Security Sphere: Digital Image Processing

by

Matthew James Yarosz

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degrees of
Bachelor of Science in Electrical Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science
at the Massachusetts Institute of Technology
May 23, 2001

Copyright 2001 Massachusetts Institute of Technology. All rights reserved.
Abstract

This thesis presents an intelligent motion detection algorithm and a real-time video compression scheme for use in a proposed wireless, panoramic surveillance device named the Security Sphere. The motion detection algorithm recognizes repetitive motion and ignores the motion of small or narrow objects in an attempt to trigger only upon human motion. A wireless surveillance device equipped with this motion detection feature could save power, issue alerts, or eliminate the need for a human monitor of the video output. The algorithm is implemented on a TMS320C6211 digital signal processor. The video compression reduces the video data rate for transmission of the data over a 2.5-Mb/s wireless channel. The video compression scheme performs wavelet-transform based intraframe compression independently on each frame of video. The compression is implemented through the use of the ADV601 wavelet compression chip. A demonstration platform was developed that tests the motion detection and compression features using the output from a 60-frame-per-second digital video camera. The compressed and subsequently uncompressed video output is displayed on a NTSC monitor.

Thesis Supervisor: Professor Charles G. Sodini
Title: Professor of Electrical Engineering and Computer Science
Acknowledgements

Many people have helped to make this thesis possible and to make the past year more enjoyable. I thank Professor Charles Sodini for supporting this project, which proved to be a more valuable learning experience than I could have anticipated. I thank him also for setting aside time for weekly meetings that provided encouragement and guidance throughout. Furthermore, Professor Sodini organized the classes 6.191 and 6.192 that significantly eased my search and preparation for a thesis project.

Sam Lefian was my safety net for those times when something just wouldn't work. Sam offered the suggestions that would point me in the right direction and guide me to the actual cause of various malfunctions. Sam also freely bestowed his wealth of practical knowledge, which frequently helped me when I was doing things the hard way.

I thank my thesis partners, Jelena Madic and Oluwamuyiwa Olubuyide, with whom I shared the experience of the past year. Not only did they provide technical assistance and advice, but their camaraderie was especially helpful during some of those late nights.

Victor Lum, formerly of MIT's Artificial Intelligence Laboratory, ended my search for a motion detection algorithm by introducing me to the method used in this project for repetitive motion rejection. Keith Fife, a former Master of Engineering Student, created and well documented the digital video camera that proved invaluable to the testing and demonstration of this project. Texas Instruments assisted the making of this project with the donation of software, an evaluation board, and several components.

My parents have earned my gratitude for their words of support and assurance during weekly telephone calls.
# Table of Contents

Introduction .................................................................................................................................................. 11
1.1. Overview ............................................................................................................................................. 11
1.2. Motion Detection for Surveillance ....................................................................................................... 13
1.3. Video Compression for Wireless Surveillance ...................................................................................... 14
1.4. Combining Motion Detection with Compression .................................................................................. 15
1.5. System Description and Performance .................................................................................................. 16

Compression Background .......................................................................................................................... 19
2.1. Compression Fundamentals .................................................................................................................. 19
2.2. Transform-Based Image Compression ..................................................................................................... 19
  2.2.1. A Change of Basis .............................................................................................................................. 19
  2.2.2. Subimage-by-Subimage Compression ................................................................................................. 22
  2.2.3. The Lossy Step: Quantization ............................................................................................................ 23
  2.2.4. Popular Image Compression Techniques .......................................................................................... 24
2.3. Interframe Video Compression ............................................................................................................. 25

The Video Compression Feature .................................................................................................................. 27
3.1. Selection of a Video Compression Solution .......................................................................................... 27
3.2. ADV601 Wavelet Video Compression .................................................................................................... 27
3.3. Video Compression Circuit Functional Description .............................................................................. 30
  3.3.1. Conversion to the ITU-R BT.656 Format ............................................................................................. 30
  3.3.2. ADV601 and Supporting Components ............................................................................................. 32

The Motion Detection Feature ..................................................................................................................... 35
4.1. Motion Detection Algorithm Development ............................................................................................ 35
  4.1.1. Algorithm Search and Selection ......................................................................................................... 35
  4.1.2. Detailed Algorithm Description ......................................................................................................... 35
  4.1.3. Verification of Algorithm Functionality ............................................................................................... 39
4.2. Motion Detection Circuit Description .................................................................................................... 40
4.3. Motion Detection Hardware/Software Unification .............................................................................. 42

Testing and Demonstration .......................................................................................................................... 47
5.1. Overview .............................................................................................................................................. 47
5.2. Motion Detection Testing ....................................................................................................................... 47
5.3. Format Conversion Testing .................................................................................................................... 49
5.4. Motion Enabled Output .......................................................................................................................... 51
5.5. Results .................................................................................................................................................. 51

Conclusion ................................................................................................................................................. 55
6.1. Summary ............................................................................................................................................. 55
6.2. Future Work ....................................................................................................................................... 55

Code ......................................................................................................................................................... 57
A.1. VHDL Video Format Conversion Code ................................................................................................. 57
A.2. Motion Detection Code .......................................................................................................................... 58
  A.2.2. Motion Detection Algorithm C Code ................................................................................................... 58
  A.2.2. TMS320C6211 Boot and Configuration Assembly Code .................................................................. 60
  A.2.3. TMS320C6211 Linker Command File ................................................................................................. 61
  A.2.4. TMS320C6211 Hex Conversion Utility Command File .................................................................... 61
  A.2.5. Matlab Version of Motion Detection Algorithm ............................................................................... 62

Schematics ................................................................................................................................................. 65
List of Figures

Figure 1: The Security Sphere ................................................................. 12
Figure 2: The intensity basis ................................................................. 20
Figure 3: A scaled Fourier basis ........................................................... 21
Figure 4: A highly specific basis ........................................................... 22
Figure 5: Run-length coding example .................................................. 23
Figure 6: ADV601 functional block diagram ....................................... 28
Figure 7: Original image (top) and its wavelet transform (bottom) ....... 29
Figure 8: A line of ITU-R BT.656 formatted video ................................ 31
Figure 9: The format converter ............................................................ 33
Figure 10: Stage 1 of the motion detection algorithm ......................... 36
Figure 11: Sample Stage 1 output (human, bird, and tree limb) .......... 37
Figure 12: Stage 2 of the motion detection algorithm ......................... 38
Figure 13: TMS320C6211 software development flow ....................... 42
Figure 14: The TMS320C6211 memory map ....................................... 44
Figure 15: Testing and demonstration platform .................................... 48
Figure 16: Video output at a 2.5 Mb/s compressed data rate ............... 52
Figure 17: Algorithm initially triggers on metronome (top). Repetitive motion is recognized and output turns off (bottom) ....... 53
Chapter 1.

Introduction

1.1. Overview

Video surveillance usage has proliferated in recent years as camera costs decline and as corporations and individuals look to exploit new technologies to protect themselves and their investments. Video surveillance cameras most often appear in retail outlets to discourage shoplifting. Increasingly, homeowners are gaining access to security cameras to monitor their houses when they travel. Governmental organizations, too, are investigating and experimenting with the use of security cameras for everything from discouraging crime in downtown parks to enforcing traffic laws. But despite their wide applicability, most video surveillance devices are ill-equipped to properly handle even simple monitoring and security applications.

First, most security cameras fail to capture a full picture of their surroundings due to their limited field of view. The camera points at the most likely area of interest, but activity beyond this area goes entirely unnoticed. Some cameras feature mechanical rotation and tilt to compensate for the limited angle of view, but still, at any given time, the cameras monitor only a small portion of their surroundings. Installing multiple cameras serves as another option to solve this problem. For example, mounting cameras in each corner of a room provides total coverage of the activity in that room.

However, lack of portability, a second inadequacy of many surveillance devices, can complicate the implementation of this option. A typical video security system relies on a physical connection between the video camera and the screen upon which the video is displayed. For each camera in the system, another wire must be routed, which might be unsightly or, worse, prohibitively difficult. The need for a local power supply further hinders portability of many security cameras. For outdoor security applications, a power outlet may be unavailable for use by the camera.

A third inadequacy exhibited by nearly all surveillance systems is a lack of intelligence. Security cameras do not discern between interesting and uninteresting activity within their field of view. Most do not discern even between activity and no activity at all. Small businesses and
corporations who install video surveillance systems find the equipment cheap; their major expense lies in hiring a guard to watch the screen on which the video is displayed. Perhaps unchanging for hours, the screen nonetheless demands the attention of the guard at all times. To combat this problem, a few security cameras offer simple detection of motion; however, this feature fails when persistent movement such as machinery or swaying foliage occupies the camera's field of view.

The Security Sphere, a video surveillance device conceived by Jelena Madic, Oluwamuyiwa Olubuyide, and this author, solves or alleviates these problems associated with conventional security cameras. The Security Sphere, shown in Figure 1, employs four cameras arranged on the lower half of a 16-cm radius sphere to capture 180-degree panoramic video of its surroundings. Dispensing with the physical connection between the Sphere’s cameras and its display unit, the Security Sphere instead compresses the captured video data in real time and wirelessly sends the compressed data to a display unit using a power-saving transmitter. The Security Sphere also features an intelligent motion detection algorithm that takes large steps toward the goal of distinguishing between human and non-human motion. This thesis project implements and demonstrates the motion detection algorithm and the compression scheme intended for integration with the Security Sphere.

![Figure 1: The Security Sphere](image.png)
1.2. Motion Detection for Surveillance

Those who install surveillance systems most commonly want to monitor human activity and movement. Homeowners install security cameras to alert them of the presence of intruders or vandals. Some vacationing parents hope to reveal the activities of the babysitter in their absence. Retailers rely on security cameras to deter or to detect shoplifters. In these and many other applications, the surveillance device must function only in the presence of human movement.

Motion detection features, therefore, sometimes accompany surveillance devices to add to their utility. Battery-powered cameras that are motion-enabled can operate much longer than their non-motion-enabled counterparts. Rather than paying a guard to look for motion on a monitor, companies can save money by installing security cameras that trigger an alarm upon detection of motion.

But simple motion detection often falls short in real-life instances. The swinging pendulum of a grandfather clock could quickly foil the surveillance attempts of a vacationing homeowner. He may have installed a battery-powered motion-enabled camera, only to find that the battery died after a single day of the motion detection algorithm triggering repeatedly on the moving clock. Similarly, a company that installs alarmed motion-sensing cameras to secure its premises may have alarms blaring all of the time at the detection of harmless blowing leaves. Clearly, motion detection for surveillance ideally should focus and trigger solely on human motion.

The goal of detecting exclusively human motion in an electronic system presents two related challenges. The first challenge lies in specifying the fundamental characteristics that separate human motion from other types of motion. Developing a feasible algorithm to recognize these characteristics introduces the second challenge. These challenges are related because only some of the separating characteristics lend themselves well to the development of real-time algorithms.

Successful motion detection algorithms for surveillance must exploit characteristics of motion that offer the best of both worlds. The characteristics must provide reliable separation between human and non-human movement while requiring relatively little processing to implement. The specific motion detection algorithm used in this project takes advantage of the
repetitive nature of a large class of non-human motion as it differs from the normally more variable human motion. This characteristic of repetitiveness strikes a good compromise between effectiveness and feasibility of implementation. The algorithm for this project also makes subordinate use of simple shape- and size-based characteristics of motion.

1.3. Video Compression for Wireless Surveillance

The high demand for compact storage and fast transmission of video has thrust image and video compression to the top of the priority list within several academic and corporate research groups. Commercial products such as digital cameras must cope with the memory-hungry image data that they capture. Compression of digital photographs or video prior to permanent storage allows a photographer to capture more shots or to film for longer periods of time before the memory becomes full. DVD technology utilizes video compression to fit full-length movies onto single, low-cost disks. The internet exemplifies the need to compress images and video for fast transmission. The JPEG standard for compressing and decompressing images allows for the quick download of much of today’s visual internet content. In addition, the growing prevalence of internet streaming video clips necessitates tremendous amounts of compression for the majority of internet users limited to slow modem access.

Video compression for wireless surveillance devices offers another example of compression for fast transmission. Security cameras can gather data at rates above 50 Mb/s, dwarfing the data rate of most practical transmitters. To transmit the video information in a timely manner or even to be able to transmit the information in its entirety, the camera must compress the video before transmission. But unlike for the previously discussed compression applications, compression for surveillance devices must be performed in real time. Video compression for DVDs for example can take days, but a wireless security camera running at 30 frames per second must compress each frame within one thirtieth of a second.

This real-time constraint greatly complicates video compression for wireless security devices as compared with the offline compression that suffices for other applications. First, the constraint imposes strict limitations on the amount of processing available to compression algorithms in video surveillance applications. MPEG-2 encoding, a popular, highly-effective coding scheme used for DVDs and many internet streaming video programs, pushes the envelope of digital signal processor performance in its real-time implementations. Currently, the exorbitant cost of real-time MPEG-2 encoder chips makes this compression scheme undesirable for video surveillance. This project utilizes a less computationally intensive video compression scheme,
wavelet compression, to reduce the input video rate to the 2.5 Mb/s data rate of the Security Sphere transmitter.

1.4. Combining Motion Detection with Compression

For a surveillance system with a physical connection between the camera and the display unit, convenience would suggest that any motion detection features be implemented at the display unit. Whereas the camera typically falls under strict size limitations, no such limitations govern the display unit. In fact, a full-sized desktop PC could accompany the display unit and easily perform all computations necessary to implement a motion detection algorithm. Similarly, a wireless camera not dependent upon video compression (perhaps by running at relatively few frames per second and by using a high data rate transmitter) could also locate the motion detection features at the display unit. A wireless camera operating from battery power benefits even further by moving all possible power consuming processing from the camera to the receiving display unit.

With lossy compression inserted into the data path prior to transmission, the situation changes. The image received and decompressed at the display unit, though visually acceptable, suffers degradation and exhibits artifacts from the compression process. A noisy wireless channel could exacerbate the degradation, as compressed data streams often have little tolerance for bit errors. Positioning the motion detection features at the receiving display unit now becomes a less attractive option. The video received at the display, degraded and potentially laden with compression artifacts, lends itself poorly to reliable motion detection.

The alternative, though fraught with tradeoffs, is to locate the motion detection features at the transmitting camera. The negatives include greater power consumption and increased circuit size at the camera, which may be battery powered and may need to fit in small enclosures. Also, power and size constraints limit the amount of processing available to algorithms implemented on the camera as opposed to at the display unit. The big advantage, however, derives from the availability of clean, uncompressed image data at the input of the motion detection processor. As chips grow smaller, faster, and less power-hungry, this advantage begins to outweigh the disadvantages. For this project, the motion detection processing occurs before compression and transmission in a digital signal processor connected to the Security Sphere’s cameras.
1.5. System Description and Performance

A high-level description of the demonstration system for this project is as follows: This project accepts digital video data from a video camera upon which it performs motion detection and then video compression. Next, it decompresses the video data and displays the output on a NTSC monitor.

At each stage in this high-level description of the data path, this project must satisfy certain functional parameters dictated by the performance specifications of the proposed Security Sphere or by its need to interface with other sections of the Security Sphere.

The Security Sphere operates at full power only when human motion lies within its field of view. To meet this specification, the motion detection algorithm developed in this project compares all detected motion against three criteria. First, it ensures that the motion is not repetitive. A flashing light or a spinning ceiling fan will not trigger the algorithm. Second, the algorithm requires that the motion have significant extent in both the vertical and horizontal directions. For example, a moving human head or body are of a shape that is both tall and wide enough to trigger the algorithm. In contrast, an electrical power line swaying in the wind has the wrong shape to trigger the algorithm. It has extent in the horizontal direction, but insufficient extent in the vertical direction. Third, the algorithm accepts motion only of sufficient size to likely be that of a human. For example, a falling leaf, which might have sufficient horizontal and vertical extent to satisfy the second criterion, has insufficient size to satisfy the third criterion.

To demonstrate the potential of future low-power operation, the Security Sphere ceases transmission when the algorithm detects no motion. As an indication of its ability to carry out this feature and as a demonstration of its proper functionality, the motion detection algorithm in this project blanks the output NTSC monitor when no motion is detected. When motion again enters the camera’s field of view, the NTSC monitor again displays the output video.

The video compression parameters for this project are governed more by the interface issues with other parts of the Security Sphere. Each of the Security Sphere’s four cameras provide 30 frames per second in which each frame consists of 256x256 8-bit pixels. This yields a combined input data rate of nearly 64 Mb/s. The data rate of the Security Sphere’s transmitter is a fixed 2.5 Mb/s. The video compression scheme for this project, therefore, must handle the necessary 25 to 1 compression to reduce the input data rate to the output data rate. The scheme also has the capability to adjust the amount of loss in the compression to hold constant the output data rate, regardless of the material in the input video. For example, video of a gray wall can be compressed much more easily than video of a lush forest, but the compression scheme must adjust to fit both into a 2.5 Mb/s data stream.
The final parameter of this project is a mechanical one. The motion detection and the image compression circuitry for this project must fit on a single circuit board within the 16 cm radius spherical shell that houses the components of the Security Sphere.
Chapter 2.

Compression Background

2.1. Compression Fundamentals

The removal of redundant or unnecessary information provides the foundation for all types of data compression. In a sample voice clip, for example, pauses and extended vowel sounds add only redundant information. A pause might indicate the end of one sentence and the beginning of the next, but absent the pause, a listener can still distinguish one sentence from the next. Extended vowel sounds reinforce the intelligibility of the words in which they occur, yet the words are usually understandable even when the vowel sounds are truncated. A common feature on answering machines compresses recorded messages by removing pauses and extended vowel sounds. The compressed message transmits the same information as the original, but in a shorter period of time. In this special case, the human brain carries out the decompression; the listener mentally inserts the pauses and extended vowel sounds where appropriate. Though the output remains intelligible, this compression creates loss. The listener’s brain does not and cannot exactly reproduce the original sound clip. However, the information that is lost, such as the length of pauses between sentences and the length of the vowel sounds, is of no vital importance. The basic ideas in this audio example, removing redundant and unnecessary data, apply throughout the rest of this discussion on image and video compression.

2.2. Transform-Based Image Compression

2.2.1. A Change of Basis

All digital images consist of pixel values. Often, these pixel values are thought of as the intensity of the image at a given location. Pixel values mean something completely different under another, more interesting, interpretation. In this interpretation which is demonstrated in Figure 2, pixel values serve as coefficients in a linear combination of basis images. The particular set of basis images shown in Figure 2, called the intensity basis, is one of infinitely many sets of basis images.
Sample Four-Bit Image

<table>
<thead>
<tr>
<th></th>
<th>30</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>94</td>
<td>70</td>
</tr>
</tbody>
</table>

Basis Image 1

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Basis Image 2

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Basis Image 3

<table>
<thead>
<tr>
<th></th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Basis Image 4

<table>
<thead>
<tr>
<th></th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Linear Combination

\[
94 \times \begin{bmatrix}
0 \\
1 
\end{bmatrix} + 70 \times 
\begin{bmatrix}
0 \\
1 
\end{bmatrix} + 30 \times 
\begin{bmatrix}
1 \\
0 
\end{bmatrix} + 6 \times 
\begin{bmatrix}
0 \\
0 
\end{bmatrix} = 
\begin{bmatrix}
30 \\
94 
\end{bmatrix}
\]

Figure 2: The intensity basis

Images represented as linear combinations of the intensity basis images (i.e. represented in the intensity domain) correspond to the images that humans see. For an image to be displayed, it must first be represented in the intensity domain. Unfortunately, the information contained in most natural images is distributed quite randomly among all of the intensity basis images. For this reason, to accurately represent a natural image, nearly all of the coefficients must be maintained. The amount of memory required to store an image represented in the intensity domain is often too large. Similarly, the bandwidth required to transmit the image in a timely manner is often unavailable. The first step in image compression is to find a set of basis images that more compactly represents the given natural image data.

The discrete Fourier basis, a scaled version of which is shown in Figure 3, has been found to be one of the bases that achieves the desired compact representation for a large class of images. As demonstrated in Figure 3, the representation of the sample four-bit image under the discrete Fourier basis requires only three coefficients instead of the four required under the intensity basis. This stems from the fact that the first image in the discrete Fourier basis has a rather high correlation with the sample image, and the last image in the discrete Fourier basis has no correlation with the sample image. The bottom image shown in Figure 3 is the discrete Fourier transform of the original sample image. For large, natural images with thousands of pixels, a majority of the discrete Fourier basis coefficients which appear in the transform are zero or close to zero. These small coefficients, which most commonly correspond to the higher frequency
Fourier basis images, reside in a large contiguous group toward the upper right of the transformed image. The more important large coefficients all reside in the lower left of the image. The grouping of the large coefficients into a tight region of the transformed image is referred to as energy compaction. 4

Sample Four-Bit Image

\[
\begin{array}{cc}
30 & 6 \\
94 & 70
\end{array}
\]

Basis Image 1

\[
\begin{array}{cc}
1 & 1 \\
1 & 1
\end{array}
\]

Basis Image 2

\[
\begin{array}{cc}
-1 & -1 \\
1 & 1
\end{array}
\]

Basis Image 3

\[
\begin{array}{cc}
1 & -1 \\
1 & -1
\end{array}
\]

Basis Image 4

\[
\begin{array}{cc}
-1 & 1 \\
1 & -1
\end{array}
\]

Linear Combination

\[
50 \times \begin{array}{cc}
1 & 1 \\
1 & 1
\end{array} + 32 \times \begin{array}{cc}
-1 & -1 \\
1 & 1
\end{array} + 12 \times \begin{array}{cc}
1 & -1 \\
1 & -1
\end{array} = \begin{array}{cc}
30 & 6 \\
94 & 70
\end{array}
\]

Transformed Image

\[
\begin{array}{cc}
12 & 0 \\
50 & 32
\end{array}
\]

Figure 3: A scaled Fourier basis

The discrete Fourier basis yields decent compaction of the original sample image. However, infinitely many bases exist, and some of them might provide even better compaction. Perhaps the sample four-pixel image can be represented by even fewer than three coefficients under another basis. Figure 4 shows an experiment with a third set of basis images. In this case, the sample image can be represented by just one number, for the linear combination contains only one non-zero coefficient. This new basis offers incredible compaction, and thus incredible compression. On the surface, the popularity of the discrete Fourier basis for compression seems puzzling. After all, this third basis offers such significant improvement over the discrete Fourier basis representation. The success of the discrete Fourier basis rests in part in its nearly universal applicability. It can provide decent energy compaction for a large class of images that includes...
most natural images. The basis demonstrated in Figure 4, on the other hand, is effective only for the one sample image used in these examples.

The discrete Fourier basis, as might be expected, can be used to divide a given image into its separate frequency components. An image of a clear blue sky offers an example of images containing almost exclusively low-frequency components. A photograph of a plaid or checkered shirt contains much larger high frequency components. Most natural images, though, have more significant low frequency components than high frequency components. This characteristic of natural images underlies the ability of the discrete Fourier basis to represent an image in a convenient, compact form for image compression.

Sample Four-Bit Image

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>6</td>
</tr>
<tr>
<td>94</td>
<td>70</td>
</tr>
</tbody>
</table>

Basis Image 1

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>3</td>
</tr>
<tr>
<td>47</td>
<td>35</td>
</tr>
</tbody>
</table>

Basis Image 2

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Basis Image 3

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Basis Image 4

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Linear Combination

\[
2 \times \begin{bmatrix} 15 & 3 \\ 47 & 35 \end{bmatrix} = \begin{bmatrix} 30 & 6 \\ 94 & 70 \end{bmatrix}
\]

Figure 4: A highly specific basis

The discrete Fourier basis, however, is not the ideal basis for image compression. Other bases have been discovered whose basis images correlate even more strongly with most natural images. Chief among these is the discrete cosine transform basis, or the DCT basis. The DCT basis better handles the artificial boundaries that exist at the edges of images than does the discrete Fourier basis. And as with the discrete Fourier basis, the DCT basis also provides individual access to various frequency components of an image.

2.2.2. Subimage-by-Subimage Compression

The four-pixel sample image, while useful for illustrative purposes, does not reveal all of the issues associated with practical, full-sized images. Expressing an image with a new set of basis
images, known as transforming the image, requires significant computation to obtain the new coefficients. The amount of computation increases with the size of the image to be transformed, and for the DCT the increase is dramatic. For this reason, the widely used compression schemes that employ the DCT basis divide the input image into smaller blocks, and transform each one separately. For example, a 256x256 image might first be divided into 1024 8x8 blocks to reduce the computation requirements. Compression performed on these individual blocks is referred to as subimage-by-subimage compression. ⁶

This division into blocks yields a second advantage that compression schemes can exploit. The frequency content within a single block often lies within a narrow band, whereas the frequency content of the entire image occupies much of the spectrum. An image of a plaid shirt displayed in front of a blank, white wall, for instance, has frequency content over a wide spectrum. A small block of the image showing just the white wall, however, contains exclusively low-frequency information. A small block showing just the plaid shirt contains almost exclusively high-frequency information. By transforming single blocks independently of the rest of the image, energy compaction can be markedly increased.

### 2.2.3. The Lossy Step: Quantization

The transforming process does not itself yield any compression, but rather it sets the stage for significant compression to occur. As previously mentioned, the energy compaction groups the large coefficients in a tight area of the transformed image, and the rest of the transformed image consists of a contiguous block of smaller coefficients. If all of the coefficient values that fall beneath some low threshold are set to zero, then the resulting transformed image will contain long strings of zeros. Assuming a low enough threshold, the image obtained by transforming back to the intensity domain will not be noticeably affected. When transmitting or storing the transformed image, the strings of zeros can be stored as just one number indicating the number of zeros in the string rather than storing each zero individually. It is during this process of replacing long strings of identical values with a single code word, called run-length coding, that the actual compression occurs. An example of run-length coding is provided in Figure 5.

```
0000000000000000000000000000000000000000000000000000000000000000000(Uncompressed)
57 Zeros (Compressed)
```

Figure 5: Run-length coding example
Knowledge of the human visual system can greatly aid the compression process. The human visual system acts as a low pass filter. Humans easily detect changes to low-frequency components of an image such as to its average intensity, but changes to high-frequency components can go largely unnoticed. The threshold below which coefficients are zeroed should, thus, depend upon frequency. A relatively high threshold can apply to coefficients representing the higher frequency components while an extremely low threshold should apply to the coefficients representing the lower frequency components.

In practice, coefficients are not explicitly zeroed, but, rather, they are quantized. Each pixel in the original image typically is represented by between 8 and 10 bits. The transformed image uses the same number of bits per pixel as used in the original image. When compressing a transformed image, a target number of bits per pixel is specified for the resulting compressed image. The less memory space available to store the image or the less bandwidth available over which to send the image, the fewer bits per pixel available to the compressed image. Each pixel in the compressed image, however, need not use the same number of bits as every other pixel in the image. Instead, the pixels are divided disproportionately in favor of the pixels in the transformed image that represent lower frequency components of the original image. The pixels representing the very highest frequency components may receive as few as zero pixels. These pixels are completely ignored. The pixels representing slightly lower frequencies may be allotted one bit, allowing them to take one of two values. The number of bits allotted increases for the pixels representing ever lower frequency components, and the few pixels representing the lowest frequency components may be allotted the full 8 or 10 bits.

2.2.4. Popular Image Compression Techniques

Currently, the most popular image compression standard is the JPEG standard. This standard specifies that the input image first be divided into 8x8-pixel blocks. In the next step, each of those blocks is transformed, typically using the DCT, and then the pixels in each resulting transformed block are quantized according to a fixed quantization table. The pixels, taken in order of those representing the lowest frequency component to those representing the highest frequency component, are then run-length coded to create the compressed image. Decompression occurs by first decoding the run-length-coded data stream followed by an inverse transform of each of the 8x8 blocks. In the best cases, the resulting image is indistinguishable from the original.

When high compression is desired, the resulting image quality necessarily suffers. For the JPEG standard just discussed, higher compression means more severe quantization of the higher frequency components in each 8x8 block. However, since each block is compressed
independently of the others, compression artifacts can begin to appear at high compression levels. Two adjacent blocks may contain markedly different image data, and thus their transforms will contain different values. If, by chance, the values in one of the transformed blocks all lie close to the quantization points, this block will be nearly unaffected by the compression. If, also by chance, the values in the adjacent block happen to lie far from the quantization points, this block will be noticeably affected. In the resulting full output image, a visible difference can be seen between these two blocks. Known as the blocking effect, this compression artifact overlays a faint checkerboard appearance onto highly compressed images when using DCT subimage-by-subimage compression.

The attempt to find a compression technique that leads to more graceful image degradation at high compression levels has, in part, contributed to the development of wavelet compression. Wavelet-based compression uses many of the same techniques as DCT-based compression. The input image is transformed to a new domain, the transform coefficients are quantized, and last the transformed and quantized image is run-length coded. Techniques employing wavelet compression, however, need not first divide the image into 8x8 blocks, and as a result do not suffer from blocking effects at high compression ratios. Finding coefficients for the wavelet basis requires less computation than finding DCT coefficients, enabling a feasible wavelet transform of the entire image without first dividing the image into smaller blocks. Despite operating on the entire image simultaneously, some wavelet bases also provide good energy compaction.

In practice, the wavelet transform is implemented with several stages of filters separated by downsampling operations. This architecture is known as a filter bank. Altering the filters in the filter bank effectively alters the wavelet basis with which the transformed image is represented. The filter bank has several outputs, that grows in number with the number of filters used in the filter bank. Each output contains the components of the input image that lie within a particular frequency band. To obtain compression from these outputs, the steps are familiar. The high-frequency band output is assigned relatively few bits per pixel. Lower frequency band outputs are assigned a larger number of bits per pixel. The resulting data is run-length coded to yield a compressed image. The decompressed version of the original image can be obtained by sending the compressed image through a second filter bank that executes the inverse operation.

2.3. Interframe Video Compression

For video, a second and potentially more powerful form of compression can be utilized that takes advantage of temporal redundancy from one frame to the next. Some video clips lack movement for extended periods of time. During this time, each new frame may be identical to
the last. Rather than saving the same image data repeatedly, a few code words can be inserted into the compressed data to indicate that the image has not changed. This is the basic rationale for temporal compression, also called interframe compression, but in practice more sophisticated methods are employed to squeeze more temporal redundancy from the video.

For example, most video clips display limited areas of motion in the foreground that overlays a large, unchanging background scene. By dividing the frames, again, into smaller blocks, the blocks containing only the changeless background can be compressed temporally. The blocks that contain the foreground activity of the video clip can be compressed spatially, using the JPEG standard for example. Even more compression can be garnered for each block by spatially compressing only the difference between the current values in the block and the previous values in the block.

This idea of compressing only temporal differences can be carried yet further through the use of motion estimation. Much of the change that occurs from one frame to the next in video is the result of rigid body movement. For example, a video clip of a passing car shows movement, but the movement is almost entirely a simple shift of pixel values due to the rigid-body nature of the car. If the speed and direction of the car's movement in the current frame is known, then its position in the next frame can be accurately estimated. Based upon the estimation of the car's position, accurate estimates of the pixel values in the next frame can also be made. Now, only the differences between the estimated values of the current frame and the actual values of the current frame need be spatially compressed. If the motion estimation proves effective, these differences can be exceedingly small. In the popular and flexible MPEG video compression standard, intraframe JPEG compression which eliminates spatial redundancy and interframe motion estimation which eliminates temporal redundancy are combined into one powerful algorithm.
Chapter 3.

The Video Compression Feature

3.1. Selection of a Video Compression Solution

Several strict requirements exist for the compression scheme in this project. It must accept four 8-bit 256x256 digital frames at 30 frames per second and compress them to a 2.5 Mb/s output data stream. This conversion must happen in real-time. Also, the solution must be implemented on a circuit board that can fit within the Security Sphere.

The initial idea was to implement a simple profile of the MPEG algorithm on a digital signal processor. It was discovered, however, that a team of engineers at a chip manufacturing company were currently working on that very project, and that the work had yet to be completed. Due to the large development time for a custom real-time video compression algorithm, off-the-shelf solutions became the focus of the compression scheme selection process.

The search for MPEG encoders turned up numerous devices, but very few operated in real-time and none met the input/output requirements of the application. Some that showed promise were prohibitively expensive. A more general search for video encoders eventually led to the Analog Devices ADV601 low-cost video codec. This dedicated signal processor IC accepts digital video in various formats at 60 digital frames per second. It offers real-time wavelet compression at a selectable compression ratio anywhere between 4 to 1 and 350 to 1, and it outputs the compressed data stream on a selectable 8-bit to 32-bit wide bus. The ADV601 can also be configured as a decompression chip, so that two ADV601 chips working in tandem can compress and subsequently decompress video in real time. Analog Devices also offers a standalone evaluation board for the ADV601 that demonstrates its performance. Due to its flexibility, its low cost, and promising results from the evaluation board, the ADV601 was selected as the compression solution for this project.

3.2. ADV601 Wavelet Video Compression

The ADV601, though designed for video compression applications, actually executes only repeated image compression. The compression performed by the ADV601 acts on each frame of
the input video independently of all other frames. Unlike compression specified by the MPEG standard that uses motion estimation, ADV601 compression fails to take advantage of the large amount of temporal redundancy that exists in most video data streams. More succinctly, the ADV601 employs only intraframe compression and no interframe compression. Interframe compression requires large amounts of memory for frame stores and significant computation for motion estimation, both of which are inconsistent with the ADV601’s low-cost and real-time characteristics. To compensate for its lack of interframe compression, the ADV601 must often garner high levels of compression from intraframe techniques. Such a situation calls for the graceful degradation offered by wavelet compression, which is exactly the type of compression used by the ADV601.

![ADV601 functional block diagram](image)

The ADV601 operates as outlined in the block diagram of Figure 6. Digital video data, formatted in one of five selectable digital video formats, enters the ADV601 through an 8-bit video data bus. Using the sync signals that accompany the video stream, the ADV601 extracts each digital frame for processing. After the data for a frame has been accumulated in an SRAM internal to the ADV601, the data passes through a bank of digital low-pass filters, high-pass filters, and decimators. The filter bank outputs 14 blocks that comprise the wavelet transform of the image. An example taken from the ADV601 datasheet of an original image and of its wavelet transform is shown in Figure 7. Each block contains a different band of frequency content from the original image. An external DRAM buffers the transformed data before the next step in the data path.
Figure 7: Original image (top) and its wavelet transform (bottom.)

3.2. ADV601 WAVELET VIDEO COMPRESSION
Statistics about each of the 14 frequency blocks is sent to a supporting external digital signal processor (DSP). Based upon the statistics, the DSP calculates the amount of quantization required for each block, and it sends the quantization information back to the ADV601. Changing the program within the DSP allows for variable levels of compression and for developing a feedback system that maintains a constant output bit rate. Analog Devices provides programs for the external DSP to handle various needs. The Analog Devices program allowing for a constant bit rate was used for this project.

With quantization levels supplied by the DSP, a quantizer internal to the ADV601 appropriately quantizes the frequency blocks in the transformed image. This quantization is the first step that provides actual compression, and is the only lossy step in the process. The quantized transformed image next undergoes run-length coding to further compress the data stream. The last step in ADV601 compression is Huffman coding. This step replaces common sequences that appear in the data stream with short code words. This compressed data stream, buffered by a 512-word internal FIFO, is output in parallel format from the ADV601's host data bus.

When in decode mode, the ADV601 executes the process in reverse. Decoding is somewhat easier, because no quantization is needed and, therefore, no quantization levels must be calculated by an external DSP.

### 3.3. Video Compression Circuit Functional Description

The video compression circuit has two functional sections. The first section converts the raw digital data from the cameras to the ITU-R BT.656 format that the ADV601 can understand. The second section is the ADV601 and its supporting components. Detailed schematics can be found in Appendix B.1.

#### 3.3.1. Conversion to the ITU-R BT.656 Format

The ADV601, into which the digital video data must be fed, accepts digital data only in specific formats. In particular, the ADV601 supports a glueless interface to video data in the format specified by the International Telecommunications Union Recommendation ITU-R BT.656. Glueless interfaces operate properly without any externally supplied logic signals. Some video equipment directly outputs digital video data that complies with this format. Also available are ICs that convert popular video formats such as NTSC to the ITU-R BT.656 format.

The digital video data supplied by the four cameras of the Security Sphere comes at 30 frames per second and with a pixel sync and a frame sync, but it conforms to no popular formats.
Therefore, a custom format converter was created in hardware on the circuit board to convert the camera outputs to the ITU-R BT.656 format that the ADV601 can understand. Before explaining the conversion circuitry, a description of the ITU-R BT.656 format is given.

The ITU-R BT.656 format was designed specifically to provide a digital counterpart to the NTSC analog video standard and to allow for easy conversion between the two. Many of the parameters in the ITU-R BT.656 format are identical to those specified by the NTSC standard. For example, the horizontal blanking and vertical blanking time periods are identical. Both formats also use 525 lines for each frame, divided into two fields of 262.5 lines. In many ways, an ITU-R BT.656 digital data stream is just a sampled version of a corresponding NTSC analog signal using a 27 MHz sample clock that alternately samples the luminance and chrominance components. A line of NTSC video, which lasts for 63.5 $\mu$s, yields 1716 samples when sampled with a 27 MHz clock. The ITU-R BT.656 standard, therefore calls for 1716 eight-bit words per line. Like a line of NTSC, a line of ITU-R BT.656 video consists of an active region and a blanking region as shown in Figure 8. For the ITU-R BT.656 standard, this active region consumes 1440 of the 1716 words in each active line. The 276-word blanking region contains the expected digital blanking data, but it also contains syncing code words. At the beginning and end of the blanking region, the ITU-R BT.656 data stream contains timing reference signals that provide horizontal and vertical syncing. This feature enables an ITU-R BT.656 data stream to be transmitted without any accompanying sync signals, aside from a 27 MHz clock that is synchronous with the data. The 1440 words in the active region alternate between luminance and chrominance, leaving 720 words of each. Of the 525 lines per frame in the ITU-R BT.656 format, 40 lines are used for blanking and contain digital blanking words. This leaves 485 active lines per frame. Thus, the total active frame size is 720x485 pixels, a large enough frame size in which to pack the data from the Security Sphere's four 256x256 pixel video images.

**Figure 8: A line of ITU-R BT.656 formatted video**
The job of the format converter is to insert, in the proper order and at the proper times, data words, code words, and blanking words into the data stream that feeds the ADV601’s eight-bit video input. The format converter hardware consists only of the FPGA and four FIFO buffers as seen in Figure 9. All of the logic associated with controlling the format conversion is programmed into the FPGA using VHDL. The VHDL code can be found in Appendix A.1. The FIFO buffers act as the interface between the raw digital data stream from the cameras and the ITU-R BT.656 data stream that feeds the ADV601. Data from the cameras is clocked into the FIFOs by the pixel sync signal that the cameras supply. When a data word is needed for the data stream, the FPGA sends a signal to clock a word out of one of the FIFOs. The FIFOs are needed due to the fact that the output data rate from the camera and the input data rate to the ADV601 temporarily differ, even though on average they both move video data at 30 frames per second. By using the FIFOs instead of large frame buffers, component cost and latency in the data path are reduced. When a code word or a digital blanking word is needed, the FPGA itself outputs the appropriate byte, as instructed by the VHDL code. The FPGA and the FIFOs, therefore, take turns driving the ITU-R BT.656 data bus. When one device is driving the bus, the other shows high impedance outputs to prevent bus contention. The FPGA determines when to insert various words simply by counting clock cycles referenced by the frame sync.

3.3.2. ADV601 and Supporting Components

The ADV601 takes in the video data in the ITU-R BT.656 format and compresses it. However, the ADV601 relies on a large supporting cast of components to complete this task. First, at power-up, the FPGA configures the ADV601 by writing to its internal control registers through its host data bus. To support the actual compression process, a DRAM and a general purpose DSP operate in conjunction with the ADV601. The ADV601 uses the DRAM to store and reorder the wavelet-transformed image data as it awaits input to the ADV601’s internal quantizer. The interface between the ADV601 and the DRAM is glueless, as the ADV601 automatically controls all reading, writing, and refresh of the DRAM. The external, general purpose DSP calculates quantization levels for the ADV601. The two chips communicate via their serial ports. The DSP itself requires another chip, an external EEPROM, in which the code run by the DSP is stored.

The FPGA comes into play again at the output of the ADV601. Internal to the ADV601 is a 512-word FIFO that buffers the compressed video data for output to its host data bus. When it finishes compression processing on a frame, the ADV601 begins to fill its internal FIFO. External pins indicate the status of this FIFO. The FPGA monitors these status pins and clocks compressed data from the ADV601 host data bus at the proper rate to prevent the FIFO from
growing full or empty. The data clocked out of the ADV601 is immediately clocked into an external FIFO. The output of this external FIFO feeds an FPGA-controlled parallel-to-serial converter that prepare the compressed data for wireless transmission. The external FIFO is the safety net for the FPGA, allowing the FPGA to control the ADV601 output and the parallel-to-serial converter input largely independently of one another. The parallel-to-serial converter outputs the video data at a constant 2.5 Mb/s to the wireless transmitter portion of the Security Sphere.

Figure 9: The format converter
Chapter 4.

The Motion Detection Feature

4.1. Motion Detection Algorithm Development

Developing a feasible algorithm is the first step in implementing a motion detection feature with the desired properties. Algorithm development must precede hardware choices and printed circuit board layout, as the algorithm specifies such parameters as the amount of memory and the processing speed required of the hardware.

4.1.1. Algorithm Search and Selection

A brief investigation into motion detection techniques that can distinguish between human and non-human motion led to the discovery of a promising algorithm concept presented by Victor Lum of the MIT Artificial Intelligence Laboratory. The basis for this concept is that human motion differs from much non-human motion in its variability. The motion of humans is erratic and random, rarely following the same path twice. The motion of most mechanical objects, such as a machine in a factory or the pendulum of a grandfather clock, follows the same path repeatedly. The repetitive nature of this motion can be determined by monitoring individual pixels over time. For example, the gold colored pendulum of the grandfather clock swings in front of the brown wood of the clock's backing. The pixels through which the pendulum swings alternate repeatedly through the same two colors, brown and gold. These pixels never assume any other value. The algorithm used for this project exploits this simple observation about this large class of repetitive non-human motion.

4.1.2. Detailed Algorithm Description

The operation of the algorithm when presented with the swinging pendulum of the grandfather clock is described below. At some point in time, a camera equipped with the motion detection algorithm is powered up and positioned to point at the grandfather clock. The algorithm will initially declare that motion exists at a pixel through which the pendulum swings. After some specified period of time during which that pixel assumes only two different values, the algorithm asserts that no human motion exists at that pixel by clearing a motion bit for that pixel. The two values that the pixel assumes are stored in memory. If, however, at any later time
someone were to walk in front of the grandfather clock causing the pixel through which the pendulum swings to take on a brand new value, the algorithm asserts that human motion exists at that pixel by setting a motion bit for that pixel. The declaration of motion at this pixel does not expire until the pixel again assumes just two different values for the specified period of time. The algorithm repeats this process for every pixel in the frame.

Figure 10: Stage 1 of the motion detection algorithm
This pixel-by-pixel processing that checks for repetitive motion constitutes Stage 1 of the total motion detection algorithm. Stage 1 of the algorithm is outlined in graphical flowchart form in Figure 10. The timer variable defined in the flowchart indicates when the declaration of motion for a pixel will expire. When non-repetitive motion is detected, the motion bit for that pixel is set and the timer for that pixel is set to ten. Therefore, the motion bit for that pixel will not be cleared until ten consecutive frames pass without non-repetitive motion at that pixel.

For each frame of video, Stage 1 outputs a binary image which consists of the motion bits for each pixel. An example of how this might look for a 32x32-pixel image is shown in Figure 11. This cartoon image shows the motion of a human, a flying bird, and a swaying tree limb. It is upon this sort of binary image that Stage 2 of the motion detection algorithm operates. The purpose of Stage 2 is to prevent small or non-human-shaped motion from triggering the algorithm.

First, Stage 2 determines if the shapes of the regions of motion can possibly be human through the use of simple morphological processing. Each area of motion is eroded in both the horizontal and vertical directions to eliminate objects that are too narrow to be human. For example, the motion of the tree limb would be eliminated by this erosion operation and the pixels
that indicate its motion are cleared. The areas that indicate the motion of the human and the bird survive this morphological processing, because these areas have sufficient extent in the vertical and horizontal directions.

The second operation in Stage 2 is a simple count of the remaining pixels that display motion. If the number exceeds a chosen threshold, then motion is declared for this frame. The pixels triggered by the bird alone are not enough to exceed the threshold; the larger motion of the human is required. This completes Stage 2, which is summarized as a flowchart in Figure 12. When the next frame arrives the entire algorithm repeats, beginning again with Stage 1. Upon declaration of motion for a frame, an externally accessible bit is set to indicate to the rest of the system that motion has been detected. When motion is initially declared for a frame, the bit goes high and remains high for at least ten seconds, regardless of whether motion is detected in subsequent frames. If no frames indicate motion for ten consecutive seconds, then the externally accessible motion bit returns to its low value.

![Figure 12: Stage 2 of the motion detection algorithm](image-url)
4.1.3. Verification of Algorithm Functionality

After having precisely specified the operation of the motion detection algorithm, functional testing and verification of the algorithm began. The first tests were executed in Matlab using a Matlab script which can be found in Appendix A.2.5. Matlab provides several built-in image processing functions, such as those for morphological processing, that facilitated the size and shape checking features of Stage 2 of the algorithm. Matlab also provides an easy means to test the algorithm on a sequence of real images. Using a digital camera, about 80 close-up images of a running digital stopwatch were captured and stored within the computer. These images were then indexed in time to create 80 frames of digital video for test. From one image to the next, the only changes were in the digits displayed on the LCD of the watch. This caused sections of the LCD to alternate between light and dark, producing a good simulation of repetitive motion. The binary output images of Stage 1 of the algorithm were examined. These images indicated proper operation of the algorithm, opening the door to hardware selection.

Based upon estimated memory and processing requirements of the algorithm, the Texas Instruments TMS320C6211 digital signal processor (DSP) was chosen to implement the motion detection algorithm. Texas Instruments offers a C compiler that outputs optimized assembly code for the TMS320C6211 (abbreviated as 'C6211.) Therefore, the next step in algorithm testing was to translate the Matlab scripts to equivalently functioning C code. Matlab's built-in image processing functions used to implement Stage 2 of the algorithm have no direct C analogues. However, Texas Instruments has its own image processing toolbox for use with the 'C6211 that contains optimized morphological functions. Using these functions, the translation was completed and a new set of simulated images were created to test the algorithm.

To help in the testing process Texas Instruments donated a 'C6211 evaluation board and Code Composer software that works closely with it. The Code Composer software translates C code to functionally equivalent instruction codes that can be loaded directly from a computer onto the evaluation board through the computer's parallel port. The motion detection C code was translated using the software and tested on the evaluation board.

The TMS320C6211 has a fixed-point processor, which caused some rounding problems in the original C code. More seriously, however, the morphological functions from the image processing toolbox did not work. Consultation with representatives from Texas Instruments revealed bugs in these functions. The decision was made to forsake the functions from the image processing toolbox and instead design custom morphological functions. Using processor friendly logical and shift operators to minimize computation, successful replacement functions were developed. After several tests of the C-code, the evaluation board supplied corroborating
evidence that the motion detection algorithm would work in its eventual hardware implementation.

4.2. Motion Detection Circuit Description

At the heart of the motion detection circuit is the TMS320C6211 DSP. With a 150 MHz processor and 1 GB of addressable external memory space, the C6211 is well suited to the demands of the application. The device comes equipped with 64 kB of internal scratch pad RAM that can simultaneously be used for code and data storage. Due to the simplicity of the motion detection algorithm, 64 kB offers plenty of space to store the entire program internally. The C6211 contains a 32-bit processor and a 32-bit external memory interface. This 32-bit interface allows the 8-bit outputs from the four cameras of the Security Sphere to be read simultaneously. The C6211 also comes equipped with two 32-bit timers that each interface to the outside world with an input pin and an output pin. The timer inputs and outputs can serve as general purpose inputs and outputs.

The disadvantages associated with C6211 stem from its ball grid array (BGA) package and its power supply requirements. Though highly desirable for compact production designs, BGA packages can be problematic for prototyping in an academic setting. A BGA package cannot be hand-soldered, for it requires special heating equipment. The equipment is unavailable at MIT, so the device and the circuit board were sent to a local company for soldering. PCB layout for a BGA package also can be difficult and add to the expense of the circuit board fabrication. In order to route all of the connections from the array, four circuit board layers were necessary. These layers included two signal layers, one power layer, and one ground layer. Another disadvantage of the BGA package arises from the inaccessibility of the device’s pins, as they are buried beneath the device after soldering. The power supply to the C6211 must deliver both 3.3 volts and 1.8 volts. When not using a desktop power supply, external regulator circuitry must be utilized to create the 1.8-volt supply. Despite these disadvantage, the C6211 offers a workable solution for this application.

The C6211 interacts with various other devices to perform the functions required of it. Chief among these are external memory chips. For each pixel, the motion detection algorithm must save two 8-bit values that store the two most recent past intensities, an 8-bit timer value to specify when the declaration of motion for each pixel expires, an 8-bit value to expedite the process of updating the two most recent past intensities, and, of course, a 1-bit motion bit. This adds up to a storage requirement of 33 bits for each pixel. With 4x256x256 = 262144 pixels used by the Security Sphere, the total storage requirement comes to just over 8 Mb. While the C6211
external memory interface can easily address that amount of storage space, it is difficult to find an SRAM that is both large enough to store 8 Mb and fast enough to meet the processing requirements of a real-time system. Every access to external memory consumes significant processing time. One solution is to use multiple small and fast SRAMs, but this would increase circuit size, further complicate the already congested PCB routing near the DSP, and create the need for additional memory mapping circuitry to prevent bus contention. The decision, instead, was made to alter the algorithm such that it monitors only every other pixel. With this change and by storing the motion bits for each pixel in the DSP's internal RAM, the total external memory requirement falls to 4 Mb. This still pushes the limits on the size of fast SRAMs on the market. Fortunately, a 4 Mb SRAM with a relatively fast access time (15 ns) was found, and this SRAM was chosen for the design. The SRAM has an 8-bit I/O bus, and therefore occupies only the lowest byte of the DSP’s 32-bit external memory interface.

More circuitry external to the DSP is required to get the video data delivered from the cameras into the DSP. The only port available for this transaction is the external memory interface, but the lowest byte of that interface is in use by the SRAM. To prevent bus contention, the 32-bit digital video bus from the cameras first passes through two 16-bit tri-state buffers before connecting to the DSP's external memory interface. The tri-state buffers are memory mapped in extra room within the addressable external memory space. When the DSP needs to get the next pixel value from the cameras, it simply reads from the mapped location in memory.

The C6211 has no internal ROM from which it can load program code at reset or power-up. The program instead is initially stored in an external EEPROM, which is memory mapped to a specific address space and is accessed through the lowest byte of the external memory interface. At reset, the instruction codes are transferred from the EEPROM to the DSP's internal code memory. The byte-wide bus between the DSP and the EEPROM suffices even though the DSP uses 32-bit instruction codes. Through the use of the external mode pins on the DSP that are sampled at power-up, the boot process can be customized to use an 8-, 16-, or 32-bit interface. The 8-bit interface was chosen for this project to simplify PCB routing and to eliminate the need for multiple external EEPROMs.

A small amount of external clock circuitry is needed to activate a phase lock loop (PLL) internal to the C6211. This PLL multiplies the frequency of the input clock signal by four to create the processor clock. Due to its ready availability, an external clock of 27 MHz was applied to the DSP, yielding an internal processor clock of 108 MHz. Though significantly less than the 150 MHz that the DSP can handle, it proved fast enough for the purposes of this project. The full design schematics can be found in Appendix B.1.

4.2. MOTION DETECTION CIRCUIT DESCRIPTION
4.3. Motion Detection Hardware/Software Unification

Given a working algorithm written in C and the hardware in place to run the algorithm, the next step is to link the two together. The C code written on the computer must somehow get into and run on the DSP. The Texas Instruments software development programs and the use of handwritten assembly resolve the issues associated with unifying the hardware and software. The software development path is shown in Figure 13 and explained in the following sections.

Texas Instruments provides four software development programs that combine C code and handwritten assembly and then convert the combined output to machine code that can be run on the C6211 DSP. The first of these programs is the C compiler, which accepts C code and converts it to C6211 assembly code. This C compiler supports the ANSI C standard, but it also supports several pragmas specific to this DSP. Pragmas are commands typed directly in the C source file to control how the compiler handles functions and symbols such as variable names. The interrupt pragma and the data section pragma are used in the C code in this project. The

Figure 13: TMS320C6211 software development flow
interrupt pragma applies to a C function that serves as the interrupt service routine. It instructs the compiler to follow special interrupt conventions when converting this C function to assembly code. Interrupt conventions include saving all registers altered by the interrupt before executing the routine and then restoring the registers to their original value and branching to the proper address at completion of the routine. Each data section pragma applies to a variable defined in the C code, and its inclusion in the C code is the first step in the assignment of these variables to specific places in the memory space of the DSP.

The second program in the software development flow is the assembler. The assembler accepts one or more assembly files and converts them to the corresponding ‘C6211 32-bit instruction codes. ‘C6211 assembly files, however, contain more than just assembly language code. They also contain expressions called assembler directives. Assembler directives primarily are used to declare and group together variables and to group together lines of assembly code into named sections in the assembler output file. For example, lines of code that are used in the boot process might be placed in a section called "boot_code." All of the variables intended to be stored in an external SRAM might be grouped together in a section called "ext_SRAM." The data section pragma discussed above is a way to specify the variable groupings directly in the C code. The output file of the assembler, therefore, is divided into several named sections each of which contains either grouped variables or grouped ‘C6211 32-bit instruction code words.

The third program in the software development flow is the linker. Aptly named, it links or assigns sections from the assembler’s output file to specific address ranges within the memory space of the ‘C6211 DSP. Continuing from the previous example, the linker might be instructed to assign the "boot_code" section of the assembler output file to an address range in the memory space that selects an external boot EEPROM. The "ext_SRAM" section might be assigned to an address range that selects an external SRAM. A linker command file tells the linker program where in the DSP memory space to assign each of the sections from the assembler output file. The linker command file created for this project is shown in Appendix A.2.3. The next paragraph provides a full description of the DSP memory space.

The TMS320C6211 memory space is divided into several sectors, as specified by its memory map shown in Figure 14. Low addresses access the DSP’s internal memory, into which all time-critical code should be linked. Starting at address 01800000h, the addresses access the configuration registers, which contain settings for all aspects of the DSP’s operation. These registers configure the timers, serial ports, memory interfaces, and interrupts, among other functions. In the upper region of the address space are the CEn external memory spaces. The ‘C6211 chip has four chip enable output pins called CE0, CE1, CE2, and CE3 that go low when
an address in the corresponding CEn external memory space is selected. All memory mapped I/O devices such as the external SRAM, the tri-state buffers, and the boot EEPROM are selected by addresses in the CEn memory spaces. For example, the 4 Mb SRAM used in this project is mapped to the CE0 memory space. Therefore, when the DSP reads from address 80000000h, it actually reads from the base of the SRAM.

<table>
<thead>
<tr>
<th>Address Range (Hex)</th>
<th>Size (Bytes)</th>
<th>Description of Memory Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0000 - 0000 FFFF</td>
<td>64K</td>
<td>Internal RAM</td>
</tr>
<tr>
<td>0001 0000 - 017F FFFF</td>
<td>24M-64K</td>
<td>Reserved</td>
</tr>
<tr>
<td>0180 0000 - 01A3 FFFF</td>
<td>2M+256K</td>
<td>Internal configuration bus</td>
</tr>
<tr>
<td>01A4 0000 - 7FFF FFFF</td>
<td>2G-256K</td>
<td>Reserved and Miscellany</td>
</tr>
<tr>
<td>8000 0000 - 8FFF FFFF</td>
<td>256M</td>
<td>External memory interface CE0</td>
</tr>
<tr>
<td>9000 0000 - 9FFF FFFF</td>
<td>256M</td>
<td>External memory interface CE1</td>
</tr>
<tr>
<td>A000 0000 - AFFF FFFF</td>
<td>256M</td>
<td>External memory interface CE2</td>
</tr>
<tr>
<td>B000 0000 - BFFF FFFF</td>
<td>256M</td>
<td>External memory interface CE3</td>
</tr>
<tr>
<td>C000 0000 - FFFF FFFF</td>
<td>1G</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

Figure 14: The TMS320C6211 memory map

The linker output is in a form that can execute in the DSP, so the next step is to load this executable instruction code into the DSP memory space. For this project, the code is loaded into the DSP from a boot EEPROM, which first must be programmed by a device programmer. Device programmers, however, require that the input file be converted to one of several hex file formats. The hex conversion utility, the last program in the software development flow, takes the linker output file and converts it to the desired hex file format. A hex command file created for this project to control the hex conversion utility is shown in Appendix A.2.4.

The program, though, still must get from the EEPROM into the DSP. A boot process carries out this transfer. At reset, the DSP automatically copies a 1-KB block of code data, called the boot block, from the base of the CE1 memory space to internal memory at address zero. Hence, the EEPROM is enabled by the CE1 pin from the DSP, i.e. it is mapped to the CE1 memory space. After the boot block transfer, the DSP starts execution from address zero, executing the code in the boot block. The boot block code must instruct the DSP to copy the remaining code
from the EEPROM to the address ranges to which that code is assigned in the linker command file.

The boot block normally is hand written in assembly. In addition to transferring code into the proper memory locations, it also configures the DSP by writing the proper values to the configuration registers. When finished with its tasks, it instructs the processor to jump to the location where the main algorithm begins. The boot-block assembly code used for this project is shown in Appendix A.2.2.
Chapter 5.

Testing and Demonstration

5.1. Overview

After tests of individual parts of the project were complete, a platform was devised to allow first for testing of the overall functionality of the system and then for demonstrating this functionality. A video camera with both NTSC and digital output created by Keith Fife serves as the digital video source in this combined testing and demonstration platform. To provide a revealing and meaningful visual output, the ITU-R BT.656 data stream on the project circuit board feeds directly into an ADV601 Video Pipe evaluation board. The evaluation board first compresses and then decompresses the video data before displaying the resulting video to a NTSC monitor. To compare the input video quality with the output video quality, the video camera source simultaneously displays its NTSC output on a second monitor. A diagram of the platform is shown in Figure 15. Accommodation of this testing and development platform necessitated some changes to the original project design. Furthermore use of this platform to test the project uncovered flaws in the original design, which led to additional design changes to correct these flaws. All design alterations and the reasons for these alterations are described in the following sections.

5.2. Motion Detection Testing

The results of a successful test of the motion detection algorithm would indicate that the algorithm triggers on all new motion of sufficient size, has the ability to recognize repetitive motion, and ignores motion of small size and shape. The first step in conducting a test, however, was ensuring that the video camera used for the testing and development platform interfaced properly with the motion detection circuit. The digital output from the video camera used in the platform provides 8-bit 256x256-pixel frames at 60 frames per second and is accompanied by a pixel sync and a frame sync. This is largely identical to the input expected from one of the Security Sphere’s four cameras, except that it is at twice the rate. The Security Sphere’s cameras
provide data at 30 frames per second. Fortunately, as testing progressed, a method was devised that makes the input data rate irrelevant.

In the original implementation, the pixel sync signal that accompanies the video data was connected to an external interrupt on the DSP. For each new pixel, the pixel sync would trigger the interrupt and run the C-coded interrupt service routine. For each even-numbered pixel, this routine read the pixel value from the tri-state buffers and then performed the necessary computations on that value. For an odd-numbered pixel, the interrupt would simply return to the main program. As discussed in Section 4.2, only every other pixel was monitored in order to ease memory constraints. By observing certain signals controlled by the interrupt routine, it became clear that the interrupt processing time greatly exceeded the time between new pixels.

To resolve this issue, the pixel sync signal was rewired from the external interrupt to one of the DSP’s timer inputs. Changes to the boot assembly code configured the timer to count pixels and to trigger a timer interrupt on every 30th pixel, which in turn would call the interrupt service routine. Again, the routine would read the pixel value from the tri-state buffers and do processing. This system interrupts only at every even-numbered pixel location once every 15 frames, yielding a 30-fold increase in processing time. At 60 frames per second, the program

Figure 15: Testing and demonstration platform
monitors every even-numbered pixel four times per second. As an extra benefit, the available processing time can be easily increased or decreased by changing the timer interrupt value.

Testing of the altered design, however, initially yielded unexpected and undesired results. When shown any natural video, the algorithm would always declare motion, regardless of whether motion was present in the video. Only when shown video of a globally uniform value, such as when the camera shutter was closed or when a sheet of paper was held in front of the lens, did the algorithm declare that no motion was present. The problem stemmed from the variability of interrupt processing. When an external event triggers an interrupt, such as when the timer counts the 30th pixel, the event immediately sets an internal bit in the DSP to request interrupt servicing. However, the DSP cannot always jump immediately to the interrupt service routine upon receiving this request. If, for example, the request comes just before an imminent branch instruction, the DSP cannot jump to the service routine until after the branch has been executed due to processor pipelining. As a result, when the interrupt routine is finally called, it will sometimes read the wrong pixel value from the tri-state buffers, or worse, read from the tri-state buffers during the transition between one pixel and the next.

To create a more reliable timing framework, one of the FIFOs previously unused in the testing and development platform is employed as a latch. For timer interrupts, the internal bit that requests interrupt servicing can also be made available on the timer output pin. This bit, and thus the timer output pin, pulses high immediately upon receiving the 30th pixel sync pulse. By routing the timer output pin to the load clock of the extra FIFO, the current pixel value is also immediately stored in the FIFO. The value waits safely in the FIFO until the interrupt service routine gets around to accessing it. With this change implemented, the motion detection algorithm worked largely as desired, with only slight changes to algorithm parameters required to achieve the fully desired performance. The final version of the C code can be found in Appendix A.2.2. The revised circuit schematics are in Appendix B.2.

5.3. Format Conversion Testing

The format conversion block of the design is much more sensitive to changes in the video source than is the motion detection block. Alterations to both the circuitry and the VHDL code were made to account for differences between the digital output provided by the camera in the testing and demonstration platform and digital output expected from the Security Sphere cameras. The first challenge revolved around properly fitting frames of data from the test camera into frames of the ITU-R BT.656 data stream.
The test camera outputs 256x256-pixel frames at 60 frames per second. Each line of its output video stream lasts for 65.1 $\mu$s. A 41.7 $\mu$s data period, which contains the 256 horizontal pixel values, and a 23.4 $\mu$s horizontal blanking period combine to make each line. The ITU-R BT.656 format allows for the transmission and interlaced display of alternating 720x243-pixel fields and 720x242-pixel fields at 60 fields per second. Each line of ITU-R BT.656 video lasts for the NTSC-standard 63.5 $\mu$s. A 53.3 $\mu$s data period, which contains the 720 horizontal pixel values, and a 10.2 $\mu$s horizontal blanking period combine to make each line.

The number of visible lines in a field of ITU-R BT.656 video is 243 or 242, and 20 blanking lines fill the vertical blanking regions of the field. To insert the 256 lines from the camera into an ITU-R BT.656 field, the 13 or 14 lines that do not fit in the visible region must simply be deposited in the vertical blanking regions. They are not seen on the display monitor, but the missing portion of the frame should be unnoticeable.

More complicated is the task of distributing the 256 horizontal pixels per line from the camera within the 720 visible pixels available in a line of ITU-R BT.656 video. This must be done with consideration given to the 4 to 3 aspect ratio of a display monitor and the 1 to 1 aspect ratio of the 256x256 square array from which the digital input originates. After interlacing, each input pixel value from the video camera is two lines tall on the display monitor, taking up $2/(242+243) = 0.412\%$ of the visible vertical dimension. Each input pixel value should then consume $0.412 \times 3/4 = 0.309\%$ of the visible horizontal dimension. 0.309\% of 720 is approximately 2.2. Thus for a nearly perfect output aspect ratio, each pixel from the input line should be repeated in 2.2 pixels in the ITU-R BT.656 output line. At the sacrifice of a slightly skewed aspect ratio, each input pixel is repeated twice in the output line. Only 512 of the 720 visible pixels in an ITU-R BT.656 line will then contain image data. The rest are blanked, leaving two 104 pixel-wide black stripes on each side of the display monitor.

In order to implement this scheme in hardware, the source video camera sends the same data to two separate FIFOs. The VHDL code instructs the FPGA to alternate between the FIFOs for each request of a video word to insert into the ITU-R BT.656 data stream.

After making the necessary VHDL and circuit alterations, the system was powered up and tested. For the first 4 seconds, the output video display worked as desired. However, after 4 seconds, the image began a slow horizontal scroll. This problem was eventually traced to small, but significant differences in the average rate of data into and out of the FIFOs. The source video camera sends data into the FIFOs at 60 frames per second to the accuracy of its clock and the FPGA takes data from the FIFOs at 60 frames per second to the accuracy of its clock. However, a small deviation from the nominal frequency in one or both of the clocks caused the FPGA to
take approximately one less frame every thirty minutes from the FIFO than what the video camera provides. This translates to taking one less line every seven seconds, causing the screen to scroll horizontally approximately every seven seconds as a line of data is lost.

To correct this problem, several words were removed from the blanking lines in the ITU-R BT.656 data stream to slightly increase the output frame rate. This fix creates an ITU-R BT.656 data stream that contains small errors. Specifically, some of the blanking lines are one or two words shorter than specified in the standard. Concerns that this would cause malfunction within the ADV601 proved unfounded as data rates were equalized and the ADV601 operated correctly.

5.4. Motion Enabled Output

As a visual indication of the operation of the motion detection algorithm, the testing and demonstration platform description calls for the output video to be displayed only when motion is present in the video. The one-bit output from the motion detection circuitry, the motion bit, is sent to the FPGA, which controls the format conversion process. When the motion bit goes low, signaling that no motion is present in the input video, the FPGA responds by replacing the ITU-R BT.656 video data with a steady stream of blanking values. It no longer unloads video data from the FIFOs, which quickly grow full, and it no longer inserts code words into the data stream. The blanking values cause the output NTSC monitor to go black, making it appear as though it is off. At the resumption of motion, the FPGA must wait for the frame sync pulse from the video camera. After the frame sync pulse arrives, the FPGA clears the FIFOs and resets its internal counters to create the ITU-R BT.656 video anew, once again causing video to be displayed on the output NTSC monitor.

5.5. Results

Upon completing all changes, evaluation of the system performance began. The format conversion and video compression circuitry was judged on the basis of output image quality. The format conversion provided a reasonably aesthetic image, with only minor problems. Most noticeably and expectedly, the conversion leaves the two black strips at each side of the screen as discussed in Section 5.3. The slight aspect ratio skew and the undisplayed data hidden in the blanking regions were unnoticeable. The only additional problem came from a small horizontal syncing error. The leftmost pixel in each line actually appears in a narrow vertical strip on the right side of the image. Immediately after the FPGA receives the frame sync pulse when recovering from a reset, it clears the FIFOs so that the camera can load data into them. However,
the first pixel in a frame comes simultaneously with the frame sync pulse, just before the FIFOs are cleared.

The video compression itself causes some reduction in image quality, but the resulting output video is sufficient for most surveillance applications. At higher compressed bit rates, such as 7 Mb/s, the only noticeable degradation is graininess and some false contours that resemble the effects of modestly reducing the number of bits per pixel. As the compressed bit rate is lowered to the goal of 2.5 Mb/s, some blurriness appears, but people remain generally recognizable as seen in Figure 16. Because all compression occurs after motion detection, the degradation has no effect on the performance of the motion detection algorithm.

The performance of the motion detection algorithm is judged based first on its ability to trigger on human motion. When a human moves around in front of the camera, the algorithm triggers on that movement for as long as it occurs, displaying the output to the NTSC monitor. If the human stops moving for approximately ten seconds, the declaration of motion expires and the algorithm declares instead that no motion exists, as expected. It subsequently shuts off the output video. The algorithm fails to trigger in the presence of human motion only when that motion moves too quickly through the visual range. For example, an arm swiped once, quickly in front of the camera may not cause the algorithm to trigger. Each pixel location is monitored once every four seconds. If the time that the arm covers a certain pixel is less than .25 seconds, the algorithm may not pick up that motion. Nearly all human motion of interest, however, likely would be of sufficient distance from the camera to make such quick movements across the visual area humanly impossible.
The second performance evaluation tested the ability of the camera to reject repetitive motion. A metronome, providing a large amount of motion but of a repetitive nature, was placed before the camera as shown in Figure 17. The algorithm initially triggered upon the motion, but it quickly realized the repetitive nature of the motion, turning off the output video approximately 10 seconds later. The successful operation of the repetitive motion feature has also been tested and verified with fans and blinking lights.

The next tests examined the algorithm’s ability to ignore small or narrowly-shaped motion. When presented with a close up of the popular star-field screen saver, the algorithm successfully ignores this motion. The many randomly moving white dots on the black background of this
screensaver, though producing significant amounts of motion, are each too small to trigger the algorithm. As a second example, several thin wires were waved in front of the camera without triggering the algorithm. The wires are too narrow to be of human origin.

The final performance measure is the ability of the output image to respond to the presence or absence of motion. The video turns on and off as desired, however, a slight delay of approximately one second exists between the time that motion begins and the time that the video turns on. Though a minor annoyance for demonstration purposes, such a delay would likely cause no problems in surveillance applications.
Chapter 6.

Conclusion

6.1. Summary

The increasing demand for video surveillance in turn demands smarter and more convenient surveillance devices. This project shows that a simple algorithm, exploiting basic observations about motion, can augment the performance of a surveillance device by closely approaching the goal of distinguishing between human and non-human motion. Using a fairly modest processor running at just over 100 MHz and significantly less memory than that available, the motion detection circuit developed in this project accomplished all of its original goals, including recognizing repetitive motion and ignoring small or narrow motion.

Conceivably freeing surveillance devices from the inconvenience of wires, the project demonstrates video compression circuitry that reduces 60-frame-per-second video to a 2.5 Mb/s data stream while maintaining acceptable image quality. It also offers a potential method to pack the video data from four cameras operating at 30 frames per second onto a 2.5 Mb/s data stream. Though not the highest quality compression scheme available, the wavelet compression solution used in this project balances the demands of real-time performance against image quality to meet the data rate requirements of the Security Sphere’s wireless channel.

6.2. Future Work

The intelligent features of the motion detection algorithm operate correctly with the tests performed in the lab, including those using a metronome and a screen saver. The algorithm’s ability to distinguish between human and non-human motion in a larger variety of potentially important cases has yet to be examined. Chief among these are outdoor cases in which wind plays a role. If the surveillance device is to have the potential to monitor the exterior of buildings, it must be able to reject repetitive motion such as the swaying of trees and smaller motion such as blowing leaves. If these tests fail, the solution may lie in simple changes to algorithm parameters, or possibly greater algorithm complexity may be needed.
At the start of this project, real-time implementations of the powerful MPEG compression scheme were hard to come by. In just the one year since development for this project began, the feasibility and availability of real-time MPEG compression has markedly increased. Future work could replace the wavelet compression scheme used in this project with a real-time MPEG encoder. The weakness of the ADV601 compression used in this project rests in the fact that it performs no interframe compression, from which significant reduction in redundancy can be achieved. MPEG compression exploits this interframe redundancy and would yield higher quality images at the output.

Possible changes to the circuit itself are many. Foremost are the changes that have already been wired onto the board. A new circuit board layout could streamline the board, making it more portable and durable. The FPGA should be more greatly utilized to reduce the amount of circuitry. It could, for example, implement the parallel-to-serial conversion required at the board output, replacing the shift register that is currently assigned that task. The current circuit also contains two DSPs. One does the processing for the motion detection algorithm, and the other supports the ADV601 compression chip. With an upgrade to a faster processor or through intelligent use of interrupts, one DSP could handle both tasks, significantly reducing board size and complexity.

Following all fine tuning, cleanup, and other improvements, the eventual goal should be to combine this board with the other components of the proposed Security Sphere surveillance system. The board would, as expected, take video from the four sources, do motion detection, image compression, and send the video in serial format to a transmitter. To complete such a system, an additional receiver section and display unit would need to be developed. A receiver board would convert the data back to parallel format and then decompress it using an ADV601 set to decode mode (or using an MPEG decoder.) Next, a frame grabber could take the video data into a computer where further processing would occur. The four separate images would be unpacked from the data stream into which they were combined. The last step would be to display the four images on the computer screen in a pleasing format.
Appendix A.

Code

A.1. VHDL Video Format Conversion Code

Library synplify;
Use synplify.attributes.all;
Library IEEE;
Use IEEE.std_logic_1164.all;
Use IEEE.std_logic_unsigned.all;
Use IEEE.std_logic_arith.all;
Library xc4000;
Use xc4000.components.all;

Entity SphereBoard is
Port
  fsl: std_logic; -- Frame Sync from Video Camera
  muxenable: std_logic; -- Pixel Sync from Video Camera
  Clk32MHz: std_logic; -- Actually a 27.027 MHz clock
  MOTION: std_logic; -- Motion bit from DSP
  FFL0n: std_logic; -- FIFO full signal from FIFO 0
  even: std_logic; -- Disables motion control of output
  f2: std_logic; -- Eliminates 2 words from a blanked line
  frst: std_logic; -- Eliminates 2 words from a blanked line
  finl: std_logic; -- Eliminates 2 words from a blanked line
  rst8: std_logic; -- Eliminates 5 words instead of 2 in above
  FRESn: BUFFER std_logic; -- Clears the FIFOs
  C6RESn: BUFFER std_logic; -- Resets the DSP
  VID9DT2: OUT std_logic_vector(7 downto 0); -- Makes sync codes for ITU-R BT.656
  FUNCK: OUT std_logic_vector(1 downto 0) -- Unclocks FIFOs
End Entity SphereBoard;

Architecture behave of SphereBoard is
  Signal F,V,H: std_logic; -- Parity check bits
  Signal linecnt: std_logic_vector(9 downto 0); -- Tells us what line we are on
  Signal wordcnt: std_logic_vector(10 downto 0); -- Tells us what sample we are on
  Signal addon: std_logic_vector(1 downto 0);

Begin
  Process
    Wait until falling_edge(Clk32MHz); -- Actually a 27.027 MHz clock
    addon <= fl & (NOT f1);
    FRESn <= (fsl OR FRESn) AND (FFL0n OR (NOT muxenable)) AND (MOTION OR EVEN);
    C6RESn <= (fsl OR C6RESn);
    IF wordcnt = 1715 OR FRESn = '0' THEN
      wordcnt <= "00000000000";
    ELSIF (wordcnt = 1230 AND linecnt = 2) OR
      (wordcnt = 1450 AND linecnt = 3) OR
      (wordcnt = 1451 AND linecnt = 4) OR
      (wordcnt = 1451 AND linecnt = 6 AND fs2 = '1') OR
      (wordcnt = 1451 AND linecnt = 7 AND f2 = '1') OR
      (wordcnt = 1451 AND linecnt = 1 AND frst = '1') OR
      (wordcnt = 1230 AND linecnt = 3 AND finl = '1') OR
      (wordcnt = 1230 AND linecnt = 4 AND rst8 = '1') THEN
      wordcnt <= wordcnt + (addon & '1'); -- Fine tuning to match data rates
    ELSE
      wordcnt <= wordcnt + 1;
    END IF;
    IF linecnt = 526 OR FRESn = '0'

Abstract

The code provided below is a motion detection algorithm written in C. It includes definitions for various variables and structures, as well as code blocks for the main functions of the algorithm. The code is designed to detect motion in an input image and is part of a larger software framework.

A.2. Motion Detection Code

A.2.1. Motion Detection Algorithm C Code

```c
#include <stdio.h>

#define SIZE 32768 /* Total number of monitored pixels */
#define THRESHOLD 16 /* # of pixels that must have motion to trigger algorithm*/
#define TIME 16 /* Lifetime of declaration of motion for a pixel */
#define SHIFT 3 /* 2^SHIFT = noise margin */

struct controller (unsigned int status);
struct controller *TMO_CTRL = (struct controller *) 0x01940000;
  /* provides software access to output pin */

#pragma DATA_SECTION(im, "sect_CE3");
unsigned int im;
#pragma DATA_SECTION(s1, "sect_SRAM");
unsigned char s1[SIZE];
#pragma DATA_SECTION(s2, "sect_SRAM");
unsigned char s2[SIZE];
#pragma DATA_SECTION(s3, "sect_SRAM");
unsigned char s3[SIZE];

unsigned char s1[SIZE];
unsigned char s2[SIZE];
unsigned char s3[SIZE];
unsigned int im;
struct controller *TMOCTRL;
```

A.2.2. Motion Detection Algorithm C Code

```c
linecnt = "000000001";
ELSIF wordcnt = 1715 THEN
  linecnt = linecnt + 1;  -- Increment linecnt at the end of each line
END IF;
IF linecnt = 266 THEN
  F = '1';
ELSIF linecnt = 4 THEN
  F = '0';
END IF;
IF linecnt = 1 or linecnt = 264 THEN
  V = '1';
ELSIF linecnt = 20 or linecnt = 283 THEN
  V = '0';
END IF;
IF wordcnt = 1536 THEN
  H = '0';
ELSIF wordcnt = 2 THEN
  H = '1';
END IF;
P3 = V XOR H;
P2 = P XOR H;
P1 = P XOR V;
P0 = P XOR H P1;  
IF (wordcnt >= 206 AND wordcnt <= 1228) AND {
  ( 8 <= linecnt AND linecnt <= 263) OR
  (270 <= linecnt)) THEN
  CASE wordcnt((1 downto 0) is
    WHEN '00' => FUNC &= "01";
    WHEN '10' => FUNC &= "ZZZZZZZ";
    WHEN '01' => FUNC &= "ZZZZZZ";
    WHEN others => FUNC &= "00";
  END CASE;
ELSE IF wordcnt = 1439 OR wordcnt = 1711 THEN
  VIDE9DT2 &= "111111111";
  FUNC &= "00";
ELSIF (wordcnt = 1440 OR wordcnt = 1444 OR
  wordcnt = 1712 OR wordcnt = 1713) THEN
  VIDE9DT2 &= "0000000000";
  FUNC &= "00";
ELSIF wordcnt(2 = '1' AND wordcnt = 1714 THEN
  VIDE9DT2 &= '1'&F&V&H&P3&P2&P1&P0;
  FUNC &= "00";
ELSIF wordcnt(0) = '1' THEN
  VIDE9DT2 &= "1000000000";
  FUNC &= "00";
ELSE
  VIDE9DT2 &= "001000000";
  FUNC &= "00";
END IF;
END Process;
END Behave;
```
#pragma DATA_SECTION(var, "sect_SRAM"); /* store the var array in the SRAM */
unsigned char var[SIZE];

#define BIT_PAD 0

#pragma DATA_SECTION(toggle, "sect_SRAM"); /* store the toggle array in the SRAM */
unsigned char toggle[SIZE]; /* tracks least recently altered history */
unsigned short int count;
#pragma DATA_ALIGN(bin, 4); /* proper aligns the binary image in mem */
unsigned int bin[SIZE>>5]; /* binary image of pixel motion bits */
unsigned int bin_sum; /* total pixels displaying motion */
unsigned char pix[SIZE]; /* array of current pixel values */
unsigned short int n; /* tracks which pixel on which we operate */
unsigned int change; /* doesn't let processing beat interrupts */

void main() {
    int i;
    int j; /* declare some multi purpose variables */
    unsigned int temp; /* allows for addition without overflow */
    unsigned char stick; /* puts some stickiness on output motion bit */
    unsigned char var[SIZE];

    start:
        if (n < 2) { /* if no motion has appeared at the pixel for several frames */
            bin[n+5] = bin[n+5] & ~((1 << (n & -(-0 << 5)))); /* clear pixel's motion bit */
            var[n] = TIME; /* and reset the time to expiration of motion bit */
        }
        if (((s1[n]-pix[n])>>SHIFT) == 0) || (((pix[n]-s1[n])>>SHIFT) == 0) { /* If pixel history 1 differs from current val by 8 or less */
            temp = s1[n] + pix[n]; /* average history val with current val and store in s1 */
            s1[n] = (temp >> 1); /* set history 2 equal to current value */
            --var[n]; /* indicate that another frame has passed without motion */
        } else if (((s2[n]-pix[n])>>SHIFT) == 0) || (((pix[n]-s2[n])>>SHIFT) == 0) { /* If pixel history 2 differs from current val by 8 or less */
            temp = s2[n] + pix[n]; /* do the same stuff */
            s2[n] = (temp >> 1);
            --var[n]; /* do the same stuff */
        } else if (((s3[n]-pix[n])>>SHIFT) == 0) || (((pix[n]-s3[n])>>SHIFT) == 0) { /* If pixel history 3 differs from current val by 8 or less */
            temp = s3[n] + pix[n];
            s3[n] = (temp >> 1); /* do the same stuff */
            --var[n]; /* do the same stuff */
        } else if (toggle[n] == 2) { /* If current val differs from history and history 3 least recently altered */
            s1[n] = pix[n]; /* set history 3 equal to current value */
            toggle[n] = 1; /* and note now that history 2 is most recently altered */
            bin[n+5] = bin[n+5] | (1 << (n & -(-0 << 5))); /* and set pixel's motion bit */
            var[n] = TIME; /* and set motion bit's lifetime to TIME */
        } else if (toggle[n] == 1) { /* If current val differs from history and history 2 least recently altered */
            s2[n] = pix[n]; /* set history 2 equal to current value */
            toggle[n] = 0; /* and note now that history 2 is most recently altered */
            bin[n+5] = bin[n+5] | (1 << (n & -(-0 << 5))); /* and set pixel's motion bit */
            var[n] = TIME; /* and set motion bit's lifetime */
        } else { /* If current val differs from history but history 1 least recently altered */
            s1[n] = pix[n]; /* set history 1 equal to current value */
            toggle[n] = 1; /* and note now that history 1 is most recently altered */
            bin[n+5] = bin[n+5] | (1 << (n & -(-0 << 5))); /* and set pixel's motion bit */
            var[n] = TIME; /* Do the same stuff only to history 1 this time */
        }
        if (n == SIZE-1) { /* If last pixel in frame start Stage 2 */
            n = 0;
            for(i=0; i<((SIZE)>>5); i++) { /* dilates each 32-bit word using 111 */
                temp = bin[i] & bin[i] & (bin[i]<1); /* ensures motion at least 3 bits tall */
                for(j=0; j<32; j++) { /* determine how many pixels display motion */
                    bin_sum += (bin[i] >> j) & 1;
                }
            }
            if(bin_sum < THRESHOLD) { /* ensure enough motion to surpass THRESHOLD */
                if(stick == 0) { /* and that most recent motion declaration has worn off */
                    TMO_CTRL -> status = 0; /* no motion, set output bit low */
                }
                else { /* indicate another frame has past since last motion detected*/
                    --stick;
                }
            }
        }
}

A.2. MOTION DETECTION CODE

59
else
    TMO_CTRL -> status = 4; /* motion, set output bit high */
    stick = 32; /* reset lifetime of declaration of motion */
}while CHANGE == count) /* wait for next interrupt and new pixel value */
    change = count;
goto start; /* repeat the process with the next pixel value */

interrupt void newpix()
{
    pix[count>>l] = (im>>24); /* store output of 3-state buffers */
    count = count + 30; /* increment the pixel counter */
}

A.2.2. TMS320C6211 Boot and Configuration Assembly Code

.ref _c_int00 ; c_int00 is function that sets up C environment
.ref _newpix ; newpix is name of C-coded ISR function
.sect "heyvecs" ; assembler directive that puts following code into
    ; a section named heyvecs
    ; the CPU jumps here at reset
reset:
    NOP
    NOP
    NOP
    B  myinit
    NOP
    NOP
    NOP

myinit: ; Makes SRAM accesses faster
    MVKL 0x01800008, A4 ; Put the address of the
    MVKH 0x01800008, A4 ; CE0_SCR into A4
    MVKL 0x10514101, B3 ; Write to CE0_SCR
    MVKH 0x10514101, B3 ; a 1/1/1 and 1 configuration
    STW B3, *A4 ; strobe/hold/setup and TA
    NOP
    NOP
    MVLK 0x01940000, A5 ; Put the address of the
    MVLK 0x01940000, A5 ; TM1_CTRL into A5
    ZERO B2 ; Write 0 to TM1_CTRL
    STW B2, *A5 ; to indicate no motion at startup
    NOP
    MVKL 0x01980000, A4 ; Put the address of the
    MVKH 0x01980000, A4 ; TM1_CTRL into A4
    MKV 1, B2 ; Write 1 to TM1_CTRL
    STW B2, *A4 ; to put TSTAT on output to control 3-state buffers
    NOP
    MVKL 0x01980004, A5 ; Write to the Timer 1 Period Reg
    MVKH 0x01980004, A5
    MVK 30, B2 ; Specifies how many pixels between interrupts
    STW B2, *A5
    MVC CSR, B0 ; Globally enable interrupts
    OR 1, B0, B0
    MVC B0, CSR
    indinten:
    MVKL 0x00008002, B1 ; Enable both int15 and NMI
    MVC IER, B0
    OR B1, B0, B0
    MVC B0, IER
    copyinit: ; Copies rest of code from the EEPROM to internal mem
    MVKL 0x90000200, A4
    MVKH 0x90000200, A4
    MVKL 0x90000200, A5
    MVKH 0x90000200, A5
    MVKL 0x00000200, A6
    MVKH 0x00000200, A6
    SUB A5, A4, A1

60
loop:
    { [A1] B done
        LDW *A4++, B3
        NOP 4
        ; branch occurs
        STW B3, *A6++
        SUB A1, 4, A1
        B loop
        NOP 5
        ; branch occurs
    }

    done:
        MVKL 0x01800014, A4
        ; Put the address of the
        MVKH 0x01800014, A4
        ; CE3_SCR into A4
        MVKL 0x10514121, B3
        ; Write to CE1_SCR
        MVKH 0x10514121, B3
        ; the 2 means 32 bits wide
        STW B3, *A4++
        SUB A1, 4, A1
        B loop
        NOP 5
        ; branch occurs

        MVKL 0x01980000, A4
        ; Put the address of the
        MVKH 0x01980000, A4
        ; TM1_CTRL into A4
        MVKL 0x00000081, B2
        ; Write 84h to TM1_CTRL
        STW B2, *A4
        B _c_intOG
        MVKL 0x01980000, A4
        ; Write a 1/1/1 and 1 config
        STW B3, *A4
        B _newpix
        ; to turn on the counter to start counting pixels

        NOP
        ; call occurs

    .sect "heyint15"
int15:
    B _newpix
    ; The CPU jumps here for timer interrupt
    NOP
    NOP
    NOP
    NOP
    NOP
    ; branch occurs

A.2.3. TMS320C6211 Linker Command File

    -c /* follow C conventions for linking */
    heyvecs1.obj /* input file (came from assembly code) */
    hey.obj /* input file (came from C code) */
    -o hey.out /* name the output file */
    -heap 0x200 /* specify the size of heap used by C code */
    -stack 0x200 /* specify the size of the stack used by C code */
    -l rts6201.lib /* the c_int00 function comes from this library */
    MEMORY /* names ranges within the memory map */
    {
        VECS: origin = 0x00000000 length = 0x00000250
        INT15: origin = 0x000001E0 length = 0x00000070
        PMEM: origin = 0x00000200 length = 0x000000E0
        DMEM: origin = 0x00000400 length = 0x000000C0
        CEO: origin = 0x80000000 length = 0x01000000
        CEOCS: origin = 0x90000000 length = 0x000000E0
        CE1INT15: origin = 0x90000000 length = 0x00000000
        CE1PMEM: origin = 0x90000020 length = 0x00000000
        CE1init: origin = 0x90000040 length = 0x00000000
        CEO: origin = 0x80000000 length = 0x01000000
        CE1: origin = 0x00000000 length = 0x01000000
    }

    SECTIONS /* assigns sections to named ranges in memory */
    {
        heyvecs: load=CE1VECS, run=VECS
        heyint15: load=CE1INT15, run=INT15
        .text: load=CE1PMEM, run=PMEM
        .cinit: load=CE1init, run=DMEM
        .const: load=CE1init, run=DMEM
        .data: load=CE1init, run=DMEM
        sect_SRAM: > CEO
        sect_CE3: > CE3
        .clo: > DMEM
        .far: > DMEM
        .stack: > DMEM
        .bss: > DMEM
        .sysmem: > DMEM
    }

A.2.4. TMS320C6211 Hex Conversion Utility Command File

    hey.out /* input file */
-byte /* provides byte addresses for EEPROM programmer */
-memwidth 8 /* specifies width of bus used for booting */
-romwidth 8 /* specifies width of the ROM(s) used for booting */

ROMS
{  
  EPROM: origin = 0x90000000, length = 0x20000,  
  files = (heyl.hex) /* names the output file */
}

A.2.5. Matlab Version of Motion Detection Algorithm

function motion = motalg(F)
  
  % F is an array of images.
  % F(i,j,k) is an integer value contained in [0, 255].
  
  % parameters
  n = size(F,2);
  m = size(F,1);
  pixel_number_threshold = min(n,m)*.9;
  initial_state1 = 128;
  initial_state2 = 128;
  initial_var = 10;
  asympt_var = 20;
  decay = .9;
  decay_threshold = .5;
  weight = .5;
  
  % for clarity
  number_of_frames = size(F,3);
  
  D = double(F);

  % initialization
  S1 = initial_state1;
  S1 = S1(ones(m,n));
  S2 = initial_state2;
  S2 = S2(ones(m,n));
  VAR = initial_var;
  VAR = VAR(ones(m,n));
  TOGGLE = ones(m,n);
  motion = zeros(number_of_frames, 1);
  BINARY = zeros(m,n);
  A = ones(m,n);
  B = A;
  C = A;
  H = A;
  E = A;
  K = A;

  for t = 1:number_of_frames,
    TEST1 = (abs(S1 - D(:,:,t)) < (VAR + asympt_var));
    TEST2 = (abs(S2 - D(:,:,t)) < (VAR + asympt_var)) & ~TEST1;
    TEST3 = TOGGLE & ~TEST2 & ~TEST1;
    TEST4 = ~TEST3 & ~TEST2 & ~TEST1;
    S1 = (S1 + weight*(D(:,:,t) - S1)).*TEST1 + D(:,:,t).*TEST4 + S1.*(~TEST1 & ~TEST4);
    S2 = (S2 + weight*(D(:,:,t) - S2)).*TEST2 + D(:,:,t).*TEST3 + S2.*(~TEST2 & ~TEST3);
    VAR = decay*VAR.*((TEST1 | TEST2)) + initial_var*ones(m,n).*((TEST3 | TEST4));
    TOGGLE = TOGGLE.*((TEST1 | TEST2) + TEST4);
    TEST5 = VAR > (decay_threshold*initial_var);
    BINARY = zeros(m,n) | (TEST5 | (TEST3 | TEST4));

    if and(t>=8, t<13)
      A = [A BINARY];
      elseif and(t>=13, t<18)
      B = [B BINARY];
      elseif and(t>=18, t<23)
      C = [C BINARY];
      elseif and(t>=23, t<28)
      H = [H BINARY];
      elseif and(t>=28, t<33)
      E = [E BINARY];
      elseif and(t>=33, t<38)
      K = [K BINARY];
    end

    BINARY = bwmorph(BINARY, 'fill');
    BINARY = erode(BINARY, ones(10));
    motion(t) = sum(sum(BINARY)) > 0;
  end
G = [A; B; C; H; E; K];
G = -G;
imshow(G);
Appendix B.

Schematics

B.1. Original Motion Detection and Compression Circuit
FIFOs and Tri-State Buffers

C6211 Clock, Interrupts, JTAG, and Unused Pins

B.1. ORIGINAL MOTION DETECTION AND COMPRESSION CIRCUIT
C6211 EMIF, Boot EPROM, and External SRAM

ADV601 and Supporting DRAM
ADSP-2181, Boot EPROM, and SPORT Connection to ADV601

Transmitter Interface

B.1. ORIGINAL MOTION DETECTION AND COMPRESSION CIRCUIT
Power Decoupling C6211

Power Decoupling Other
B.2. Revisions to Circuit for Testing and Demonstration Platform
Xilinx FPGA and Configuration PROM

FIFOs and Tri-State Buffers
C6211 Clock, Interrupts, JTAG, Timers, and Unused Pins
References

4 Lim JS; *Two Dimensional Signal and Image Processing*; pg. 644; Prentice Hall; 1990.
5 Lim JS; *Two Dimensional Signal and Image Processing*; pg. 645-646; Prentice Hall; 1990.
6 Lim JS; *Two Dimensional Signal and Image Processing*; pg. 647; Prentice Hall; 1990.
7 Taken from the Analog Devices ADV601 datasheet; http://www.analog.com/pdf/2011_0.pdf.
11 Fife K; *A Stereo Vision System with Automatic Brightness Adaptation*; Master's thesis; Massachusetts Institute of Technology; May 1999.