



5-1998

Design, feasibility study and programming of malleable signal processors

Amiya A. Chokhvala

Follow this and additional works at: https://trace.tennessee.edu/utk_gradthes

Recommended Citation

Chokhvala, Amiya A., "Design, feasibility study and programming of malleable signal processors. " Master's Thesis, University of Tennessee, 1998.
https://trace.tennessee.edu/utk_gradthes/10179

This Thesis is brought to you for free and open access by the Graduate School at TRACE: Tennessee Research and Creative Exchange. It has been accepted for inclusion in Masters Theses by an authorized administrator of TRACE: Tennessee Research and Creative Exchange. For more information, please contact trace@utk.edu.

To the Graduate Council:

I am submitting herewith a thesis written by Amiya A. Chokhvala entitled "Design, feasibility study and programming of malleable signal processors." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Electrical Engineering.

Donald W. Bouldin, Major Professor

We have read this thesis and recommend its acceptance:

Danny Newport, Chandra Tan

Accepted for the Council:

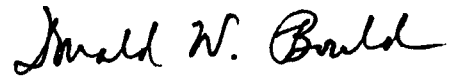
Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

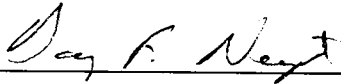
To the Graduate Council:

I am submitting herewith a thesis written by Amiya A. Chokhvala entitled "Design, Feasibility Study and Programming of Malleable Signal Processor." I have examined the final copy this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Electrical Engineering.

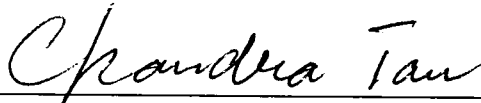


Dr. Donald W. Bouldin, Major Professor

We have read this thesis
and recommend its acceptance:

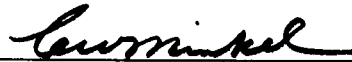


Dr. Danny Newport



Dr. Chandra Tan

Accepted for the Council:



Associate Vice Chancellor
and Dean of The Graduate School

DESIGN, FEASIBILITY STUDY AND
PROGRAMMING OF MALLEABLE SIGNAL
PROCESSORS

A Thesis
Presented for the Degree of
Master of Science
The University of Tennessee, Knoxville

Amiya A. Chokhvala
May, 1998

ACKNOWLEDGMENT

First of all I thank my mother Arati and father Anand, who gave me their support throughout my studies. I also grateful to Dr. Don Bouldin, my major professor and advisor, for guiding me in a maze of present day technological advances with an exceptional patience and also for tolerating my whims and giving them in more productive direction. I would like to thank Dr. Danny Newport for his technical advice and help in getting many softwares started and configuring them to cater my research needs. I also thank Dr. Chandra Tan for entertaining my many questions, some of which he must have found dumb. Finally, I also appreciate Dr. Jim Lyke for his direction throughout the research and Jeff Weaver, whom I have never met in person, for his guidance in programming MSP-0.

This work was partially sponsored by the Phillips Air Force Research Labs under contract F29601-92-C-0137 to Maxwell Technologies and Subtask 03-07 to the University of Tennessee.

ABSTRACT

The advent of Field Programmable Gate Arrays (FPGA) has started a new field of malleable processors. Which is FPGA-based processor whose architecture can be changed as required. These tailor-made architectures make malleable processors ten to one hundred times faster than CPU-based processors for the same application. Based on the concept of COTS (Commercial off the shelf) various available components and malleable architectures were studied. A few malleable processors were designed and studied for the requirements put forward by Phillips Laboratory. One architecture is proposed for further study and making a hardware copy of it. A malleable processor similar to the proposed one is designed and manufactured by Great River Technologies, for which some programming in VHDL is done as a part of this research.

Table of Contents

1. Introduction	1
2. Background	5
2.1 Available Building Blocks	6
2.1.1 FPGAs	6
2.1.2 FPIDs	9
2.2 Implemented Architectures	11
2.2.1 Splash-2	12
2.2.2 PAM.....	16
3. System requirements and Architecture Tradeoffs	19
3.1 Design considerations	19
3.2.1 Specifications of the MSP	21
3.3 System Requirements	25
3.3.1 Flexibility.....	25
3.4 Candidate Architectures	28
3.4.1 Module-1.	31
3.4.2 Module-2.	32
3.4.3 Module-3.	33
4. Derived Architectures and Calculations	36
4.1 Module -1.	36
4.2 Module - 2	37
4.2.1 Architecture- 1	37
4.2.2 Architecture - 2	39
4.3 Module - 3	41
4.3.1 Architecture- 1	41
4.4 Specific Implementations	43
4.4.1 Architecture- 1	43
4.4.2 Architecture - 2	45
4.5 Recommendation	47
5. Software Development for MSP	48
5.1 Problem Definition	50
5.2 ROI Algorithm	51
5.3 Finer Details of Programming	55
5.3.1 Shift-and-Add multiplication Algorithm	56
5.3.2 Comparator	56
5.4 Simulation Results	57
6. Summary, Conclusions and Future Work	60
List of References	65
Appendices	68
A. Program Listing.....	69
B. Synthesis Results.....	80
Vita	83

List of Figures

Figure 2.1: Altera FLEX 10K Device Block Diagram.....	7
Figure 2.2 : Xilinx XC4000.....	10
Figure 2.3: Splash System Level Architecture.....	12
Figure 2.4: Interface Board Architecture.....	14
Figure 2.5: Array Board Architecture.....	16
Figure 2.6: PAM Architecture.....	17
Figure 3.1: Peripheral Connections to the MSP.....	22
Figure 3.2: Module Level Flexibility.....	26
Figure 3.3: Global Flexibility.....	27
Figure 3.4: MSP Proposed by Phillips Lab.....	29
Figure 3.5: Module-1 with No Connections to Other Modules.....	31
Figure 3.6: Module-2 Includes Busses for Inter-module Connections.....	33
Figure 3.7: Module-3 with Global Memory and I/Os for Global Connection.....	34
Figure 4.1: Candidate MSP Based on a Splash Architecture Using Module-2.....	37
Figure 4.2: Candidate MSP Based on PAM Architecture Using Module-2.....	39
Figure 4.3: Candidate MSP Based on Splash Architecture Using Module-3.....	41
Figure 4.4: Altered MSP Architecture Proposed by Phillips Lab.....	43
Figure 4.5: Detailed MSP Based on Splash Architecture Using Module-2.....	45
Figure 5.1: MSP-0 made by Great River Technologies.....	48
Figure 5.2: Region-of-Interest.....	51
Figure 5.3: Algorithm Implemented in VHDL for ROI server.....	52
Figure 5.4: Connections of different Modules of ROI server.....	54
Figure 5.5 : Simulation Results of the ROI-server.....	56
Figure 5.6: Simulation for Infrared Image.....	58

Chapter 1

Introduction

The concept of malleable hardware using FPGAs and VHDL offers immense untapped possibilities primarily in the field of digital electronics. FPGAs are programmable chips that can be programmed in VHDL for specific tasks, and run much faster than a general-purpose microprocessor for the applications that can be parallelized or pipelined. In practice, the speedup is 10 to 1000 times the average microprocessor, which is comparable to supercomputer performance.

This new method drastically changes the programming methodology used for programming microprocessors. So far, the programming for normal microprocessors meant that execution of instructions would be sequential, but FPGAs are quite different in that the "hardware" can be changed. This means that the FPGA programmer designs application specific hardware. The hardware can be described by the designer in many different ways, one of which is a hardware description language (HDL). Two HDLs are

widely used: Verilog HDL and Very High Speed Integrated Circuit (VHSIC) Hardware Description language (VHDL).

The primary goals of this research were to do a study of malleable architectures already designed, manufactured, and tested by other institutions and then to propose one or more candidate architectures for use by Phillips Laboratories. The chosen architecture was implemented physically by Great River Technologies (GRT), a consulting firm near Phillips Laboratories. Since programming the architecture is a formidable task, the software development process is discussed in this report.

To achieve the stated goals, we decided to study common computer architectures from "Computer Architecture - A Quantitative Approach" [1] written by Hennessy and Patterson. The same book also discusses the different methods of quantizing the performance of a machine, which could be used to compute performance of malleable machines with some modifications. Equipped with the knowledge of computer architecture, the next logical step was to study currently available malleable machine designs. The study focussed primarily the Splash-2 and Programmable Active Memory (PAM) machines. These two are very different architectures and have been demonstrated to be highly effective in solving computationally intensive problems. After studying these architectures, we started designing potential architectures that would meet the constraints placed by Phillips Laboratory. At the same time information on different ways of optimizing digital circuits was gathered from "Synthesis and Optimization of Digital

Circuits" written by G. De Micheli [2]. These optimization techniques could be used to design software for a malleable processor.

In the second chapter of this thesis, a survey is given of programmable devices available in the market and how they are used in malleable machines. The available programmable devices support a wide range of applications. FPGAs by Xilinx and Altera were studied. However, FPGAs are not the only programmable parts in a large malleable system. Since the interconnections between different programmable logic devices may also need to be programmable to achieve maximum flexibility. An example of such a reprogrammable interconnect device is Field Programmable Interconnect Devices (FPID). Chapter 2 further discusses existing and functioning malleable architectures such as Splash 2 and PAM. Splash 2 and PAM are well tested architectures and are used as one of the basis of this research. The signal processing done on malleable machines is also studied as the final application of this project is primarily for image processing.

The third chapter describes the system requirements presented by Phillips Laboratory. The preliminary architecture proposed by Phillips Laboratory is also reviewed. A few probable and improved architectures that meet the constraints are introduced in this chapter. In this chapter, only conceptual designs are discussed, without discussing the specifications such as which programmable devices would be used in the final design.

The fourth chapter looks into the details of the architectures proposed in chapter three and discusses the pros and cons of different architectures at the physical level. This chapter also discusses an architecture proposed by Phillips Laboratory with some added improvements that are based on newer interconnect devices. At the same time, an attempt is made to achieve a balance between the unprogrammable programmable connections between FPGAs and memory.

The fifth chapter deals with a software example that could be used on a Malleable Signal Processor. The malleable signal processor is designed as an intermediate stage between sensors that take pictures of the outside world and a much larger fusion processor that is intended to make sense out of these pictures. Thus, the processing that takes place in a malleable signal processor is very low level, such as non-uniformity correction and Region-of-interest extraction. The region-of-interest algorithm has been synthesized, simulated, and tested in the experimental Malleable Signal Processor (MSP-0) designed by Great River Technologies.

The sixth chapter is the summary detailing achievements of the research and future possibilities for development.

Chapter 2

Background

In this chapter, a survey is given of programmable devices available in the market and how they are used in malleable machines. The available programmable devices support a wide range of applications. FPGAs by Xilinx and Altera are studied. However, FPGAs are not the only programmable parts in a large malleable system. Since the interconnections between different programmable logic devices may also need to be programmable to achieve maximum flexibility. An example of such a reprogrammable interconnect device is Field Programmable Interconnect Devices (FPID). This chapter further discusses existing and functioning malleable architectures such as Splash 2 and PAM. Splash 2 and PAM are well tested architectures and are used as one of the basis of this research. The signal processing done on malleable machines is also studied as the final application of this project is primarily for image processing.

2.1 Available Building Blocks

2.1.1 FPGAs

Mainly two types of FPGAs are available in the market: antifuse-based FPGAs and SRAM-based FPGAs. In antifuse-based FPGAs, programming is done by melting antifuses with high voltage. This process is irreversible so that this programming is permanent.

SRAM-based (Static Random Access Memory) FPGAS are programmed after each power up by downloading a binary file, which turns ON or OFF the SRAM switches, which in turn connect or disconnect different types of logic elements and I/Os of the FPGAs. SRAM-based FPGAs can also be programmed whenever the need arises during an operation. In this study, we have concentrated only on SRAM-based FPGAs because they are continually malleable.

2.1.1.1 Altera

One of the leading manufacturers of SRAM-based FPGAs is the Altera Corporation. The largest capacities FPGAs produced by Altera are their FLEX 10K devices as shown in Figure 2.1. These devices are important to this research mainly because of their size, which varies from 10,000 gates to 250,000 gates. FLEX devices are

based on reconfigurable CMOS SRAM elements. FLEX stands for Flexible Logic Element MatriX, which comprises logic elements (LE), embedded array blocks (EAB) and interconnections. The I/O Element (IOE) that connects a flexible logic element to the outer world is also programmable.

The gate-level architecture of the FLEX 10K is based on that of embedded gate arrays. Embedded gate arrays apply normal binary logic using the same traditional "sea of gates" architecture just as standard gate arrays. The difference is that embedded gate

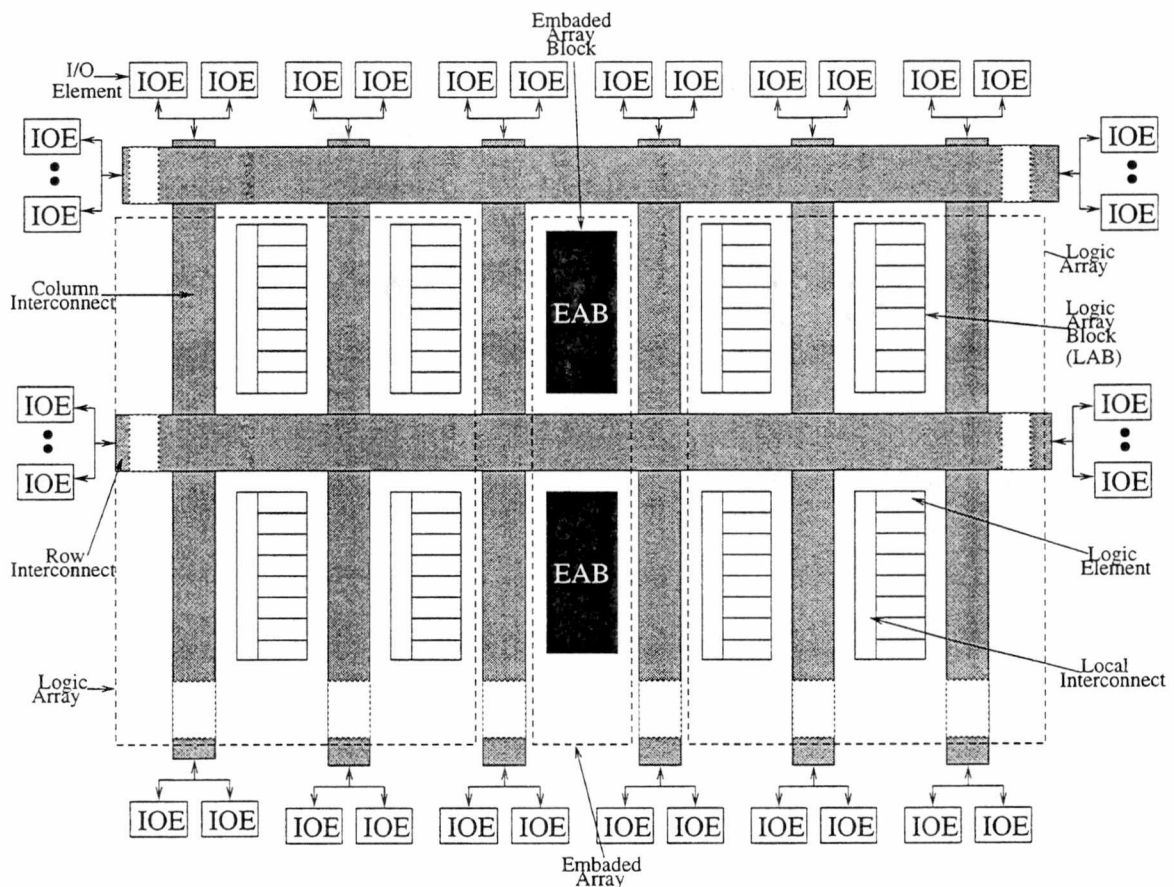


Figure 2.1: Altera FLEX 10K Device Block Diagram.

arrays have dedicated areas on the silicon to implement large Redefined functions. This approach increases speed and reduces required die area of a particular function. The possible drawback of embedded gate arrays is that it rules out customization and programmability. This disadvantage is overcome by making the embedded array block a memory block with SRAM-based interconnects. Also, in FLEX 10K devices, two primary building blocks, LE and EAB, are programmable which in turn makes the whole chip programmable.

Altera also provides EPROM (Electrically Programmable Read Only Memory) to configure FLEX10K devices upon power up, which can be used for a stand-alone design. The configuration time for FLEX 10K devices is less than 200 ms. The short time required for reprogramming enables the designer to reconfigure the device during system operation.

"FLEX 10K devices contain an optimized microprocessor interface that permits the microprocessor to configure FLEX10K devices serially, in parallel, synchronously, asynchronously. The interface also enables the microprocessor to treat the FLEX10K device as a memory and configure the device by writing to a virtual memory location, making it very easy for the designer to reconfigure the device." [8]

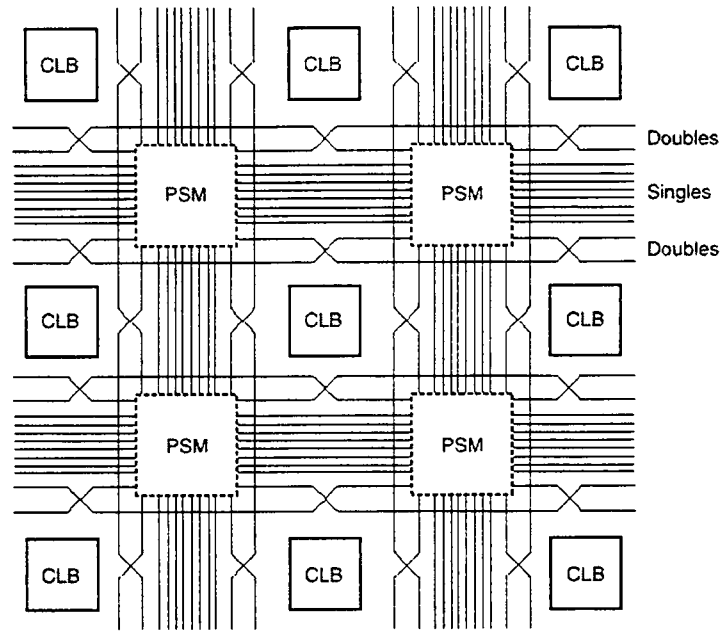
2.1.1.2 Xilinx

The other leading SRAM-based FPGA manufacturer is Xilinx. The Logic Cell Array (LCA) architecture of Xilinx contains three primary programmable parts: Configurable Logic Block (CLB), Input/Output Block (IOB), and interconnections as shown in Figure 2.2. CLBs are basic elements that provide logic functionality to the user. IOBs connect internal paths of a chip to package pins. Interconnects provide programmable routing pathways for data movement between inputs and outputs of CLBs and IOBs in the chip.

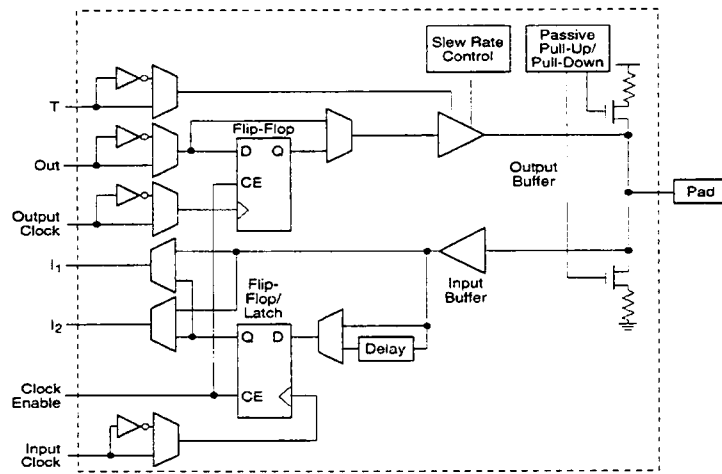
The Logic Cell Array architecture is built around CLBs, which functionally constitute the brain cells of Xilinx FPGAs. CLBs have two function generators each with four inputs and a couple of D flip-flops. Function generators can be accessed separately by logic design tools, which improves usage, as most functions have four or less combinatorial inputs. A larger function can be implemented by using a third function generator, which combines the results of the first two function generators with an extra input H 1.

2.1.2 FPIDs

Another important programmable member for a malleable signal processor is the Field Programmable Interconnect Device (FPID). For small system implementation, one or two FPGAs are enough and the interconnections between them can be static. However,



(a) Xilinx XC4000 Architecture



(b) Xilinx XC4000 IOB Architecture

Figure 2.2 : Xilinx XC4000.

for large and malleable function implementation when more than two FPGAs are needed, static interconnect places a heavy load on placement and routing resources of the FPGAs and quite often requires the helping hand of FPIDs.

FPIDs have an SRAM-configured routing structure, and they are completely prefabricated, dynamically reconfigurable devices, which are reconfigured the same way as SRAM-based FPGAs. These devices are used to interconnect programmable logic devices. The number of pins on an FPID could be around 1,000 pins, so if efficiently used, a few such devices are enough on a single circuit board.

An FPID does not have any logic capacity; therefore, the exclusive use for an FPID is to interconnect its I/O pins arbitrarily. This feature combined with a large number of I/O pins make the FPID an ideal device for large malleable architecture design with FPGAs. For example, a chip with N I/Os requires three chips with $2/3$ the I/Os to match the flexibility [6]. FPIDs are specifically designed for interconnections; therefore, its propagation delay is much smaller than that in a FPGA. For example the propagation delay in Aptix FPID is 4.5 ns; whereas the delay for a Altera FPGA is 14 ns.

2.2 Implemented Architectures

Many FPGA-based architectures have been proposed and manufactured. In this discussion, we will concentrate on multi-FPGA architectures involving high bandwidth for data transfer. These generally have a one-dimensional array of FPGA or a

two-dimensional array of FPGAs. The one-dimensional array is ideal for pipelining applications, whereas a two-dimensional array can be used for parallel processing applications as well as for pipelining applications.

2.2.1 Splash-2

Splash 2 is the best-known one-dimensional array architecture [3]. Even though the one-dimensional array of Splash-2 is hardwired, the programmable interconnection are provided for data transmission between FPGAs that are out of the sequence of the array on the board.

The system level architecture of Splash 2 is shown in Figure 2.3. Splash 2 is attached to a microprocessor that controls programming and data transfer. Splash 2 is made-up of two types of boards. The Interface Board is responsible for interfacing

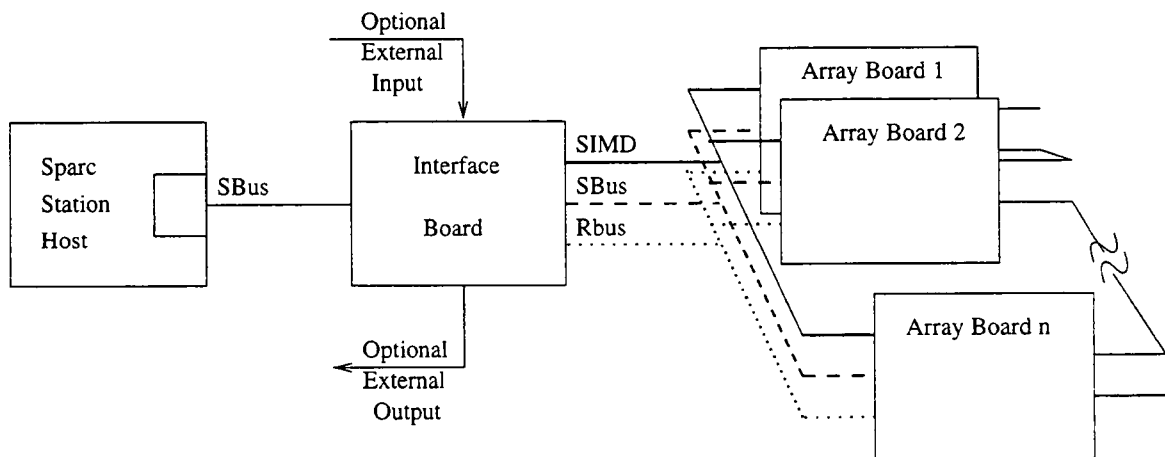


Figure 2.3: *Splash System Level Architecture.*

Splash 2 with a microprocessor and the Array Board is a programmable logic processing board.

The two fundamental modes of computation supported by the Splash 2 architecture are SIMD (Single Instruction Multiple Data) mode and linear mode. The programmer, who is using Splash 2 as a SIMD machine, can broadcast data through the SIMD bus to each array board simultaneously. The X0 on an Array Board can then broadcast the data to all other FPGAs on that board through a crossbar switch. This is parallel processing at the system level.

If the programmer views the machine as having a linear data path, then the SIMD bus from the Interface Board can be used to transmit data to the FPGA X0 of the first Array Board. Further data is moved through the linear datapath of one Array Board and transmitted to the next Array Board. The last Array Board transmits the data to the Interface Board through its last FPGA in the linear data path.

The above two data paths need not be mutually exclusive. A mix of these two is not an unlikely scenario as all busses are programmable and an application-specific data path can be programmed.

2.2.1.1 Interface Board

The primary functions of the Interface Board are SBus control and data transfer, processing the data and preparing it for an Array Board, receiving data from an Array Board and processing it for further interpretation, and generating the system clock. Figure 2.4 shows the Interface Board architecture. Address (SA) and data (SD) of the host bus (SBus) are both buffered. The host address bus SA[2:24] is buffered and decoded on the Interface Board for further use. The host data bus is gated and buffered to drive the memory data bus of the Array Board bank.

The clock can be programmed to different frequencies by the host. The clock circuit can be programmed to stop, single cycle, or run for a fixed number of cycles. This

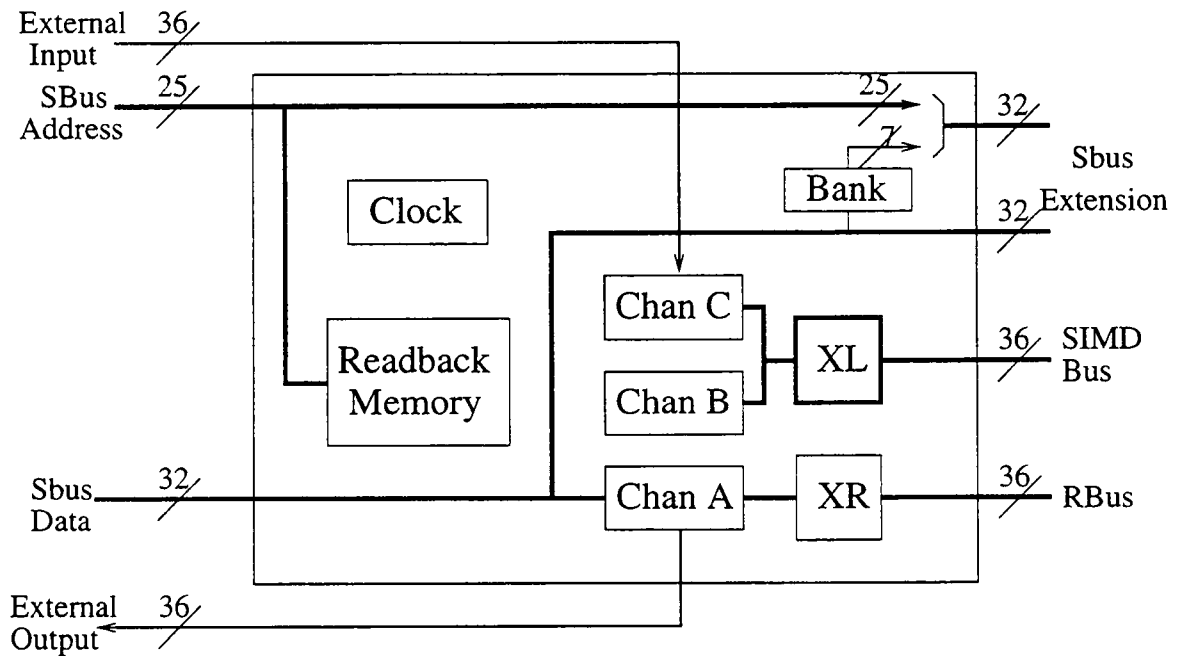


Figure 2.4: Interface Board Architecture

is helpful in the process of debugging programs. The clock can also be programmed by user-defined logic that is loaded in the Xilinx 4010 XL or XR. The host address bus (SA) is decoded and supplied to the Slave Control, which generates read and write signals for the Interface Board in response to the SBus slave cycle. User-programmable FPGAs XL and XR provide the interface between the DMA channels and the backplane bus. Channel B and C are controlled and the 36-bit SIMD bus is driven by XL. XR controls Channel A and can receive data from or transmit to the 36-bit RBus. Output registers of XR decide the direction of data transfer.

2.2.1.2 Array board

The Array Board of the Splash 2, as shown in Figure 2.5, has seventeen Xilinx XC4010 FPGAs (X0 to X16) chips as the logic processing elements. A 256 K x 16-bit memory is connected to each FPGA with a 16-bit data path. X0 is the control chip that does most of the housekeeping on the Array Board. The other sixteen processing elements X1 to X16 are connected linearly by the 36-bit wide data path. X1 to X16 are also connected with the crossbar by the 36-bit wide data path.

The crossbar is composed of nine 4-bit Texas Instrument SN74ACT8841 crossbar chips. Each crossbar chip can be loaded by eight different interconnection configurations, which can be changed whenever required in effectively one clock cycle. The crossbar allows the programmer a great deal of flexibility.

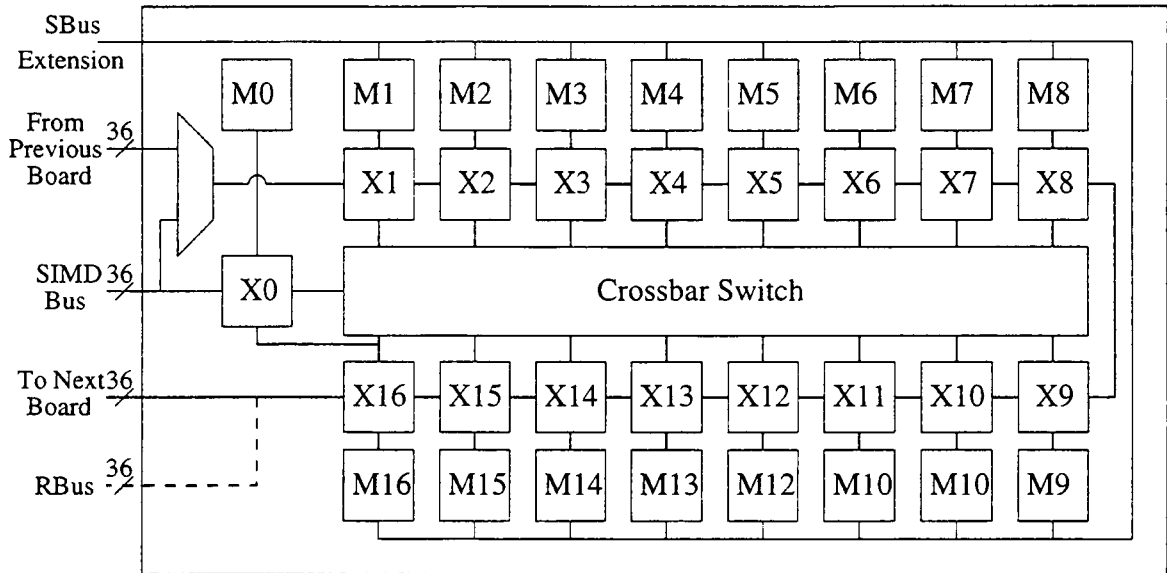


Figure 2.5: *Array Board Architecture.*

The housekeeper FPGA X0 controls which program should be loaded in a crossbar chip by a 3-bit connection. The X0 is also responsible for broadcasting data received from the SIMD Bus to other processing elements through the crossbar.

2.2.2 PAM

The second design of interest to this research is a two-dimensional hard wired array of FPGAs. The two dimensional array is useful in applications that are huge and would not fit in a single FPGA. An implementation of PAM (Programmable Active Memory) [4] is shown in Figure 2.6. All interconnections are hard wired and each FPGA is connected to four FPGAs around it. PAM allows implementation of very large functions as well as functions that can be pipelined. The drawback of this architecture is that it has hard wired interconnections, which reduces the flexibility, but signal

propagation time is much less for a hard wired connection than a programmable connection.

DECPeRLe-1 is a programmable machine designed by Digital Equipment Corporation, and based on PAM architecture, which is shown in the Figure 2.6. DECPeRLe-1 is being used at as many as a dozen scientific locations world wide.

In the above the DECPeRLe-1 architecture of a 4 x 4 matrix of logic elements (FPGA XC3090) is the logic core of the machine. Each logic element has four 16-bit wide I/Os named E, W, N. and S. which connect FPGA to its adjoining neighbors. There

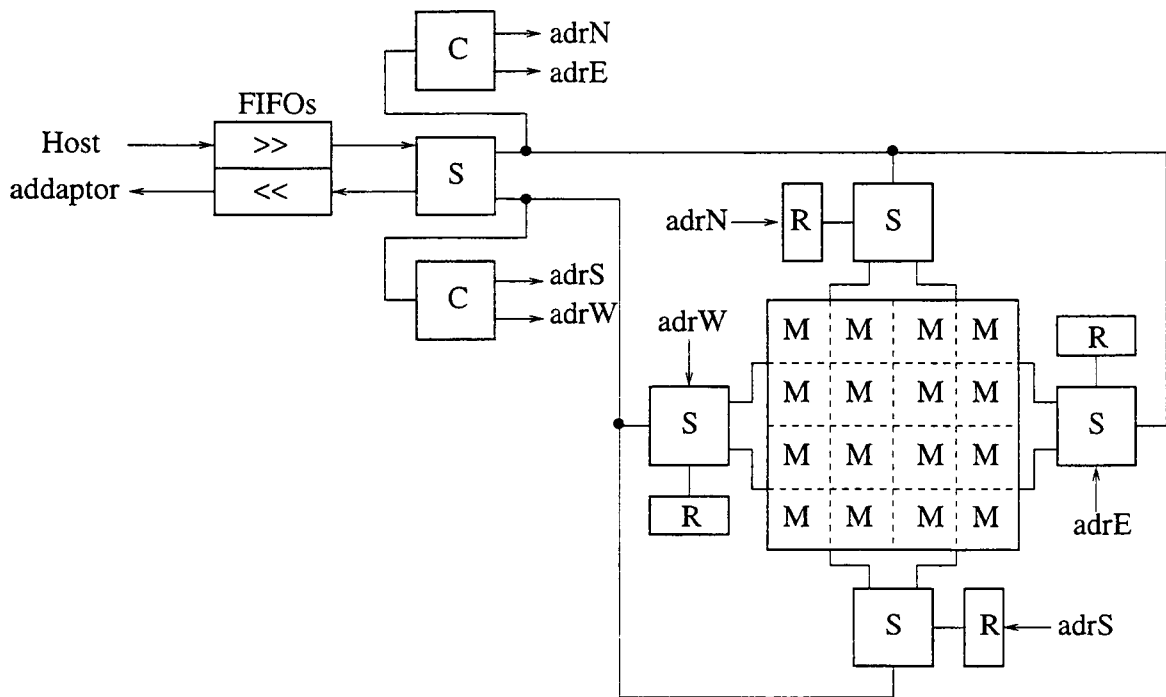


Figure 2.6: PAM Architecture.

is one 64-bit bus in each of the E, W, N, and S direction. This bus divides into four 16-bit buses and are shared by all four logic elements of the corresponding row or column.

The amount of cache RAM available to DECPeRLe-1 is 4MByte. E, W, N, and S each have 256 K x 32bit RAM R attached to it through Switch FPGA S. The address to RAM is supplied by two Controller FPGAs C, as shown in Figure 2.6.

Chapter 3

System requirements and Architecture Tradeoffs

3.1 Design considerations

Phillips Laboratory wanted to acquire or build a generic malleable processor capable of meeting most digital image processing and passive sensor subsystem control requirements. A wide range of architectures were considered in this study and several candidate solutions were proposed that can perform specific applications such as non-uniformity correction and region-of-interest operations. The architecture recommended by this study was selected based on hardware simplicity and application friendliness.

3.2 Pros and Cons

The need for a malleable processor arises from many different existing realities. First of all, the advent of programmable chips has made designing malleable circuits

possible. Using available FPGAs and FPIDs, a one million gate-equivalent programmable hardware system is feasible.

Secondly, it is desirable that the system be highly flexible. All projects have deadlines and most large projects have a reasonably large portion that remains undecided until the project approaches its final stages. At this type of critical juncture, use of malleable hardware is a logical and feasible solution.

Thirdly, interfacing two systems can become a major problem, which needs a quick solution. Traditionally microprocessors have been used to meet the time-to-market and final product flexibility requirements. In many cases this solution may not meet performance constraints and lacks the concurrency possible in a hardware design. Therefore in a typical design process, the overall design is partitioned into hardware and software components. An interface is defined and the design process continues along two parallel paths. Sometime later, the software and hardware components must be integrated. Problems usually develop at this point because of interface misinterpretation or partitioning that cannot meet design requirements. This influences the hardware, the software and the schedule. If the hardware design is realized in programmable logic, the hardware can be manipulated as easily as the software using hardware description languages such as Verilog and VHDL.

Fourthly, microprocessors are general-purpose devices and execute instructions in series. Contrary to that, parallel execution of instructions is possible in specially designed

hardware. A malleable processor can perform the iterative operations that consume excessive CPU time.

Fifthly, the malleable processors are typically 10 to 1000 times faster than a CPU when executing applications that can be parallelized or pipelined, because special purpose hardware can be programmed in the malleable processor. Since the malleable processor in the present project will be used for processing signals from sensors (including images), it has been designated the Malleable Signal Processor (MSP).

3.2.1 Specifications of the MSP

The Phillips Laboratory MSP effort involves the development of a generic processor capable of meeting most digital image processing and passive sensor subsystem control requirements. Figure 3.1 describes all the peripheral connections to the MSP. The MSP will interface to at least one neuromorphic, two-waveband sensor with special cryogenic analog front-end processor. The MSP will provide an electrical interface to gather raw and preprocessed digitized data, and it will supply clocks and biases for this sensor. The performance of the MSP is based on the ability to support several channels of input data. The capability of data throughput in each of five channels is based on the rate required to support a 512 X 512 sensor, digitized to 16 bits, operating at 100 frames/second. The outputs of the MSP are primarily five fiber-channel interfaces, which have a theoretical transport capability of 1.2 gigabits/second each, but in fact each operate at a more modest rate. These fiber-channel interfaces are viewed as *video-on-demand*

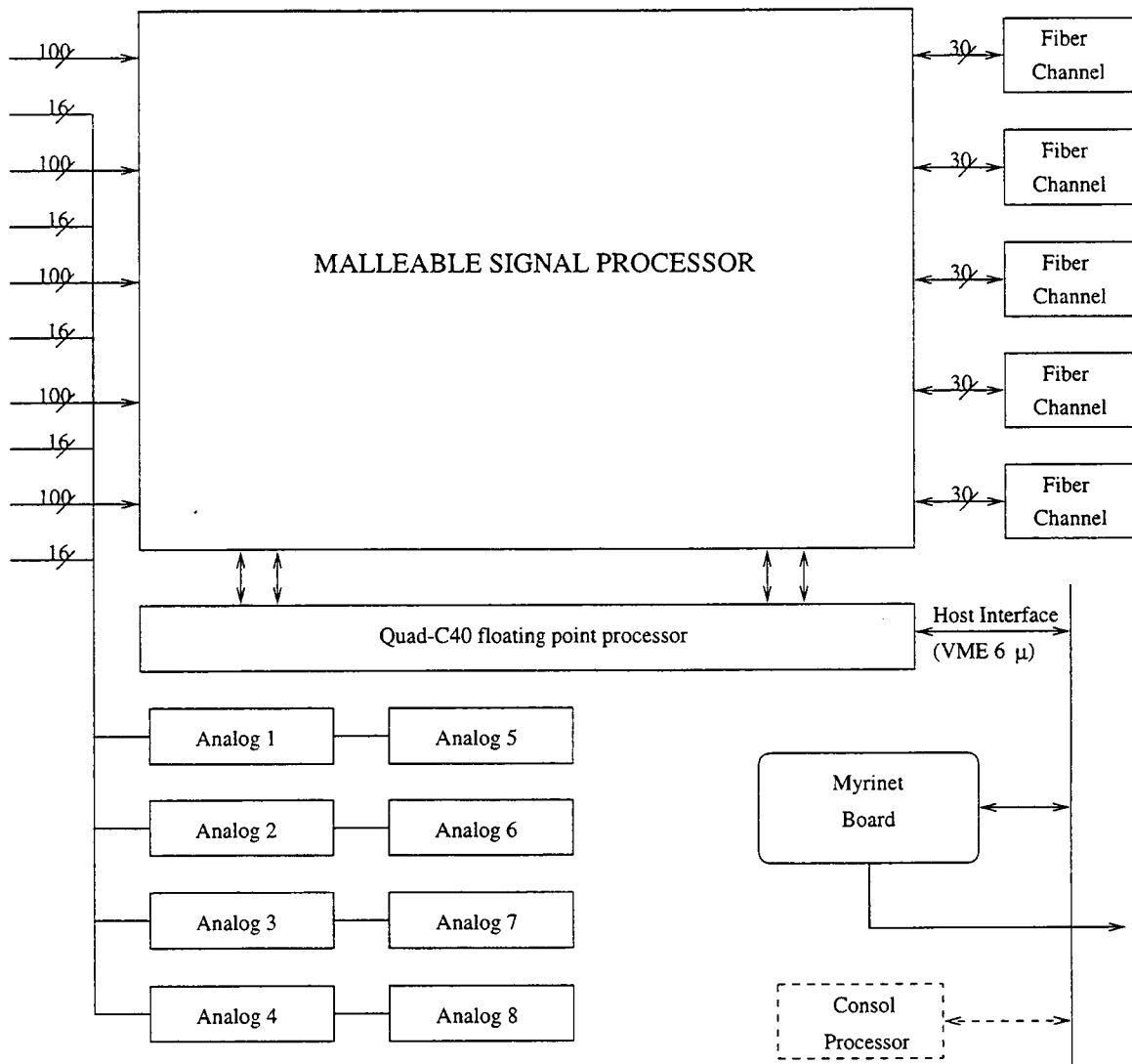


Figure 3.1: *Peripheral Connections to the MSP.*

portals to the MSP. The system control interface between the MSP and ASTP system integrator's command and data handling computer is a high-performance Myrinet (*gigabit Local Area Network*). Most commands issued from the fusion processor are through the Myrinet interface.

Software development on the MSP is limited primarily to drivers for the imaging sensors, fiber-channel, and Myrinet interfaces. Software algorithms will be reviewed at a level higher than the coding level, although non-uniformity correction and bad pixel removal may be performed in the 'stock' software that comes with the MSP. Tracking and stabilization, as well as other algorithms will be studied for implementation on the MSP, but no plans for complete algorithm development in a ready-to-use form are planned. The console processor interface and support requirements will be defined, and a user's guide based on the hardware and software developed will be prepared to guide the integrator in his own software development exercises.

It is expected that the integrator will develop the application code, including the operational flight program, for the MSP, mostly concentrated within the QC40 (a digital signal processing subsystem). Since the MSP is itself under development, it is minimally expected that the integrator would work closely with the MSP development team to insure its compatibility with the integration effort.

3.2.1.1 The MSP reconfigurable core

A million-gate reconfigurable processor will be developed for real-time embedded processing. This is the heart of the MSP and enables it to interface sensors to the fusion processor. Though primarily intended for data mapping and routing, the MSP reconfigurable core is capable of performing potentially several billion operations per second on the data for simple time-intensive operations such as gain and offset correction, gamma circumvention, etc.

3.2.1.2 The Quad-C40 (QC40).

The QC40 combines four TMS320C40 processors in a tightly-coupled multiprocessing configuration to support local floating-point processing requirements. Interestingly, the QC40 can support approximately 120-150 megaflops (MFLOPS) and up to 500 operations per second (MOPS). The primary control of the QC40 is performed through a VME backplane.

3.2.1.3 Fiber-channel interfaces.

The primary interface for real-time data for the fusion processor and the system command and data handling processor (CDHP) is a set of five fiber channel interfaces. The information presented at these fiber channel interfaces is configured through commands issued to the MSP via the Myrinet interface.

3.2.1.4 Myrinet Interface.

The Myrinet may be thought of as a powerful gigabit local area network (LAN). It is to be used as the primary command channel for the control of the passive sensor subsystem (PASS).

3.2.1.5 Console Processor.

The console processor will be a general-purpose personal computer or Unix-based computer for monitoring and providing overall control of the MSP, including FPGA personalization and calibration coefficients. The console processor arbitrates test and calibration processes.

3.2.1.6 Analog supplies.

A set of manually adjustable power supplies is provided as a part of the MSP for convenience to provide biases to sensors. A total of eight power supplies are bussed to all sensor interfaces ports.

3.3 System Requirements

3.3.1 Flexibility

Flexibility is inherent in any design using FPGAs, but this intrinsic flexibility might be restricted if the architecture does not take the application requirements into consideration. In order to have maximum flexibility, many architectures were considered

in this study and their flexibility were compared with each other. The flexibility was classified into two types as described below:

3.3.1.1 Local (module level).

Module level flexibility may be defined as the number of programmable interconnects available in the path of data-flow within a given module. The path of data begins with the wide area surveillance sensor and proceeds through the FPGA and then to memory and then to another FPGA before being output. Figure 3.2 shows an architecture with 100 % module level flexibility. In the MSP design, 100 % module level flexibility is not required because the planned applications for portions of the modules are predefined. Adding very high degree of flexibility would slow down the flow of data through the

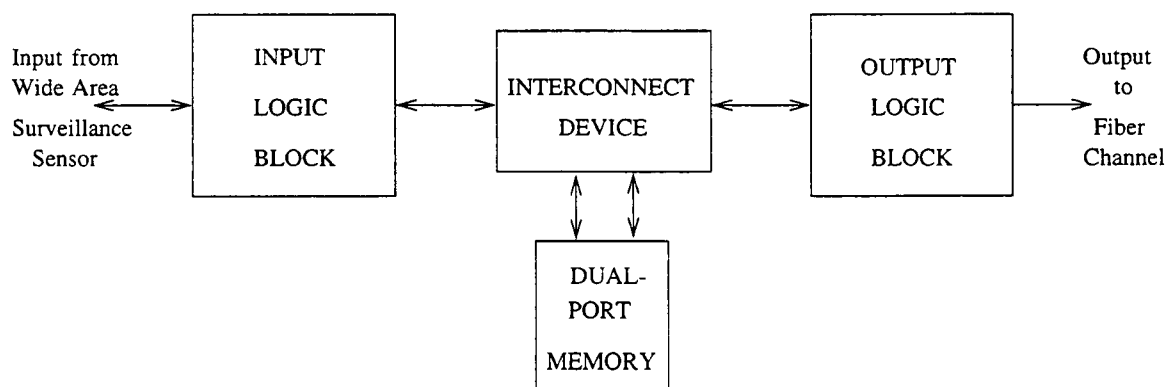


Figure 3.2: *Module Level Flexibility*

3.3.1.2 Global (inter-module).

At the global level, flexibility may be defined as the number of programmable interconnects available for data transfer from one module to another. Figure 3.3 shows the way one could have 100 % global flexibility, which could overlap with some module level flexibility. 100 % flexibility is not practical even at the global level since latency of the processing would be increased significantly. However, a moderate amount of global flexibility is prudent since large logic functions may exceed the capacity of a single logic block and it may be necessary to cascade multiple blocks. These mega functions probably

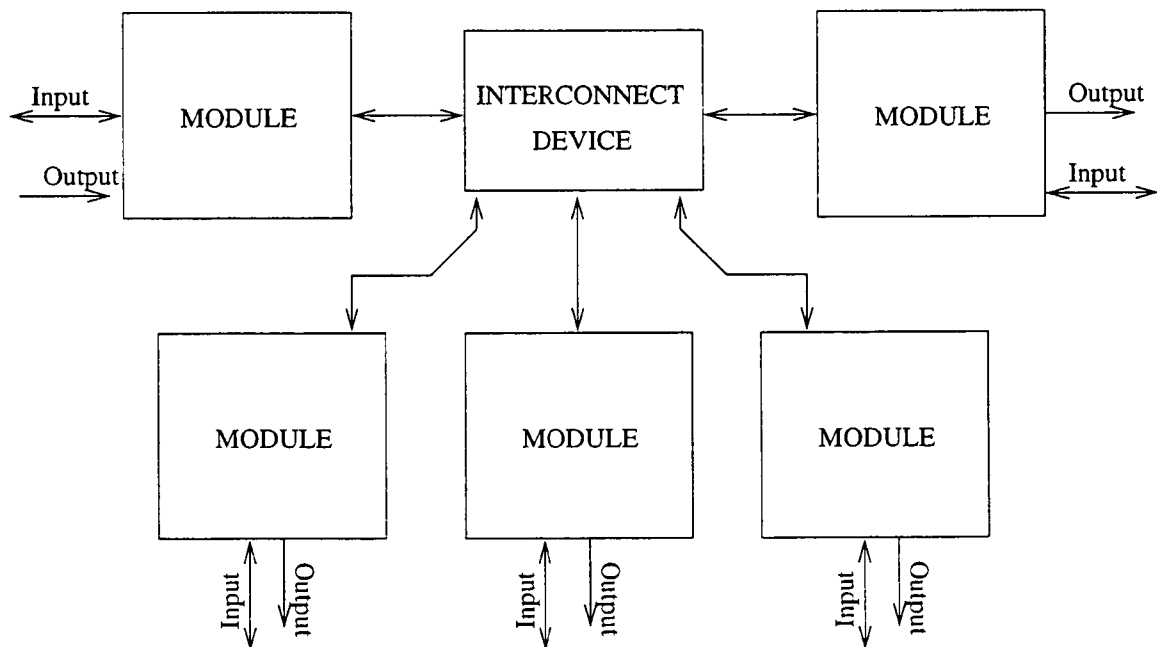


Figure 3.3: *Global Flexibility.*

will change, even after the final version of MSP is made, in accordance with the received results of the processed data.

3.3.1.3 Bit and Region-of-Interest Operations.

One of the motivations behind designing the MSP is image/data processing. This image/data processing will be bit-level operations such as non-uniformity correction and bad pixel removal and global operations such as region-of-interest as well as tracking and stabilization operations.

3.3.1.4 Required Performance

100 512x512 pixel frames/second is the necessary throughput. This requirement puts a limit on the minimum required pixel processing rate to $512 \times 512 \times 100 = 26.21$ M pixel/second.

3.4 Candidate Architectures

Many different architectures were considered throughout this study. The architectures were evaluated on the basis of criteria set up in the previous two chapters. A few of the winners are discussed here.

Before going further, let us introduce the architecture for MSP as suggested by Phillips Laboratory. In the architecture of Figure 3.4, the input logic blocks collect the wi-

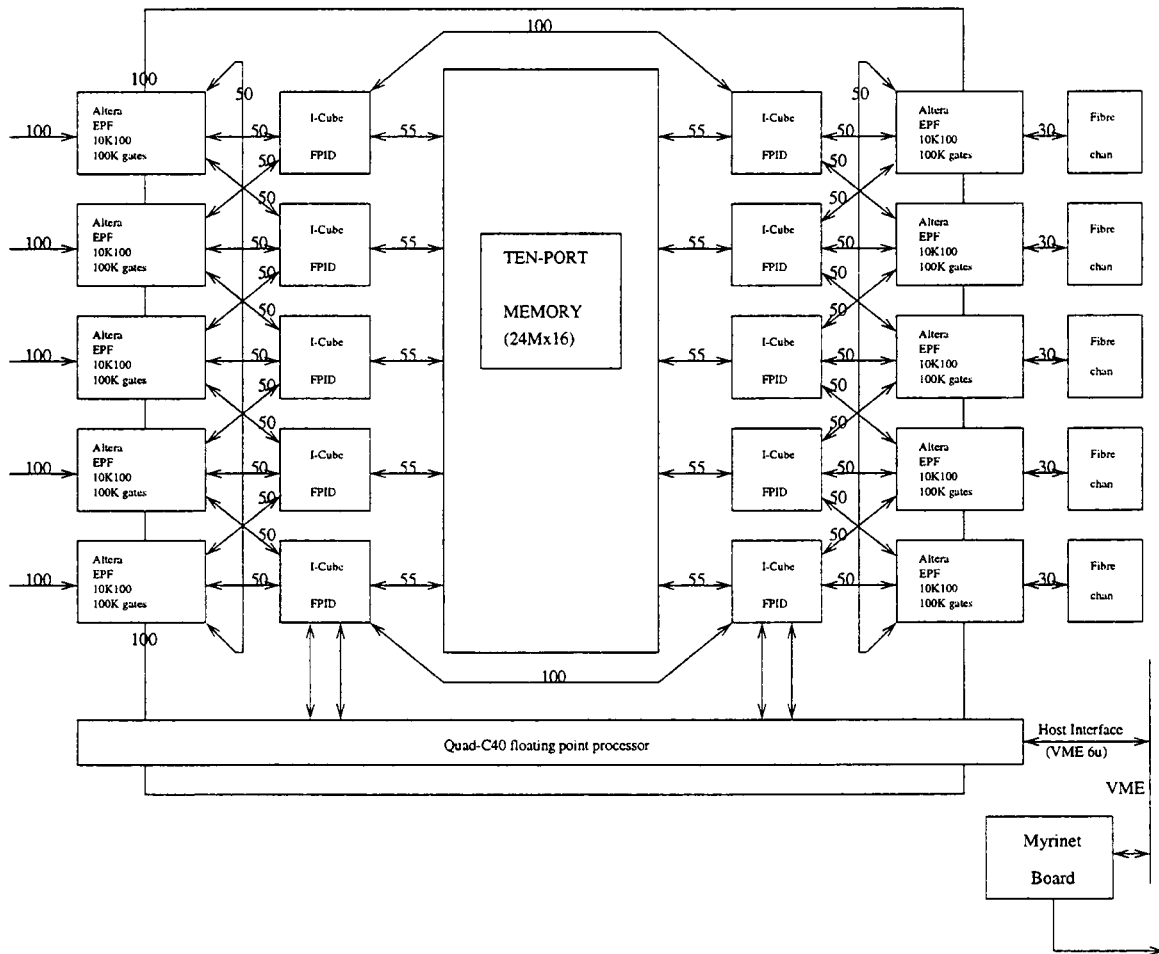


Figure 3.4: MSP Proposed by Phillips Lab.

de area surveillance sensor data for processing. These logic blocks have connections with memory and other input logic blocks through interconnect devices as shown in Figure 3.4. Output logic blocks can access data stored in memory, or data can be supplied to them directly by input logic blocks. Each input has a 100-pin connection to a wide area surveillance sensor and each output has a 30-pin connection to a fiber channel.

The starting point of this research was to understand microprocessor and system design from [1] and [2]. Further research on malleable processors was done by studying architectures such as Splash [3], PAM [4], EVC [5] and the architecture proposed by Phillips Laboratory. Incorporating the prerequisites with these architectures gave rise to many different options, which were more or less global solutions (i.e. solution for the whole problem that includes all five channels). PAM is a good example of a global solution. Additional study of the problem led us to consider the strategy of divide and conquer. Breaking up the problem into smaller modules and designing them with different amounts of flexibility gives a basic structure to the whole problem. Connecting these modules in a different fashion, as one would put tiles on a floor, produced a wide range of architectures, some of which are discussed in later chapters. Breaking up problems into smaller modules also helps in designing a stacked MCM (if needed for a future implementation) by making a module on one or two traditional MCMs. Then the interconnections between modules can be done through the outer wall of the stack.

3.4.1 Module-1.

The design shown in Figure 3.5 is the simplest one. The input data comes from the wide area surveillance sensor to the input logic block, which processes the image for bit-level operations. Then the image is stored in memory or is passed to the output logic block. If the image is stored in memory the output logic block can access data as needed. The output logic block primarily processes each image at the region level, and performs complex functions such as filtering and region-of-interest operations.

Module-1 is an isolated module. This architecture does not permit any communication with other neighboring modules. This may also mean that the user's desire to have fewer number of sensors and more processing in the MSP would not be feasible. In short, the flexibility for the programmer and future changes is not offered by this architecture.

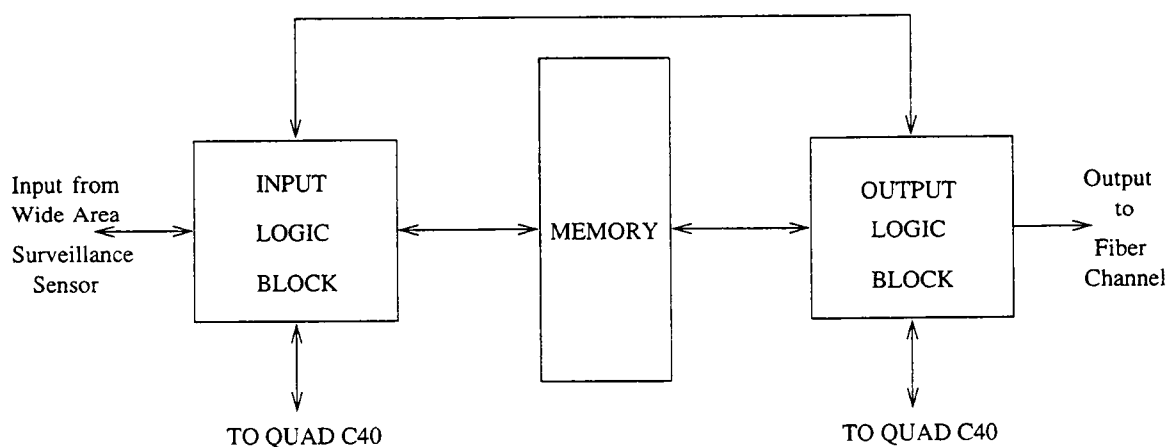


Figure 3.5: *Module-1 with No Connections to Other Modules.*

On the other hand, the primary advantage of Module-1 is its simplicity. Module-1 is very simple to build and the data flow is simple to manage. If the application is not computationally demanding, then this architecture is ideal. Further, since the memory is accessible by both the input and output logic blocks, a dual-port memory will be necessary.

3.4.2 Module-2.

A little refinement of Module-1 results in Module-2 shown in Figure 3.6. Module-2 has the same input logic block, memory, and output logic block connections, except one additional connection is provided to the input logic block and the output logic block. This extra connection provides the software designer an added flexibility but at the same time adds complexity for both the hardware and software designers.

The interconnection between different modules allowed by Module-2 permits the design of an integrated system, which can effectively have one million gates for one application. Using Module-2, one can pipeline data to hardwired adjacent modules or even use a FPID and pipeline the data as the requirement arises. Further, Module-2s can be cascaded to build a much larger MSP if required.

Because the memory is not directly accessible from outside a given module, all communication takes place through logic blocks only. Since the memory is accessible by

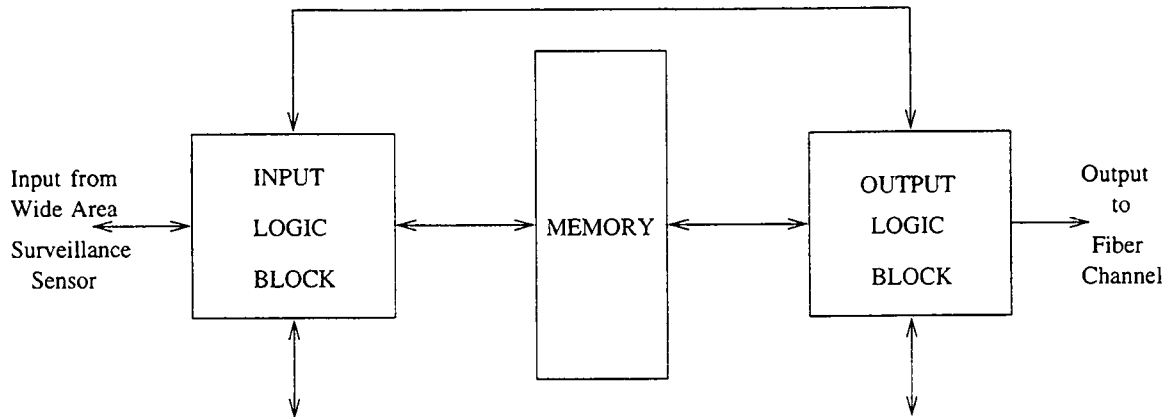


Figure 3.6: *Module-2 Includes Busses for Inter-module Connections.*

both logic blocks, a dual-port memory would be required. A MSP based on Splash is very easy to design using this Module-2 architecture. This will be discussed later.

3.4.3 Module-3.

Module-3 offers the maximum flexibility to a composite architecture designer and FPGA programmer. Figure 3.7 shows the architecture of Module-3, whose input logic block takes data from a wide area surveillance sensor. The input logic block processes the image data and prepares it for the next stage. The next stage in MSP design using Module-3 could be the output logic block of the same module or memory or any other logic block of the MSP. Any logic block chosen by the MSP designer could access the data stored in memory. The data could be passed to fiber channels by any other output logic block. This flexibility in accessing memory requires some sort of memory management and might necessitate a ten-port memory controller for the MSP.

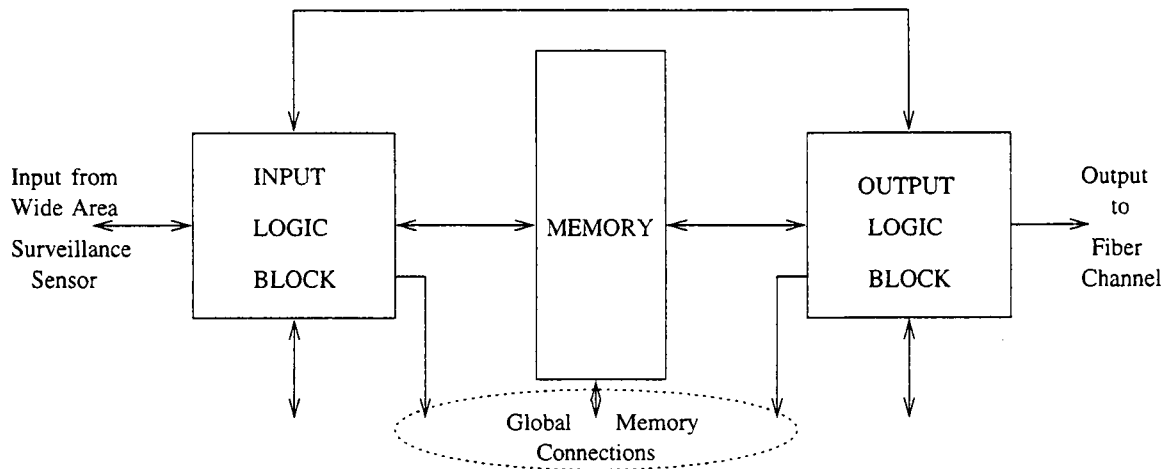


Figure 3.7: *Module-3 with Global Memory and I/Os for Global Connection.*

The main disadvantage of Module-3 is its complexity, which also brings out the primary advantages of the module. Due to added different types of intra-module as well as inter-module connections, the netlist becomes complex. To use the module at its fullest potential, designing a multi-port memory controller and the use of a FPID would be imperative.

The advantages of this design on the other hand are numerous. The first advantage is that one can have a design larger than two hundred thousand gate equivalent and this mega function theoretically could occupy up to one million gates. Secondly, memory is accessible directly by any other module, though the total number of modules that could have this type of accessibility depends on the number of extra address lines provided in Module-3. Thirdly, it gives the designer absolute flexibility in designing the MSP.

Fourthly, Module-3 is cascadable just like Module-2. Fifthly, memory management could be done hierarchically as is done in modern computers.

Chapter 4

Derived Architectures and Calculations

The fourth chapter looks into the details of the architectures proposed in chapter three and discusses the pros and cons of different architectures at the physical level. This chapter also discusses an architecture proposed by Phillips Laboratory with some added improvements that are based on newer interconnect devices. At the same time, an attempt is made to achieve a balance between the unprogrammable and programmable connections between FPGAs and memory.

4.1 Module -1.

The design of Module-1 is straightforward but, since each module is isolated, an integrated MSP is not possible. The designer simply employs five self-contained modules for the application.

4.2 Module - 2

Module-2 can be used as a tile with which one can design different architectures as per requirement of the project. Two well tested arrangements are discussed in the following sections.

4.2.1 Architecture- 1

Module-2 can be treated as a cell of the Splash architecture. Properly arranging them using a FPID produces us a Splash with different, faster, and larger parts as shown in Figure 4.1.

As Figure 4.1 shows, in a Splash-type design, input and output logic blocks can work as a single connected logic block, and both of them have access to dual-port memory. The image data is supplied by five wide area surveillance sensors to input logic blocks. The input address/data/control bus for the wide area surveillance sensor is a 100-bit bus. The output to each fiber channel is 30 bits wide.

Input data to each input logic block can come from the wide area surveillance sensor, a hardwired adjacent output logic block, or any other logic block of the same MSP. Between two neighboring logic blocks *d-bit* wide data/control bus is hardwired. From every logic block of MSP, an *e-bit* wide bus is connected to the interconnection module.

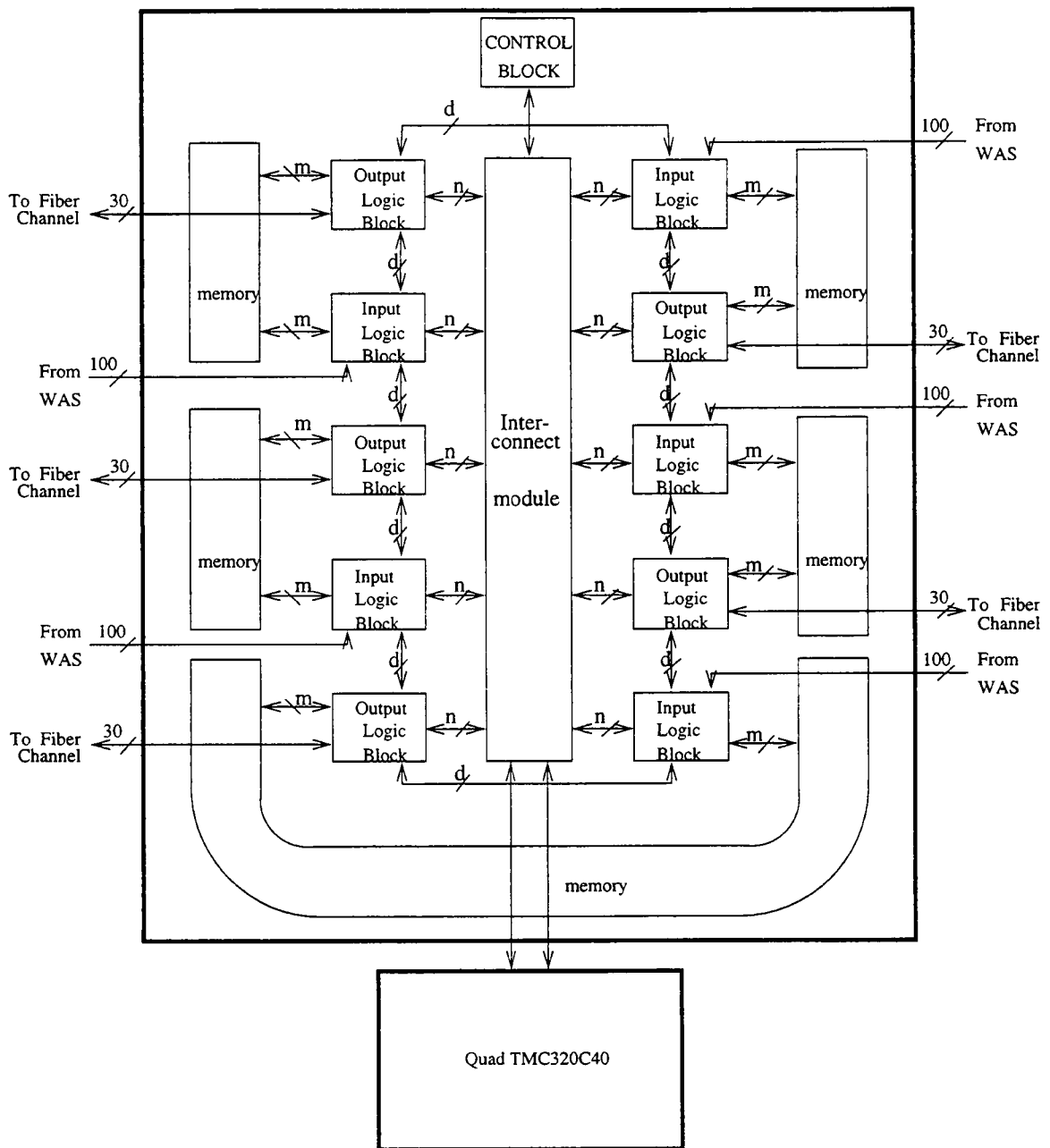


Figure 4.1: Candidate MSP Based on a Splash Architecture Using Module-2.

The interconnection module is also connected to a Quad TMC320C40, which manages the booting as well as programming of the logic blocks and interconnect module. Each logic block is connected to the dual-port memory through an *m-bit* wide data/address bus.

4.2.2 Architecture - 2

Using Module-2, a PAM-based array of FPGAs can be used to construct a MSP, but project requirements and availability of much larger FPGAs makes a PAM-type design impractical. Therefore, further consideration of this type of architecture was not pursued. We were able to produce a design shown in Figure 4.2 of an MSP where five Module-2s are connected in a two-dimensional array. This is not exactly a PAM-type design but it shows the possibilities presented by a modular design, which can be tiled to produce a variety of dissimilar architectures.

In Figure 4.2 four modules are connected in a ring and the fifth is placed in the center of the ring. All modules are connected to the interconnect block, through which the central module can be accessed by the outer modules.

This module is not identical to PAM, because in PAM, each logic block has four I/O busses from all four sides called E, W, N, S. and each logic block is connected to others through these busses in a two-dimensional array. Apart from this difference, in

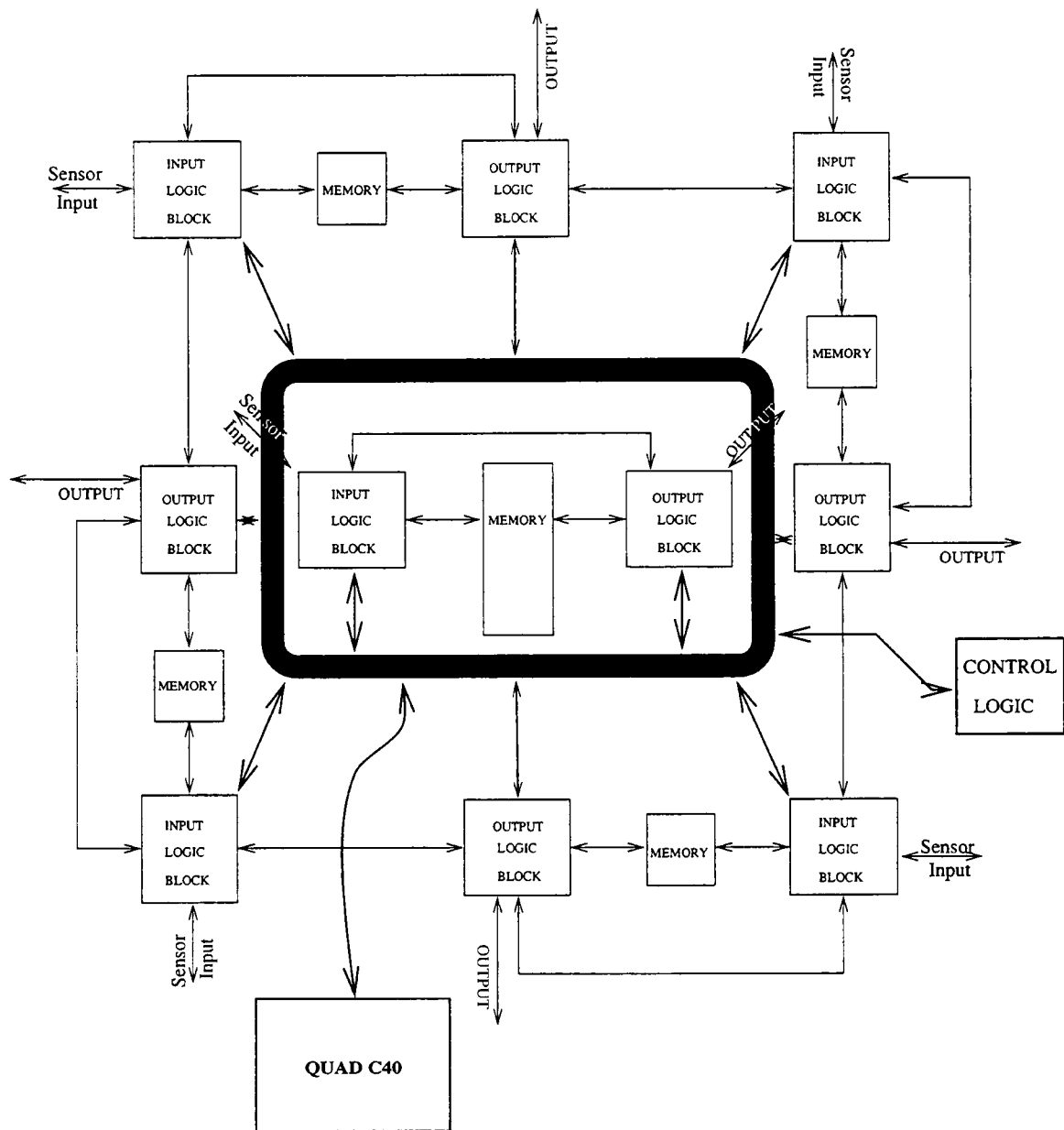


Figure 4.2: Candidate MSP Based on PAM Architecture Using Module-2.

the memory is placed outside the array, and no interconnect devices are used in PAM.

The requirement of five wide area surveillance sensor inputs and five separate outputs make a two-dimensional array configuration of PAM impractical. The two-dimensional array is ideal for implementation of mega functions using much smaller, 13,000 effective gate XILINX XC4013 FPGAs. Presently available 100,000 effective gate ALTERA 10K100 in a single chip is already more than seven times larger, and five different input and output bus requirements make PAM an inefficient design for the present application.

4.3 Module - 3

Module-3 has all the features of Module-2 and some extra added features of its own so all architectures discussed for Module- 1 and Module-2 can be created using Module-3.

4.3.1 Architecture- 1

As mentioned earlier the major difference between Module-2 and Module-3 is that the latter has memory accessible by logic blocks outside a given module. This change obviously affects the memory management of MSP. An example of this is shown in Figure 4.3.

The flexibility offered by Module-3 is well used in the above architecture, which

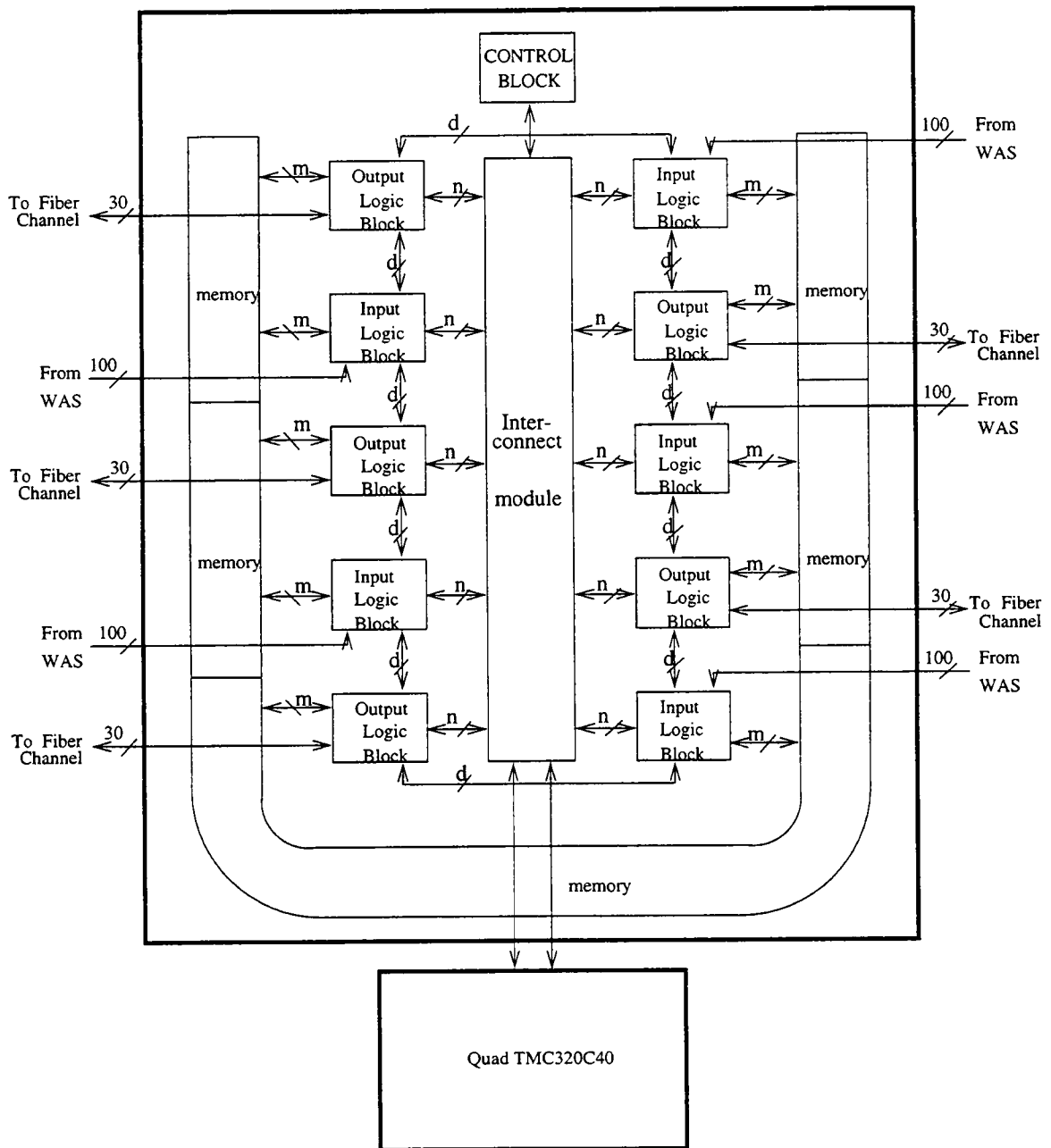


Figure 4.3: Candidate MSP Based on Splash Architecture Using Module-3.

is a derivation of the Splash architecture. In Splash, each FPGA has its own local memory that is not accessible by other FPGAs. However, using Module-3, memory is made accessible to all logic blocks through *m-bit* address/data lines. All other features remain the same as in the architecture- 1 design using Module-2.

4.4 Specific Implementations

Replacing all logic blocks in all architectures by Altera FLEX 10K100s, interconnect devices by Aptix AX1024, and memory by IBM dual-clock memory IBM041810QLAB, we can achieve actual designs.

4.4.1 Architecture- 1

The architecture in Figure 4.4 is an altered version of MSP suggested by the Phillips lab. This architecture takes advantage of larger FPIDs available in the market. Some connections are changed to avoid FPID to FPID connections, which would waste FPID connections and slow down the MSP. In addition, FPGA-to-FPGA connections and FPGA-to-memory connections are through different FPIDs to avoid passing signals through two FPIDs. However, use of multiple FPIDs in series will enable the designer to connect any two pins of the 10K100s.

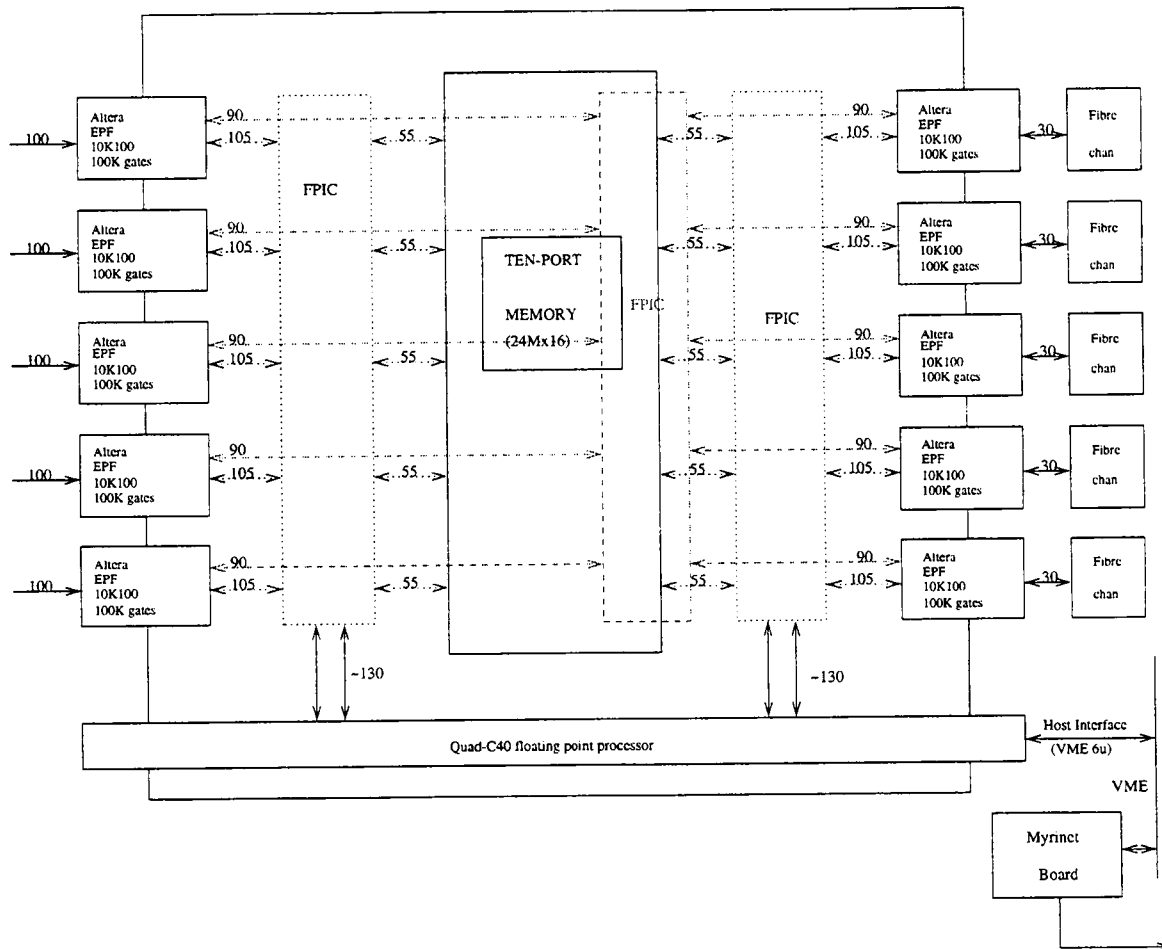


Figure 4.4: Altered MSP Architecture Proposed by Phillips Lab.

The above architecture was designed as a single entity and not as a composition of smaller modular parts. This jumbo 1,000,000 programmable gate design is not a practical way of designing any circuit, because debugging the hardware would be a monstrous task and a ten-port memory controller would be expensive and difficult to design.

4.4.2 Architecture - 2

Figure 4.5 shows the details of the architecture based on Splash using Module-2. FPID- 1 and FPID-2 are in parallel to each other to avoid FPID-to-FPID connections. This arrangement increases throughput and also increases the availability of FPID pins for connections to FPGAs since a FPID is not connected to another FPID.

The modular design of the architecture shown in Figure 4.5 makes manufacturing it much easier. If hardware of one module is properly debugged, then other modules can be manufactured without much difficulty. The memory is separate for each module. This eliminates the need for a ten-port memory and it will also simplify the software, although data-sharing between different modules could become a cumbersome problem.

In Figure 4.5 besides having two interconnecting FPIDs, one 10K100 is also included with five Module-2s. This additional 10K100 will perform the housekeeping tasks such as loading programs, communicating with the QC40 and keeping up with the different stages of data processing which would be done by different modules. In short, the additional 10K100 is a manager.

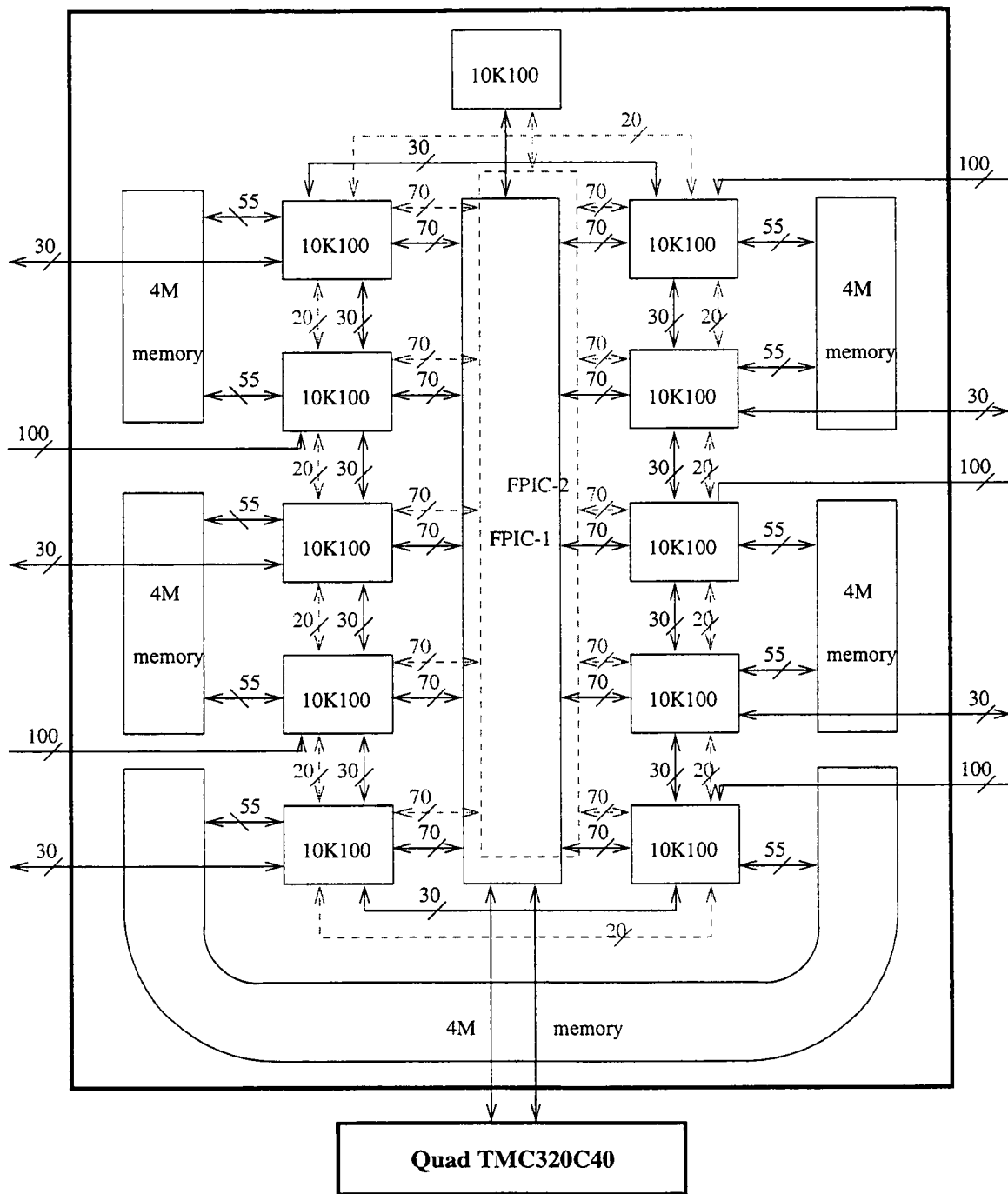


Figure 4.5: Detailed MSP Based on Splash Architecture Using Module-2

4.5 Recommendation

This discussion leads us to a choice of architecture. What we need is, an architecture that is flexible and can also be as fast as the technology allows it to be. These two primary constraints gives a designer a freedom to make choices in accordance with the application. We recommend the architecture discussed in the section 4.4.2 as it is based on module 2 which has hardwired interconnections between different components of it but at the same time inter-module connections are programmable. This gives a balanced architecture that has fast intra-module interconnections and flexible inter-module interconnections. The module-3 has been ignored here because it would require a ten-port memory, which would be a daunting task and could end-up becoming a bottleneck.

Chapter 5

Software Development for MSP

Programming malleable processors is very much like designing ASIC (Application Specific Integrated Circuit). This task could be accomplished by the following methods:

- (1) Using Hardware Description Languages such as Verilog HDL and VHDL.
- (2) Graphically representing system using softwares such as COSSAP digital signal processing development system, which allows user to enter the design at concept, algorithm, and architecture level. Then one can convert it to the choice of HDL.
- (3) Describing the system in equations and generating HDL of choice.

(4) Programming in high level computer programming language such as C and creating netlist from it.

(5) The old fashioned way, by connecting each and every connection manually.

Manual programming of FPGAs defeats the purpose since the FPGAs are used to speed up the design cycle of the project and manual programming is a cumbersome and slow process.

The software discussed and developed in this chapter was suggested by Jeff Weaver of Maxwell Technologies, as shown in the Figure 5.1, who is in charge of the

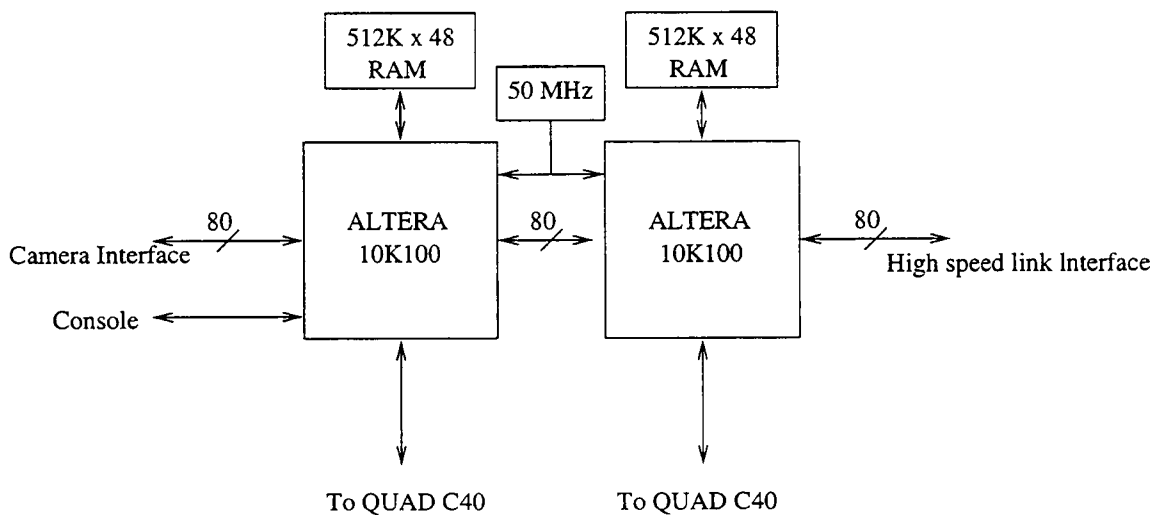


Figure 5.1: *MSP-0 made by Great River Technologies.*

overall software design of the Malleable Signal Processor. The suggested problem for programming was region-of-interest. The created netlist will be downloaded in MSP-0, which is an experimental and first version of MSP.

For programming MSP we chose VHDL. This was based on my familiarity with VHDL, our desire to be independent of the software development toolset, and the FPGAs used.

5.1 Problem Definition

The image delivered to a channel of the MSP consists of 12-bit infrared (IR) data arranged as a 480 (row) x 640 (column) frame. The output for the region-of-interest (ROI) program will be the data that is in the specified ROI.

Two memory storage banks MEMA (memory-A) and MEMB (memory-B) are connected to MSP-0 as shown in the Figure 5.1. Each memory bank can store the whole 480 x 640 frame of nonuniformity corrected data in real time. The even number frame is stored in MEMA and the odd number frame in MEMB. In other words, IR image frames are loaded in MEMA and MEMB in alternate manner. When MEMB is being loaded with frame number N from the IR camera, data of frame N-1 is read and sent to the Fiber Channel interface (which is also 12-bits). During the next cycle, the N+1 frame data is loaded in MEMA and data from MEMB (frame N) is being sent to the same Fiber Channel interface. This technique of constantly shifting the data destination between MEMA and MEMB, which is called double buffering, permits the MSP-0 to keep up

with real time non-uniformity correction and at the same time outputting the data to the Fiber Channel interface.

The problem suggested to me is to design an "ROI server". The MSP-0 will have four Fiber Channel interfaces each with a dedicated ROI server. The ROI server will make sure that only the defined region will be sent to the Fiber Channel interface to which it is connected. All Fiber Channel interfaces are identical except the region-of-interest could be different and its output go to different Fiber Channel interfaces.

5.2 ROI Algorithm

The 1 2-bit IR image is stored in the memory sequentially in a row by row fashion. Pixels 0 to 639 of row 0 of the image are stored sequentially in the first 640 memory locations, pixel 0 to 639 of row 1 are stored sequentially in the next 640 memory locations and so on up to 479th row.

The region-of-interest is defined by *upper*, *left*, *bottom*, and *right* as shown in the Figure 5.2. Each of these variables provides information on the boundaries of the region-of-interest. It is assumed that $0 \leq \text{Upper} \leq \text{Bottom} < 479$ and $0 \leq \text{left} \leq \text{right} < 640$. The ROI server does not check these conditions.

The flowchart for ROI server is shown in the Figure 5.3, which presents the big picture of the working of the ROI server. When the ROI server is downloaded to MSP-0,

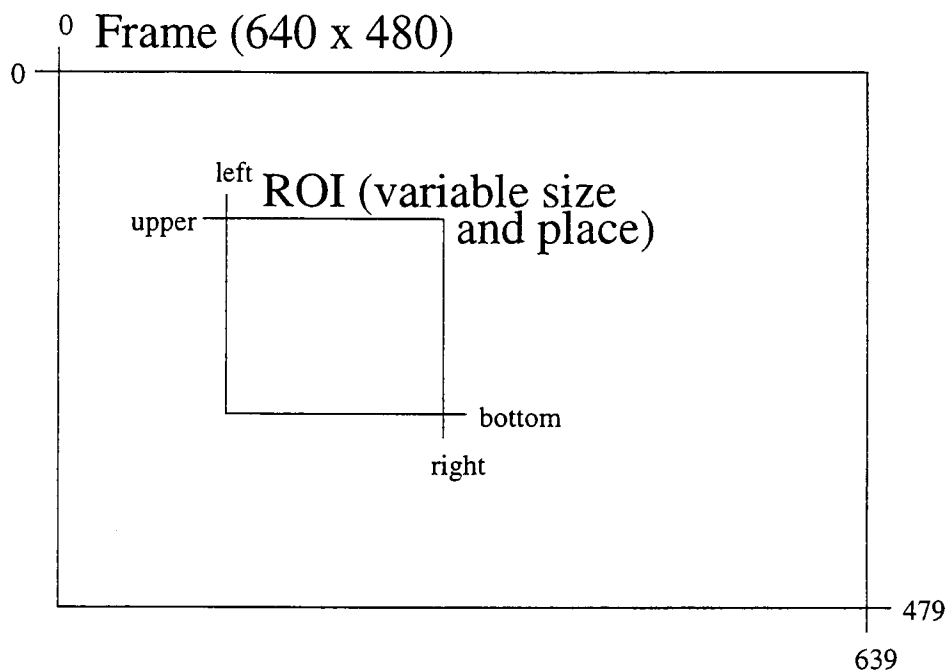


Figure 5.2: *Region-of-Interest*.

the circuit requires a *reset*. The most important things happen when the *reset* is HIGH are: MEMA is selected as the next memory for ROI server access, and the output *done* (which indicates whether the circuit is ready or not for new region-of-interest processing) is set to HIGH, implying the server is ready.

The ROI server can be used only after initial *reset* which sets *done* to high. After *reset* is set to LOW, the ROI server is ready to pull out ROI data from MEMA. Setting *start* to HIGH starts the process of pulling the ROI data from the memory. Variable *done* is set to ZERO and the ROI boundary-marking variables *upper*, *left*, *bottom*, and *right* are stored in a register for future use. During the same first clock tick, the starting address of

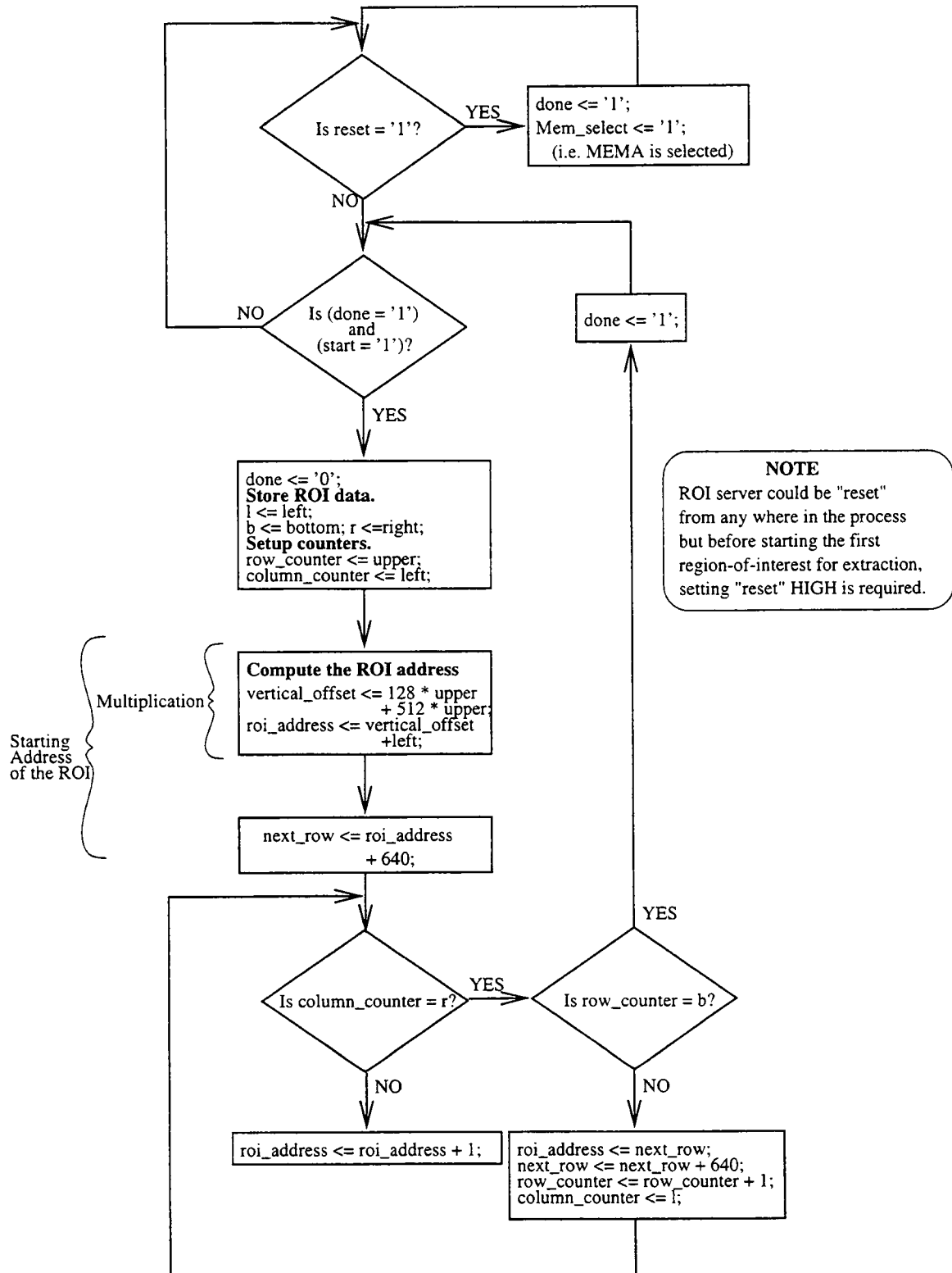


Figure 5.3: Algorithm Implemented in VHDL for ROI server.

the region-of-interest in the memory is computed. At present, the starting address of the image in both MEMA and MEMB is set to the 00000 HEX, which could be changed as per requirements. This starting address is placed on the address bus of the memory. The row-counter is set to *upper*, and the column-counter is set to *left*. The next address is computed by adding one in the present address, and the address of the next row is computed by adding 640 in the present address.

At the next clock tick, the computed next address is placed on the address bus of MEMA and a new next address is computed by incrementing the present address by one and the column counter is incremented by one. This process goes on until the column counter is equal to the ROI boundary *right*. When the column counter is equal to the ROI boundary *left*, the address of next row is assigned to the next address. A new next line address is computed by adding 640 in the present next line address. The column-counter is set to the ROI boundary *left* and the row-counter is incremented by one.

When the row-counter is equal to the ROI boundary *bottom* and column-counter equal to the ROI boundary *right*, all addresses of the region-of-interest have been placed on the address bus of the memory. The variable *done is* set to HIGH and MEMB would be selected for the next access, and the ROI server waits for the variable *start* to go HIGH. The cycle explained above goes on with MEMA and MEMB being accessed alternately.

5.3 Finer Details of Programming

The VHDL source code for the ROI server is given in appendix A. The connections between different VHDL programs is shown in the Figure 5.4. The ROI server needs a multiplier to find the starting address of the region-of-interest and comparators to check whether the end of the region-of-interest has been reached or not. These algorithms are discussed below.

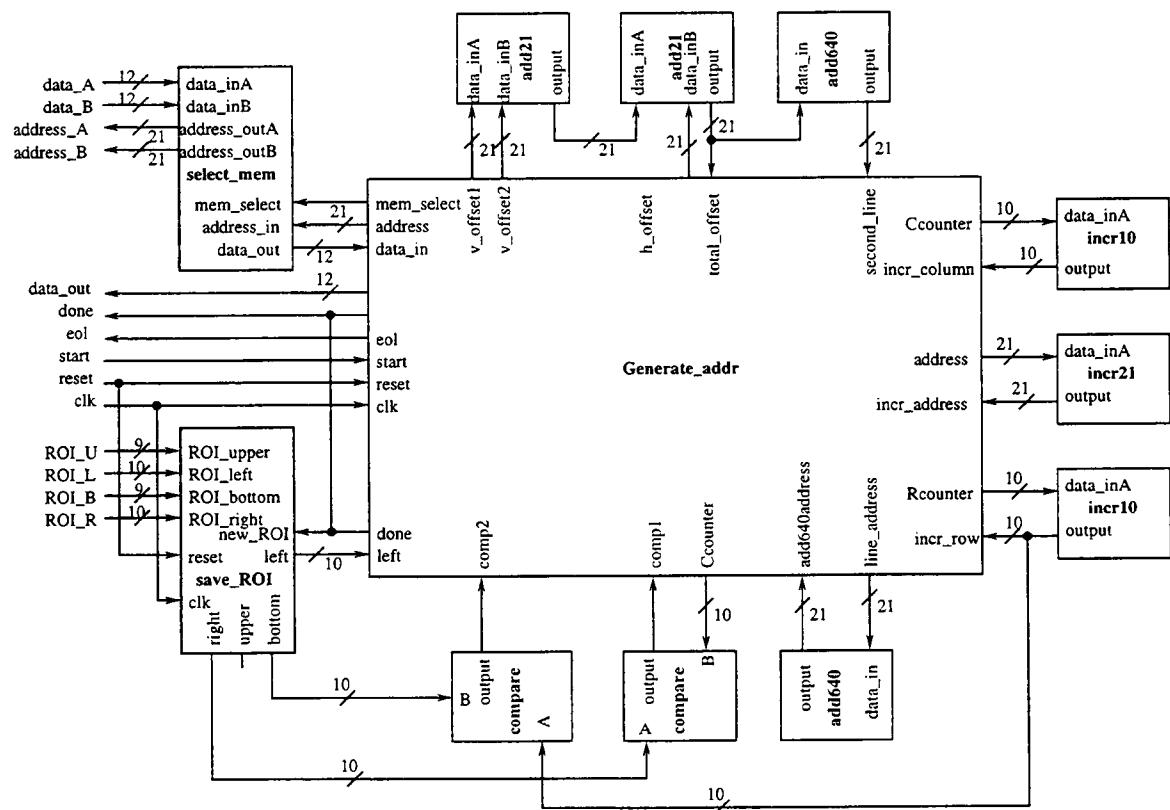


Figure 5.4: Connections of different Modules of ROI server.

5.3.1 Shift-and-Add multiplication Algorithm

The ROI server needs a multiplier to compute the starting address of region-of-interest. The multiplication between 640 and the ROI boundary *upper* is implemented using a "shift-and-add" multiplication algorithm. A further fine-tuning of the "shift-and-add" algorithm is done by utilizing the fact that 640 is $29 + 27$, i.e. binary "101000000". This also means that only one addition of a 9-bit left-shifted ROI boundary *upper* and 7-bit left-shifted ROI boundary *upper* is needed to achieve the same effect as multiplying the ROI boundary *upper* with 640. Realization of this fact made multiplication an easy task, which was performed by designing a ripple-carry adder using a FOR loop.

5.3.2 Comparator

A 10-bit comparator was designed to compare the row-counter and column-counter with their end conditions *bottom* and *right* respectively. The 10-bit inputs A, and B are compared bit-by-bit, starting from the MSB and rippling the result of each bit-comparison to the next lower significant bit comparison. After comparing the Least Significant Bits (LSB), the result is returned. The result is '1' if the condition $A > B$ is satisfied; otherwise, the result is '0'.

5.4 Simulation Results

Figure 5.5 illustrates the results of simulation of region-of-interest code. The region-of-interest is simulated for a 3x3 pixel region of the image defined by *upper* = 2, *left* = 4, *bottom* = 4 and *right* = 6. One could observe that after *start* signal is set HIGH, *done* goes LOW and after every three clock cycles an *eol* (end-of-line) signal is set HIGH for one clock cycle. When the addresses of region-of-interest are generated *done* goes HIGH, indicating that the ROI-server is ready for next region-of-interest. The ROI-server generates addresses continuously without weighting for the *eol* is reached or not. One could further observe that these results are directed towards MEMA or MEMB automatically.

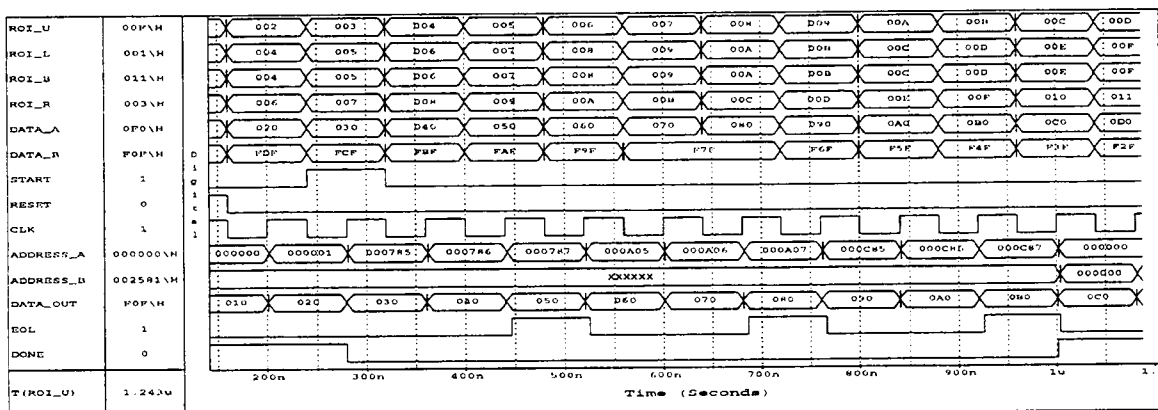
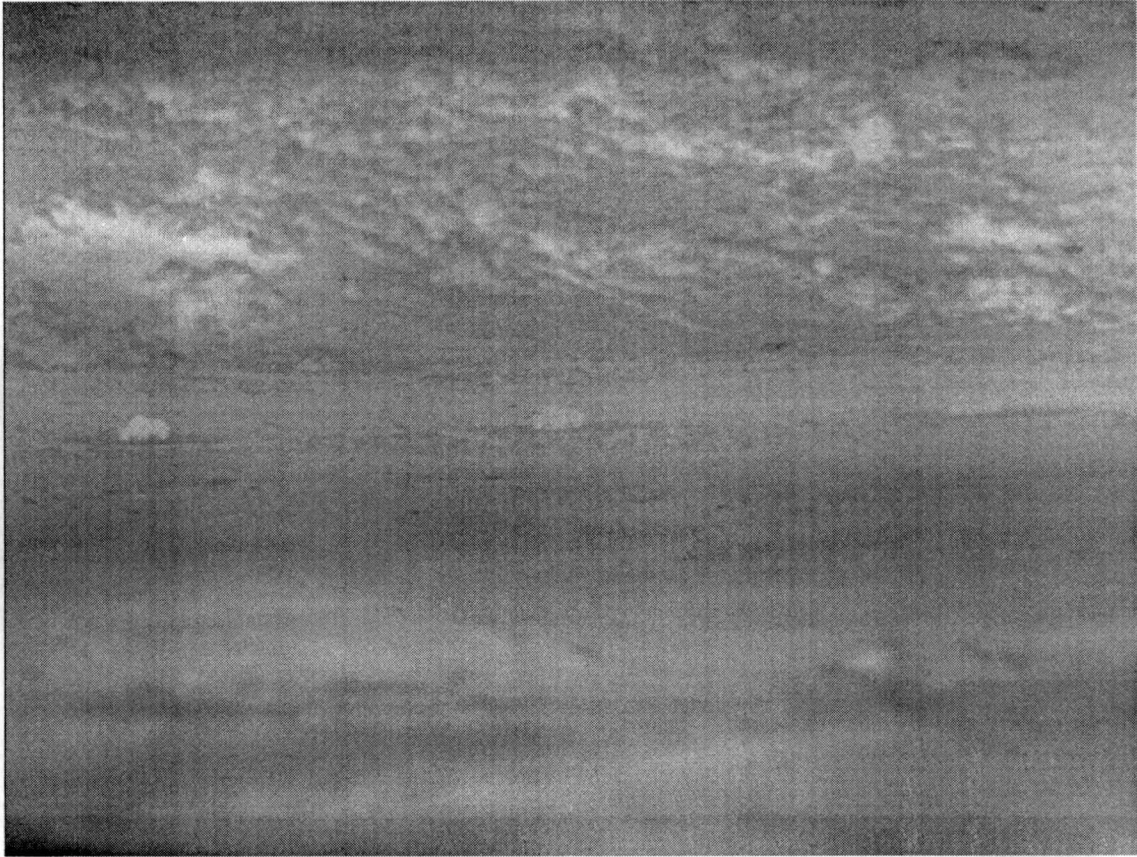
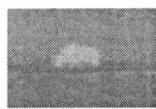


Figure 5.5 : Simulation Results of the ROI-server.

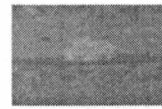
The ROI server was further tested by simulating it for an infrared image shown in the Figure 5.6 (a). This image has two region-of-interest. The ROI server was simulated with these two region-of-interest data in succession. The *upper, left, bottom, and right* coordinates of the region (a) are 208, 268, 265, 357 and that of region (b) are 212, 40, 268, 125. The addresses generated by the ROI server are listed in the Appendix B. These addresses perfectly correspond with the offset-addresses of the image. Further, the signals *eol, done*, and the memory selected for data retrieval also match with the intended signal patterns. Figures 5.6 (b) and 5.6 (c) show the regions-of-interest whose coordinates are sent to the ROT server.



(a): *Input Infrared Image.*



(b): ROI 208,268,265,357.



(c): ROI 212,40,268,125

Figure 5.6: *Simulation for Infrared Image*

Chapter 6

Summary, Conclusions and Future Work

In this research, we studied malleable architectures already designed, manufactured, and tested by other institutions and one proposed candidate architecture for Phillips Laboratories. The chosen architecture was implemented physically by GRT. The software development process was also discussed.

A survey of programmable devices available in the market and how they are used in malleable machines was done. It was found that the available programmable devices support a wide range of applications. FPGAs by Xilinx and Altera. Further, programmable interconnect devices were studied as the interconnections between different programmable logic devices may also need to be programmable to achieve maximum flexibility. We also discussed existing and functioning malleable architectures such as Splash 2 and PAM. Splash 2 and PAM are well tested architectures and were

used as one of the basis of this research. The signal processing done on malleable machines was also studied as the final application of this project is primarily for image processing.

The system requirements and the preliminary architecture proposed by Phillips Laboratory was reviewed. A few probable and improved architectures that meet the constraints were introduced. Details of the different architectures and their pros and cons were discussed. At the same time, an attempt was made to achieve a balance between the unprogrammable programmable connections between FPGAs and memory in these architectures and a balanced architecture was proposed for physical copy.

A software example that could be used on a Malleable Signal Processor was designed and sent to Phillips Laboratories for testing. The VHDL source code was written to implement a region-of-interest function between the MSP and a much larger fusion processor that is intended to make sense out of these pictures. This region-of-interest algorithm has been synthesized and simulated before sending it for testing to the Phillips Laboratories.

Malleable systems provide considerable improvement in performance for many types of applications. With conventional hardware, change or improvement in hardware architecture design without replacing it is not possible. Using malleable processors for the same application would give the designer flexibility needed for future changes at a cost of lower clock speed. The primary reason for lower clock speed in FPGAs than an

equivalent feature size microprocessor is that the signals traveling through nets of FPGAs must pass through slow switches of the interconnection matrix which connect two logic elements of FPGAs are not required in microprocessors. Furthermore, the number of equivalent gates that could be crammed on a single chip is much less in malleable chip than on conventional CMOS chip (the ratio is about 1:8. Altera FLEX10K250 compared with Pentium-II). In the most of the cases this drawback is overcome by the application specific nature of the downloaded designs, parallel and/or pipelined architectures, and changing the architecture of malleable processor with change in applications.

Present day FPGAs are extremely expensive compared to microprocessors (about 4.5 times more expensive per-gate). The reason behind such a high price is that the FPGA feature size is behind present day technology by 1 to 2 years and they are not sold in as large quantities as microprocessors.

Another important consideration for future development of malleable processor is the Moore's law, which states that circuit density of semiconductors has and will continue to double every eighteen months. When Moore's law bottoms out due to either economical reasons (Ross 1995) and/or due to the limitations imposed by the realities of physics, the feature size would stop shrinking and the improvements in performance of microprocessor will be based less on decreased feature size and more on improved architecture. This emerging scenario clearly indicates that changeable or malleable hardware has potential of producing an answer to future needs. The combination of a

simple microprocessor with a malleable processor is a solution to the present problem of microprocessors becoming obsolete every five years.

The MSPs designed in previous chapters are for generic processing of specific inputs. As the previous discussion clearly shows that design of an MSP is a balancing act, there are no clear winners for MSP. Timing, flexibility, hardware and software complexity, power consumption etc. are factors one has to ponder to get a balanced design representing global minima. The architecture shown in Figure 4.1 using Module-2 is the closest to the global minima. This architecture is not extremely complex, as is the case with the architecture in figure 4.3, which requires a ten-port memory. Unlike Module-1, Module-2 is flexible and the power consumption remains nearly the same as that of Module-1. The recommended design also has a balance between direct FPGA-to-FPGA connections and connections through FPIDs which gives the programmer complete control over data transfer for processing. Further, designing a module, replicating it, and connecting them as tiles to achieve the recommended architecture is an easy and efficient way of manufacturing the MSP.

Malleable systems show a lot of promise but at the same time a lot is left to be done to fulfill the promise. Pushing the concept of having a balance between a malleable system and present day microprocessors in a computer further, inclusion of reprogrammable functional units would provide very high bandwidth application-specific coprocessors in a application-generic manner. This amalgamation can deliver custom designed accelerators for non-predefined functions and for critical applications can

provide custom hardware which otherwise will not be available due to its lack of cost-effectiveness.

List of References

List of References

- [1] Hennessy, J. and D. Patterson, *Computer Architecture - A Quantitative Approach*, San Francisco, CA: Morgan Kaufmann Publishers (1996).
- [2] De Micheli, G., *Synthesis and Optimization of Digital Circuits*, New York: McGraw-Hill Publishers (1994).
- [3] Buell, D., Arnold, J. and W. Kleinfelder, *SPLASH-2: FPGAs in a Custom Computing Machine*, Los Alamitos, CA: IEEE Computer Society Press (1996).
- [4] Vuillemin, J., Bertin, P., Roncin, D., Shand, M., Touati, H. and P. Boucard, "Programmable Active Memories: Reconfigurable Systems Come of Age", *IEEE Transactions on VLSI Systems*, Vol 4, No. 1, (March 1996).
- [5] Casselman, S. "Virtual Computing and The Virtual Computer", *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 43-48, Napa, CA (April 1993).
- [6] Huck, S., Borriello, G. "Pin assignment for multi-FPGA Systems", *IEEE Transactions on Computer-Aided Design of Integrated Circuits & systems*, Vol. 16, No. 9, pp. 956-964, (September 1997).

[7] Ross, Philip E. "Moore's Second Low", *Forbe's*, pp. 116-117, (March 25, 1995).

[8] Altera Corporation. "1996 Data Book", (June 1996).

[9] Xilinx Incorporation. "The Programmable Logic Data Book", (1997).

Appendices

Appendix A

Program Listing

In this appendix VHDL code generated for the region-of-interest server are listed.

```
-- roi.vhd
-- Source code by Amiya A. Chokhavala
-- Region-of-interest server.
-- The region-of-interest is defined by the inputs ROI_U, ROI_R,
-- ROI_B, and ROI_L based on present memory access cycle ROI server
-- will put address on address_A or address_B and fetch the data
-- from data_A or data_B.
-- After start signal is set HIGH, done goes LOW and after every
-- three clock cycles an eol (end-of-line) signal is set HIGH for one
-- clock cycle. When all the addresses of region-of-interest are
-- generated done goes HIGH, indicating that the ROI-server is ready
-- for next region-of-interest. The ROI-server generates addresses
-- continuously without weighting for the eol is reached or not.

library ieee;
use ieee.std_logic_1164.all;

entity roi is
port(
    ROI_U      : in std_logic_vector(8 downto 0);
    ROI_R      : in std_logic_vector(9 downto 0);
    ROI_B      : in std_logic_vector(8 downto 0);
    ROI_L      : in std_logic_vector(9 downto 0);
    data_A     : in std_logic_vector(11 downto 0);
    data_B     : in std_logic_vector(11 downto 0);
    start      : in std_logic;
    reset      : in std_logic;
    clk        : in std_logic;

    address_A  : out std_logic_vector(20 downto 0);
    address_B  : out std_logic_vector(20 downto 0);
    data_out   : out std_logic_vector(11 downto 0);
    eol        : out std_logic;
    done       : out std_logic
);

end roi;
```

architecture structural of roi is

```
component select_mem
port(
    signal Mem_select : in std_logic;
    signal data_inA   : in std_logic_vector(11 downto 0);
    signal data_inB   : in std_logic_vector(11 downto 0);
    signal address_in : in std_logic_vector(20 downto 0);

    signal data_out      : out std_logic_vector(11 downto 0);
    signal address_outA  : out std_logic_vector(20 downto 0);
    signal address_outB  : out std_logic_vector(20 downto 0) );
end component;

component save_ROI
port( signal ROI_upper : in std_logic_vector(8 downto 0);
      signal ROI_left  : in std_logic_vector(9 downto 0);
      signal ROI_bottom : in std_logic_vector(8 downto 0);
      signal ROI_right  : in std_logic_vector(9 downto 0);
      signal new_ROI    : in std_logic;
      signal reset      : in std_logic;
      signal clk        : in std_logic;

      signal upper      : out std_logic_vector(9 downto 0);
      signal left       : out std_logic_vector(9 downto 0);
      signal bottom     : out std_logic_vector(9 downto 0);
      signal right      : out std_logic_vector(9 downto 0) );
end component;

component add21
port( signal data_inA : in std_logic_vector(20 downto 0);
      signal data_inB : in std_logic_vector(20 downto 0);
      signal output    : out std_logic_vector(20 downto 0) );
end component;

component incr_21
port( signal data_inA : in std_logic_vector(20 downto 0);
      signal output    : out std_logic_vector(20 downto 0) );
end component;

component add10
port( signal data_inA : in std_logic_vector(9 downto 0);
      signal data_inB : in std_logic_vector(9 downto 0);
      signal output    : out std_logic_vector(9 downto 0) );
end component;

component add640
port( signal data_in : in std_logic_vector(20 downto 0);
      signal output  : out std_logic_vector(20 downto 0) );
end component;

component incr_10
port( signal data_inA : in std_logic_vector(9 downto 0);
      signal output    : out std_logic_vector(9 downto 0) );
end component;

component cmp10bit
port(
```

```

    A, B : in std_logic_vector(9 downto 0);
    output      : out std_logic      );
end component;

component Generate_addr
port(
    signal ROI_upper : in std_logic_vector(8 downto 0);
    signal ROI_left  : in std_logic_vector(9 downto 0);

    signal left      : in std_logic_vector(9 downto 0);

    signal start      : in std_logic;    -- starts generating
addresses           : in std_logic;    -- based on present ROI values.

    signal reset      : in std_logic;

    signal comp1      : in std_logic;
    signal comp2      : in std_logic;
    signal Mem_select : out std_logic;
    signal Ccounter   : out std_logic_vector(9 downto 0);
    signal Rcounter   : out std_logic_vector(9 downto 0);
    signal line_address : out std_logic_vector(20 downto 0);
    signal address     : out std_logic_vector(20 downto 0);

    signal v_offset1 : out std_logic_vector(20 downto 0);
    signal v_offset2 : out std_logic_vector(20 downto 0);
    signal h_offset   : out std_logic_vector(20 downto 0);
    signal total_offset : in std_logic_vector(20 downto 0);
    signal second_line : in std_logic_vector(20 downto 0);

    signal incr_address : in std_logic_vector(20 downto 0);
    signal incr_column  : in std_logic_vector(9 downto 0);

    signal add640address : in std_logic_vector(20 downto 0);
    signal incr_row       : in std_logic_vector(9 downto 0);
    signal data_in        : in std_logic_vector(11 downto 0);
    signal data_out       : out std_logic_vector(11 downto 0);
    signal eol            : out std_logic;
    signal done           : out std_logic;
    signal clk           : in std_logic
);
end component;

```

```

    signal comp1, comp2      : std_logic;
    signal Mem_select        : std_logic;
    signal Ccounter          : std_logic_vector(9 downto 0);
    signal Rcounter          : std_logic_vector(9 downto 0);
    signal line_address      : std_logic_vector(20 downto 0);
    signal address           : std_logic_vector(20 downto 0);

    signal u, l, b, r : std_logic_vector(9 downto 0);

    signal v_offI, v_offII : std_logic_vector(20 downto 0);
    signal v_off, h_off    : std_logic_vector(20 downto 0);
    signal tot_off, IInd_line: std_logic_vector(20 downto 0);

    signal inc_addr          : std_logic_vector(20 downto 0);
    signal next_row_addr     : std_logic_vector(20 downto 0);

```

```

    signal inc_column : std_logic_vector(9 downto 0);
    signal inc_row    : std_logic_vector(9 downto 0);
    signal data_in    : std_logic_vector(11 downto 0);
    signal nexts      : std_logic;

begin

done <= nexts;

U1   :   select_mem
port map (Mem_select, data_A, data_B, address, data_in,
         address_A, address_B);

U2   :   save_ROI
port map (ROI_U, ROI_L, ROI_B, ROI_R, nexts, reset, clk,
         u, l, b, r);

U3   :   add21
port map (v_offI, v_offII, v_off);

U4   :   add21
port map (v_off, h_off, tot_off);

U5   :   add640
port map (tot_off, IInd_line);

U6   :   incr_21
port map (address, inc_addr);

U7   :   incr_10
port map (Ccounter, inc_column);

U8   :   add640
port map (line_address, next_row_addr);

U9   :   incr_10
port map (Rcounter, inc_row);

U10  :   cmp10bit
port map (r, Ccounter, comp1);

U11  :   cmp10bit
port map (inc_row, b, comp2);

U12  :   Generate_addr
port map (ROI_U, ROI_L, l, start, reset, comp1, comp2,
         Mem_select, Ccounter, Rcounter, line_address, address,
         v_offI, v_offII, h_off, tot_off, IInd_line, inc_addr,
         inc_column, next_row_addr, inc_row, data_in, data_out,
         eol, nexts, clk);

end structural;

```

```

-- select_mem.vhd
-- Source code by Amiya A. Chokhavala
-- When mem_select is HIGH then MEMA is enabled otherwise MEMB is
-- enabled.

library ieee;
use ieee.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity select_mem is
port(
    Mem_select    : in std_logic;
    data_inA      : in std_logic_vector(11 downto 0);
    data_inB      : in std_logic_vector(11 downto 0);
    address_in    : in std_logic_vector(20 downto 0);

    data_out      : out std_logic_vector(11 downto 0);
    address_outA  : out std_logic_vector(20 downto 0);
    address_outB  : out std_logic_vector(20 downto 0)    );

end select_mem;

architecture behavior of select_mem is
begin

selection:process(Mem_select, address_in, data_inA, data_inB)
begin

    if (Mem_select = '1') then
        address_outA <= address_in;
        data_out <= data_inA;
    else
        address_outB <= address_in;
        data_out <= data_inB;
    end if;

end process selection;

end behavior;

```

```

-- Generate_addr.vhd
-- Source code by Amiya A. Chokhavala
-- This is main program that does most of the house keeping and
-- keeps up with different components of ROI server.

library ieee;
use ieee.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity Generate_addr is
port(
    ROI_upper    : in std_logic_vector(8 downto 0);

```

```

ROI_left      : in std_logic_vector(9 downto 0);
left          : in std_logic_vector(9 downto 0);

start        : in std_logic;           -- starts generating addresses
reset        : in std_logic;           -- based on present ROI values.

comp1        : in std_logic;
comp2        : in std_logic;
Mem_select   : out std_logic;
Ccounter     : out std_logic_vector(9 downto 0);
Rcounter     : out std_logic_vector(9 downto 0);
line_address : out std_logic_vector(20 downto 0);
address      : out std_logic_vector(20 downto 0);

v_offset1    : out std_logic_vector(20 downto 0);
v_offset2    : out std_logic_vector(20 downto 0);
h_offset     : out std_logic_vector(20 downto 0);
total_offset : in std_logic_vector(20 downto 0);
second_line  : in std_logic_vector(20 downto 0);

incr_address : in std_logic_vector(20 downto 0);
incr_column  : in std_logic_vector(9 downto 0);

add640address : in std_logic_vector(20 downto 0);
incr_row      : in std_logic_vector(9 downto 0);
data_in       : in std_logic_vector(11 downto 0);
data_out      : out std_logic_vector(11 downto 0);
eol           : out std_logic;
done         : out std_logic;
clk          : in std_logic
);
end Generate_addr;

```

architecture behavior of Generate_addr is

```

signal p_Mem_select, n_Mem_select : std_logic;
signal p_address, n_address       : std_logic_vector(20 downto 0);
signal p_done, n_done             : std_logic;
signal p_countR, n_countR        : std_logic_vector(9 downto 0);
signal p_countC, n_countC        : std_logic_vector(9 downto 0);
signal p_line_addr, n_line_addr   : std_logic_vector(20 downto
0);
signal p_data, n_data             : std_logic_vector(11 downto 0);
--signal p_eol, n_eol             : std_logic;

begin

v_offset1 <= "000" & ROI_upper & "000000000";
v_offset2 <= "00000" & ROI_upper & "0000000";
h_offset <= "000000000000" & ROI_left(9 downto 0);

Mem_select <= p_Mem_select;
line_address <= p_line_addr;
address <= p_address;
Ccounter <= p_countC;
Rcounter <= p_countR;

```

```

data_out <= p_data;

--eol <= p_eol;
done <= p_done;

clock:process(clk)

begin

  if (clk'event and clk = '1') then
    p_Mem_select <= n_Mem_select;
    p_address(20 downto 0) <= n_address(20 downto 0);
    p_line_addr(20 downto 0) <= n_line_addr(20 downto 0);
    p_done <= n_done;
    p_countR(9 downto 0) <= n_countR(9 downto 0);
    p_countC(9 downto 0) <= n_countC(9 downto 0);
    p_data <= n_data;
    -- p_eol <= n_eol;
  end if;
end process clock;

xxx:process(p_Mem_select, p_address, p_line_addr, reset, p_done,
  start, ROI_upper, ROI_left, total_offset, second_line, left,
  comp1, comp2, p_countC, p_countR, incr_row, incr_column,
  incr_address, add640address)

begin

n_Mem_select <= p_Mem_select;
n_address(20 downto 0) <= incr_address(20 downto 0);
n_line_addr(20 downto 0) <= p_line_addr(20 downto 0);
n_done <= p_done;
n_countR(9 downto 0) <= p_countR(9 downto 0);
n_countC(9 downto 0) <= incr_column(9 downto 0);
n_data <= data_in;
--n_eol <= p_eol;
eol <= '0';

if (reset = '1') then
  n_Mem_select <= '1';
  n_address <= "00000000000000000000";
  n_done <= '1';
  n_countR <= "0000000000";
  n_countC <= "0000000000";
  n_line_addr <= "00000000000000000000";
  eol <= '0';
else
  if ((p_done = '1') and (start = '1')) then
    n_Mem_select <= p_Mem_select;
    n_done <= '0';
    n_countR <= '0' & ROI_upper;
    n_countC <= ROI_left;
    n_address <= total_offset;
    n_line_addr <= second_line;
  else
    if p_done = '0' then
      if (comp1 = '0') then
        eol <= '1';
      end if;
    end if;
  end if;
end if;
end process xxx;

```

```

        n_countC <= left;
        n_countR <= incr_row;
        n_address <= p_line_addr;
        n_line_addr <= add640address;
    end if;
    if ((comp2 = '1') and (comp1 = '0')) then
        n_done <= '1';
        n_Mem_select <= (p_Mem_select xor '1');
        n_address <= "000000000000000000000000";
        n_countR <= "0000000000";
        n_countC <= "0000000000";
        n_line_addr <= "000000000000000000000000";
        eol <= '1';
    end if;
end if;
end if;
end if;
end process xxx;

end behavior;

```

```

-- add21.vhd
-- Source code by Amiya A. Chokhavalala
-- The ROI server needs a multiplier to compute the starting address
-- of region-of-interest. The multiplication between 640 and the ROI
-- boundary upper is implemented using a "shift-and-add"
-- multiplication algorithm. A further fine-tuning of the
-- nshift-and-add~ algorithm is done by utilizing the fact that 640
-- is  $2^9 + 2^7$ , i.e. binary "1010000000". This also means that only one
-- addition of a 9-bit left-shifted ROI boundary upper and 7-bit
-- left-shifted ROI boundary upper is needed to achieve the same effect
-- as multiplying the ROI boundary upper with 640. Realization of this
-- fact made multiplication an easy task, which was performed by
-- designing a ripple-carry adder using a FOR loop.
-- This is implemented by using two add12.vhd and one add640.vhd.

```

```

library ieee;
use ieee.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity add21 is
port( data_inA      : in std_logic_vector(20 downto 0);
      data_inB      : in std_logic_vector(20 downto 0);
      output        : out std_logic_vector(20 downto 0) );
end add21;

architecture behavior of add21 is
begin
addr_21_bit:process(data_inA, data_inB)
variable CARRY : std_logic;
begin
    carry := '0';
    for i in 0 to 20 loop

```

```

        output(i) <= data_inA(i) xor data_inB(i) xor carry;
        carry := (data_inA(i) and data_inB(i)) or
                 (data_inA(i) and carry) or
                 (carry and data_inB(i));
    end loop;
end process addr_21_bit;

end behavior;

```

```

-- add640.vhd
-- Source code by Amiya A. Chokhavala

```

```

library ieee;
use ieee.std_logic_1164.all;
use IEEE.std_logic_arith.all;

```

```

entity add640 is
port( data_in      : in std_logic_vector(20 downto 0);
      output      : out std_logic_vector(20 downto 0) );
end add640;

```

```

architecture behavior of add640 is
begin

```

```

    addr640:process(data_in)
    variable CARRY : std_logic;
    variable B : std_logic_vector(20 downto 0);

```

```

begin
    carry := '0';
    B := "0000000000001010000000";
    for i in 0 to 20 loop
        output(i) <= data_in(i) xor B(i) xor carry;
        carry := (data_in(i) and B(i)) or
                 (data_in(i) and carry) or
                 (carry and B(i));
    end loop;
end process addr640;

```

```

end behavior;

```

```

-- compare.vhd
-- Source code by Amiya A. Chokhavala
-- A 10-bit comparator was designed to compare the row-counter and
-- column-counter with their end conditions bottom and right
-- respectively. The 10-bit inputs A, and B are compared bit-by-bit,
-- starting from the MSB and rippling the result of each bit-comparison
-- to the next lower significant bit comparison. After comparing the
-- Least Significant Bits (LSB), the result is returned. The result is
-- '1' if the condition A > B is satisfied; otherwise, the result is
-- '0'.

```

```

library ieee;
use ieee.std_logic_1164.all;
use IEEE.std_logic_arith.all;

```

```

entity cmp10bit is
port(
    A, B : in std_logic_vector(9 downto 0);
    output : out std_logic );
end cmp10bit;

architecture behavior of cmp10bit is

function comp(Ax, Bx : std_logic_vector(9 downto 0))
return std_logic is
variable finish : std_logic;
begin
    finish := '0';
    for i in 9 downto 0 loop
        finish := (Ax(i) and not Bx(i)) or finish;
    end loop;
    return(finish);
end;

begin
compare10bit:process(A, B)

begin

    output <= comp(A, B);

end process compare10bit;

end behavior;

```

```

-- save_ROI.vhd
-- Source code by Amiya A. ChoRhavala
-- The region-of-interest needs to be stored for future use and to
-- reduce the load from interfacing program.

```

```

library ieee;
use ieee.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity save_ROI is
port( ROI_upper : in std_logic_vector(8 downto 0);
ROI_left : in std_logic_vector(9 downto 0);
ROI_bottom : in std_logic_vector(8 downto 0);
ROI_right : in std_logic_vector(9 downto 0);
new_ROI : in std_logic;
reset : in std_logic;
clk : in std_logic;

upper : out std_logic_vector(9 downto 0);
left : out std_logic_vector(9 downto 0);
bottom : out std_logic_vector(9 downto 0);
right : out std_logic_vector(9 downto 0) );

end save_ROI;

```

architecture behavior of save_ROI is

```
    signal pU, nU, pL, nL, pB, nB, pR, nR    : std_logic_vector(9 downto
0);

begin

upper <= pU;
bottom <= pB;
left <= pL;
right <= pR;

clock:process(clk)
begin

    if (clk'event and clk = '1') then
        pU(9 downto 0) <= nU(9 downto 0);    -- Store ROI values at the
        pL(9 downto 0) <= nL(9 downto 0);    -- start of new frame.
        pB(9 downto 0) <= nB(9 downto 0);
        pR(9 downto 0) <= nR(9 downto 0);
    end if;
end process clock;

store_ROI:process(new_ROI,ROI_upper,ROI_left,ROI_bottom,ROI_right,reset)
begin

if (reset = '1') then
    nU(9 downto 0) <= "0000000000";
    nL(9 downto 0) <= "0000000000";
    nB(9 downto 0) <= "0000000000";
    nR(9 downto 0) <= "0000000000";
else
    if (new_ROI = '1') then
        nU(9 downto 0) <= '0' & ROI_upper(8 downto 0);
        nL(9 downto 0) <= ROI_left(9 downto 0);
        nB(9 downto 0) <= '0' & ROI_bottom(8 downto 0);
        nR(9 downto 0) <= ROI_right(9 downto 0);
    end if;
end if;
end process store_ROI;

end behavior;
```

Appendix B

Synthesis Results

The addresses generated while synthesizing the ROI server are given below. The first region-of-interest shown in the Figure 5.6(a) had its coordinates *upper* = 208, *left* = 268, *bottom* = 265, and *right* = 357. Each discontinuity indicates *eol* (end-of-line). These addresses are passed to the address busses of MEMA.

2090C	2090D	2090E	2090F	20910	20911	20912	20913	20914	20915
20916	20917	20918	20919	2091A	2091B	2091C	2091D	2091E	2091F
20920	20921	20922	20923	20924	20925	20926	20927	20928	20929
2092A	2092B	2092C	2092D	2092E	2092F	20930	20931	20932	20933
20934	20935	20936	20937	20938	20939	2093A	2093B	2093C	2093D
2093E	2093F	20940	20941	20942	20943	20944	20945	20946	20947
20948	20949	2094A	2094B	2094C	2094D	2094E	2094F	20950	20951
20952	20953	20954	20955	20956	20957	20958	20959	2095A	2095B
2095C	2095D	2095E	2095F	20960	20961	20962	20963	20964	20965
20B8C	20B8D	20B8E	20B8F	20B90	20B91	20B92	20B93	20B94	20B95
20B96	20B97	20B98	20B99	20B9A	20B9B	20B9C	20B9D	20B9E	20B9F
20BA0	20BA1	20BA2	20BA3	20BA4	20BA5	20BA6	20BA7	20BA8	20BA9
20BAA	20BAB	20BAC	20BAD	20BAE	20BAF	20BB0	20BB1	20BB2	20BB3
20BB4	20BB5	20BB6	20BB7	20BB8	20BB9	20BBA	20BBB	20BBC	20BBD
20BBE	20BBF	20BC0	20BC1	20BC2	20BC3	20BC4	20BC5	20BC6	20BC7
20BC8	20BC9	20BCA	20BCB	20BCC	20BCD	20BCE	20BCF	20BD0	20BD1
20BD2	20BD3	20BD4	20BD5	20BD6	20BD7	20BD8	20BD9	20BDA	20BDB
20BDC	20BDD	20BDE	20BDF	20BE0	20BE1	20BE2	20BE3	20BE4	20BE5
20E0C	20E0D	20E0E	20E0F	20E10	20E11	20E12	20E13	20E14	20E15
20E16	20E17	20E18	20E19	20E1A	20E1B	20E1C	20E1D	20E1E	20E1F
20E20	20E21	20E22	20E23	20E24	20E25	20E26	20E27	20E28	20E29
20E2A	20E2B	20E2C	20E2D	20E2E	20E2F	20E30	20E31	20E32	20E33
20E34	20E35	20E36	20E37	20E38	20E39	20E3A	20E3B	20E3C	20E3D
20E3E	20E3F	20E40	20E41	20E42	20E43	20E44	20E45	20E46	20E47
20E48	20E49	20E4A	20E4B	20E4C	20E4D	20E4E	20E4F	20E50	20E51
20E52	20E53	20E54	20E55	20E56	20E57	20E58	20E59	20E5A	20E5B
20E5C	20E5D	20E5E	20E5F	20E60	20E61	20E62	20E63	20E64	20E65

.....
.....

.....

297B4	297B5	297B6	297B7	297B8	297B9	297BA	297BB	297BC	297BD
297BE	297BF	297C0	297C1	297C2	297C3	297C4	297C5	297C6	297C7
297C8	297C9	297CA	297CB	297CC	297CD	297CE	297CF	297D0	297D1
297D2	297D3	297D4	297D5	297D6	297D7	297D8	297D9	297DA	297DB
297DC	297DD	297DE	297DF	297E0	297E1	297E2	297E3	297E4	297E5

The image in the Figure 5.6(a) has ended. The *done* is HIGH, indicating the ROI server is ready for the new image.

The region-of-interest shown in the Figure 5.6 (b) had its coordinates *upper* = 212, *left* = 40, *bottom* = 268, and *right* = 125. Each discontinuity indicates eol (end-of-line). These addresses are passed to the address busses of MEMB.

21228	21229	2122A	2122B	2122C	2122D	2122E	2122F	21230	21231
21232	21233	21234	21235	21236	21237	21238	21239	2123A	2123B
2123C	2123D	2123E	2123F	21240	21241	21242	21243	21244	21245
21246	21247	21248	21249	2124A	2124B	2124C	2124D	2124E	2124F
21250	21251	21252	21253	21254	21255	21256	21257	21258	21259
2125A	2125B	2125C	2125D	2125E	2125F	21260	21261	21262	21263
21264	21265	21266	21267	21268	21269	2126A	2126B	2126C	2126D
2126E	2126F	21270	21271	21272	21273	21274	21275	21276	21277
21278	21279	2127A	2127B	2127C	2127D				

214A8	214A9	214AA	214AB	214AC	214AD	214AE	214AF	214B0	214B1
214B2	214B3	214B4	214B5	214B6	214B7	214B8	214B9	214BA	214BB
214BC	214BD	214BE	214BF	214C0	214C1	214C2	214C3	214C4	214C5
214C6	214C7	214C8	214C9	214CA	214CB	214CC	214CD	214CE	214CF
214D0	214D1	214D2	214D3	214D4	214D5	214D6	214D7	214D8	214D9
214DA	214DB	214DC	214DD	214DE	214DF	214E0	214E1	214E2	214E3
214E4	214E5	214E6	214E7	214E8	214E9	214EA	214EB	214EC	214ED
214EE	214EF	214F0	214F1	214F2	214F3	214F4	214F5	214F6	214F7
214F8	214F9	214FA	214FB	214FC	214FD				

.....

.....

29E28	29E29	29E2A	29E2B	29E2C	29E2D	29E2E	29E2F	29E30	29E31
29E32	29E33	29E34	29E35	29E36	29E37	29E38	29E39	29E3A	29E3B
29E3C	29E3D	29E3E	29E3F	29E40	29E41	29E42	29E43	29E44	29E45
29E46	29E47	29E48	29E49	29E4A	29E4B	29E4C	29E4D	29E4E	29E4F
29E50	29E51	29E52	29E53	29E54	29E55	29E56	29E57	29E58	29E59
29E5A	29E5B	29E5C	29E5D	29E5E	29E5F	29E60	29E61	29E62	29E63
29E64	29E65	29E66	29E67	29E68	29E69	29E6A	29E6B	29E6C	29E6D
29E6E	29E6F	29E70	29E71	29E72	29E73	29E74	29E75	29E76	29E77
29E78	29E79	29E7A	29E7B	29E7C	29E7D				

The image in the Figure 5.6(b) has ended. The *done is* HIGH, indicating the ROI server is ready for the new image.

Vita

Amiya Chokhvala was borne in Wardha, a small town in central India. He spent first four years in Mumbai. Then he moved to Surat and studied at Saint Xavier's High School. He received his twelfth grade diploma in 1987. He did his B.Sc. (Bachelor of Science) in Physics with minor in Mathematics in 1991 and did D.C.A. (Post Graduate Diploma in Computer Science and Applications) in 1992 from South Gujarat University. He came to U.S.A. in 1994 and did BS in Electrical Engineering from University of Tennessee, Knoxville in December 1995. He finished MS in Electrical Engineering in May 1998 from University of Tennessee, Knoxville.