Performance Measure of Digital Image Processing on FPGA based Co-processor and DSP Processor

By

TANK PRANAV S.

(06MCE016)

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
INSTITUTE OF TECHNOLOGY
NIRMA UNIVERSITY OF SCIENCE & TECHNOLOGY
AHMEDABAD 382 481
MAY 2008

Major Project

On

Performance Measure of Digital Image Processing on FPGA based Co-processor and DSP Processor

Submitted in partial fulfillment of the requirements

For the degree of

Master of Technology in Computer Science & Engineering

By

Tank Pranav S.
(06MCE016)

Under Guidance of

Dr. S.N. Pradhan

Department Of Computer Science & Engineering
Institute Of Technology
Nirma University Of Science & Technology
Ahmedabad 382 481
May 2008

This is to certify that Dissertation entitled

Performance Measure of Digital Image Processing on FPGA based Co-processor and DSP Processor

Submitted by

Tank Pranav S.
(06MCE016)

has been accepted towards fulfillment of the requirement

for the degree of

Master of Technology in Computer Science & Engineering

Dr. S. N. Pradhan	Prof. D. J. Patel
P. G. Coordinator	Head of The Department

Prof. A. B. Patel
Director, Institute of Technology

CERTIFICATE

This is to certify that the work presented here by Mr. Tank Pranav entitled "Performance Measure of Digital Image Processing on FPGA based Co-processor and DSP Processor" has been carried out at Institute Of Technology, Nirma University during the period September 2007 – May 2008 is the bonafide record of the research carried out by him under my guidance and supervision and is up to the standard in respect of the content and presentation for being referred to the examiner. The results embodied in this major project, to the best of my knowledge, haven't been submitted to any other university or institution for award of any masters degree.

Dr. S. N. Pradhan
Guide and P.G. Coordinator,
Department of Computer Science & Engineering,
Institute of Technology,
Nirma University,
Ahmedabad.

Date: / /2008

ACKNOWLEDGEMENTS

The successful completion of a project is generally not an individual effort. It is an outcome of the cumulative efforts of a number of persons, each having own importance to the objective. This session is a vote of thanks and gratitude towards all those persons who have directly or indirectly contributed in their own specials way towards the completion of this project.

It gives me great pleasure in expressing thanks and profound gratitude to Dr. S. N. Pradhan, Guide and P.G. Coordinator, Department of Computer Engineering, Institute of Technology, Nirma University, Ahmedabad for his valuable guidance and continual encouragement throughout the project. I heartily thankful him for his time to time suggestions and the clarity of the concepts of the topic that helped me a lot during the project. I also express thanks to Mrs. Swati Jain, Project Co-Coordinator and Asst. Professor, Department of Computer Engineering, Institute of Technology, Nirma University, Ahmedabad for her support, guidance and continuous encouragement throughout the dissertation work.

I would like to give my special thanks to Prof. D. J. Patel, Head, Department of Computer Engineering, Institute of Technology, Nirma University for his continual kind words of encouragement and motivation throughout the Project. I am thankful to all faculty members for their special attention and suggestion towards the project work.

I extend my sincere thanks to my colleagues specially Jaimin Chavda, Gaurang Thakkar, Kapil Ruchandani for their support in my dissertation work. I would like to express my gratitude towards my family members who have always been my source of inspiration and motivation.

Tank Pranav

Roll No. 06MCE016

ABSTRACT

As the processing capability of processor increases the computation requirements of application two folds, especially Digital Image Processing algorithms applied on satellite images require tremendous amount of calculations because high resolution images are common in aerial and satellite surveillance, even larger amounts of data are coming in near future. Convolution and cross-correlation are fundamental algorithms required in many image processing tasks, in order to speed up their computation often they are implemented on parallel hardware.

Therefore, to reach constraints imposed, two approaches are possible; one is to use DSP processors having characteristics of SIMD and VLIW architecture, MAC operations which can exploit parallelism. The other is to use hybrid architecture consisting of DSP and FPGA as coprocessor. To implement above mentioned approaches proper DSP need to be selected. As a part of this, some basic image processing algorithms have been studied, implemented and simulated for various leading DSPs. The same algorithms have been implemented for P-IV and the results obtained are compared and analyzed.

The second approach has been implemented for 1D FFT algorithm. As FPGA get rid of pipeline hazards, it can result in faster execution. Here algorithm computation distribution has been suggested which implies implementation of complex multiplications to be implemented in FPGA and other calculations to be done in DSP. For this scenario the speed up is measured and analyzed.

CONTENTS

Certificate I
Acknowledgement III
Abstract IV
Contents V
List of Figures VII
List of Tables VIII

Chapter 1 INTRODUCTION 1
1.1 General 1
1.2 Motivation 3
1.3 Scope of the Work 4
1.4 Outline of Report 6

Chapter 2 LITERATURE SURVEY 7
2.1 Introduction of stereo imaging 7
2.2 Architectural Overview 10
2.3 Architecture of tiger-sharc dsp processor 16
2.4 Simulation tools 24

Chapter 3 IMAGE PROCESSING ALGORITHMS 29
3.1 Overview of algorithms 29
3.2 Profiling of algorithms 33
3.3 Intel integrated performance primitives 34

Chapter 4 SIMULATION OF IP ALGORITHM ON P-IV PROCESSOR .. 38
4.1 Simple-scalar simulator 38

Chapter 5 FFT ALGORITHMS ON HYRID ARCHITECTURE 45
5.1 Hybrid architecture (fpga & dsp) 45
5.2 FPGA interface to the c6000 dsp platform using EMIF .. 47
5.3 Theoretical calculation of performance 49

Chapter 6 RESULTS 55
6.1 Benchmarking results 55
6.2 1024 point, 1-D Fast Fourier Transform 56

Chapter 7 CONCLUSION 57

References 58

Appendix – A 60

Appendix – B 62

LIST OF FIGURES

2-1	TMS320C67x DSP Block Diagram	10
2-2	L1 Cache	14
2-3	Tiger Sharc Architecture	17
3-1	Butterfly diagram for DIT algorithm	30
3-1	Sobel Operator	32
3-3	Automatic CPU Detection and Code Dispatch	35
5-1	Hybrid architecture	46
5-2	FIFO Based EMIF Interface Design Example	48
5-3	Flow chart of 1024 point 1-D FFT	52
5-3	1024 points 1-D FFT on DSP & FPGA architecture	53

LIST OF TABLES

Table-1 Specifying the processor core

Table-2 Specifying the memory hierarchy

Table-3 Pentium 4 Processor Configuration

Table-4 Simulation result of Convolution algorithm

Table-5 Time taken on hybrid architecture

Table-6 Simulation result for various image processing algorithms

Table-7 Theoretical result for 1-D FFT algorithm

1 INTRODUCTION

1.1 GENERAL

Digital image processing is characterized by very high computational demands. Although it can be handled by “standard” computer hardware, such solution is not viable for an embedded system, where dimensions of the computer system, power consumption or data throughput are of concern. For these reasons, specialized hardware solutions based on a digital signal processor (DSP) or a Field Programmable Gate Array (FPGA) are usually used in embedded systems. As increasingly complex algorithms are implemented using digital signal processing, the performance demands of these algorithms rise exponentially. For cost-sensitive, high-volume applications like stereo image processing, online video processing, development of extremely specialized ASP(Application specific processors) are driven. However, for many other applications, the only options for implementing high-performance digital signal processing have been general-purpose DSPs and, more recently, FPGAs. Available Processor types range from general purpose processors that handle a wide variety of applications, to application-specific processors like DSPs, which are specific to a particular application class such as signal processing, to single purpose processors, which are customized to a very specific function.

The heart of any digital signal processing architecture is the Multiply-and-Accumulate (MAC) unit. Most signal processing applications utilize a great deal of multiplication: The MAC unit of a DSP accelerates this type of calculation by performing the multiplication of two numbers and then adding the result to all of the previous multiplications in what is called an "accumulator". Another key enabling technology of DSPs is the ability to process several operations at the same time. One way that DSPs can execute four operations at the same time is to use what is known as Very Long Instruction Word (VLIW) architecture. A VLIW is a single instruction that actually represent several operations. DSPs have typically been used to implement many of these applications. Although DSPs are programmable through software, the DSPs’ hardware architecture is not

flexible. Therefore, DSPs are limited by fixed hardware architecture such as bus performance bottlenecks, a fixed number of MAC blocks, fixed memory, fixed hardware accelerator blocks and fixed data widths. The DSPs' fixed hardware architecture is not suitable for many applications that require customized DSP function implementations. The architecture of FPGA, on the other hand, is designed with fine-grain parallelism, which makes it well suited for massively parallel algorithms. The basic characteristics of FPGA are relatively small capacity of the on-chip memory and relatively narrow throughput of memory interfaces, lack of wide-word processing units, and high cost of performing complex numerical operations, such as division, square root, logarithmic, exponential, and trigonometry functions (in smaller devices, these operations cannot be implemented at all). FPGAs provide a reconfigurable solution for implementing traditional applications and offer higher throughput along with DSP. Systems implemented in FPGA and DSPs can have customized architecture, customized bus structure, and customized memory, customized hardware accelerator blocks and a variable number of MAC blocks. A major advantage of FPGAs for many system architectures is that FPGA can behave as an acceleration purpose processor along with DSP , to increase throughput of the over all system. System architects use this capability to create products with various price points and performance capabilities without significantly affecting development costs or inventory. FPGA devices provide a reconfigurability which can be useful in changing design which is already ported into FPGA. FPGA devices incorporate a variety of embedded features such as embedded processors, memory blocks, etc.

Stereo Image Processing is a special class of image processing area where high amount of computation power is required. Stereo Image Processing implements functions like image correction, image rectification, image matching and disparity calculations like tasks. In this tasks image matching is a very basic but most expensive task. Stereo image matching requires two preprocessed images and matching of these images done by suitable algorithm. Results of algorithms for image matching may vary because different algorithms have different computation costs, running

time, accuracy as well as artificial intelligence may also be involved. In this dissertation Hierarchical matching algorithm is used for the level by level comparisons and generating accurate results.

1.2 MOTIVATION

Image processing is a one of the fast developing research area of computer vision. Faster image processing is very essential in current scenario for work automation .This work is useful in developing the vision using the computerized analysis , object detection and classification of the images captured by the sensors for better interpretation and analysis. When it comes to processing the digital images of high resolution in order of 4Kx4K, 8Kx8K and higher then this ,normal workstations can able to deliver the output but by taking the more computational resources in terms of memory and time to generate the output (or with decreased performance). Space agencies use the special purpose satellites for the earth surveillance which results the high resolution satellite images. Processing of these satellite images and video takes more time in processing .Sometimes it is possible that the real time faster processing is required and that work is generally not supported by the workstations. To resolve this we require special purpose processors like DSPs or ASPs specifically designed for these types of applications.

Digital Image Processing is a special class of Digital Signal Processing functions where image is a function of integer values. Using Digital Signal Processors resides between ASPs and GPPs(General Purpose Processor). It provides faster execution than GPPs while providing more flexibility than ASPs. Digital Signal Processors are useful for providing High throughput while operating at lower frequency than GPPs. DSP processors are microprocessors designed to perform digital signal processing—the mathematical manipulation of digitally represented signals. Today's DSP processors (or "DSPs") are sophisticated devices with impressive capabilities. Implementing basic image processing algorithms focus on the image enhancement, correction, preprocessing, frequency domain analysis like tasks. Some of the specific applications like stereo imaging, medical imaging like applications needs special treatment than that of the

basic tasks. Developing this kind of application on DSPs requires more attention than just implementation. Stereo imaging library requires large amount of computation power which is not available at the workstation end and also these workstations are not efficient and capable handling the resource requirement for these applications. These applications require special attention for real time execution in timely manner so that the constant high throughput can be achieved. This thesis aims at the proposing the solution for faster throughput by the means of using the DSP and FPGA's hybrid architecture. To achieve the mentioned goal would require the development of Library of Image processing functions for DSP and FPGA designs developed as special purpose processors.

1.3 SCOPE OF WORK

As the title suggests the goal of this dissertation focuses on the providing solution of high computation need for stereo image processing. To achieve this goal, work carried out in this research is useful for the organizations which require high computation power for processing of high resolution satellite images. Stereo imaging is one of the application which demands more computation power for generating disparity map and depth calculation of objects on the earth. By testing basic image processing algorithms on different DSPs as well as on FPGAs we decide suitable processor in performing image matching for the calculation of disparity between the stereo image pair. Analysis and benchmarking on image processing algorithms tested on TI DSP c64xx(Texas Instruments), TS-201(Analog Devices) and Pentium-4 (on Simple Scalar 3.0 Simulator Toolkit) and comparing the performance of each Processor .This includes selection of right DSP processors and development of algorithms by partitioning the parallel computation tasks between the FPGA(Spartan-3) and DSP Processor for increased throughput. Wherever intensive parallelism is available in algorithm that module is performed in FPGA and the remaining module is developed in DSP. DSPs are used for the complex functions implementation which requires more resources on FPGA as well as which can not be done parallel or pipelined manner. DSPs are providing solution for the applications which require complex functions to be implemented on DSPs for faster than GPPs. In this dissertation focus is on

the FPGA Development is done in Handel-C language and verification of results is done through the Handel-C simulator using Xilinx Spartan-3(1500L) FPGA. Procedure of testing and benchmarking of image processing algorithms is carried out for various steps of Stereo Reconstruction Pipeline. At the end of this dissertation work one can have outputs in terms of disparity map and depth map for the stereo image pair. These images can be used for efficient stereo reconstruction (Generating 3d space from 2d images).

1.4 OUTLINE OF THE THESIS

This thesis is organized a follows:

Chapter 2, Literature survey, consist of basic theories related with architecture of different Digital Signal processors, various simulation tools.

Chapter 3, Benchmarking Image Processing Algorithms, which describes various DIP algorithms and its profiling, as well as brief overview of Intel Integrated performance primitives, optimized libraries.

Chapter 4, Simulation of DIP algorithm on Pentium-IV processor, discussed architecture of Simple Scalar simulator, different simulation engine as well method of performance analysis for profiling.

Chapter 5, FFT algorithms on hybrid architecture, describes hybrid architecture as a FPGA co-processor, clock cycle saving with hybrid architecture.

Chapter 6, Result discusses the benchmarking of DIP algorithm, also clock cycle saving with hybrid architecture in comparison with single DSP.

Chapter 7, This chapter discusses the conclusion, which DSP is better in execution of DIP algorithms.

2 LITERATURE SURVEY

2.1 INTRODUCTION OF STEREO IMAGING

2.1.1 General

Stereo Imaging is the process of constructing the 3-Dimentional model using the 2-Dimentional Images for better human understanding. The task of building a general purpose computational-vision system is a grand challenge due to the compute-intensive nature of many vision algorithms. However, researchers have been successful in designing algorithms and building systems that deal with some specific tasks of the human vision system. One important feature of the human vision system is its ability to perceive depth of a viewed scene. This ability to perceive depth, known as stereo vision, or stereopsis is made possible by the difference in viewpoints of the scene when sensed by our left and right eyes. The information about depth in a scene is of great importance because it helps us navigate in a three-dimensional environment and aids us in recognizing objects of interest, among other tasks. In computer based stereo-vision systems, a stereo-rig is a pair of cameras placed side-by-side, much like our eyes, to capture the left and right images. In a stereo-vision system, this processing is carried out using a computing platform that can be based on software, hardware, or a mixture of the two. The depth information is encoded in the disparity [1], defined as the difference in pixel locations of corresponding points in the image pair. The disparity is inversely proportional to the distance of an object from the cameras, so the disparity increases as objects get closer to the cameras. The estimation of this disparity then becomes the primary task of a stereo-vision system.

In the simplest setup of a stereo-rig, where the optical axes of the two cameras [2] are parallel and the vertical axes are aligned, corresponding pixels lie at the same vertical coordinate in the image pair. The search for the corresponding pixel is therefore limited to the same scanline in the image pair, which allows processing of each scanline as they arrive. In the more general case where the cameras are not aligned as described above, the search for corresponding pixel may span across numerous scanlines

[3] and this increases the computational load of the system. When the cameras are not in the ideal setup, Image rectification of input images can be performed. Rectification is the process by which the input image pair is warped to resemble the output from an aligned stereo-rig.

Often, when viewing a scene from different viewpoints as in a stereo setup, objects visible in one image may not be visible in the other image. A foreground object hides, or occludes, different parts of the background in the left and right views, a phenomenon known as occlusion. In addition, the information present at the left edge of the image captured by the left camera is not available in the right image and vice-versa as this part of the scene falls outside the viewing area of the other camera. This further complicates the task of accurate disparity estimation because pixels visible in one image may not have a corresponding match in the other image of the pair. Related areas of Stereo Imaging:

- Robotic Vision/Machine Vision.
- 3D Computer Graphics.
- Computer Vision Geometry.

2.1.2 TMS320C6000 DSP Family Overview

The TMS320DSP family consists of fixed-point, floating-point, and multiprocessor digital signal processors (DSPs). TMS320DSPs have an architecture designed specifically for real-time signal processing.

With a performance of up to 6000 million instructions per second (MIPS) and an efficient C compiler, the TMS320C6000 DSPs [4] give system architects unlimited possibilities to differentiate their products. High performance, ease of use, and affordable pricing make the C6000 generation the ideal solution for multifunction applications, such as, 3-D transformations, Image compression/transmission, Image enhancement, Pattern recognition, Robot vision, etc. The newest member of the C6000 family, the C64x, brings the highest level of performance for processing data in this era of data convergence. At clock rates of 600 MHz and greater, the C64x can process information at a rate of 4800-6400 MIPS.

In addition to clock rate, more work can be done each cycle with the VelociTI architecture. These extensions include new instructions to accelerate performance in key applications and extend the parallelism of the architecture. Increased clock rate and increased CPU throughput are only part of the solution. Processing data at these extremely high rates increases the need for I/O bandwidth.

Three flexible Multi–channel Buffered Serial Ports can each supply 100Mbits/sec each of additional throughput. The internal DMA engine can provide over 2Gbytes/sec of I/O bandwidth with 64 independent channels. The C64x goes beyond a core and peripheral set to bring the maximum level of performance for processing digital data quickly. The tight coupling of the CPU architecture and the compiler help to maximize processor throughput. The RISC like instruction set and extensive use of pipelining allow many instructions to be scheduled and executed in parallel. The key extensions made to the 'C62x architecture that allow the 'C64x to perform more work each clock cycle include wider data paths, a larger register file, greater orthogonality and new instructions that support packed data processing.

2.1.3 TMS320C64x DSP Architecture

The C64x central processing unit (CPU) [5], as shown in Figure 1, consists of eight functional units, two register files, and two data paths. Like the C62x/C67x, two of these eight functional units are multipliers. The C64x multiplier has been enhanced so that it is capable of performing two 16-bit x 16-bit multiplies every clock cycle. Four 16-bit x 16-bit multiplier can be executed every cycle on the C64x. Using 600 MHz to represent early C64x performance, this means 2.4 billion 16-bit multiplies can occur every second. Moreover, each multiplier on the C64x has the capability of performing four 8-bit x 8-bit multiplies every clock cycle. At 600 MHz, this is equivalent to 4.8 billion 8-bit multiplies occurring every second.

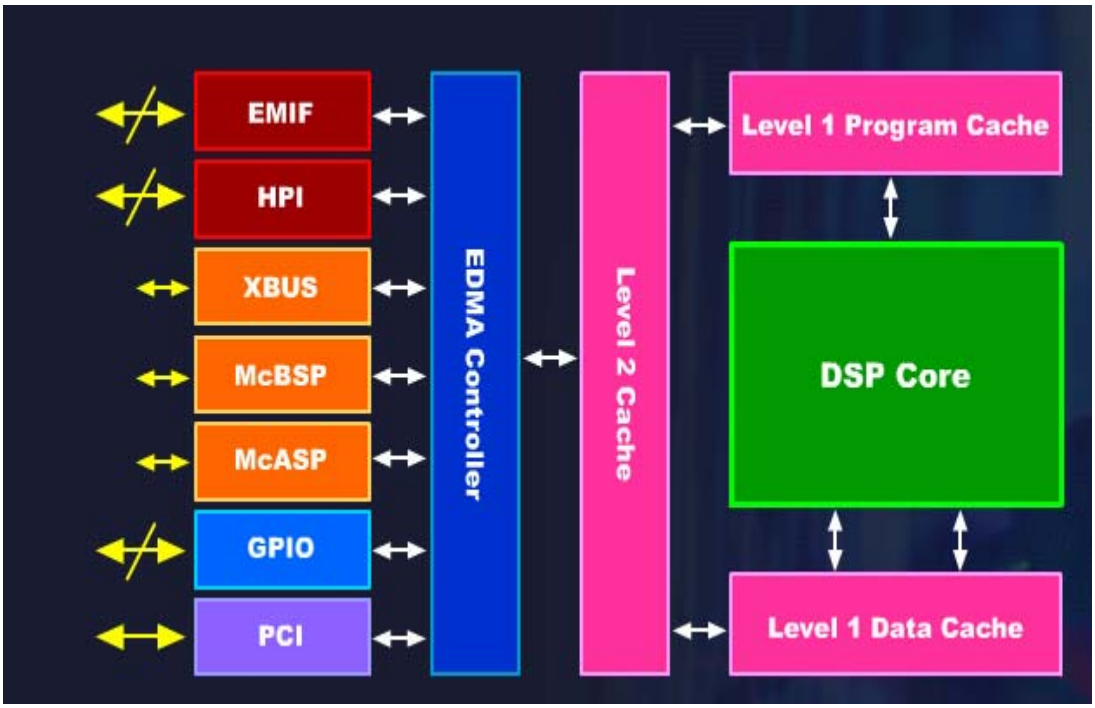


Figure 1: TMS320C67x DSP Block Diagram

The dual 16-bit extensions built into the multiply functional unit are also present in the other six functional units. These include dual 16-bit addition/subtraction, compare, and shift, min/max, and absolute value operations. The quad 8-bit extensions built into the multiply functional unit are found in four of the six remaining functional units. These include quad 8-bit addition/subtraction, compare, average, min/max, and bit expansion operations. Packed 8-bit and 16-bit data types are used by the code generation tools to take full advantage of these extensions. By doubling the registers in the register file and doubling the width of the data path as well as utilizing advanced instruction packing, the C6000 compiler can improve performance with even fewer restrictions placed upon it by the architecture. These additions and others make the C64x an even better compiler target than the original C62x architecture, while reducing code size by up to 25%.

2.2 ARCHITECTURAL OVERVIEW

2.2.1 C6000 CPU

The C6000 CPU components consist of:

- Two general-purpose register files (A and B)

- Eight functional units (.L1, .L2, .S1, .S2, .M1, .M2, .D1, and .D2)
- Two load-from-memory data paths (LD1 and LD2)
- Two store-to-memory data paths (ST1 and ST2)
- Two data address paths (DA1 and DA2)

2.2.2 Register File Enhancements

There are two general-purpose register files (A and B) in the C6000 data paths. For the C62x/C67x, each of these files contains 16 32-bit registers (A0-A15 for file A and B0-B15 for file B). The general-purpose registers can be used for data address pointers, or condition registers. The C64x register file doubles the number of general-purpose registers that are in the C62x/C67x cores with 32 32-bit registers per data path (A0-A31 for file A and B0-B31 for file B). On the C64x, A_0 may be used as a condition register as well, bringing the total to six condition registers. In all C6000 devices, registers A4-A7 and B4-B7 can be used for circular addressing. The Values larger than 32 bits, such as 40-bit long and 64-bit float quantities, are stored in register pairs, with the 32 LSBs of data placed in an even-numbered register and the remaining 8 or 32 MSBs in the next upper register (which is always an odd-numbered register). The C64x register file supports all the C62x data types and extends this by additionally supporting packed 8-bit types and 64-bit fixed-point data types. Packed data types store either four 8-bit values or two 16-bit values in a single 32-bit register or four 16-bit values in a 64-bit register pair.

2.2.3 Instruction Set

The TMS320C64x uses an opcode operand assembly language format where each instruction has an opcode field for the operation and an operand field for one to four operands. All instructions on the TMS320C64x can be executed conditionally. Six designated general purpose registers can be used as conditional registers. Many TMS320C64x instructions are simple and RISC-like , as on the TMS320C62x. However , the addition of extensive SIMD capabilities and application specific instructions means that some TMS320C64x instructions support multiple parallel operations within a single instruction.

The TMS320C64x does not support hardware looping, so all loops must be implemented in software. However, the parallel architecture of the processor allows the implementation of software loops with virtually no overhead. Because the TMS320C64x is a highly parallel architecture, obtaining maximum performance requires the programmer or code generation tools to schedule instruction carefully. This can be a challenge because the TMS320C64x has a complex architecture and long variable instruction latencies. Texas instruments assembly optimizer tools and C compiler simplify code development by automating the scheduling and parallelization processes.

2.2.4 Functional Units

The eight functional units in the C6000 data paths can be divided into two groups of four; each functional unit in one data path is almost identical to the corresponding unit in the other data path. The C64x is object code compatible with the C62x. Besides being able to perform all the C62x instructions, the C64x also contains many 8–bit and 16–bit extensions to the instruction set. For example, the MPYU4 instruction performs four 8x8 unsigned multiplies with a single instruction on a .M unit. The ADD4 instruction performs four 8–bit additions with a single instruction on a .L unit.

2.2.5 Register File Paths

Each functional unit reads directly from and writes directly to the register file within its own data path. That is, the .L1, .S1, .D1, and .M1 units write to register file A, and the .L2, .S2, .D2, and .M2 units write to register file B. Most data lines in the CPU support 32-bit operands, and some support long (40-bit) and double word (64-bit) operands. Each functional unit has its own 32-bit write port into a general-purpose register file. Each functional unit has two 32-bit read ports for source operands src1 and src2. Four units (.L1, .L2, .S1, and .S2) have an extra 8-bit-wide port for 40-bit long writes, as well as an 8-bit input for 40-bit long reads. Because each unit has its own 32-bit write port, all eight units can be used in parallel with every cycle when performing 32 bit operations. Since each C64x multiplier can return up to a 64-bit result, an extra write port has

been added from the multipliers to the register file, as compared to the C62x.

2.2.6 Increased Orthogonality

When we talk about orthogonality in the VelociTI architecture, we mean that there is a great deal of generality in the architecture. We have already discussed that the register file is general purpose. The registers can be a pointer to data or can contain data. We have also discussed how an ADD instruction can be performed on six of the eight functional units. This flexibility allows the compiler to achieve maximum performance. The C64x contains even more orthogonality than the original C62x/C67x architecture. The .D unit can now perform 32-bit logical instructions in addition to the .S and .L units. Also, the .D unit now directly supports load and store instructions for double-word data values. The C62x does not directly support loads and stores of double words, and the C67x only directly supports loads of double words. The .L and .D units can now be used to load 5-bit constants in addition to the .S unit's ability to load 16-bit constants. There is an additional factor that provides the compiler with more flexibility. On the C62x/C67x, one long source and one long result per data path could occur every clock cycle. On the C64x, up to two long sources and two long results can be accessed on each data path every clock cycle.

2.2.7 Two–Level Cache Architecture

On initial C64x devices, the CPU interfaces directly to dedicated level–one program (L1P) and data (L1D) caches of 16 Kbytes each. These caches operate at the full speed of CPU access. A second level unified L2 program/data memory provides flexible storage. Figure 2 depicts an example L2 of size 1024 Kbytes; the size and segmentation of the L2 cache in the C64x family may change over time. One configuration for L2 is entirely mapped SRAM. The other configurations have both SRAM and a 4-way set associative cache of various sizes. Changing the way memory can be mapped,

Figure 2: L1 Cache

Allows the memory can be mapped allows the user to lock critical code such as interrupt service routines or commonly called functions in on–chip RAM. It also allows critical data sections such as the software stack and often re–used coefficients to be locked on–chip.

2.2.8 Powerful Enhanced DMA Controller

The C64x EDMA can provide over 2Gbytes/sec of external bandwidth on initial implementations. The EDMA supports up to 64 channels triggered by independent events. A total of 85 parameter sets are available for linking or chaining. Linking allows a sequence of transfers to be issued when a single event occurs. Chaining allows one EDMA channel to trigger another channel upon data transfer completion. Linking and chaining allow continuous auto–initialization of DMA operation with only initial configuration by the CPU. These features also allow circular buffers, ping–pong buffers, and transfers of complex data structures. Transfers can be triggered on an element by element or frame by frame basis. Programmable triggering allows both sample by sample transfers and buffer by buffer transfers. Each channel supports both one and two–dimensional transfers. Strides are independently programmable for each dimension. Using 1–D and 2–D the user can transfer sub frames of an image as well as automatically interleave or de–interleave time–division multiplexed (TDM) digital streams. Byte, word, half-word, and double-word data sizes are supported.

2.2.9 Three External Buses

The initial C64x chip architecture supports up to 3 parallel external buses: two external memory interfaces (EMIFs) and one host port interface (HPI).

One EMIF (EMIFA) is 64–bits wide and is intended for direct connection to high– speed synchronous memory. A second 16–bit EMIF (EMIFB) is intended for external I/O peripherals such as FIFOs and parallel data converters. Decoupling memory from I/O devices both simplifies board design and provides I/O concurrency. Although the intent of the two EMIFs are different, they are identical except for their width, allowing for a variety of system designs. On initial implementations, these EMIFs have a maximum bus rate of 133 MHz. Each EMIF has four chips enable (CE) spaces. EMIFA can support read and write operations to 64–, 32–, 16–, and 8–bit external devices. Similarly, EMIFB can support read and write operations to both 16– and 8–bit devices. Variable width support allows interoperability with many external I/O peripherals and allows the system designer to make bandwidth/cost/power tradeoffs. Each EMIF has three memory controllers. The SDRAM controller supports 16 Mbit – 256 Mbit SDRAM devices. Finally, a programmable asynchronous controller with independent setup, strobe, and hold control allows easy interface to many asynchronous SRAMs, FIFOs, and peripheral devices. The EMIFs operate with dedicated external clock inputs that decouple CPU operating frequency from bus frequency. In addition, particular controllers can operate at 1x, 1/2x or 1/4x the bus input clock. All these features are independently configurable for each CE space of each EMIF. A 32–bit wide HPI provides dedicated connection to a variety of industry standard host processors and PCI bridge chips. The HPI can operate in either a 32–bit (HPI32) or 16–bit (HPI16) wide mode. An additional use of the HPI is as a slave port through which a mastering peripheral can stream data into the DSP. In some C64x devices, the 32-bit wide HPI is replaced by a dedicated PCI port. The C64x PCI port supports connection of the DSP to a PCI host via the integrated PCI master/slave bus interface and features a 32-bit address/data bus at 33MHz. The C64x PCI port contains the logic required to implement a fully compliant PCI Specification revision 2.2 bursting master/slave with access into the DSP's memory map (peripherals, on-chip RAM, and external memory through the EMIF). The C64x PCI port interfaces to the DSP via the EDMA internal address generation hardware. This architecture allows for both PCI master and

slave transactions, while keeping the EDMA channel resources available for other applications.

2.3 ARCHITECTURE OF TIGER-SHARC TS-201 DSP PROCESSOR

2.3.1 Overview

The ADSP-TS201S TigerSharc processor is an ultrahigh performance, static superscalar processor optimized for large signal processing tasks and communications infrastructure. The DSP combines very wide memory widths with dual computation blocks supporting floating-point (IEEE 32-bit and extended precision 40-bit) and fixed-point (8-, 16-, 32-, and 64-bit) processing— to set a new standard of performance for digital signal processors. The TigerSharc architecture [6] lets the DSP execute up to four instructions each cycle, performing 24 fixed-point (16-bit) operations or six floating-point operations. Four independent 128-bit wide internal data buses, each connecting to the six 4M bit memory banks, enable quad-word data, instruction, and I/O access and provide 33.6G bytes per second of internal memory bandwidth. Operating at 600 MHz, the ADSP-TS201S processor's core has a 1.67 ns instruction cycle time. Using its single-instruction, multiple-data (SIMD) features, the ADSP-TS201S processor can perform 4.8 billion, 40-bit MACS or 1.2 billion, 80-bit MACS per second.

Above Functional Block Diagram shows the ADSP-TS201S processor's architectural blocks. These blocks include:

- Dual compute blocks, each consisting of an ALU, multiplier, 64-bit shifter, 128-bit CLU, and 32-word register file and associated data alignment buffers (DABs)
- Dual integer ALUs (IALUs), each with its own 31-word register file for data addressing and a status register

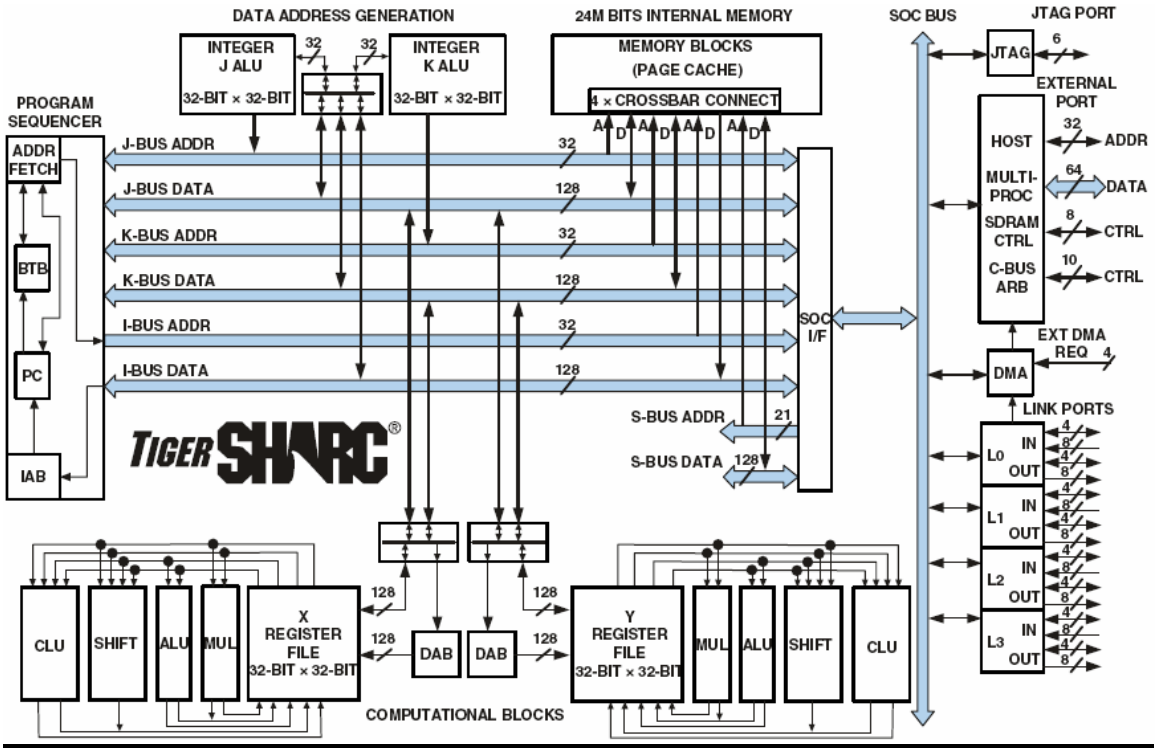


Figure 3: Tiger Sharc Architecture

- A program sequencer with instruction alignment buffer (IAB) and branch target buffer (BTB)
- An interrupt controller that supports hardware and software interrupts, supports level- or edge-triggers, and supports prioritized, nested interrupts
- Four 128-bit internal data buses, each connecting to the six 4M bit memory banks
- On-chip DRAM (24M bit)
- An external port that provides the interface to host processors, multiprocessing space (DSPs), off-chip memory mapped peripherals, and external SRAM and SDRAM
- A 14-channel DMA controller
- Four full-duplex LVDS link ports
- Two 64-bit interval timers and timer expired pin
- An 1149.1 IEEE-compliant JTAG test access port for on chip emulation

The TigerSharc DSP uses Static Superscalar architecture. This architecture is superscalar in that the ADSP-TS201S processor's core can execute simultaneously from one to four 32-bit instructions encoded in a very large instruction word (VLIW) instruction line using the DSP's dual compute blocks. Because the DSP does not perform instruction re-ordering at runtime— the programmer selects which operations will execute in parallel prior to runtime—the order of instructions is static. With few exceptions, an instruction line, whether it contains one, two, three, or four 32-bit instructions, executes with a throughput of one cycle in a 10-deep processor pipeline. For optimal DSP program execution, programmers must follow the DSP's set of instruction parallelism rules when encoding an instruction line. In general, the selection of instructions that the DSP can execute in parallel each cycle depends on the instruction line resources each instruction requires and on the source and destination registers used in the instructions. The programmer has direct control of three core components—the IALUs, the compute blocks, and the program sequencer. The ADSP-TS201S processor, in most cases, has a two-cycle execution pipeline that is fully interlocked, so—whenever a computation result is unavailable for another operation dependent on it—the DSP automatically inserts one or more stall cycles as needed. Efficient programming with dependency-free instructions can eliminate most computational and memory transfer data dependencies. In addition, the ADSP-TS201S processor supports SIMD operations two ways—SIMD compute blocks and SIMD computations. The programmer can load both compute blocks with the same data (broadcast distribution) or different data (merged distribution).

2.3.2 Dual Compute Blocks

The ADSP-TS201S processor has compute blocks that can execute computations either independently or together as a single instruction, multiple-data (SIMD) engine. The DSP can issue up to two compute instructions per compute block each cycle, instructing the ALU, multiplier, shifter, or CLU to perform independent, simultaneous operations. Each compute block can execute eight 8-bit, four 16-bit, two 32-bit, or one 64-bit SIMD computations in parallel with the operation in the other block.

These computation units support IEEE 32-bit single-precision floating-point, extended-precision 40-bit floating point, and 8-, 16-, 32-, and 64-bit fixed-point processing. The compute blocks are referred to as X and Y in assembly syntax, and each block contains four computational units—an ALU, a multiplier, a 64-bit shifter, a 128-bit CLU—and a 32- word register file. Fully orthogonal register file is used for transferring data between the computation units, data buses and for storing intermediate results. Instructions can access the registers in the register file individually (word-aligned), in sets of two (dual-aligned), or in sets of four (quad-aligned), as

- ALU—the ALU performs a standard set of arithmetic operations in both fixed- and floating-point formats. It also performs logic operations
- Multiplier—the multiplier performs both fixed- and floating- point multiplication and fixed-point multiply and accumulate
- Shifter—the 64-bit shifter performs logical and arithmetic shifts, bit and bit stream manipulation, and field deposit and extraction operations
- Provide 8 MACS per cycle peak and 7.1 MACS per cycle sustained 16-bit performance and provide 2 MACS per cycle peak and 1.8 MACS per cycle sustained 32-bit performance (based on FIR)
- Execute six single-precision floating-point or execute 24 fixed-point (16-bit) operations per cycle, regular operations performance at 600 MHz
- Perform two complex 16-bit MACS per cycle
- Execute eight trellis butterflies in one cycle

2.3.3 DUAL INTEGER ALU (IALU)

The ADSP-TS201S processor has two IALUs that provide powerful address generation capabilities and perform many general-purpose integer operations. The IALUs are referred to as J and K in assembly syntax and have the following features:

- Provide memory addresses for data and update pointers

- Support circular buffering and bit-reverse addressing
- Perform general-purpose integer operations, increasing programming flexibility
- Include a 31-word register file for each IALU

As address generators, the IALUs perform immediate or indirect (pre- and post-modify) addressing. They perform modulus and bit-reverse operations with no constraints placed on memory addresses for the modulus data buffer placement. Each IALU can specify either a single-, dual-, or quad-word access from memory. The IALUs have hardware support for circular buffers, bit reverse, and zero-overhead looping. Circular buffers facilitate efficient programming of delay lines and other data structures required in digital signal processing, and they are commonly used in digital filters and Fourier transforms. Each IALU provides registers for four circular buffers, so applications can set up a total of eight circular buffers. The IALUs handle address pointer wraparound automatically, reducing overhead, increasing performance, and simplifying implementation. Circular buffers can start and end at any memory location. Because the IALU's computational pipeline is one cycle deep, in most cases integer results are available in the next cycle. Hardware (register dependency check) causes a stall if a result is unavailable in a given cycle.

2.3.4 Program Sequencer

The ADSP-TS201S processor's program sequencer supports the following:

- A fully interruptible programming model with flexible programming in assembly and C/C++ languages; handles hardware interrupts with high throughput and no aborted instruction cycles
- A 10-cycle instruction pipeline—four-cycle fetch pipe and six-cycle execution pipe—computation results available two cycles after operands are available
- Supply of instruction fetch memory addresses; the sequencer's instruction alignment buffer (IAB) caches up to five fetched instruction lines waiting to execute; the program sequencer

extracts an instruction line from the IAB and distributes it to the appropriate core component for execution

- Management of program structures and program flow determined according to JUMP, CALL, RTI, RTS instructions, loop structures, conditions, interrupts, and software exceptions
- Branch prediction and a 128-entry branch target buffer (BTB) to reduce branch delays for efficient execution of conditional and unconditional branch instructions and zero-overhead looping; correctly predicted branches occur with zero overhead cycles, overcoming the five-to-nine stage branch penalty

2.3.5 Flexible Instruction Set

The 128-bit instruction line, which can contain up to four 32-bit instructions, accommodates a variety of parallel operations for concise programming. For example, one instruction line can direct the DSP to conditionally execute a multiply, an add, and a subtract in both computation blocks while it also branches to another location in the program. Some key features of the instruction set include:

- Algebraic assembly language syntax
- Direct support for all DSP, imaging, and video arithmetic types
- Eliminates toggling DSP hardware modes because modes are supported as options (for example, rounding, saturation, and others) within instructions
- Branch prediction encoded in instruction; enables zero overhead loops
- Parallelism encoded in instruction line
- Conditional execution optional for all instructions
- User-defined partitioning between program and data memory

2.3.6 DSP Memory

The DSP's internal and external memory is organized into a unified memory map, which defines the location (address) of all elements in the system. The memory map is divided into four memory areas—host space, external memory, multiprocessor space, and internal memory—and each

memory space, except host memory, is subdivided into smaller memory spaces.

The ADSP-TS201S processor internal memory has 24M bits of on-chip DRAM memory, divided into six blocks of 4M bits (128K words × 32 bits). Each block—M0, M2, M4, M6, M8, and M10—can store program instructions, data, or both, so applications can configure memory to suit specific needs. Placing program instructions and data in different memory blocks, however, enables the DSP to access data while performing an instruction fetch. Each memory segment contains a 128K bit cache to enable single cycle access to internal DRAM. The six internal memory blocks connect to the four 128-bit wide internal buses through a crossbar connection, enabling the DSP to perform four memory transfers in the same cycle. The DSP's internal bus architecture provides a total memory bandwidth of 33.6G bytes per second, enabling the core and I/O to access eight 32-bit data-words and four 32-bit instructions each cycle. The DSP's flexible memory structure enables:

- DSP core and I/O accesses to different memory blocks in the same cycle
- DSP core access to three memory blocks in parallel—one instruction and two data accesses
- Programmable partitioning of program and data memory
- Program access of all memory as 32-, 64-, or 128-bit words—16-bit words with the DAB

2.3.7 External Port (Off-Chip Memory/Peripherals Interface)

The ADSP-TS201S processor's external port provides the DSP's interface to off-chip memory and peripherals. The 4G word address space is included in the DSP's unified address space. The separate on-chip buses—four 128-bit data buses and four 32-bit address buses—are multiplexed at the SOC interface and transferred to the external port over the SOC bus to create an external system bus transaction. The external system bus provides a single 64-bit data bus and a single 32-bit address bus. The external port supports data transfer rates of 1G byte per second over the

external bus. The external bus can be configured for 32-bit or 64-bit, little-endian operations. When the system bus is configured for 64-bit operations, the lower 32 bits of the external data bus connect to even addresses, and the upper 32 bits connect to odd addresses. The external port supports pipelined, slow, and SDRAM protocols. Addressing of external memory devices and memory mapped peripherals is facilitated by on-chip decoding of high order address lines to generate memory bank select signals.

The ADSP-TS201S processor provides programmable memory, pipeline depth, and idle cycle for synchronous accesses; and external acknowledge controls to support interfacing to pipelined or slow devices, host processors, and other memory mapped peripherals with variable access, hold, and disable time requirements.

2.3.8 Host Interface

The ADSP-TS201S processor provides an easy and configurable interface between its external bus and host processors through the external port . To accommodate a variety of host processors, the host interface supports pipelined or slow protocols for ADSP-TS201S processor access of the host as slave or pipelined for host access of the ADSP-TS201S processor as slave. Each protocol has programmable transmission parameters, such as idle cycles, pipe depth, and internal wait cycles. The host interface supports burst transactions initiated by a host processor. After the host issues the starting address of the burst and asserts the BRST signal, the DSP increments the address internally while the host continues to assert BRST. The host interface provides a deadlock recovery mechanism that enables a host to recover from deadlock situations involving the DSP.

2.3.9 DMA Controller

The ADSP-TS201S processor's on-chip DMA controller, with 14 DMA channels, provides zero-overhead data transfers without processor intervention. The DMA controller operates independently and invisibly to the DSP's core, enabling DMA operations to occur while the DSP's core continues to execute program instructions. The DMA controller performs DMA transfers between internal memory, external memory, and memory-

mapped peripherals; the internal memory of other DSPs on a common bus, a host processor, or link port I/O; between external memory and external peripherals or link port I/O; and between an external bus master and internal memory or link port I/O. The DMA controller performs the following DMA operations:

- External port block transfers. Four dedicated bidirectional DMA channels transfer blocks of data between the DSP's internal memory and any external memory or memory mapped peripheral on the external bus. These transfers support master mode and handshake mode protocols
- Link port transfers. Eight dedicated DMA channels (four transmit and four receive) transfer quad-word data only between link ports and between a link port and internal or mode protocol. DMA priority rotates between the four receive channels
- AutoDMA transfers. Two dedicated unidirectional DMA channels transfer data received from an external bus master to internal memory or to link port I/O. These transfers only use slave mode protocol, and an external bus master must initiate the transfer

2.4 SIMULATION TOOLS

2.4.1 Key components of the Code Composer Studio IDE (Texas Instruments)

- Intelligent IDE with Code Maestro Technology
- C/C++ Compiler, Assembly Optimizer and Linker (Code Generation Tools)
- Real-Time Operating System (DSP/BIOS)
- Real-Time Data Exchange between host and target (RTDX)

2.4.1.1 Code Composer Studio Setup

Code Composer Studio Setup is a utility that is used to define the target board or simulator you will use with the Code Composer Studio IDE [7]. This information is called the system configuration and consists of a device driver that handles communication with the target plus other

information and files that describe the characteristics of your target, such as the default memory map. The Code Composer Studio IDE needs this information to establish communication with your target system and to determine which tools are applicable for the given target. With the exception of a DSP Starter Kit (DSK), which comes automatically configured for the DSK board, Code Composer Studio IDE will be configured for a simulator by default. You may wish to change the system configuration to match your environment prior to launching Code Composer Studio IDE.

2.4.1.2 Code Generation Tools

In the past, developing high performance DSP code has required to developer to optimize assembly code by hand and to have an intimate knowledge of the particular DSP architecture. Because time-to-market is becoming increasingly important, while the time and skill to optimally code a DSP are increasingly hard to find, there is a need for a more robust code development environment. The Code Composer Studio compile tools address this need by shifting the burden of optimization from hand-coded assembly to the C Compiler. With these tools it is possible to exploit the high performance of TI's DSP platforms without ever writing hand-coded assembly.

2.4.1.3 Interactive Profiler

Performance is a key issue for embedded systems developers. As programs grow in size and complexity it becomes more difficult for developers to isolate the subtle problems that cause poor performance. Profiling helps reduce the time it takes to identify and eliminate performance bottlenecks. The profiler analyzes program execution and shows where your program is spending its time. For example, a profile analysis can report how many cycles a particular function takes to execute and how often it is called. Profiling helps you to direct valuable development time toward optimizing the sections of code that most dramatically affect program performance.

Code Composer Studio IDE represents the evolution of the DSP development environment. It contains all of the functionality needed by

today's larger, distributed, global project teams. The intelligent IDE can help save valuable development time by making developers more productive enabling them to focus their energies on innovation instead of repetitive tasks and tool development.

2.4.2 Development tool – VDSP++ for Tiger – Sharc DSP

The ADSP-TS201S processor is supported with a complete set of CROSSCORE software and hardware development tools, including Analog Devices emulators and VisualDSP++ development environment. The same emulator hardware that supports other TigerSHARC processors also fully emulates the ADSP-TS201S processor.

The VisualDSP++ project management environment lets programmers develop and debug an application. This environment includes an easy to use assembler (which is based on an algebraic syntax), an archiver (librarian/library builder), a linker, a loader, a cycle-accurate instruction-level simulator, a C/C++ compiler, and a C/C++ run-time library that includes DSP and mathematical functions. A key point for theses tools is C/C++ code efficiency. The compiler has been developed for efficient translation of C/C++ code to DSP assembly. The DSP has architectural features that improve the efficiency of compiled C/C++ code. The VisualDSP++ debugger has a number of important features. Data visualization is enhanced by a plotting package that offers a significant level of flexibility. This graphical representation of user data enables the programmer to quickly determine the performance of an algorithm. As algorithms grow in complexity, this capability can have increasing significance on the designer's development schedule, increasing productivity. Statistical profiling enables the programmer to poll the processor as it is running the program.This feature, unique to VisualDSP++, enables the software developer to passively gather important code execution metrics without interrupting the real-time characteristics of the program.

Essentially, the developer can identify bottlenecks in software quickly and efficiently. By using the profiler, the programmer can focus on those areas

in the program that impact performance and take corrective action. Debugging both C/C++ and assembly programs with the VisualDSP++ debugger, programmers can:

- View mixed C/C++ and assembly code (interleaved source and object information)
- Insert breakpoints
- Set conditional breakpoints on registers, memory, and stacks
- Trace instruction execution
- Perform linear or statistical profiling of program execution
- Fill, dump, and graphically plot the contents of memory
- Perform source level debugging
- Create custom debugger windows

The VisualDSP++ IDE lets programmers define and manage DSP software development. Its dialog boxes and property pages let programmers configure and manage all of the TigerSHARC processor development tools, including the color syntax highlighting in the VisualDSP++ editor. This capability permits programmers to:

- Control how the development tools process inputs and generate outputs
- Maintain a one-to-one correspondence with the tool's command line switches
- The VisualDSP++ Kernel (VDK) incorporates scheduling and resource management tailored specifically to address the memory and timing constraints of DSP programming. These capabilities enable engineers to develop code more effectively, eliminating the need to start from the very beginning when developing new application code. The VDK features include threads, critical and unscheduled regions, semaphores, events, and device flags. The VDK also supports priority-based, preemptive, cooperative, and time-sliced scheduling approaches. In addition, the VDK was designed to be scalable. If the application does not use a specific feature, the support code for that feature is excluded from the target system

- The Analog Devices families of emulators are tools that every DSP developer needs in order to test and debug hardware and software systems. Analog Devices has supplied an IEEE 1149.1 JTAG test access port (TAP) on each JTAG DSP. The emulator uses the TAP to access the internal features of the DSP, allowing the developer to load code, set breakpoints, observe variables, observe memory, and examine registers. The DSP must be halted to send data and commands, but once an operation has been completed by the emulator, the DSP system is set running at full speed with no impact on system timing

3 IMAGE PROCESSING ALGORITHMS

3.1 OVERVIEW OF ALGORITHMS

3.1.1 FFT (Fast Fourier Transform)

FFT algorithm has been implemented on 1024 inputs, 512-complex twiddle factors. The N-point Discrete Fourier Transform (DFT) of a finite duration sequence x(n) is defined as follows.

$$X(k) = \sum_{n=0}^{N-1} x(n) W^{nk} \quad k = 0,1,\ldots N-1$$

where W= e ^ $^{-j\,(2nn/N)}$ is referred as the twiddle factor, N is the transform size and j = √-1. The FFT is an efficient algorithm to compute the DFT and its inverse (Cooley and Tukey). It generally falls into two classes: Decimation In Time (DIT), and Decimation In Frequency (DIF). The DIT algorithm first rearranges the input elements in bit reversed order and then builds the output transform. The DIF algorithm first transforms and then rearranges the output values. The basic idea of these algorithms is to break up an N–point DFT transform into successive smaller and smaller transform known as a butterfly (basic computational element). The smallest transform used is a 2-point DFT known as radix-2, it processes groups of 2 samples.

To calculate FFT for N number of inputs, Decimation in Time algorithm requires following points.

- L = $\log_2 N$ stages
- ((N/2) * L) number of complex multiplications
- (N * L) number of additions
- (N/2) twiddle factors should be stored

In order that the computation may be done in place, the input sequence must be stored in a non-sequential order. In fact, the order in which the input data are stored and accessed is referred to as bit-reversed order. Butterfly DIT FFT algorithm can be explained from the following figure-4.

If (n_2, n_1, n_0) is the binary representation of the index of the sequence x[n], then the sequence value x [n_2, n_1, n_0] is stored in the array position

$X_0[n_0, n_1, n_2]$. That is, in determining the position of x $[n_2, n_1, n_0]$ in the input array, one must reverse the order of the bits of the index n.

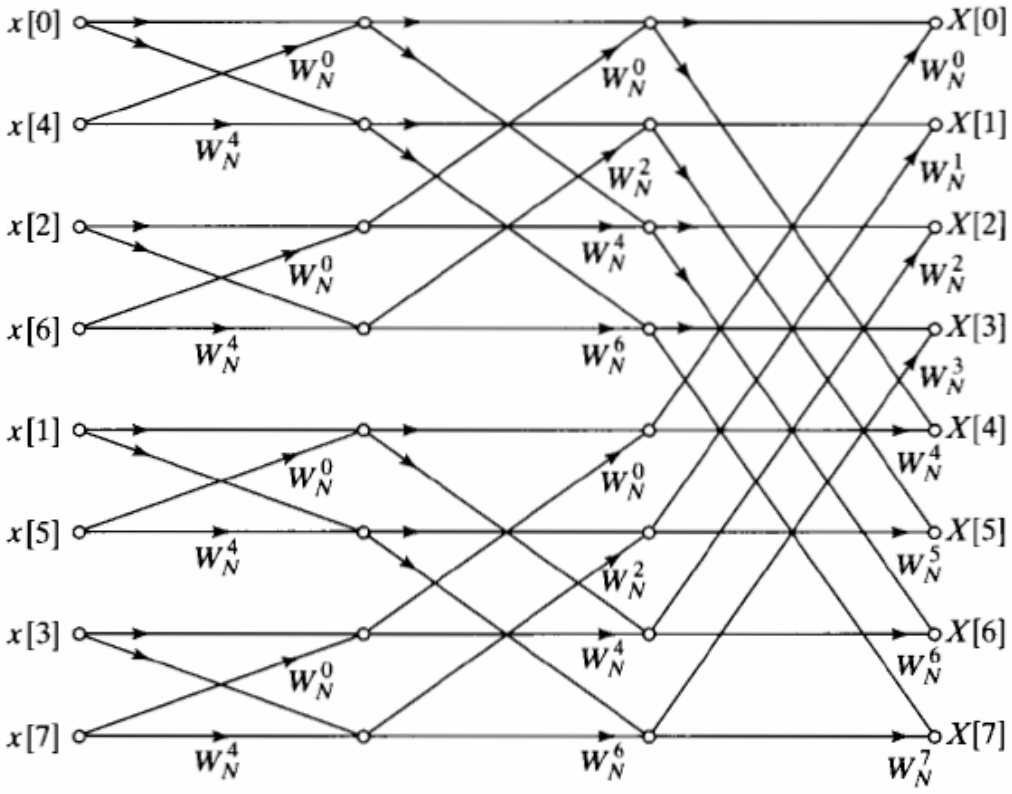


Figure 4: Butterfly diagram for DIT algorithm

3.1.2 Image Convolution

Image Convolution has been implemented on 256 x 3 inputs, and 3 x 3 mask. The process by which we move a mask from pixel to pixel in an image, and compute a predefined quantity at each pixel, is the foundation of the convolution process. Formally, the discrete convolution of two function f(x,y) and h(x,y) of size M X N is denoted by f(x,y) * h(x,y) and is defined by the expression

$$f(x,y) * h(x,y) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} f(m,n)\, h(x-m,y-n)$$

The minus sign in particular simply mean that function h is mirrored about the origin. Above equation is really nothing more than an implementation for

- Flipping one function about the origin
- Sifting that function with respect to the other by changing the values of (x,y)
- Computing a sum of products over all values of m and n, for each displacement (x, y)

3.1.3 Image Correlation

Image Correlation has been implemented on 720 x 3 inputs and 3 x 3 masks. Consider the scenario for finding matches of sub image w(x, y) of size J x K within an image f(x, y) of size M x N, where we assume that J <= M and K <= N. In its simplest form, the correlation between f(x, y) and w(x, y) is

$$C(x, y) = \sum_s \sum_t f(s, t)\, w(x+s, y+t)$$

For x = 0 ,1 ,2 , … , M-1 and y = 0,1,2,…,N-1 , and the summation is taken over the image region where w and f overlap. For one value of (x, y), say(x_0, y_0) inside f, application of above equation yields one value of c. As x and y are varied, w moves around the image area, giving the function c(x,y).The maximum value (s) of c indicated the position where w best matches f. The correlation function given in above equation has disadvantage of being sensitive to changes in the amplitude of f and w. For example, doubling all values of f doubles the value of c(x, y). An approach frequently used to overcome this difficulty is to perform matching via the correlation coefficient, which is defined as

$$N.C. = \frac{\sum_s \sum_t [f(s, t) - f'(s, t)]\, [w(x+t, y+t) - w']}{\left\{ \sum_s \sum_t [f(s,t) - f'(s,t)]^2 \sum_s \sum_t [w(x+t,y+t) - w']^2 \right\}^{1/2}} \quad \text{eq(3)}$$

where x = 0,1,2,M-1 and y = 0,1,2,N-1 , w' is the average value of the pixels in w(computed only once), f' is the average value of f in the region coincident with the current location of w, and the summations are taken over the coordinates common to both f and w. The correlation coefficient

is scaled in the range -1 to 1, independent of scale changes in the amplitude of f and w.

3.1.4 Sobel Operator

This algorithm has been implanted on 256 inputs. The difference between the third and first rows of 3 x 3 image region approximates the derivative in the x

-1	-2	-1
0	0	0
1	2	1

-1	0	1
-2	0	2
-1	0	1

Figure 5: Sobel Operator

direction, and the difference between the third and first columns approximates the derivative in the y direction. The masks shown in the figure-5 are called the Sobel operators. The idea behind using a weight value of 2 is to achieve some smoothing by giving more importance to the center point. The coefficients in all the masks shown in figure 5 sum to zero, indicating that they would give a response of zero in an are of constant gray level, as expected of a derivative operator.

3.1.5 Median Filter

Median filter has been implemented on 256 x 3 inputs and 3 x 3 sobel mask. Order-statistics filters are nonlinear spatial filters whose response is based on ordering (ranking) the pixels contained in the image area encompassed by the filter, and then replacing the value of the center pixel with the value determined by the ranking result.

The best-known example in this category is the median filter, which, as its name implies, replaces the value of a pixel by the median of the gray levels in the neighborhood of that pixel (the original value of the pixel is included in the computation of the median). Median filters are quite

popular because, for certain types of random noise, they provide excellent noise-reduction capabilities, with considerably less blurring than linear smoothing filters of similar size. Median filters are particularly effective in the presence of impulse noise, also called salt-and-pepper noise because of its appearance as white and black dots superimposed on an image. The median, j, of a set of values is such that half the values in the set are less than or equal to j, and half are greater than or equal to j. In order to perform median filtering at a point in an image, we first sort the values of the pixel in question and its neighbors, determine their median, and assign this value to that pixel. For example, in a 3*3 neighborhood the median is the 5th largest value, in a 5*5 neighborhood the 13th largest value, and so on. When several values in a neighborhood are the same, all equal values are grouped. For example, suppose that a 3*3 neighborhood has values (10, 20, 20, 20, 15, 20, 20, 25, and 100). These values are sorted as (10, 15, 20, 20, 20, 20, 20, 25, 100), which results in a median of 20. Thus, the principal function of median filters is to force points with distinct gray levels to be more like their neighbors. In fact, isolated clusters of pixels that are light or dark with respect to their neighbors, and whose area is less than $n^2/2$ (one-half the filter area), are eliminated by an n*n median filter. In this case "eliminated" means forced to the median intensity of the neighbors. Larger clusters are affected considerably less.

3.2 PROFILING OF ALGORITHMS

Above algorithms were implemented in Code composer studio as well as VDSP++ simulators. Parameters of profiling are measured, by keeping equal number of inputs and same source code on both of the simulators. After compiling the given algorithm, we need to load the assembly code in to memory to execute it. In CCS simulator, it provides us profiling information in terms of clock cycles , while in VDSP ++ simulator , one need to generate clock measuring source code. After executing algorithm on both of the simulators, one can match the results of both of the simulators. Execution time can be found out by dividing clock cycles with the frequency of the DSP processor.

For example, 1-D FFT took 43214 cycles on CCS simulator from Texas Instrument. Operating speed of TI DSP is 600 MHz.
Execution time = (43214) / (600 * (10 ^ 6)) = 72.06 μs.

3.3 INTEL INTEGRATED PERFORMANCE PRIMITIVES

Intel Integrated Performance Primitives (Intel® IPP) is an extensive library of multi-core-ready, highly optimized software functions for multimedia, data processing, and communications applications. Intel IPP is available as a standalone product, or with the Intel Compiler Professional Editions for a more complete and cost-effective solution. Intel IPP offers thousands of optimized functions covering frequently-used, fundamental algorithms in:

- Video coding
- Signal processing
- Audio coding
- Image processing
- Speech coding
- JPEG coding
- Speech recognition
- Computer vision
- Data compression
- Image color conversion
- Vector/Matrix mathematics

3.3.1 Basic Features

The Intel Integrated Performance Primitives [8], like other members of the Intel Performance Libraries, is a collection of high-performance code that performs domain-specific operations. It is distinguished by providing a low-level, stateless interface. Based on experience in developing and using Intel Performance Libraries, Intel IPP has the following major distinct features:

- The Intel IPP provides basic low-level functions for creating applications in several different domains, such as signal processing, image and video processing, and operations on small matrices

- Intel IPP functions follow the same interface conventions including uniform naming rules and similar composition of prototypes for primitives that refer to different application domains
- Intel IPP functions use abstraction level which is best suited to achieve superior performance figures by the application programs.

To speed up performance, Intel IPP functions are optimized to use all benefits in variation of different Intel architecture processors. Besides that, most of Intel IPP functions do not use complicated data structures, which helps reduce overall execution overhead.

3.3.2 Automatic CPU Detection and Dispatching

Automatic CPU detection and dispatch is a key feature in the Intel IPP for Pentium and Intel Xeon processors; this feature enables highly-optimized cross-platform development. With this transparent mechanism, the built in dispatcher determines the best Intel IPP code to execute based on the underlying processor. Automatic CPU detection and dispatch enhances the advantage of a common API by removing the burden of platform optimization from the developer.

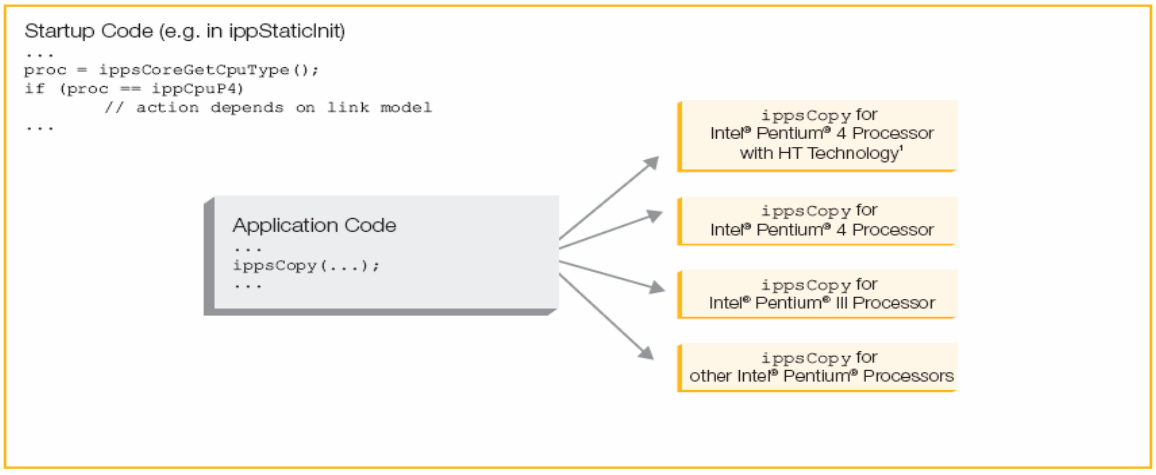


Figure 6: Automatic CPU Detection and Code Dispatch

3.3.3 Performance

Intel offers a full suite of library functions that can help developers easily create the fastest software possible on Intel architecture. The libraries provide high-quality code that is optimized to take advantage of the specific performance features built into each of the Intel Intel IPP functions are designed to deliver performance beyond what optimized

compilers alone can deliver by matching the function algorithms to low-level optimizations based on the processor's available features such as Streaming SIMD Extensions (SSE) and other optimized instruction sets.

The Intel IPP library provides cross-platform support through a single API. The large set of utility functions that Intel IPP provides supports a broad range of Intel processor families. For each of the supported target processors, Intel IPP provides API-conformant, variant code that is specifically written to yield the best system performance for the target processor, taking into account memory bandwidth and caching behavior of the target environment. Wherever applicable, the code was written with thread safety in mind. These libraries also offer a variant form that takes advantage of execution thread concurrency for those functions that can realize performance increases due to threading. Intel IPP functions yield significantly better performance than equivalent compiler-generated C code. By replacing sections of C code with equivalent Intel IPP functions, application programs are able to complete key tasks in significantly shorter timeframes. This ability to complete critical tasks in a timely manner is particularly important for event-driven and real-time constrained operations. Such operations typically take place within device drivers, audio and video processing and motion picture rendering.

3.3.4 Compatibility

Intel is committed to smoothing application program migration through the evolving generations of CPUs and computer architecture. By adhering to the abstract model defined by the Intel Performance Libraries' single API, developers are freed to focus on their own code development without worrying about library compatibility issues. Intel IPP 5.3 supports Windows Vista and 64-bit Mac OS X applications. Code re-use and cross-platform operating system development is simplified by having one library API for Windows, Linux and Mac OS. Intel IPP is fully compatible with other development tools from Intel, such as compilers, performance and threading analyzers, and other Intel Performance Libraries. In addition, Intel IPP is easily used and integrated with popular development tools and environments, such as Microsoft Visual Studio, Xcode, Eclipse, and the

GNU Compiler Collection (GCC). Multi-core processors, including Intel Core2 Quad, Intel Core2 Extreme, Intel Core2 Duo, Intel Core Duo, Intel Xeon and Intel Pentium D processors. Intel 64 architecture-based systems, including Intel Core 2 processors, Intel Xeon processors, Intel Pentium D processors, and compatible AMD processors. IA-32 architecture-based processors, including Intel Core processor family, Intel Pentium processors, and compatible AMD processors.

3.4 Implementation of IP algorithms using IPP

Intel IPP functions perform two-dimensional finite linear convolution operation between two source images and write the result into the image. Convolution is used to perform many common image processing operations including sharpening, blurring, noise reduction, embossing, and edge enhancement.

3.4.1 Convolution Function

IppStatus ippiConvFull_<mod>(const Ipp<datatype> pSrc1, int src1Step, IppiSize src1Size, const Ipp<datatype> pSrc2, int src2Step, IppiSize src2Size, Ipp<datatype> pDst, int dstStep, int divisor);

For convenience, one can represent any digital image f as a matrix with M_f columns and N_f rows that contains pixel values f [i,j], $0 \leq i < M_f$, $0 \leq j < N_f$. Here input size of matrix has been taken as 4x4 and mask size as 3x3. We must get output matrix size of 6x6. After comparing output of image processing functions obtained by IPP optimized function and the function used in Texas instrument DSP, we can measure the performance of IPP functions. Profiling of such optimized library IPP function can be obtained by GetCpuClock () function which also belongs to IPP optimized libraries family. By taking help of this function one can achieve performance in terms of clock cycles on Pentium processor.

4 SIMULATION OF IP ALGORITHMS ON P-IV PROCESSOR

4.1 SIMPLE-SCALAR SIMULATOR

The Simple-Scalar tool set is a system software infrastructure used to build modeling applications for program performance analysis, detailed micro architectural modeling, and hardware-software co-verification. Using the Simple Scalar tools, users can build modeling applications that simulate real programs running on a range of modern processors and systems. The tool set includes sample simulators ranging from a fast functional simulator to a detailed, dynamically scheduled processor model that supports non-blocking caches, speculative execution, and state-of-the-art branch prediction.

Simple-Scalar simulators [12] can emulate the Alpha, PISA, ARM, and x86 instruction sets. The tool set includes a machine definition infrastructure that permits most architectural details to be separated from simulator implementations. All of the simulators distributed with the current release of Simple-Scalar can run programs from any of the above listed instruction sets. Complex instruction set emulation (e.g., x86) can be implemented with or without microcode, making the Simple-Scalar tools particularly useful for modeling CISC instruction sets.

The PISA instruction set (a.k.a. the portable instruction set architecture) is a simple MIPS-like instruction set maintained primarily for instructional use. A GNU GCC-based cross-compiler and pre-built libraries are also available for this target. The PISA target is particularly useful for computer engineering instruction as the tools can be built on a wide range of host platforms, including Linux/x86, Win2000, SPARC Solaris, and others.

Simple-Scalar builds on most 32-bit and 64-bit flavors of UNIX and Windows NT-based operating systems. The internal software architecture of the tool set includes a host interface module, permitting fast porting to other host platforms. The host interface module permits cross-endian emulation, thus it is possible to use emulate a target on a host platform

with a different endian, e.g., running Alpha ISA emulation on a SPARC Solaris host platform

4.1.1 Learning Objectives

To be able to use the Simple-Scalar tool set to test out a range of architectural parameters, such as cache size and configuration, relative fetch/issue rates, inorder/out-of-order strategies, and load/store buffering strategies. Also to gain experience with setting up special benchmark trials to profile the performance of a new CPU. Finally, to experiment with dramatically different architectures, pipeline lengths, and number.

When developing a simulator the three opposing dimensions, Performance, Flexibility and Detail need to be reconciled for the needs of the user. The sim-outorder simulator is the most detailed CPU simulator provided with the Simple-Scalar tool set and suffering some reduction in performance. It was originally based on the MIPS-4 instruction set architecture and models a very modern superscalar microprocessor with 10 execution units (pipelines). Newer ARM and x86 versions have also been developed by other research labs. It attempts to maximally exploit ILP (Instruction Level Parallelism) and keep all the execution units busy by using out-of-order instruction execution. In many ways it realistically models the current processors found in the latest workstations/PCs. Sim-fast is a functional simulator, with a single, serial instruction stream, no caching and no command line switches. Also it does not capture timing information, unlike sim-outorder. An alternative, slower but more detailed simulation can be carried out by sim-safe. Simple-Scalar is widely used in academic research as well as commercial product development. The performance is good, on a P4/1.8 GHz host, sim-fast will emulate 10 Mips, while sim-outorder achieves 350 kips.

4.1.2 The Simple-Scalar architecture

The Simple-Scalar architecture is derived from the MIPS-IV ISA. The tool suite defines both little-endian and big-endian versions of the architecture to improve portability (the version used on a given host machine is the one that matches the endianness of the host). The semantics of the

Simple-Scalar ISA are a superset of MIPS with the following notable differences and additions:

- There are no architected delay slots: loads, stores, and control transfers do not execute the succeeding instruction
- Loads and stores support two addressing modes—for all data types—in addition to those found in the MIPS architecture. These are: indexed (register + register), and auto-increment/decrement
- A square-root instruction, which implements both single and double-precision floating point square roots
- An extended 64-bit instruction encoding

4.1.2.1 Functional simulation

Sim-fast does no time accounting, only functional simulation—it executes each instruction serially, simulating no instructions in parallel. Sim-fast is optimized for raw speed, and assumes no cache, instruction checking. A separate version of sim-fast, called sim-safe, also performs functional simulation, but checks for correct alignment and access permissions for each memory reference. Although similar, sim-fast and sim-safe are split (i.e., protection is not toggled with a command-line argument in a merged simulator) to maximize performance. Neither of the simulators accepts any additional command-line arguments. Both versions are very simple: less than 300 lines of code—they therefore make good starting points for understanding the internal workings of the simulators.

4.1.2.2 Cache simulation

The Simple-Scalar distribution comes with two functional cache simulators; sim-cache and sim-cheetah. These simulators are ideal for fast simulation of caches if the effect of cache performance on execution. Sim-cache accepts the following arguments:

- cache:dl1 <config> configures a level-one data cache
- cache:dl2 <config> configures a level-two data cache
- cache:il1 <config> configures a level-one instr. cache
- cache:il2 <config> configures a level-two instr. cache
- tlb:dtlb <config> configures the data TLB
- tlb:itlb <config> configures the instruction TLB

The defaults used in sim-cache are as follows:

- L1 instruction cache: il1:256:32:1:l (8 KB)
- L1 data cache: dl1:256:32:1:l (8 KB)
- L2 unified cache: ul2:1024:64:4:l (256 KB)
- Instruction TLB: itlb:16:4096:4:l (64 entries)
- Data TLB: dtlb:32:4096:4:l (128 entries)

4.1.2.3 The Cheetah

This engine simulates fully associative caches efficiently, as well as simulating a sometimes-optimal replacement policy. This policy was called MIN, although the simulator refers to it as opt. Opt uses future knowledge to select a replacement; it chooses the block that will be referenced the furthest in the future (if at all). This policy is optimal for read-only instruction streams. It is not optimal for write-back caches because it may be more expensive to replace a block referenced
further in the future if the block must be written back, as opposed to a clean block referenced slightly less far in the future.

Both of these simulators are ideal for performing high-level cache studies that do not take access time of the caches into account (e.g., studies that are concerned only with miss rates).

4.1.2.4 Out-of-order processor timing simulation

The most complicated and detailed simulator in the distribution, by far, is sim-outorder. This simulator supports out-of-order issue and execution, based on the (RUU) Register Update Unit. The RUU scheme uses a reorder buffer to automatically rename registers and hold the results of pending instructions. Each cycle the reorder buffer retires completed instructions in program order to the architected register file. The processor's memory system employs a load/store queue. Store values are placed in the queue if the store is speculative. Loads are dispatched to the memory system when the addresses of all previous stores are known. Loads may be satisfied either by the memory system or by an earlier store value residing in the queue, if their addresses match. Speculative loads may generate cache misses, but speculative TLB misses stall the pipeline until the branch condition is known.

Following table-1 explains the defalut configuration of simple scalar simulator. Table-2 explains the cache configuration of cache memory.

Table-1: Specifying the processor core

fetch: ifqsize <size>	Set the fetch width to be <size> instructions. Must be a power of two. The default is 4.
fetch: speed <ratio>	Set the ratio of the front end speed relative to the execution core (allowing <ratio> times as many instructions to be fetched as decoded per cycle).
fetch: plat <cycles>	Set the branch misprediction latency. The default is 3 cycles.
decode: width <insts>	Set the decode width to be <insts>, which must be a power of two. The default is 4.
issue:width <insts>	Set the maximum issue width in a given cycle. Must be a power of two. The default is 4.
issue:inorder	Force the simulator to use in-order issue. The default is false.
issue:wrongpath	Allow instructions to issue after a miss peculation. The default is true.
ruu:size <insts>	Capacity of the RUU (in instructions). The default is 16.
lsq:size <insts>	Capacity of the load/store queue (in instructions). The default is 8.
res:ialu <num>	Specify number of integer ALUs. The default is 4.
res:imult <num>	Specify number of integer multipliers/dividers. The default is 1.
res:memports <num>	Specify number of L1 cache ports. The default is 2.
res:fpalu <num>	Specify number of floating point ALUs. The default is 4.
res: fpmult <num>	Specify number of floating point multipliers/dividers. The default is 1.

Table-2: Specifying the memory hierarchy

cache:dl1lat <cycles>	Specify the hit latency of the L1 data cache. The default is 1 cycle.
cache:d12lat <cycles>	Specify the hit latency of the L2 data cache. The default is 6 cycles.
cache:il1lat <cycles>	Specify the hit latency of the L1 instruction cache. The default is 1 cycle.
cache:il2lat <cycles>	Specify the hit latency of the L2 instruction cache. The default is 6 cycles.
mem: lat <1st> <next>	Specify main memory access latency (first, rest). The defaults are 18 cycles and 2 cycles.
mem: width <bytes>	Specify width of memory bus in bytes. The default is 8 bytes.
tlb: lat <cycles>	Specify latency (in cycles) to service a TLB miss. The default is 30 cycles.

Table-3 explains the exact Pentium-IV processor's configuration, so that user can simulate the performance parameters by concentrating Pentium-IV processor [9].

Table-3: Pentium 4 Processor Configuration

Instruction fetch queue size (in insts) -fetch: ifqsize	64
Extra branch mis-prediction latency -fetch: mplat	3
bimodal predictor BTB size -bpred:bimod	2048
2-level predictor config (<l1size> <l2size><hist_size>) -bpred:2lev	1 1024 8
Instruction decode B/W (insts/cycle) -decode:width	4
Instruction issue B/W (insts/cycle) -issue:width	4
run pipeline with in-order issue -issue: inorder	false
issue instructions down wrong execution paths -issue: wrongpath	true
register update unit (RUU) size -ruu: size	16
load/store queue (LSQ) size -lsq: size	8
l1 data cache config, i.e., {<config>|none} -cache: dl1	dl1:128:64:4:l
l1 data cache hit latency (in cycles) -cache: dl1lat	1
l2 data cache config, i.e., {<config>|none} -cache:dl2	ul2:16384:64:8:l
l2 data cache hit latency (in cycles) -cache: dl2lat	6
l1 inst cache config, i.e., {<config>|dl1|dl2|none} -cache:il1	il1:512:32:1:l
l1 instruction cache hit latency (in cycles) -cache:il1lat	2
l2 instruction cache hit latency (in cycles) -cache:il2lat	7
flush caches on system calls -cache:flush	false
convert 64-bit inst addresses to 32-bit inst equivalents -cache: compress	false
memory access latency (<first_chunk> <inter_chunk>) -mem:lat	18 2
Memory access bus width (in bytes) -mem: width	8
Instruction TLB config, i.e., {<config>|none} -tlb: itlb	dtlb: 16:4096:4: l
data TLB config, i.e., {<config>|none} -tlb: dtlb	dtlb: 16:4096:4: l
inst/data TLB miss latency (in cycles) -tlb: lat	30
total number of integer ALU's available -res:ialu	2
total number of integer multiplier/dividers available -res:imult	1
total number of floating point ALU's available -res:fpalu	1
number of floating point multiplier/dividers available -res:fpmult	1

Execution of convolution algorithm obtained through Simple-Scalar simulator. Because of using Release version in running DIP algorithm with DSP, one can also use optimization option –O3 in this simulator. The results of convolution algorithm through Simple-Scalar simulator are shown in the following table.

Table-4: Simulation result of Convolution algorithm

sim_num_insn	92918	Total number of instructions committed.
sim_num_refs	23342	Total number of loads and stores committed.
sim_num_loads	18910	Total number of loads committed.
sim_num_stores	4432.0000	Total number of stores committed.
sim_num_branches	7042	Total number of branches committed.
sim_elapsed_time	1	Total simulation time in seconds.
sim_inst_rate	92918.0000	Simulation speed (in insts/sec).
sim_total_insn	96741	Total number of instructions executed.
sim_total_refs	25626	Total number of loads and stores executed.
sim_total_loads	21125	Total number of loads executed.
sim_total_stores	4501.0000	Total number of stores executed.
sim_total_branches	7137	Total number of branches executed.
sim_cycle	53047	Total simulation time in cycles.
sim_IPC	1.7516	Instructions per cycle.
sim_CPI	0.5709	Cycles per instruction.
sim_IPB	13.1948	Instruction per branch.

Total no. of cycles = no of instruction executed * (cycles / instruction)

For convolution algorithm,

Total no. of cycles = 96741 * 0.5709

= 55229.4369 ~ 55230

Execution time = Total number of cycles/operating frequency of µp

= 55230 / 1.8 GHz

= 30.6830205 µsec

5 FFT ALGORITHMS ON HYRID ARCHITECTURE

5.1 HYBRID ARCHITECTURE (FPGA & DSP)

5.1.1 Architecture

Programmable digital signal processors have been utilized in a wide range of signal processing applications. They have been designed with optimized instruction sets to execute digital signal processing algorithms like FFTs and finite impulse response (FIR) filters. Unfortunately, programmable digital signal processor performance has not kept up with the demands of the newest system applications, which often require dramatically higher data rates and increased channel counts. This has forced system designers to implement costly arrays of digital signal processors to satisfy these needs. However, these arrays tend to occupy more board real estate and require increased power consumption, which affects the overall system cost and poses significant implementation challenges, including the arbitration of shared memory between different processors. Hybrid architecture can be explained in the following figure 7.

Xilinx provides designers with the flexibility to implement an FPGA co-processor design that easily interfaces to a wide range of digital signal processors or general purpose processors (GPPs). This co-processor model can be adopted to fit virtually any target application because of the programmable nature of the FPGA's device fabric. Additionally, designers are able to customize and construct functions in a way that fully exploits the parallel nature of a hardware implementation within the FPGA, enabling power-efficient multichannel designs (useful in communication systems) with high data throughputs. The following steps provide a high-level description of the FPGA co-processor design flow:

1. Profile applications in software to identify computationally intensive algorithms suitable for off-loading to co-processors.
2. Integrate an off-the-shelf co-processor to develop a custom co-processor block using a hardware design language like Handel-C or using a hardware description language (HDL).

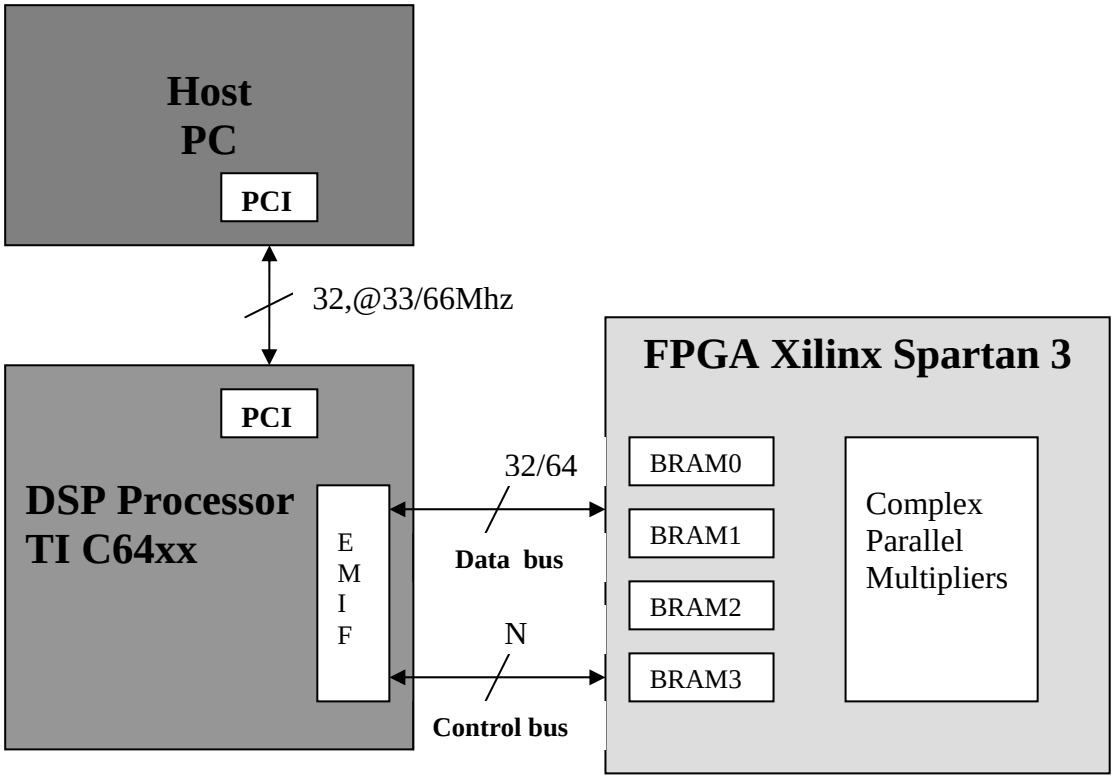


Figure 7: Hybrid architecture

3. Evaluate co-processor system architectures and select a suitable processor interface.
4. Integrate the hardware and software design components.
5. Verify the system in simulation and hardware.

The FFT co-processor [10] in this reference design is a relatively simple example. For larger co-processing systems that may consist of several co-processor functions, design considerations need to be made to maximize the data processing within the FPGA. The larger co-processing reduces the data-transfer overhead between the digital signal processor and the FPGA relative to the data processing time, thereby maximizing the overall system throughput performance. Several enhancements to the solution presented in this reference design can be considered if the user is starting a new board-level FPGA co-processor design. First, both the TI digital signal processor and FPGA can be integrated on the same board. Care must be taken when routing board-level interconnections between the TI digital signal processor and the FPGA to ensure that the maximum data throughput and clock rate of the EMIF can be leveraged to reduce the

round-trip delay time. The current C64x family of digital signal processors can support up to 64-bits of EMIF data at clock rates up to 133 MHz.

5.2 FPGA INTERFACE TO THE TMS320C6000 DSP PLATFORM USING EMIF

To describe an interface between Virtex™-II, Virtex-II Pro™, or Spartan™-3 devices to a Texas Instruments TMSC6000 DSP platform. The External Memory Interface (EMIF) [11] in the TMS DSP platform is used as the interface to the FPGA. Normally, the EMIF connects to different types of memory devices (SRAM, Flash RAM, DDR-RAM, etc.). In this application note, the EMIF connects to the FPGA, making the FPGA perform as a coprocessor, high-speed data processor, or high-speed data transfer interface. Texas Instruments has published EMIF application notes for memory designs. The design interface example is a seamless connection to the FPGA block RAM. One side of the dual-port block RAM is used to communicate with the DSP in Read/Write, FIFO, or memory mode. The other side is used for communication with internal FPGA logic or processor(s).

The flexibility of the FPGA makes it possible to create different designs, performing as different types of memory, with selectable bus widths (8-bit to 64-bit). Interfaces can be designed for the FPGA to work as synchronous or asynchronous standard memory, or as synchronous or asynchronous FIFOs. Interfaces can be designed for the FPGA to interface synchronously or asynchronously with the EMIF interface. In synchronous mode, the ECLKOUTx clock is used to clock the FPGA interface logic. It is even possible to clock the entire FPGA from this clock. An FPGA possesses an enormous amount of processing power using its logic functions, dedicated multipliers, processors (PPC405 or MicroBlaze), etc. The FPGA can thus serve as a co-processor or a high-speed data processing and transfer device. The memory size of an FPGA is smaller than the possible addressable memory space in the TMSC64x type DSP. FPGA memory must be assembled using the available FPGA block RAM. The TMSC64x to FPGA interface described is of a FIFO type structure and an FPGA interface reacting as a memory block.

5.2.1 TMSC64x to FPGA Interface Signals

For a FIFO interface, the standard TMSC64x EMIF FIFO interface scheme can be used. This scenario is explained in figure-8.

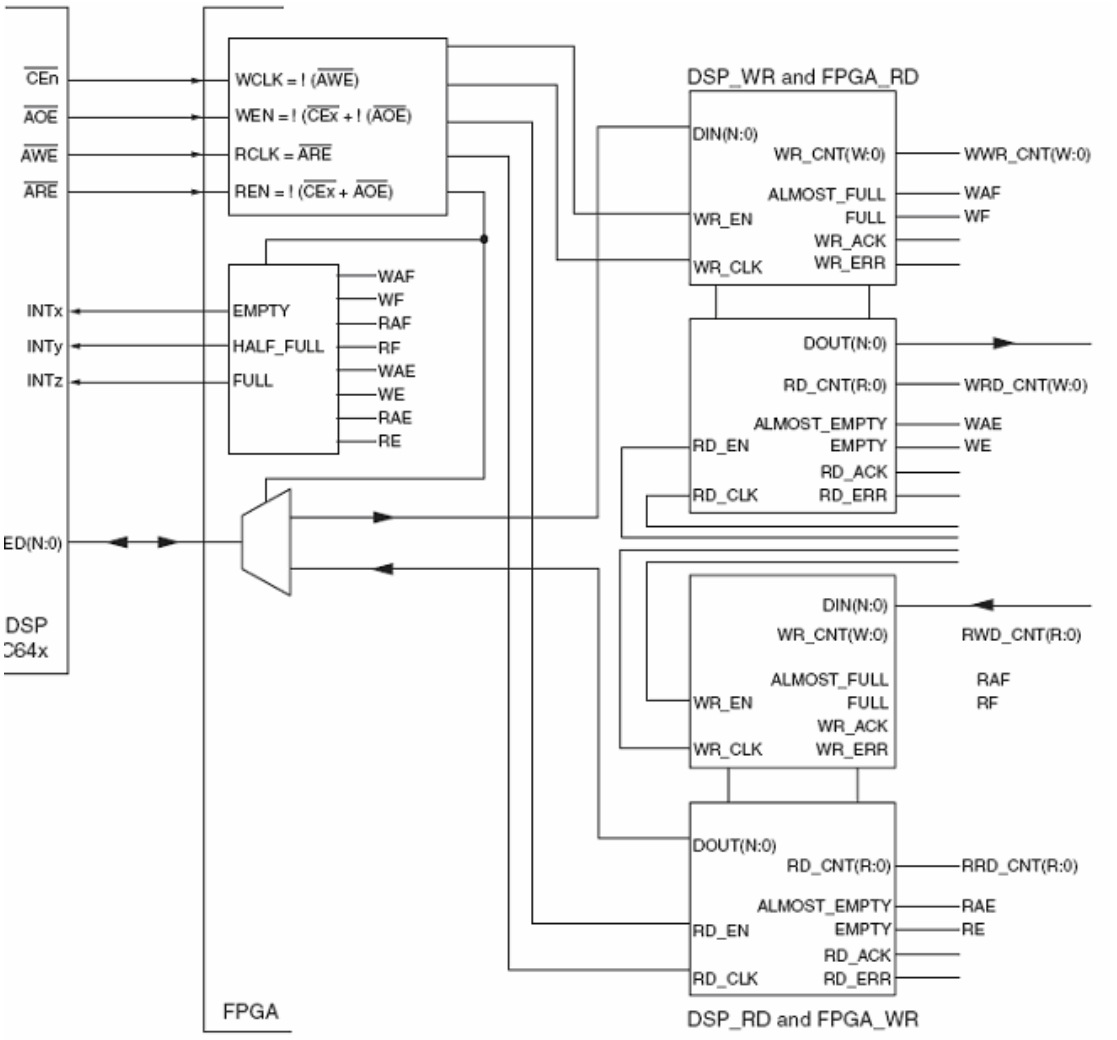


Figure 8: FIFO Based EMIF Interface Design Example

- EMIF signals:
  - CEn DSP Output : EMIF active-Low chip select for memory space
  - AOE DSP Output : Active-Low o/p enable for memory interface
  - AWE DSP Output : Active-Low write strobe for memory interface
  - ARE DSP Output : Active-Low read strobe for memory interface
  - INTx DSP Input : Interrupt signal x.
  - INTy DSP Input : Interrupt signal y.
  - INTz DSP Input : Interrupt signal z.

- ED[63:0] DSP Bidirectional : EMIF 64-bit,32-bit or 16-bit data bus I/O

The FIFO requires a contiguous read clock and continuous write clock. These clocks are generated from the ARE and AWE signals, and are routed using the local clocking capabilities of the FPGA. After exchange of all required hand-shake signals data transfer starts between DSP and FPGA.

5.2.2 Block ram

All recent Xilinx architectures have access to block memories. These 4Kbit blocks in the Virtex, Virtex-E, and Spartan-II devices were increased in size to 18Kbit blocks in the Virtex-II, Virtex-II Pro, and Spartan-3 devices. The blocks are fully synchronous, true dual-port memories. The user can read from or write to each port independently (with the exception of simultaneous reads and writes to the same address). In addition, each port has a separate clock, and the data widths for each port are independently programmable.

In the proposed algorithm of 10 stage FFT, use have to forward 1024 inputs from DSP to FPGA, through EMIF. Generally, EMIF consist of 32 or 64 bit wide parallel bus, so user can transmit or receive 1 float operand between DSP and FPGA. Bandwidth of EMIF interface is 1 operand per clock cycle. For 1024 inputs EMIF whole process of either transmitting or receiving operands to and from DSP to FPGA will take 1024 clock cycles.

5.3 THEORETICAL CALCULATION OF PERFORMANCE:-

5.3.1 ALGORITHMS DESIGN USING DSP AND FPGA

Image will be transferred from the Host Pc to DSP.

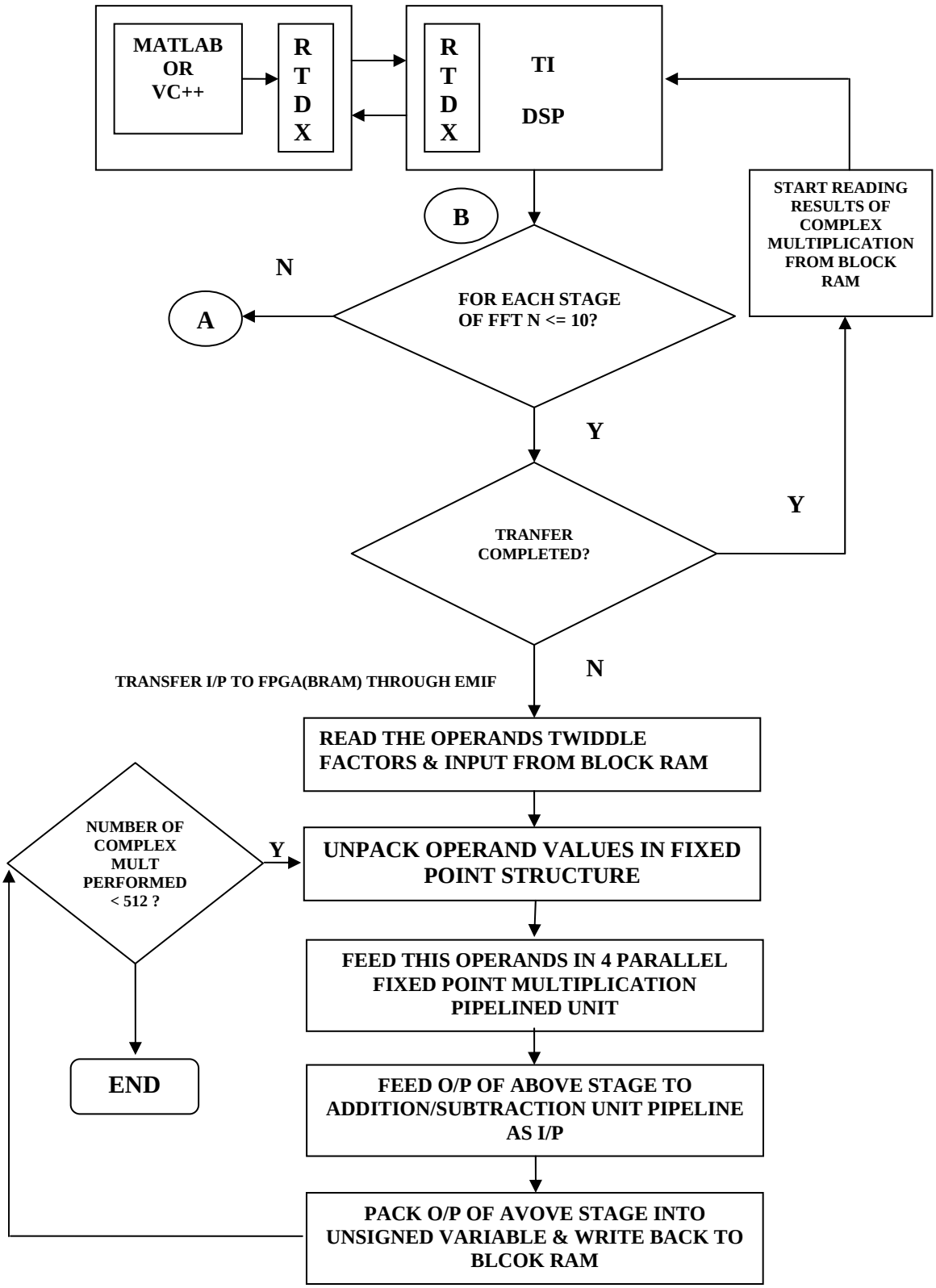
a. Data transfer to and from is done through EMIF from DSP side and Block-ram at the FPGA side.
b. In this algorithm number of inputs are 1024 and already calculated complex twiddle factors has been used, also a module of calculation of FFT algorithm will be divided in two parts.

c. For the particular FFT algorithm summation operation is done by the DSP processor and the multiplication which can be run parallel, are done on FPGA. Which will include following data transfers:

1. Data transfer between DSP and FPGA (only inputs).
2. Data transfer between FPGA and DSP (only outputs).

- For performing 1-complex multiplication 4 fixed point multiplications are required.
- As this experiment on FPGA (Spartan-3) simulator, one can able to compile 64 parallel fixed point multiplications. After getting one complex multiplication output, this algorithm proceeds in pipelining manner for further multiplication. So, at each clock cycles four fixed point multiplication can be achieved.
- To accomplish this task, 1024 fixed point operands (32-bit-operand) should be transferred from DSP RAM to FPGA block RAM.
- Those operands are to be read from Block RAM to an array and perform parallel multiplication and write back the results to block RAM.
- After getting results of multiplications, those results are to be written to DSP RAM to start addition of single stage FFT.
- According to research papers latency of EMIF is 1 cycle per 32-bit operand. To transfer 2K operands, it will take 2K cycles for this transfer.
- Generally block RAMs are dual port RAMs, so one can at a moment read operands in to an array of FPGA as well as write operands in to Block RAM.

The following figure-8 explains fft algorithm division between DSP and FPGA. Schematic diagram representation is explained in figure-9.


MATLAB OR VC++
RTDX
RTDX
TI DSP
START READING RESULTS OF COMPLEX MULTIPLICATION FROM BLOCK RAM
B
N
A
FOR EACH STAGE OF FFT N <= 10?
Y
Y
TRANFER COMPLETED?
N
TRANSFER I/P TO FPGA(BRAM) THROUGH EMIF
READ THE OPERANDS TWIDDLE FACTORS & INPUT FROM BLOCK RAM
NUMBER OF COMPLEX MULT PERFORMED < 512 ?
Y
UNPACK OPERAND VALUES IN FIXED POINT STRUCTURE
FEED THIS OPERANDS IN 4 PARALLEL FIXED POINT MULTIPLICATION PIPELINED UNIT
END
FEED O/P OF ABOVE STAGE TO ADDITION/SUBTRACTION UNIT PIPELINE AS I/P
PACK O/P OF AVOVE STAGE INTO UNSIGNED VARIABLE & WRITE BACK TO BLCOK RAM

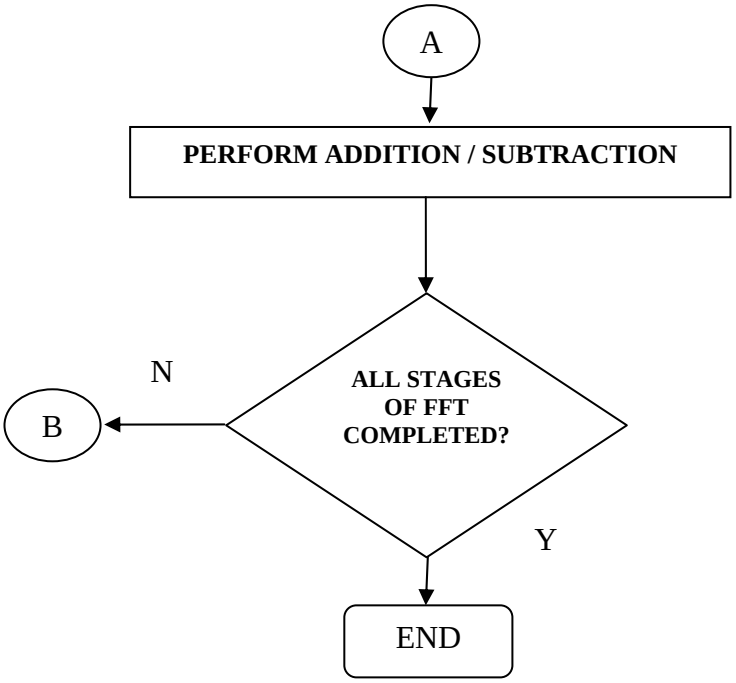


Figure: 9 Flow chart of 1024 point 1-D FFT.

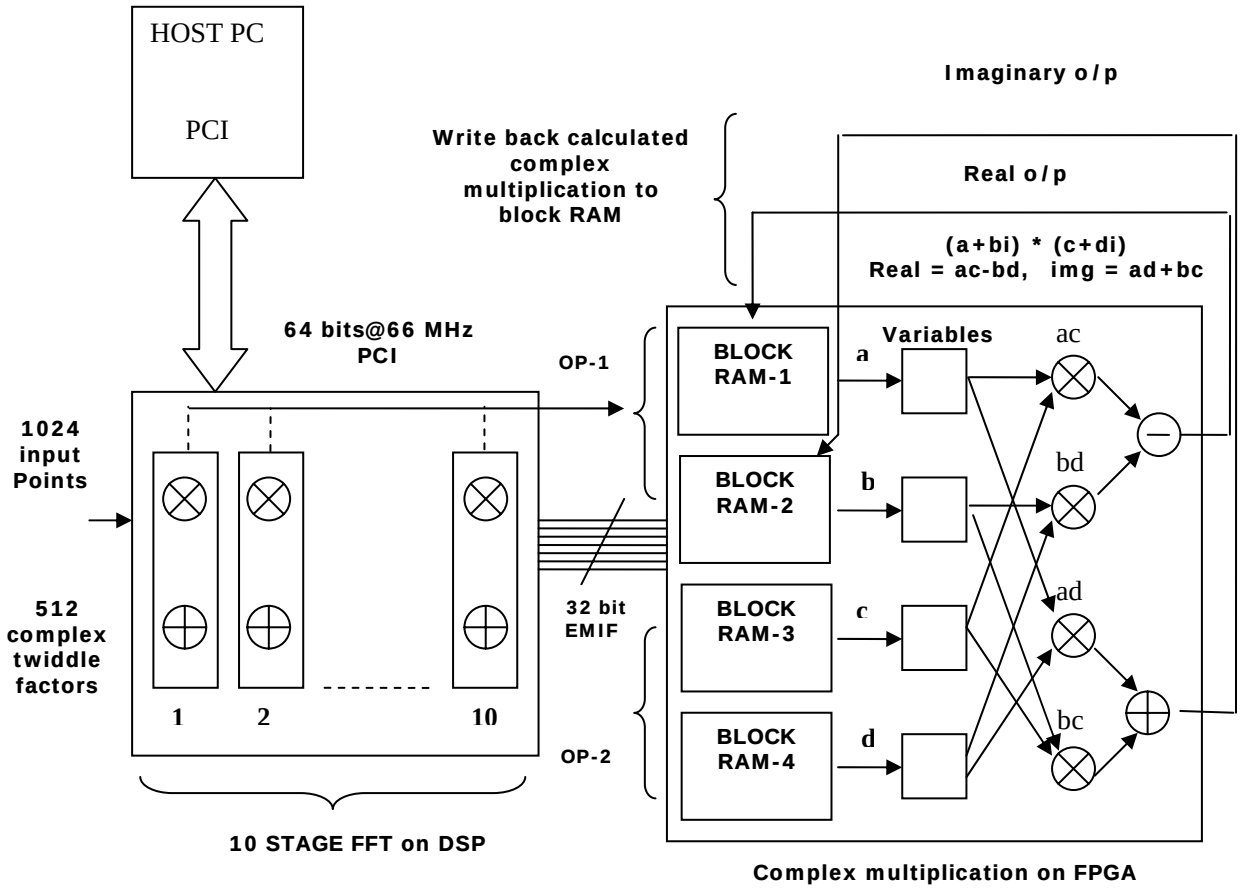


Figure 10: 1024 points 1-D FFT on DSP & FPGA architecture

For calculating 10 stage FFT for 1024 inputs, Total time for execution of FFT algorithm

= Data transfer between DSP and FPGA (only inputs) +
Execution time of multiplication on FPGA +
Data transfer between FPGA and DSP (only outputs).

= 1024 + 515 + 1024

Here one thing is to be noted, that as and when multiplication are getting done in FPGA, simultaneously result of those multiplication are to be sent from FPGA to DSP. So, if we don't consider 515 multiplication time then total time for execution of single stage FFT algorithm will be,

= 1024 + 1024

= 2048 for one FFT stage,

= 2048 * 10 for all 10 stages,

= 20480

How much time 10 stage FFT with 1024 points inputs will take on hybrid architecture, can be shown from following table-5.

TABLE 5: TIMES TAKEN ON HYBRID ARCHITECTURE

1,33,000	Clock cycles on Single DSP.
33,062	Clock cycles without multiplications on DSP.
20,480	Clock cycles multiplication on FPGA.
53,542	Clock cycles with hybrid architecture.
(1,33,000-53,542) = 79,458	Clock cycles benefit in comparison with single DSP.

- 133,000 clock cycles on Single DSP.
- 33,062 clock cycles without multiplications on DSP.
- 20,480 clock cycles multiplication on FPGA.
- 53,542 clock cycles with hybrid architecture.
- (133,000 - 53,542) = 79,458 clock cycles benefit in comparison with single DSP.

Depending on the result of the table-5, one can conclude that rather using image processing on single DSP processor, if such algorithms are executed on hybrid architecture such as DSP & FPGA, in which FPGA will serve the purpose as a co-processor then number of clock cycles can be saved.

6 RESULTS

6.1 Benchmarking results

Following table-6 shows the benchmarking result of clock cycles taken by several image processing algorithms executed on Texas instrument DSP simulator named Code composer studio and Tiger sharc simulator named Visual DSP++. Also, third column shows the clock cycles taken by image processing algorithms on Pentium-IV processor using simple scalar simulator.

TABLE 6: SIMULATION RESULT FOR VARIOUS IMAGE PROCESSING ALGORITHMS.

Algorithms		TI C64xx 600MHz	TigerSharc 600MHz	P4 1.8GHz
		CCStudio	VDSP++	SimpleScalar
1-D FFT	Clock	43214	57555	63339
	Exe.Time(µsec)	72.02	95.92	35
Convolution	Clock	2065	6629	60361
	Exe.Time(µsec)	3.41	11.09	33.534
Correlation	Clock	3554	9804	61562
	Exe.Time(µsec)	5.9	16.34	34
Median Filter	Clock	2893	9676	52185
	Exe.Time(µsec)	4.821	16.12	28.99
Sobel Operator	Clock	1714	3346	24716
	Exe.Time(µsec)	2.85	5.57	13.71

As we can see in the table, number of clock cycles needed to execute given five algorithms on different processors are shown: Now looking at every row we can realize that TMS320C64xx TI DSP takes quite less clock cycles compare to other DSP & p-4 processor. For every algorithm the clock cycles for TMS320C64xx are as less as twice the other processors. Only in case of 1-D FFT clock cycles are not as less as other algorithms. The strength of DSP is VLIW architecture that means DSP can execute 8 instructions in same clock cycle by packing all instruction in very large

instruction word. For VLIW architecture, all instruction which are packed in long word has to be data independent from each other. In case of FFT the calculations of one stage depends on outcome of previous level. Hence, the data dependency is high in case of FFT which results in large number of instruction, hence more clock cycles.

6.2 1024 point, 1-D Fast Fourier Transform (Hybrid Architecture)

Following table-7 shows the clock cycles details of 1-D FFT algorithm executed on single DSP as well as on hybrid architecture.

Table 7: Theoretical result for 1-D FFT algorithm

Clock cycles	Result
133,000	Clock cycles on SINGLE DSP.
33,062	Clock cycles without multiplications on DSP.
20,480	Clock cycles multiplication on FPGA.
53,542	Clock cycles executed on HYBRID ARCHITECTURE.
(133,000 - 53,542) = 79,458	Clock cycles SAVED.

Above table explains that hybrid architecture is far better in executing image processing algorithms in comparison with single DSP, which saves clock cycles as well as execution time.

7 CONCLUSION

Rather than executing Digital Image processing algorithms on single DSP, it's very much efficient to execute such algorithms on hybrid architecture which involves FPGA as a co-processor with DSP. To design DSP with FPGA co-processor hybrid architecture, the selection of apt DSP is required.

Putting together the experimental results from investigation and discussion it could be realized that Texas Instrument DSP (TMS320C64xx) can compute the required DIP algorithms using less clock cycles than the Tiger Sharc DSP (TS-201) processor. Though Pentium-IV processor operates at speed of 1.8 GHz, Texas Instrument DSP operating at speed of 600MHz, gives better performance in executing DIP algorithms, because of its special features like MAC operations and VLIW architecture. Hence, it could be concluded that Texas Instruments DSP should be used in hybrid architecture.

A novel approach to calculate FFT calculation has been proposed which exploits the parallelism using FPGA. Bottleneck of the improvement in cycles is communication latency of EMIF between DSP and FPGA. If this latency can be improved, better results could be achieved.

REFERENCES

[1] Stephen t. Barnard and martin a. Fischler. “Computational Stereo”. SRI International, Menlo Park, California 94025, ACM Computing Surveys (CSUR), Volume 14.

[2] Naveed Bin Rais. Hammad A. Khan, Dr. Farrukh Kamran. Dr. Habibullah Jamal. “A new algorithm of Stereo matching using Epipolar Geometry”, International Journal of Computer Vision.

[3] Richard Hartley. “Multiple View Geometry in Computer Vision”, University of Oxford, UK.

[4] “A BDTI analysis of Texas Instrument TMS320C64x DSP”, http://www.bdti.com/articles/c64_summary_report.pdf, Berkely Design Technology, Inc.

[5] “TMS320C64x Technical Overview and architecture”, focus.ti.com/lit/ug/spru395b/spru395b.pdf, Texas Instruments.

[6] “Architecture of analog DSP processor ADSP-TS201S”, http://www.analog.com/UploadedFiles/Data_Sheets/ADSP_TS201S.pdf, White paper from Analog Devices.

[7] John Stevenson. “Code Composer Studio IDE v2 White Paper”, http://focus.ti.com/dsp/docs/dspsupporttechdocsc.tsp?sectionId=3&tabId=409&abstractName=spra004, Texas Instruments.

[8] “Cross-Platform Software Development with Intel Integrated Performance Primitives”, http://isdlibrary.inteldispatch.com/isd/107/cross-arch_with_ipp.pdf, White paper from Intel Software Development Products.

[9] Glenn Hinton, Dave Sager, Mike Upton, Darrell Boggs, Doug Carmean, “The Micro architecture of the Pentium-4 Processor”, http://www.intel.com/technology/itj/q12001/pdf/art_2.pdf, Intel Technology Journal Q1.

[10] Luxin Yan, Tianxu Zhang and Sheng Zhong, ”A DSP/FPGA -Based Parallel Architecture for Real-time Image Processing”, Proceedings of the 6th World Congress on Intelligent Control and Automation, June 21 - 23, 2006, Dalian, China.

[11] Marc Defossez, “FPGA Interface to the TMSC6000 DSP Platform Using EMIF”. Application Note: Virtex-II Pro Family.

[12] Rob Williams, “SimpleScalar CPU Simulator”, http://www.cems.uwe.ac.uk/~rwilliam/ACA_ufeEHK-20-3/Worksheets/simplescalar.pdf

APPENDIX-A

TMS320C64xx TEXAS INSTRUMENT DSP PROCESSOR

Internal architecture and Peripherals

- Two general-purpose register files (A and B)
- Eight functional units (.L1, .L2, .S1, .S2, .M1, .M2, .D1, and .D2)
- Two load-from-memory data paths (LD1 and LD2)
- Two store-to-memory data paths (ST1 and ST2)
- Two data address paths (DA1 and DA2)
- Two register file data cross paths (1X and 2X)
- Harvard architecture of memory (program memory and data memory)
- L1D 80-KB & L1P 32-KB while L2 is of 128-KB cache
- CPU Operating speed is 600 MHz
- Enhanced direct memory access controller
- Peripheral component interconnect
- Universal test and operation PHY interface for ATM (UTOPIA)
- External memory interfaces
- Multi-channel buffered serial ports
- Host port interfaces
- 32-bit expansion bus
- Serial RapidIO
- Phase locked loop (PLL)

Applications

Wireless infrastructure (adaptive antennas, base stations, gateways), telecom infrastructure (RAS, PBX, VoIP), digital video (conferencing, surveillance, encoders, imaging (medical, machine vision/inspection, defense/radar/sonar)

Features

- VelociTI.2 architecture extensions with new instructions to accelerate performance in key applications
- Increased parallelism with quad 16-bit and octal 8-bit multiply-accumulate performance

- Improved orthogonality with frequently used instructions available in more functional units
- Double the bandwidth resulting from more registers, wider load/store data paths and enlarged two-level cache

APPENDIX-B

TIGER-SHARC (TS-201) ANALOG DSP PROCESSOR

Key Features

- Up to 600MHz, 1.67 ns instruction cycle rate 24M bits of internal—on-chip—DRAM memory
- 25 mm × 25 mm (576-ball) thermally enhanced ball grid array package
- Dual-computation blocks—each containing an ALU, a multiplier, a shifter, a register file, and a communications logic unit (CLU)
- Dual-integer ALUs, providing data addressing and pointer Manipulation
- Integrated I/O includes 14-channel DMA controller, external port, four link ports, SDRAM controller, programmable flag pins, two timers, and timer expired pin for system integration
- 1149.1 IEEE-compliant JTAG test access port for on-chip emulation
- Single-precision IEEE 32-bit and extended-precision 40-bit floating-point data formats and 8-, 16-, 32-, and 64-bit fixed-point data formats
- Eight 16-bit MACs/cycle with 40-bit accumulation
- Two 32-bit MACs/cycle with 80-bit accumulation
- Specific support for Viterbi decoding through the implementation of add compare-select (ACS) sequencing
- Add-subtract instruction and bit reversal in hardware for FFTs
- IEEE floating-point compatible highly integrated
- Three variants offering 24-Mb, 12- Mb, and 4-Mb on-chip embedded DRAM
- Glue less multiprocessing
- Four link ports—1 GBps transfer rate each
- 64-bit external port, 125 MHz, 1 GBps
- 14 DMA channels Flexible Programming in Assembly and C Languages
- User-defined partitioning between program and data memory
- 128 general-purpose registers

- Algebraic assembly language syntax
- Optimizing C compiler
- VisualDSP++ tools support
- Single-instruction, multiple-data (SIMD) instructions, or direct-issue capability
- Predicated execution
- Fully interruptible with full computation performance

KEY BENEFITS

- Provides high performance static superscalar DSP operations, optimized for telecommunications infrastructure and other large, demanding multiprocessor DSP applications
- Performs exceptionally well on DSP algorithm and I/O benchmarks Supports low overhead DMA transfers between internal memory, external memory, memory-mapped peripherals, link ports, host processors, and other (multiprocessor) DSPs
- Eases DSP programming through extremely flexible instruction set and high-level-language-friendly DSP architecture Enables scalable multiprocessing systems with low communications Overhead
- Provides on-chip arbitration for glue less multiprocessing