem[18] = ezw(mem[17])
do_ezw_slot

reg = mem[19]
mem[19] = ezw(mem[17])
do_ezw_slot

mem[26] = mem[26]
mem[27] = mem[27]
mem[30] = mem[30]
mem[31] = mem[31]

reg = mem[22]
mem[22] = ezw(mem[17])
```

```
do_ezw_slot

reg = mem[23]
mem[23] = ezw(mem[17])
do_ezw_slot

mem[16] = mem[16]
mem[17] = mem[17]
mem[20] = mem[20]
mem[21] = mem[21]

;;; lvl 3 LH

reg = mem[24]
mem[24] = ezw(mem[20])
do_ezw_slot

reg = mem[25]
mem[25] = ezw(mem[20])
do_ezw_slot

reg = mem[28]
mem[28] = ezw(mem[20])
do_ezw_slot

reg = mem[29]
mem[29] = ezw(mem[20])
do_ezw_slot

mem[18] = mem[18]
mem[19] = mem[19]
mem[22] = mem[22]
mem[23] = mem[23]

;;; lvl 3 HH

reg = mem[26]
mem[26] = ezw(mem[21])
do_ezw_slot

reg = mem[27]
mem[27] = ezw(mem[21])
do_ezw_slot

reg = mem[30]
mem[30] = ezw(mem[21])
do_ezw_slot.

reg = mem[31]
mem[31] = ezw(mem[21])
do_ezw_slot

jmp_if_counter ezw_loop

set_sub_OK_bit
```

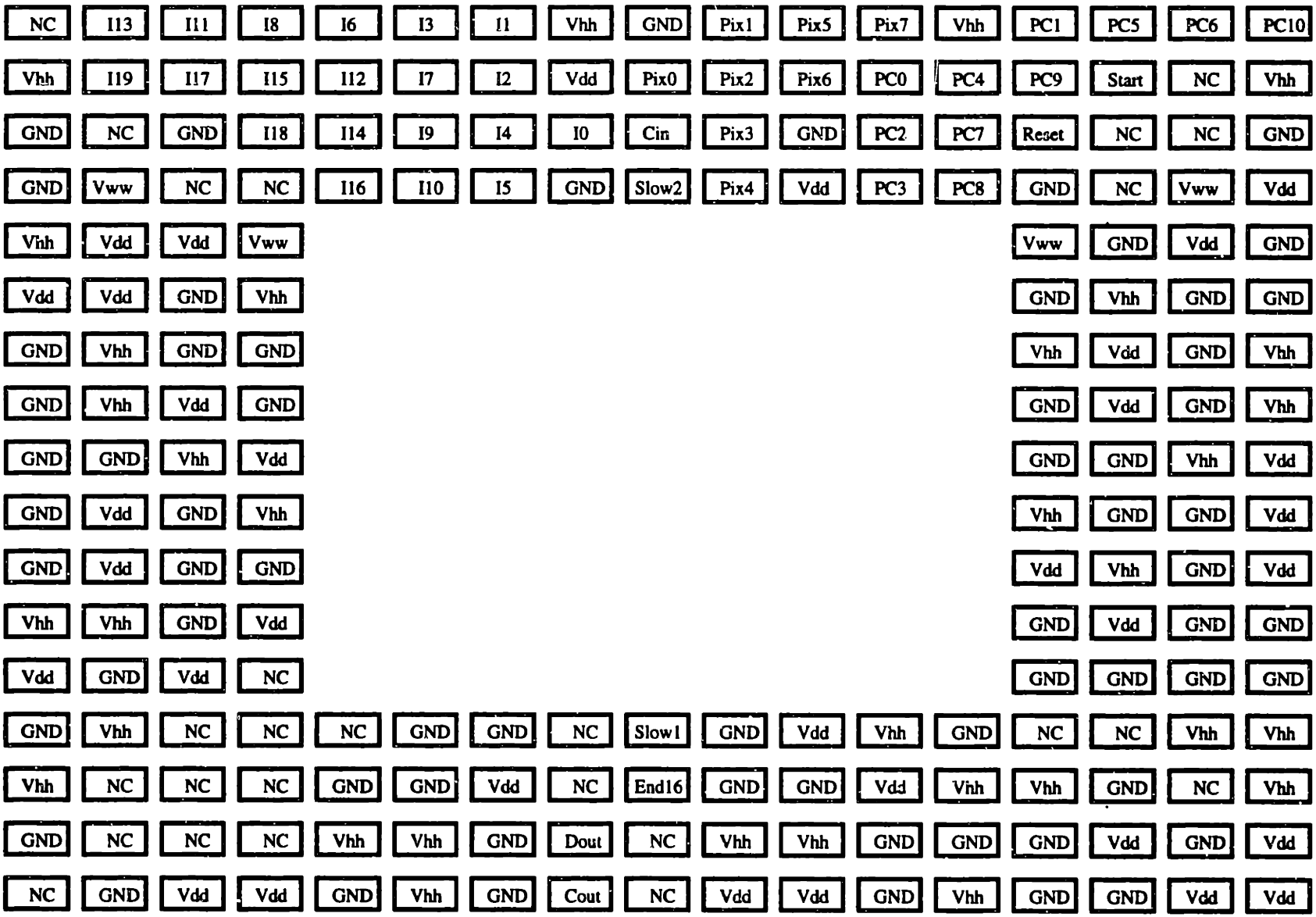
```
jmp_if_not_mean_msb mean7
label mean7
jmp_if_not_mean_msb mean6
label mean6
jmp_if_not_mean_msb mean5
label mean5
jmp_if_not_mean_msb mean4
label mean4
jmp_if_not_mean_msb mean3
label mean3
jmp_if_not_mean_msb mean2
label mean2
jmp_if_not_mean_msb mean1
label mean1
jmp_if_not_mean_msb mean0
label mean0

jmp_if_new_group new_group

load_counter(ezw_passes)
jmp_if_counter start_frame

label top_bit_pass_n

reg = mem[18]
mem[18] = ezw(news)
reg = mem[19]
mem[19] = ezw(news)
reg = mem[22]
mem[22] = ezw(news)
reg = mem[23]
mem[23] = ezw(news)
reg = mem[24]
mem[24] = ezw(news)
reg = mem[25]
mem[25] = ezw(news)
reg = mem[28]
mem[28] = ezw(news)
reg = mem[29]
mem[29] = ezw(news)
reg = mem[26]
mem[26] = ezw(news)
reg = mem[27]
mem[27] = ezw(news)
reg = mem[30]
mem[30] = ezw(news)
reg = mem[31]
mem[31] = ezw(news)

dec_counter

jmp_if_counter top_bit_loop

jmp_if_new_group new_group
```

```
load_counter(ezw_passes)
jmp_if_counter start_frame
```

Appendix D

Chip I/O Pins

The next page shows a diagram of the test chip's pins. *Slow*1 slows the clock frequency of the arithmetic coder by about 15%, if asserted. The signal *Slow*2 does the same for the relative timing differences generated by the sequencer for PE controls. The other pin labels are self explanatory.


NC I13 I11 I8 I6 I3 I1 Vhh GND Pix1 Pix5 Pix7 Vhh PC1 PC5 PC6 PC10
Vhh I19 I17 I15 I12 I7 I2 Vdd Pix0 Pix2 Pix6 PC0 PC4 PC9 Start NC Vhh
GND NC GND I18 I14 I9 I4 I0 Cin Pix3 GND PC2 PC7 Reset NC NC GND
GND Vww NC NC I16 I10 I5 GND Slow2 Pix4 Vdd PC3 PC8 GND NC Vww Vdd
Vhh Vdd Vdd Vww Vww GND Vdd GND
Vdd Vdd GND Vhh GND Vhh GND GND
GND Vhh GND GND Vhh Vdd GND Vhh
GND Vhh Vdd GND GND Vdd GND Vhh
GND GND Vhh Vdd GND GND Vhh Vdd
GND Vdd GND Vhh Vhh GND GND Vdd
GND Vdd GND GND Vdd Vhh GND Vdd
Vhh Vhh GND Vdd GND Vdd GND GND
Vdd GND Vdd NC GND GND GND GND
GND Vhh NC NC NC GND GND NC Slow1 GND Vdd Vhh GND NC NC Vhh Vhh
Vhh NC NC NC GND GND Vdd NC End16 GND GND Vdd Vhh Vhh GND NC Vhh
GND NC NC NC Vhh Vhh GND Dout NC Vhh Vhh GND GND GND Vdd GND Vdd
NC GND Vdd Vdd GND Vhh GND Cout NC Vdd Vdd GND Vhh GND GND Vdd Vdd