Image Chipping with a Common Architecture for Microsensors (CAμS)

Jonathan E. Scalera

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Electrical Engineering

Dr. Peter M. Athanas, Chair
Dr. Mark T. Jones
Dr. Amy E. Bell

July 23, 2001
Blacksburg, Virginia

Keywords: Image Processing, Region of Interest (ROI), Microsensors, CAμS, FPGA

Image Chipping with a Common Architecture for Microsensors (CAμS)

Jonathan E. Scalera

Dr. Peter M. Athanas, Chair

The Bradley Department of Electrical and Computer Engineering

Recent interest has emerged in microsensor platforms that are capable of supporting reconnaissance, surveillance and target acquisition operations. These devices typically consist of one or more sensors, signal conditioning and processing subsystems, a radio link and a power source. Sensors employed can range from acoustic, to seismic, to magnetic, to visible/infrared imagers.

A notable shortcoming of these systems is the fact that they are battery powered. The use of a finite power source places an upper limit on the lifespan of such a system. Thus, a major thrust in the development and usage of these microsensor platforms lies in the conservation of their limited energy resources. In attempt to reduce power consumption and hence extend the system's lifespan, communication bandwidths are often limited. In order to reduce the required bandwidth, much of the signal processing necessary to achieve a desired functionality must be performed within the microsensor platform itself.

This thesis effort provides this crucial bandwidth reduction by implementing in hardware an algorithm developed by the University of Maryland, which limits transmissions to the best view Regions-of-Interest (ROI) data, on the CAμS platform by BAE Systems. The hardware implementation was verified with a Matlab script that compared its results with those of the original algorithm. It was shown that these implementations were consistent for all of the data sets tested. Moreover, a subjective analysis, in which the detected ROIs were visually inspected, was performed to corroborate the former quantitative results.

To my mother, without whom nothing would have been possible.

Acknowledgments

Unfortunately I undertake the writing of this section with the expectation of falling far short from extending thanks to all those who deserve it. This end result is in no way intended, but rather is attributed to the utter impossibility to achieve completeness and thus perfection in such a section. Owing to my poor long-term memory, most of the thanks extended here are to those who have most recently touched my life. Sadly, this characteristic does not reflect the fact that I am indebted to all those who have affected my life in the past and have made me the person that I am today. I therefore begin this section by thanking you all collectively from the past who have created the foundation upon which all these efforts were based.

I first made contact with Dr. Peter Athanas in the spring of my senior year at the University of New Hampshire. Despite our differences of opinion at times and a great deal of sarcasm on my part, Dr. Athanas has been on a relentless pursuit to allow me to direct my research efforts as I please. This ongoing path of persistence began with Dr. Athanas recommending me as a candidate for the Bradley Fellowship. This fellowship opened up numerous doors through which I could choose the direction of my research. Keeping in line with the spirit of the fellowship, Dr. Athanas continued his efforts towards my research flexibility by permitting me to not only choose the initial project upon which I worked, but also afforded me the option to change to another when I so desired. In fact, even when I proposed a new project be brought to the Configurable Computing Laboratory, the project that served as the basis of this thesis effort, he welcomed it with open arms. I cannot thank Dr. Athanas and the Via family enough for this ongoing flexibility. Such flexibility is typically only dreamed about by graduate students.

However, it should be noted that none of this work could have taken place without the support both

financially and personally from the people at BAE Systems (formerly Sanders). Through both my internship last summer and their collaborative efforts throughout this project I have established numerous friendships that I expect to last my entire lifetime. It was only with these new found friends and the excellent team of students and professors here at Virginia Tech that permitted this project to become a success. The fully functioning implementation that resulted from these efforts was far from the Mark Falco dreaded incomplete and thus useless final product, which he referred to as a "paper weight".

Dr. Jones was not only the individual who introduced me to the Configurable Computing Laboratory, but also to the field of configurable computing itself. His constant downplaying of oftentimes difficult concepts and tasks served as a great motivation for me to figure out things by myself and to catch up to speed to meet what I perceived to be his expectations. Oddly enough, it is only now that I have gotten to know him personally that I realize that these ongoing understatements are a direct result of Dr. Jones' high intelligence and simple grasping of high-level concepts, as well as, details. Throughout my career at Virginia Tech I have stopped by Dr. Jones' office numerous times to tap into his wealth of knowledge. Each and every visit was a worthwhile venture. Dr. Jones, I thank you for your assistance throughout.

Dr. Bell taught me that professors can not only be great mentors, but also can become excellent friends. As a student in her DSP class, I developed a great respect for her thoroughness and dedication to her students; however, it was not until we began to share common difficult experiences in our lives that I began to see her as more than a great professor. She had become a true friend. In some sense, it could be said that this friendship serves as the basis for my high opinion of Virginia Tech. Dr. Bell, I thank you and I hope that you continue to touch the lives of your students.

In addition to Dr. Athanas, Dr. Jones and Dr. Bell, I would also like to thank the others that assisted in bringing this thesis to fruition. Most certainly I want to thank my family who took time out of their busy schedules to read the boring technical chapters that follow. Also, I extend thanks to my outstanding friend, Dennis Collins, who took time out of his busy golf schedule to proof read some chapters. I commend him for standing true to our friendship and pulling through for me during a difficult time.

Speaking of friendship, I would like to thank my friends in the Configurable Computing Laboratory.

They not only accepted my poor humor in the lab, but also listened with a smile. Specifically, I would like to thank Jonathan Ballagh, who was always on the ball and pushed me to achieve his high standards. Zahi and Shashank have both taught me that although people come from completely different backgrounds, they can still develop into individuals who are considered universally great persons. The same can be said for Dennis Collins and Jon. Even though I have only mentioned a few people's names, this attribute is true for all the students that work in the Configurable Computing Laboratory. You all are great colleagues who have provided me with a great deal laughs and I can only hope that I have somewhat returned the favor. I wish you all the best of luck in your future endeavors. The last member of the Configurable Computing lab that I would like to thank is Wendy Akers. She is an excellent conversationalist and has always reminded me that good kind-hearted innocent people still exist in the world today.

Finally, I would once again like to thank my family. Certainly, their assistance throughout this thesis effort is commendable, but more notable is their persistent support and high standards that have driven me throughout my life. Without them being the excellent people that they are, I would never have become half the man that I am today. With their continued achievements and backing, I know that I will continue to grow as an individual as I attempt to keep up with their fast moving pack. As a final thought, I extend a special thanks to my brother Steve who I often called for advice and whose innate talent to pull all the strings necessary at BAE Systems established my summer internship and this project here at Virginia Tech.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

It is the belief of the United States Army that it has "a non-negotiable contract with the American people to fight and win our[its] Nation's wars"[1]. This belief stems from the fact that, by law, The Army is responsible for defending the United States and its Territories; supporting national policies and objectives; and defeating nations whose aggression endangers the peace and security of the United States[2]. It is noted that fulfilling this responsibility is a formidable task. After all, on any given day, "more than 140,000 Army personnel are forward stationed or deployed around the world"[2]. Moreover, it is mentioned that the challenge of upholding this responsibility is increasing every day. Consider the fact that since 1989, Army contingency deployments have increased from an average frequency of one every four years to one every fourteen weeks[1, 2].

Although the 21^{st} century holds great promise for the United States, it also poses many potential threats. With the advent of new technology comes increased dangers to national security. Potential dangers include:

> regional of the kind we see in the Balkans, in Southwest Asia, and on the Korean Peninsula; threats that cut across geographical and ideological boundaries, including the drug trade, ethnic, tribal and religious strife, and organized crime; and "asymmetric" dangers, such as terrorism, the threatened use of weapons of mass destruction, and information warfare[3].

To combat these emerging threats General Eric K. Shinseki, Chief of Staff, and Louis Caldera, Secretary of the Army, have proposed The Army Vision, "Soldiers on point for the Nation...Persuasive in Peace, Invincible in War"[4]. Based on this Vision, Shinseki hopes to dominate in what he refers to as the four rules of thumb of war[3]:

- We want to initiate combat on our own terms – at a time and place and with a method of our choosing.
- We want to gain the initiative and never surrender it.
- We want to build momentum quickly.
- And we want to win decisively.

In order to bring the Vision to fruition, The Army will undergo a transformation as proposed by The Transformation Strategy to become more responsive, dominant, deployable, agile, versatile, lethal, survivable and sustainable than present forces[1, 3, 4]. The Transformation Strategy not only calls for changes in personnel and training, but also for improvements in technology, which will vastly enhance the capabilities of these forces. To motivate these technological improvements The Army has posed the following three question to the science and technology community.

> How do we reduce armored volume in combat vehicles while increasing survivability? How do we increase deployability without sacrificing survivability and lethality? How do we reduce in-theater support needs, and thereby reduce strategic lift requirements?[2].

Merely the fact that the President's Budget for FY2001 called for $1.3 billion to be used for the answering of these questions and others[1], goes to show how important these solutions are to the United States. Of this $1.3 billion, $500 million was allocated for the development of future combat systems[1]. Future Combat Systems (FCS) is a joint effort of the U.S. Army and DARPA (Defense Advanced Research Projects Agency) that is charged with the challenge of developing such systems. Recognizing that microsensors are capable of supporting reconnaissance, surveillance and target acquisition (RSTA) operations[5], all of which fit into the The Army's Vision of a more technologically advanced force, a recent thrust in funding for the development of such platforms has arisen.

Microsensor platforms typically consist of one or more sensors, signal conditioning and processing subsystems, a radio link and a power source[6]. Sensors on these devices can range from acoustic, to seismic, to magnetic, to visible/infrared imagers[7]. An extremely attractive attribute of these systems is their ability to provide continual coverage of strategic battle areas without requiring personnel (after dissemination)[7]. With this coverage, these microsensors provide situation awareness of the battlefield that can be used to make key tactical decisions.

However, a major drawback of these devices is the fact that they are battery powered. By using a finite power source, the lifespan of the system becomes finite as well[8]. Thus, a major thrust in the development and usage of these microsensor platforms lies in the conservation of their limited energy resources. In attempt to reduce power consumption and hence extend the system's lifespan, communications bandwidths are often limited[5]. Owing to this limitation, much of the signal processing required to achieve a desired functionality must be performed on the sensor node itself[6].

One example in which this preprocessing is of particular importance, is when a imager is interfaced to such a system. After all, the extraordinarily large bandwidth required to transmit full frame-rate video significantly reduces the lifespan of these system. Hence, a means by which to reduce this required bandwidth must be developed. It is the objective of this thesis to address this issue by implementing a solution that provides this crucial bandwidth reduction. That is, this thesis effort consisted of the implementation of an algorithm developed by the University of Maryland, which achieves this bandwidth reduction by limiting transmissions to only best view Regions-of-Interest (ROI) data, on the CAμS (Common Architecture for Micro-Sensors, pronounced as "cause") platform by BAE Systems (formerly Sanders).

1.2 Previous Research

Prior to the commencement of this thesis effort significant research and development had been undertaken. The results of this work provided the foundation upon which this thesis is built. This foundation consisted of two major components that were amalgamated by this thesis' effort.

The first of these fundamental building block corresponded to the previous efforts of BAE Systems.

BAE Systems had designed, constructed and debugged its first generation CAμS system, the 6.1 effort. Furthermore, an acoustic Line of Bearing (LOB) algorithm had been mapped onto this hardware for the detection and localization of ground vehicles[5]. Additional functionality was added to this algorithm to "take the acoustic Line of Bearing (LOB) estimates and cue a mirror-based panning assembly containing an infrared imager"[5].

Occurring in parallel with the BAE Systems' effort, the second building block was developed, which consisted of an image processing algorithm written in Matlab by the University of Maryland. This algorithm, referred to as the UMD image chipping algorithm throughout this thesis, determined which fragments of the incoming video stream needed to be transmitted. That is, it reduces the transmitted data by the selection of ROIs for every frame and the determination of which of these ROI is the best among multiple frames. The ultimately selected and transmitted ROIs are referred to as chips in this thesis; hence, the chosen naming of the developed algorithm. To clarify this best view selection process, the following example is presented.

Consider the case where a tank directly approaches a camera and then recedes. As the tank approaches, it encompasses a larger and larger portion of the field of view of the camera. On the contrary, in the process of receding this portion and therefore quality of imagery reduces. For each of the frames captured, the algorithm determines a RIO, which in this example would correspond to a bounding box containing the tank in the given frame. Considering that when the tank is at its closest point the captured images possess the most complete ROI, it is at this inflection point between approaching and receding that the algorithm will deem the best view and will select as the ultimate chip.

Certainly, there are numerous variations of this presented example, but it is hoped that this one has established a course picture in the readers mind of the top-level functionality provided by the UMD image chipping algorithm. Further discussions in this thesis will expound on this brief introduction to elucidate this concept.

1.3 Significance of this Thesis

As was mentioned above, it was the work associated with this thesis that brought the two past efforts together and integrated them to a sole solution. Considering that prior to this thesis effort the video from the IR imager was continuously transmitted on a separate communications link[5], it was critical that such work was undertaken to reduce transmitted data, which in turn extended the life of the system. Thus, the UMD image chipping algorithm was mapped onto the CAμS 6.1 stack as the crux of this thesis effort. Supplementary efforts were directed towards the design, fabrication and verification of an additional piece of hardware that was employed to simplify the implementation of this algorithm. This hardware, the CAμS video board, permitted the existing hardware the ability to interface with both analog and video streams. A final notable benefit attained in addition to the lower power achieved by this implementation, is the capability to corroborate the results of the acoustic sensor that it provides[5].

1.4 Thesis Organization

This thesis is organized as follows. Chapter 2 presents a brief overview of microsensors and the challenges faced by them. To assist in this discussion other existing microsensor platforms will referenced and a indication of how their chosen designs address these challenges is presented.

Chapter 3 discuss the pre-existing CAμS hardware prior to this thesis effort. Both a top-level discussion of the design decisions made that lead to resulting hardware and a detailed presentation of the individual boards that makes up the system are given. Finally, this chapter concludes by introducing the motivations for the supplementary hardware, the CAμS video board, developed as part of this thesis effort.

In Chapter 4, the discussion of the CAμS stack resumes with a continued presentation of the CAμS video board. This chapter lays out the specifications placed on the video board and how they affected its design flow. The resulting hardware is broken down into its subcomponents, each of which is discussed in detail. Finally, the chapter culminates by describing how this new board interfaced with the existing hardware and the software Application Programmers Interface (API)

written to assist in this process.

Chapter 5 introduces the reader to the top-level functionality provided by the UMD image chipping algorithm. Then, the algorithm's specific computations preformed to achieve its functionality are discussed. Finally, a presentation of the partitioning of its operations across the hardware components of the CAμS stack and the reasons for this chosen division of labor is presented.

Chapters 6 and 7 give detailed discussions of the hardware and software implementations, respectively, which abide to the division of labor called out at the end of Chapter 5. Each of the sub-blocks contained in these implementation are thoroughly discussed and its operations performed are related back to the top-level algorithm discussion presented in Chapter 5.

In Chapter 8, two essential tools employed in the debugging and verification of the implemented algorithm are introduced. The first, a Windows NT based application entitled *Chip View* permitted the ability to display raw image data received from the CAμS stack pictorially on a personal computer as just one of its many features. Second, a Matlab script which built upon the capabilities provided by the *Chip View* application to be used in the verification process discussed in Chapter 9.

Chapter 9 addresses the means by which the implementation was verified both quantitatively and qualitatively. A presentation of the modifications made to the original algorithm to improve its performance and to address the limitations inflicted by hardware is also given. This chapter culminates with a discussion of the performance exhibited by the implemented algorithm and the resource utilization required by it.

Finally, Chapter 10 provides concluding remarks, as well as, outlines future work that could be undertaken to improve upon the efforts of this thesis.

Chapter 2

Overview of Related Work

The development of microsensors and the associated networks that incorporate them comprise a multifaceted engineering feat. Owing to this complexity and the financial backing of the United States government, many Universities and private industries are collaboratively working to solve this formidable problem. This chapter attempts to shed some light on the breadth of challenges faced by this microsensor developmental effort. It is hoped by introducing the reader to these challenges he will obtain a better understanding of how the effort placed forth by this thesis fits into the bigger picture of microsensor research. Even though there is a great deal of overlap between them, this chapter is broken into two sections, Microsensor Nodes and Microsensor Networks, to ease its reading. These sections are presented below.

2.1 Microsensor Nodes

As was noted in Chapter 1, microsensor platforms typically consist of one or more sensors, signal conditioning and processing subsystems, a radio link and a power source[6]. Typical military applications of microsensors include reconnaissance, surveillance and target acquisition operations[4]. However, it is noted that these devices can also be used for much more benign tasks, such as security, machine monitoring, remote medical patient care, inventory control among others[9]. To accomplish these above-listed tasks, microsensors employ one or more sensors to gather information

about their surroundings. Common sensors, particulary for military applications, include acoustic, seismic, magnetic and visible and/or Infrared imagery[7]. By incorporating multiple sensor into these devices, one can corroborate the results of any given sensor[5], which in turn will give the node a better perception of its surrounding environment. This improved awareness is extremely important to the quality of data produced by these node. For instance, by corroborating the results of multiple sensors, a microsensor performing an automatic target recognition (ATR) algorithm could significantly reduce the number of false detections that it finds.

The requirements placed on these microsensor platforms are numerous. Seeing as these sensors are often placed in hostile environments where human intervention is difficult if not impossible, they are expected to work for long periods of time off their finite energy source[8]. Moreover, they have to be small, lightweight enough to be hand-carried and most importantly inexpensive[5]. Microsensors are expected be deployed by either strategically placing them by hand or by dropping form an aircraft[8]. Although a benefit to the latter method, small sized sensors are paramount in the former. After all, it has to be remembered that "a soldier can only carry so many items with him, and if the system is to be hand deployed, it should not significantly reduce the space in his rucksack for food and other necessities."[5]

It is noted that a reduced size is not only essential to making these nodes easy-to-deploy, it is also critical in making these microsensor inexpensive[10]. Unfortunately, a physically reduced size leads to a reduction in the size of the battery as well, which further limits the energy resources of such devices. Certainly, to help counteract this reduction, solar cells could be employed to gather energy from the sun[10]. However, even with this addition the majority of the nodes energy will come for the battery. Thus, to extend the lifetime of these sensor nodes, energy conscious strategies must be used to reduce power consumption. Power reduction techniques are exhibited throughout the microsensor developmental effort. They span from the initial design of such nodes to algorithms implemented on them to communications between them. A few of these techniques are explored below.

Because limited data transmission is paramount in extending the lifespan of these systems, many of the computations required to implement the above-described functionalities are pushed down into the sensor node itself[6]. To handle these operations, microsensors need to incorporate some sort of

processing element. Potential candidates to accomplish this processing include application specific integrated circuits (ASIC), general purpose processors (GPPs) and field programmable gate arrays (FPGAs)[6]. Of the microsensor platforms studied, none of them made use of ASICs, which is most likely due to the fact that these prototypes were comprised of custom circuit boards with mostly commercial, off-the-shelf (COTS) components[11].

However, ASICs were specifically avoided by BAE Systems in the development of its CAμS microsensor platform because it was believed that either many ASICs or one large one capable of interfacing to multiple sensors would have to be developed to permit flexility in the design. In the latter case, it is noted that flexibility is limited because the large ASIC developed would only be capable of interfacing with a fixed number of pre-determined sensors. The former case, the development of multiple ASICs, was also undesirable because it required one to incur the non-recurring engineering (NRE) costs associated with each ASIC development cycle to afford flexibility[6].

Because power reduction is key, it is important to select a GPP for low power embedded systems, such as the StrongArm, MIPS, Motorala 56xxx, when one is employed[6]. Such processors were selected for the micro-Adaptive Multi-domain Power-aware Sensors (μAMPS) sensor node developed at Massachusetts Institute of Technology (MIT) and the CAμS microsensor platform developed by BAE Systems, which employ a StrongARM SA-1100 and Motorola 56307 as their GPPs, respectively[12, 6]. The microsensor platform developed by Rockwell Science Center also makes use of a StrongARM SA-1100 processor[13].

Even though low power GPPs are a reasonable choice to handle the processing demands placed on a microsensor platform, it is noted that their performance is limited by the resources that they provide. Considering that the architecture of GPPs cannot be tailored to a specific application, they pay for this inability by suffering in performance. This reduction in performance may lead to an increase in power consumption owing to the increased number of devices required to combat it[6]. Moreover, it is noted that a GPPs power consumption cannot be reduced by controlling the number of gates switching simultaneously[6].

Due to these above-described limitations of GPPs, the CAμS microsensor platform incorporated an FPGA in addition to its Motorola DSP. FPGAs provide the two capabilities that were outlined as weakness of GPPs. Namely, FPGAs can be tailored for the specific application at hand and

provide control over the number of simultaneously switching gates. Because previous efforts in reconfigurable computing showed "that a dynamic matching between application and architecture led to spectacular energy savings for signal-processing applications," the PicoNode project at Berkeley Wireless Research Center (BWRC) plans to incorporate configurable processing elements in their architecture as well[14].

Although GPPs do not afford the same desirable functionality that FPGAs do, power reduction can still be achieved with them. By employing dynamic voltage scaling (DVS) processor power consumption can be reduced. DVS involves the reduction in both the processor voltage and "clock frequency together to trade off latency for energy savings"[11]. DVS is controlled by the embedded operating systems that run on these devices. These operating systems can further the reduction in power by dynamically powering down some or all components on the node when no interesting events are occurring[11].

It is noted that software can also be structure in such a way that computational accuracy can be traded off with energy consumption. Combining these two concepts, DVS and computational accuracy, MIT has shown that by varying the number of filter taps of a finite impulse response (FIR) filter and using DVS, up to 60% less energy can be consumed compared to a fixed voltage processor[12]. Finally, high-level low power application programming interfaces (API) that hide the underlying complexity of these nodes can be developed[11]. These APIs will permit end-users, whom may be experts in their field but not necessarily in distributed wireless networks, to create low power designs.

Seeing as radio communications on microsensor platforms consume the most power, techniques that reduce this consumption need to be developed. The UMD image chipping algorithm implemented as the basis of this thesis effort addresses this power reduction aspect by minimizing the required transmissions over the radio link. In addition to reduced bandwidth requirements, other novel techniques can be employed to reduce the radios power consumption in these platforms. For example, power control of the radio can be used to minimize the transmit power needed for a node to communicate with its neighbors[15].

Nonetheless, this last point introduces the fact that microsensors typically belong to a network of nodes. A great deal of the communication performed by nodes will be with others in the network.

"Since communications is the driving power factor, maximizing the information content of what is transmitted is essential"[5]. Thus, techniques to reduce these inter-node communications have been and still need to be developed to achieve power reduction. These networks and the power saving techniques used in them are discussed in the following section.

2.2 Microsensor Networks

Microsensor networks must address many issues. One such "issue which includes trades among cost, functionality, and reliability is whether the network should be constructed of homogeneous or heterogeneous elements"[16]. An argument for heterogenous nodes is the desired ability to have nodes with higher power radios to transmit the messages of other nodes in the network back to a user at distant ground station. In such a heterogenous network, the remaining nodes can have lower power radios that will permit them to lengthen their lifespan. On the other hand, homogenous networks benefit from the cost reduction achieved by mass production of a single sensor node type.

Considering that nodes have finite energy sources, as time progresses more and more sensors will go out of service because of their exhausted batteries. The sensor network must have the ability to adapt to such changes while at the same time make the best use of the remaining sensors[8]. Thus, owing to this characteristic, it could be said that the network needs to be self-healing[17]. Node placement is another important issue that has to be addressed. Both the determination of optimal placement for nodes that are strategically placed by hand and algorithms to handle the random placement that occurs when nodes are disseminated from an aircraft have to be developed.

Network traffic in these microsensor network can be classified into three main groups including *user-to-sensor* traffic, *sensor-to-user* traffic and *sensor-to-sensor* traffic[13]. *User-to-sensor* traffic includes user commands and queries sent to the network. *Sensor-to-user* traffic is composed of reports sent to the user from the sensor. Finally, *sensor-to-sensor* traffic is used for collaborative signal processing by sensors in the network before results are presented are presented to the user[13]. This description of this final type of traffic touches on the fact that sensors can share intermediate results to produce an overall better response. That is, similar to the corroboration of multiple sensors on a single node, the results from many sensor nodes can be corroborated to produce a

higher degree of situational awareness[16].

Just as power management at the node level led to significant savings in power consumption, so too can management at the distributed system level. Studies at Virginia Tech have shown that by intelligently powering down redundant nodes in microsensor networks, substantial increases in the lifetime of these networks can be achieved[18].

Remembering that communications is the ultimate power consumer, the most power reduction will be achieved by minimizing the amount of communication required in the microsensor network. This reduction can be achieved through the development of efficient routing schemes. That is, by optimizing the protocol stack, significant power reduction can be gained. Although routing protocols are not addressed by this thesis, the following example has been included to exemplify how an efficient protocol layer can improve power consumption.

The Media-Access Control (MAC) layer can affect power consumption in two manners. First, "careful control of access to the aether reduces the number of wasted transmissions, corrupted by interference of neighboring nodes"[14]. Second, power-management by the MAC-layer can minimize the standby power of a network by developing a tight coordination between the nodes that allows them to wake up their radios exactly when they need to transmit or receive data[14].

Although numerous other techniques for the reduction in power at the network system level either exist or are being developed, the discussion of these techniques is halted at this point so as not to stray to far from the focus of this thesis. After all, the reader is reminded that the implementation developed by this thesis effort addressed this desirable power reduction aspect at the node level not at the network system level. Thus, the reader is asked to seek additional information pertaining to network system solutions from alternative sources.

2.3 Summary

Even though the discussions presented in this chapter are far from exhaustive, it is believed that they are sufficient to give the reader a bit of insight to the challenges posed to the microsensor developmental effort. It is noted that a common thread of desired power reduction has run through-

out these discussions. This concept was emphasized because it is paramount that power reductions are achieved whenever possible. After all, reduction in power consumption directly extends the lifespan of the system, which permits it to provided is assigned functionality longer. It is solely this provided functionality that justifies the microsensors existence.

Chapter 3

CAμS Stack

As an attempt to satisfy the growing interest in the development of a small, lightweight, inexpensive, low power and easily deployable microsensor platform, Sanders, a BAE Systems Company, designed and constructed its first generation common architecture for microsensors (CAμS), pronounced as "cause". Even though this original CAμS architecture, the 6.1 effort, was developed as a proof of concept, it is extremely capable and well thought out. Seeing as the CAμS 6.1 sensor node served as the platform for the implementation of the image chipping algorithm, which was the crux of this thesis effort, this chapter briefly introduces the platform and discusses the hardware of which it is composed.

Albeit the original CAμS platform was extremely capable, it lacked the ability to handle video processing, a desirable characteristic for a microsensor platform. Thus, as a means to meet this desired functionality, rose the necessity for an additional piece of hardware to be developed. This chapter concludes with a complete discussion of the motivations for this supplementary hardware, which is covered in detail in the next chapter.

3.1 Existing Hardware

Shown in Figure 3.1, the original CAμS stack was composed of four boards that abided to the PC-104 form factor (4" by 4" stackable cards). Since experience had shown that "FPGAs are well suited

to the development of deeply pipelined datapath applications typical in signal processing, while general purpose processors are better suited to system control and communications applications"[6], the proof of concept system was based on a commercially available DSP processor and FPGA. It was intended that the FPGA would perform most of the computationally intensive portions of an algorithm while the general purpose processor would be lightly loaded handling control, communications and housekeeping tasks[6].

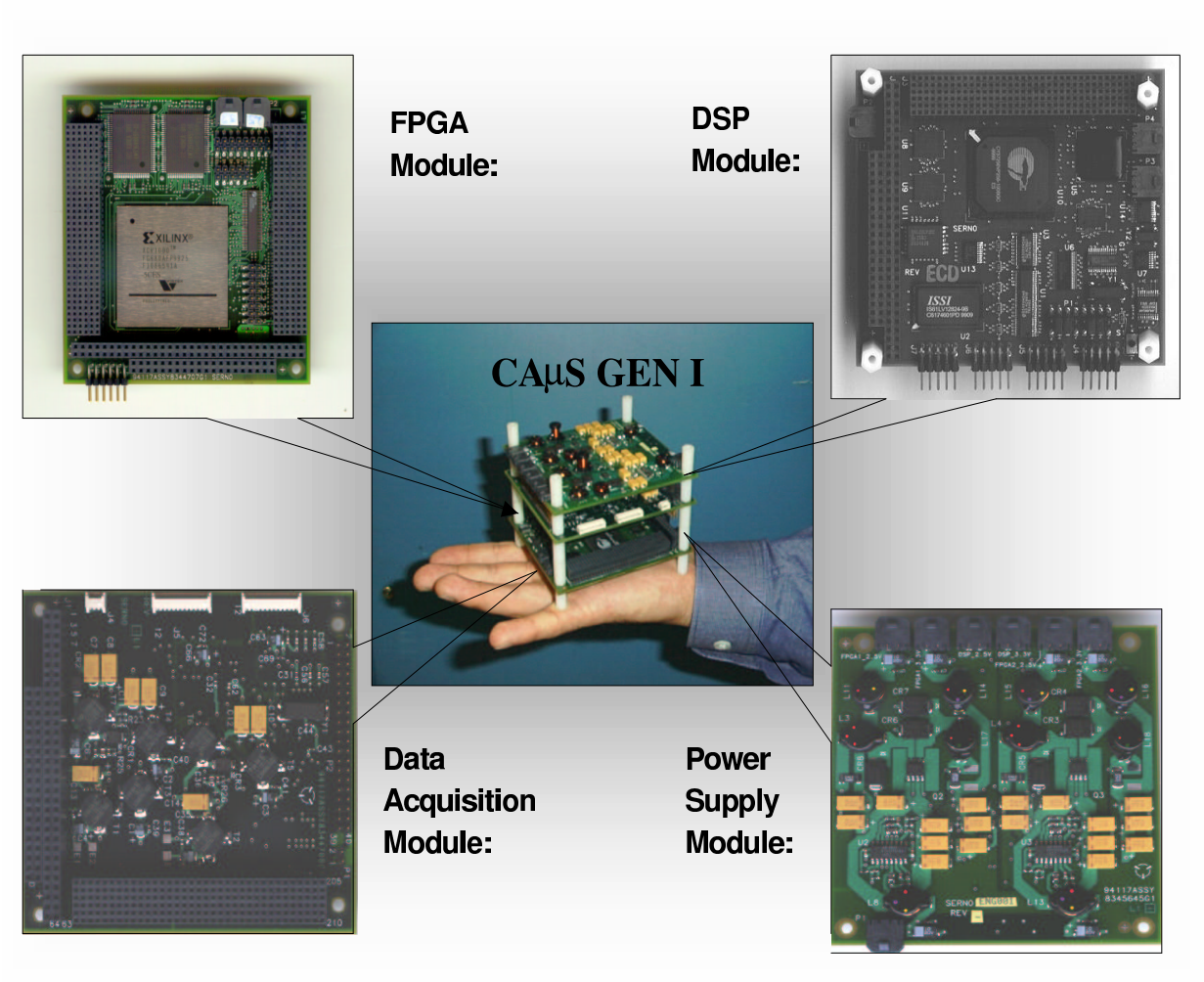


Figure 3.1: Original CAμS Stack

Although application specific integrated circuit (ASIC) technology is just as capable of addressing the performance and power requirements of a microsensor platform, they were avoided because they

correspond to a static solution. Without flexibility, each sensor type would require a unique design or a large ASIC that is capable of supporting a finite number of preexisting sensors would have had to been developed[6]. Moreover, by constructing a configurable platform, the non-recurring engineering (NRE) costs associated with an ASIC's development cycle are circumvented. A notable characteristic of ASICs is that a large production run, which is not typical of military systems, is required to make them a cost effective option[6]. A final mentioned downfall of ASICs is that once one has been developed, it is "locked into the current silicon processing technology (number of usable gates, power/gate/frequency, operating voltage etc)", and therefore cannot leverage the benefits gained from "subsequent silicon process advancements unless it is converted or re-designed"[6].

Consequently, to avoid the above-listed pitfalls associated with ASICs, the CAμS platform was designed with a commercial off the shelve (COTS) FPGA and DSP as its central processing core. To support this processing core, additional components were added, such as memory and configurable clocks. To familiarize the reader with the CAμS 6.1 architecture, each of the boards and its associated components is discussed in the sections that follow.

3.1.1 Field Programmable Gate Array (FPGA) Board

As can be seen in Figure 3.2, the CAμS FPGA board was composed of a Xilinx Virtex XCV1000 FPGA and five Samsung KM736V849 256Kx36-Bit Pipelined NtRAM^{TM} memories. Additional support hardware on the board includes a clock driver chip to minimize clock skew and a bus multiplexer to permit the selection of one of the two user I/O banks. Two 210-pin stacking connectors are used to route the User I/O, Intra-FPGA/DSP bus signals between boards. Considering that the stack was designed to permit the usage of two FPGA boards, the IFU bus was incorporated to be used for the transfer of data between FPGAs and the User I/O bus was split into two selectable banks so that the FPGAs could either operate off the same external inputs or off unique ones. Finally, signals from the DSP (i.e. address and data buses) were routed from the stacking connector to the FPGA to allows its interfacing.

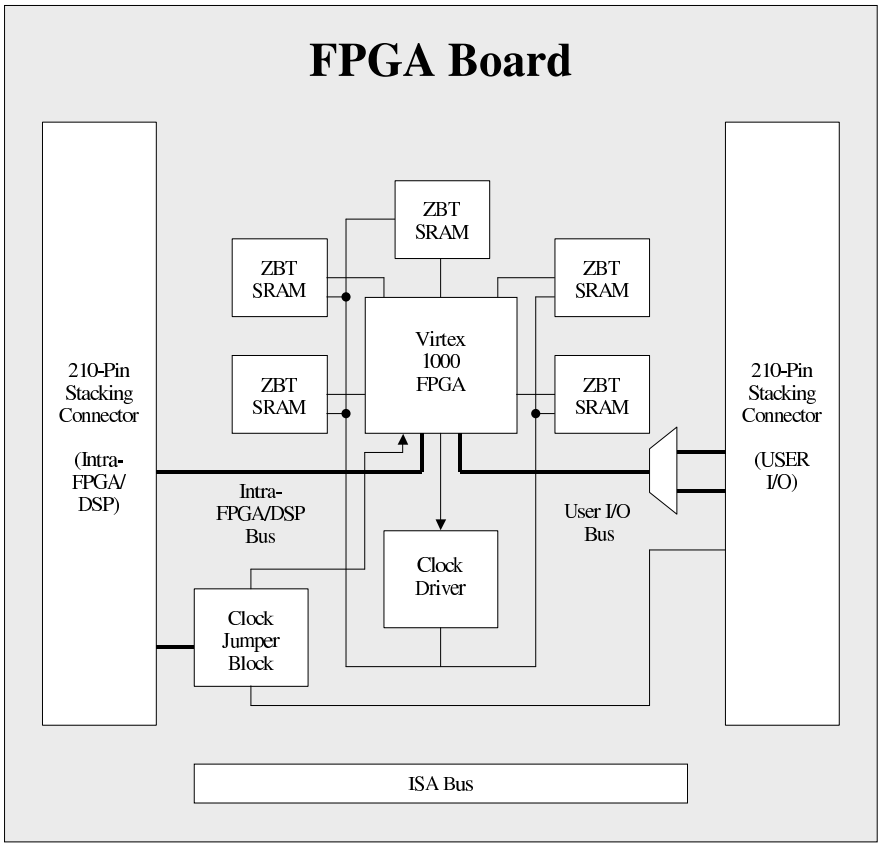


Figure 3.2: Block Diagram of CAμS FPGA Board

3.1.2 Digital Signal Processing (DSP) Board

The DSP board, whose block diagram is depicted in Figure 3.3, is composed of a Motorola 56307 DSP processor and additional support devices. The ICS525 configurable clock by ICS, whose configuration bits are contained in a latch, is used as the input clock to the processor. The Cypress CY37256P256 CPLD acts as glue logic between the processor, the memories and FPGA. The CPLD is also responsible for programming the FPGAs from the configuration flash. The configuration flash is capable of containing two XCV1000 bitfiles, which permits the usage of two FPGA boards, or two configurations for a single board. The boot flash holds the program to be executed on the DSP while the SRAM is used for run-time storage. Finally, a Watchdog real time clock (RTC) has been included to reset the processor if it should ever enter an unknown state or infinite loop.

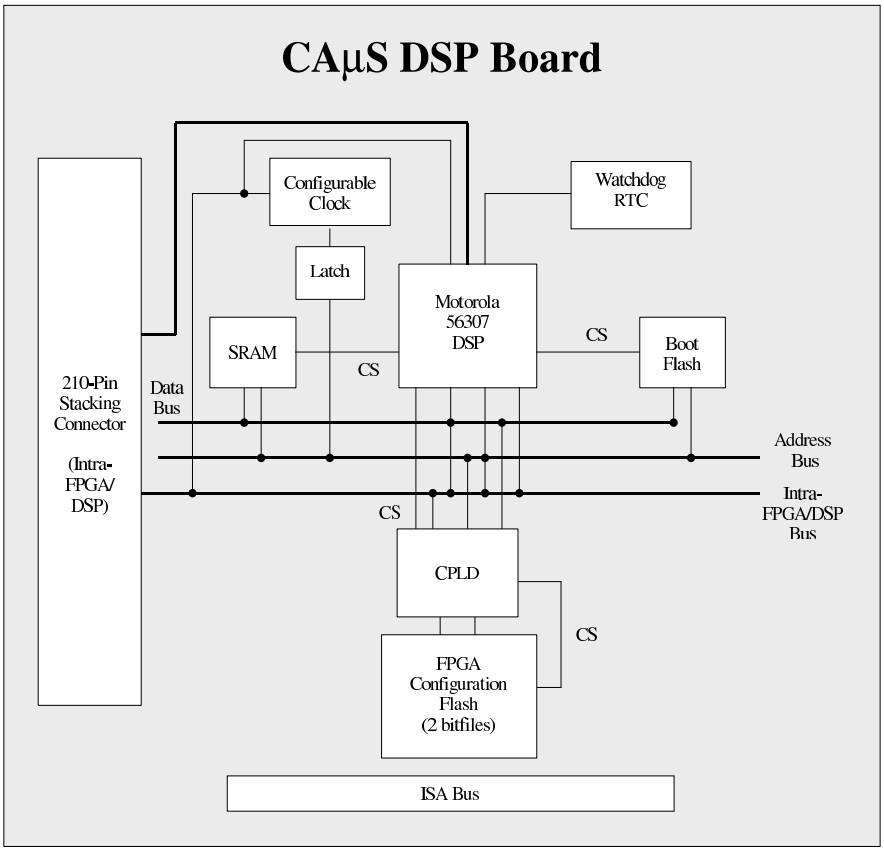


Figure 3.3: Block Diagram of CAμS DSP Board

3.1.3 Data Acquisition Board

The CAμS data acquisition board allows the ability to process analog signals. As can be seen in Figure 3.4, two Analog Devices AD73360 sigma-delta analog to digital convertors (ADC) have been included, which sample the analog signals and translate them to digital signals that can easily be accessed through a serial port interface. An ICS525 configurable clock was used for the sampling clock. The frequency of the sampling clock is set with a jumper block. Two Linear Technology LT1433 DC/DC convertors were incorporated to step the 12 volts passed up through the ISA bus to 3.3V, +7.5V and -7.5V to be used by external devices. The signals of the ADCs are routed to a 210-pin stacking connector the contains the User I/O bus so that they can be accessed by the FPGA.

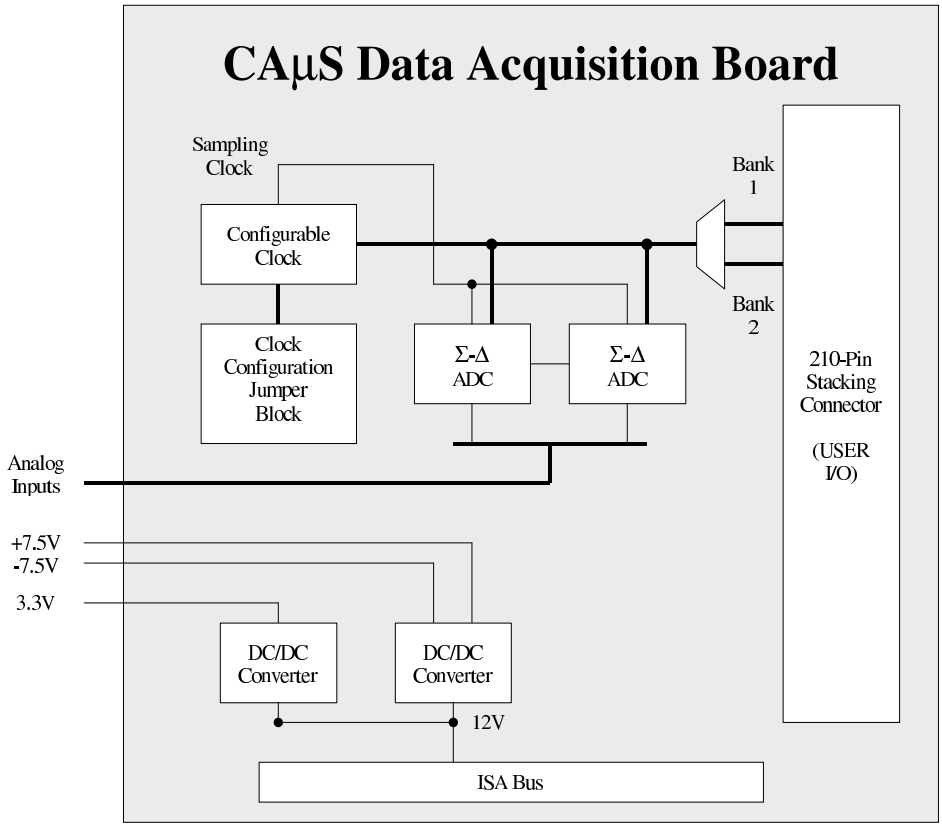


Figure 3.4: Block Diagram of CAμS Data Acquisition Board

3.1.4 Power Supply Board

The CAμS power supply board employs three Linear Technology LT1433 DC/DC convertors and the associated discrete components required to step the input supply voltage of 12V down to the voltages required by the remaining boards in the stack. As can be seen in the block diagram of this board depicted in Figure 3.5, both the FPGA and DSP boards required two voltages to operate, 2.5V and 3.3V.

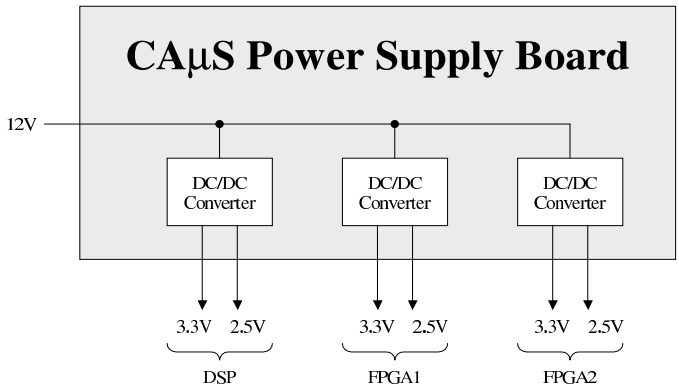


Figure 3.5: Block Diagram of CAμS Power Supply Board

3.2 Supplementary Hardware

The above sections have present a brief overview of the original hardware of the CAμS 6.1 microsensor stack. However, an additional board has been developed as a part of this thesis effort that expanded upon the already overwhelming functionality provided by the CAμS stack. The following subsection discusses the motivation for the creation of this additional hardware while the next chapter presents a detailed discussion of the new hardware that came into fruition.

3.2.1 Motivation

Even though the original hardware of the CAμS stack provide a great deal of functionality, the type of sensors that it could process was limited. That is to say, with a simple ADC front end to the analog signals, processing of complicated analog signals was quite difficult. One such complicated signal that was considered desirable to process, was an analog video stream. After all, having a sensor that can *see* is extremely useful in many battlefield and civilian environments. In fact, the image chipping (region of interest detecting) algorithm discussed later in this thesis made use of this ability to *see* to detect tanks and personal passing the senor node.

Furthermore, it is noted that video processing was not only a desirable capability for the microsensor platform, but also was more computationally intensive than the initial acoustic processing algorithms implemented on the hardware. Due to this increased complexity, successful processing

of imagery on the sensor node would show off the high-level processing ability of the platform, which in turn would make it a more attractive product. Thus, an additional board was designed to permit the ability to handle video processing. Details pertaining to this board, the CAμS video board, are presented in the next chapter.

3.3 Summary

This chapter introduced the CAμS microsensor platform developed by BAE Systems on which the implementation of the UMD image chipping algorithm was placed. To familiarize the reader with the original hardware that existed prior to this thesis effort, a brief top-level description of each of the initial boards was presented. Finally, the chapter concludes with a discussion of one of the limitations of the original hardware that served as the motivation for the design and construction of a new board to be added to the existing system. This additional piece of hardware, the CAμS video board, expanded the capabilities of the CAμS stack to include image processing.

Chapter 4

CAμS Video Board

In order to answer the call for video processing on the CAμS stack, the CAμS video board was developed to enable a means by which to inject both analog and digital video sources into the micro-sensor platform. This chapter first introduces the requirements placed on the video board, which in turn steered the course taken by its design. Then, a discussion of the hardware architecture and its associated components employed to meet these required specifications is presented. Because interfacing to the existing CAμS hardware was another important factor that drove the direction of the design process, a short section has been incorporated in this chapter that discusses how this interfacing was accomplished. Finally, to complete the discussion of the CAμS video board, a brief description of the I^2C bus and its usage to program the chosen video chips has been included.

4.1 Requirements

It was fortunate that the requirements placed on the design of the video board were few, because this limited number of specifications permitted a great deal of design flexibility. The essential capabilities and design considerations that were demanded of the CAμS video board were that it accept inputs from both NTSC and digital video sources, it was constructed according to a relaxed version of the PC-104 form-factor[19] in order to interface with the existing hardware and that an attempt for a low-power solution was made. While the first two requirements demanded of the

video board were unambiguous and definable, the third was left open to a matter of interpretation. For this very reason, the design of the video board was prioritized such that the interfacing to video streams and the existing hardware was met first and then was modified to achieve a lower power solution, if possible. The following sections put forth a brief discussions of the required video streams to be accepted.

4.1.1 Interfacing to the NTSC Analog Video Format

The National Television System Committee (NTSC) is the group that establishes the broadcast television standards in North America. This committee standardized the NTSC television broadcast system in 1953[20], which was an augmentation of the existing monochrome composite video signal. The term NTSC is commonly used to refer to an analog television video format that can be recorded on numerous tape formats including VHS, 3/4" U-matic and Betacam[20]. While the standard for NTSC specifies that is has a fixed vertical resolution of 525 horizontal lines, the number of columns for given row can vary with the electronics and formats employed. This video format is interlaced, which means that one full frame is composed of two fields. There exist two types of fields, odd and even. The odd and even fields contain the odd and even rows of the full-frame, respectively. Fields are displayed with a frequency of 59.94 fields/sec and therefore the frame rate of the standard is approximately 30 frames/sec[20].

Because NTSC is still used today in the United States, Canada, Mexico, Japan and some ten other countries[21], it was a desirable format to which to be capable to interface. Providing the ability to interface with this video format, permitted the video board to interface to both video cassette recorders (VCR) and camcorders, as well as to display its outputs on a standard television.

4.1.2 Interfacing to Digital Video Formats

It is noted that infrared imaging, especially in the 8–12 micron atmospheric window, "is useful for security and surveillance, navigation and targeting, industrial process control, medical imaging, the astronomy of cold objects, the study of atmospheric chemistry (e.g. monitoring and measuring the compositions and levels of pollutants and gases in the air), etc."[22] Quantum Well

Infrared Photodetector (QWIP) Focal Plane Arrays (FPA) have become the front-runners of sensitive, cost-effective LWIR (Long Wave Infrared) arrays[22]. With the belief that coincident and spatially-registered imagery from multiple infrared spectral bands will assist with "clutter rejection, camouflage detection, false alarm reduction, decoy discrimination, countermeasures resistance, target ID enhancement and absolute graybody target temperature measurement"[22], a 2-color QWIP[1] is being developed by Army Research Laboratory in conjunction with the partners of the Advanced Sensors Consortium, headed by Sanders, a BAE Systems company.

Due to its numerous above-listed uses, infrared imagery was considered to be an extremely desirable source with which to interface. Unfortunately, the output of this 2-color QWIP is in a proprietary digital format. It was this desired future interfacing with the 2-color QWIP camera that drove the design of the CAμS video board to include the ability to accept digital video channels in addition to the already required analog sources, and to provide the capability two process two channels of video simultaneously.

Another infrared camera whose interfacing was desired, was the Alpha camera by Indigo Systems shown in Figure 4.1. In fact, as will be discussed in later chapters, this camera, a long-wavelength (7.0 – 14.0 microns) uncooled microbolometer[23], provided the input video source for the image chipping algorithm implemented as the central work of this thesis effort. This camera produces video streams in both the standard NTSC format, as well as in a proprietary digital video format. With the capabilities incorporated in the CAμS video board, it could accept the video stream in either of the formats provided.

[1]The two colors in this camera's name refer to the two infrared frequencies from which this it provides grey-scale imagery.

Figure 4.1: Alpha Uncooled Microbolometer Camera by Indigo Systems

4.2 Hardware Overview

As was mentioned earlier in this chapter, the requirements placed on the CAμS video board were minimal. The flexibility afforded by this characteristic not only simplified the design process of the video board, but also permitted the brainstorming for additional capabilities to be employed by future unforeseen projects and the inclusion of the results thereof in the final design. Figure 4.2 includes images of the top and bottom views of the resulting CAμS video board. In this figure the major components contained on the video board are identified. Section 4.2.1, presents a top-level architectural discussion of the video board that introduces these high-level components employed in its design. Then, the remaining sections briefly outline the capabilities of each of these high-level components present on the board and the reasons for their inclusion.

4.2.1 Top-Level Architectural View

Figures 4.3 shows a top-level block diagram of the video board that depicts its major components and their associated interconnect. As can be seen in this figure, there exists a great deal of symmetry

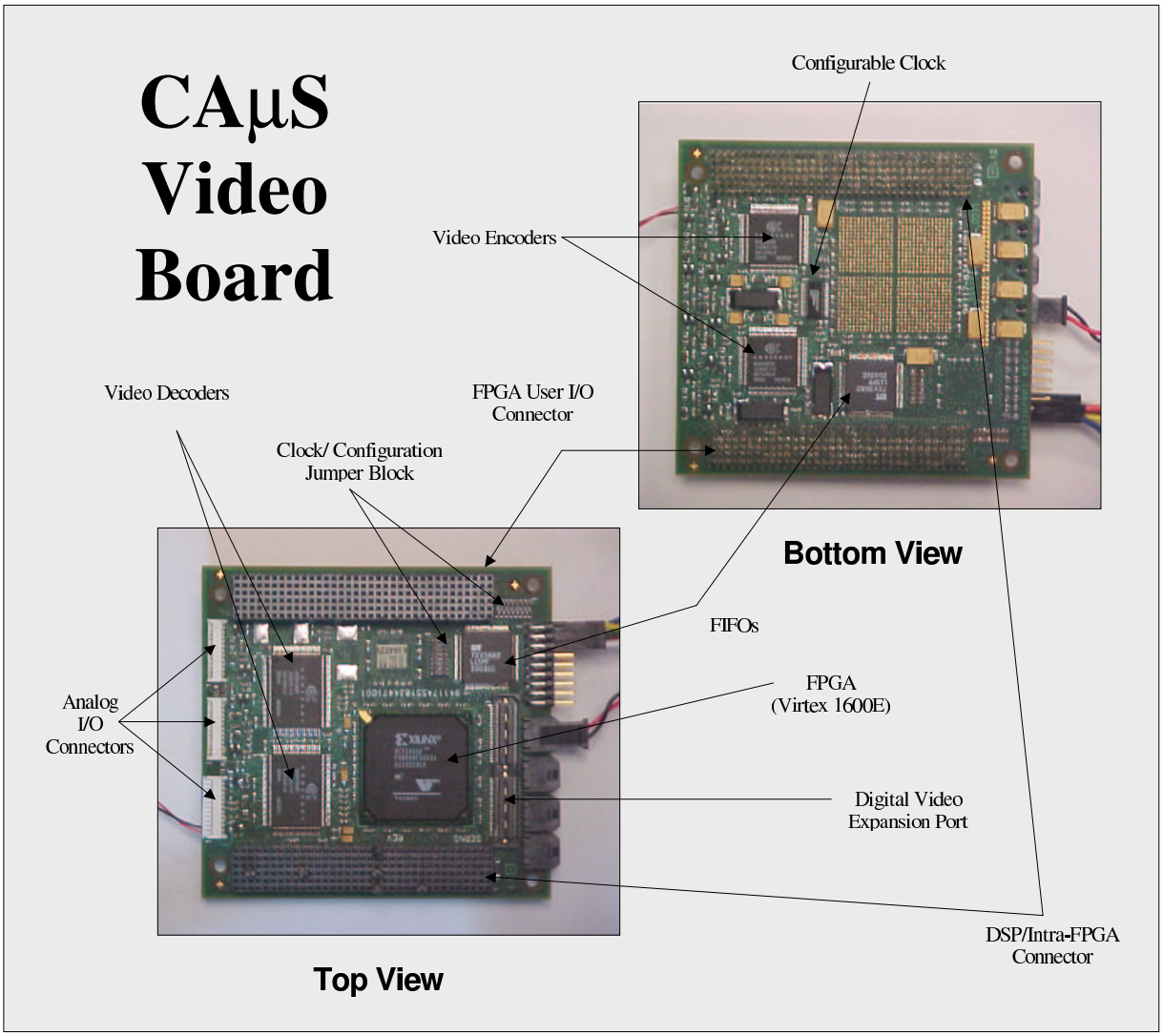


Figure 4.2: Picture of CAμS Video Board

in the final design of the video board. This symmetry resulted from the necessity to process two channels of video simultaneously. That is, after it was determined what hardware was deemed necessary for the processing of a single channel of video, it was duplicated. This duplication process produced the above-mentioned regularity.

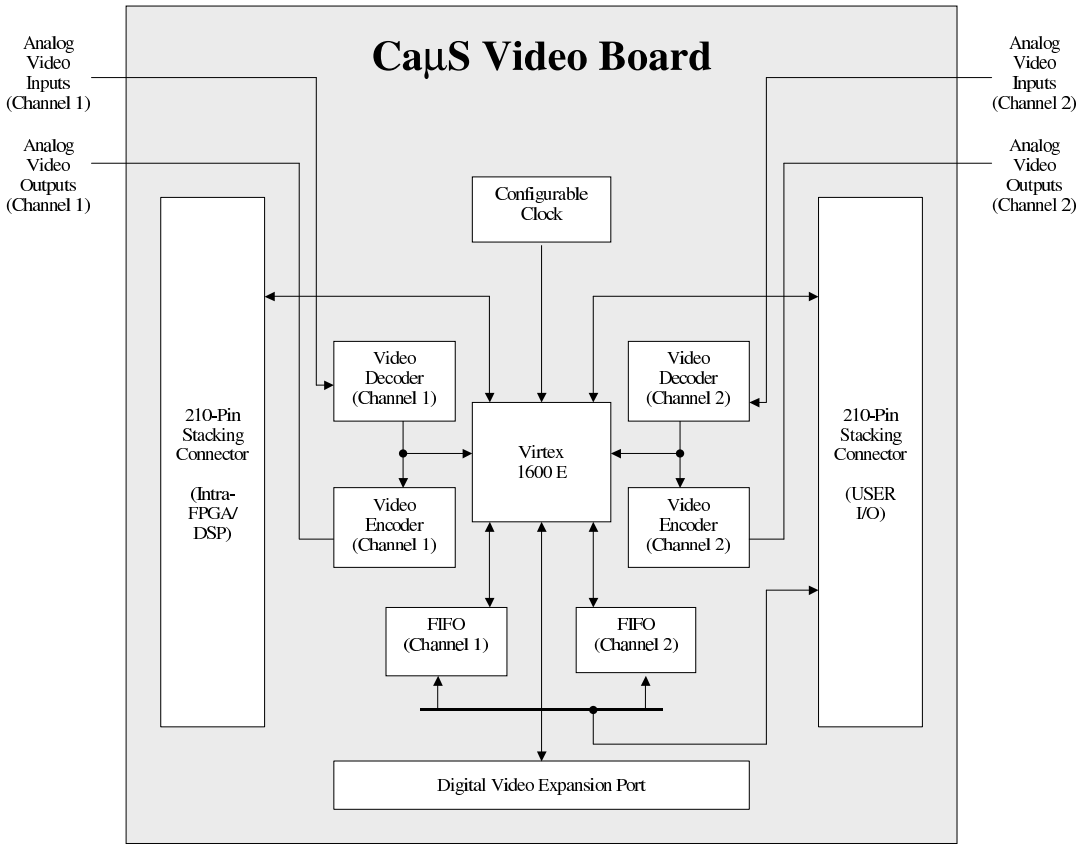


Figure 4.3: Block Diagram of CAμS Video Board

The components dedicated solely to the processing of a specific stream of video are indicated in Figure 4.3 by explicitly labelling them according to the channel to which they belong. These dedicated components correspond to the video decoder, video encoder and bidirectional FIFO memory chips. The remaining devices present on the CAμS video board, which include the configurable clock and field programmable gate array (FPGA), assist in the processing of both channels of video. The following sections present the particulars of these chips and their reason for inclusion.

4.2.2 Video Decoders

The video decoder chips were included on the video board to satisfy the required capability to interface to an NTSC video stream. Yet, with the chosen devices, the Bt835 VideoStream III Decoder by Conexant Systems, formerly Rockwell Semiconductor Systems, interfacing to composite video, S-Video, SECAM and multiple versions of NTSC and PAL sources is possible. These inputted analog video sources are converted by the decoder to either the CCIR601 (ITU-R 601) (4:2:2) or CCIR656 (ITU-R 656, Bytestream) digital video format.

This device was not only chosen because it fulfilled the role of the required interfacing, but also because of its numerous other capabilities, which made it such an attractive option. Its hardware closed-caption decoder, arbitrary temporal decimation, programable hue, brightness, saturation and contrast, auto NTSC/PAL format detect, software selectable four-input analog MUX, two-wire Inter-Integrated Circuit (I^2C) bus interface and image scaling and cropping features are just a few of the many qualities that lead to the final selection of this device[24]. The reader is referred to [24] for a comprehensive listing and discussion of the features of the selected video decoder.

As an aside, it is noted that the above-mentioned image scaling capability was employed in the implementation of the image chipping algorithm to easily scale the analog video stream to the required image size (160 columns $\times$ 119 rows). This usage goes to show that the additional features that were incorporated in the design of the video board proved to be beneficial to a future unforeseen application as was originally hoped.

The final basis for the selection of this particular device over other just as capable chips produced by different manufactures, was its claim to possess a low operating power and its ability to be powered down. As was mentioned earlier, the low-power characteristic was of the least importance of the specified requirements. Thus, the selection of this part was only made because it permitted a lower power solution without the expense of sacrificing functionality.

4.2.3 Video Encoders

Although not a specific requirement placed upon the video board, the video encoder chips were incorporated into the video boards final design to permit the displaying of digital video on a

device which accepted one a number of analog video standards. Similar to the selected video decoder chip, the video encoder supports numerous analog video standards, which include NTSC-M, PAL (B,D,G,H,I), PAL-M, PAL-N, NTSC-443, PAL-Nc, PAL-60 and SECAM. This chip can simultaneously output composite baseband video (CVBS), such as unmodulated NTSC, with YUV and S-Video or CVBS with RGB and S-Video.

Once again like the decoder chip, it was the numerous other capabilities of the Bt860 Multiport YCrCb to NTSC/PAL/SECAM Digital Video Encoder chip by Conexant Systems, that lead to its selection. The programmable adjustment of brightness, contrast, color saturation and hue, three 8-bit YCrCb 4:2:2 inputs for overlay or blending, Closed Captioned, Teletext and Extended Data Services encoding, internal color bar and blue field generation are only a handful of the features of this chip[25]. In order to learn the remaining features available with the selected video encoder, the reader is directed to [25].

However, the most important characteristic of the chosen video encoder and which ultimately lead to its selection, was its glueless interfacing with the previously selected video decoder chip. Yet, as an added benefit, the selected encoder had several low power options, including sleep mode where only the serial programming interface and PLL are operational, and the ability to disable individual DACs and the PLL[25] that assisted in making the design a low power solution.

4.2.4 Field Programmable Gate Array

The field programmable gate array (FPGA) integrated into the video board was intended to be used for pre-processing of a digital video stream received from either an external digital camera or a video decoder chip. However, with its claimed gate equivalency of 2,188,742 system gates[26], the chosen Virtex XCV1600E FPGA by Xilinx is capable of performing more complicated tasks. Similar to the incorporated FPGA of the original hardware, the included FPGA was seen as means by which to provide flexibility in the design. It was hoped that this additional flexibility would reduce the limitations placed upon the end-user, which often occurs with a rigid design. This flexibility inserted into the design was also cost effective because it permitted the ability to potentially correct errors made in the board design that were not discovered until after fabrication. Due to its large number of input/out blocks (IOBs), 724 user I/O pins[27], and the above-described desired flexibility, most

of the signals on the video board were routed to the FPGA as can be seen in Figure 4.3.

4.2.5 FIFO Memories

The IDT72V3682 bi-directional synchronous FIFO memories by IDT were incorporated into the video board design to allow the co-existence of separate clock domains on the CAμS video and FPGA boards. While still permitting communication and passing of data, this separation allowed the two FPGAs in the system to operate off different clocks. After all, it was envisioned that the FPGA on the video board would typically operate off the clocks presented to it by its video chips or from a digital camera, while the FPGA of the existing FPGA board would operate off the configurable clock of the DSP board or the DSP's clock itself (a divided down in-phase version of the configurable clock of the DSP board) so that it could interface with the DSP. Due to the overwhelming number of clocks present in the design, the video board incorporated a clock jumper block, which is shown in Figure 4.2, to switch the operational clock to whichever is deemed the most appropriate for the current application.

The chosen FIFO memories have a storage capacity of $16,388 \times 36 \times 2$ bits and a data bus width of 36 bits[28]. To support bi-direction data transfer between the video board and the FPGA board, bi-directional FIFOs were chosen. Although this task could have been accomplished by two separate uni-directional FIFOs, bi-directional FIFOs, which essentially are two uni-directional FIFOs in a single package, were selected because the data bus sharing that they employ. That is, the data buses of the two internal FIFOs of a bi-directional FIFO are multiplexed between the external pins of the chip. Therefore, as a direct result of this multiplexing, the throughput of the bi-directional FIFO is exactly half that of two separate uni-directional FIFOs. This reduction in throughput was considered acceptable because its associated decreased pin usage left more user I/O pins of the FPGA of the existing hardware open to be used by external devices. Section 4.3, which discusses the interfacing of the video board to the existing hardware, explains the benefits of this pin reduction further.

4.2.6 Configurable Clock

An OSCaR^{TM} user configurable clock by Integrated Circuit System (ICS) was added to the video board to allow flexibility in the clocking. With a maximum output frequency of 250 MHz, the configurable clock increases the processing speed by more than order of magnitude over the pixel clock produced by the decoder chip at 14.318MHz. Therefore, with internal buffering of data by FIFO memories, the FPGA could consume more than ten clock cycles per pixel and still maintain frame rate processing. In fact, with the proper buffering, this number of clock cycles could even be further increased due to the fact that the pixel stream is not constant because of the horizontal and vertical blanking regions in the analog stream from which it is derived. Nevertheless, regardless of the chosen frequency and its usage, the fact remains that the configurable clock provides yet another aspect of flexibility in the video board design that has been left up to future users to decide its optimal usage.

4.2.7 Digital Video Expansion Port

Because there were unused IOBs left on the FPGA, a digital video expansion port was integrated into the design of the video board. This port allowed an alternative means to bring digital video streams into the CAμS stack than using the existing IFU/DSP and USER I/O buses, which would have been the only channels without its inclusion. An 80-pin high density connector with a central ground bus from Samtec was used for this port to save precious board real estate. Referring back to Figure 4.2, it can be seen that the chosen high-density connector is significantly smaller then the 210-pin stacking connectors (which only provide approximately 100 signals because of the interlaced ground pins) used for the IFU/DSP and User I/O buses.

4.3 Interface to Existing Hardware

Considering that the video board was designed to be used as additional hardware to the existing CAμS stack, interfacing and compatibility with this existing hardware highly influenced and limited the design process. The necessity to use the large 210-pin stacking connectors to connect to the

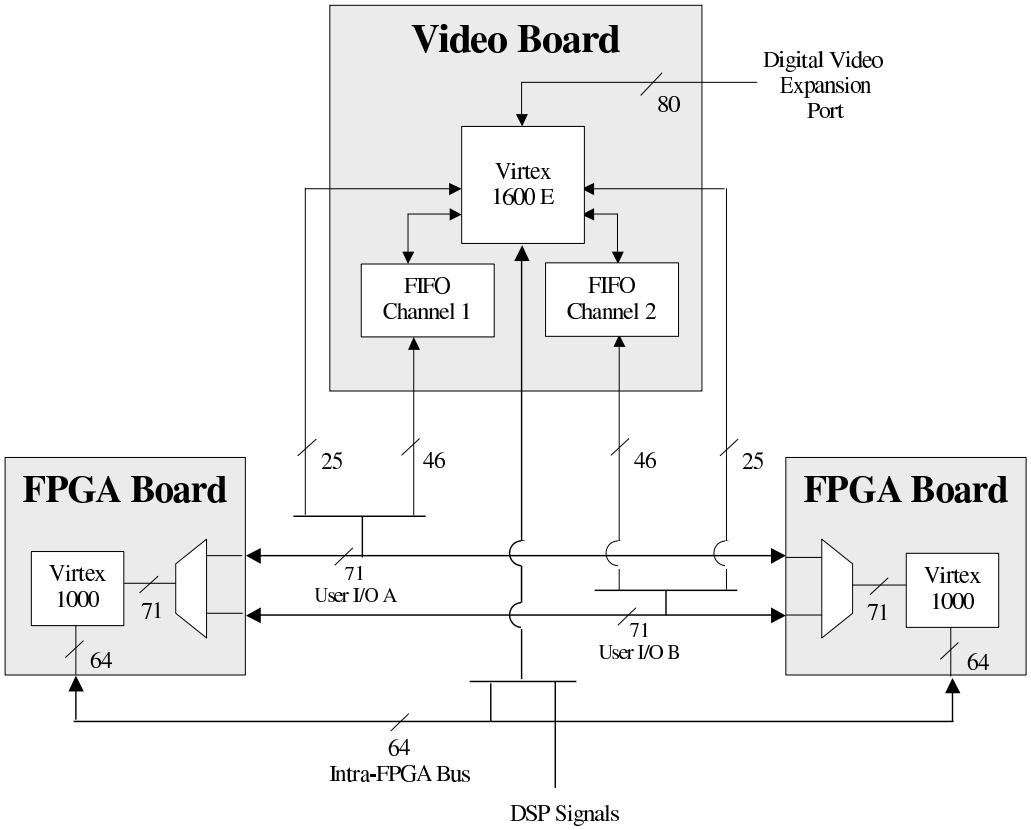


Figure 4.4: Block Diagram of Board to Board Interconnect

existing hardware was one limitation that directly resulted from this required interfacing. These connectors consumed a great deal of the minimal board real estate that was caused by the PC-104 form-factor stipulation.

Figure 4.4 depicts the means by which the designed video board connected to the existing hardware. As can be seen in this figure, both the IFU/DSP and User I/O buses were run between video board and the existing hardware. Considering that the User I/O bus resources were limited (being only 71 bits wide), minimal dedicated usage of these signals was paramount. After all, the fewer pins required for dedicated tasks resulted in more for general purpose usage and hence more flexibility in the design. Therefore, only the associated pins of the FIFO memories used for buffering between the boards were directly connected to the User I/O connector as dedicated signals.

As previously mentioned, the digital video expansion port was included to eliminate the necessary

use of the User I/O bus for injecting digital video to the system, which also left more User I/O bus signals open for external connections. Connection to the IFU/DSP bus was made to gain access to the signals that it carried. This connection allowed the video board's FPGA to act as yet another processing element that communicated through the intra-FPGA bus, as well as, permitted direct interfacing to the DSP. Interfacing directly to the DSP was essential because the DSP was assigned the duty of programming the internal registers of the video encoder and decoder chips. Using the Inter-Integrated Circuit (I^2C) bus implemented with its GPIO, the DSP accomplished this assigned task. A detailed discussion of how this programming was achieved and the resulting API construct to simplify the process is presented in the next section.

4.4 Software Support

Referring to [25] and [24], one will find that the internal registers of the selected video encoder and decoder chips are programmed through the Inter-Integrated Circuit (I^2C) bus. The task of programming these devices was initially assigned to the DSP processor because it was considered the simplest device to provide this functionality due to its ability to be programmed with the high-level programming language, C. Seeing as the I^2C-bus is bi-directional, the GPIO pins of the DSP were used to implement the protocol because they provided the required input/output directionality and could be easily manipulated.

It is noted that even though the DSP was initially chosen to be the controller of the I^2C-bus, all the associated bus signals were routed though the FPGA of the video board. With these additional routes, the FPGA is capable of being the bus controller and an I^2C slave device. Therefore, if so desired, with some additional effort and hardware design one could relinquish the DSP of this responsibility of programming the video chips and place it upon the FPGA if it was considered beneficial to a future design. Once again, this above-described choice of implementation highlights the concept of flexibility afforded to future efforts that was a driving force behind the video board's design. The following sections introduce the I^2C-bus and the former approach that was taken to implement its control.

4.4.1 I^{2}C Bus

The Inter-Integrated Circuit (I^{2}C) bus is a simple bi-directional 2-wire bus developed by Phillips for efficient inter-IC control[29]. The two signals that compose the bus are a serial data line (SDA) and serial data clock line (SCL)[29]. Each component connected to the bus possesses a unique address and is software addressable. Bus transfers occur according to a protocol discussed in [29], at a rate of 0 to 3.4 Mbit/s in which the upper bound is limited by what mode (Fast-mode or Standard-Mode) devices are connected to the bus. By being completely integrated into the to the devices, the I^{2}C-bus protocol eliminates the need for an external address decoding and any other glue logic[29]. For a more detail discussion of the I^{2}C-bus, the reader is directed to [29].

4.4.2 Application Programming Interface (API)

Although the I^{2}C-bus protocol was considered to be fairly trivial, it was still believed to reside at too low of a level for the end-user. Thus, a software API was developed to provide a layer of abstraction above the raw bus signaling. By being built in a hierarchical fashion and making each layer of the hierarchy accessible to the end-user, the API in itself provided multiple layers of abstraction. That is, low-level functions were written to handle the control of the GPIO signals such that they implemented the fundamental signaling required for bus transfers by the I^{2}C-bus protocol (i.e. generating START and STOP conditions, performing byte read/writes with proper checking for acknowledgement). Then, higher level functions were constructed that called these fundamental functions as subroutines and hid their corresponding details.

At the top-most level, a simple interface was presented to the end-user. With all the low level details hidden, the end-user needs only to specify the desired values and addresses of the registered to be programmed for each channel in a convenient tabular fashion to the top-level function that presents this interface. This top-level function, through its employment of lower-level subroutines, guarantees reliable programming of the specified registers. Thus, all the complexity of the protocol has been concealed from the end-user and the only task that remains is the determination of the correct register values to be programmed such that the video decoder and encoder chips will be configured to operate as desired.

4.5 Summary

This chapter has presented a brief discussion of the CAμS video board. First, the requirements placed on the design and the decisions made to meet them were explained. Then, a top-level discussion of the architecture was presented as a means by which to introduce each of the components employed in the final design. The capabilities of each of these components and the reasons for their inclusion were then covered. Finally, the video board's interfacing with existing hardware and the developed API for the simplification of its use were placed forth. As a final note, it is mentioned that a common thread of design flexibility runs throughout the discussions of the video board presented in this chapter. This underlying theme of design flexibility was included to emphasize the effort placed forth to provide this desirable attribute and to bring to the forefront its successful incorporation in the final design. This successful inclusion of flexibility attests to the fact that the resulting video board is inline with the fundamental spirit that was the motivation for the development of the original CAμS hardware.

Chapter 5

Image Chipping Algorithm

Considering that communication bandwidth is often limited by power constraints on microsensor platforms[6], algorithms that keep data transmissions to a minimum are extremely desirable. After all, the less power consumed by data transmissions directly translates to an increase in the system's lifespan. One such algorithm that attempts to reduce the transmission overhead was developed at the University of Maryland. This algorithm, referred to in this thesis as the UMD image chipping algorithm, was implemented on the CAμS stack as the main portion of this thesis effort. This chapter introduces the top-level functionality that this algorithm provides and then discusses the specific computations preformed to achieve this functionality. Finally, a discussion of the partitioning of its operations across the hardware components of the CAμS stack and the reasons for this chosen division of labor is presented.

5.1 Overall Functionality

With remote surveillance of battlefields being an important component of Future Combat Systems (FCS)[30], motion detection algorithms that are capable of operating on microsensor platforms are desirable. The UMD image chipping algorithm was developed specifically with its microsensor implementation in mind. In order to meet the low power constraints of these platforms and still operate in real-time, this algorithm had to be computationally simple[30]. Even though this algo-

rithm detects the moving target in every frame, it cannot transmit all frames because of the low channel bandwidth available (300bps)[30].

To compensate for this transmission limitation, the best view selection portion of the algorithm was incorporated. This section of the algorithm determines which of the subsections of the original images, referred to as chips, is considered the *best*. With this addition, the final step in the reduction of transmitted data from the original full frame rate, to subsections of images at frame rate, to a single subsection of many images, is possible. Not only does this reduction in transmitted data conserve power, but it also lessens the load on the ground station and its operator. After all, the reader is reminded that the vision of FCS is to have many of these sensors operating at the same time. If every sensor were to transmit back either full frame rate video or even subsections of every frame, the ground station would be overwhelmed. Thus, this algorithm determines the *best* chip and transmits only this subsection of its associated image. In fact, even though it was not implemented, it is noted that as a final step to reduce the transmission overhead, the UMD algorithm compresses the chip before transmission.

In order to clarify the above-described algorithm and its functionality, in particular the vague term *best* when used in regards to chips, the following section presents a detailed discussion of the specific computations performed by the UMD algorithm. This section first explains the operations required to track objects in each frame and then covers the selection criterium of the *best* chip from a sequence of frames. A brief explanation for the inclusion of each operation is also given.

5.2 Specific Computations

The UMD image chipping algorithm was developed in Matlab and employed sampled IR sequences from the Alpha camera by Indigo Systems for testing and validation. These sequences contained images of size 320×240 pixels that were captured at 30 frames/sec. It was the responsibility of this thesis effort to implement the resulting algorithm on the CAμS 6.1 microsensor platform. As the first step taken in this implementation process, the original Matlab source was examined and a flow diagram of its operations was generated. Figure 5.1 depicts a polished version of the original flow diagram extracted from this code analysis. This figure will be referred to throughout the

discussion of the image chipping algorithm to pictorially represent the operations presented in the text. Figure 5.2 has also been included to assist the reader in visualizing each of operations shown in Figure 5.1 as they are explained. Referring to Figure 5.1, it can be seen that the image chipping algorithm is cyclic with each repetition commencing upon the capture of a new frame of video. Owing to this characteristic, the following walk-through of the operations performed in a single iteration will in actuality provide a complete coverage of the algorithm's operation.

Beginning at the START state shown in Figure 5.1, it can be seen that the first block of the UMD image chipping algorithm is a circular buffer of length two images. This small buffer size is ideal for its hardware implementation on a microsensor platform, because oftentimes memory on these devices is limited. In fact, as will be discussed later in this section, the algorithm only requires the storage of one additional frame, the saved image, for proper operation. Thus, only three full images must be stored at any given time, the current image, the previous image and the saved image.

In the Matlab implementation, every sixth frame of a 30 frame per second video stream was used. Therefore, the frame rate of the algorithm was determined to be (30/6 frames/sec) 5 frames/sec, which established the target frequency for the hardware implementation. The first two images in Figure 5.2 labeled "Original Image 0" and "Original Image 1", correspond to two images from the IR footage used to test the algorithm in Matlab that were sampled at 5 frames/second.

Following the collection of two images, their absolute difference is computed. The output of this computation is yet another image that has non-zero pixel values where changes have occurred from one image to the next. Considering that the algorithm requires that the camera be stationary while operating, non-zero pixels in the difference image typically correspond to moving objects in the cameras field of view. However, these non-zero pixels could also be caused by erroneous pixel values or sudden changes in light intensity from one image to the next. Because the intent of the absolute difference computation is to determine moving targets[31], later blocks in the algorithm attempt to compensate for the non-zero values caused by images variations that are not due to movement. The third image in Figure 5.2 is the absolute difference of the first two shown in this figure. As can be seen in this image, the pixels with the brightest intensity correlate to movement in the images. That is, the people walking across the field of view of the camera appear the brightest in the absolute difference image.

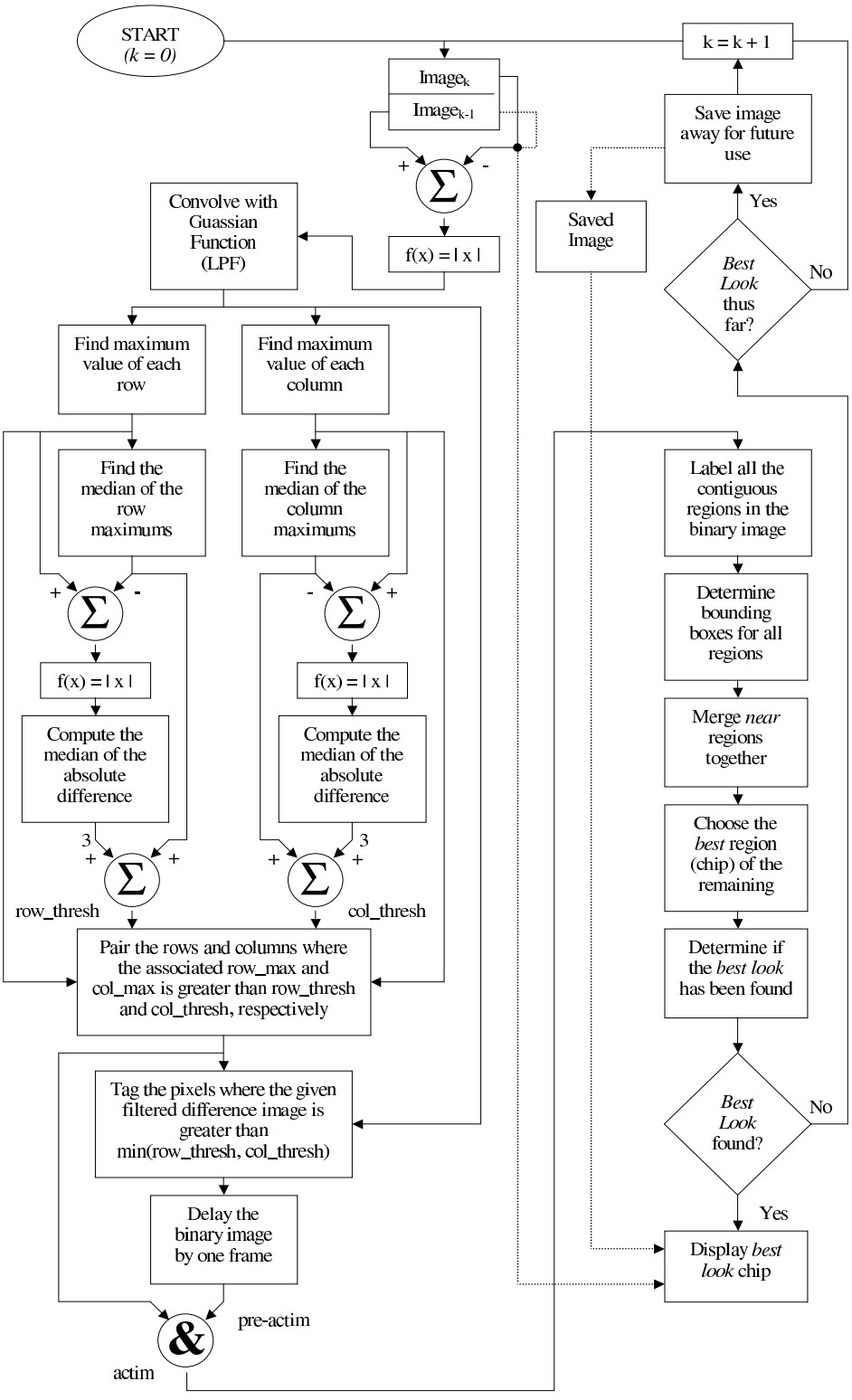


Figure 5.1: Flow Diagram of UMD Image Chipping Algorithm

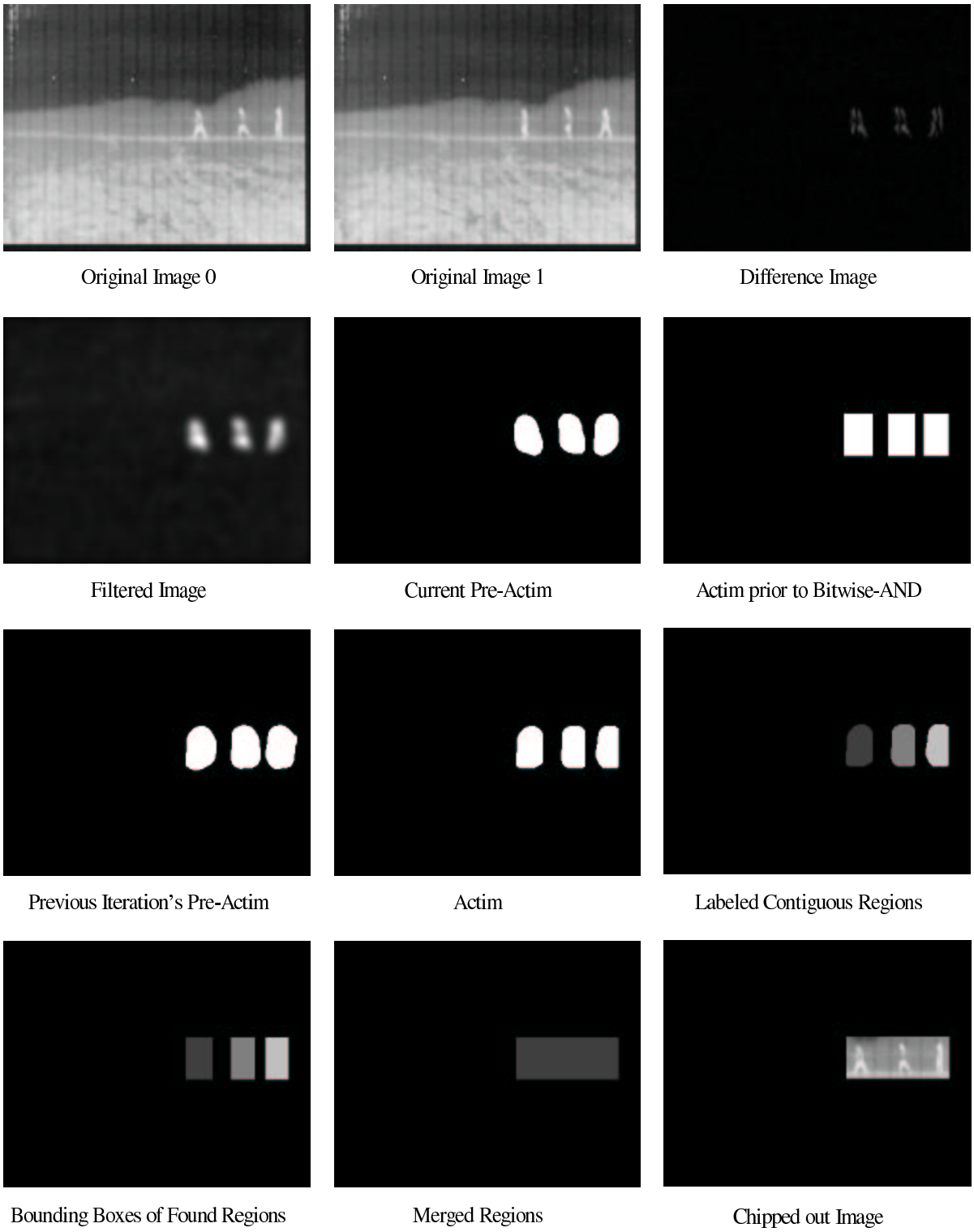


Figure 5.2: Pictorial Representation of Computations Performed by UMD Image Chipping Algorithm

Traversing to the next block in the flow diagram, one will see that the next operation performed by the algorithm is a 2D convolution of the absolute difference image with a Gaussian filter. The chosen filter is a low pass filter with a kernel size of 37×37. This filter minimizes abrupt discontinuities in the absolute difference image. These discontinuities can be caused by either erroneous pixels or by small objects moving in the image such as leaves blowing in the wind[31]. Regardless of the cause of these abrupt discontinuities, they are undesirable objects to track. Therefore, it is hoped that the *smoothing* of the filter will cause them to reside below the dynamically computed thresholds and thus not be included in the regions of interest inspected by the latter portion of this algorithm. The image entitled "Filtered Image" corresponds to the image produced by applying the 2D low-pass filter to the difference image shown in this figure and normalizing the resulting image over the range 0 to 255.

From the filtered absolute difference image, the algorithm next determines the dynamic thresholds, *row_thresh* and *col_thresh*, that are used to produce the regions of interest (ROI). In an attempt to avoid the pitfalls associated with static thresholds, these thresholds are computed dynamically. After all, with static thresholds undesirable ROIs will result if the selected values are either too high or too low. If the thresholds are too high, desirable ROIs for further analysis will not be identified. On the other hand, thresholds that are too low will produce false detects for ROIs. For example, a threshold that is too low will cause an entire frame to be incorrectly identified as a ROI when an abrupt brightness change occurs from one image to the next.

Moreover, it should be noted that static thresholds are only appropriate for a limited data set. Nonetheless, static thresholds can be determined off-line prior to operation from sample sets of data. This off-line selection saves power by eliminating the necessity to determine the thresholds at run-time. However, to permit the microsensors to operate in a wide range of locations and at all times of day, the small power expended to their dynamic computation is considered worthwhile. By computing the thresholds dynamically, one can tailor their values to the given image such that only the desired sections of the filtered image are tagged as ROIs[31].

The computation of the dynamical thresholds, *row_thresh* and *col_thresh*, commences with the determination of the maximum pixel values contained in each column and each row of the filtered image. This operation produces two vectors, which will be referred to as *row_maxs* and *col_maxs*

for the maximums of the rows and columns, respectively. The median of each of these vectors is then determined. Let *median_max_rows* and *median_max_cols* be defined as the median of the row and column maximums, respectively. The remaining computations required for the determination of the dynamic thresholds is best explained by Equations 5.1 and 5.2.

$$row\_thresh = median\_max\_rows + 3 * median(|row\_maxs - median\_max\_rows|) \tag{5.1}$$

$$col\_thresh = median\_max\_cols + 3 * median(|col\_maxs - median\_max\_cols|) \tag{5.2}$$

The above-described computations for the determination of the dynamic thresholds are shown pictorially in Figure 5.1 by the sub-blocks that reside between the convolution block and the arrows labeled with their respective resulting thresholds.

Figure 5.3 contains two plots that show the dynamically computed thresholds for the filtered image shown in Figure 5.2 and their respective maximums on the same axes. By comparing these plots to the filtered image from which they were generated, it can be seen that the computed thresholds values are such that only the rows and column maximums of the rows and columns which correspond to movement in the image exceed their respective threshold. Considering that the row and column maximums are compared against their corresponding dynamic thresholds to produce the ROIs, the ROIs determined from this image will correctly correspond to regions of motion. The reader is reminded that the production of this desirable result was the exact reason that dynamic threshold computations were employed.

The next four sub-blocks of Figure 5.1 show the exact operations required for the production of the binary image that represents the regions of interest. It can also be seen in this figure that the production of this binary image, *actim*, makes use of an intermediate binary image, *pre-actim*. Following the flow diagram, it can be deduced that the construction of the *pre-actim* and *actim* binary images are quite similar. In fact, prior to the bitwise AND operation, *actim* is essentially *pre-actim* with the condition that the specific pixel value of the filtered image exceed the minimum of the thresholds being relaxed. The images entitled "Associated Pre-Actim Image" and "Associated Actim Image prior to Bitwise AND" in Figure 5.3 (also shown in Figure 5.2 entitled "Current Pre-Actim" and "Actim prior to Bitwise AND") show these binary images with the filtered image from which they were generated.

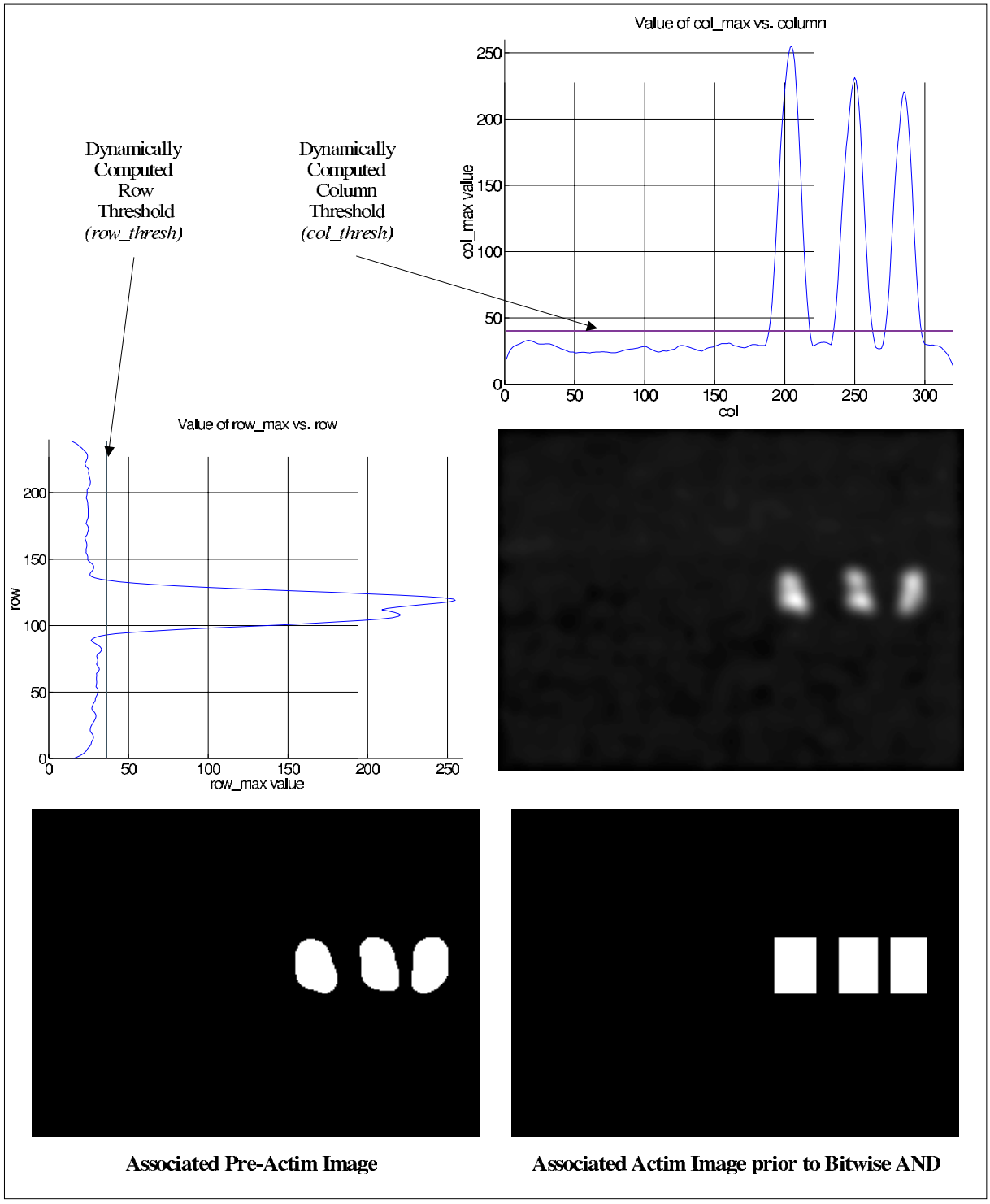


Figure 5.3: Pictorial Representation of Dynamic Thresholds and Row/Column Maximums

Referring back to Figure 5.1, it will be seen that the *pre-actim* image is delayed by one frame before it is bitwise AND'ed to produce *actim*. This delay introduces inter-frame dependency. That is, by creating the binary image of regions of interest based off the previous iteration's *pre-actim*, tagged regions of interest not only correspond to regions that exceed the thresholds in the current filtered image, but also to activity in the previous filtered image. This inter-frame dependency ensures that the indicated regions of interest correspond to only sections of the image that have consistently exceed their computed thresholds. This correspondence prevents the labeling of spurious movement or apparent movement caused by erroneous pixels.

This AND'ing also eliminates additional regions introduced by the *actim* images that do not correspond to activity in the image. These additional regions in the *actim* binary image are produced because only the *row_max* and *col_max* values are used for its generation. Thus, pixels which whose associated row and column maximums exceed their corresponding thresholds will be tagged even if the specific pixel value does not. An example of one of these binary images is the image entitled "Actim" in Figure 5.2, which is the result of bitwise AND'ing the images entitled "Actim prior to Bitwise AND" and "Previous Iteration's Pre-Actim".

It is noted that on the first iteration of the algorithmic loop there exists no *pre-actim* image. To handle this special case, the bitwise AND operation is skipped in the production of *actim* during the first run. Therefore, the images entitled "Actim prior to Bitwise AND" and "Associated Actim Image prior to Bitwise AND" in Figures 5.2 and 5.3, respectively, would correspond to the final *actim* binary image during the first iteration of the algorithm.

Once the binary image has been produced, the UMD image chipping algorithm determines the contiguous regions contained within it. This operation is shown by the first sub-block of the second column in Figure 5.1. These regions, often called connected components, are checked for 8-connectivity. Eight-connectivity means that two pixels are considered connected if they are either adjacent or diagonal neighbors of one another. The image entitled "Labeled Contiguous Regions" in Figure 5.2 uses different grey-scale levels to shows the found connected components in the "Actim" image.

Following the connected components' discovery, their associated bounding boxes are determined. A bounding box is a rectangular region that is just large enough to incorporate the minimum

and maximum rows and columns of a given connected component. That is, the left edge of the bounding box corresponds to the location of the leftmost pixel in associated connected component, the top-edge to the topmost pixel in the connected component and so and so forth. Figure 5.4 depicts the corresponding bounding boxes for some common shapes to clarify this point. The associated bounding boxes for the "Labeled Contiguous Regions" image in Figure 5.2 are shown in the "Bounding Boxes of Found Regions" image in this figure. To give the reader the relationship of this operation to the entire algorithm, he is referred to Figure 5.1 where it is depicted as the second sub-block in the second column.

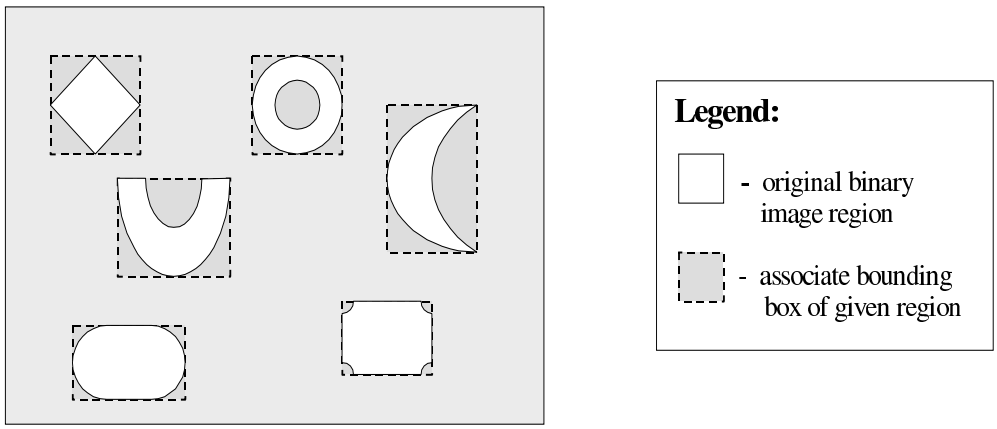


Figure 5.4: Pictorial Representation of Bounding Box Sub-Block

Continuing down this second column of Figure 5.1, the next operation of the UMD image chipping algorithm, merging of *near* regions, is found. Merging of regions involves determining the bounding box that incorporates the two *near* sub-regions and replacing the two separate regions bounding boxes with the single larger one. The region merging is performed for two reasons. First, this merging combines the front and tail end of a hot object that can often be separated if the center is of a similar temperature. After all, if the temperature of an entire object is nearly the same, it will show up with the same intensity on an IR camera.

As this object moves through a scene, only the front and tail ends of this moving object will cause large values in the difference image as it moves across the cool background. The difference between the front and middle and middle and rear sections will produce small values because of their similar temperature. Thus, there exists the potential that the center portion of the moving object will not

exceed the dynamic thresholds and in turn the single moving object will appear as two separate objects. Considering that two separate objects does not reflect the true motion in the scene, a merging of the multiple objects back to a single is desirable. Also, this merging improves the quality of the chipped image because it constructs a bounding region around the entire moving object and therefore all the pixel values associated with it will be transmitted if it is selected as the best chip.

Second, the merging of regions groups common motion together. For example, take the sample images shown throughout this chapter that contain three people walking across the field of view of the camera. As will be explained later, the image chipping algorithm only operates on a single chipped region during the *best* chip selection portion of the algorithm. Owing to this characteristic, there exist two possibilities. Either a single person is used in the determination of the best chip or the three people are grouped together into a single chip and it is employed. Considering that the latter option provides more situational awareness, it is more desirable and therefore is chosen.

With the discussion presented thus far, the metric *near* is still open to a matter of interpretation. However, in reality a explicit set of criterium have been established that clearly define this term. Two bounding boxes are considered near if a corner point of the second lies within the expanded regions about the first box's corner points. That is, each corner point of one of the bounding boxes is placed in the center of a square with each side of length 2×NEAR_DIST, where NEAR_DIST is defined as a fixed number of pixels. If a corner of any other box resides within one of these squares, it will be merged. This operation is repeated for all the bounding boxes of the connected components found in the binary image.

Figure 5.5 has been included to clarify this merging operation. In this figure, the bounding boxes are referred to as blobs. As can be seen, two regions in this figure meet the required criterium and therefore are merged into the larger block indicated by the dotted box, while the third does not. To continue along with the pictorial example of the UMD algorithm, the reader is directed to the image labeled "Merged Regions" in Figure 5.2, which shows the corresponding merged region for the chips contained in "Bounding Boxes of Found Regions".

As was shown in Figure 5.5, there may exist more than one bounding box following the merge region operation. However, the *best* view selection portion of the algorithm only operates on a

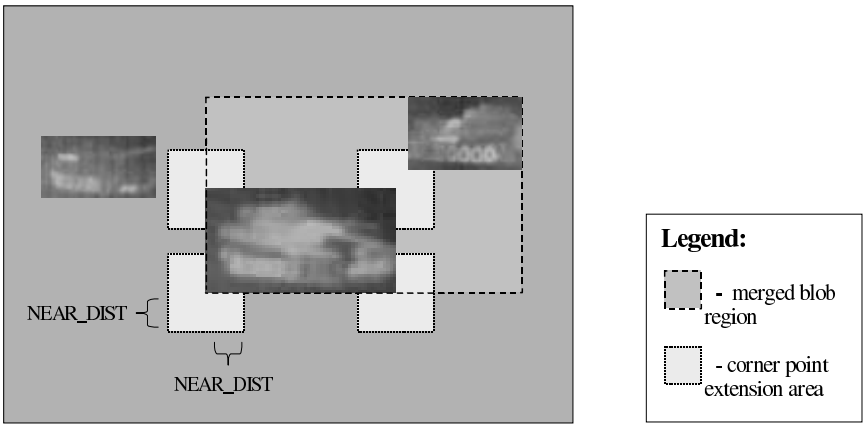


Figure 5.5: Depiction of Blob Merging Constraint

single bounding box. Therefore, if more than one bounding box remains, one of the remaining must be selected. Remembering that the *best* chip selection portion of the algorithm functions on chips between iterations, it can be in one of two states. It is either tracking a chip from the previous iterations or has not yet found a chip worth tracking. In the former case, the bounding box with the closest center point to the previously tracked chip is selected in attempt to continue the tracking of the same moving object. On the other hand, when no previous chip is being tracked, the chip with the largest value of its height plus width is selected. The assumption made here is that the largest chip contains the most interesting and useful moving object in the field of view. This discussed operation is shown in the second column of Figure 5.1.

At this point in the algorithm, the moving target has been detected in the current frame. This concept is shown pictorially by the image entitled "Chipped Out Image" in Figure 5.2, which corresponds to the detected moving target for "Original Image 0." It is now the responsibility of the *best* view portion of the algorithm to determine whether or not this chip is the best view chip. The best view selection is size based[32] and is implemented as a series of complicated if/then/else statements and associated constants. The algorithm "simply waits till the size of the image chips exceed a pre-defined threshold"[32].

Specifically, the best view chip is the largest chip detected, which is above a minimum size threshold while at the same time resides at least predefined number of pixels away from the image

boundary[33]. The best view chip is selected when either size of the largest (tracked) chip begins decreasing or the current chip is the largest detected in the last 15 consecutive frames[33]. A single frame buffer, saved image, is used to hold the maximum sized chip seen thus far. The final criterium for selection of the best chip occurs when same chip from the sequence is chosen for 15 frames[33]. Upon exiting the best view portion of the algorithm, two outcomes can result. It will either determine that the best chip has been found or not. If the best chip has been found, it is extracted from the current image if it corresponds to the current iterations chip, or from the saved image if it is from a former and is displayed.

If the best view was not found, a determination of whether this current chip is the best look thus far is made. If the chip for the current image corresponds to the best look thus far, the current image takes the place of the saved image and the algorithm iterates again. If not, no replacement is conducted and the algorithm repeats immediately. The algorithm continues to iterate until either a best view chip is found, upon which it extracts and displays this chip and exits, or the number of captured frames exceeds a predefined constant, *max_frame*. When *max_frame* is exceeded, the algorithm extracts and displays the best look thus far chip if one exists and then exits. These final operations make up the remaining sub-blocks of the righthand column in Figure 5.1.

With the coverage of these final blocks complete, the entire algorithm has been presented owing to its iterative nature. In conclusion, the overall functionality of the algorithm is once again summarized. The UMD image chipping algorithm detects moving objects in every frame of video, but waits till the *best* chip has been found before any image information is transmitted (or displayed in the original Matlab implementation). There are several operations required to perform this image chipping algorithm. However, these operations can be encapsulated into sub-groups consisting of frame capturing, difference computation, filtering, threshold computation, binary image production, region labeling and combination, best look selection, and chip extraction and transmission. The following section discuss how and why these listed grouped operations were partitioned for the hardware of the CAμS 6.1 stack for this algorithm's implementation.

5.3 Hardware Partitioning of Algorithm

The operations or the UMD image chipping algorithm, described in Section 5.2, were partitioned into hardware and software modules to be mapped onto the CAμS 6.1 platform. Using the Xilinx Virtex^{TM} XCV1000 FPGA present on the FPGA processing board, the hardware modules were implemented with VHDL. Software modules were written in C to be executed on the Motorola 56307 DSP on the DSP board. Keeping in line with the hardware's intended use, the operations were separated such that the processor was mainly responsible for control, communication and housekeeping tasks, while the computationally intensive portions of the algorithm were assigned to the FPGA.

This partitioning was not only based on previous experience, but also on the results of some initial testing. Owing to the short life cycle of the project, it was originally planned to implement the entire algorithm on the DSP by porting the Matlab source to C. Then, the computationally intensive portions would be extracted and implemented in hardware. This approach was desirable because it was known that conversion to software would be a great deal easier than to hardware. Therefore, the simple porting to software would be initially made for all operations and then only operations whose throughput could be significantly improved would have undergone the arduous process of translation to hardware. Unfortunately, by using the software simulator provided by Motorola, it was determined that the operations required up to and including the production of a binary image would only have been able to operate at a rate of approximately 1 frame every 30 seconds (or at 1 frame every 3.33 seconds with optimizations enabled) with a 30 MHz clock speed, which is far from the desired 5 frames/sec. Thus, this approach was deemed to be not a viable solution and was altered.

Owing to their unfavorable performance in software, these initial operations, which consisted mainly of simple, repetitive computations, were assigned to the hardware implementation portion of the algorithm. In hardware, they could exploit the inherent parallelism of FPGAs to provide significant performance improvements over their straightforward software implementations[34]. The remaining operations, which in general were less repetitive and possessed much conditional branching, remained assigned to the software implementation portion of the algorithm[34] where they could

easily be implemented. The ultimate partitioning of the algorithm is shown in Table 5.1

Table 5.1: Partitioning of UMD Image Chipping Operations

Operation	Assigned Implementation
frame capturing	hardware
difference computation	hardware
filtering	hardware
threshold computation	hardware
binary image production	hardware
region labeling and combination	software
best look selection	software
chip extraction and transmission	software

Considering that the tasks assigned to the hardware implementation consisted of mainly simple streaming operations, very little logic had to be dedicated to their computations. It was believed that the majority of the logic required to implement these operations in hardware would be allocated to control. Initial size estimates and resource allocation predicted that with the exception of the 2D filter, the implementation of all operations could easily be contained in a single Virtex XCV1000 FPGA. Because concern about the filters required resources existed, an alternate choice was sought. That is, the final 2D filter implemented in hardware was only a 9×9 2D filter instead of the 37×37 filter used in the Matlab implementation. This reduction in size not only lessened the hardware resources required for it implementation, but was more appropriate for the reduced image size employed by the hardware implementation. For a detailed discussion about this filter modification, the reader is directed to Section 9.2, which further explains the motivation for this algorithmic change and the process by which the selected filter was chosen.

The design path of the implementation was forked such that it permitted the injection of known data and extraction of results for debug capability, while at the same time incorporated the necessary functionality to operate as a fully-functioning microsensor platform. Figure 5.6 shows the division of labor listed in Table 5.1, as well as the splitting of the frame capturing and chip extraction/transmission operations to permit debugging capabilities.

Considering that it was intended that the sampled IR footage used to test the Matlab implementation would also be used to initially verify the hardware implementation, the CAμS video board

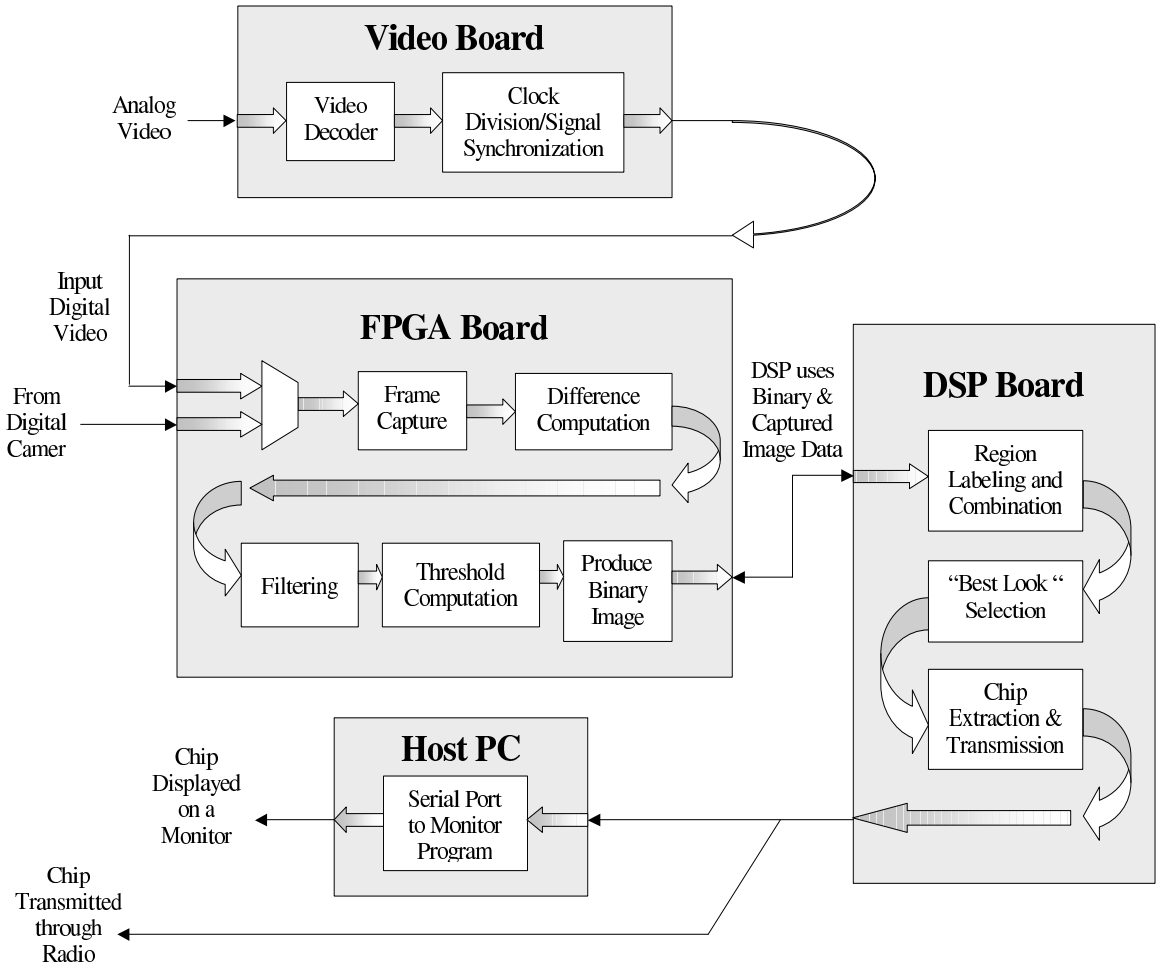


Figure 5.6: Block Diagram of Division of Tasks Among Hardware Components

was incorporated into the design. This video board was assigned the task of converting the analog NTSC video stream from a VCR to a digital stream that could be accepted by the FPGA board and in turn be injected into the algorithm. Yet, it must be remembered that the long term plan of the project was to have live video produced by the Alpha camera as the video source. Therefore, functionality to accept video from either of these sources was included.

The last operation listed in Table 5.1, chip extraction and transmission, was also implement such that it could support debug and full operation modes. Even though a GUI was developed to operate on a personal computer for debugging of the algorithm, the fully functioning algorithm was to transmit its chips over a radio to a ground station. These two chip transmission paths are also

shown in Figure 5.6.

The operations performed on the CAμS FPGA and video boards are referred to as the hardware implementation portion of the algorithm and are presented in detail in Chapter 6. The software implementation portion, which operated on the DSP board, is explained in Chapter 7. Finally, the support software written to be executed on the PC that assisted in the debugging of the algorithm's implementation is covered in Chapter 8. The reader is directed to these chapters for further information.

5.4 Summary

This chapter presented both an overall and detailed discussion of the UMD image chipping algorithm. In the overall discussion, the functionality of the algorithm was given from a top-view perspective, while the detailed discussion covered the specific computations and operations required to provide this functionality. Finally, this chapter closes with a section that explains how and why the UMD algorithm was partitioned for its implementation on the CAμS 6.1 microsensor stack.

Chapter 6

Hardware Implementation

As was explained in Chapter 5, the UMD image chipping algorithm's computations were partitioned between hardware and software based upon the tradeoffs between ease of implementation and performance. This chapter presents a detailed discussion of the computational sub-blocks implemented on Virtex XCV1000 FPGA. Figure 6.1 depicts the sub-blocks that were designed to undertake the computational burden assigned to the hardware. As can be seen in this figure, the hardware was partitioned in to several sub-blocks with a trivial interconnect. This division of hardware not only permitted the development and simulation of these smaller, more manageable, blocks to occur concurrently, but also allowed their testing and verification to take place independently. A discussion of each of these sub-blocks and the related support hardware is presented in the following sections.

6.1 Top-Level Block

Although not explicitly shown in Figure 6.1, a crucial top-level hardware block is also present within the FPGA. This block is responsible for the control of all the other illustrated sub-blocks. That is, this block controls which of the computational sub-blocks is active at a given moment and sets the *ram_interface*, and *bram_interface* control signals appropriately to provide the currently active sub-blocks with their necessary interconnect. Section 6.1.1 further explains the control of the computational sub-blocks, which is accomplished by using a finite state machine (FSM). The top-

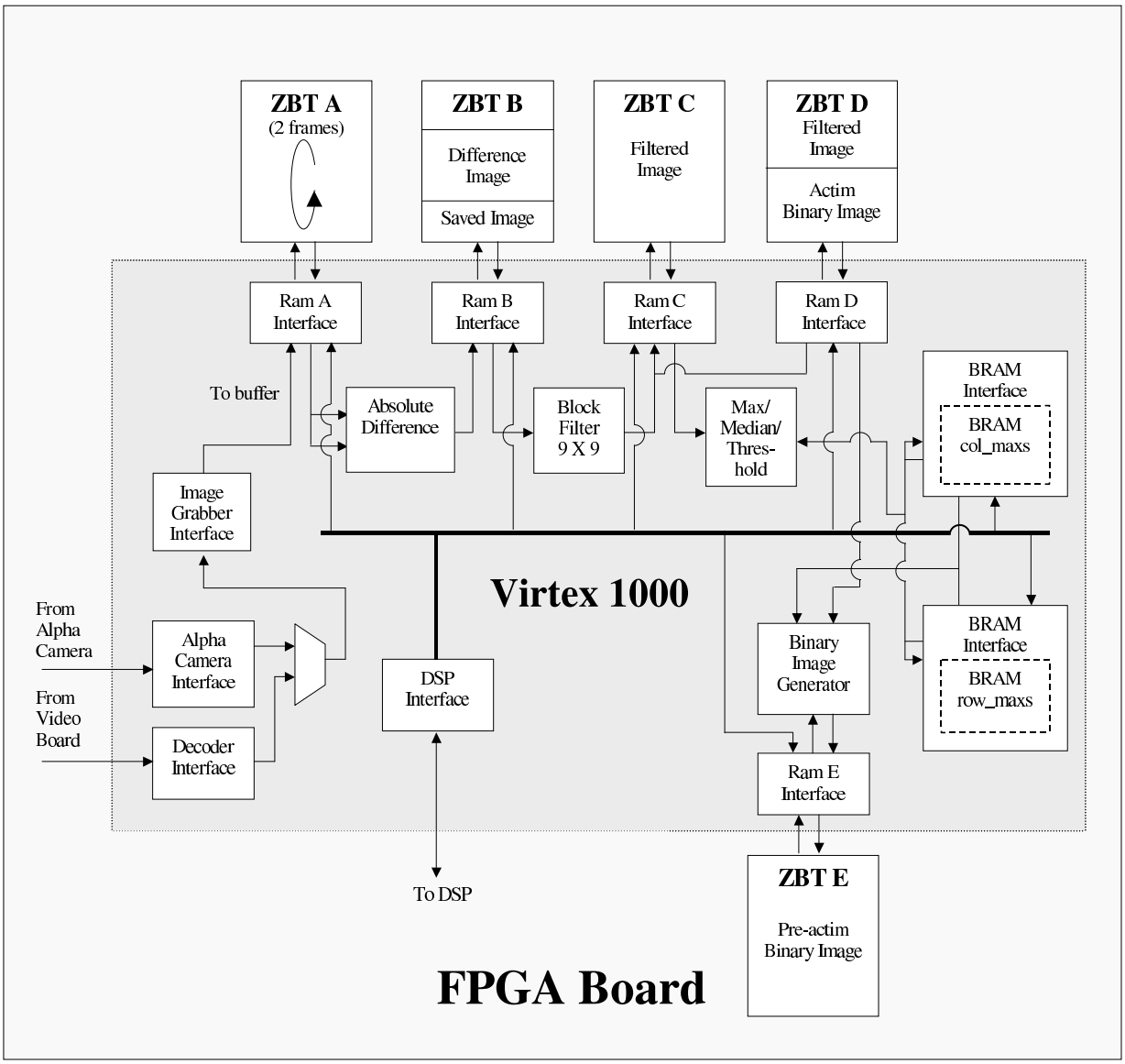


Figure 6.1: Block Diagram of Sub-Blocks Embedded within the Virtex XCV1000 FPGA

level block is also the repository for the debug/control registers, which are explained in Section 6.9.2

6.1.1 State Machine Architecture

To simplify the initial implementation and with the belief that the computational time budget could still be easily met, it was decided to only permit one computational sub-block to be active at any given moment. Although this decision prohibited the greater throughput that could have been gained in the FPGA with a pipelined design, it greatly reduced the complexity of the control block and eliminated the complicated handshaking required in a pipelined design. Even though this initial assumption proved to be correct, in that the time budget was easily met with such an implementation, the design was constructed in such a fashion that the ability to further reduce the computation time of the hardware by operating more than one computation block at a time was possible.

Each of the computational sub-blocks was constructed with two essential signals that permitted the top-level FSM to control the entire system. These signals consisted of a *start* and *done* signal. The *start* signal was employed by the top-level FSM to communicate to a given sub-block that it should commence computation. On the other hand, the *done* signal was used by each sub-block to convey that it had completed its assigned computation. For each computational sub-block, the top-level FSM possessed a corresponding state in which it would reside while the associate sub-block performed its assigned computations. Each sub-block was guaranteed that the top-level FSM would not issue a *start* signal until its inputs had been completely updated and were ready to be processed. Once a *start* signal had been delivered, the FSM would remain in the same state until the associated sub-block finished its computations, upon which the FSM would assert a *start* signal to the next block and transition to the next state.

The state machine implemented in the hardware not only fulfilled the role of control of the hardware by issuing and awaiting these above-described signals, but also acted as the FSM for the entire algorithm. That is, the software implementation section of the algorithm was treated in the exact same manner as the hardware sub-blocks and employed the same control signals. Thus, the DSP was viewed as yet another sub-block and therefore required an associated state in the top-level FSM. Referring back to Figures 5.6 and 6.1, it can be seen that the sections of the algorithm assigned to

the hardware consist of image capturing, absolute image differencing, filtering, thresholding and the generation of a binary image. With the additional state required for the DSP, the top-level FSM contained the following states - START, IMAGE_GATHERING, DIFFERENCING, FILTERING, THRESHOLDING, BINARY_IMAGER.

In order to synchronize the hardware with the software running on the DSP, the DSP's general-purpose I/O (GPIO) lines were employed. Table 6.1 lists the GPIO pins and indicates the drive direction of the hardware, as well as the usage of each. The hardware drive directions were determined by analyzing both the video board and the FPGA board's usages of the GPIO while executing the image-chipping algorithm. Signals dsp_done and dsp_start, listed in this table, fulfill the required *start* and *done* signals by the top-level FSM for the DSP.

Table 6.1: Hardware Connections to DSP's General Purpose I/O

#	Signal Name	Hardware Drive Direction	Usage
1	GPIO< 0 >	In/Out	I^2C data bus (CHANNEL 1)
2	GPIO< 1 >	In/Out	Image-Chipping's State Machine Reset/I^2C data bus (CHANNEL 2)
3	GPIO< 2 >	In	Image-Chipping's State Machine Start Signal (dsp_done)/I^2C clock
4	GPIO< 3 >	In	I^2C Data Bus Direction Bit (1 - DSP drives the bus, 0 - video board drives the bus)
5	GPIO< 4 >	In	Video Board Reset Signal
6	GPIO< 5 >	Out	Image-Chipping's State Machine Done Signal (dsp_start)

The top-level FSM was designed to operate in two different modes each of which include the above listed states as a foundation. Selection of the current operating mode can be made through the debug/control registers, which are discuss in Section 6.9.2. These modes include *Normal Operation Mode* and *Debug Operation Mode* which are discussed below.

Normal Operation Mode

The hardware operates in the *Normal Operation Mode* by the default. It is in this mode that the hardware functions when it is running the image chipping algorithm. Figure 6.2 shows a pictorial representation of the operation of the FSM when *Normal Operation Mode* is selected. From this figure, it can be seen that the states that compose this FSM correspond to those mentioned above.

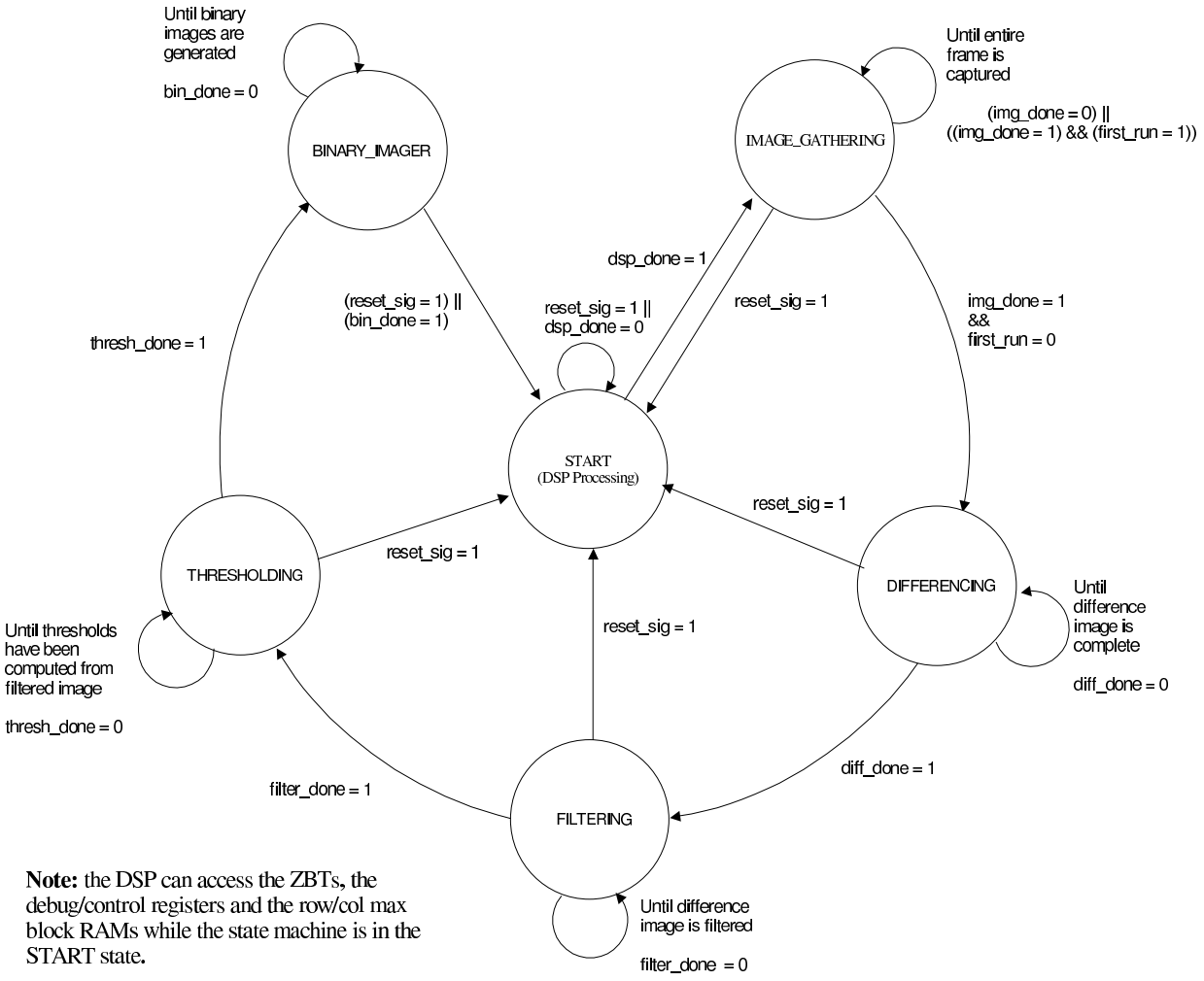


Figure 6.2: State Transition Diagram for the Top-Level State Machine Operating in Normal Mode

With the exception of *reset_sig*, a signal controlled by the DSP to place the FSM in a known state, all state transitions occur when the associated sub-block asserts its *done* signal. The IMAGE_GATHERING state has an additional requirement for a state transition to occur. That is, the *first_run* flag must be cleared in order for the top-level FSM to transition out of the IMAGE_GATHERING state to DIFFERENCING. This supplementary condition is due to the fact that the difference block cannot compute the absolute difference of two images until two images have been gathered. Thus, the first time the FSM traverses the loop from the START state through all the others back to the START state after the assertion of the *reset* signal, the FSM visits the IMAGE_GATHERING state twice.

Also shown in Figure 6.2, the START state corresponds to both the state returned to when the *reset_sig* signal is asserted and the state in which the FSM resides while the DSP is computing the software implementation part of the image chipping algorithm. In the START state, the DSP can access the ZBT memories, the debug/control registers and the row/col max block RAMs. Although when running the image chipping algorithm, the DSP need only the ability to access the debug/control registers and the ZBT memory in which the binary image resides, the added capability to access the row/col max block RAMs was incorporated for testing, as well as, for unforeseen future use.

Debug Operation Mode

Similar to how software can be compiled with additional debug information, yet still perform the same computations with a reduction in performance, the *Debug Operation Mode* is simply an expansion of the *Normal Operation Mode* that allows the user to gather additional information. Typically, in *Normal Operation Mode*, the user can neither view the intermediate results of the hardware sub-blocks as they are computed individually, nor can he force the inputs to a given sub-block. It is this ability to constrain the inputs to a given sub-block that proved to be essential in the debugging of the image chipping algorithm. Hence, it was this desired debug capability that drove the design process to separate the hardware into many sub-blocks each of which stored and gathered its outputs and inputs, respectively, from memory.

Figure 6.3 shows the operation of the FSM when the *Debug Operation Mode* is selected. Comparing

Figures 6.2 and 6.3, one can see that the two FSMs differ only by the fact that the *Debug Operation Mode* FSM has had an additional state, DEBUG, inserted between all the transitions which are caused by an assertion of a *done* signal in *Normal Operation Mode*. Identical to the START state, while in the DEBUG state the DSP can access the ZBT memories, the debug/control registers and the row/col max block RAMs. In this figure, it can also be seen that all transitions out of the DEBUG state are caused by either an assertion the *dsp_done* or *reset_sig* signal. In the *Debug Operation Mode* the *reset_sig* serves the same purpose as it did in the *Normal Operation Mode* in that it places the FSM in a known state. On the other hand, the usage of *dsp_done* signal to transition out of the DEBUG state permits the hardware to perform the next sub-block computation.

It is this halting in the DEBUG state until the *dsp_done* signal is asserted that affords the programmer the ability to pre-load the inputs to a given hardware sub-block and extract its outputs upon completion with the DSP. A discussion of this pre-loading/extracting technique employed to initially verify all the hardware sub-blocks is presented in Section 9.1.

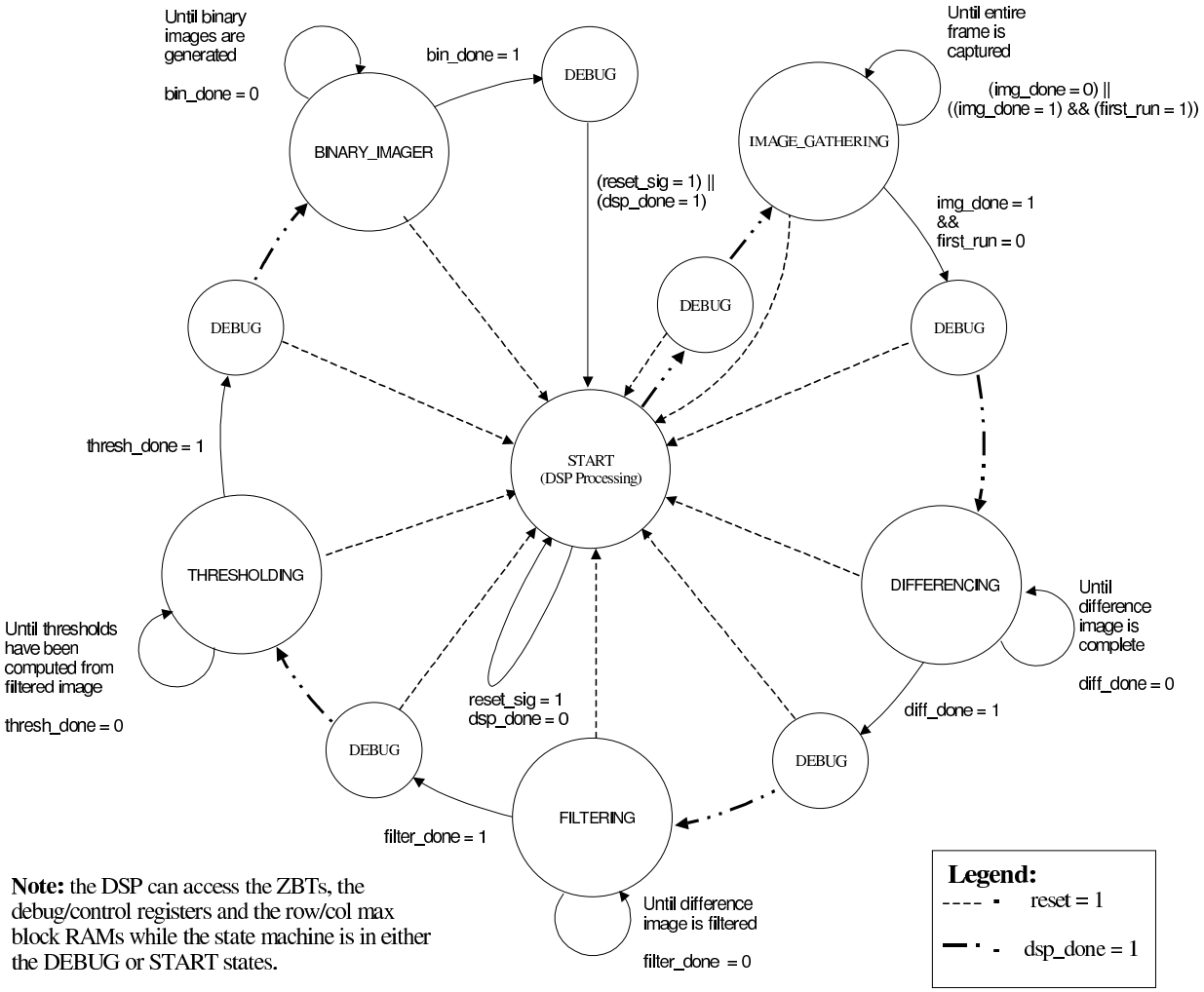


Figure 6.3: State Transition Diagram for the Top-Level State Machine Operating in Debug Mode

6.2 Memory Interface

Considering that the Virtex XCV1000 FPGA does not have enough internal block RAM (total number of block SelectRAM bits is 131,072[27]) to store all the intermediate images and values generated by the computational sub-blocks of the non-pipelined design, employment of the external ZBT RAM chips on the CAμS FPGA board was a necessity. As mentioned in Section 3.1.1, the FPGA board possesses five Samsung KM736V849 256Kx36-Bit Pipelined NtRAM^{TM}. Each memory chip can hold 9,437,184 bits[35]. With each captured image only being 228,480 bits (12 bits/pixel $\times$ 160 columns $\times$ 119 rows) and each intermediate image being 152,320 bits (8 bits/pixel $\times$ 160 columns $\times$ 119 rows) owing to the fact that only 8-bit processing is used following the absolute difference block, it is noted that a single memory chip would have been sufficient to retain all the required data of the image chipping algorithm.

However, to initially reduce the hardware computation time by allowing a sub-block to read and write within the same clock cycle, and to permit further reduction by operating more than one sub-block at a given time as mentioned in Section 6.1.1, each sub-block was connected to two memory chips while active. Typically, a computational sub-block reads its inputs from one memory and writes its outputs to a another. Having four computational sub-blocks within the FPGA (the difference block, filter block, threshold computing block and binary image generation block) required the use of all five memory chips contained on the FPGA board, all of which were connected through a RAM interface block as shown in Figure 6.1.

Each of the sub-blocks in the design uses a common memory interface to communicate with the ZBT RAM chips on the FPGA board. This interface, called *ram_interface* and illustrated in Figure 6.4, is used to hide some the low-level details of the late-late ZBT memory chips and to simplify their timing requirements by adding a layer of abstraction.[1] This module hides the required delayed presentation of data by two clock cycles relative to the address and read/write signal during a memory write. All the signals - data, read/write enable and address - are sent on a single clock edge by a computational sub-block and the RAM interface takes care of the two-clock-cycle latency. The RAM interface registers all signals that go into the ZBTs, as well as all signals that come out

[1]The *ram_interface* sub-block was developed in conjunction with Maneesh Soni and was based off of a design implemented by Wojciech Krawiec.

of ZBTs. Therefore, it takes four clock cycles for data to appear on the data bus after the address and read enable signals are applied.

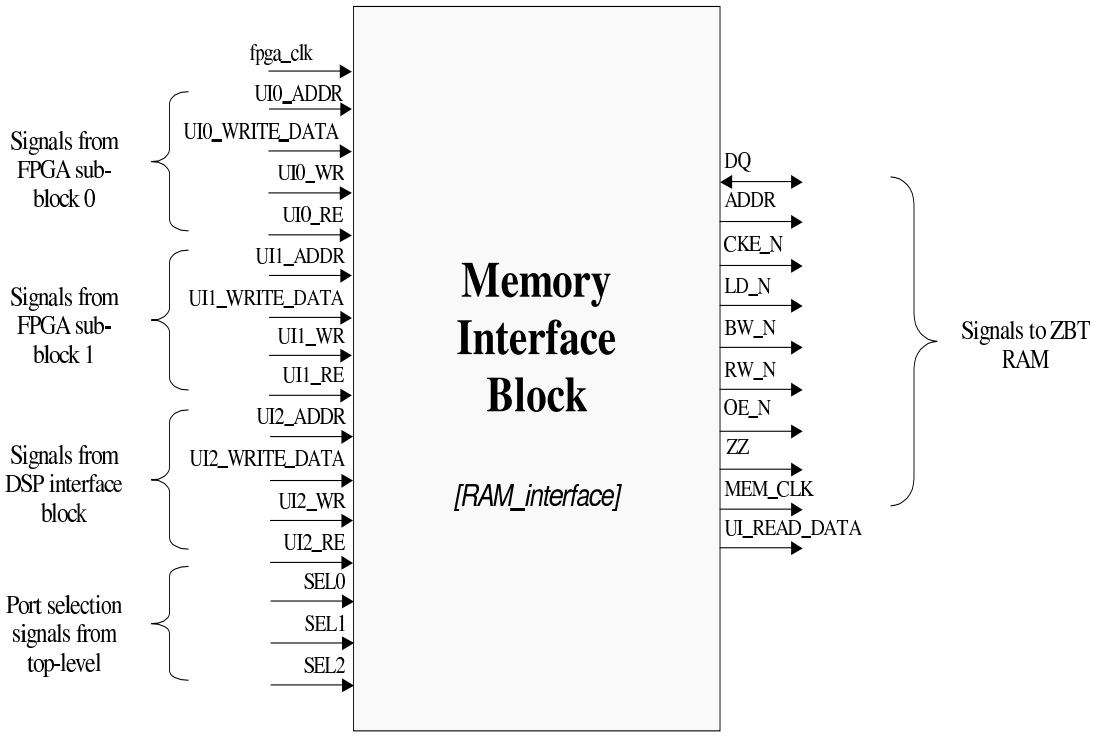


Figure 6.4: Block Diagram of Memory Interface Block

Table 6.2 provides a description of all the ports of the *ram_interface* along with their associated bit widths and drive directions. With each sub-block writing its outputs to memory, which in turn become the inputs for the following block, each memory needed the ability to communicate with at least two sub-blocks. Also, to allow the DSP to extract results from and insert data into the memory banks, the *ram_interface* needed the ability to communicate with the DSP. To keep all the ports of the *ram_interface* consistent so that a simple multiplexing of signals could be used, a *dsp_interface* was developed to convert the signals from the DSP into the format expected by the *ram_interface*. Further detail about the *dsp_interface* is presented in Section 6.9.

Figure 6.1 shows which sub-blocks can communicate with a given memory chip in addition to the DSP. Because each of the ZBTs are accessed by three blocks in the FPGA design, a 3-to-1 multiplexer has been used in the *ram_interface* block to support the required switching of input

Table 6.2: Input and Output Signals for the RAM Interface Block

#	Signal Name	Type	Bit Width	Description
1	DQ	In/Out	36	Data Inputs/Outputs
2	ADDR	Out	18	Address Inputs
3	CKE_N	Out	1	Clock Enable
4	LD_N	Out	1	Address Advance/Load
5	BW_N	Out	1	Byte Write Inputs
6	RW_N	Out	1	Read/Write Control Input
7	OE_N	Out	1	Output Enable
8	ZZ	Out	1	Power Sleep Mode
9	MEM_CLK	Out	1	Memory Clock
10	FPGA_CLK	In	1	Internal FPGA Clock
11	UI0_ADDR	In	18	User Port 0 Address
12	UI0_WRITE_DATA	In	36	User Port 0 Write Data
13	UI0_WR	In	1	User Port 0 Write Enable (active low)
14	UI0_RE	In	1	User Port 0 Read Enable (active low)
15	UI1_ADDR	In	18	User Port 1 Address
16	UI1_WRITE_DATA	In	36	User Port 1 Write Data
17	UI1_WR	In	1	User Port 1 Write Enable (active low)
18	UI1_RE	In	1	User Port 1 Read Enable (active low)
19	UI2_ADDR	In	18	User Port 2 Address
20	UI2_WRITE_DATA	In	36	User Port 2 Write Data
21	UI2_WR	In	1	User Port 2 Write Enable (active low)
22	UI2_RE	In	1	User Port 2 Read Enable (active low)
23	UI_READ_DATA	Out	36	All User Ports Read Data
24	SEL0	In	1	User Port Select Line 0
25	SEL1	In	1	User Port Select Line 1
26	SEL2	In	1	User Port Select Line 2

signals. The address bus, the write data bus and the read/write enable signals are multiplexed. Essentially, the *ram_interace* makes the single-ported ZBT RAM appear as a tri-ported singely active memory. The top-level FSM sends active-high select signals (sel0, sel1, or sel2) to control the multiplexers to connect the active sub-block to its required memory chips. It is also the responsibility of the top-level FSM to issue the select signals to the *bram_interface* discussed in the next section.

6.3 BRAM Interface

The block RAM interface, *bram_interface*, serves the same purpose for the internal block RAMs of the FPGA that the *ram_interface* provided for the external ZBT memories. That is, it makes the dual-ported block RAMs appear as quad-ported doubly active memories. In other words, each of the ports of the dual-ported block RAM is multiplexed between two user-ports. Therefore, at any given time only two of the four user-ports can communicate with the block RAM. However, unlike the *ram_interface*, the block RAM for which the *bram_interface* provides an interface is a sub-block of it. Figure 6.5 shows the black-box representation of the *bram_interface*, which depicts this encapsulation.

Considering the small size of the of data produced by the *threshold_block*, it was decided to keep the results that it produces internal to the FPGA. Referring back to Figure 6.1, it can be seen that the block RAMs were used to hold the *row_max's* and *col_max's* of the filtered absolute difference image. Also shown in this figure is the allocation of the four ports of each *bram_interface*. The *threshold_block* employs two user ports of each interface by itself, while the other two are distributed among the *binary_image_generator* block and the *dsp_interface*. Once again, the ability of the DSP to extract/set the contents of the block RAMs was not a necessity for the operating image chipping design, but was included to assist in the debug/verification process of the *threshold_block* and *binary_image_generator* sub-blocks.

Table 6.3 has been included to assist the reader in interpreting the ports shown in Figure 6.5. It provides further information about the input/output ports of the *bram_interface*, their associated widths and drive direction.

Table 6.3: Input and Output Signals for the BRAM Interface Block

#	Signal Name	Type	Bit Width	Description
1	CLKA	In	1	Port A clock
2	CLKB	In	1	Port B clock
3	UI0_A_ADDR	In	9	User 0 port A address
4	UI0_A_WRITE_DATA	In	8	User 0 port A input data bus
5	UI0_A_WR	In	1	User 0 port A write enable
6	UI0_A_EN	In	1	User 0 port A enable
7	UI0_B_ADDR	In	9	User 0 port B address
8	UI0_B_WRITE_DATA	In	8	User 0 port B input data bus
9	UI0_B_WR	In	1	User 0 port B write enable
10	UI0_B_EN	In	1	User 0 port B enable
11	UI1_A_ADDR	In	9	User 1 port A address
12	UI1_A_WRITE_DATA	In	8	User 1 port A input data bus
13	UI1_A_WR	In	1	User 1 port A write
14	UI1_A_EN	In	1	User 1 port A enable
15	UI1_B_ADDR	In	9	User 1 port B address
16	UI1_B_WRITE_DATA	In	8	User 1 port B input data bus
17	UI1_B_WR	In	1	User 1 port B write
18	UI1_B_EN	In	1	User 1 port B enable
19	UI_A_READ_DATA	Out	8	Port A read data bus
20	UI_B_READ_DATA	Out	8	Port B read data bus
21	SEL0	In	1	User 0 select pin
22	SEL1	In	1	User 1 select pin

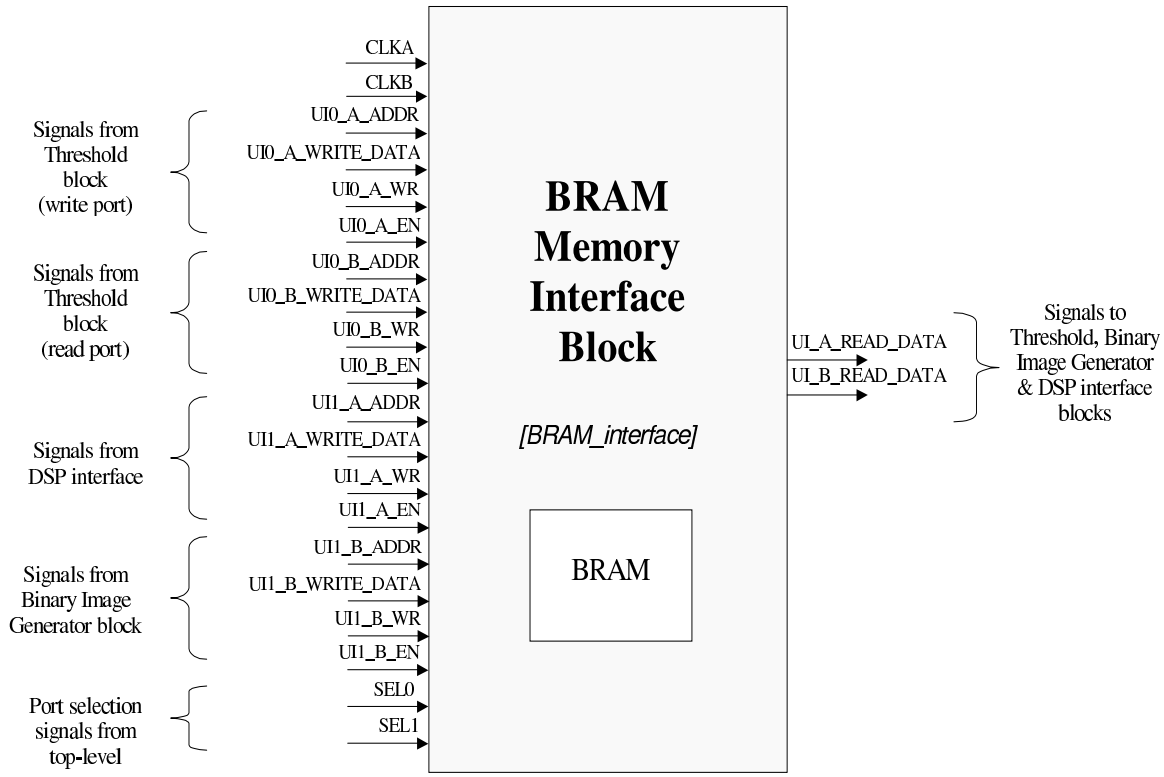


Figure 6.5: Block Diagram of BRAM Memory Interface Block

6.4 Image Capturing System

In order for the image-chipping algorithm to operate, it needs the ability to capture frames from a video source. This module is responsible for performing this essential task of capturing images and storing them to memory for further processing. This section describes the implementation of the image capturing system and its ability to acquire images from various video sources. Figure 6.6 shows to which subsection of the UMD algorithm the image capturing system corresponds. From this figure, one can see that the captured images are stored in a circular image buffer with size of two images. In actuality, this circular image buffer is stored in ZBT A as portrayed in Figure 6.1.

The image capturing system is made up of the following sub-blocks on the FPGA board:

1. **DECODER_INTERFACE**: This is a simple module that converts input signals from the video decoder chip to the inputs expected by the *image_grabber* block.

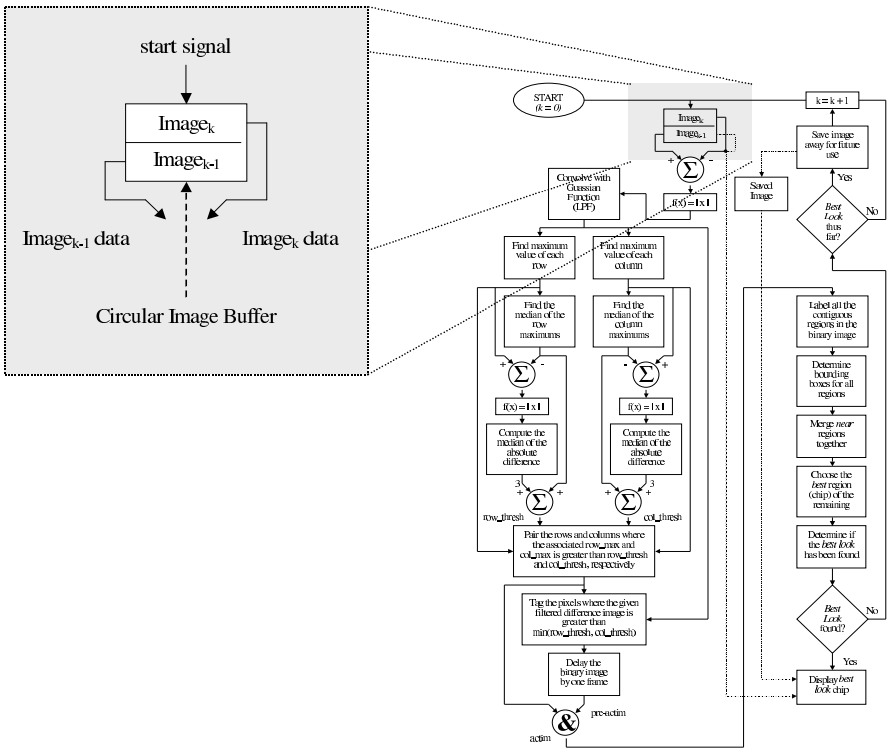


Figure 6.6: Flow Diagram of Image Capturing System

2. **ALPHA_INTERFACE**: This block converts inputs from an Indigo Alpha camera to the inputs expected by the *image_grabber*.

3. **BUS MUX**: This is a high-level multiplexer that connects the inputs of the *image_grabber* block to either the outputs of the *decoder_interface* block or the *alpha_interface* block, based on the state of a debug/control register.

4. **IMAGE_GRABBER**: This block accepts a stream of digital video and stores images to memory.

Figure 6.7 is a diagram of the image capturing system that depicts how the sub-blocks are connected, as well as, portrays the division of the input signals among its sub-components. Referring back to Figure 6.1, one can see the relationship of the above-described sub-blocks of the image capturing system to the remaining hardware sub-blocks within the FPGA.

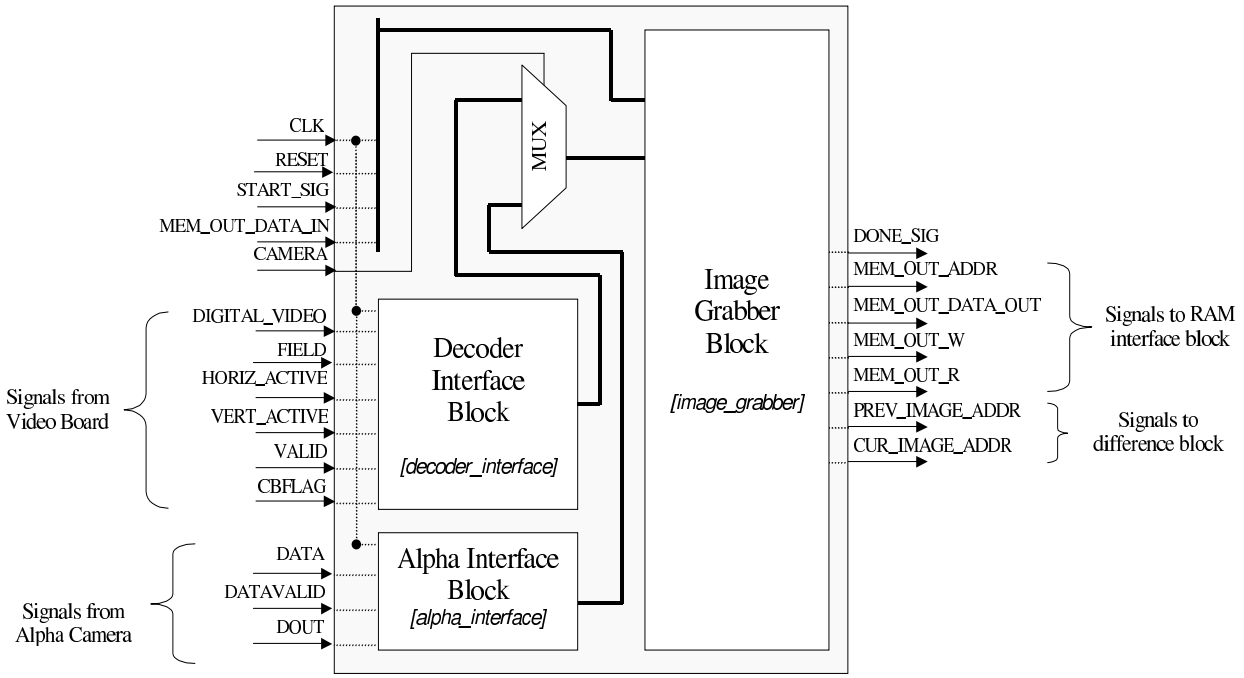


Figure 6.7: Components of Image Capturing System

In order to give the reader a better understanding of this system, each of the sub-blocks will be discussed in more detail in the following sections. Finally, a brief discussion of the hardware layout of the video board has been included to complete the analysis of the image capturing system as a whole when the video input source is NTSC, PAL or SECAM.

6.4.1 DECODER_INTERFACE Block

When the image-chipping algorithm is processing data from an NTSC, PAL or SECAM video stream it employs the *decoder_interface*, whose black-box representation is shown in Figure 6.8. The *decoder_interface* is responsible for the conversion of the signals passed from the CAμS video board in the standard CCIR 601 (ITU-R 601) 4:2:2 digital video format, to the expected format of the *image_grabber* sub-block.

The *decoder_interface* block generates the appropriate inputs to the *image_grabber* block, which is discussed in Section 6.4.4, in the following manner:

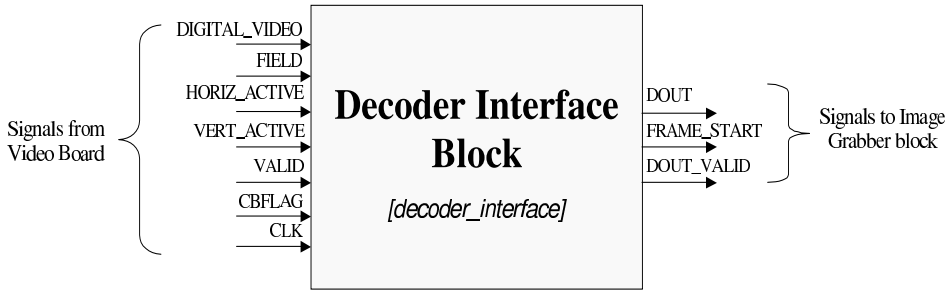


Figure 6.8: Block Diagram of Decoder Interface Block

1. **DOUT**: This signal is simply a delay version of DIGITAL_VIDEO so that it correctly aligns with FRAME_START. The first and last 4 pixels of each row of video have been zeroed to eliminate artifacts due to accidentally sampling the horizontal blanking interval.

2. **FRAME_START**: This signal is produced as the output of a rising-edge detector on the VERT_ACTIVE input signal.

3. **DOUT_VALID**: This signal reflects the clock cycles where DOUT has valid pixel data. Thus, the signal is produced by logically AND'ing HORIZ_ACTIVE, VERT_ACTIVE, VALID and FIELD, which reflects the active region of the video stream. The FIELD input was included in the generation of this signal to ensure that the system is always sampling the same video field. The output of the AND gate is appropriately delayed so that is correctly aligns with the corresponding DOUT data.

Table 6.4 lists all the inputs and outputs of the *decoder_interface* block and expounds upon the knowledge gained from Figure 6.8.

Table 6.4: Input and Output Signals for the Decoder Interface Block

#	Signal Name	Type	Bit Width	Description
1	DIGITAL_VIDEO	In	8	8bit digital video input
2	FIELD	In	1	Video field flag
3	HORIZ_ACTIVE	In	1	Horizontal video active
4	VERT_ACTIVE	In	1	Vertical video active
5	VALID	In	1	Valid video flag
6	CBFLAG	In	1	CB flag (Not Used)
7	DOUT	Out	12	Pixel Data
8	FRAME_START	Out	1	Frame Start
9	DOUT_VALID	Out	1	Data Valid
10	CLK	In	1	Bt835 Clock

6.4.2 ALPHA_INTERFACE Block

The *alpha_interface* sub-block shown in Figure 6.9 serves a similar purpose to the above-describe *decoder_interface* sub-block in that it performs a conversion from an incompatible video stream to one that can be used by the *image_grabber* block.[2] However, the *alpha_interface* differs from *decoder_interface* in that it converts from a non-standard proprietary video stream format, which is the output of the Indigo Alpha IR camera. Table 6.5 lists all the inputs and outputs of the *alpha_interface* block.

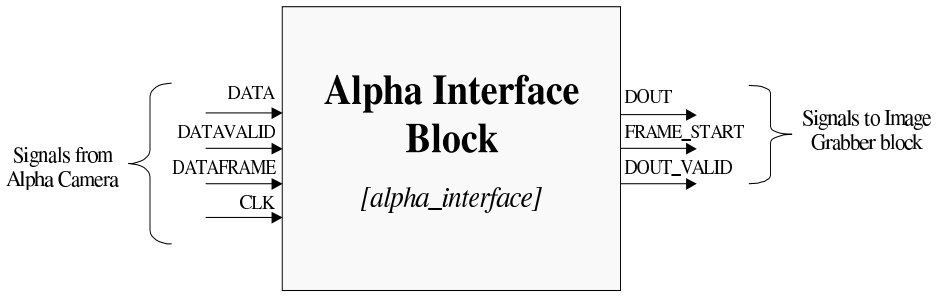


Figure 6.9: Block Diagram of Alpha Interface Block

[2]The *alpha_interface* sub-block was developed by Wojciech Krawiec.

Table 6.5: Input and Output Signals for the Alpha Interface Block

#	Signal Name	Type	Bit Width	Description
1	DATA	In	4	Pixel input (from the camera)
2	DATAFRAME	In	1	Frame input (from the camera)
3	DATAVALID	In	1	Valid data input (from the camera)
4	DOUT	Out	12	Pixel Data
5	FRAME_START	Out	1	Frame Start
6	DOUT_VALID	Out	1	Data Valid
7	CLK	In	1	Alpha Clock

6.4.3 Bus Multiplexer Block

The bus multiplexer is not actually a sub-block of the image-chipping algorithm. That is, this component is created at the top-level design in which all the remaining sub-blocks are instantiated and connected just like the top-level FSM described in Section 6.1.1. However, a brief discussion has been included at this point to inform the reader how to change the control signal to the bus multiplexer. The select line for the multiplexer is controlled by one of the debug/control registers that reside in the top-level design. Therefore, to change the camera input interface one must change the corresponding debug/control register with the DSP. A detailed discussion of the top-level debug/control registers is presented in Section 6.9.2. The methods by which one can manipulate the debug/control registers are included in Section 7.1.1. The reader is directed to these two sections for further information pertaining to controlling the camera interface selection bus multiplexer.

6.4.4 IMAGE_GRABBER Block

The *image_grabber* block is illustrated in Figure 6.10. As described above in Section 6.4, this block is responsible for capturing an image from an incoming digital video stream and storing it to memory so that it can be used later for further processing. In order to accomplish this task, this sub-block needs the ability communicate with a digital video stream source (i.e., the *decoder_interface* or the *alpha_interface* block), a *ram_interface*, the *difference* block and the top-level state machine.

Table 6.6 lists all the signals employed by the *image_grabber* block to communicate with these other sub-blocks.

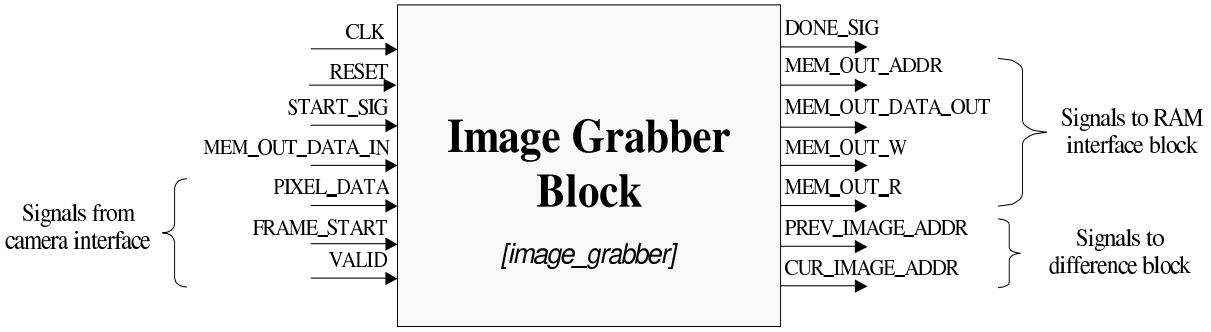


Figure 6.10: Block Diagram of Image Grabber Block

Table 6.6: Input and Output Signals for the Image Grabber Block

#	Signal Name	Type	Bit Width	Description
1	CLK	In	1	Clock (rising edge active)
2	RESET	In	1	Asynchronous reset to bring the block to a known state
3	START_SIG	In	1	Start processing signal
4	DONE_SIG	Out	1	Done processing
5	MEM_OUT_ADDR	Out	18	Output RAM address bus
6	MEM_OUT_DATA_IN	In	36	Output RAM data in bus (Not Used)
7	MEM_OUT_DATA_OUT	Out	36	Output RAM data out bus
8	MEM_OUT_W	Out	1	Output RAM write signal (Active Low)
9	MEM_OUT_R	Out	1	Output RAM read signal (Active Low)
10	PIXEL_DATA	In	12	Pixel data from camera interface
11	FRAME_START	In	1	Frame start signal from camera interface
12	VALID	In	1	Data valid signal from camera interface
13	PREV_IMAGE_ADDR	Out	18	Start address of the previous image
14	CUR_IMAGE_ADDR	Out	18	Start address of the current image

The *image_grabber* block uses the RESET and START_SIG signals from the top-level finite state machine to enter a known state and to determine when to capture a frame, respectively. That is, when the *image_grabber* block receives a start signal from the top-level FSM, it begins the process of acquiring a new frame into memory. The first step of acquiring an image is to wait for the next FRAME_START signal given by the digital video stream source block.

Upon receiving a FRAME_START signal from either the *alpha_interface* or *decoder_interface*, depending on which camera interface is selected, the *image_grabber* block monitors the VALID input signal and generates the appropriate signals to the *ram_interface* to store the current contents of the input PIXEL_DATA whenever VALID is asserted. Once an entire image has been received (of size 160 rows $\times$ 119 columns), the *image_grabber* block transitions to the DONE state for a clock cycle in which it informs the top-level FSM that it has finished its task by asserting the DONE_SIG. Then, the *image_grabber* block transitions back to the START state in which it waits for the next START_SIG indicating that it should capture another frame. Figure 6.11 presents the state transition diagram for *image_grabber* block.

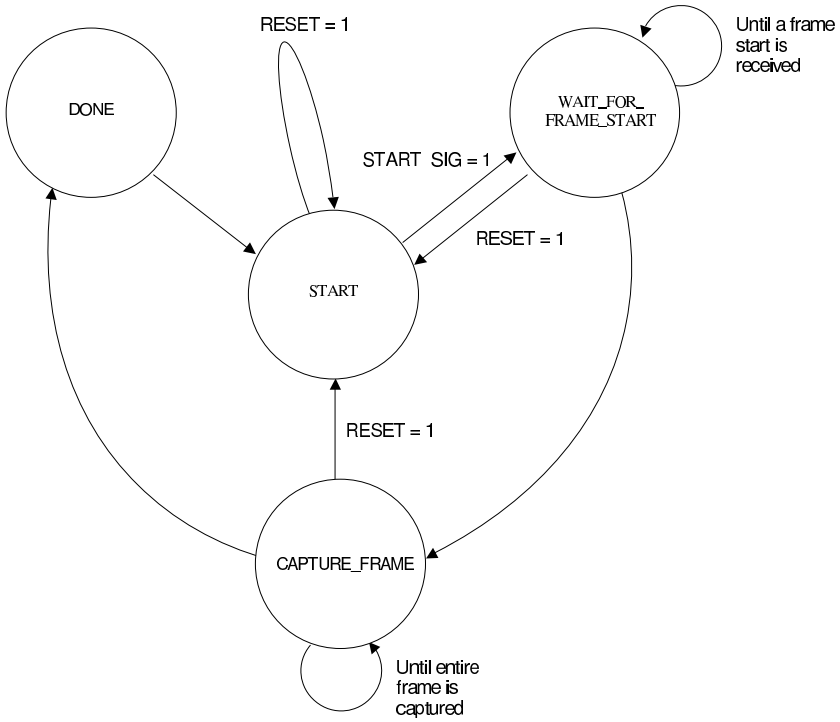


Figure 6.11: State Transition Diagram for state machine in Image Grabber Block

Because the next sub-block in the image-chipping algorithm computes the difference of the current frame and the previous captured frame, the *image_grabber* block was designed to capture the images into a circular buffer of length two. The start addresses of the current and previous images swap each time a new image is captured. To keep the synchronization of these start addresses consistent

between the *image_grabber* block and the following *difference* block, two additional output signals were added, PREV_IMAGE_ADDR and CUR_IMAGE_ADDR. These signals connect between these two sub-blocks and allow the *image_grabber* to convey the start locations of these two images to the *difference* block.

6.4.5 Video Board Hardware Layout

When one employs the *decoder_interface* in the Virtex XCV1000, it is assumed that a CAμS video card has been mated to the CAμS stack and is properly functioning. In order to accept an NTSC video stream into the image-chipping algorithm, one must employ a CAμS video card.

When functioning properly, the video board uses its video decoder chip (Bt835) to convert analog NTSC video into digital CCIR 601 4:2:2 video. The decoder chip not only performs this conversion, but also scales the normally un-scaled active video frame size of 720 $\times$ 480 pixels down to 160 $\times$ 119, which is the operating frame size for the image-chipping algorithm.

Figure 6.12 presents a top-level block diagram of the FPGA design that resides in a Virtex XCV1600E on the video board. The diagram depicts all the signals that the FPGA employs to communicate with the two video decoder (Bt835) chips, the two encoder (Bt860) chips, and the configurable clock, which are present on the video board. Table 6.7 has been included to provide a brief description of each signal shown in this figure. As shown in the figure, the video board design consists of the following sub-components:

1. **Clock Division Logic**: This block divides down the CLKX2 pixel clock from the video decoder chip of CHANNEL1 to a CLKX1. A rising edge detector on the VERT_ACTIVE_1 signal is used to synchronize the CLKX1 with the luminance values, rather than the chrominance values.

2. **I^{2}C Interface Block**: This block allows the DSP to communicate using the I^{2}C protocol to the video encoder and decoder chips to program their internal registers.

3. **Registered signals**: All the signals that are passed to the FPGA board to be used by the *decoder_interface* are registered to limit clock skew.

Table 6.7: Input and Output Signals for Video Board Top Level Block

#	Signal Name	Type	Bit Width	Description
1	GCK0	In	1	Clock 2X from video decoder (CHANNEL 1)
2	DIGITAL_VIDEO_1	In	16	Digital video from video decoder (CHANNEL 1)
3	FIELD_1	In	1	Field signal from video decoder (CHANNEL 1)
4	HORIZ_ACTIVE_1	In	1	Horizontal active signal from video decoder (CHANNEL 1)
5	VERT_ACTIVE_1	In	1	Vertical active signal from video decoder (CHANNEL 1)
6	VALID_1	In	1	Valid video signal from video decoder (CHANNEL 1)
7	BT835_RESET_1_N	Out	1	Video decoder device reset (CHANNEL 1)
8	BT835_CBFLAG_1	In	1	Chrominance Blue flag from video decoder (CHANNEL 1)
9	BT835_OE_1_N	Out	1	Video decoder output enable (CHANNEL 1)
10	BT835_PWRDN_1	Out	1	Video decoder power down (CHANNEL 1)
11	BT835_I2CCS_1	Out	1	Video decoder I^2C address selector (CHANNEL 1)
12	BT835_I2C_CLK_1	Out	1	Video decoder I^2C clock (CHANNEL 1)
13	BT835_I2C_DATA_1	In/Out	1	Video decoder I^2C data (CHANNEL 1)
14	BT860_RESET_1_N	Out	1	Video encoder device reset (CHANNEL 1)
15	BT860_ALTADDR_1	In/Out	1	Video encoder I^2C address selector (CHANNEL 1)
16	BT860_I2C_DATA_1	In/Out	1	Video encoder I^2C data (CHANNEL 1)
17	BT860_I2C_CLK_1	Out	1	Video encoder I^2C clock (CHANNEL 1)
18	BT835_RESET_2_N	Out	1	Video decoder device reset (CHANNEL 2)
19	BT835_CBFLAG_2	In	1	Chrominance Blue flag from video decoder (CHANNEL 2)
20	BT835_OE_2_N	Out	1	Video decoder output enable (CHANNEL 2)
21	BT835_PWRDN_2	Out	1	Video decoder power down (CHANNEL 2)
22	BT835_I2CCS_2	Out	1	Video decoder I^2C address selector (CHANNEL 2)
23	BT835_I2C_CLK_2	Out	1	Video decoder I^2C clock (CHANNEL 2)
24	BT835_I2C_DATA_2	In/Out	1	Video decoder I^2C data (CHANNEL 2)
25	BT860_RESET_2_N	Out	1	Video encoder device reset (CHANNEL 2)
26	BT860_ALTADDR_2	In/Out	1	Video encoder I^2C address selector (CHANNEL 2)
27	BT860_I2C_DATA_2	In/Out	1	Video encoder I^2C data (CHANNEL 2)
28	BT860_I2C_CLK_2	Out	1	Video encoder I^2C clock (CHANNEL 2)
29	USERIO_A	In/Out	25	FPGA Board USER I/O A (Not Used)
30	DSP_PD	In/Out	6	DSP General Purpose I/O Pins
31	CONFIG_CLOCK_PD_N	Out	1	Configurable clock power down pin
32	CONFIG_CLOCK_R	Out	7	Configurable clock frequency selector bus R
33	CONFIG_CLOCK_S	Out	3	Configurable clock frequency selector bus S
34	CONFIG_CLOCK_V	Out	9	Configurable clock frequency selector bus V
35	IFU	Out	14	Intra-FPGA Bus (used to transmit video signals)

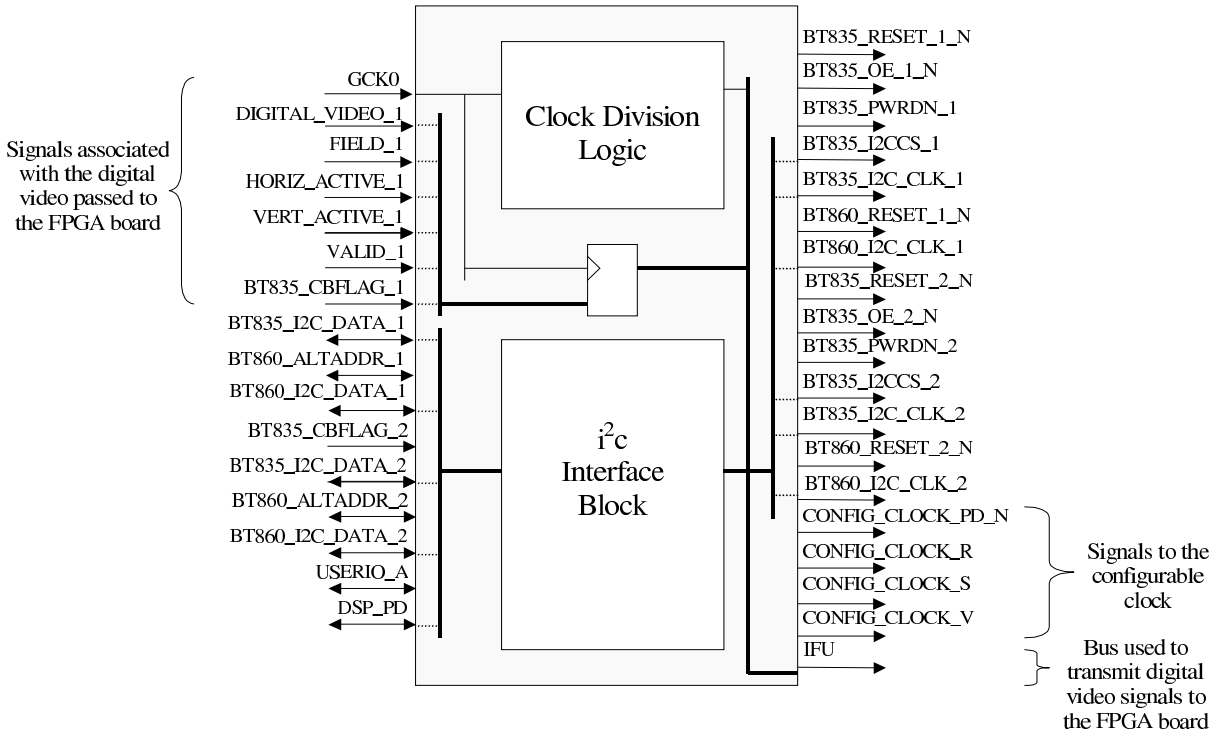


Figure 6.12: Block Diagram of the Video Board Top Level

To connect the registered video decoder signals from the video board to the FPGA board, the intra-FPGA user (IFU) bus was utilized. In an attempt to eliminate any confusion that could arise due to the loss of signal name appropriateness by assigning these signals to a non-descriptive bus, Table 6.8 has been included to provide a mapping between the helpful registered signals' name and their corresponding IFU bus bits.

Table 6.8: Registered Digital Video Connections to IFU Bus

#	Signal Name	IFU Bus Bit
1	DIGITAL_VIDEO_1< 8 >	0
2	DIGITAL_VIDEO_1< 9 >	1
3	DIGITAL_VIDEO_1< 10 >	2
4	DIGITAL_VIDEO_1< 11 >	3
5	DIGITAL_VIDEO_1< 12 >	4
6	DIGITAL_VIDEO_1< 13 >	5
7	DIGITAL_VIDEO_1< 14 >	6
8	DIGITAL_VIDEO_1< 15 >	7
9	HORIZ_ACTIVE_1	8
10	VERT_ACTIVE_1	9
11	VALID_1	10
12	FIELD_1	11
13	BT835_CBFLAG_1	12
14	GCK0 divided by 2	13

6.5 Differencing

The objective of the *difference* block is to enhance the portions of the frame that have changed from one image to the next.[3] Computing the absolute difference of two consecutive images produces a third image that contains non-zero pixel values only where there are discrepancies between the two input images. Ideally, these non-zero pixels correspond to moving objects in the field of view; however, if the intensity from one image to the next changes, the *diffference* block will also produce non-zero pixels throughout the difference image. Similarly, if there are erroneous pixels due to noise or an un-calibrated pixel on the focal-plane array, non-zero pixels will exist in the difference image even if there were no moving objects in field of view.

In both of the above-described cases, the non-zero pixels in the resulting absolute difference image could be falsely interpreted as motion. It is the computations performed by the blocks that follow the *difference* block that attempt to eliminate these false positives of motion. The effect of the low-pass *filter* block is a *smoothing* the image. This smoothing reduces the large discontinuities introduced between adjacent pixels by a erroneous pixel. On the other hand, the dynamic *thresh-*

[3]The *difference* sub-block was developed in conjunction with Mark Buchero.

olding block attempts to determine the appropriate threshold values to be used such that common pixel offsets in the difference image, caused by intensity changes, are ignored. More complete discussions of the *filter* and *threshold* blocks can be found in Sections 6.6 and 6.7, respectively.

Figure 6.13 illustrates the subsection of the UMD image chipping algorithm assigned to the *difference* block. As can also be seen in this figure, the *difference* block is not only responsible for computing the absolute difference of the current and the previous images, it must also store away the previous image as the *saved* image if the software deems necessary. The latter operation was assigned to the *difference* block because it was concluded that it was the most appropriate block to implement this process owing to the fact that it already needed to stream in the previous image to compute the absolute difference. Therefore, it was simply a matter of storing away the already loaded image back to memory if so necessary.

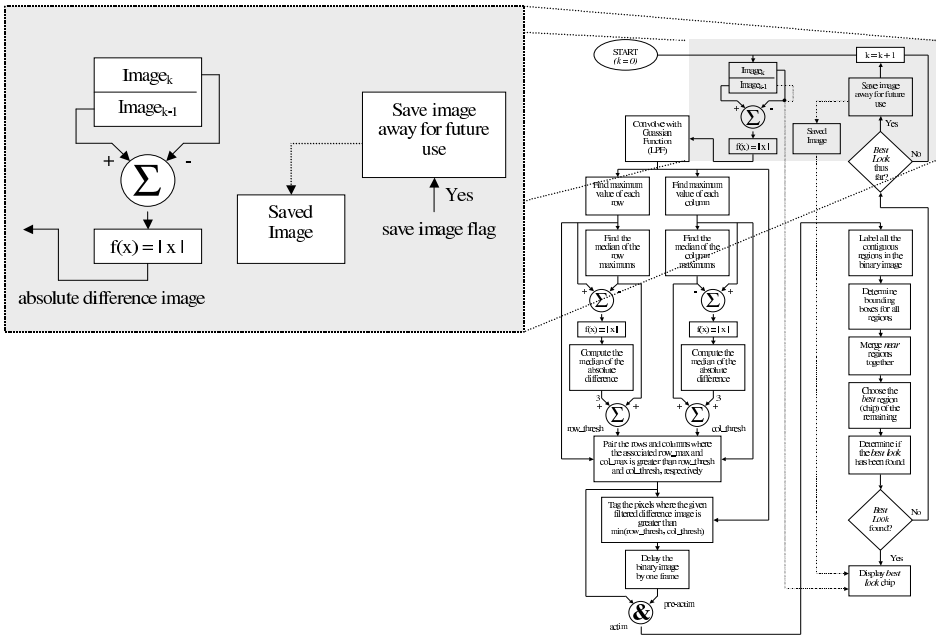


Figure 6.13: Flow Diagram of Difference Block Computation

Figure 6.14 portrays the black-box representation of the *difference* block, while Table 6.9 includes additional details pertaining to the input/outputs ports shown in this figure. Included in both the table and figure are the START_ADDRESS_1 and START_ADDRESS_2 ports. These buses are driven by the *image_grabber* block and are used to determine the addresses of the current and

previous images in the circular buffer contained in ZBT A. Passing these signals directly from one block to the next ensures synchronization of image locations between the *image_grabber* and *difference* blocks.

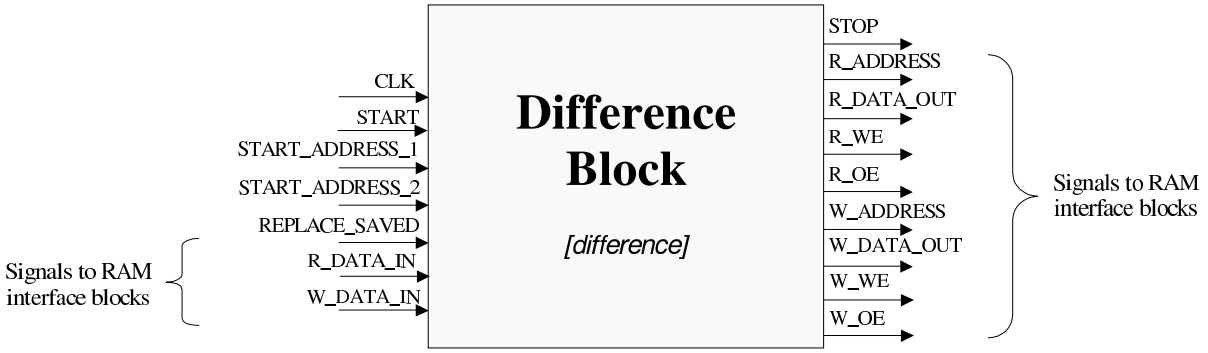


Figure 6.14: Block Diagram of Difference Block

High-level control of the *difference* block is accomplished with a synchronous finite state machine. Although it was initially thought that an FSM was not required to control the trivial streaming operations that the *difference* block performs, after further thought it was realized that it would greatly simplify the complex signaling demanded by the pipelined architecture of the *ram_interface*. Looking at Figure 6.15 in which the high-level FSM of the *difference* block is shown, one can clearly see that the majority of its states are related to the synchronization with valid data received from the *ram_interface*. This high percentage of states dedicated to memory synchronization will become more apparent after reading the following discussion of the *difference* block's FSM's operation.

Upon receiving a START signal from the top-level FSM, the FSM of the *difference* block transitions to the FIRST_READ state. In the FIRST_READ state, the *difference* block presents the appropriate address of the first pixel of the previous image stored in the circular buffer to the *ram_interface* of ZBT A. Next, it transitions to the SECOND_READ state in which it presents the address of the first pixel of the current image to the *ram_interface* of ZBT A. The FSM then iterates through the two states one more time during which it presents the address to the second pixels of the previous and current image in states FIRST_READ and SECOND_READ, respectively. It is noted that due to the latency associated with the *ram_interface* no valid image data will have been received yet from the *ram_interface*. Therefore, the *difference* block withholds computation of the difference

Table 6.9: Input and Output Signals for the Difference Block

#	Signal Name	Type	Bit Width	Description
1	CLK	In	1	Clock (rising edge active)
2	START	In	1	Start processing signal
3	STOP	Out	1	Done processing
4	START_ADDRESS_1	In	18	Address of the first image in memory
5	START_ADDRESS_2	In	18	Address of the second image in memory
6	REPLACE_SAVED	In	1	Indicator to save image 1 as the "best so far" image
7	R_ADDRESS	Out	18	Address being read from image 1 or image 2
8	R_DATA_IN	In	36	Data received from image 1 or image 2
9	R_DATA_OUT	Out	36	Not used
10	R_WE	Out	1	Active Low Write Enable signal for ZBT
11	R_OE	Out	1	Active Low Read Enable signal for ZBT
12	W_ADDRESS	Out	18	Address being written to - Either the difference or saved image
13	W_DATA_IN	In	36	Not used
14	W_DATA_OUT	Out	36	Data being written to W_ADDRESS
15	W_WE	Out	1	Active Low Write Enable signal for ZBT
16	W_OE	Out	1	Active Low Read Enable signal for ZBT

and the storage of the previous image in the saved image memory space.

Because it takes five clock cycles to get valid data back from a memory including the cycle in which the address and read/write enables signals are presented, one additional clock cycle must occur before valid data returns from the memory interface to be processed by the *difference* block. Therefore, the FSM enters the FIRST_READ_DELAY state in which it presents the address of the third pixel of the previous image to the *ram_interface*. Upon transitioning into the GET_IMAGE1_DATA state, the first pixel of the previous image is present on the read data bus of the *ram_interface*. This pixel is immediately written back out to ZBT B, if necessary.

The FSM then alternates between the GET_IMAGE1_DATA and GET_IMAGE2_DATA states until the entire difference image has been generated. During this ping-ponging of states, the *difference* block continues process of alternating reads from previous and current image, computing and storing the difference image to ZBT B, and storing the previous image in ZBT B, if indicated to do so by the corresponding debug/control register. Once the entire difference image has been computed and written back out to memory, the FSM transitions to the DONE state in which it asserts the

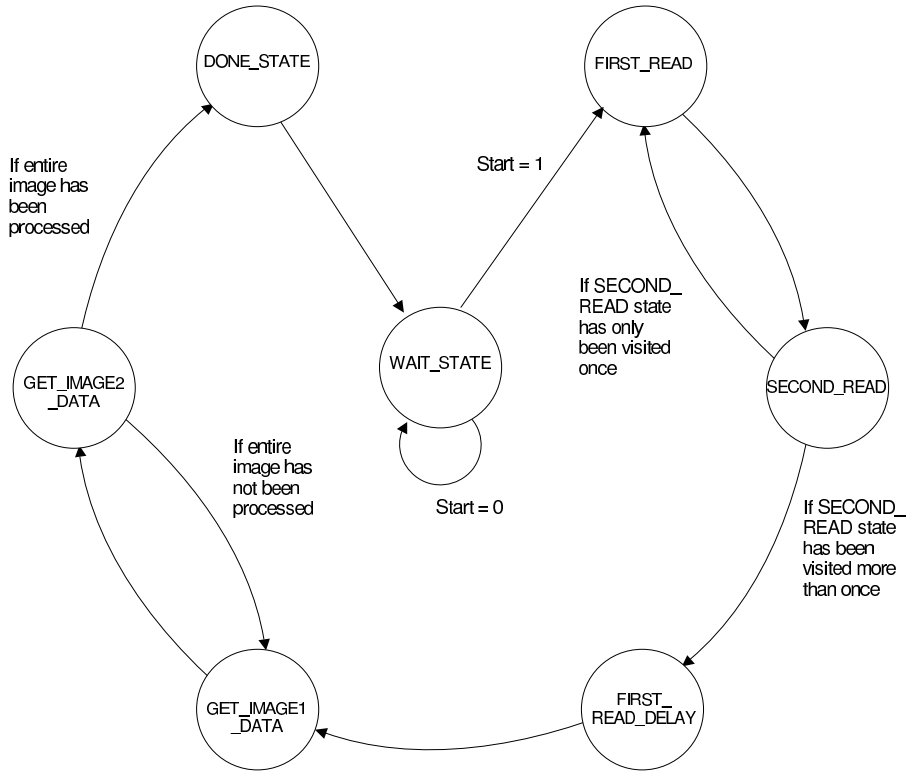


Figure 6.15: State Transition Diagram for state machine in the Difference Block

DONE signal to inform the top-level FSM that the difference computation has been completed. The *difference* block's FSM then returns to the WAIT_STATE in which it remains awaiting another START signal to begin its operation again.

As a final note about the *difference* block, a brief discussion about bit widths has been included. The original image data entering the *difference* block captured from the Indigo Alpha camera possesses 12-bits of resolution per pixel. However, the remaining computational blocks that follow the *difference* block expect to process 8-bit data. Therefore, it was the duty of the *difference* block to translate this 12-bit data down to 8-bit data during the absolute differencing process.

This bit reduction was accomplished by computing the full 12-bit absolute difference. If the result of this computation was greater than 255 (the largest value that can be held in an unsigned 8-bit data word), then it was set to 255. Otherwise, the value stored back to memory as the difference image was truly the absolute difference. This approach of masking off the lower 8-bits of the difference and forcing it to a maximum value if the difference exceeds 255 was chosen over simply masking the most significant 8-bits of the difference because of the results of an analysis of the data produced by the Alpha camera. That is, the data generated by the Alpha camera represents absolute temperature. By examining the data of multiple images it was found that typically the upper nibble of most pixels in a given image possess the same value. Therefore, assuming no extreme changes in the field of view of the camera between one captured image to the next occurs, masking off the upper eight bits of the 12-bit difference would typically result in four out of eight bits being significant. This low percentage of significant bits is caused by the fact that upper nibble of the 12-bit difference normally results in zero. Consequently, in an attempt to retain the greatest amount of significant data, the lower 8-bits of the difference were chosen with the added stipulation that the difference would have an upper bound of 255.

6.6 Filtering

The *filter* block applies a 2-dimensional low-pass filter to the difference image computed by the previous stage. As mentioned in Section 6.5, this computation is performed to reduce the high pixel values contained in the difference image due to erroneous pixels captured in the original images. The *filter* block is responsible for extracting the difference image from memory, filtering it, and storing the resulting filtered image back into memory. Figure 6.16 shows the subsection of the UMD algorithm to which these above-described operations correspond. Details about the specific filter implemented in the *filter* block can be acquired in Section 9.2. Figure 6.17 gives a block diagram of the *filter* block. This figure depicts all input and output ports that are used by this block. Table 6.10 contains additional information pertaining to the ports of the *filter* block.

The *filter* block developed for the image chipping algorithm was merely a wrapper for the *fir2d* module developed at BAE Systems by Wojciech Krawiec. The *fir2d* module accepts pixels in a

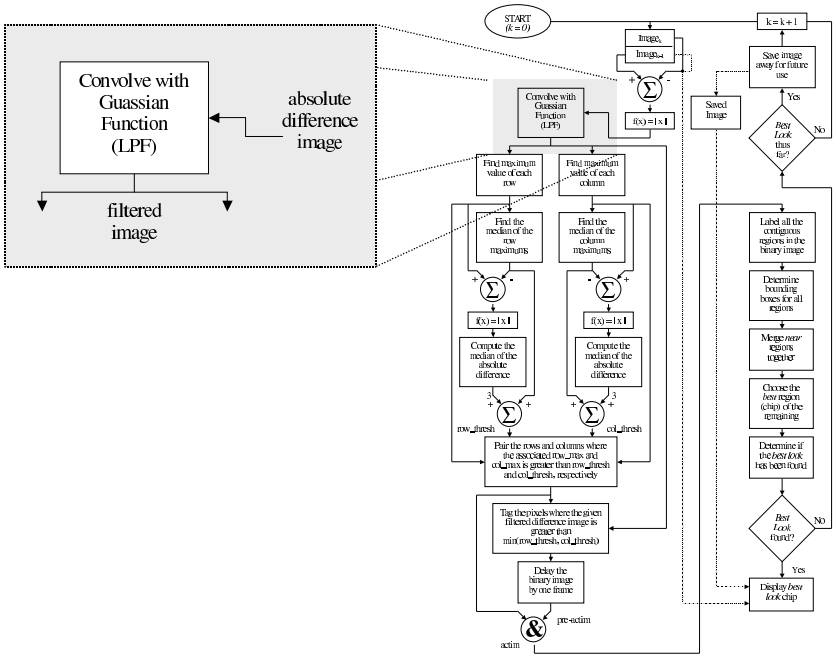


Figure 6.16: Flow Diagram of Filter Block Computation

raster scan format and outputs the 2D filtered image in a raster-scan format as well. Simply stated, the *filter* block developed for the this algorithm extracts pixel data from memory, feeds it to the input of the *fir2d* block and stores the outputs of the *fir2d* sub-block back to memory for further processing.

The *filter* block must communicate with the top-level FSM, three *ram_interfaces* and its sub-block *fir2d*. Similar to the *image_grabber* block, the *filter* block uses the RESET and START_SIG to return to a known state and to determine when to operate, respectively.

Upon receiving a START_SIG from the top-level FSM, the *filter* block enters the READ_ONLY state. In this state, the *filter* block begins the process of extracting pixels out of ZBT RAM B in raster-scan order. Since the filter has a (known) latency associated with it, no valid pixel data is present at its output until a known number of clock cycles have elapsed. Until valid data becomes available, the *filter* block does not write any results of the filter out to memory but does continue to read from memory and fill the filter. Hence, the state is labeled READ_ONLY.

After the filter latency has been exceeded and valid pixels are present at the output of the filter, the

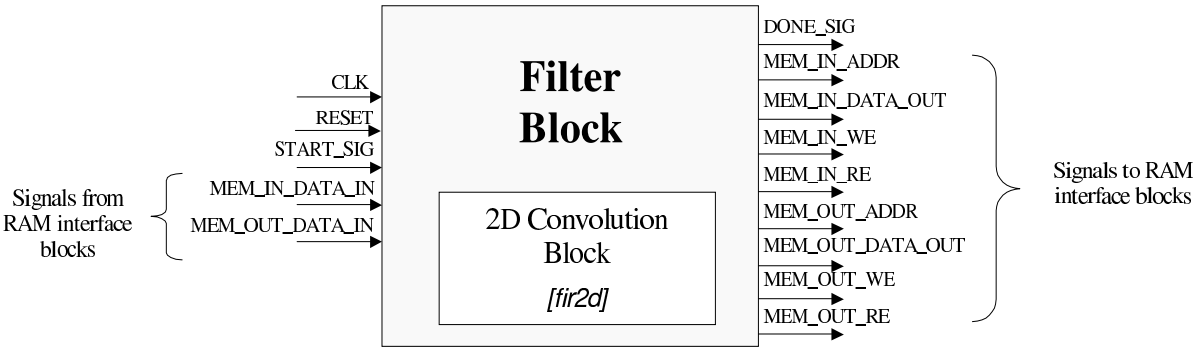


Figure 6.17: Block Diagram of Filter Block

filter block transitions to the READ_WRITE state. In the READ_WRITE state, the *filter* block not only continues to read from ZBT RAM B, but also begins to write the results to ZBT RAMs C and D to be used later by the *threshold block* and the *binary_image_generator* block, respectively. The *filter* block remains in this mode of operation until the entire image (160 columns $\times$ 119 rows) has been extracted from memory, upon which it transitions to the WRITE_ONLY state.

In the WRITE_ONLY state, the block feeds the input to the *fir2d* block with zeros to pad the filter with zeros where no real image data exists. At the same time, the *filter* block continues to store the output-filtered image to ZBT RAMs C and D.

Finally, when the entire filtered image has been stored to ZBT RAMs C and D, the *filter* block transitions to the DONE state for a clock cycle in which it informs the top-level FSM that it has completed by asserting the output DONE_SIG. The state transition diagram for the *filter* block has been included in Figure 6.18.

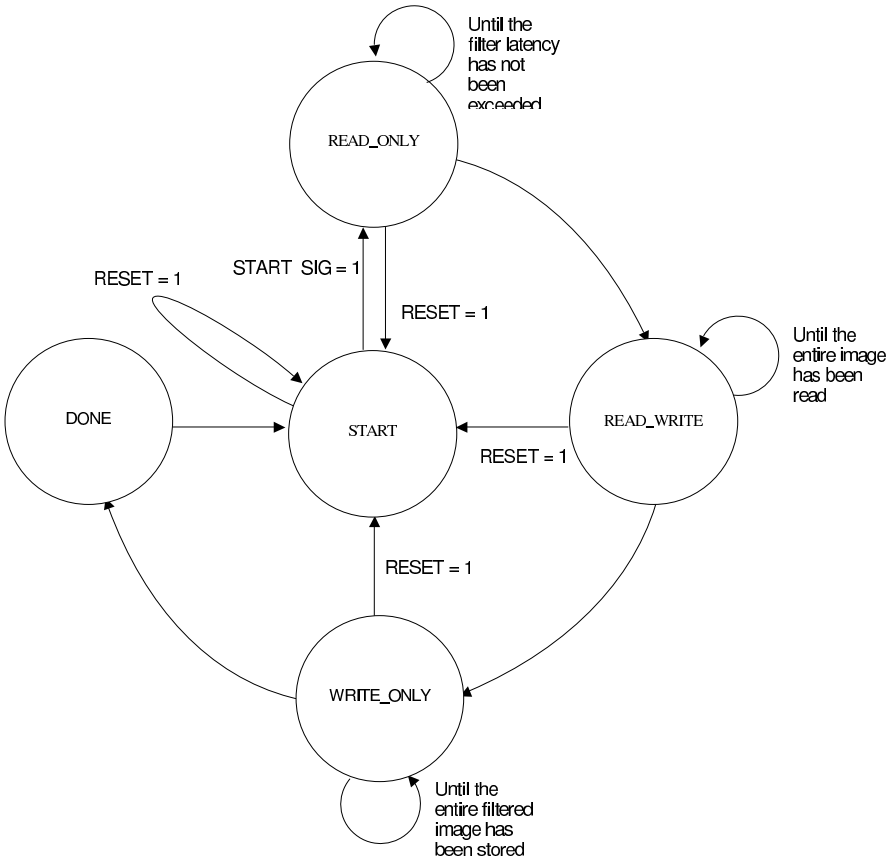


Figure 6.18: State Transition Diagram for state machine in the Filter Block

Table 6.10: Input and Output Signals for the Filter Block

#	Signal Name	Type	Bit Width	Description
1	CLK	In	1	Clock (rising edge active)
2	RESET	In	1	Asynchronous reset to bring the block to a known state
3	START_SIG	In	1	Start processing signal
4	DONE_SIG	Out	1	Done processing
5	MEM_IN_ADDR	Out	18	Input RAM address bus
6	MEM_IN_DATA_IN	In	36	Input RAM data in bus
7	MEM_IN_DATA_OUT	Out	36	Input RAM data out bus (Not Used)
8	MEM_IN_RE	Out	1	Input RAM read signal (Active Low)
9	MEM_IN_WR	Out	1	Input RAM write signal (Active Low)
10	MEM_OUT_ADDR	Out	18	Output RAM address bus
11	MEM_OUT_DATA_IN	In	36	Output RAM data in bus (Not Used)
12	MEM_OUT_DATA_OUT	Out	36	Output RAM data out bus
13	MEM_OUT_WE	Out	1	Output RAM write signal (Active Low)
14	MEM_OUT_RE	Out	1	Output RAM read signal (Active Low)

6.7 Thresholding

The *threshold* block determines the dynamic row and column thresholds, *row_thesh* and *col_thresh*, from the filtered absolute difference image according to the algorithm depicted in Figure 6.19.[4] These dynamic thresholds are used in conjunction with the filtered image to produce a binary image which represents regions of interest to be further examined. The production of this binary image by making use of these above described inputs is accomplished by the *binary_image_generator* block, which is presented in Section 6.8. The thresholds are computed dynamically for each filtered image as opposed to employing constant thresholds in an attempt to eliminate false detections of motion as mentioned in Section 6.5, as well as, to prevent the oversight of motion that could occur if the thresholds were fixed. Because the *threshold* block was required to perform multiple complicated operations, unlike the streaming data operations assigned to the other hardware sub-blocks, its required computations were distributed among internal sub-blocks.

The threshold computation block, illustrated in Figure 6.20, consists of the following modules:

[4]The *threshold* sub-block was developed in conjunction with Maneesh Soni.

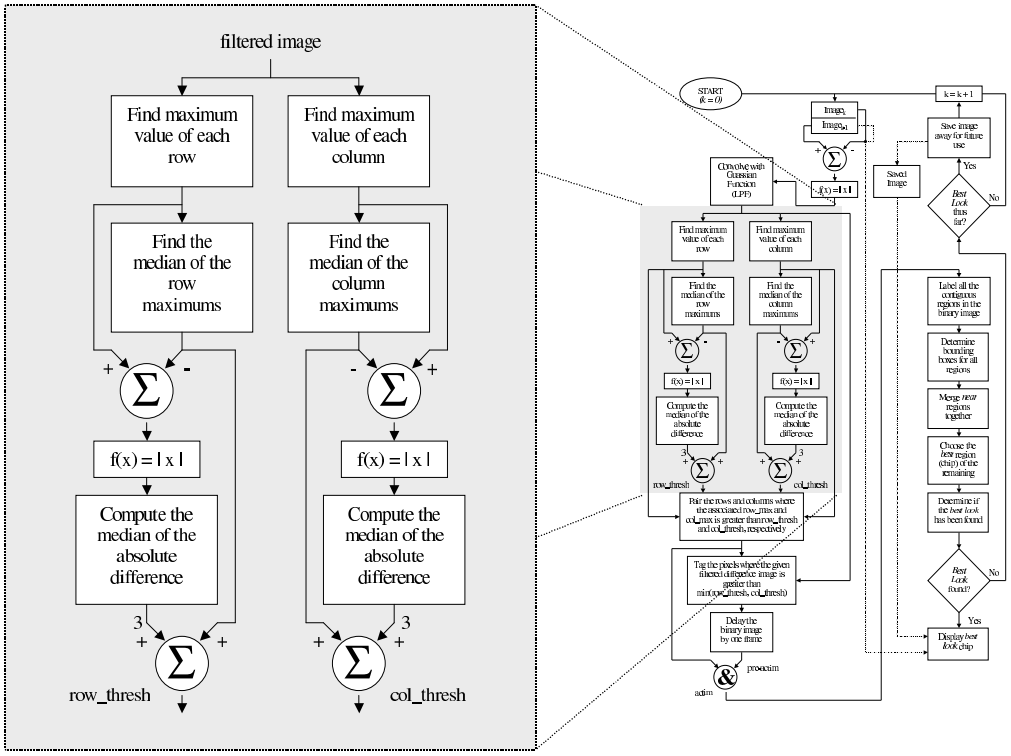


Figure 6.19: Flow Diagram of Threshold Block Computation

1. **MUX2TO1**: A 9-bit-wide bus multiplexer used to switch the address lines to each of the block RAMs that are used to store the row and column maximum values.

2. **MAXCALC**: This block reads the filtered difference image data from ZBT RAM C, determines the maximum intensity pixels in each row and each column of the image and stores these maximums in two block RAMs in the FPGA to be used by the *mediancalc* module and the *binary_image_generator* block.

3. **MEDIANCALC**: Using the maximum pixel values determined by the *maxcalc* sub-block, two *mediancalc* blocks are activated to find threshold values for row and column, respectively.

All input and output signals to the dynamic threshold computation block are listed in Table 6.11. A discussion of each of the modules of the *threshold* block is contained in the following sections.

Table 6.11: Input and Output Signals for the Threshold Block

#	Signal Name	Type	Bit Width	Description
1	CLOCK	In	1	Clock (rising edge active)
2	RESET	In	1	Asynchronous reset to bring the block to a known state
3	START	In	1	Start processing signal
4	DONE	Out	1	Done processing
5	ADDR	Out	18	Address bus for ZBT that stores filtered difference image
6	DATA_IN	In	36	Data Input Bus for ZBT that stores filtered difference image
7	DATA_OUT	Out	36	Data Output Bus for ZBT that stores filtered difference image
8	RE_N	Out	1	Active Low Read Enable signal for ZBT
9	WR_N	Out	1	Active Low Write Enable signal for ZBT
10	ROWMAX_ADDR_READ	Out	9	Address Read Bus for ROWMAX Block RAM
11	ROWMAX_DATA_READ	In	8	Data read bus for ROWMAX Block RAM
12	ROWMAX_ADDR_WRITE	Out	9	Address write bus for ROWMAX Block RAM
13	ROWMAX_DATA_WRITE	Out	8	Data write bus for ROWMAX Block RAM
14	ROWMAX_WE	Out	1	Write enable signal for ROWMAX Block RAM
10	COLMAX_ADDR_READ	Out	9	Address Read Bus for COLMAX Block RAM
11	COLMAX_DATA_READ	In	8	Data read bus for COLMAX Block RAM
12	COLMAX_ADDR_WRITE	Out	9	Address write bus for COLMAX Block RAM
13	COLMAX_DATA_WRITE	Out	8	Data write bus for COLMAX Block RAM
14	COLMAX_WE	Out	1	Write enable signal for COLMAX Block RAM
15	ROW_THRESHOLD	Out	8	Value of row threshold
16	COL_THRESHOLD	Out	8	Value of column threshold
17	row_medianvalue	Out	8	Intermediate row threshold value (used for debugging)
18	row_median2value	Out	8	Intermediate row threshold value (used for debugging)
19	col_medianvalue	Out	8	Intermediate column threshold value (used for debugging)
20	col_median2value	Out	8	Intermediate column threshold value (used for debugging)

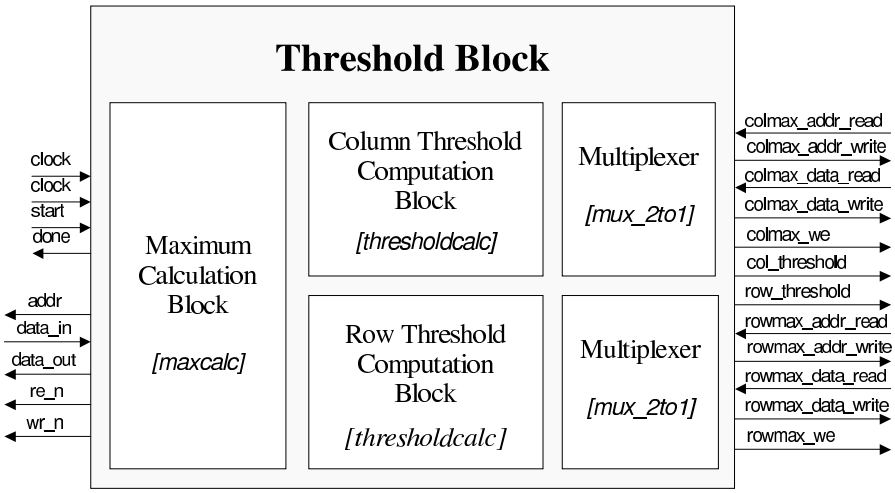


Figure 6.20: Block Diagram of Threshold Block

6.7.1 MUX2TO1 Block

Because the block RAMs in which the *row_max* and *col_max* values are stored needed to be accessed by both *maxcalc* and *mediancalc* blocks, and the *maxcalc* block employed the two user ports provided to *threshold* block from the *bram_interface*, one of these *bram_interface's* user ports had to be shared between the *maxcalc* and *mediancalc* blocks. In order to use that same port by both modules, the address bus for each of the user ports to the *bram_interfaces* were multiplexed. This multiplexing was done by the *mux2to1* block.

6.7.2 MAXCALC Block

After the filter smooths the difference image and stores it in ZBT RAMs C and D, the top-level FSM asserts the START signal to the *threshold* block. As the first step in the generation of the dynamic thresholds, the filtered image data is processed by the *maxcalc* block. The *maxcalc* block determines the associated row and column maximums for each row and column in the filtered image and stores these values in a block RAM for further processing. The determination of these desired maximums is accomplish in the following manner.

As the pixels of the filtered difference image are streamed in from ZBT C in a raster-scan format,

the values of the row and column maximums are updated as deemed necessary. In order to ensure that the true maximum for every row is determined, the value of the first pixel is placed in the register which holds the running maximum value of the current row. As mentioned above, while the data for a given row is read from memory the running maximum value is updated if the read pixel value exceeds that contained in the maximum register. Upon the completion each row of the filtered image, the maximum of the given row has been found and resides in the running maximum register. This value is written out to the appropriate location in the *row_max* block RAM so that the next row's maximum can be found.

The mechanism of finding a maximum in a column is similar to the one used for rows. However, rather than using 160 running maximum registers, the *col_max* block RAM was used to retain the running maximum for each column. Identical to the treatment of the rows, pixels in the first row of the filtered image are, by default, written as current maximum values of their respective columns. This ensures that the true maximums of all columns are computed and are not erroneous due to their comparison to previous unknown memory values as would be the case if this precaution were not taken.

The FSM used to control the *maxcalc* module was simple and is shown in Figure 6.21. From this figure it can be seen that is composed of only two states, MAX and SUSPEND. The START signal used to trigger the *maxcalc* module's FSM to transition to the MAX state is directly connected to the START port of the *threshold* block. Thus, it is issued by the top-level FSM of the image chipping algorithm. Once in the MAX state the FSM remains until the last pixel has been processed, upon which a DONE signal is issued that is sent as a START signal to the *mediancalc* blocks and the FSM transitions back to the SUSPEND state.

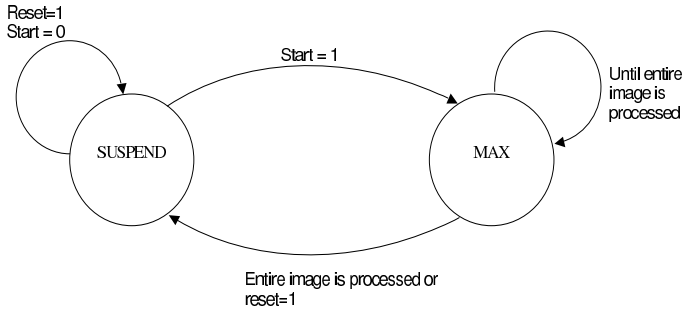


Figure 6.21: State Transition Diagram for state machine in the Maxcalc Block

6.7.3 MEDIANCALC Block

Upon receiving a START signal from the *maxcalc* module, the *mediancalc* block begins processing. The processing of the *row_max* and the *col_max* values is done concurrently. Because the number of columns is larger than the number of rows, the processing time required for the *col_max* data is larger than that for the *row_max's* . Consequently, the *mediancalc* module operating on the column maximums data will always complete after the one assigned to the row maximums and therefore has its DONE signal directly connected to the *threshold* blocks DONE port to be sent to the top-level FSM.

The process of computing the threshold values shown pictorially in Figure 6.19 can be broken into the following four steps once the column and row maximums have been found[34]:

1. Find the median of *col_max* values, and assign this to median1.
2. Find the value |median1 − *col_max*[i]| for each *col_max* value.
3. Find the median of those absolute values, and assign this to median2.
4. Calculate the threshold value: median1 + 3 × median2.

Note: The steps listed above refer to the computations performed to determine the column threshold. The row threshold can be produced by substituting *row_max* for every *col_max* in the list.

To find the median, a novel approach suggested by Dave Campagna, formerly of BAE System, was implemented. Because the pixels could only take on 256 values, the pixel values could be used

to address into a block RAM to produce a histograms of the row and column maximums. Once generated, these histograms could be analyzed to determine center of mass, which in turn lead to the median value[34]. Each *mediancalc* block employs two additional block RAMs. One block RAM is used to retain the histogram while the other stores the absolute values. The top-level control for the *mediancalc* block was accomplished with a synchronous finite state machine. In order to perform the complicated operations listed above, the *mediancalc's* FSM was quite intricate as can be seen in Figure 6.22.

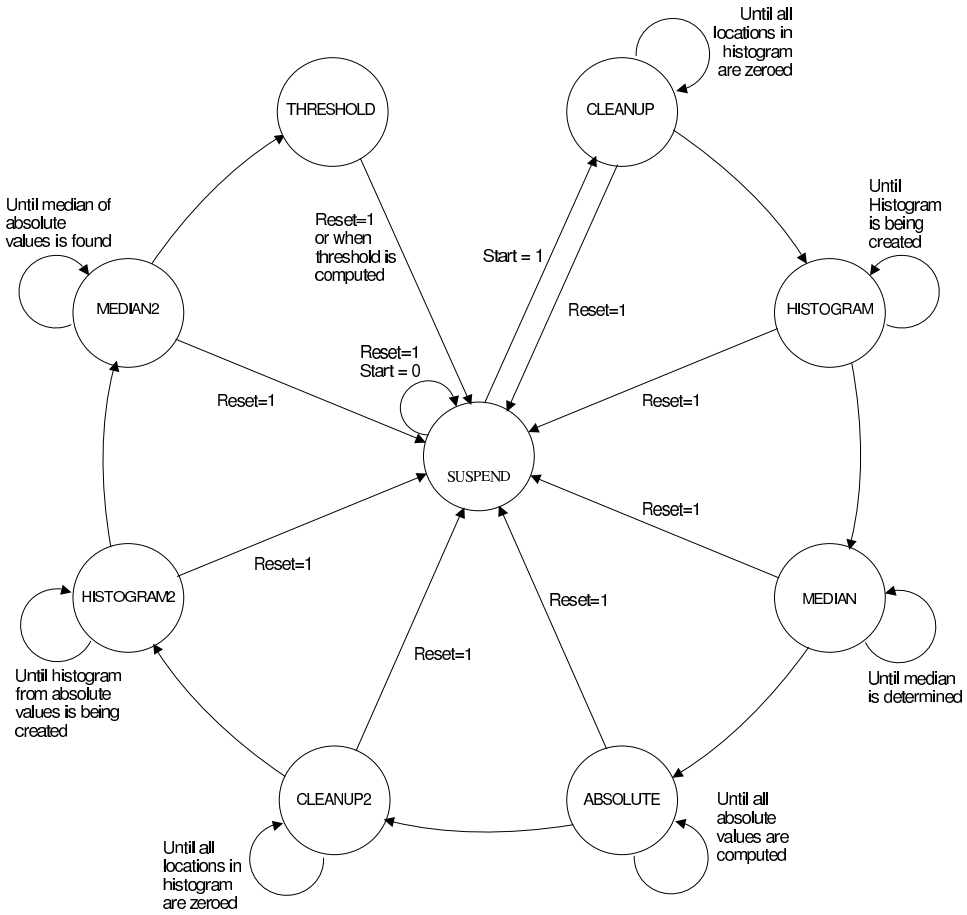


Figure 6.22: State Transition Diagram for state machine in the Thresholdcalc Block

After receiving a START signal from the *maxcalc* block, the state machine makes a transition from the SUSPEND state to the CLEANUP state. In the CLEANUP state, all locations of the

block RAM containing the histogram are cleared. After all the 256 possible locations of the histogram block RAM have been cleared, the FSM transitions to the HISTOGRAM state in which the histogram of the associate maximum data is constructed. Next, the block transitions to the to the MEDIAN state, where it remains until the histogram entries have been accumulated and the median value has been determined. Then, in the ABSOLUTE state, Step 2 is computed.

States CLEANUP2, HISTOGRAM2 and MEDIAN2 repeat the process outlined for states CLEANUP, HISTOGRAM and MEDIAN with the exception that these states determine the median of the result of Step 2 rather than that of the associated maximums. These states accomplish the computation explained in Step 3. Finally, the FSM transitions to the THRESHOLD state in which the computation lain out by Step 4 is performed an assertion of the DONE signal is generated to indicate the completion of the computation to the top-level FSM. The FSM then returns to the SUSPEND state to be run again.

6.8 Binary Image Generation

Following the computation of the dynamic thresholds, the system produces a binary image. This binary image is created using the previously computed row and column thresholds, the filtered difference image and the row and column maximums of this image. With all of these inputs the *binary_image_generator* block produces a binary image in accordance with computation depicted in Figure 6.23.

As mentioned above, in order to generate the binary image the *binary_image_generator* has to extract previously obtained results from various sub-blocks. These results include the *row_thresh*, *col_thresh*, *row_max's* and *col_max's* from the *threshold_block* and its associated block RAMs, and the filtered difference image from memory. Since the *binary_image_generator* read/writes its intermediate output (*pre-actim*) to a different memory from which it reads the filtered difference image, this block needs to be able to communicate with two *ram_interfaces*, in addition to the block RAMs and direct connections to the *threshold_block* and the top-level FSM.

Figure 6.24 shows the large number of number of ports that are required by the *binary_image_generator*. Table 6.12 describes the signals in this figure. As with most the computational blocks, the *bi-*

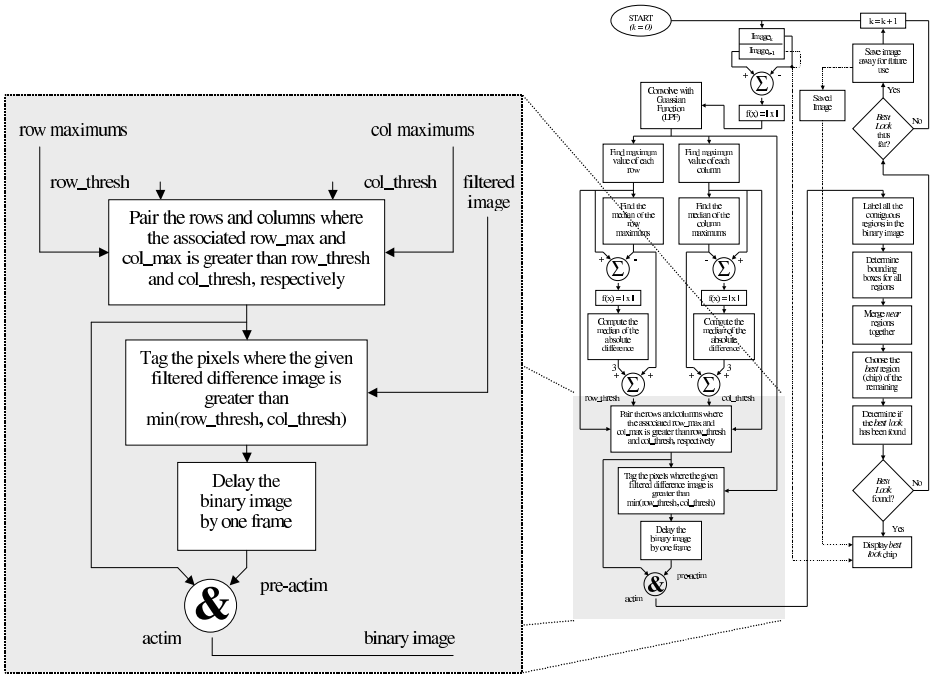


Figure 6.23: Flow Diagram of Binary Image Generation Block

nary_image_generator block uses the RESET and START_SIG to return to a known state and to determine when to operate, respectively. Upon receiving a START_SIG the *binary_image_generator* block transitions to the PAIR_MAX_AND_PRE_ACTIM in which it makes use of the *row_max's*, *col_max's*, *row_thresh*, *col_thresh*and *pre-actim* from the last run to produce the binary image (*actim*).

Table 6.12: Input and Output Signals for the Binary Image Generator Block

#	Signal Name	Type	Bit Width	Description
1	CLK	In	1	Clock (rising edge active)
2	RESET	In	1	Asynchronous reset to bring the block to a known state
3	START_SIG	In	1	Start processing signal
4	DONE_SIG	Out	1	Done processing
5	MEM_IN_ADDR	Out	18	Input RAM address bus
6	MEM_IN_DATA_IN	In	36	Input RAM data in bus
7	MEM_IN_DATA_OUT	Out	36	Input RAM data out bus
8	MEM_IN_RE	Out	1	Input RAM read signal (Active Low)
9	MEM_IN_WR	Out	1	Input RAM write signal (Active Low)
10	MEM_OUT_ADDR	Out	18	Output RAM address bus
11	MEM_OUT_DATA_IN	In	36	Output RAM data in bus
12	MEM_OUT_DATA_OUT	Out	36	Output RAM data out bus
13	MEM_OUT_WE	Out	1	Output RAM write signal (Active Low)
14	MEM_OUT_RE	Out	1	Output RAM read signal (Active Low)
15	ROW_THRESH	In	8	Threshold of the rows
16	COL_THRESH	In	8	Threshold of the cols
17	BRAM_COL_Q	In	8	Column Max Block RAM output
18	BRAM_COL_D	Out	8	Column Max Block RAM input
19	BRAM_COL_ADDR_IN	Out	9	Column Max Block RAM input address
20	BRAM_COL_ADDR_OUT	Out	9	Column Max Block RAM output address
21	BRAM_COL_WE	Out	1	Column Max Block RAM write enable
22	BRAM_COL_CLK	Out	1	Column Max Block RAM clock
23	BRAM_ROW_Q	In	8	ROW Max Block RAM output
24	BRAM_ROW_D	Out	8	ROW Max Block RAM input
25	BRAM_ROW_ADDR_IN	Out	9	ROW Max Block RAM input address
26	BRAM_ROW_ADDR_OUT	Out	9	ROW Max Block RAM output address
27	BRAM_ROW_WE	Out	1	ROW Max Block RAM write enable
28	BRAM_ROW_CLK	Out	1	ROW Max Block RAM clock

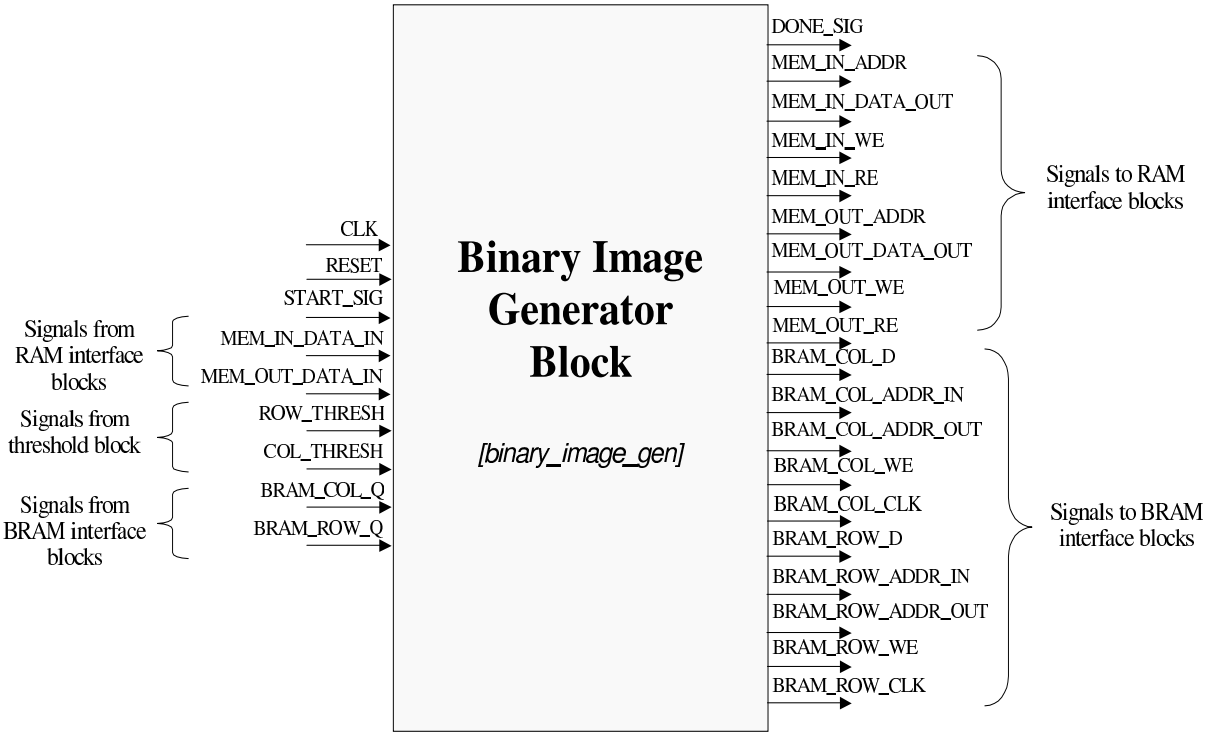


Figure 6.24: Block Diagram of Binary Image Generator Block

Once the entire binary image has been generated, the *binary_image_generator* transitions for one clock cycle to the PRE_ACTIM_ADDRESS_SETUP state in which it configures the address bus signals for the production of the new *pre-actim* binary image. On the next clock edge, the block cycles into the GENERATE_PRE_ACTIM state where it produces the *pre-actim* image with all the same inputs that were used to produce *actim*, with the exception of *pre-actim* and the addition of the filtered image pixels. Because the *pre-actim* from the previous run has already been used, the new *pre-actim* that is generated overwrites the same memory that held the old *pre-actim* .

Upon generating the entire *pre-actim* image, the state machine transitions to the DONE state for a single clock cycle to indicate to the top-level FSM that it has completed and then returns to the START state and awaits another START_SIG. Figure 6.25 shows a state machine for this block.

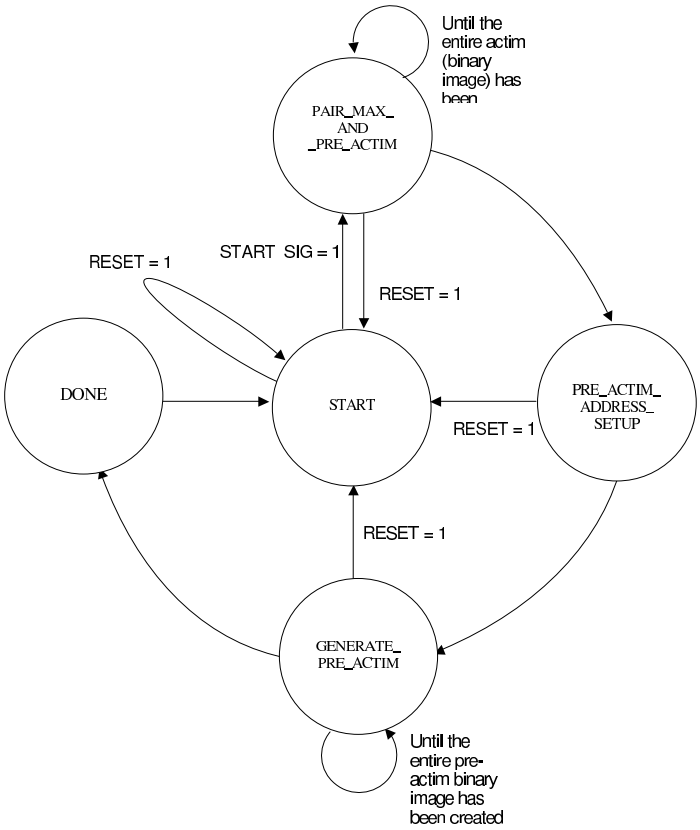


Figure 6.25: State Transition Diagram for state machine in the Binary Image Generator Block

6.9 DSP Interface

As was pointed out in Section 6.2, the *dsp_interface* was developed to convert the signals from the DSP used to communicate with an asynchronous SRAM to the format required by the synchronous *ram_interface*, *bram_inferface* and debug/control registers. Even though the number of input/ouput ports of the *dsp_interface* is high, as shown in Figure 6.26 and listed in Table 6.13, in no way does this reflect the level of complexity internal to the sub-block.

Table 6.13: Input and Output Signals for the DSP Interface Block

#	Signal Name	Type	Bit Width	Description
1	DSP_CLK	In	1	Clock (rising edge active)
2	DSP_ADDR	In	18	DSP address bus

3	DSP_DATA	In/Out	24	DSP data bus
4	DSP_BUS_CNTRL_R	In	1	DSP read enable line (active low)
5	DSP_BUS_CNTRL_W	In	1	DSP write enable line (active low)
6	DSP_EX_CS_A	In	1	DSP external chip select
7	MEM_CLK	Out	1	Clock to RAM interfaces
8	DSP_EX_ADDR	In	3	DSP extended address lines
9	UI0_ADDR	Out	18	Port 0 address bus
10	UI0_WRITE_DATA	Out	36	Port 0 write data bus
11	UI0_READ_DATA	In	36	Port 0 read data bus
12	UI0_RE	Out	1	Port 0 read enable (active low)
13	UI0_WR	Out	1	Port 0 write enable (active low)
14	UI1_ADDR	Out	18	Port 1 address bus
15	UI1_WRITE_DATA	Out	36	Port 1 write data bus
16	UI1_READ_DATA	In	36	Port 1 read data bus
17	UI1_RE	Out	1	Port 1 read enable (active low)
18	UI1_WR	Out	1	Port 1 write enable (active low)
19	UI2_ADDR	Out	18	Port 2 address bus
20	UI2_WRITE_DATA	Out	36	Port 2 write data bus
21	UI2_READ_DATA	In	36	Port 2 read data bus
22	UI2_RE	Out	1	Port 2 read enable (active low)
23	UI2_WR	Out	1	Port 2 write enable (active low)
24	UI3_ADDR	Out	18	Port 3 address bus
25	UI3_WRITE_DATA	Out	36	Port 3 write data bus
26	UI3_READ_DATA	In	36	Port 3 read data bus
27	UI3_RE	Out	1	Port 3 read enable (active low)
28	UI3_WR	Out	1	Port 3 write enable (active low)
29	UI4_ADDR	Out	18	Port 4 address bus
30	UI4_WRITE_DATA	Out	36	Port 4 write data bus
31	UI4_READ_DATA	In	36	Port 4 read data bus
32	UI4_RE	Out	1	Port 4 read enable (active low)
33	UI4_WR	Out	1	Port 4 write enable (active low)
34	UI5_ADDR	Out	18	Port 5 address bus
35	UI5_WRITE_DATA	Out	36	Port 5 write data bus
36	UI5_READ_DATA	In	36	Port 5 read data bus
37	UI5_RE	Out	1	Port 5 read enable (active low)
38	UI5_WR	Out	1	Port 5 write enable (active low)
39	UI6_ADDR	Out	18	Port 6 address bus
40	UI6_WRITE_DATA	Out	36	Port 6 write data bus
41	UI6_READ_DATA	In	36	Port 6 read data bus
42	UI6_RE	Out	1	Port 6 read enable (active low)
43	UI6_WR	Out	1	Port 6 write enable (active low)

44	UI7_ADDR	Out	18	Port 7 address bus
45	UI7_WRITE_DATA	Out	36	Port 7 write data bus
46	UI7_READ_DATA	In	36	Port 7 read data bus
47	UI7_RE	Out	1	Port 7 read enable (active low)
48	UI7_WR	Out	1	Port 7 write enable (active low)

In addition to performing this simple conversion, the *dsp_interface* serves a similar purpose to the DSP as the *ram_interfaces* do for the external ZBTs. That is, by using the DSP's extended address bits for control, the *dsp_interface* multiplexes the single port of the DSP between seven sub-blocks. Figure 6.26 shows to which the seven ports of the interface are connected. The following sections present further details about communicating with the devices connected to the *dsp_interface's* user ports.

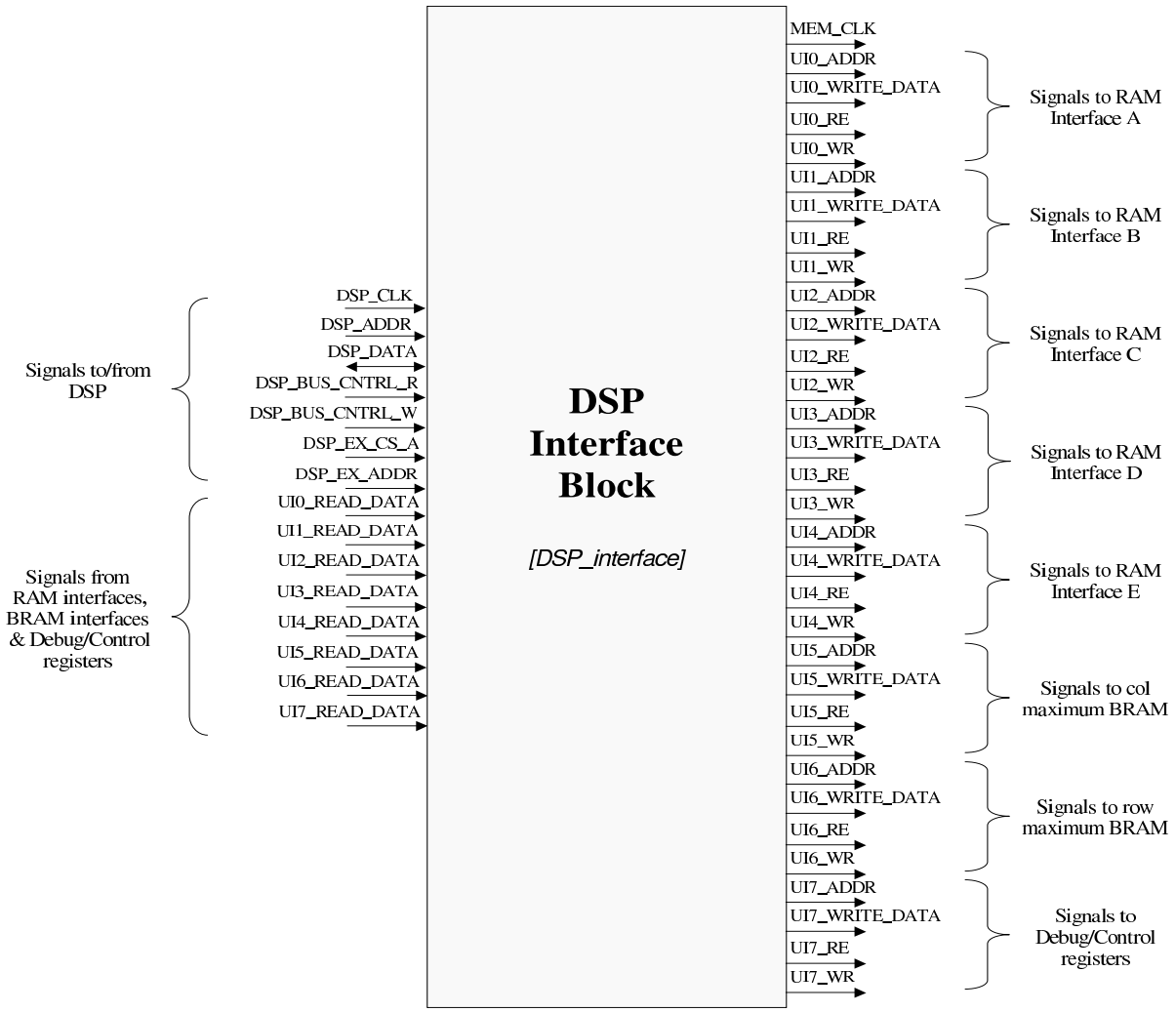


Figure 6.26: Block Diagram of DSP Interface Block

6.9.1 Accessing ZBT Memories and Internal Block RAMs

As mentioned above, the *dsp_interface* converts the signals of the DSP to a format compatible with the *ram_interface*. Accessing the ZBT memories from DSP is simply a matter of connecting the converted signals of the *dsp_interface* to the *ram_interface* of the desired memory. As can be seen in Figure 6.1, the *dsp_interface* is connected to all of the memory interfaces. However, as was discussed in Sections 6.2 and 6.3, the memory interfaces multiplex the physical ports to their associated memory cells between multiple user ports. Therefore, for the DSP to access a given memory its associated user port must be selected.

The top-level FSM controls the state of the select pins for all memory interfaces. When in either the START or DEBUG states, the top-level switches all the memory interfaces to their associated DSP user port. Therefore, when the top-level FSM is in either of these two states, the final task required to interconnect a given memory with the *dsp_interface*, thus permitting communication with it, is to page the *dsp_interace* appropriately.

6.9.2 Debug/Control Registers

The debug/control registers reside within the top-level design and consist of those listed in Table 6.14. As indicated by their given name, these registers were included for two purposes. First, they provided a mechanism by which the DSP could modify the functionality of the hardware during operation. Second, they allowed the ability to view any internal signal of the FPGA. Although the former of these two purposes was essential for proper functionality, the latter was in no way less important.

In fact, the inclusion of the debug registers significantly simplified the debugging process. After all, without their inclusion, the task of viewing internal signals would have been quite tedious. To view internal signals without the debugging registers, one would have to route them out to unused pins on a header and capture their values with a logic analyzer or oscilloscope. Although this technique is effective, in addition to being tedious and error prone, it limits the number of internal signals that can be viewed at a given time to the number of *spare* header pins that exist. However, it does benefit from the fact that the signals can be viewed at all times unlike when a debug register is

used.

Remembering that the debug/contol registers are accessed by the DSP with memory accesses, the closest one can get to constantly monitoring an internal signal would be by polling its value. Because the DSP has been configured to employ seven wait states for external memory access, the potential of missing a value of a quickly changing internal signal is high. In fact, if the signal changes any quicker than every eighth clock cycle, missing values is inevitable.

Therefore, the two above-described techniques for monitoring internal FPGA signals were used during the debug process. However, it is noted that the use of the debug registers was a far superior and simpler technique for extracting the internal signals listing in Table 6.14 while the top-level FSM was in the DEBUG state because there was no potential loss of values. After all, when the top-level FSM is in the DEBUG state, all the hardware sub-blocks are not operating and thus the listed signals are constant.

Table 6.14: Debug/Control Registers

Address	Register Name	Attributes	Usage	Description
0x00	Camera Selection	Read/Write	Control	Used to select the camera interface block
0x01	Debug Operation	Read/Write	Control	Used to select the operation of the top-level FSM
0x02	Next State	Read Only	Debug	Used to view the next state the top-level FSM will enter upon the assertion of a *done* signal
0x03	State	Read Only	Debug	Used to view the current state the top-level FSM
0x04	Thresholds	Read Only	Debug	A concatenation of *row_thresh* and *col_thresh*
0x05	Median1	Read Only	Debug	A concatenation of *median1* values of the columns and rows maximums
0x06	Median2	Read Only	Debug	A concatenation of *median2* values of the columns and rows maximums
0x07	Previous Image Address	Read Only	Debug	Start address of the previous image contained in ZBT A.
0x08	Current Image Address	Read Only	Debug	Start address of the current image contained in ZBT A.
0x09	Save Flag	Read/Write	Control	Used to inform the difference block to update the saved image

6.10 Memory Usage and Organization

Because of an early design decision, the original and saved image data are *packed* with two pixels per memory location, while all intermediate images (i.e. difference image, filtered image, *pre-actim*, binary image (*actim*)) are stored in the *typical* memory layout with one pixel per memory location. This discrepancy in memory storage requires the DSP to possess two interpretations of images stored in memory.

Figures 6.27 has been included to aid the reader in understanding the terms *packed* and *normal* . This figure depicts the mapping of an image in raster scan format to both a *packed* and *normal* memory layout. As pointed out in Section 6.5, even though the full 12-bits per pixel of the original image maintained, after the first computational block (*difference* block), the rest of the algorithm processes 8-bit data and thus every intermediate image has only 8-bit pixel values. The intermediate 8-bit pixels are stored in memory bits 7 down to 0. Also shown is this figure is the lack of use of memory bits 24 through 35. These bits have been left unused because it simplified the extraction of data from the ZBTs by the DSP which possesses only a 24-bit data bus.

As a final aid for the reader, a block diagram depicting the memory and addresses spanned by all the images and intermediate images has been included in Figure 6.28. This figure elaborates upon the memory usage first introduced in Figure 6.1. It also pictorially portrays the type of memory storage (*typical* or *packed*) used for each image while operating the image chipping algorithm.

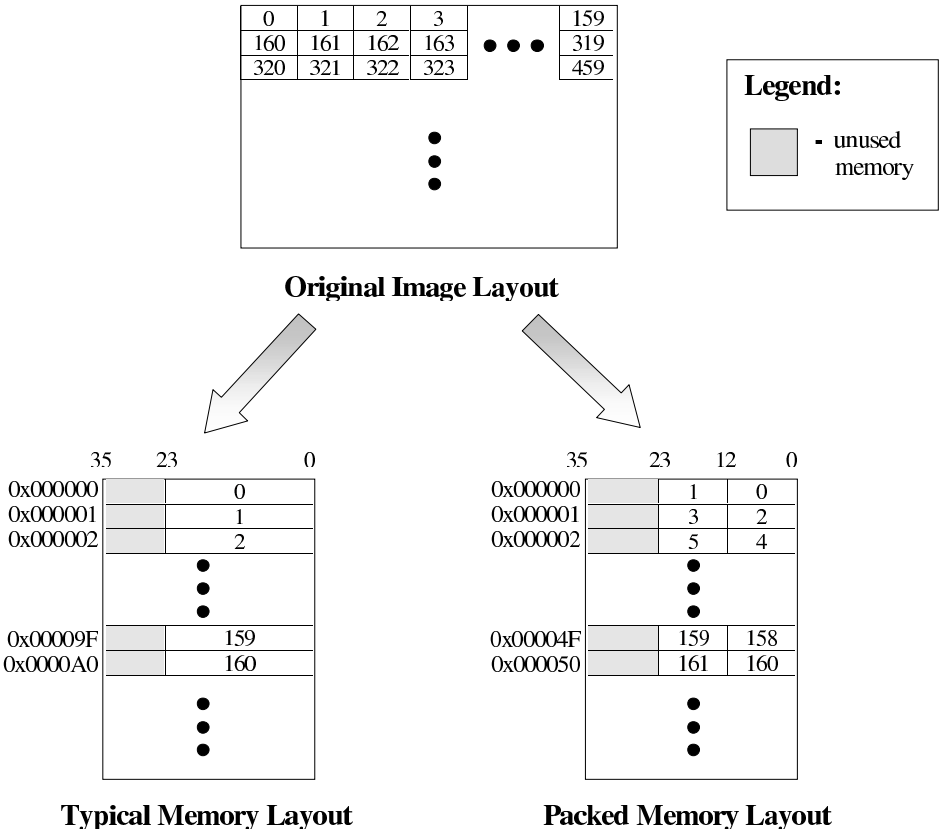


Figure 6.27: Block Diagram of Memory Layout for Packed and Normal Image Storage

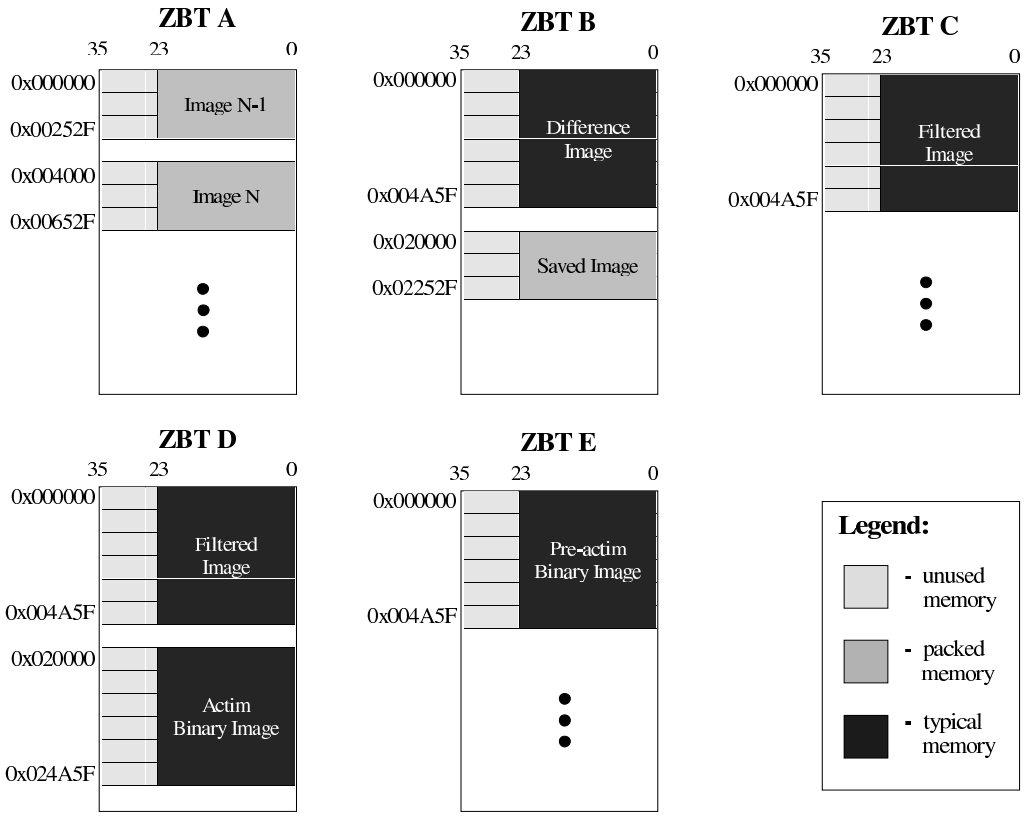


Figure 6.28: Block Diagram of Memory Usage for Image Chipping Algorithm

6.11 Summary

This chapter has presented a detailed discussion of the portion of the UMD algorithm that was implemented on the CAμS FPGA board. First, the top-level block and the operation of its internal FSM was presented. Then, each of the internal sub-blocks was explained with the assistance of a block diagram depicting its black box representation and a table that elaborated upon the ports displayed in the block diagram. Additionally, for each computational sub-block, a flow diagram portraying the associated computations performed by the sub-block in relation to the entire algorithm and a state transition diagram for its internal FSM were included to assist the reader. Finally, a memory mapping and its associated storage format was included with a brief discussion.

Chapter 7

Software Implementation

Utilizing the binary image generated by the hardware sub-blocks discussed in the previous chapter, the software running on the DSP performs the remaining operations specified by the UMD algorithm to determine the *best view* chip.[1] This chapter addresses the responsibilities of the software implementation sub-section of the image chipping algorithm. It is noted that the software implementation not only carried out the remaining operations following the generation of the binary image, but also served as the top-level controller for the implemented algorithm and handled the transmission of the *chips* produced by it. Figure 5.6 shows the relation of these three top-level operations assigned to the software to those performed by the other modules used to implement the algorithm. The following sections expand on the specific tasks accomplished by the software.

7.1 High-level Control

Even though the DSP was treated in many ways like a hardware sub-block by the top-level FSM for the image chipping algorithm, it differs from the others in that it possesses control over the FSM. That is, by manipulating an additional signal, the *reset* signal to the FSM, through its GPIO, the DSP has the ability to force the FSM to the START state. Referring back to Figures 6.2 and 6.3, the reader will be reminded that once in the START state, the top-level FSM will remain until the

[1] The *DSP software was developed in conjunction with Creed Jones.*

dsp_done signal is asserted. In other words, with these two signals, *reset* and *dsp_done*, the DSP can force the FSM to a known state in which it halts until otherwise indicated by the DSP.

It is this above-described ability to control the top-level FSM that permits the essential synchronization of the hardware and software implementations of the image chipping algorithm to occur. Once synchronized by the assertion and de-assertion of the *reset* signal, the normal operation of the software performing the image chipping algorithm is to issue a *dsp_done* signal to the top-level FSM and wait until the *dsp_start* signal is asserted indicating the hardware sub-blocks have completed their operations and have produced their associated outputs. Upon completion of the hardware sub-blocks, there exists an updated binary image in ZBT D that can be used by the DSP to carry out the remaining operations of the image chipping algorithm.

Considering that the image chipping algorithm is iterative, when the DSP finishes its associated operations to determine the *best look* chip, it will either re-issue the *dsp_done* signal to the top-level FSM if the *best look* has not yet been found, or begin the process of chip extraction and transmission if it has. With this above-described master-slave interaction between the software and hardware implementations and the DSP's usage of the results produced by the hardware, it is noted that the hardware implementation essentially serves as a coprocessor to the DSP.

In addition to serving as the top-level controller for the image-chipping algorithm, the DSP is also responsible for the configuration of the CAμS video board when it is being used to input an analog video source to the algorithm. This configuration process involves setting the internal registers of the video decoder chip (Bt835) of CHANNEL_1 such that it scales the image size to 160 columns $\times$ 119 rows. A high level function, *init_video()*, which is built upon the I^2C API described in Section 4.4.2, reliably implements this register programming task. The video board configuration process occurs prior to the synchronization of the implementations and the issuing of the first *dsp_done* signal when the input video source is NTSC, SECAM or PAL.

Run-time configuration of the hardware is the final control responsibility performed by the software implementation subsection. Run-time configuration refers to the DSP's ability to dynamically change the functionality of the hardware implementation during operation. The DSP employs the debug/control registers, which are discussed in the following section, to accomplish this dynamic modification of hardware functionality.

When operating the image chipping algorithm, only a subsection of these registers - *camera selection, debug operation, save flag* - are used. These select registers correspond to those which control the functionality of the hardware implementation. The *camera selection* register is set to the appropriate value such that the desired video interface is selected. The *debug operation* register is set to FALSE to force the top-level FSM to operate in *Normal Operation Mode*. Finally, the *save flag* register is dynamically adjusted during each iteration of the algorithm according to the results of the *best view* determination block.

The above-described top-level control of the image chipping algorithm is implemented in the *image_chipper()* function. This function accepts two arguments, *max_frame* and *camera*, which correspond to the maximum number of frames the algorithm will process before returning if no *best view* chip is found prior and the desired video interface to be used, respectively. Currently, the *camera* argument can take on two defined values, ALPHA_CAM and VIDEO_DECODER, which selects the *alpha_interface* and *decoder_interface* hardware sub-block to be connected to the *image_grabber* sub-block, respectively. Once the *image_chipper()* function resets the top-level FSM and initializes its internal variables (i.e. framenum) and the hardware, it enters a loop in which it remains until the *best view* chip is found or the maximum number for frames to be processed, *max_frame*, is exceeded. The top-level function performs the following steps during each iteration of the loop:

1. Signal hardware to produce a new binary image. {*assert_done(), dessert_done()*}
2. Wait for the hardware to complete. {check_start()}
3. Increment frame number.
4. Find connected components (*blobs*) of the binary image. {*ccp()*}
5. Merge *near* blobs. {*merge_blobs()*}
6. Reduce merged blobs to one. {*mult_frames()*}
7. Determine if the *best* chip has been found. {*choose_frame()*}
8. Set *save* flag if the chip selection function deems necessary.

The implementation of above-listed operations has been divided between the top-level *image_chipper()* function and its associated sub-routines. This division is shown in the list by placing the name of the corresponding subroutine in curly braces following the operations that were not performed by

the top-level function. The following sections discuss operations four through seven in more detail and present the particulars of their associated subroutines.

7.1.1 Accessing Debug/Control Registers, BRAMs and ZBTs

The debug/control registers are internal registers of the hardware implementation that can be read from and possibly written to by the DSP. Manipulation of the debug/control registers is accomplished in the following manner. First, one must page the *dsp_interface* such that the converted DSP signals are routed through to the user port (port 7) associated with the debug/control registers. This paging operation is accomplished through the extended address bits of the DSP address bus that are stored in the CPLD on the DSP board. Then, a read/write to a valid address of a debug/control register by the DSP to external memory will extract/change the contents of the associated register. Table 6.14 lists all the debug/control registers and there corresponding addresses.

Although only a small percentage of the debug/control registers are modifiable (only those associated with hardware control), all the registers are readable. It is this ability to view the contents of these registers, of which contain the values of important internal signals of the FPGA, that proved essential to debugging and verification of numerous hardware sub-blocks. In particular, the debug/control registers were crucial to correcting an initially malfunctioning *threshold* block because it stored all of its results within the FPGA. Hence, a significant number of these registers dedicated to strictly debugging are associated with the *threshold* block.

Finally, it is noted that accessing the ZBTs and block RAMs (BRAMs) from the DSP is accomplished with the same process used for the debug/control registers, but has the added requirement that the top-level FSM be in a state the permits accessibility of these devices from the DSP. The DEBUG and START states are the only ones that allow the DSP to communicate with the block RAMs and ZBTs on the FPGA board. After all, these are the only two states that set the control signals to the *ram_interface* and *bram_interface* blocks such that their associated devices are connected to their corresponding *dsp_interface* user port.

7.2 Region Labeling and Bounding

After a binary image has been produced by the hardware, in which foreground (non-zero) pixels represent areas of activity to be analyzed further, the first operation performed by the DSP is to determine which sets of foreground pixels are spatially connected. This operation is often referred to as region labeling or connected-component labeling[34]. This initial process executed by the DSP corresponds to the first of the two sub-blocks of the UMD image chipping algorithm that are shown in Figure 7.1. In actuality, the second of these sub-blocks, the determination of the bounding box for each of the connected components contained in the binary image, is produced as a direct result of the first in the software implementation. Even though this implementation did not take the same approach depicted in these blocks, the overall functionally of the software directly correlates with that of the shown blocks.

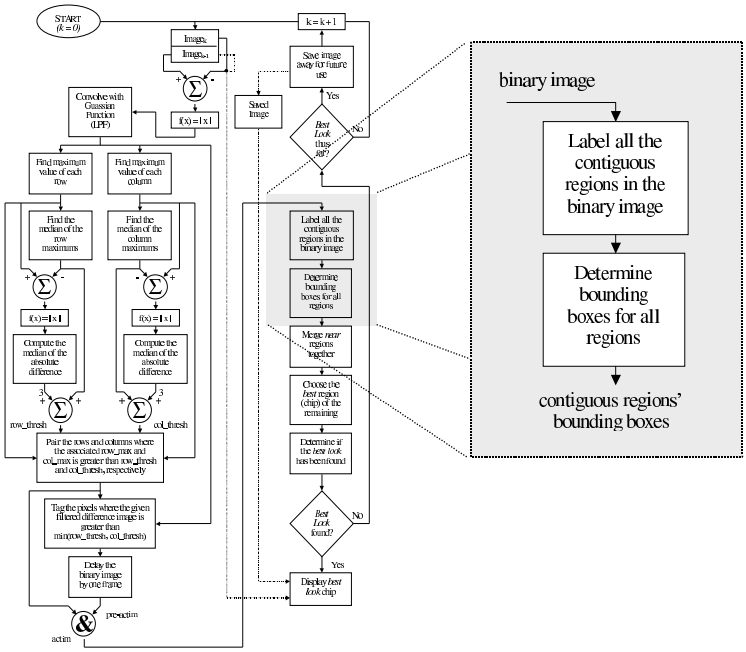


Figure 7.1: Flow Diagram of Region Labeling and Bounding

To determine the connected components of a binary image, there exist two major techniques, raster-oriented and contour-oriented[34]. The raster-oriented approach processes the pixels of the image in a row-column order, while the contour-oriented technique operates by tracing the boundaries of re-

gions. Due to the fact that processing speed and consistency were important to the implementation of the image chipping algorithm in order to maintain a high and constant frame rate, the former approach, the raster method, was taken because its computation time variance is influenced less by image density[34]. Pseudo-code for the algorithm used to determine the connected components, which is reproduced from [34], is shown in Figure 7.2.

```
Refer to the following labels to interpret the pseudo-code:
p is the pixel currently being examined; d is its neighbor to the left
in the same image row, while a, b and c are its neighbors on the
previous line.

/*      a b c   */
/*      d p     */

for each row in 1 to Rows-1
    for each pixel in 1 to Columns-1
        if foreground
            if (pixel c)
                if (pix d && blob[c] != blob[d])
                    combine blobs of c and d
                else if (pix a && blob[d]!=blob[a])
                    combine blobs of c and a
                else
                    add p to blob of c
            else if b
                add p to blob of b
            else if a
                add p to blob of a
            else if d
                add p to blob of d
            else
                new blob - add p
  END OF FIRST PASS

The second pass consists of assimilating all ‘‘blobs" that were found
to be linked, into a single region (for example, combining the two
branches of a letter ‘‘U" once the connection at the bottom is discovered).
```

Figure 7.2: Pseudo-code for Region Labeling Algorithm

The top-level function *ccp()* and its associated subroutines, *make_new_blob(), add_pix_to_blob(), mark_linked_blobs(), copy_blob(), combine_blobs()*, handle the task of finding the connected components and their associated bounding boxes. This function accepts one input argument, *p_img*, which is a pointer to start address of the binary image contained in ZBT D. It reads each pixel of

the binary image starting from *p_img* through the *dsp_interface* and the *ram_interace* of the designated ZBT RAM on the FPGA board and checks for connected components according to listed pseudo-code. This function determines the eight-connected regions of the binary image. That is, two pixels are considered connected if they are adjacent or are diagonal neighbors of one another.

Attributes of each found connected component, referred to as a *blob*, was stored in a global array of user-defined structure shown in Figure 7.3. These structures contained the dimensions of the associated bounding box of their blob, as well as, the number of foreground pixels contained in the blob. The final structure element, *linked_blob*, was used during the second pass described in the pseudo-code to determine which blobs generated during the first pass were linked, so that they could be assimilated to reflect the true connected components that exist in the binary image.

```
struct blob_data
{
    int area;           /* Number of pixels in a given blob                    */
    int xl;             /* Left edge of bounding box for given blob            */
    int xr;             /* Right edge of bounding box for given blob           */
    int yt;             /* Top edge of bounding box for given blob             */
    int yb;             /* Bottom edge of bounding box for given blob          */
    int linked_blob;    /* Number of another blob to which this one is linked  */
                        /* Set to itself if it is not linked to another blob   */
};
```

Figure 7.3: Structure Used to Hold Blob Properties

Upon exiting from *ccp()*, the global array of *blob_data* structures contains the attributes of the assimilated blobs contained in the binary and the number of found blobs is returned. This information assembled by *ccp()* is then employed by the region merging portion of the algorithm discussed in the next section.

7.3 Region Merging

Following the determination of the bounding boxes of the regions of interest (*blobs*) produced by the hardware implementation, the software implementation performs the region merging portion of the UMD image chipping algorithm shown in Figure 7.4. This merging operation combines the bounding boxes of *near* regions of interest together such that the resulting bounding box of one of

the *near* blobs reflects the bounding rectangle which contains the two *blobs* and the other has been zeroed. Whether or not two blobs are considered *near* is related to the distance in pixels that the bounding boxes of two given blobs, as was described in Section 5.2. The reader is referred back to this section for a more detailed discussion of this attribute.

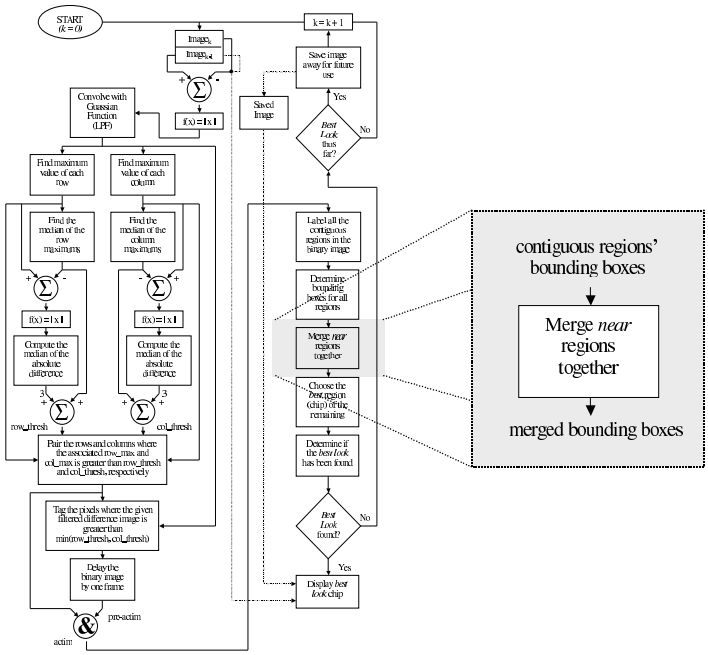


Figure 7.4: Flow Diagram of Region Merging

The top-level function *merge_blobs()* and its associated subroutine, *assimilate_one_blob()*, handle the region combination task. The attributes of the blobs to be merged are extracted from the global array of *blob_data* structures constructed by the *ccp()* function. The *merge_blobs()* function accepts one argument, *numblobs*, which indicates the number of valid blobs contained in this global array of structures. Two nested *for* loops that cycled from 0 to *(numblobs - 1)*, were used to pair the blobs such that all possible combinations were checked for being *near*. The resulting attributes of the merged regions are store back in same global array of *blob_data* structures as employed by the *ccp()* function and the number of remaining blobs after merging is passed as the return argument *merge_blobs()*. The next operation of the software implementing, region reduction, uses the results produced by *merge_blobs()*.

7.4 Region Reduction

Once the merging of blobs has been performed by the previous functional section of the software implementation, there may still exist multiple regions of interest. Considering the *best view* chip selection part of the UMD image chipping algorithm only operates on a single input blob, a further reduction in the number of blobs must occur if more than one remains. Of course, if only a single or no blobs exist, no reduction is necessary. This reduction of blobs is accomplished by the selection of the most appropriate blob of those that remain. The term "most appropriate" takes on two different meaning depending on whether or not a previous chip is being tracked by the chip selection portion of the software implementation. If so, then the most appropriate blob corresponds to the blob that has the closest center point to the tracked region. Selection based on this criterium attempts to ensure that the chosen region of activity (*blob*) corresponds to the same activity represented by the tracked chip. On the other hand, if no chip is being tracked, the blob with the largest sum of height and width is chosen. The operations of the region reduction subsection correspond to the sub-block of the UMD algorithm depicted in Figure 7.5.

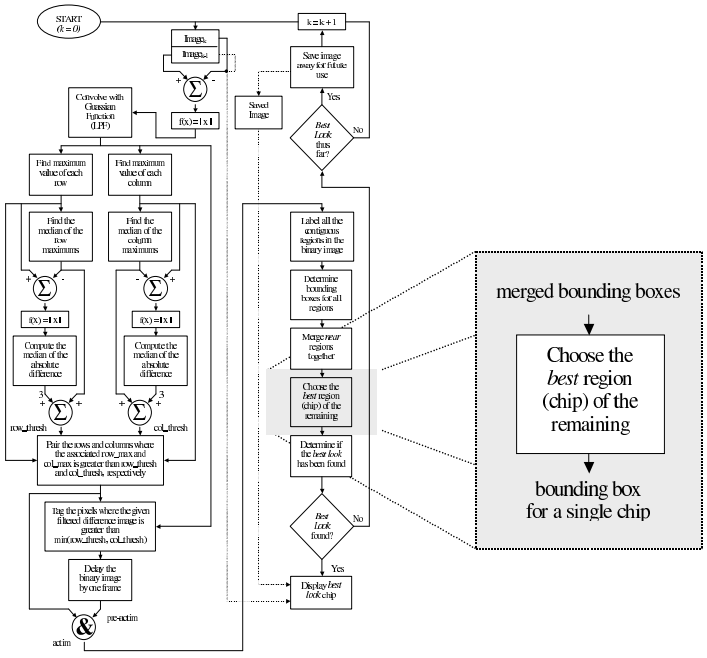


Figure 7.5: Flow Diagram of Region Reduction

The *mult_frames()* function that performs the above-described operations accepts one input argument, *numblobs*, and has no return argument. The input is used to determine the number of remaining valid blobs in the global array of *blob_data* structures following the *merge_blobs()* function. Once the most appropriate blob has been selected, the *update_chip()* subroutine is used to insert the contents of the chosen blob to a *chip* structure. Similar to the *blob_data* structure, the *chip* structure is user defined. In fact, these two structures share many common members as can be seen by comparing Figures 7.3 and 7.6.

```
struct chip
{
    int framenum;       /* Frame number associated with this chip       */
    int area;           /* Number of pixels in a given chip             */
    int xl;             /* Left edge of bounding box for given chip     */
    int xr;             /* Right edge of bounding box for given chip    */
    int yt;             /* Top edge of bounding box for given chip      */
    int yb;             /* Bottom edge of bounding box for given chip   */
    int width;          /* Width of bounding box for given chip         */
    int height;         /* Height of bounding box for given chip        */
};
```

Figure 7.6: Structure Used to Hold Chip Properties

Moreover, two of the members that they differ by, *width* and *height*, are generated directly from members in which they have in common (*width = xr - xl, height = yb - yt*). The final unaccounted for member, *framenum*, is assigned the frame number of the currently captured frame relative to the commencement of the image chipping algorithm. As a final point, it is noted that if number of merged blobs is zero, then contents of the *chip* structure are set for a zero-sized chip to pass this information along to the chip selection subsection.

7.5 Chip Selection

Even though it requires very little processing power, the chip selection operation of the UMD algorithm is the most complex portion. After all, the selection process is based upon a complex sequence of if-then-else rules that operate on the computed region features of the selected chip. Due to this complexity, this portion of the UMD algorithm was ported directly from the existing

Matlab functions to the C programming language and it implementation in hardware was avoided. The chip selection functionality is implemented by the *choose_frame()* function. From a top-level perspective, Figure 7.7 shows pictorially the subsection of the entire image chipping algorithm that *choose_frame()* performs. For further details pertaining to the exact criterium employed by the chip selection process, the readers is directed back to Chapter 5 where a thorough discussion of the selection process is presented.

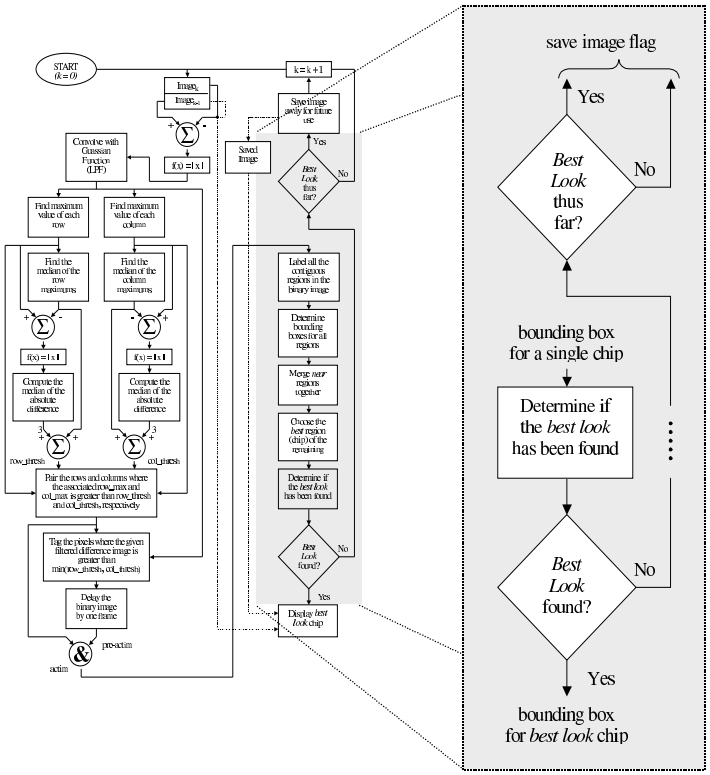


Figure 7.7: Flow Diagram of Chip Selection

It is this selection process that is essential to making the image chipping algorithm a low power solution. After all, it is this portion of the algorithm that permits the final step in the reduction of the transmission overhead incurred from sending full-bandwidth frame rate video, to sub-images of regions of interest at frame rate, to a single blob for a set of images to occur. The *choose_frame()* function uses five *chip* structures to determine the *best view* chip. One of these structures contains the particulars of the current *blob* that resulted from the *ccp(), merge_blobs()*, and *mult_frames()* routines. The others hold the information corresponding the previously and currently chosen *best*

look chip.

Upon exiting, the return argument of *choose_frame()* is set to TRUE if the *best view* chip was found or FALSE, otherwise. Regardless of the determination of *choose_frame()*, it always stores the information relating to the best seen chip thus far in a global variable to be used again by itself upon next invocation or by the calling *image_chipper()* function when either a chip has been found or the frame limit has been exceeded. This global variable is also used by the top-level function, *image_chipper()*, to control the *save* flag. Specifically, the *save* flag is set when the frame number of the best chip thus far matches the current frame number and is cleared otherwise. Therefore, the hardware will be indicated to replace the saved image with the current image when necessary. When the return argument of *choose_frame()* is TRUE or the frame limit has been reached, the software implementation extracts the chip from memory based upon the members of the *chip* structure and transmits it over the serial port as explained in the next section.

7.6 Chip Transmission

Considering the end goal of the image chipping algorithm was to select the *best view* chip from a video sequence to then be displayed or further processed at a ground station, a means by which to get the chipped image contents out of the sensor node had to be developed. Figure 7.8 shows the sections of the image chipping algorithm that correspond to this extraction and displaying. With previous algorithms implemented on the CAμS stack, a radio connected to the serial port of the DSP was successfully used to get the results of the algorithm off board and into a remote ground station. Building upon this previous success, a similar technique was used for the image chipping algorithm. Once again, the serial port of the DSP was used but an addition fork was added to the end of the development path in which one leg continued and the other terminated. This split in developmental efforts simplified the debugging and verification effort while at the same time provide a convenient point to return, which consisted of a near-complete implementation, once the design was checked to exhibit proper functionality.

This branching is shown in Figure 5.6. As can be seen with the connection between the DSP board and the Host PC blocks, a fork exists which either transmits the chipped images to a host computer

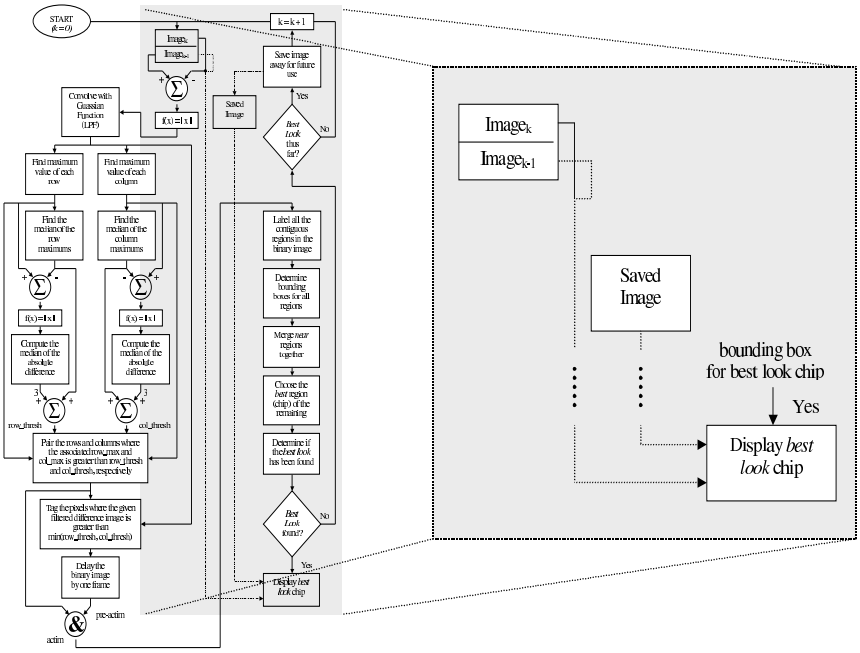


Figure 7.8: Flow Diagram of Chip Transmission

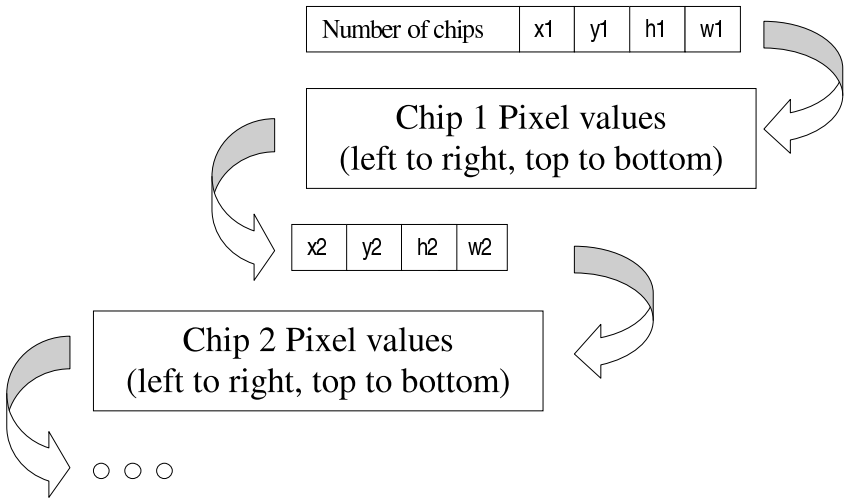
or through a radio back to a ground station. Even though a proprietary visualization suite was used for the software running on the ground station, it is noted that the two specified paths did not have to be completely disjoint. After all, the physical connection from the DSP board to the host computer, could have been replaced with a wireless connection and the same host software used during the debugging and verification process could have been employed for visualization at the ground station. Nevertheless, regardless of the end user software, a communication protocol had to be developed to transmit the chipped images from the DSP to the visualization package. The comminations protocol developed as part of the software implementation is explained in the following section.

7.6.1 Communication Protocol

Because the specifications to interact with the proprietary visualization suite were not yet developed, a custom protocol was constructed to transmit the pixels of the chipped image between the DSP and host processor for developmental purposes. Considering that this protocol was expected to

be temporary, it was intentionally made simple to expedite its implementation and to leave a great deal of flexibility for future protocols. An assumption that the communication channel was lossless was originally made because it afford significant simplification of the protocol, but would have to be removed when the connection became wireless. After all, it is the complicated handshaking and data validation (parity bits, checksums, CRC) used to make the like reliable that complicates a protocol.

The simple communications protocol used by the DSP to communicate with the *chip view* program is shown pictorially along with its affiliated pseudo-code in Figure 7.9. Considering that the serial port was known to have been extremely useful to the debugging of previously implemented algorithms by using it to transmit text messages that could be displayed on a host machine, a minor modification to the simple protocol described by the pseudo-code in Figure 7.9 was made because it did not permit this functionality. To incorporate debug message capabilities, the following exception was made to the protocol. It was decided that setting the number of chips contained within the image equal to 255 would be a flag to indicate that a debug message followed. The next byte after the 255 indicated the length of the debug message in bytes excluding the 255 and the length byte. This special case of the protocol is depicted in Figure 7.10.




```
Transmission Steps:

1.  Transmit the number of chips in the image sent. (1 byte)
2.  N = 0;
3.  Transmit the (x,y) location and height and width for the Nth chip in
    this order. (each 1 byte)
4.  Transmit the pixel values for the Nth chip in a raster-scan fashion,
    left to right, top to bottom. (2 bytes per pixel to hold the 12 bit
    data with the least significant byte first)
5.  N = N + 1;
6.  if (N == number of chips in this image)
        goto 1 to begin sending the next image when necessary;
    else
        goto 3;
```

Figure 7.9: Communications Protocol between the DSP and the Chip View Windows Application

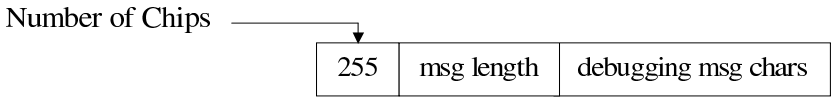


Figure 7.10: Communications Protocol for Embedded Debug Messages

To simplify the usage of this protocol, a small API was developed to act as an upper layer of abstraction that hid the details of the protocol. The API consists of the functions listed in Figure 7.11. The arguments to the *pprintf()* function are identical to the standard *printf()* function, while those for the other two functions are defined in the figure. This API was used to communicate with the *chip view* application discussed in the next chapter throughout the entire development process and surprisingly enough was only slightly modified to interact with the proprietary visualization suite.

```
Functions:

/* Printf type debug function (sends output to serial port) */
void pprintf(const char *format [, argument]... );

/* Send out chips with pixels packed one per memory address */
void sendChips(char num\_chips, char memory, int* image, char dimensions[][4]);

/* Send out chips with pixels packed two per memory address */
void sendPackedChips(char num\_chips, char memory, int* image, char dimensions[][4]);

Arguments:

*format   - format control string
argument  - optional arguments
num_chips - the number of chips in the image to be sent
memory    - the number of the memory in which the image is (0-A, 1-B, 2-C, 3-D,
            4-E)
*image    - pointer to the beginning of the image (NOTE: add 0x300,000 to the
            address in ZBT to cause the DSP to use external memory on the FPGA
            board)
dimensions[][4] - two-dimensional array of chip information.  The first dimension
            indicates the chip number, while the elements of the second dimension
            are the corresponding (x,y,h.w) for the associated chip.
```

Figure 7.11: Communication Protocol API

When the image chipping algorithm is operating, upon the selection of a chip, the original image data is read from memory and formatted for transmission to a host via the serial port. This operation is performed by the function *chip_out()*, which translates the information contained in *chip* structure filled by *choose_frame()* to the arguments accepted by sendPackedChips() and then invokes this subroutine passing the translated attributes. The subroutine, *sendPackedChips()* handles the actual communicates with the serial port, which directly communicated with the host computer at 115.2 kbps.

7.7 Summary

This chapter discussed the functional role that the DSP served as the top-level controller to the image chipping algorithm and its ability to access the debug/control registers, block RAMs and ZBT memories. Although only the ability to communicate with the control registers and ZBTs was necessary for the fully-functioning algorithm, it was explained how the power to view/change the debug registers, block RAMs and ZBTs proved essential to the debugging process. Then, a discussion each of the subsections of the UMD algorithm assigned to the software implementation and its associated function was presented. Finally, the chip transmission operation and the protocol employed for its execution was presented.

Chapter 8

Support Software

To assist in the verification and debugging of the image chipping algorithm's implementation, two indispensable software programs were developed. The first and more important of the two, was the *Chip View* program, an application that accepted the debug messages and image chips from the DSP via the serial port and displayed them in a graphical user interface (GUI). The second, a Matlab script that compared the results produced by the hardware and software implementations to those computed by the UMD algorithm, was used in the final stages of development to ensure complete consistency between the original algorithm and its implementation. A comprehensive discussion of these support tools is contained in the sections that follow.

8.1 *Chip View* Application

Chip View is a windowed application that was developed for the Windows NT operating system as a tool to be used strictly in the debugging and verification of the implementation of the UMD image chipping algorithm. Due to this intended use, the functionality of the originally simple application, which was only capable of graphically displaying chipped images received from the serial port, rapidly grew throughout the developmental and verification stages of the image chipping algorithm to supply all newly required capabilities. Because the motivations for each of the features of this application are considered to be self-evident, the following sections simply explain the incorporated

functionality in the final *Chip View* program and not the reasons for its existence.

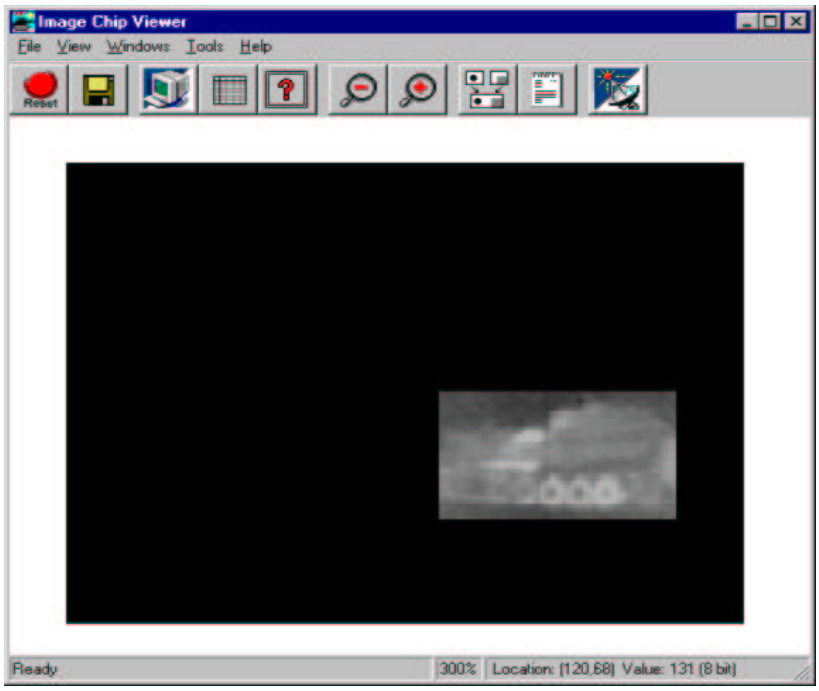
By default, this application accepts chipped images transmitted according to the protocol specified in Section 7.6.1 from COM Port 1, which is configured for 115 kbps, 8 bit data with 1 stop bit and no flow control or parity. However, the port and its associated configurations can be changed, if so desired. This mention of the ability to change the COM port settings only introduces the flexibility and convenience of use afford by the *Chip View* application that is further discussed in the following sections, which explain the program and its associated child windows in great detail.

8.1.1 Graphical View

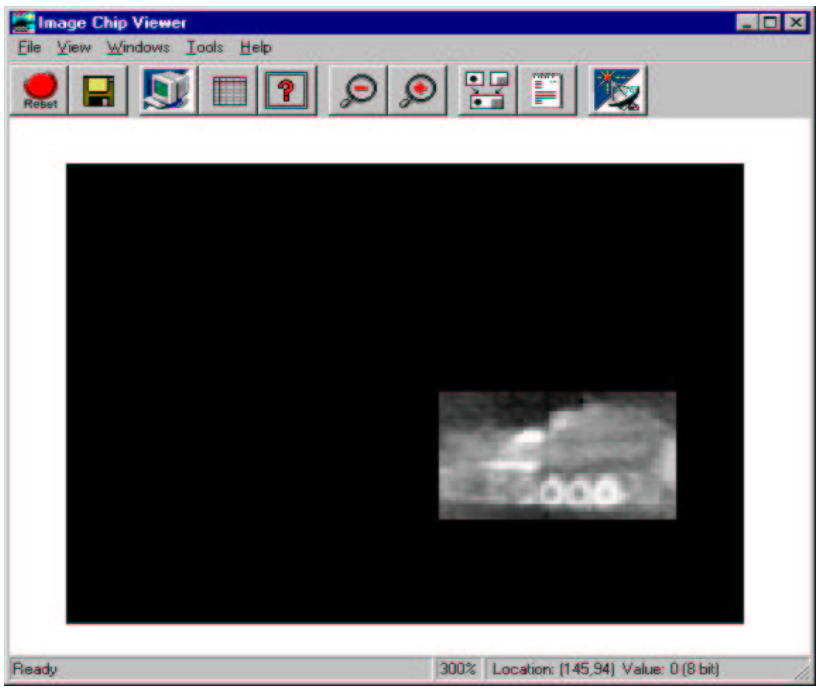
After a chipped image has been received on the serial port, it is graphically displayed in the main window of the *Chip View* application. Two screen shots of this main application window are shown in Figure 8.1. As can be seen in this figure, the program places the subsections of the images (*chips*) in their appropriate locations within the full image and sets the remaining pixels to black. Also shown in the figure is the ability to display an image with its true pixel values (normal view) or with the values normalized over the range 0 to 255.

The application also provides the user the ability to view the fine details of an image with its zooming capability. The zooming of an image is accomplish by simple pixel replication. Interpolation was not used during zooming because it simplified the implementation of this feature and was considered undesirable due to its introduction of new pixel values that were not present in the original data received from DSP. Considering that the *Chip View* tool was created to assist in the verification of the hardware and software implementations, it was considered senseless to present modified data from these components to the user and thus simple replication was employed.

The GUI of the *Chip View* program incorporated many of the features that are standard to most Windows applications. These features include a menu bar, toolbar and status bar. The toolbar, as well as, accelerator keyboard short-cuts were included to ease the selection of common menu items. The status bar not only provided a view into the current operation of the *Chip View* program, but also the percentage of magnification and the value and (x,y) location of the pixel pointed by the cursor in its second and third partitions, respectively. The GUI of the main application provides



Normal Graphical View of Chip



Normalized Graphical View of Chip

Figure 8.1: Screen-shots of Main Window of Chip View Application

access to all the features and windows discussed in the following sections.

8.1.2 Image Information Window

A child window of the *Chip View* application, the Image Information Window displays the attributes of the image currently portrayed in the main application window. As a stream of data is received from the serial port, the (x,y) location, height and width of each chip is extracted and stored. This information is used by the application to decipher the contents of the data stream, as well as to position the chips correctly within the image. These image characteristics, the number of chips and the number of the image relative to the last reset of the program, are displayed in the Image Information Window. Figure 8.2 shows a screen shot of the Image Information Window that depicts the above-described image details being displayed.

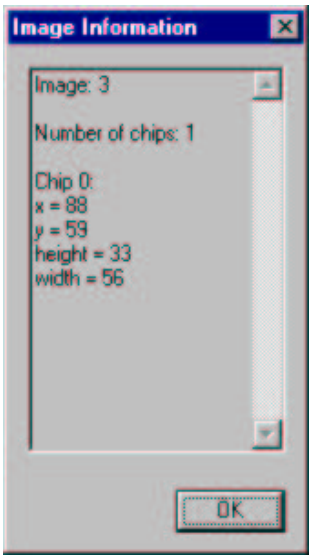


Figure 8.2: Screen-shot of Image Information Window

8.1.3 Pixel Viewer Window

Although one can position the cursor to determine the value of a single pixel, the Pixel Viewer Window permits him to view the values of a collection of pixels. Figure 8.3 contains a screen shot

of the Pixel Viewer Window, a child window of the *Chip View* application. As can be seen in this figure, the grouping of the pixels displayed in this window is determined by the selected chip, which is chosen with a drop-down list box. For the selected chip, the (x,y) location, height and width are shown in addition to the raw or normalized pixel values printed in hexadecimal notation depending on the mode of the application.

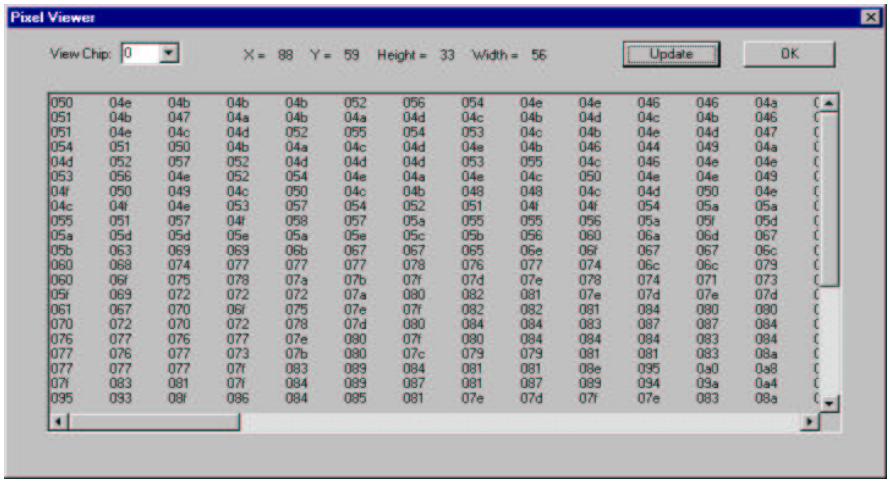


Figure 8.3: Screen-shot of Pixel Viewer Window

8.1.4 Output Window

As was discussed in Section 7.6.1, the protocol developed to communicate between the DSP and the *Chip View* application permits the ability to embed debug messages in the serial stream. The *Chip View* applications extracts these embedded messages and redirects them to a child window to be displayed. The child window responsible for this displaying of debug messages is the Output Window. Figure 8.4 shows a screen shot of this child window during a typical run of the image chipping algorithm. The ability to view these text messages proved to be extremely useful during debugging. They provided the capability to trace program flow and to examine the contents of internal variables and registers not normally viewable. All the messages received from the DSP are also logged to a file to be used/analyzed at a future time.

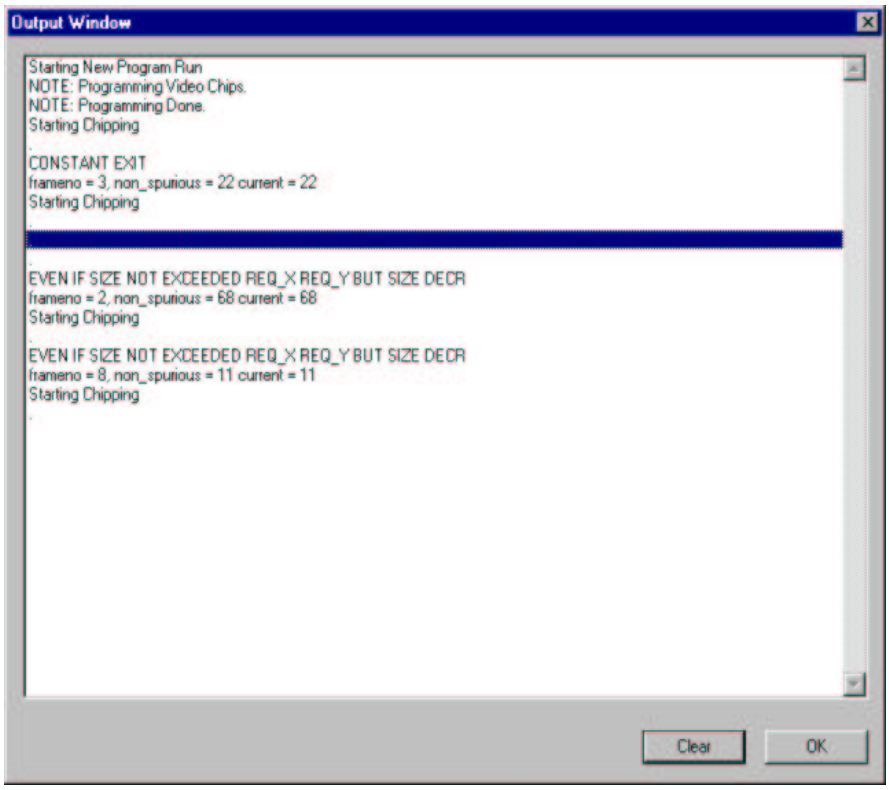


Figure 8.4: Screen-shot of Output Window

8.1.5 Saved Data

In order to permit the analysis of the captured chipped image data and its associated attributes at a later date or by another application, this data is written to disk by the *Chip View* application. For each chipped image received, three files are written to disk. The first is a text file whose contents are the information that is displayed in the Image Info Window. The second, also a text file, contains the values of each pixel of the capture image written one per line in a raster-scan format. Finally, a bitmap is written to allow the image contents to easily be graphically viewed.

As images are received, these three files are automatically written to disk. The files are entitled "img#.txt", "img#.dat" and "img#.bmp", which correspond to the image attributes, pixel values and graphical view of the image, respectively. The "#" in these file names is replaced by the number of the image relative to the last reset of the application. However, it is noted that at any

point in its operation the user can select the save option of the *Chip View* program. This option displays a common Save dialog box that allows the user to browse to specify the path and to entitle the three files as he pleases. This functionality permits the user to place the saved data in a location where it will not be overwritten or deleted and to give it a title that is more recognizable and useful than "img#".

8.1.6 Bitmap Concatenation

As was discussed in the previous section, for each chipped image received a corresponding bitmap file is written. The bitmap concatenation feature of the *Chip View* application allows the user to concatenate the images of these outputted files and to place the results of this operation into one large image that contains all them. This functionality allows the user to easily make a time-history of the chipped images or to check for proper alignment of the outputs of the computational sub-blocks.

The GUI for this concatenation feature is shown in Figure 8.5. As can be seen in this figure, the user can specify three parameters to the application to change the operation of the concatenation process. The first parameter specifies the number of images that should be included in the concatenated image, while the second indicates the starting offset. The application includes the images contained in "img{*start*}.bmp" to "img{*start+total-1*}" in the concatenated image, where *start* is the offset specified by the user and *total* is the number of images to be included. The final argument, the number of images per row, allows the user to adjust the shape of the outputted image, which is helpful in alignment debugging.

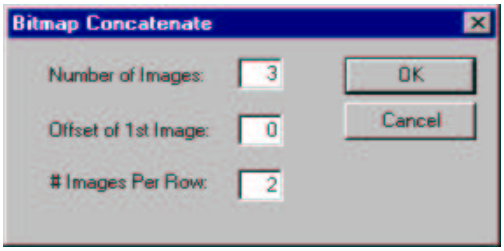


Figure 8.5: Screen-shot of BMP Concatenation Window

For example, consider the case in which one is trying to check two consecutive images for consistency in columns. He can specify that the concatenated image consists of one image per row. This specification will cause the concatenation process to place these two consecutive images one on top of the other, which allows the images to be checked for column alignment consistency by simply drawing vertical lines from one image to other. Such a test for row and column consistency is shown in Figure 8.6. In this figure, the difference of the first two images was taken to produce the third. The additional lines drawn horizontally and vertically from the edges of the white box contained in the first image show that indeed the difference block results are consistent for both the rows and columns.

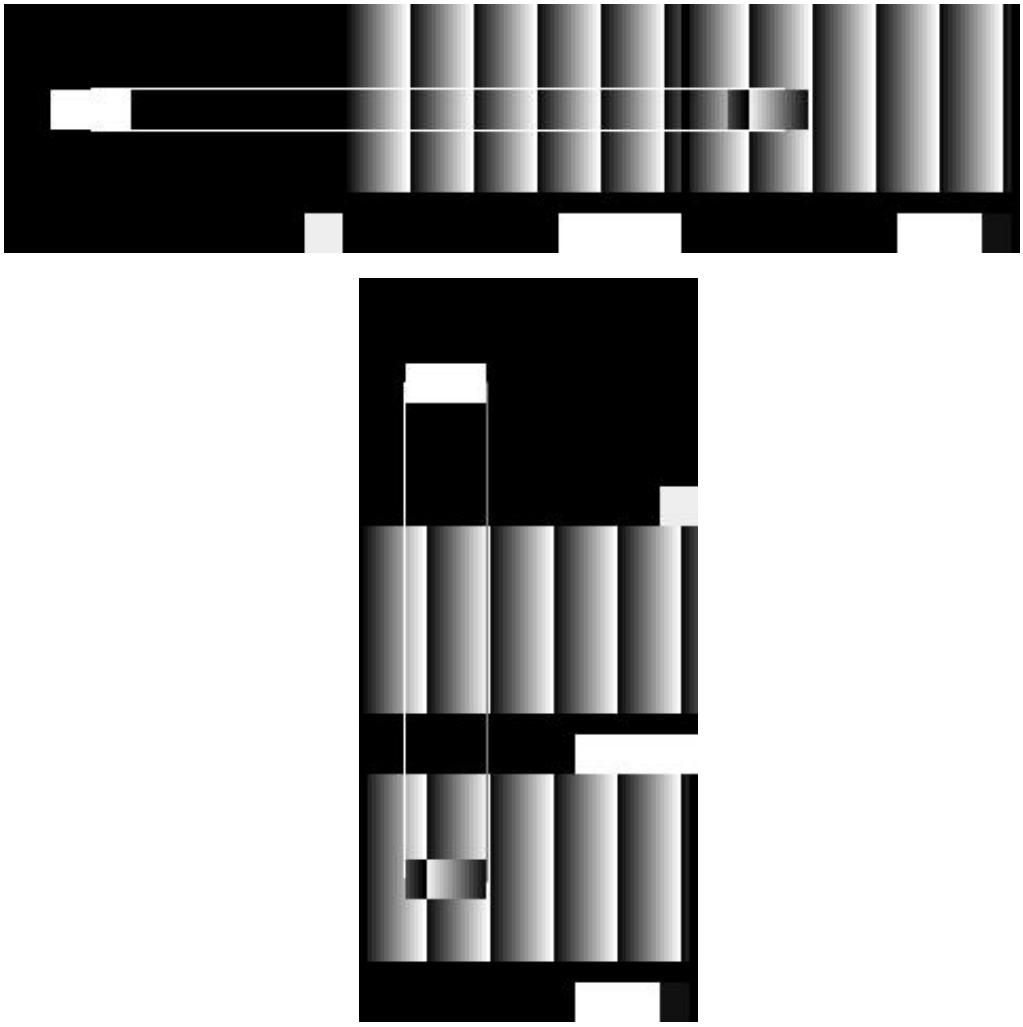


Figure 8.6: Row and Column Alignment Verification of Difference Block

8.2 Matlab Verification Script

The verification script written for Matlab is the second major support tool developed to assist in the debugging and verification of the implementation of the image chipping algorithm. This script, which was built upon capabilities provided by the *Chip View* program, checked for consistency between the algorithm and its associated implementation. That is, by programming the DSP to transmit the entire contents of the captured and intermediate images produced by the sub-blocks of the hardware implementation to the *Chip View* application, their contents were able to be written to disk where they were made accessible to the Matlab script.

Using these stored images, the Matlab script computed the operations specified by the UMD algorithm and compared the results of each step with its associated intermediate image produced by the hardware. This process of extracting the captured and resulting images produced by the hardware, permitted a direct comparison between hardware implementation and the original algorithm by providing the capability to force them to operate on the same data set. In fact, it was this script that was employed to conduct a definitive test to show complete consistency between the algorithm and its implementation. For a more detailed discussion of this validation process, the reader is directed to Section 9.1.

8.3 Summary

This chapter presented the two major support tools employed to validate the image chipping implementation. It was shown that the first of these tools, the *Chip View* application, was essential to the validation process due to its numerous capabilities. This application was dynamic. That is, modifications were made to it throughout the project to add/correct functionality whenever necessary. The statement that the *Chip View* application was essential to the verification of the algorithm's implementation is only reenforced by the fact that the second major support tool, the Matlab script, built upon its functionality. The Matlab script was developed to test for absolute correlation between the algorithm and its implementation. The next chapter presents a detailed discussion of the results obtained from the testing for this consistency conducted using this Matlab script.

Chapter 9

Results

Ideally, an implementation can be verified for proper functionality based upon its ability to satisfy quantitative metrics and required specifications. Unfortunately, few of these quantitative metrics and specifications existed for the implementation of the image chipping algorithm. Considering that the proper selection of the best view chip was deemed more important than the specific computations performed for its selection, the verification of the algorithm's implementation lent itself more to a qualitative analysis. This chapter addresses the means by which the implementation was verified both quantitatively and qualitatively. Then, a presentation of the modifications made to the original algorithm to improve its performance and to address the limitations inflicted by hardware is given. Finally, this chapter culminates with a discussion of the performance exhibited by the implemented algorithm and the resource utilization required by it.

9.1 Design Verification

Even though a specific list of design metrics would have simplified the verification process, no metrics were specified and none were known to be previously tested for this algorithm. Thus, a list was generated based on the aspects of the system which were expected to have the greatest affect on its overall functionality. In actuality, the number of metrics examined was limited and consisted of only the confirmation of the required frame rate and a subjective analysis of the implementation's

selection of the *best* view chip. Examples of other possible metrics that could have been used are the percentage of false-detections and percentage of missed detections on a given set of sequences. Yet, owing to the lack of previous testing of these metrics on the Matlab implementation and to the short life cycle of the project, the verification of the implementation consisted of qualitative analysis of the overall functionality, software checks of implemented operations and its ability to meet the few above-listed metrics.

It is noted that the verification process was resident in every step of the algorithm's implementation. That is, verification spanned from the reviewing of the initial flow diagram to the testing of the fully functioning algorithm in hardware. In fact, because the flow diagram was the basis for the entire implementation, several structured walk throughs were conducted to verify its validity. Once the flow diagram was deemed accurate and the algorithm had been partitioned, work commenced on both the software and hardware implementations. Upon there completion, initial testing was conducted. The software was tested on a personal computer with sample sets of known data and checked for the production of proper results. Owing to the fact that the C code was simply being confirmed and no timing or resources limitations were expected, execution on a general-purpose processor rather than the specific DSP on the CAμS stack was deemed acceptable for testing. The hardware implementation's sub-blocks, which were written in VHDL, were initially checked for proper functionality with VHDL simulators – Riviera by Aldec, Workview Office by Viewlogic Systems, Inc. and the Synopsys VHDL simulator. This is not to say that all the sub-blocks of the hardware implementation were simulated in all three of these packages, but rather that each block was simulated in the one which was preferred by its associated developer.

As proved to be the case with the *threshold* block, it must be remembered that simulators do not always accurately model the systems that they attempt to represent. After all, it must be remembered that all models are imperfect and oftentimes have associated assumptions that have been made to simplify their construction. It is the responsibility of the end-user to evaluate the affect these assumptions will have on the model's representation of his final system. In the case of the *threshold* block, it is believed that the lack of the inclusion of propagation delays, which could have been accomplished through back annotation, that caused the simulation of this block to exhibit proper functionality even though its hardware implementation did not.

The discovery of this improper operation of the *threshold* block was only made possible by the method in which the simulated blocks were introduced into the hardware. This process consisted of the insertion of a single hardware sub-block at a time. Owing to the design decision to separate the hardware implementation into sub-blocks, each of which operated independently of the others, verification of the blocks could be conducted in this one at a time fashion. The first block incorporated into the hardware was the all important frame capturing system. Once it was visually verified for proper functionality, which consisted of ensuring that the image was properly registered and that no blatantly obvious erroneous pixels were contained within it, the remaining sub-blocks were inserted one at a time ordered according to their sequence of operation in the flow diagram. By choosing this ordering scheme, one could make use of the previously verified sub-block's outputs to test the newly inserted sub-block's operation.

During the insertion process blocks were visually verified and checked for proper alignment. That is, by employing the *Chip View* application, intermediate images could be extracted and their contents could be displayed pictorially. At this point in the implementation process, these pictorial representations of computed intermediate results were observed for correspondence with expected results. For instance, consider the case where one moves his hand across the field of view of the camera. By extracting the two original images used to compute the difference image, as well as, the difference image itself, one is capable of determining if the computed difference image is representative of the true discrepancies that exist between the two original images. Because minute computational errors, such as single low-order bit errors, could not be perceived by this pictorial verification process, it was by no means an exhaustive test and nor was it intended to be. More thorough testing was left to be explored later in the design process and will be discussed later in this section.

As was above-mentioned, this pictorial verification process was also employed to check for proper alignment and expected outputs given forced inputs as was discussed in Section 8.1.6 and an example of which is depicted in Figure 8.5. All of the hardware sub-blocks with the exception of the *threshold* block, which did not produce an intermediate image, underwent this pictorial verification process. As for *threshold* block, an alterative means by which to verify it had to be developed. The inclusion of the debug/control registers and the ability to access the internal block RAMs, which contained

the *col_maxs* and *row_maxs*, afforded the required capability to verify this block and its internal sub-blocks.

The capability to extract the computed *row_max* and *col_max* values permitted debugging of the *threshold* block's internal sub-block, *maxcalc*, while the debug registers allowed the *mediancalc* to be checked. This accessibility to internal sub-blocks proved to be essential in the debugging of the *threshold* block because they too were sequential blocks. That is, like the *threshold* block itself, they made use of previous block's outputs. Specifically, the *mediancalc* block employed the outputs generated by *maxcalc*. Thus, initial verification of the *maxcalc* block simplified the debugging of the *mediancalc* block. Verification of the *threshold* block was performed by extracting the filtered image that this block uses to determine the dynamic thresholds. A software implementation of the threshold computation then employed this captured image to compute the dynamic thresholds. Finally, the results produced by the software and hardware were collected and checked for consistency. As was mentioned above, the initial hardware implementation of the *threshold* block was determined to exhibit improper functionality, which was eventually corrected by iterating back through the VHDL code development, simulation and hardware implementation verification cycle.

While all the hardware sub-blocks were being visually checked, in parallel the software was being initially verified with test data and was undergoing multiple comparisons with the Matlab code from which it was derived. Upon the completion of the initial verification of these sections, the entire system was composed and tested for top-level functionality. Early testing brought to light some previously undiscovered anomalies in the software implementation portion that affected the performance of the top-level system. However, once these errors were corrected, the system was subjectively determined to be a success. That is, the high-level functionality that the image chipping algorithm was developed to provide, the selection of the *best view* chip, was considered to be exhibited by its hardware implementation.

Figures A.1 through A.10 in Appendix A depict this functionality with their image sequences and associated hardware selected *best view* chips. With the exception of Figures A.1 through A.4, it is believed that the selected chips directly correspond to the subsections of these images sequences that would be inherently selected by a human who was asked to perform the same task assigned to the hardware. After all, with the approaching tank sequences, it can clearly be seen that the

algorithm appropriately withholds the selection of the best view chip until the tank has reached nearly its largest size in the sequence. Similarly, the implemented algorithm correctly waits for the full tank to enter the image beyond the prohibited boundary region and immediately selects the best chip so as to not lose any more image information in the retreating tank sequences.

Referring back to Figures A.1 and A.2, two of the several artificial sequences created to test the *best view* portion of the algorithm can be seen. These artificial sequences not only tested features of the best view selection part of the algorithm for which no IR footage was available, but also provided simple low noise sequences for which the expected best view chip was known. The reader may notice that the selected chips in Figures A.1 and A.2 do not correspond to the ones that he would most likely intuitively chose. After all, even in the decimated image sequences, there are clearly images that contain larger shots of these artificial tanks. Although at first these results were puzzling to the author as well, after further analysis it was determined that they were justifiable as explained below.

By inserting debug messages, it was determined that the best view was being selected because the choice of the same chip had been repeated for the last 15 frames. Yet, as was pointed out above, there certainly existed images that followed the selected chipped image that contain larger tanks and therefore should have replaced the selected chip. However, this explanation neglects the fact that the best view selection portion places and upper bound on the size of a chip. Therefore, considering that it is believed that the largest tanks in these sequences exceed this threshold, they would not be selected. This lack of selection will in turn cause the chip selected before this threshold crossing to be continually chosen.

By referring to Figures A.1 and A.2 it can be seen that the artificial tanks decrease in size slowly compared to their increase. It is believed that once the tanks cross back over the size thresholds they are selected because the chips associated with these images will be closer to the maximum allowable size. Then, owing to this slow decreasing in size, the best view portion will not recognize that the chip is shrinking because its aspect ratio to the currently selected chip will not exceed the preselected threshold. Moreover, considering that the decreasing tanks are by no means better choices, the algorithm will continue to select the first chip that crossed back over the size threshold. As a direct result of this continuous selection, the best view portion of the algorithm will exit.

Therefore, it has been shown that this threshold exceeding precisely explains the constant selection of the same chip for 15 frames.

The sequence in Figure A.2, an artificial sequence with Gaussian noise added to it, was developed to evaluate the algorithms ability to handle noisy images. Considering that the chip selected for this sequence nearly matches the one chosen for the sequence shown in Figure A.1, which is the same sequence without the added noise, the algorithm has exemplified it capability to continue to exhibit proper functionally in noisy environments.

As was pointed out above, the chipped out images of sequences contained in Figures A.3 and A.4 are also not inherently obvious to be the correct choice. However, unlike the sequences in Figures A.1 and A.2, where the correct choice was considered to be known yet was not selected, the sequences contained in Figures A.3 and A.4 are believed to not possess a single proper selection. Thus, it is claimed that the selection of any chip that encapsulates all the motion in the image is an acceptable result. Specifically, as long as the personnel in these images are not able to pass without detection, the result is considered a succuss. Referring to these two figures, it will be seen that indeed the hardware implementation of the algorithm correctly grouped the common regions of activity and successfully chose a chip for each sequences.

It is also noted, that the selected chips shown in the figures in Appendix A were also shown to be reproducible. That is to say, from a top-level subjective view, it was determined that the same or nearly the same chipped out image was selected from one run of the sequence to the next. This reproducibility is an important characteristic because it allows algorithmic modifications to be made and the expected results to be reflected in the fully-functioning system. For instance, if one increased the number of frames a decreasing chip is permitted to shrink before a selected best view chip is chosen, the chipped image would contain a smaller tank as would be expected. Without consistency in runs, one would not only be unable to easily make algorithmic changes, but more importantly should be concerned about the reliability of the system.

Even though the implemented algorithm's performance was satisfactory to both the implementors and the customer, one last verification testing procedure was performed. After all, as was stated above, with the exception of the *threshold* block none of the hardware sub-blocks and none of the software had been thoroughly tested. In fact, it is noted that the pictorial verification of the

pre_actim and *actim* binary images was far from complete, because the dynamic thresholds made their expected outcomes difficult to visualize. An exhaustive test of the entire implementation would be ideal, but this was not reasonable in the short time span allocated to the project. Therefore, a more thorough, but by no means exhaustive, testing scheme was developed. The central element of this verification scheme was the Matlab script introduced in Section 8.2, which built upon the functionally provided by the *Chip View* application.

As was described in Chapter 8, this script compared the extracted intermediate results of the hardware computations to the associated results produced in Matlab. Through the process described in Section 8.2, this test was capable of comparing the corresponding results of Matlab and software/hardware implementations both of which operate on the same data sets. For the sets of data run through this script it was found that the result produced by the Matlab and the algorithm's implementation matched identically. This script checked for consistency of the following operations – absolute difference, filtering, thresholding, binary images production, connected component and bounding box detection, and merging of the bounding boxes. Thus, the remaining sections of the algorithm not verified by this final test were the region reduction operation and best view portion of the algorithm. Both of these sections of code were once again reviewed in an attempt to provide as thorough of a verification of their operations as was performed for the remaining operational blocks.

9.2 Algorithm Modifications

The fact that few modifications were made to the algorithm during its translation from the original Matlab implementation to its hardware implementation on the CAμS 6.1 stack pays an incredible compliment to both its designers and implementors. The first modification made to the UMD image chipping algorithm was a reduction in the size of the 2D filter. In some sense this reduction was an unnecessary modification, because the initial size estimates that rendered this change necessary were later proved to be incorrect. Nevertheless, justification for the particular selected smaller filter came about by modifying the filter employed by the Matlab implementation and analyzing the affect of this change on the overall functionality of the system. Although the chosen filter

modification caused the algorithm to produce different results at the computational level, these discrepancies were overlooked because the overall functionality of the system remained the same. That is, even though the specific computed values and images varied, such as dynamic thresholds and even the selected best chip, the fact that the image from which resulting chip was extracted typically differed by only a frame or two from the original implementation, the filter modification was considered a viable solution to save FPGA resources. Moreover, it is noted that the size and location of the chips were also nearly identical to those of the original implementations.

The second modification made to the algorithm was the incorporation of an additional condition placed on the region merging portion of the algorithm. Rather than being dictated by and pre-implementation decision process as was the case with the filter, this change came about as a direct result of some of the initial chipped images received from the implemented algorithm. Although not exhibited by all chips, oftentimes chips with overly sized bounding boxes were detected. That is to say, these bounding boxes significantly exceeded the minimum boxes required to cover the associated movement in their corresponding images. After a small bit of investigation, it was determined that the cause of these undesirable results was due to the region merging operation. Specifically, if a small blob (perhaps only a single pixel in size) contained in the binary image resided within the *near* region of the particular chip of interest the two blobs would be merge to a single. Figure 9.1 depicts an extracted chip from the sequence shown in Figure A.3 that exhibits this above-described overly bound characteristic.

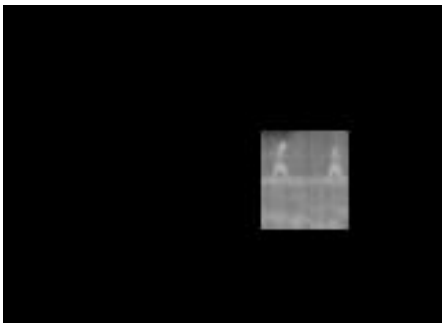


Figure 9.1: Overly Bounded Chip (Region of Interest)

Certainly, this merging is desirable if the two blobs correspond to either common regions of activity or to different portions of the same moving object as was described in Section 5.2. On the other

hand, if the small blob is due to erroneous pixel reads or other non-motion related reasons, this merging is undesirable because it can extends the height and width of the chip. Yet, it should be noted that this extension of the chip due to non-motion based blobs provides no additional information to the chipped image section. Moreover, additional radio bandwidth, which in turn translates to more power, is required to transmit this larger chip without the typical gain of beneficial imagery associated with a larger chip. To alleviate this presented problem, the additional condition specified that two blobs are merged only if both their areas exceed a minimum bound, MIN_BLOB_SIZE. The choice of this lower bound was made such that it eliminated the above-described undesirable merging results, but at the same time did not prohibit the beneficial merging effects discussed earlier.

It is noted that during the porting process, all the computations employed by the UMD image chipping algorithm were changed from double-precision float to integer arithmetic. This conversion simplified the conversion process to a hardware implementation. Even though this modification should have first been verified in Matlab by constructing a bit accurate model, time constraints did not permit this verification step. It is this lack of testing in addition to the inadequate documentation provided by UMD that complicated the algorithm's implementation. As will be shown by results presented earlier in this chapter, fortunately this unjustified modification proved to have no affect on the overall functionality provided by the algorithm.

The final modification made to the UMD image chipping algorithm during its porting to hardware was the unintentional introduction of a variable frame rate. In an attempt to provide the fastest possible frame rate, it was decided that immediately after an iteration of the algorithmic loop finished, the next one would commence following the next frame synchronization. Considering that the *threshold* block and the software implementations operating times are data dependent, one can imagine the situation where the small fluctuation introduced by this dependency could cause the termination of each algorithmic iteration to alternate on either side of next frame synchronization signal.

With that said and understood, the explanation of how this directly translates to a variable frame rate is trivial. Consider the case where computations terminate prior to a standard frame synchronization signal N from the one which the commenced that computation. Therefore, every N^{th}

frame is employed, which translates to a frame rate of 30/N frames/sec. On the other hand, if the computations extend too long such that the N^{th} frame synchronization signal is missed, the operation cannot commence again until the next one arrives. This results in (N+1) frame being used, which drops the frame rate to 30/(N+1) frames/sec. Although it is unsure the exact effect this variable frame rate characteristic has on the system, based on the favorable results discussed above, its effects appear not to be severe enough affect the system.

9.3 Performance and Resource Utilization

The resource utilization of the implementation of the UMD image chipping algorithm on the CAμS 6.1 stack was minimal. This claimed characteristic is quantified with the statistics that follow. The hardware implementation portion only employed 1,893 out of 12,288 total slices in the Virtex XCV1000 FPGA, which constitutes 15% this devices resources. Ten of the 32 Block RAMs (31%) internal to the device were used in the implementation of line buffers in the 2D filter and in the production of the dynamic thresholds. Similarly, only a small portion of the ZBT RAMs capacity was exploited. The storage of all the intermediate images and circular buffer required 1,447,040 bits ($160 \times 119 \times 8 \times 5 + 160 \times 119 \times 12 \times 3 = 761,600 + 685,440 = 1,447,040$ bits) of the 47,185,920 bits provided by the five banks (9,437,184 bits * 5), which translates to 3.06 %. Finally, even though the DSP is capable of operating at 100 MHz, it was limited 14.7456 MHz to permit proper functionality when accessing the ZBTs.

The reader is reminded that the target frame rate established by the analysis of the Matlab implementation was determined to be 5 frames/sec. Believing that an increase in this number would only improve the performance of the system, the hardware was developed to maximize this metric. Table 9.1 displays the resulting frame rate achieved by the hardware implementation. As can be seen in this table, the frame rate is dependent on the selected video stream interface. Nevertheless, regardless of the chosen interface, the hardware implementation exceeds the target frame rate.

The remaining columns of Table 9.1 have been included to inform the reader to which portions of the algorithm the most computation time is dedicated. As can be seen in this table, the computation time required by the DSP is double that consumed by FPGA processing with the *decoder_interface*,

Table 9.1: Computation Times and Associated Frame Rates for Alpha and Decoder Interfaces

Selected Interface	Average DSP Processing Time	Average FPGA Processing Time	Average FPGA Processing Time After Frame Capture	Average Total Processing Time	Average Frame Rate
decoder_interface	66 ms	33 ms	7 ms	99 ms	10 frames/sec
alpha_interface	66 ms	70 ms	9 ms	136 ms	7.35 frames/sec

while the FPGA computation time exceeds it with the *alpha_interface*. Owing to the fact that portions of the hardware and software implementations are data dependent, all the values in the table are representative of their average values.[1]

Remembering that both the *decoder_interface* and *alpha_interface* interact with the *image_grabber* block, which accepts a fixed sized image (160×119), one may question why this significant discrepancy in processing time exists. After all, as can be seen in the table, the average DSP processing time independent of this change as it to be expected. To explain this discrepancy one additional table column has been included, the Average FPGA Processing Time After Frame Capture. This column represents the time required to perform all the operations in the hardware starting at the absolute differencing up to and including the binary image generation. Therefore, this time represents the core processing time of the hardware implementation. A comparison of the times in this column associated with the interfaces still reveals a discrepancy between the two; however, this difference is not nearly as significant as the one present in the Average FPGA Processing Time column.

In actuality, this inconsistency is not related to the image size, but rather to the input clock frequency. Considering that the pixel clock for the digital output of the Alpha Camera by Indigo is 12.27 MHz while the BT835 Video Decoder produces a pixel clock of 14.318 MHz, it is only to be expected that the average processing time for the same sized data is larger for the *alpha_interface*. Though this pixel clock difference does not directly explain the other results, it is the direct cause of them. That is, the additional time required for the FPGA to process with the slower clock rate of

[1]It is noted that a loose interpretation of the mathematical term "average" should be used when interpreting these results. The results included in this Table 9.1 were collected by measuring these computation times with an oscilloscope. The "average" value represents the most often seen value during this collection process.

the Alpha camera causes the frame synchronization that is captured by the *decoder_interface* to be missed by the *alpha_interface*. As was explained in Section 9.2, this missed frame synchronization signal directly correlates to a reduction in frame rate. Remembering that the frame rate of both video streams is approximately 30 frames/sec, frame synchronization signals occur about every 33 ms.

Referring back to the Table 9.1 it can now be explained why the average FPGA processing time for the *alpha_interface* is so large when compared to that of the *decoder_interface*. If one subtracts the additional time required for awaiting the next frame synchronization signal, 33 ms, from the average FPGA processing time for the *alpha_interface*, the result will be 37 ms (70 - 33 = 37 ms). Taking into consideration that the remaining hardware sub-blocks following the frame capture take 9 ms for the *alpha_inferface*and 7 ms for the *decoder_interface*, the frame capturing time relative to a frame synchronization signal only differ by 2 ms ((37 - 9) - (33-7) = 28 - 26 = 2 ms). This difference in frame capturing time is believed to be caused by laboratory error in the measurements of these computation times and due to the differences in the CCIR (ITU-R) 601 4:2:2 video standard and the proprietary digital video stream of Indigo Systems. Nonetheless, this explantation was solely presented to justify the large discrepancy between the average FPGA processing time of the two interfaces, which has been shown to be direct result of a missed frame synchronization signal.

Figure 9.2 has been included as a final aid to the reader. This figure graphically depicts the distribution of processing time that was presented in Table 9.1. As can be clearly seen by the two plots presented in this figure, the percentage of true processing time, where true processing time is defined to be the time allocated to all the operations of the algorithm excluding the image capturing operation, is significantly reduced by the above-described frame synchronization miss. Specifically, this percentage is reduced from 74% (67% + 7%) with the *decoder_interface* to 55% (7% + 48%) when the *alpha_interface*. These percentages illuminate the fact that what can at first seemingly appear to be an insignificant discrepancy, can surprisingly have a significant effect on the performance of the implemented system. This unanticipated result goes to show the care and the thoroughness that one has to impart to succeed at the arduous task of implementing an algorithm in hardware.

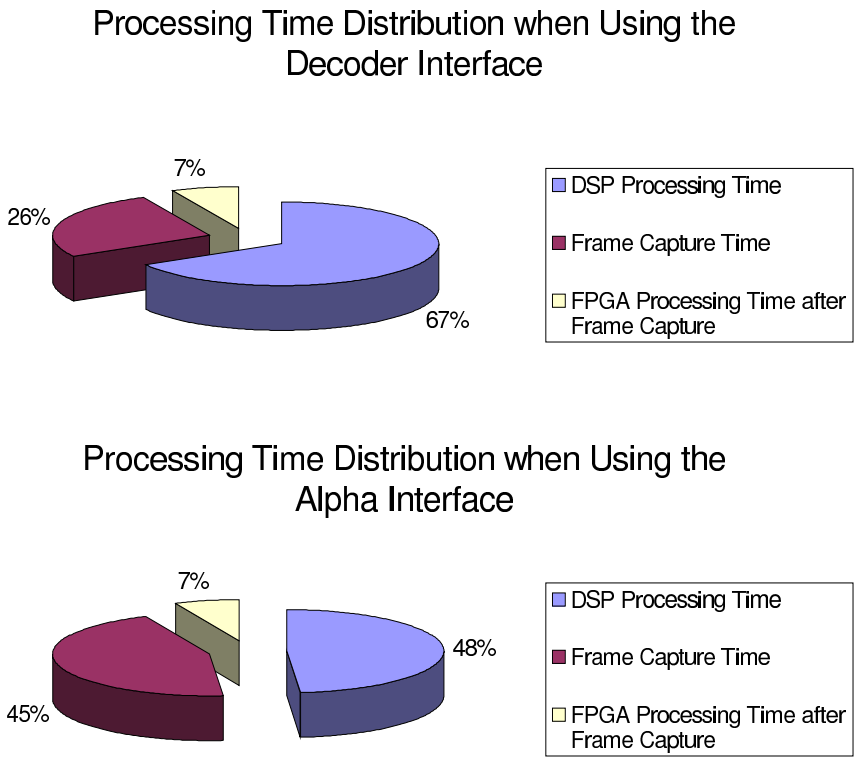


Figure 9.2: Processing Time Distribution of the Image Chipping Algorithm's Implementation for both Interfaces

Table 9.1 also demonstrates the significant improvement in processing time that was achieved by implementing the operations up to and including the binary image generation stage in hardware. The reader is reminded that initial software simulations estimated the required processing time for these operations to be executed on the Motorola 56307 DSP at 30 MHz to be approximately 3.33 seconds when optimizations were enabled. From this table, it can be seen that the FPGA was able to process these operations in approximately 7 ms, which constitutes a marked improvement. However, it should be noted that owning to the fact that the filter size was reduced in its hardware implementation, this is not an entirely fair comparison.

To address this concern, it is noted that some further software simulations were conducted for smaller kernel sizes of the filter and they too supported this claimed significant processing time reduction. These software simulations predicted the execution time for these computations, which specially employed filters of sizes 7×7 and 3×3, to be 1.633 and 0.3 seconds, respectively, operating at 30 MHz with optimizations enabled. Considering that even with the 3×3 filter, which is only one-ninth the kernel size used in the hardware implementation, there exists more than an order in magnitude reduction in the computation time, it is apparent that a significant reduction was achieved with the hardware implementation. Moreover, it is believed that by employing the unused resources and the inherent parallelism of the FPGA, a larger kernel filter could be implemented that would posses nearly and identical processing time. This larger implementation would only go to show this noted improvement further.

The pie charts contained in the Figure 9.2 also depict the beneficial processing times achieved when operations were implemented in hardware. As can be seen in both of the charts of this figure, the time consumed by the DSP is approximately one-half to two-thirds of the total processing time depending on the selected interface. Referring back to Figures 5.1 and 5.6, the reader will be reminded that the DSP is only responsible for roughly half or fewer of the operations required by the image chipping algorithm. As can be seen in these pie charts, the remaining operations, which were assigned to the hardware implementation portion of the algorithm, constitute only 7% of the total processing time.

Thus, it is concluded that marked improvements in processing speeds can be accomplish with hardware implementations. Yet, it is noted that this characteristic by no means renders software

implementations useless. As was discussed earlier, software implementations are ideal for control and bookkeeping tasks. Moreover, the ease associated with their generation compared to their counterparts in hardware, shortens their developmental life cycle. This shortened and simpler implementation lends itself to a iterative developmental process, which is ideal for high-level control tasks and responsibilities that often change throughout the life of a project. Hence, both software and hardware implementations posses their own benefits and should be used accordingly such that these benefits are exploited. These presented results prove that the division of labor developed for the implementation of the UMD image chipping algorithm successfully took advantage of these associated benefits.

Finally, considering that the CAμS stack is claimed to be a low-power microsensor solution, Table 9.2 has been included. This table presents the average power consumed by the fully-functioning algorithm for both interfaces. As can be seen by the results, the average power consumption is significantly larger with the *decoder_interface*. This additional power is mainly due to the power dissipated by the video board. The higher power consumption of the FPGA board, due to the increase in pixel clock frequency, accounts for the remaining increase. Neglecting this slight increase introduced by clock frequency changes, Figure 9.3 depicts the percentage of the total power that is employed by the original hardware and video board. As can be seen in this figure, the video board is responsible for nearly three-quarters of the power dissipated. From this plot, it is concluded that even though the video board provides important added capabilities to the CAμS stack, it should only be employed when absolutely necessary because of its high power consumption.

Table 9.2: Average Power Consumption

Selected Interface	Average Operating Power
decoder_interface	4800 mW
alpha_interface	1200 mW

Perhaps, a path similar to the one taken in the design process of this thesis' implementation will become the accepted standard. That is, the initial use of the video board for testing and debugging purposes and then its ultimate elimination once the system has been proven sound. Perhaps, with future advances in low-power video chips, this elimination will not be necessary with next

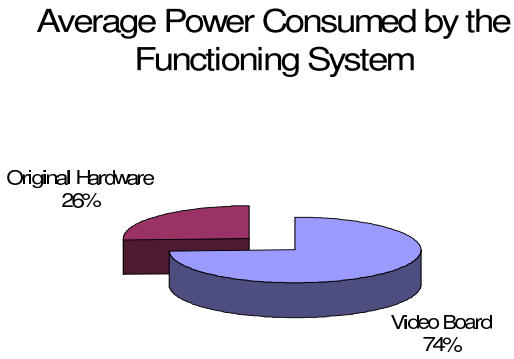


Figure 9.3: Percentage of Total Power

generation video boards. Nevertheless, it is noted that the current video board has met its required specifications and is the only means by which analog video sources can be processed. Thus, if a final design employs an analog video source, it inclusion is indispensable. Moreover, the video board can be used to accept NTSC video streams from a VCR and to display results on a television. The capability to accept video streams from a VCR permits the injection of reproducible data to the system, which is essential to the debug process. All in all, the CAμS video board is considered a success and provides noteworthy additional functionality to the original CAμS architecture.

As a final note, by using a power estimator spreadsheet provided by Xilinx, it has been determined that the FPGA consumes approximately half of the 1200 mW when using the *alpha_interface*. Not taking into the consideration the inefficiencies of the power supply board, the remaining power consumption is a attributed mainly to the DSP and memories. Remembering that even with a 3×3 filter used in the software implementation, which is only one-ninth the kernel size used in the hardware implementation, the FPGA implementation still achieves more than an order in magnitude reduction in the computation time for its associated operations than the DSP. Considering that the power consumption of the DSP is not an order of magnitude smaller than that of the FPGA, the chosen implementation proves to be a lower power design than strictly a processor-based solution.

9.4 Summary

This chapter presented a discussion of the means by which the implemented algorithm was verified for proper functionality. As was explained, verification was an ongoing process throughout the developmental stages that ultimately culminated with the development of a Matlab script. This script provided a course by which the hardware implementation could be directly compared to the original Matlab implementing. It was shown that these implementations were consistent for the data sets tested. Multiple sequences and their associated best view chips were shown to exemplify the implementation's proper functionality. A brief discussion of the modifications to the algorithm for its hardware implementation was included. Finally, the resource utilization and system performance were examined.

Chapter 10

Summary and Conclusions

This chapter presents an overview of the efforts undertaken during this thesis effort. It discusses the design, implementation, and verification of the UMD image chipping algorithm's hardware implementation on the CAμS stack by outlining the chapters of this document. Then, some conclusions and observations drawn from these chapters are introduced. Finally, this chapter culminates with a discussion of possible future work to enhance the developed implementation.

10.1 Summary

Chapter 1 introduced The Vision of the United States Army and how this vision has brought about an interest in microsensors. Microsensors were defined to be platforms that typically consist of one or more sensors, signal conditioning and processing subsystems, a radio link and a power source. It was noted that a major limitation of these platforms is their finite energy source. In attempt to reduce power consumption and hence extend the system's lifespan, this chapter explained how this thesis effort joined the previous work put forth in the development of the UMD image chipping algorithm and the CAμS microsensor stack.

Chapter 2 presented a brief overview of microsensors and the challenges faced by their developers. To assist in this discussion other existing microsensor platforms were referenced in addition to the CAμS architecture. This presentation gave the reader a perspective of how the efforts placed forth

by this thesis relate to other microsensor work currently being performed.

Chapter 3 continued the discussion of the CAμS microsensor platform that was begun in Chapter 1. A brief top-level description of each of the original boards of this microsensor platform was presented. This chapter concluded with a discussion of a limitation of the original hardware. It was shown that this limitation served as the motivation for the design and construction of the CAμS video board.

The discussion of the CAμS video board is continued in Chapter 4. An explanation of the requirements placed on the design and the decisions made to meet them was given. A top-level discussion of the architecture was put forth, which motivated a presentation of the capabilities of each of these components and the reasons for their inclusion. Considering that interfacing to the existing hardware was essential, a discussion of how this interfacing was accomplished was also presented. Finally, the API developed to ease the usage of video board by the end user was described.

In Chapter 5, the explanation of top the top-level functionality provided by the UMD image chipping algorithm was continued from Chapter 1. The algorithm's specific computations preformed to achieve this functionality were discussed. Finally, a presentation of how and why the UMD algorithm was partitioned for its implementation on the CAμS 6.1 microsensor stack was given.

Chapters 6 and 7 gave detailed discussions of the hardware and software implementations, respectively. The separation of the UMD image chipping operations among these sub-implementations abided to the division of labor called out at the end of Chapter 5. For both sub-implementations, each of its internal sub-blocks were thoroughly discussed and their operations performed were related back to the top-level algorithm discussion presented in Chapter 5. Chapter 7 also discussed the functional role that the DSP served as the top-level controller of the implemented image chipping algorithm. This control included being in command of the top-level FSM presented in Chapter 6.

In Chapter 8, two essential tools employed in the debugging and verification of the implemented algorithm were introduced. The first tool discussed was a Windows NT based application entitled *Chip View*. It was shown that this tool permitted the ability to display raw image data received from the CAμS stack pictorially on a personal computer in addition to many other desirable features. The second tool presented was a Matlab script that built upon the capabilities provided by the

Chip View application. Finally, it was pointed out that this Matlab script proved to be essential to verification process discussed in Chapter 9.

Chapter 9 addressed the means by which the implementation was verified both quantitatively and qualitatively. A presentation of the modifications made to the original algorithm to improve its performance and to address the limitations inflicted by hardware was given. Verification was explained to be an ongoing process throughout the developmental stages that ultimately culminated with the development of a Matlab script introduced in Chapter 8. This script provided a means by which the hardware implementation could be directly compared to the original Matlab implementation. From the results obtained by using the Matlab script, it was shown that these implementations were consistent for the data sets tested. Multiple sequences and their associated best view chips were included to exemplify the implementation's proper functionality when viewed subjectively. Finally, the resource utilization and system performance were examined.

This thesis presented a hardware implementation of an image chipping algorithm. The primary purpose for this implementation was to reduce power consumption while at the same time provide quality image data. By the mere fact that the implemented algorithm only transmitted fragments of the video stream, the former of these requirements was most assuredly achieved. After all, it was this exact reduction that was the motivation for the algorithm's original development. The latter of the requirements, providing quality image data, although quite vague, has also been met by the implementation. Justification for this claim resides in the numerous image sequences and associated chipped images that have been presented. Owing to its 7.35 to 10 frames/sec operating rate, this implementation has more than satisfied its real-time requirements.

10.2 Future Work

Although the implementation presented in this thesis met all its requirements, it is the belief of the author that modifications could be made to further enhance the current implementation. Three areas in particular where it is considered that improvements can be achieved are power, performance and integration. The following sections address the associated modifications that can be made to afford improvement in these sections.

10.2.1 Power Reduction

Considering that power reduction served as the fundamental motivation for the efforts undertaken by this thesis, this section presents other means by which additional power savings can be achieved with the current implementation. Remembering that the current implementation only employed 3.06 % of the ZBT RAM on the FPGA board, it is suggested that the design be modified to store all the intermediate results in a single memory. This reduction in memory usage will permit the newly unused memories to be powered down, which in turn will conserve energy.

Power reduction can also be achieved by changing the scheme employed to synchronize the hardware and software sub-implementations. Currently, the DSP uses a polling scheme to determine when the hardware implementation has finish production of a new binary image. As alternative method, it is suggested that the DSP be put into a low-power halt mode while the hardware implementation is processing. Upon the completion of the hardware sub-implementation's computations, the processor could then be awoken with an interrupt. It is also noted that a simple modification to the algorithm could further this reduction of power achieved by halting the processor. That is, if the hardware sub-implementation were to *intelligently* generate the binary image, it could deem whether or not it is worthwhile to start the DSP.

For instance, consider the worst case where the generated binary image possesses no foreground pixels. In this case, there is absolutely no need to begin the processor on the task of traversing this empty image to find connected component because it is known *a priori* that none exist. Thus, a slight algorithmic change and reworking of the implementation's control scheme would permit further power reduction by means of halting the process.

Lastly, owing to the fact that the target frame rate was only 5 frames/sec and that the hardware implementation achieved 7.35 to 10 frames/sec, the implementation could be changed such that it decimated the incoming video stream further. That is, more frames could be dropped to reduce power consumption and the target frame rate would still be achieved.

10.2.2 Performance Enhancements

This section discusses modifications that can be made to improve the performance of the current system. In some sense, due to the power/computation tradeoff, the suggestions present in this section are quite to the contrary of those put forth in Section 10.2.1. However, depending on the particular environment, an increased power consumption may be deemed tolerable because of to the computational benefits that it offers.

One example of a possible performance enhancement is an increase in the operating frame rate. Certainly, this change will increase the operating power of the system, but it will also permit the system to identify quick moving objects that may be missed without this modification. This increased frame rate can be achieved in two manners. First, multiple hardware sub-blocks could operate at a given time as was discussed in Chapter 6. Second, the clock speed of the processor could be increased. This increased processor speed could be achieved by decoupling the processing of the binary image from the FPGA board. After all, it is the seven wait states required for each pixel read that limits the processors throughput. Perhaps, rather than reading each pixel individually the processor could stream the entire binary image into its local memory and then operate on it at an increased processing speed. This suggested method eliminates the seven clock cycle halting incurred on every pixel that occurs when pixels are read individually.

Finally, considering that the implementation only made use of 15% of the Virtex XCV1000, it is mentioned that a larger kernel filter could be incorporated. Perhaps, it would be possible to incorporate the filter employed in the original Matlab implementation. However, owing to the fact that the image size was reduced by approximately half in the process of translating from the Matlab to the hardware implementation, it is believed that the ideal filter size would reside somewhere between that of the current and original implementations.

10.2.3 Integration with Existing Implementation

In some sense calling the modification presented in this section "future work" is a misnomer because it has already been accomplished. However, this term is used because the efforts associated with this change were not a part of this thesis effort and were performed by other individuals with only

the assistance of the author. The exiting implementation referred to in the title of this section is the acoustic beamforming algorithm developed prior to the this thesis' effort. The reader is reminded that this integration always existed as a step in the developmental path. In fact, the end goal of this project as a whole specified this integration.

As it currently stands, the acoustic beamforming, which employs both the DSP and the FPGA, cues the camera to point at its detected object. The FPGA is then context-switched to the image chipping configuration and processing is begun. Once the image chipping implementation determines a best view chip, it is transmitted or stored in the local processor memory. The hardware is then context-switched back to the beamforming configuration and cycle is begun again.

Bibliography

[1] E. K. Shinseki, "Statement by General Eric K. Shinseki Before the Airland Subcommittee on Armed Services," March 8, 2000.

[2] E. K. Shinseki, "Statement by General Eric K. Shinseki Before the Committee on Armed Services House of Representatives," February 10, 2000.

[3] L. Caldera, "Statement by the Honorable Louis Caldera Before the Committee on Armed Services United States Senate," February 10, 2000.

[4] E. K. Shinseki and L. Caldera, "The Army Vision: Soldiers On Point for the Nation...Persuasive in Peace, Invincible in War," *The Assistant Secretary of the Army Acquisitions Logistics Technology.*

[5] J. Freedman, "Pick a sensor, any sensor," *ASC Proceedings, ARL Federated Laboratory 5th Annual Symposium, College Park, MD*, March 20-22, 2001.

[6] S. Scalera, M. Falco, and B. Nelson, "A reconfigurable computing architecture for microsensors," *Field-Programmable Custom Computing Machines (FCCM) Proceedings, Napa, CA*, April 17-19, 2000.

[7] S. R. Blatt and P. J. Haney, "Resolution of multiple targets in a dense sensor field," *ASC Proceedings, ARL Federated Laboratory 5th Annual Symposium, College Park, MD*, March 20-22, 2001.

[8] S. Park, A. Savvides, and M. B. Srivastava, "Sensorsim: A simulation framework for sensor networks," *Proceedings of MSWiM 2000, Boston, MA, August 11, 2000*, 2000.

[9] K. Sohrabi, J. Gao, V. Ailawadhi, and G. J. Pottie, "A self-organizing wireless sensor network," *37th Allerton Conference on Communication, Control, and Computing*, 1999.

[10] J. Kahn, R. Katz, and K. Pister, "Mobile networking for smart dust," *ACM/IEEE Intl. Conf. on Mobile Computing and Networking (MobiCom 99)*, 1999.

[11] R. Min, M. Bhardwaj, S.-H. Cho, A. Sinha, E. Shih, A. Wang, and A. Chandrakasan, "Low-power wireless sensor networks," *VLSI Design 2001*, January 2001.

[12] R. Min, M. Bhardwaj, S.-H. Cho, A. Sinha, E. Shih, A. Wang, and A. P. Chandrakasan, "An architecture for a power-aware distributed microsensor node," *IEEE Workshop on Signal Processing Systems (SiPS '00)*, October 2000.

[13] A. Savvides, S. Park, and M. B. Srivastava, "On modeling networks of wireless microsensors," Technical Report TM-UCLA-NESL-2000-11-001, University of California, Los Angeles, November 2000.

[14] J. Rabaey, J. Ammer, J. da Silva Jr., and D. Patel, "Picoradio:ad-hoc wireless networking of ubiquitous low-energy sensor/monitor nodes," *WVLSI 2000. Proceedings. IEEE Computer Society Workshop on*, pp. 9–12, 2000.

[15] J. H. Yoo and C. Chien, "Distributed wireless microsensor for target tracking," *ASC Proceedings, ARL Federated Laboratory 5th Annual Symposium, College Park, MD*, March 20-22, 2001.

[16] G. Pottie, "Wireless sensor networks," *1998 Information Theory Workshop, Killarney, Ireland*, pp. 139–40, 22-26 June 1998.

[17] J. H. Yoo and C. Chien, "Distributed wireless microsensors for target tracking," *ASC Proceedings, ARL Federated Laboratory 5th Annual Symposium, College Park, MD*, March 20-22, 2001.

[18] S. Mehrotra, "Distributed algorithms for tasking large sensor networks," Master's thesis, The Bradley Department of Electrical and Computer Engineering, Virginia Polytechnic Institute and State University, 2001.

[19] PC/104 Consortium, *PC/104-Plus Specification, Version 1.0*, February 1997.

[20] NTSC Video Standard, "http://archive.ncsa.uiuc.edu/scms/training/general/details/ntsc.html."

[21] National Television System Committee, "http://www.oreilly.com/reference/dictionary/terms/n/national_television_system_committee.htm."

[22] P. N. Uppal, M. Sundaram, A. R. Reisinger, M. F. Taylor, J. C. Little, S. W. Kennerly, R. P. Leavitt, A. C. Goldberg, W. C. Beck, and K. Olver, "Status of two-color qwips for passive ir sensors," *ASC Proceedings, ARL Federated Laboratory 5th Annual Symposium, College Park, MD*, March 20-22, 2001.

[23] Indigo Systems, *Alpha Uncooled Microbolometer Camera User's Guide, Version 1.0.*

[24] Rockwell Semiconductor Systems, Inc., *Bt835 VideoStream III Decoder: Video Capture Processor and Scaler for TV/VCR Analog Input*, October 1998.

[25] Conexant Systems, Inc., *Bt860/861: Multiport YCrCb to NTSC/PAL/SECAM Digital Video Encoder*, 1999.

[26] Xilinx Inc., *Virtex^{TM}-E 1.8V Field Programmable Gate Arrays*, May 23, 2000.

[27] Xilinx Inc., *Virtex^{TM} 2.5V Field Programmable Gate Arrays*, April 2, 2000.

[28] IDT, *3.3 Volt CMOS SyncBiFIFO^{TM}*, January 2000.

[29] Phillips Semiconductors, *The I^2C-Bus Specification, Version 2.1*, January 2000.

[30] N. Vaswani and R. Chellappa, "Best view selection and compression of moving objects in ir sequences," *International Conference on Acoustics, Speech, and Signal Processing Proceedings, Salt Lake City, Utah*, 2001.

[31] R. Chellappa, "Personal communication," 2000.

[32] N. Vaswani and R. Chellappa, "Object detection and compression techniques for flir sequences," *ASC Proceedings, ARL Federated Laboratory 5th Annual Symposium, College Park, MD*, March 20-22, 2001.

[33] N. Vaswani, "Personal communication," 2000.

[34] L. Abbott, A. Mishra, P. Athanas, and M. Jones, "A CAμS for an image chipping implementation: Common architecture for micro-sensors." 2001.

[35] Samsung Electronics, *256Kx36-Bit Pipelined NtRAM^{TM}*, May 1999.

Appendix A

Image Sequences

This appendix contains a collection of video sequences and their associated *best view* chips that were selected by the implemented UMD image chipping algorithm executing on the CAμS 6.1 microsensor platform. These captured sequences have been decimated down from the original full frame rate of 30 frames/sec to approximately 2.3 frames/sec so that an overall concept of the motion in each video sequence could be presented in a compact fashion.

It is clarified that although the video sequences have been decimated for their presentation in this appendix, they were presented to the CAμS hardware as a full frame rate analog NTSC video stream via a VCR. In fact, the sequences shown in this appendix were captured in much the same fashion that the image chipping algorithm does when it employs an analog video stream. That is, the NTSC output of a VCR was connected to the CAμS video board, which is discussed in Chapter 4. The video board converted this analog video stream to a digital video stream to be used by the image chipping algorithm operating on the CAμS FPGA processing board.

To complete the sequence capturing process, the *Chip View* application discussed in Chapter 8 was employed to capture to disk the images received from the DSP. It is noted that these sequences were collected with the CAμS stack rather than by a simpler means, such as a frame capture card, because this technique provided sequences that are representative of the data processed by the algorithm.

Chips were collected from the fully-functioning UMD image chipping algorithm that was processing

at the frame rate specified in Chapter 9 of this thesis. These selected chips are shown in conjunction with their associated sequences on the pages that follow. These chips have been presented in two fashions. The first format, which has been labeled "Selected Chip", corresponds to the raw chipped image data, while the "Normalized Selected Chip" image is precisely the raw chipped image with its data values normalized over the range 0 to 255. Finally, it is noted that all the Infrared sequences depicted in this appendix were originally obtained using an Alpha camera, by Indigo Systems[23].

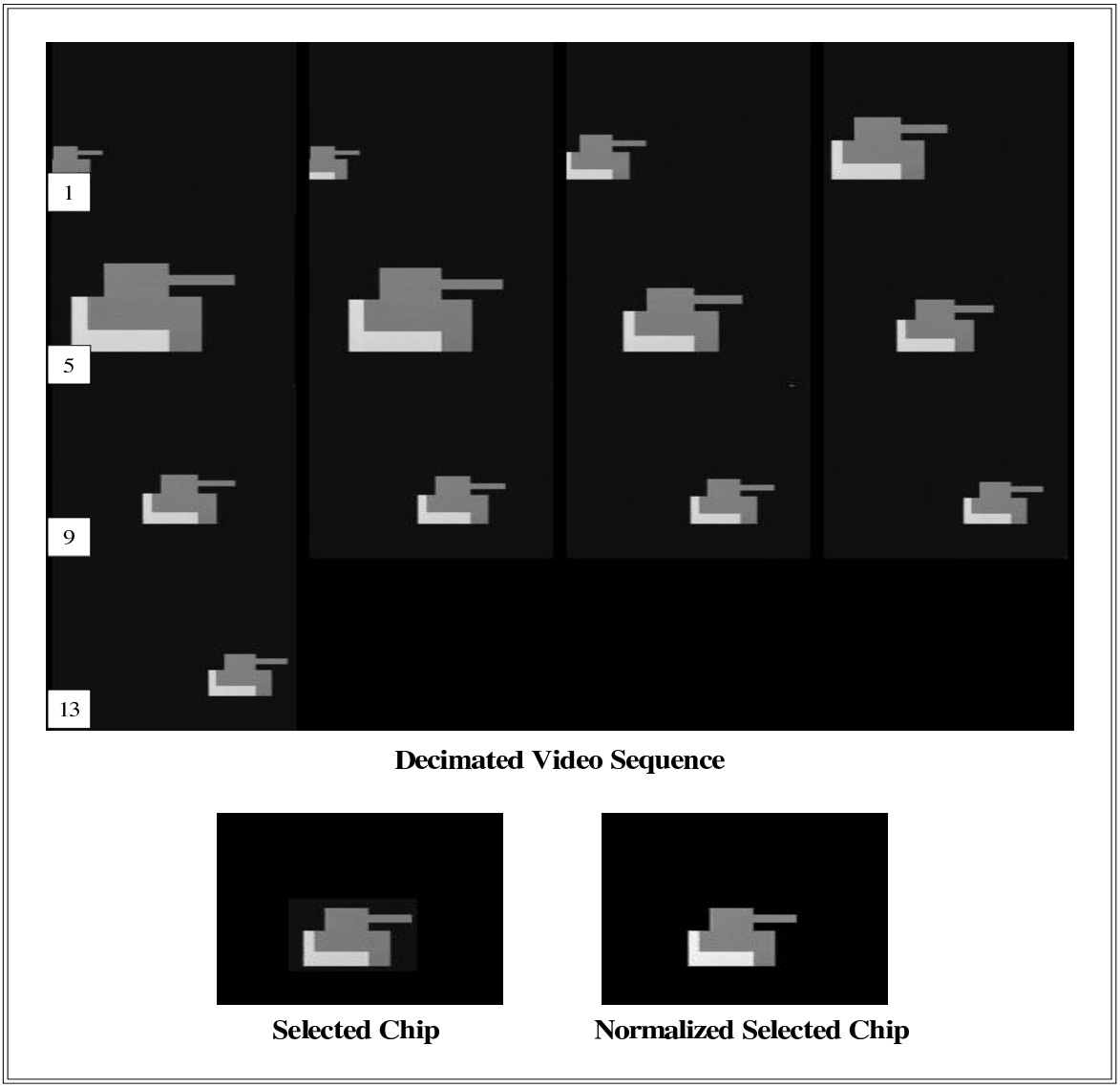


Figure A.1: Artificial Sequence and its Selected *Best View* Chip

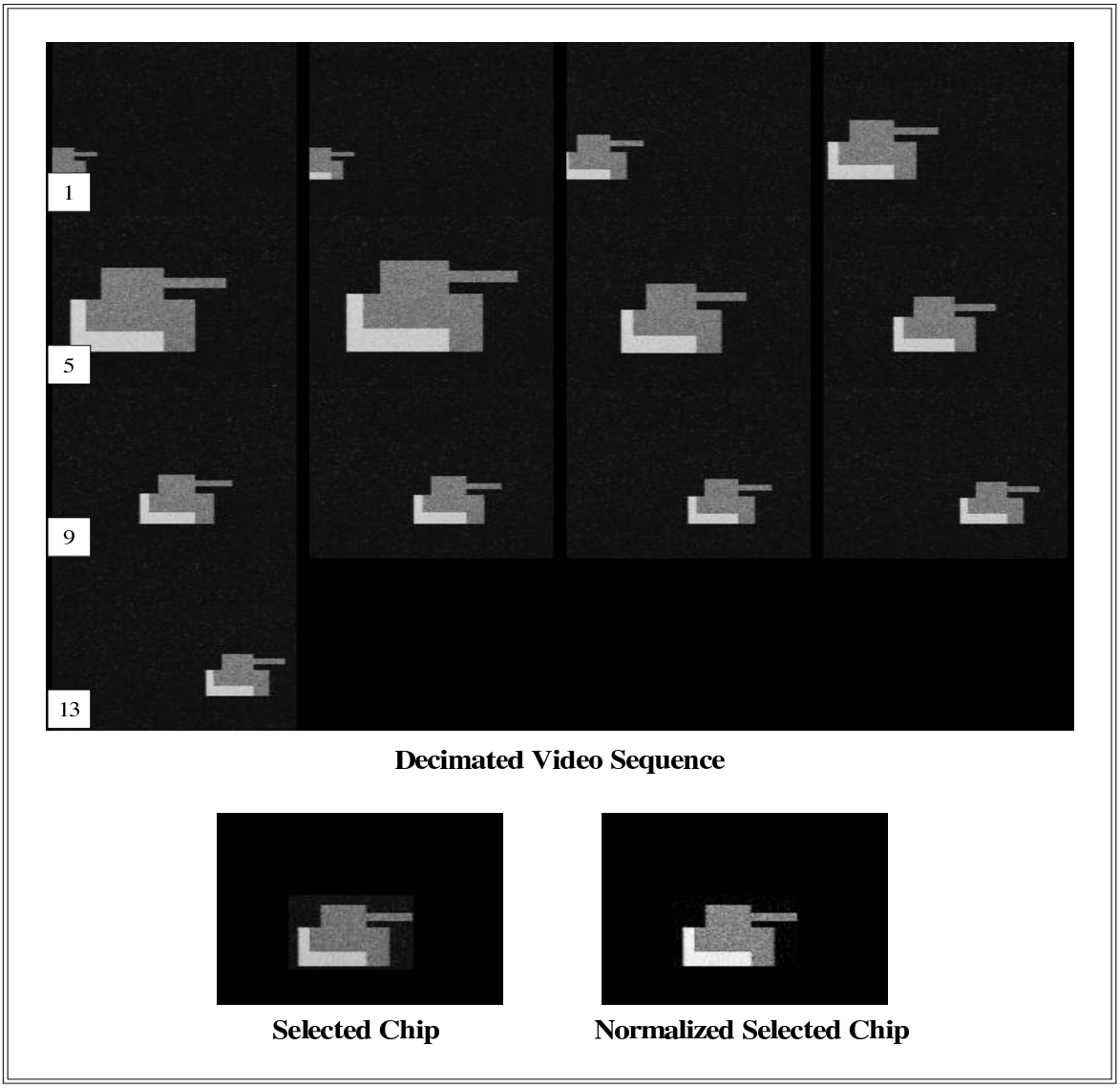


Figure A.2: Artificial Sequence with Gaussian Noise Added and its Selected *Best View* Chip

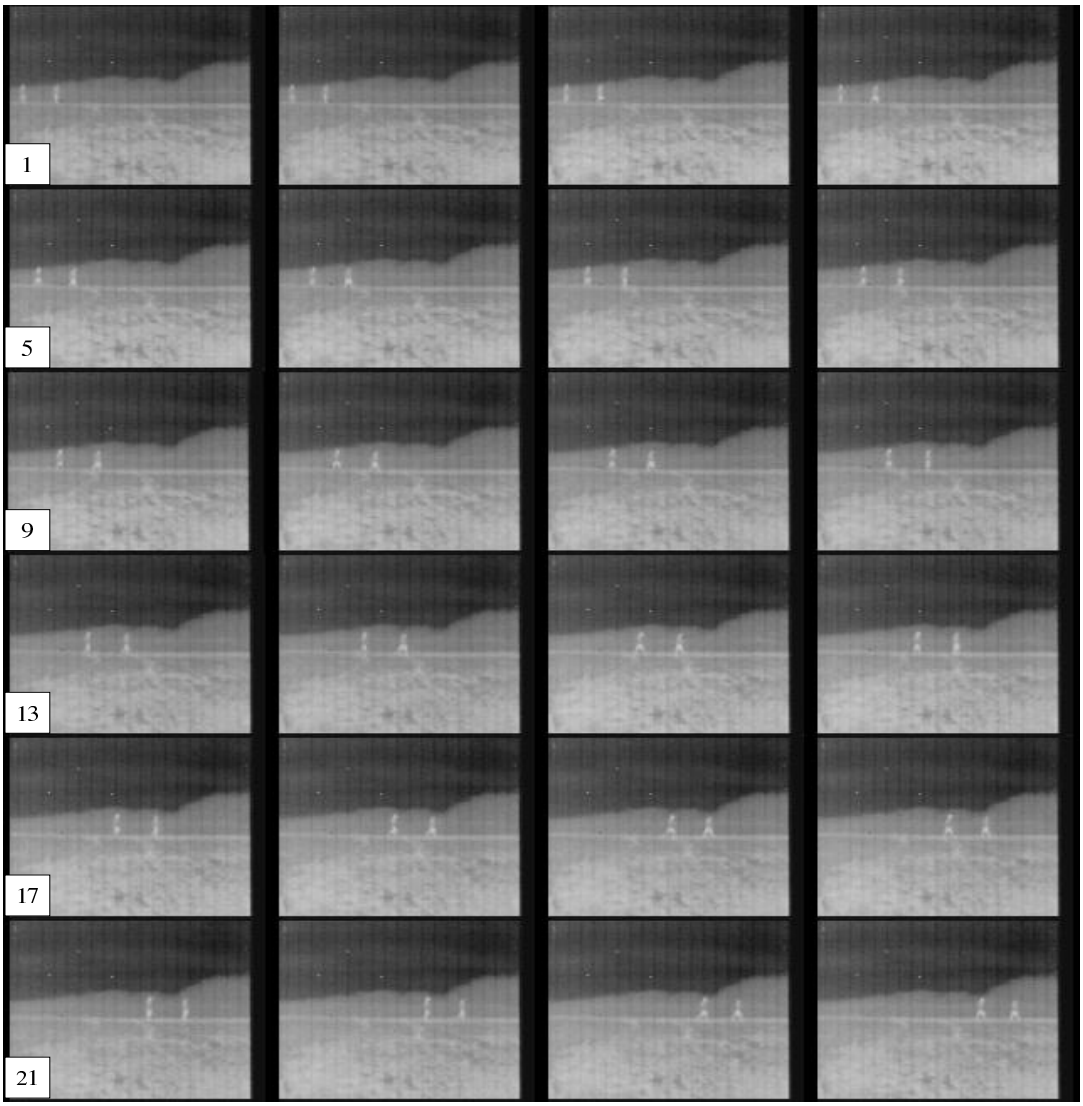


Figure A.3: Two Person Walking IR Sequence and its Selected *Best View* Chip

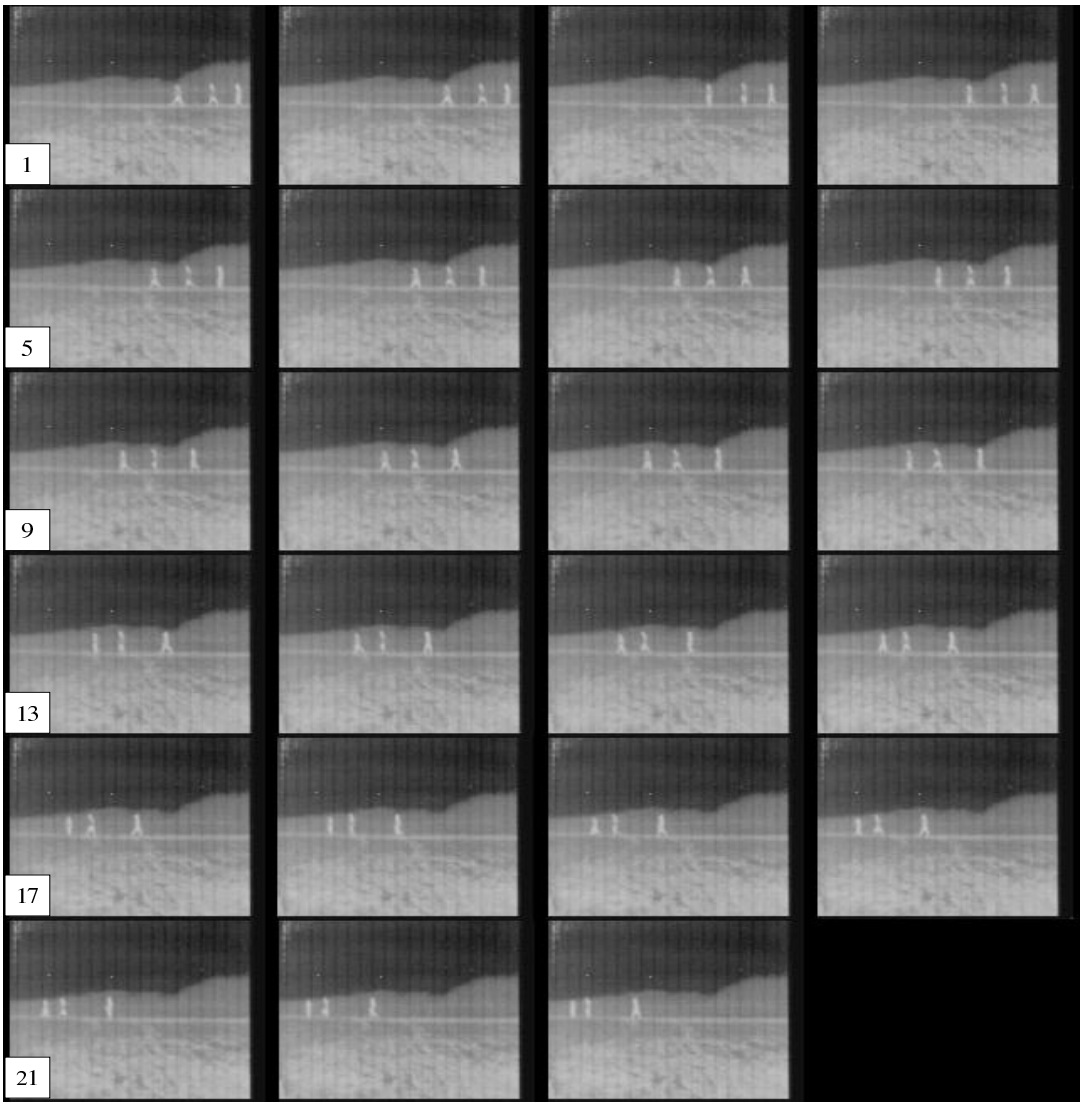


Figure A.4: Three Person Walking IR Sequence and its Selected *Best View* Chip

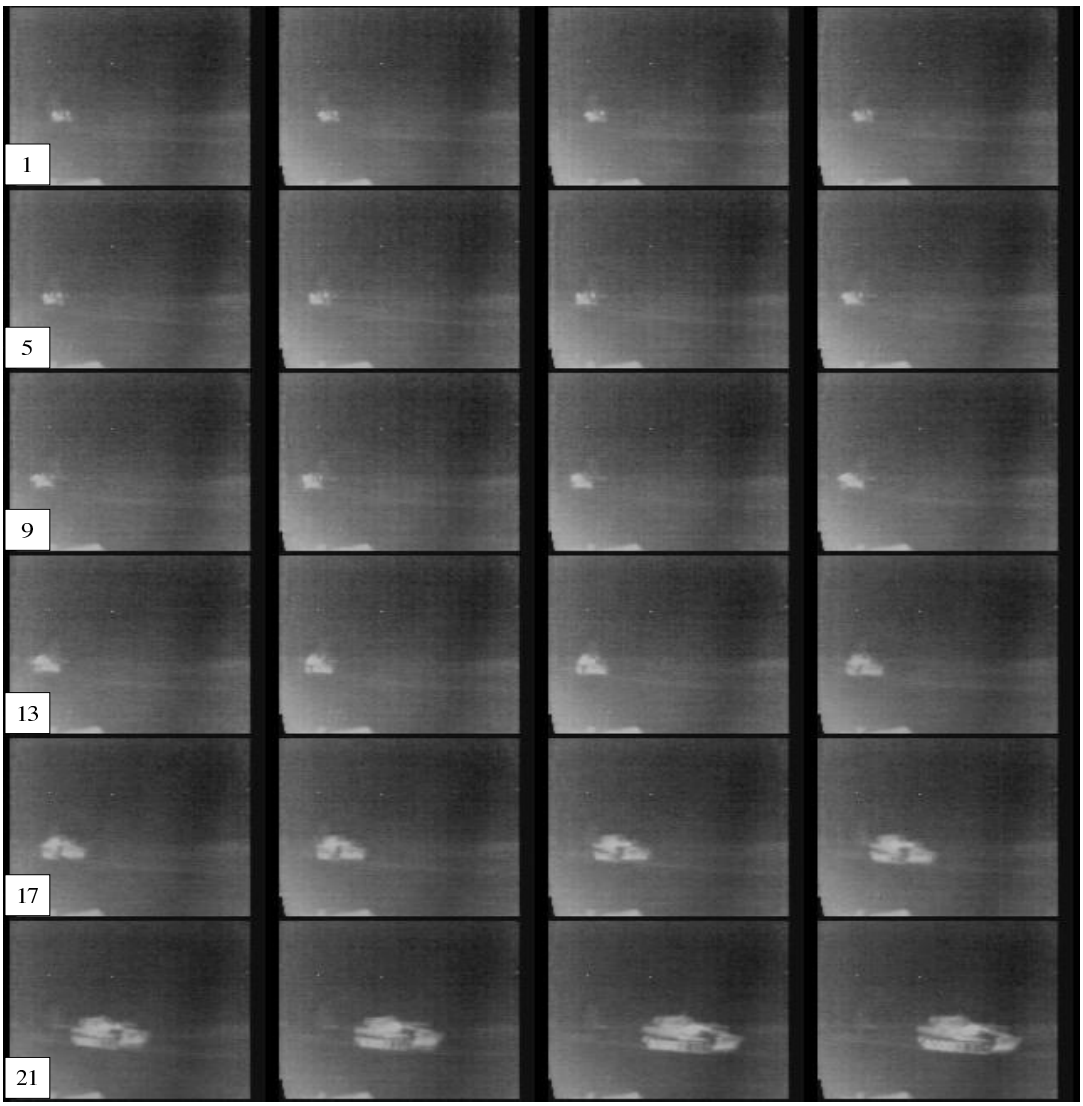


Figure A.5: Approaching Tank IR Sequence #1 and its Selected *Best View* Chip

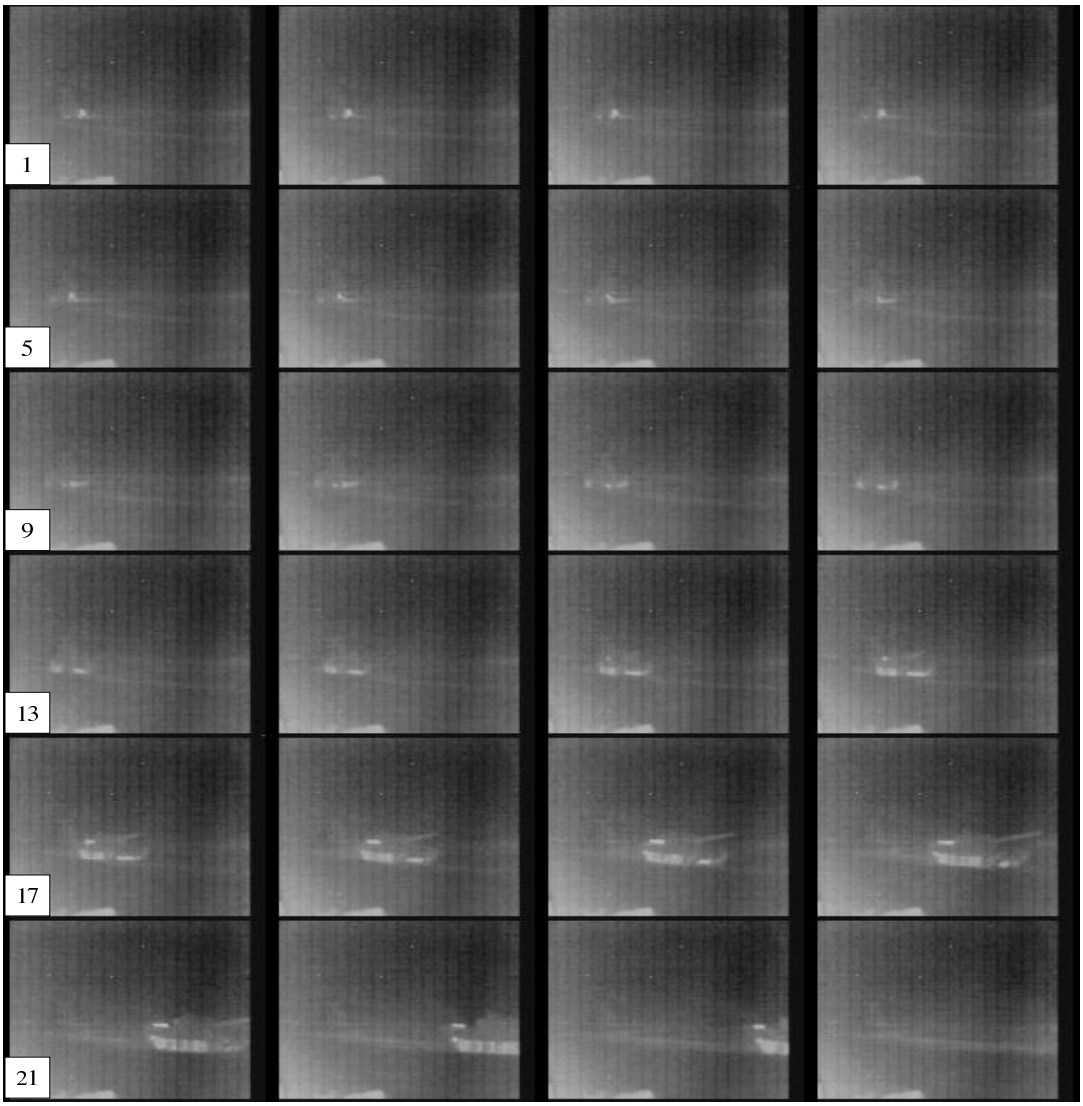


Figure A.6: Approaching Tank IR Sequence #2 and its Selected *Best View* Chip

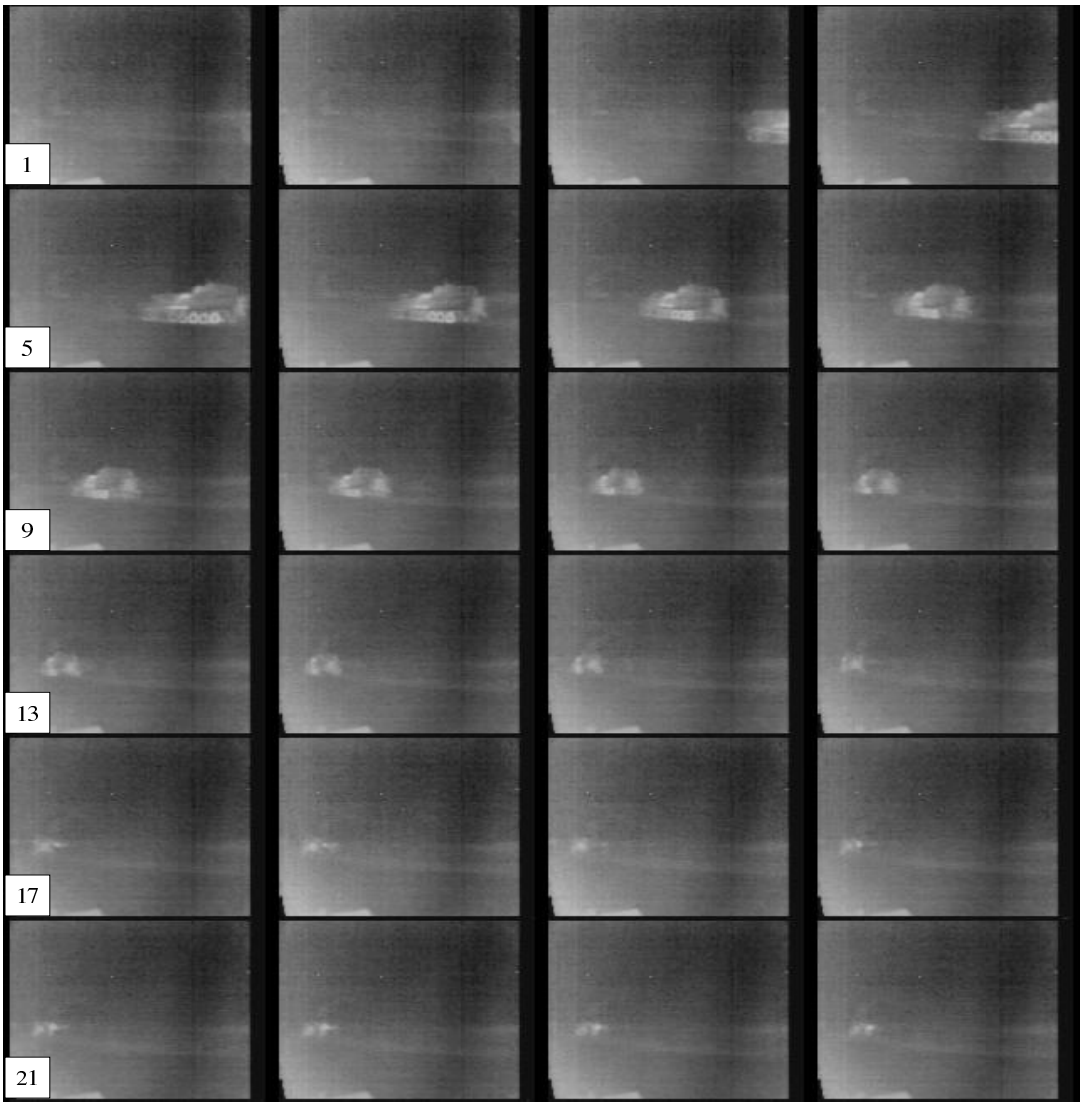


Figure A.7: Retreating Tank IR Sequence #1 and its Selected *Best View* Chip

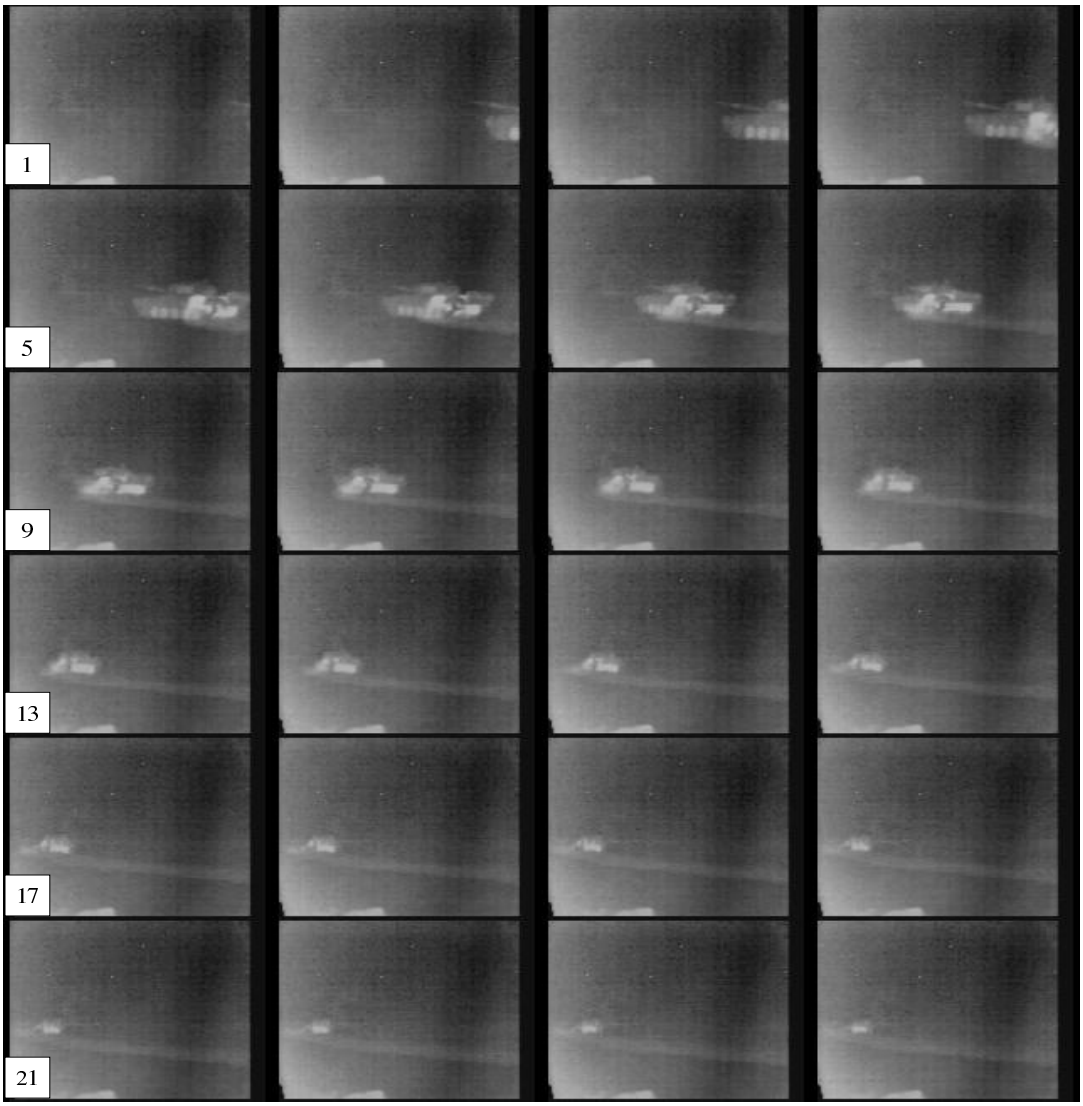


Figure A.8: Retreating Tank IR Sequence #2 and its Selected *Best View* Chip

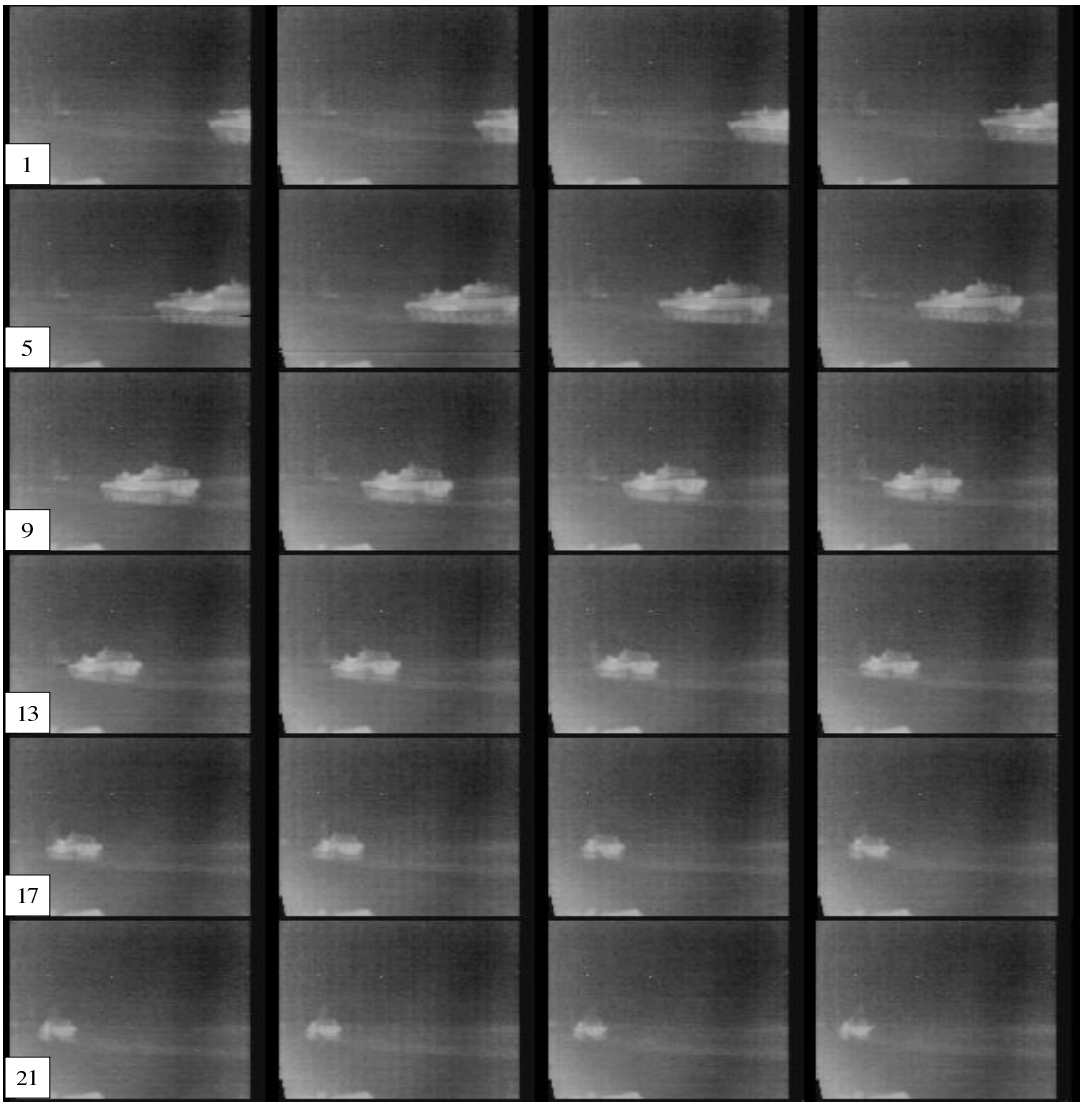


Figure A.9: Retreating Tank IR Sequence #3 and its Selected *Best View* Chip

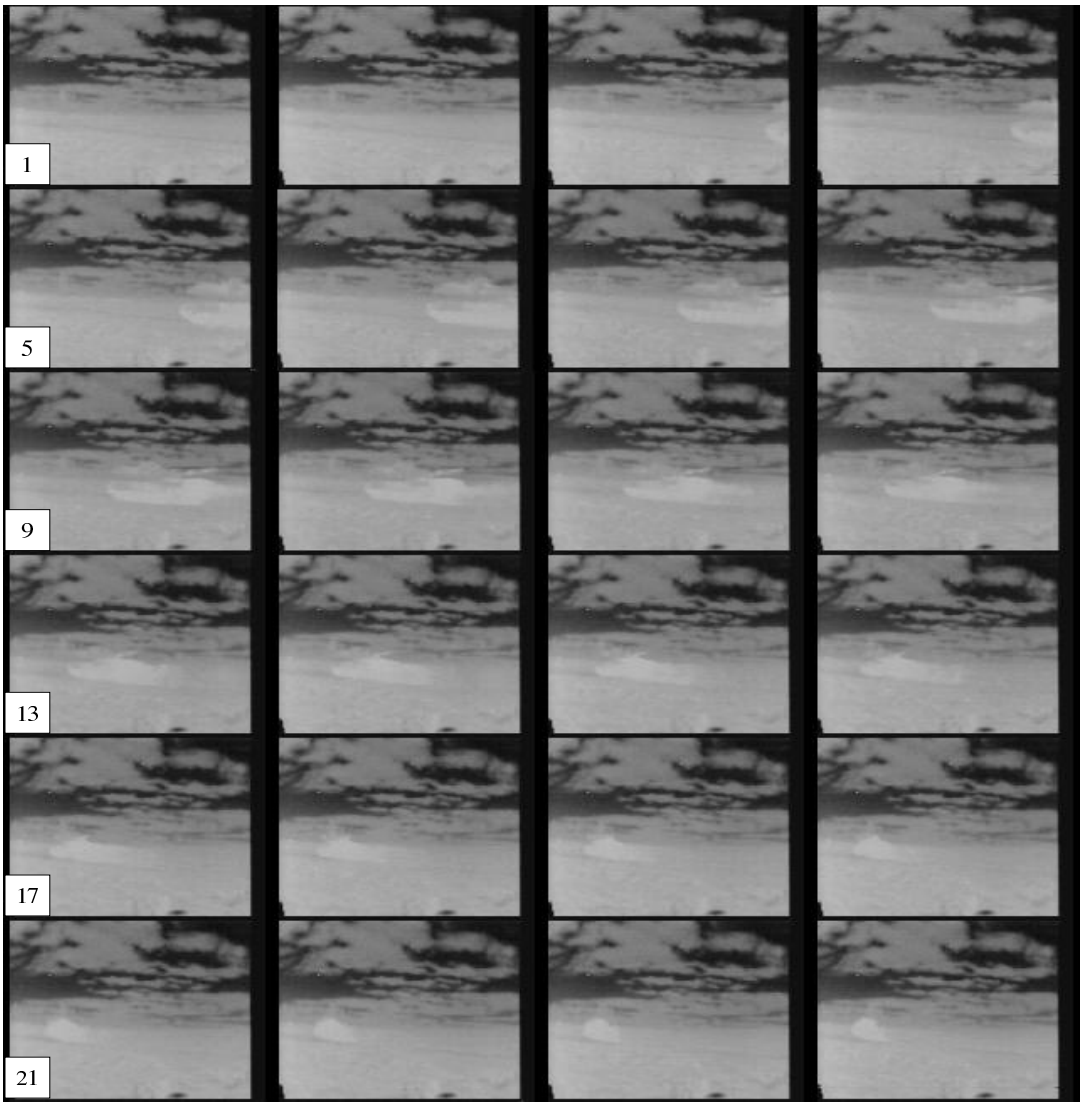


Figure A.10: Retreating Tank IR Sequence #4 and its Selected *Best View* Chip

Vita

Jon started life with notoriety. He was born early in the morning in Providence, Rhode Island on January 1, 1977, the third son of Pat and Steve. He missed being the first child of the year in Rhode Island by less than an hour. Perhaps, this event set a precedent for Jon to strive to be first in all of his future endeavors.

Although he was born in Rhode Island, the first two years of his life Jon resided in North Attleboro, Massachusetts. Jon remained in Massachusetts until his father's job required that the family move to Hampton, New Hampshire. It was here that Jon's creativity, competitiveness, and achievements grew. Since Jon's two older brothers were very successful in school, Jon fought a constant battle to get out from under their shadows. He had to prove his own identity, and show his ability to many of the same teachers that taught his brothers. This created the competitive challenge that drives Jon today.

Jon has set a goal in life with the following rule. That is, if something is considered to be worth doing, then he will do it to the best of his ability. He demonstrated this mentality over and over again as he completed his education. He received numerous achievement awards throughout his education. Most notable, he was the Valedictorian at Winnacunnet High School and was one of a very select group to win a State Scholarship to the University of New Hampshire. At Winnacunnet, Jon did not limit his activities to academics. He played soccer and was a saxophone player in the school band, as well as many other extracurricular activities. In his senior year, Jon was again recognized for his ability as he was chosen to be the Assistant Drum Major.

While at the University of New Hampshire, he won the Barry Goldwater Scholarship, which is nationally awarded to only promising students in the field of science. At graduation from the

University of New Hampshire, Jon continued to make his parents proud as he graduated being the student with the highest GPA in his Electrical Engineering class, which happened to also be the highest GPA of all students graduating from the University in 1999. His success continued as he had a number of opportunities to attend prestigious graduate schools, and he decided to go to Virginia Tech and accept the Bradley Fellowship.

Jon seems to thrive on challenging himself to excel. He is about to accept another challenging endeavor; one that will be the next in the series of challenges that drives him. Jon will continue his legacy of achievements that drive him from day to day; however, as his father, I am proud of Jon because I see that he has grown to be a great young man, who also happens to be successful.

– Stephen E. Scalera