

TMS320C64x

Technical Overview

Literature Number: SPRU395B
January 2001



Printed on Recycled Paper

IMPORTANT NOTICE

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgment, including those pertaining to warranty, patent infringement, and limitation of liability.

TI warrants performance of its semiconductor products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Customers are responsible for their applications using TI components.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards must be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance or customer product design. TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, warranty or endorsement thereof.

Mailing Address:

Texas Instruments
Post Office Box 655303
Dallas, Texas 75265

Contents

1	Introduction	1-1
1.1	Introduction	1-2
1.2	Application Areas	1-6
1.2.1	Digital Communications	1-6
1.2.2	Image Processing Applications	1-8
2	Architecture	2-1
2.1	Architectural Overview	2-2
2.1.1	C6000 CPU	2-2
2.1.2	Register File Enhancements	2-3
2.1.3	Functional Units	2-4
2.1.4	Register File Paths	2-6
2.1.5	Memory, Load and Store Paths	2-8
2.2	Unique Features of the C64x	2-9
2.2.1	Packed Data Processing	2-9
2.2.2	Additional Functional Unit Hardware:	2-10
2.2.3	Increased Orthogonality	2-12
2.3	C64x Instruction Set Extension Details	2-13
2.4	Chip Level Features	2-16
2.4.1	Two-Level Cache Architecture	2-16
2.4.2	Powerful Enhanced DMA Controller	2-17
2.4.3	Three External Buses	2-18
2.4.4	Flexible Serial Connections	2-20
2.4.5	The UTOPIA Port	2-20
2.4.6	General Purpose Input/Output	2-21
2.4.7	Additional Peripheral Information	2-21
2.5	Ease of Development	2-22
2.6	Summary	2-24
2.6.1	Register File Enhancements	2-24
2.6.2	Data Path Extensions	2-24
2.6.3	Packed Data Processing	2-24
2.6.4	Additional Functional Unit Hardware	2-24
2.6.5	Increased Orthogonality	2-25
A	Sum of Products Example	A-1
A.1	Sum of Products Example	A-2

B Image Processing Kernel Code Examples B-1
 B.1 Threshold Example B-2
 B.2 Motion Estimation Example B-4

C Glossary C-1

D Related Documents D-1

Figures

1-1	C62x/C67x and C64x CPUs	1-4
2-1	CPUs for VelociTI and VelociTI.2	2-3
2-2	C64x Data Cross Paths	2-7
2-3	C64x Memory Load and Store Paths	2-8
2-4	C64x L1/L2 Cache	2-16
2-5	EDMA	2-18
2-6	An Initial C64x Implementation – The C6415 DSP	2-20
B-1	Threshold Example	B-2
B-2	Motion Estimation Example	B-4

Tables

1–1	Digital Communications Benchmarks	1-8
1–2	Image/Video Processing Benchmarks	1-9
1–3	DSP and Image Processing Kernels	1-10
2–1	C6000 Register File	2-4
2–2	Functional Units and Operations Performed	2-5
2–3	Quad 8-bit and Dual 16-bit Instruction Set Extensions	2-10
2–4	C64x Special Purpose Instructions	2-11
2–5	Functional Unit to Additional Instruction Mapping	2-13

Introduction

Topic	Page
1.1 Introduction	1-2
1.2 Application Areas	1-6

1.1 Introduction

We live in a world driven by data: financial data, medical data, sports and entertainment data. In this era, data, be it audio, video, or the written word, are delivered through a single medium. That medium could be wireless technology, satellite broadcasting, cable, or digital subscriber loop (DSL) technology. All these media, however, have one thing in common, the need to process digital data quickly.

The C6000 family with the VelociTI architecture addresses the demands of this new era. First introduced in 1997 with the C62x and C67x cores, the C6000 family uses an advanced very long instruction word (VLIW) architecture. The architecture contains multiple execution units running in parallel, which allow them to perform multiple instructions in a single clock cycle. Parallelism is the key to extremely high performance. At a 200 MHz clock rate and 1600 million instructions per second (MIPS) at introduction, the C6201 achieved ten times the performance of earlier digital signal processing (DSP) solutions. Today, the C62x device family can achieve 2400 MIPS at clock rates of 300 MHz, which is a performance increase of 50% over the past three years.

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 1.1 GHz and greater, the C64x can process information at a rate of 8800+ MIPS or nearly nine billion instructions per second. Initial devices will be sampling in the 600 MHz-800 MHz range, giving performance levels of 4800-6400 MIPS. In addition to clock rate, more work can be done each cycle with the VelociTI.2 extensions to the VelociTI architecture. These extensions include new instructions to accelerate performance in key applications and extend the parallelism of the architecture.

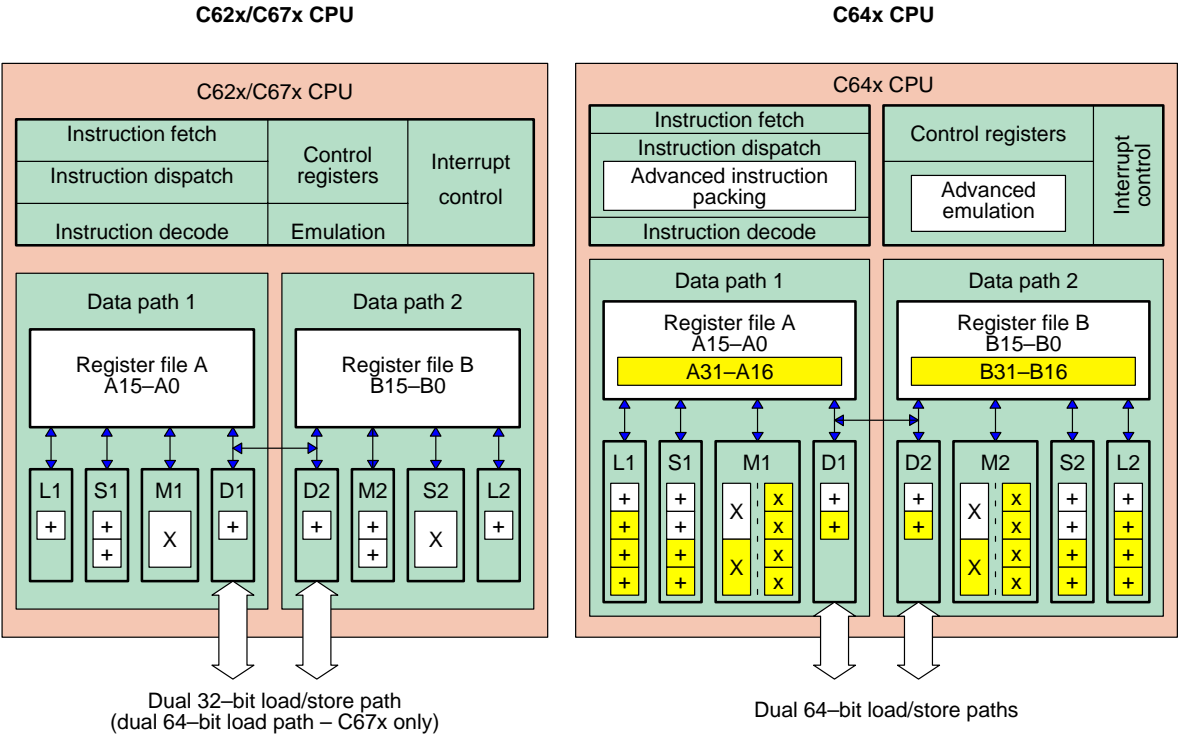
Increased clock rate and increased CPU throughput are only part of the solution. Processing data at these extremely high rates increases the need for I/O bandwidth. Initial C64x devices have three external buses with speeds of up to 133MHz. Each bus has a primary mission. One provides a fast glueless interface for synchronous and asynchronous memories at data rates as fast as 1.1Gbytes/sec. Another bus interfaces to slow peripheral devices and the third bus provides a port to support industry standard host interfaces. 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 in-

structions 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.

The C64x central processing unit (CPU), 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. This doubles the 16-bit multiply rate of the C62x/C67x; four 16-bit x 16-bit multiplies 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. Eight-bit data is common in the field of image processing, one of the application areas served by the C64x.

Figure 1–1. C62x/C67x and C64x CPUs



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, 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. The C64x goes beyond building extensions in the hardware. 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%.

In summary, the C64x VelociTI.2 is object-code-compatible with the C62x, yet contains key extensions to the existing C62x VelociTI architecture in several areas:

- ☐ Register file enhancements
- ☐ Data path extensions
- ☐ Packed data processing
- ☐ Additional functional unit hardware
- ☐ Increased orthogonality

These enhancements are examined in the Architectural Overview Section.

1.2 Application Areas

This section focuses on two application areas whose performance is greatly enhanced by the C64x VelociTI.2 extensions to the C62x/C67x architecture.

1.2.1 Digital Communications

The popularity of the Internet and its pervasiveness in every day life has grown tremendously in the past three years. Today you can make plane reservations, arrange for a gift to be sent to your host, and pay the bill for both purchases while sitting in front of your computer.

This type of data interaction has given rise to a technology called DSL (digital subscriber loop). DSL has been developed to deliver high speed communication services over the existing communications infrastructure (the local loop). The same copper telephone wires that come into your home can be used to bring in massive amounts of data required by your connection to the Web. In particular, this type of interaction is asymmetric; you are receiving much more data from your Internet Service Provider (ISP) than you are sending back. This type of DSL is called ADSL (asymmetric digital subscriber loop). The data rates achieved by this technology are 8M bits/sec from the ISP to you (downstream) and 800K bits/sec from you to the ISP (upstream).

The C6000 is the processing engine of choice in many ADSL solutions today. Specific features have been included in the C64x to further enhance the suitability of the C6000 processor family for ADSL solutions. ADSL signal processing tasks that will significantly benefit from the C64x extensions include FFT/IFFT, Reed Solomon Encode/Decode, Circular Echo Synthesis filter, Constellation Encode/Decode, Convolutional Encode, Viterbi Decode, and various other operations.

Another delivery mechanism for broadband communications is the cable modem. The cable modem utilizes the cable network that delivers cable TV to over 100M people in the US alone. The C64x capabilities will also significantly advance the development of cable modem solutions. The enhancements for operations such as Reed Solomon Encode/Decode listed above will also benefit cable modems. In addition, specific features have been included that enhance the performance of operations such as Sample Rate Conversion, Byte to Symbol Conversion and LMS (Least Mean Square) Equalization.

It is anticipated that the significant architectural extensions of the C64x, coupled with the increased clock rate, will enable several new innovative solutions. Future DSL standards will offer even higher data rates than ADSL (13-52M bits/sec downstream and 1.5 to 2.3M bits/sec upstream). Another solution made possible by this breakthrough is that larger numbers of multiple

modems can be connected with a single processor in central office applications. Alternatively, residential modems will contain significant additional processing capability so that other functions, such as media decoding, will be performed on the same digital signal processor.

Another example of digital communications is the wireless revolution. Everywhere you look someone is using a cell phone. What was once a communications device for a few individuals is now common place. The increase of wireless communications usage requires that the support infrastructure be improved immensely. The basestations must handle higher call volumes and wider calling areas, which means more channels at higher frequencies. The C62x has been widely adopted by the wireless basestation market place. The C62x can be found in 3G (third generation) basestation transceivers, smart antennas, wireless local loop basestations and wireless LANs (Local Area Networks).

Using the basestation transceiver as an example, the data rate frequency is 2.4 GHz and is down-sampled to 6 to 12 MHz. Four channels need to be processed every burst period. The key functions performed by the DSP are FFTs, channel and noise estimation, channel correction and interference estimation and detection.

Table 1–1 contains the benchmark results for our current performance on some of the key algorithms in broadband communications and wireless communications. These algorithms, collectively referred to as digital communications, are listed in Table 1. Ratios for cycle count and total performance improvements on C64x relative to C62x are shown in the table. The total performance ratio combines the cycle count improvement ratio with the clock rate improvement for a 600 MHz C64x relative to a 300 MHz C62x.

Table 1–1. *Digital Communications Benchmarks*

Digital Communications	Cycle Performance Improvement Ratio C64x:C62x	Total Performance Improvement 600 MHz C64x vs 300 MHz C62x
Byte to Symbol Conversion (Cable Modem)	15.6x	31.2x
FFT – Radix 4 Complex (ADSL)	2.1x	4.2x
LMS Equalizer (Cable Modem)	2.0x	4.0x
Reed Solomon Decode: Chien Search (ADSL, Cable Modem)	4.7x	9.4x
Reed Solomon Decode: Forney Algorithm (ADSL, Cable Modem)	3.2x	6.4x
Reed Solomon Decode: Syndrome Accumulation (ADSL, Cable Modem)	3.7x	7.4x
Reed Solomon Decode: Berlekamp Massey Algorithm (ADSL, Cable Modem)	2.0x	4.0x
Time Domain Equalizer (ADSL)	2.0x	4.0x
Viterbi Decode (GSM)	2.7x	5.4x

1.2.2 Image Processing Applications

Thus far we have examined broadband and wireless communications technology but visual communication is equally dominant in this era of data convergence. The C62x/C67x processors are currently found in many image processing application areas such as motion video, network cameras, raster image printers, digital scanners, visual inspection systems, radar/sonar and medical image processing. These processors perform image compression, image transmission, pattern and optical character recognition, encryption and image enhancements. The C64x with its 8-bit and 16-bit extensions further amplifies the ability of the C6000 family in image processing applications.

Table 1–2 contains a summary of current performance for some key image/video processing benchmarks. Ratios for cycle count and total performance improvements on C64x relative to C62x are shown in the table. The total performance ratio combines the cycle count improvement ratio with the clock rate improvement for a 600 MHz C64x relative to a 300 MHz C62x.

Table 1–2. Image/Video Processing Benchmarks

Image/Video Processing	Cycle Performance Improvement Ratio C64x:C62x	Total Performance Improvement 600 MHz C64x vs 300 MHz C62x
3 x 3 Correlation	3.5x	7.0x
3 x 3 Median Filter	4.2x	8.4x
IDCT – 8x8 ϕ	1.8x	3.6x
Morphology – Gray Scale Dilation	6.3x	12.6x
Morphology – Gray Scale Erosion	5.7x	11.4x
Motion Compensation	7.1x	14.2x
Motion Estimation – 8x8 MAD	7.6x	15.2x
Object Perimeter Computation	4.8x	9.6x
Polyphase Filter – Image Scaling	2.3x	4.6x
Thresholding	3.9x	7.8x
ϕ – The IDCT implementation is IEEE 1180-1990 compliant.		

Table 1–3 provides a summary of kernels that are common building blocks used in digital communications and/or image/video processing applications. Ratios for cycle count and total performance improvements on C64x relative to C62x are shown in the table. The total performance ratio combines the cycle count improvement ratio with the clock rate improvement for a 600 MHz C64x relative to a 300 MHz C62x.

Table 1–3. DSP and Image Processing Kernels

DSP Kernels/Image Processing Kernels	C62x Cycle Count	C64x cycle count	Cycle Performance Improvement Ratio C64x:C62x	Total Performance Improvement 600 MHz C64x vs 300 MHz C62x
Correlation – 3x3	4.5 cycles/pixel	1.28 cycles/pixel	3.5x	7.0x
FFT – Radix 4 – Complex(size = N log (N))	12.7 cycles/data	6.0 cycles/data	2.1x	4.2x
Median Filter – 3x3	9.0 cycles/pixel	2.1 cycles/pixel	4.3x	8.6x
Motion Estimation – 8x8 MAD	0.953 cycles/pixel	0.126 cycles/pixel	7.6x	15.2x
Polyphase Filter – Sample Rate Conversion	1.02 cycles/output/ filter tap	0.51 cycles/output/ filter tap	2.0x	4.0x
Polyphase Filter – Image Scaling	0.77 cycles/output/ filter tap	0.33 cycles/output/ filter tap	2.3x	4.6x
Reed Solomon Decode: Syndrome Accumulation	1680 cycles/ packet	460 cycles/ packet	3.7x	7.4x
Vector Product	0.5 cycles/data	0.25 cycles/data	2.0x	4.0x
Viterbi Decode (GSM) (16 states)	38.25 cycles/output	14 Ψ cycles/output	2.7x	5.4x
Ψ – Includes traceback				

These benchmarks along with the code that implements them can be found at the following URL: <http://www.ti.com>.

Architecture

Topic	Page
2.1 Architectural Overview	2-2
2.2 Unique Features of the C64x	2-9
2.3 C64x Instruction Set Extension Details	2-13
2.4 Chip Level Features	2-16
2.5 Ease of Development	2-22
2.6 Summary	2-24

2.1 Architectural Overview

Now that we have taken a brief glimpse into a few of the applications areas contained in the era of data convergence, let us take a closer look at the C64x CPU, the processing engine at the center of these applications.

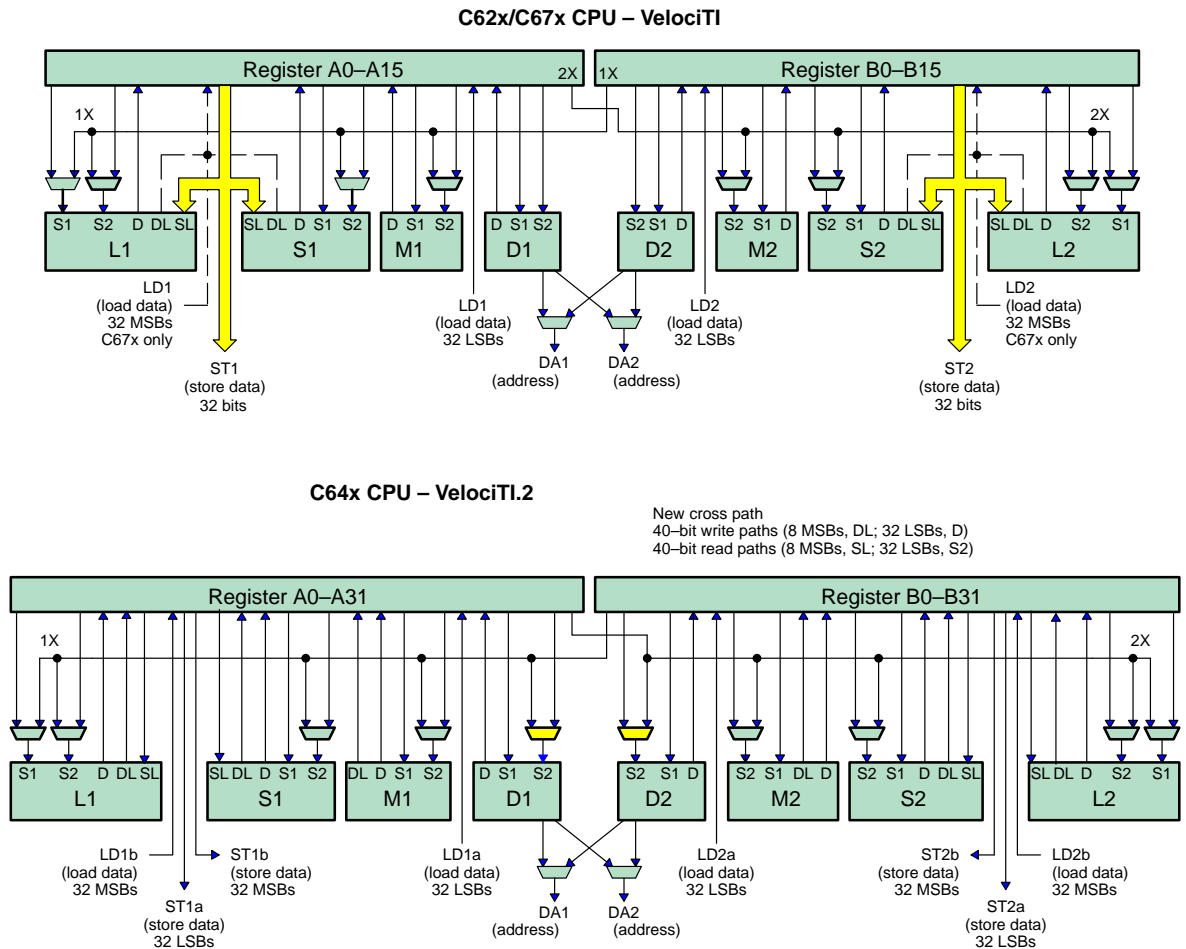
2.1.1 C6000 CPU

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)
- ☐ Two register file data cross paths (1X and 2X)

Figure 2–1 illustrates the CPUs for the VelociTI architecture and VelociTI.2 extensions.

Figure 2–1. CPUs for VelociTI and VelociTI.2



2.1.2 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, 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 C62x/C67x, registers A1, A2, B0, B1, and B2 can be used as condition registers. On the C64x, A0 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 C62x/C67x general-purpose register files support data ranging in size from packed 16-bit data through 40-bit fixed-point and 64-bit floating-point data. 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, shown in Table 2–1, 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.

Table 2–1. C6000 Register File

Register Files		
A	B	
A0	B0	C62x/C64x/C67x
A1	B1	
:	:	
A15	B15	
A16	B16	C64x only
A17	B17	
:	:	
A31	B31	

2.1.3 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 functional units and these differences are described in Table 2–2.

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. The new operations can be found in **boldface** in Table 2–2.

Table 2–2. Functional Units and Operations Performed

Functional Unit	Fixed-Point Operations
.M unit (.M1, .M2)	16 x 16 multiply operations
	16 x 32 multiply operations
	Quad 8 x 8 multiply operations
	Dual 16 x 16 multiply operations
	Dual 16 x 16 multiply with add/subtract operations
	Quad 8 x 8 multiply with add operations
	Bit expansion
	Bit interleaving/de-interleaving
	Galois Field Multiply
	Rotation
	Variable shift operations
.L unit (.L1, .L2)	32/40-bit arithmetic and compare operations
	32-bit logical operations
	Leftmost 1 or 0 counting for 32 bits
	Normalization count for 32 and 40 bits
	Byte shifts
	Data packing/unpacking
	5-bit constant generation
	Dual 16-bit arithmetic operations
	Quad 8-bit arithmetic operations
	Dual 16-bit min/max operations
	Quad 8-bit min/max operations
	Quad 8-bit subtract with absolute value
** Bold type indicates that these fixed-point operations are new.	

Functional Unit	Fixed-Point Operations
.S unit (.S1, .S2)	32-bit arithmetic operations 32/40-bit shifts and 32-bit bit-field operations 32-bit logical operations Branches Constant generation Register transfers to/from control register file (.S2 only) Byte shifts Data packing/unpacking Dual 16-bit compare operations Quad 8-bit compare operations Dual 16-bit shift operations Dual 16-bit saturated arithmetic operations Quad 8-bit saturated arithmetic operations
.D unit (.D1, .D2)	32-bit add, subtract, linear and circular address calculation Loads and stores with 5-bit constant offset Loads and stores with 15-bit constant offset (.D2 only) Load and store double words with 5-bit constant offset Load and store non-aligned words and double words 5-bit constant offset generation 32-bit logical operations Dual 16-bit arithmetic operations
** Bold type indicates that these fixed-point operations are new.	

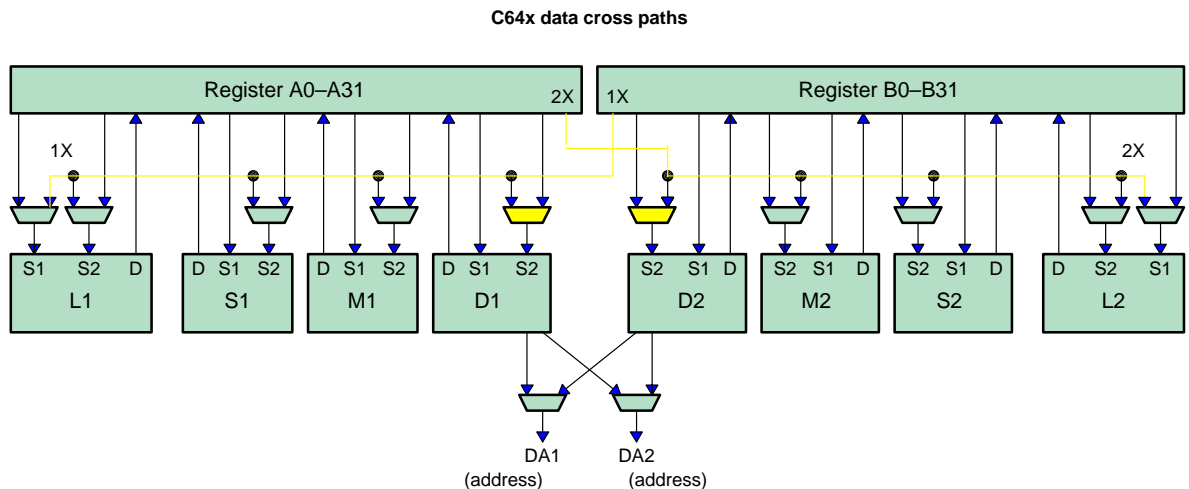
2.1.4 Register File Paths

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 (refer to Figure 2–1). 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.

The register files are also connected to the opposite-side register file's functional units via the 1X and 2X cross paths. These cross paths allow functional units from one data path to access a 32-bit operand from the opposite side's register file. The 1X cross path allows functional units from data path A to read its source from register file B. Similarly, the 2X cross path allows functional units from data path B to read its source from register file A.

Figure 2–2. C64x Data Cross Paths



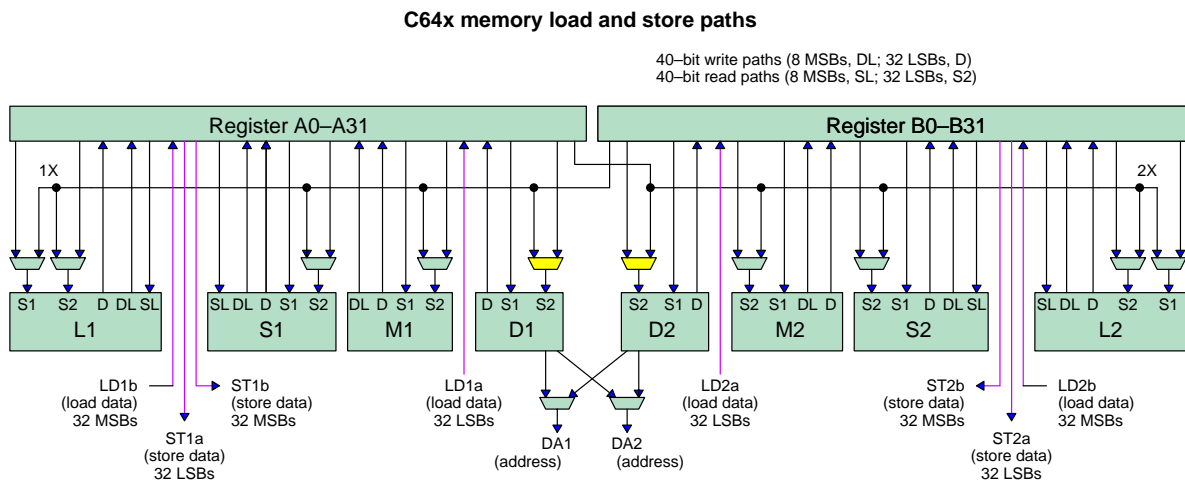
On the C64x, all eight of the functional units have access to the register file on the opposite side via a cross path. The .M1, .M2, .S1, .S2, .D1 and .D2 units' *src2* inputs are selectable between the cross path and the register file found on the same side. In the case of the .L1 and .L2, both *src1* and *src2* inputs are also selectable between the cross path and the same-side register file. For comparison, on the the C62x/C67x, only six functional units have access to the register file on the opposite side via a cross path; the .D units do not have a data cross path.

Only two cross paths, 1X and 2X, exist in the C6000 architecture. Therefore, the limit is one source read from each data path's opposite register file per cycle, or a total of two cross-path source reads per cycle. The C64x pipelines data cross path accesses allow multiple units per side to read the same cross-path source simultaneously. The cross path operand for one side may be used by up to two functional units on that side in an execute packet. In the C62x/C67x, only one functional unit per data path per execute packet could get an operand from the opposite register file.

2.1.5 Memory, Load and Store Paths

The data address paths named DA1 and DA2 are each connected to the .D units in both data paths. Load/store instructions can use an address register from one register file while loading to or storing from the other register file. Figure 2–3 illustrates the C64x memory load and store paths.

Figure 2–3. C64x Memory Load and Store Paths



The C64x supports double-word loads and stores. There are four 32-bit paths for loading data for memory to the register file. For side A, LD1a is the load path for the 32 LSBs; LD1b is the load path for the 32 MSBs. For side B, LD2a is the load path for the 32 LSBs; LD2b is the load path for the 32 MSBs. There are also four 32-bit paths for storing register values to memory from each register file. ST1a is the write path for the 32 LSBs on side A; ST1b is the write path for the 32 MSBs for side A. For side B, ST2a is the write path for the 32 LSBs and ST2b is the write path for the 32 MSBs. Wide loads are essential in sustaining processing throughput.

The C64x can also access words and double words at any byte boundary using non-aligned loads and stores. As a result, word and double-word data

does not always need alignment to 32-bit or 64-bit boundaries as in the C62x/C67x. Non-aligned loads and stores combined with the pack and unpack instructions described earlier, mean that the compiler does not have to format the data to take advantage of the 8-bit and 16-bit hardware extensions. Without these operations, significant effort would be needed to leverage the parallelism. The C64x provides a complete set of data flow operations to sustain the maximum performance improvement made possible by the 8-bit and 16-bit extensions added to the C6000 architecture.

2.2 Unique Features of the C64x

Thus far, we have looked at two areas where the C64x has extended the C62x/C67x VelociTI architecture. Those are register file enhancements (doubling the register file and increasing the data types stored in the register file) and data path extensions (doubling the load-store paths to 64 bits and allowing for non-aligned loads and stores of words/double words).

We will now more closely examine three other areas where the C64x adds unique features to the existing C62x/C67x architecture. Those areas are packed data processing (8-bit and 16-bit instruction set extensions with data flow enhancements), additional functional unit hardware, and increased orthogonality.

2.2.1 Packed Data Processing

Instructions have been added that operate directly on packed data (both 8-bit and 16-bit) to streamline data flow and increase instruction set efficiency. An extensive collection of pack and unpack instructions simplifies manipulation of packed data types. The C64x has a comprehensive collection of 8-bit and 16-bit instruction set extensions. They are included in Table 2–3.

Table 2–3. Quad 8-bit and Dual 16-bit Instruction Set Extensions

Operation	Quad 8-bit	Dual 16-bit
Multiply	X	X
Multiply with Saturation		X
Addition/Subtraction	X	X*
Addition with Saturation	X	X
Absolute Value		X
Subtract with Absolute Value	X	
Compare	X	X
Shift		X
Data Pack/Unpack	X	X
Data Pack with Saturation	X	X
Dot product with optional negate	X+	X
Min/Max/Average	X	X
Bit-expansion (Mask generation)	X	X

* = The C62x/C67x provides support for 16-bit data with the ADD2/SUB2 instructions. The C64x extends this support to include 8-bit data.

+ = Dot product with negate is not available for 8-bit data

Appendix A includes a code example using the dual 16-bit dot product instruction.

2.2.2 Additional Functional Unit Hardware:

Additional hardware has been built into the eight functional units of the C64x to expand their functionality. We have already discussed two important extensions. Each .M unit can now perform two 16x16 bit multiplies or four 8x8 bit multiplies every clock cycle. Also, the .D units can now access words and double words on any byte boundary by using non-aligned load and store instructions. The C62x/C67x only provides aligned load and store instructions.

In addition, the .L units can perform byte shifts and the .M units can perform bi-directional variable shifts in addition to the .S unit's ability to do shifts. The bi-directional shifts directly assist voice-compression codecs (vocoders). The .L units can now perform quad 8-bit subtracts with absolute value. This absolute difference instruction greatly aids motion estimation algorithms.

Special communication-specific instructions, such as SHFL, DEAL and GMPY4, have been added to the .M unit to address common operations in error-correcting codes. Bit-count and rotate hardware on the .M unit extends support for bit-level algorithms such as binary morphology, image metric calculations and encryption algorithms. Table 2–4 contains a listing of these special purpose instructions.

Table 2–4. C64x Special Purpose Instructions

Instruction	Description	Example Application
BITC4	Bit count	Machine vision
GMPY4	Galois Field MPY	Reed Solomon support
SHFL	Bit interleaving	Convolution encoder
DEAL	Bit de-interleaving	Cable modem
SWAP4	Byte swap	Endian swap
XPNDx	Bit expansion	Graphics
MPYHlx, MPYLlx	Extended precision 16x32 MPYs	Audio
AVGx	Quad 8-bit, Dual 16-bit average	Motion compensation
SUBABS4	Quad 8-bit Absolute of differences	Motion estimation
SSHVL, SSHVR	Signed variable shift	GSM

The additional functional unit hardware is key to the improvements in performance that we saw in the benchmarks found in the previous section. For the broadband communications area, the dual 16-bit arithmetic supported by six of the eight functional units paired with a bit reverse (**BITR**) instruction improves FFT (Fast Fourier Transform) benchmarks by a factor of two. The Galois field multiply instruction (**GMPY4**) provides a 4.7 times performance boost for Reed Solomon decoding using the Chien search as compared to the C62x implementation; this improvement increases to 9.4x when you include the clock cycle speed-up of 300 MHz to 600 MHz. The bit interleaving and de-interleaving hardware provides a performance boost for both DSL and cable modem. In fact, the de-interleave hardware helps improve the 64QAM byte to symbol conversion benchmark by a factor of 15.6 as compared to the C62x cycle count.

In the wireless communications area, doubling the number of 16 x 16 multiplies on the C64x doubles the throughput of filtering. The dual 16-bit compare instructions, coupled with the **MAX2/MIN2** instructions and additional registers available to store state variables, gives a 2.7 times performance boost for

GSM Viterbi decoding. The signed variable shifts greatly aid the performance of GSM vocoders.

The 8-bit hardware extensions dramatically improve image/video processing applications. The loop kernels found in these algorithms can operate on 8-bit or 16-bit data. The average instructions improve the performance of motion compensation by a factor of seven on a per clock cycle basis versus the C62x. The quad absolute difference instruction bolsters motion estimation performance by a factor of 7.6 on a per clock cycle basis for an 8x8 minimum absolute difference (MAD) computation. The dual 16-bit and quad 8-bit support and increased clock rate gives image processing applications a 15 times throughput improvement as compared to the C62x implementations (comparing C62x devices in the 150-300 MHz range to C64x devices in the 600 MHz to 1.1 GHz range).

It is important to note that the C64x provides a comprehensive set of data packing and unpacking operations to allow sustained high performance for the quad 8-bit and dual 16-bit hardware extensions. Unpack instructions prepare 8-bit data for parallel 16-bit operations. Pack instructions return parallel results to output precision including saturation support.

2.2.3 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.

2.3 C64x Instruction Set Extension Details

Table 2–5 includes a complete list of the new C64x instructions and the functional unit(s) that perform them. For a complete listing of all the instructions and their usage, please see the C6000 CPU and Instruction Set Reference Guide at the following URL: <http://www.ti.com/>.

Table 2–5. Functional Unit to Additional Instruction Mapping

Instruction	.L unit	.M unit	.S unit	.D unit
ABS2	√			
ADD2 ψ	√		√	√
ADD4	√			
ADDKPC			√	
AND ψ	√		√	√
ANDN	√		√	√
AVG2		√		
AVGU4		√		
BDEC			√	
BITC4		√		
BITR		√		
BNOP			√	
BNOP reg			√	
BPOS			√	
CMPEQ2			√	
CMPEQ4			√	
CMPGT2			√	
CMPGTU4			√	
CMPLT2			√	
CMPLTU4			√	
DEAL		√		
DOTP2		√		
DOTPN2		√		
DOTPNRSU2		√		
DOTPNRUS2		√		
DOTPRSU2		√		
DOTPRUS2		√		
DOTPSU4		√		
DOTPU4		√		
GMPY4		√		
LDDW				√
LDNDW				√
LDNW				√

Instruction	.L unit	.M unit	.S unit	.D unit
MAX2	√			
MAXU4	√			
MIN2	√			
MINU4	√			
MPY2		√		
MPYHI		√		
MPYHIR		√		
MPYIH		√		
MPYIHR		√		
MPYIL		√		
MPYILR		√		
MPYLI		√		
MPYLIR		√		
MPYSU4		√		
MPYUS4		√		
MPYU4		√		
MVD		√		
MVK ψ	√		√	√
OR ψ	√		√	√
PACK2	√		√	
PACKH2	√		√	
PACKH4	√			
PACKHL2	√		√	
PACKL4	√			
PACKLH2	√		√	
ROTL		√		
SADD2			√	
SADDU4			√	
SADDSU2			√	
SADDUS2			√	
SHFL		√		
SHLMB	√		√	
SHR2			√	
SHRMB	√		√	
SHRU2			√	
SMPY2		√		
SPACK2			√	
SPACKU4			√	
SSHVL		√		
SSHVR		√		
STDW				√

Instruction	.L unit	.M unit	.S unit	.D unit
STNDW				√
STNW				√
SUB2 ψ	√		√	√
SUB4	√			
SUBABS4	√			
SWAP2	√		√	
SWAP4	√			
UNPKHU4	√		√	
UNPKLU4	√		√	
XOR	√		√	√
XPND2		√		
XPND4		√		

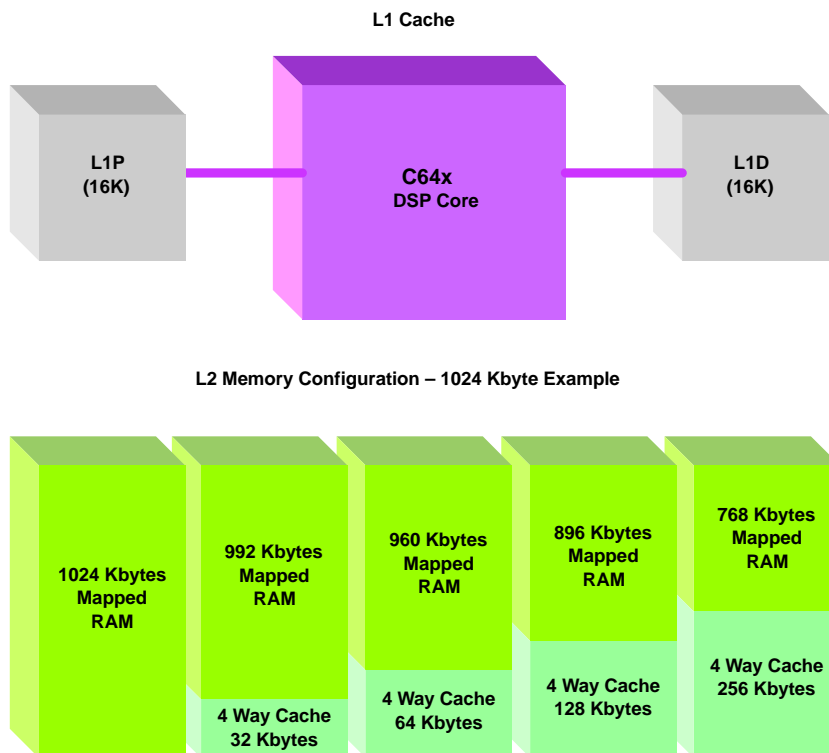
ψ – Indicates instructions that exist on the C62x/C67x but are now available on one or more additional functional units.

2.4 Chip Level Features

2.4.1 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 2–4 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 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.

Figure 2–4. C64x L1/L2 Cache

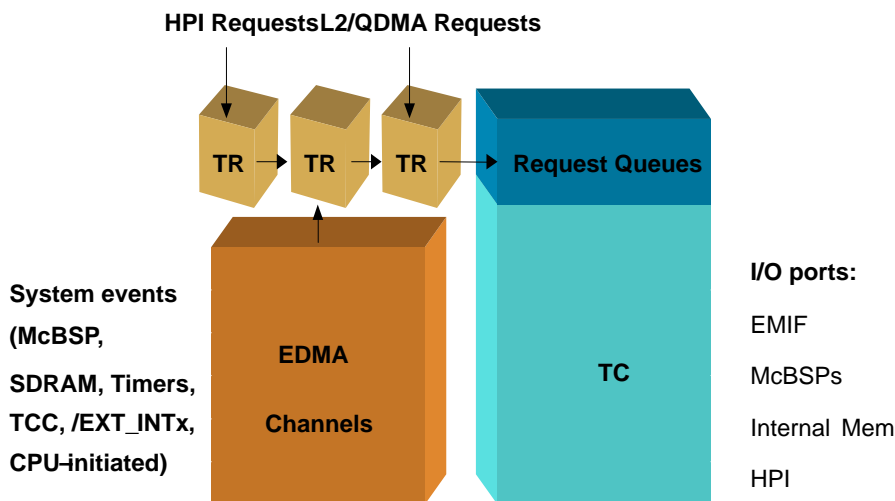


2.4.2 Powerful Enhanced DMA Controller

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 subframes 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.

The EDMA supports unsurpassed concurrency. Four independent transfer queues allow highly efficient operation. Channels on different queues can interleave transfers on a cycle by cycle basis. For example, on cycle 1 queue 0 could service a L2 cache miss to EMIFA. On cycle 2, queue 1 could move data from a serial port to EMIFB. On cycle 3, the HPI could transfer data to mapped internal memory through queue 3. On cycle 4, the EMIFA could move data to a serial port. The key system benefit in this efficiency is that the system designer can simply sum total the required bandwidth to see if the EDMA can support the system needs. Interactions between channels do not affect performance as much as in traditional DMA implementations.

Figure 2–5. EDMA



2.4.3 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.

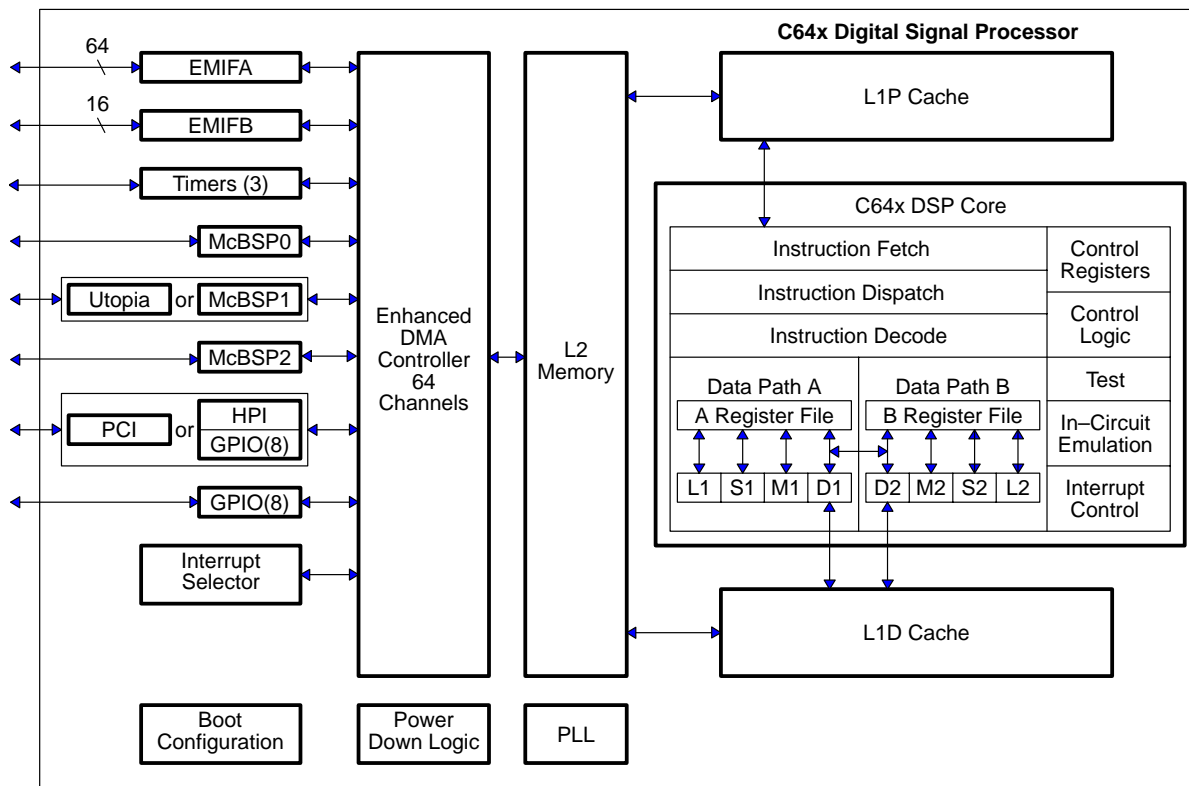
On initial implementations, these EMIFs have a maximum bus rate of 133 MHz. Each EMIF has four chip enable (CE) spaces. EMIFA can support read and write operations to 64–, 32–, 16–, and 8–bit external devices. Similarly, EMIFB can support read and write operations to both 16– and 8–bit devices. Variable 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. A programmable synchronous controller with selectable read/write latency offers direct connection to flow-through synchronous burst SRAMs, standard–write synchronous burst SRAMs, ZBT (zero bus turnaround) synchronous burst SRAMs, synchronous FIFOs, and clocked FIFOs. 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.

Figure 2–6. An Initial C64x Implementation – The C6415 DSP



2.4.4 Flexible Serial Connections

Three Multichannel Buffered Serial Ports (McBSPs) interface to a variety of standards. Each C64x McBSP supports independent channel selection at any given time for up to 128-channels. The 128 channels represent a full ST-Bus span. ST-Bus in combination with the flexible asynchronous interface provides a glueless connection to a variety of multichannel telecommunications interface products such as H.110/H.100 framers as well as T1/E1 framing chips and the IOM2 bus. Multiple audio standards such as IIS and AC97 are directly supported allowing interface to stereo multichannel audio devices. Finally, SPI mode allows connection to serial control devices and ROMs.

2.4.5 The UTOPIA Port

In some C64x devices, one of the McBSPs may be configured at reset as a UTOPIA (Universal Test and Operations Interface for ATM) port. The C64x UTOPIA peripheral is an ATM controller (ATMC) slave device that interfaces

to a master ATM controller. The UTOPIA port conforms to the ATM Forum standard specification af-phy-0039.000. Specifically, this interface supports the UTOPIA Level 2 interface that allows 8-bit slave operation up to 50MHz for both transmit and receive operations. Both the CPU and the EDMA can service the UTOPIA peripheral.

2.4.6 General Purpose Input/Output

The general-purpose input/output (GPIO) peripheral provides dedicated general-purpose pins that can be configured as either inputs or outputs. When configured as an output, the user can control the state driven on the output pin. When configured as an input, the user can detect the state of the input which is reflected in an internal register. While there are a total of 16 GPIO pins, some are multiplexed with other device pins. In addition, the GPIO peripheral can produce CPU interrupts and EDMA events in different interrupt/event generation modes.

2.4.7 Additional Peripheral Information

The C64x peripherals play an integral role in sustaining the system I/O bandwidth. For more detailed information on these new device interfaces and for specific peripheral information, please refer to the device data sheets and to the C6000 Peripherals Guide. These documents can be found at the following URL: <http://www.ti.com>.

2.5 Ease of Development

The C6000 remains a very friendly high-level language compiler target. The CPU architecture and the compiler development continue to be closely coupled.

The C64x continues the load/store architecture found in the C6000 family. By separating arithmetic and memory operations, processor throughput is maximized. The RISC like instruction set and extensive use of pipelining allow many instructions to be scheduled and executed in parallel. Also, because there is a great deal of orthogonality to the data path, register file, and instruction set, the compiler has very few restrictions. For example, the general-purpose registers can be used for data or data address pointers. The ADD instruction can execute on six of the eight functional units giving the compiler many choices of where to execute the ADD.

As with the C62x/C67x, every instruction on the C64x can be executed conditionally. This minimizes branching in the generated code. The pipeline is completely deterministic. The compiler has full visibility into the open, non-interlocked pipeline.

The C62x/C67x VelociTI architecture contains instruction packing. Eight instructions are fetched every clock cycle. Of these instructions, any, some, or all may be executed in parallel. To allow maximum usage of parallel instructions, the VelociTI architecture does not allow execute packets to cross-fetch packet boundaries. The code generation tools handled this limitation by padding fetch packets with NOP instructions. The C64x VelociTI.2 architecture extensions eliminate this limitation by including advanced instruction packing in the instruction dispatch unit. This improvement removes all execute packet boundary restrictions, thereby eliminating all of the NOPs added to pad fetch packets and helps to reduce code size.

As previously mentioned, the C64x can also access words and double words at any byte boundary using non-aligned loads and stores. Non-aligned loads and stores combined with the new data packing and unpacking instructions, mean that the compiler does not have to format the data to take advantage of the 8-bit and 16-bit hardware extensions. Without these operations, significant effort would be needed to leverage the parallelism. This is yet another example of how significant it is to tightly couple the CPU architecture and the compiler development. The C64x provides a complete set of data flow operations to sustain the maximum performance improvement made possible by the 8-bit and 16-bit extensions added to the C6000 architecture.

Other improvements to the C64x architecture, which increase compiler performance, are being able to execute logical instructions on two additional func-

tional units, doubling the register file from 32 to 64 general-purpose registers and increasing the number of condition registers. In addition, instructions have been added that further reduce code size and increase register flexibility. These include:

- ❑ BDEC and BPOS instructions combine a branch instruction with the decrement/test positive of a destination register respectively. These instructions help reduce the number of instructions needed to decrement a loop counter and conditionally branch based upon the value of that counter. Any register can be used as the loop counter, which can free up the standard condition registers (A0–A2 and B0–B2) for other uses.
- ❑ The ADDKPC instruction helps reduce the number of instructions needed to set up the return address for a function call.
- ❑ The BNOP instruction helps reduce the number of instructions required to perform a branch when NOPs are needed to fill the delay slots of a branch.

TI's C6000 Compile Tools were co-developed with the architecture to offer best-in-class performance. Examples of C6000 Compiler performance can be found at the following URL: <http://www.ti.com>. The examples at the URL above detail the C62x performance but the same underlying data path and compiler technology are used for the C64x. Compiler performance will be further enhanced as the advantages of the C64x VelociTI.2 extensions are more fully leveraged in subsequent compiler releases. For further details on specific optimization techniques for the C64x, please refer to the C6000 Programmer's Guide. For more information on compiler optimization, please see the C6000 Compiler Optimization Tutorial at the URL <http://www.ti.com> as well.

2.6 Summary

The C64x brings the highest level of performance for addressing the demands of this era of data convergence. At clock rates of 1.1 GHz and greater, the C64x can process information at a rate of 8800+ MIPS or nearly nine billion instructions per second. The C64x VelociTI.2 extensions and higher clock rate improve performance of the C62x/C67x VelociTI architecture by a factor of 9 in broadband communications and a factor of 15 in image processing applications.

These advances in performance are made possible by some key extensions made to the VelociTI architecture in several areas:

- ☐ Register file enhancements
- ☐ Data path extensions
- ☐ Packed data processing
- ☐ Additional functional unit hardware
- ☐ Increased orthogonality

2.6.1 Register File Enhancements

- ☐ The register files have doubled in size. The C62x has 32 general-purpose registers and the C64x has 64 general-purpose registers.
- ☐ The C62x uses A1, A2, B0, B1 and B2 as condition registers. The C64x can also use A0 as a condition register, bringing the total to six.
- ☐ The C62x register file supports packed 16-bit data types in addition to 32-bit and 40-bit data types. The C64x register file extends this by supporting packed 8-bit types and 64-bit types.

2.6.2 Data Path Extensions

- ☐ Each .D unit can load and store double words (64 bits) with a single instruction. The .D unit on the C62x cannot load and store 64-bit values with a single instruction.
- ☐ The .D unit can now access operands via a data cross path similar to the .L, .M and .S functional units. In the C62x, only address crosspaths on the .D unit are supported.
- ☐ The C64x pipelines data cross path accesses. This allows the same register to be used as a data cross path operand by multiple functional

units in the same execute packet. In the C62x, only one cross operand is allowed per side.

2.6.3 Packed Data Processing

- ❑ Instructions have been added that operate directly on packed data to streamline data flow and increase instruction set efficiency. The C64x has a comprehensive collection of quad 8-bit and dual 16-bit instruction set extensions.
- ❑ Extensive collection of pack and unpack instructions simplifies manipulation of packed data types

2.6.4 Additional Functional Unit Hardware

- ❑ Each .M unit can now perform two 16x16 bit multiplies or four 8x8 bit multiplies every clock cycle.
- ❑ The .D units can now access words and double words on any byte boundary by using non-aligned load and store instructions. The C62x only provides aligned load and store instructions.
- ❑ The .L units can perform byte shifts and the .M units can perform bi-directional variable shifts in addition to the .S unit's ability to do shifts. The bi-directional shifts directly assist voice-compression codecs (vocoders).
- ❑ The .L units can perform quad 8-bit subtracts with absolute value. This absolute difference instruction greatly aids motion estimation algorithms.
- ❑ Special communications-specific instructions, such as SHFL, DEAL and GMPY4 have been added to the .M unit to address common operations in error-correcting codes.
- ❑ Bit-count and Rotate hardware on the .M unit extends support for bit-level algorithms such as binary morphology, image metric calculations and encryption algorithms.

2.6.5 Increased Orthogonality

- ❑ The .D unit can now perform 32-bit logical instructions in addition to the .S and .L units.
- ❑ 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.
- ❑ On the C62x one long source and one long result per data path could occur every cycle. On the C64x, up to two long sources and two long results can be accessed on each data path every cycle.

The C64x goes beyond building extensions in the hardware to bring the maximum level of performance for processing digital data quickly in this era of data convergence. 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 and parallelism is the key to extremely high performance. In addition, a high performance two-level cache design allows the CPU to operate at the maximum rate. This two-level cache lowers development time by automating off-chip to on-chip data transfers. A high performance EDMA controller feeds the CPU through flexible high bandwidth three-bus architecture.

Aided by advanced instruction packing, doubling the number of registers in the register file and doubling the width of the data path, the C6000 compiler can improve code performance with few restrictions placed upon it by the architecture. The blending of CPU, system architecture, and compiler tools simultaneously maintain full system performance with reduced development time. These factors and others make the C64x an even better compiler target than the original C62x architecture, allowing developers to keep up with the demands of the era of data convergence.

Sum of Products Example

A.1 Sum of Products Example

One of the fundamental building blocks of any DSP algorithm be it convolution, filtering or FFTs is the sum of products equation.

$$Y = \sum_{n=1}^N a_n * x_n$$

The two basic instructions in this sum of products equation are multiply and add. We want to multiply an element in the **a** array with the corresponding element in the **x** array. We then will keep a running sum of products as we process the next elements in the arrays.

Here is a C implementation of this algorithm where the number of elements in the arrays is 40.

```
/* Main Code */
main()
{
    y = DotP(a, x, 40);
}
int DotP(short *m, short *n, int count)
{
    int i;
    int product;
    int sum = 0;
    for (i=0; i < count; i++)
    {
        product = m[i] * n[i];
        sum += product;
    }
    return(sum);
}
```

Here is the output of the compiler for the loop kernel for the example above. This is using a pre-release version of the 4.0 C6000 compiler. The customer release version became available in early 2Q00. The compiler options used were:

`-k -mv6400 -o2 -mt -mi -mx -mw`

For more information on compiler optimization, please see the C6000 Compiler Optimization Tutorial at the URL <http://www.ti.com/>.

PIPED LOOP KERNEL

LOOP:

[A0]	SUB	.L1	A0,1,A0	
[!A0]	ADD	.S1	A6,A5,A5	; keep running sum
	MPY	.M1X	B4,A4,A6	; multiply two 16-bit values
[B0]	BDEC	.S2	LOOP,B0	; decrement loop counter and branch if > 0
	LDH	.D1T1	*A3++,A4	; load 16-bit value
	LDH	.D2T2	*B5++,B4	; load 16-bit value

Notice that we are multiplying short (16-bit) values. We know that the C64x has the capability of performing four 16 x16 multiplies in a single cycle. Moreover, one of the special instructions on the C64x

is a **DOTP2** instruction. The **DOTP2** instruction returns the dot product between two pairs of signed packed 16-bit values residing in two 32-bit registers.

How can we take advantage of this instruction from the C language? **DOTP2** is available to the compiler as an intrinsic. An intrinsic function is similar to the mathematics functions available in the Run-Time Support Library. An intrinsic allows your C code to directly access the hardware while preserving the C environment. Intrinsic functions have a leading underscore with the function in lower case letters. The intrinsic for **DOTP2** is `_dotp2`.

Next we need to access the data as 32-bit values. The **DOTP2** instruction is doing two 16 x 16 multiplies which means we need two 32-bit values to be accessed every cycle. We can use the new `_amem4_const` memory intrinsic available in compiler release 4.1 to have the compiler access 32-bit values. Next because we are doing two 16 x 16 multiplies per clock cycle, we only need to perform this loop 20 times instead of 40.

Our C code now looks like the following:

```
/* Main Code */
main()
{
    y = DotP(_amem4_const(&a), _amem4_const(&x), 20);
}
int DotP(int *m, int *n, int count)
{
    int i;
    int product;
    int sum = 0;

    for (i=0; i < count; i++)
    {
        product = _dotp2(m[i], n[i]);
        sum = product + sum;
    }
    return(sum);
}
```

Here is the output of the compiler for the loop kernel for our intrinsic example above. The compiler options used were:

```
-k -mv6400 -o2 -mt -mi -mx -mw
```

```
; PIPED LOOP KERNEL
LOOP:
    [!A1]    ADD     .L2      B8,B4,B4      ; running sum 0
||          DOTP2    .M2X    B7,A6,B8      ; 2 16x16 multiplies + add ; prod 0
|| [ A0]    BDEC     .S1      LOOP,A0      ; decrement loop counter and branch if >0
||          LDW      .D1T1    *+A4(4),A3    ; load a 32-bit value
||          LDW      .D2T2    *+B5(4),B6    ; load a 32-bit value
|| [ A1]    SUB      .L1      A1,1,A1      ;
|| [!A1]    ADD     .S1      A7,A5,A5      ; running sum 1
||          DOTP2    .M1X    B6,A3,A7      ; 2 16x16 multiplies + add ; prod 1
||          LDW      .D1T1    *++A4(8),A6   ; load a 32-bit value
||          LDW      .D2T2    *++B5(8),B7   ; load a 32-bit value
```

The compiler has created a 2 cycle loop with four 16 x 16 multiplies occurring and two results produced every loop iteration. The compiler is bringing in data as 32-bit values with the **LDDW** instructions and is using the **DOTP2** instruction on both multiply functional units.

Can this code be improved further? We know that the C64x can bring in data as 64-bit values. We need eight 16-bit values every clock cycle to be able to do four 16 x 16 multiplies every clock cycle. This can be accomplished by using two **LDDW** instructions and two **DOTP2** instructions. This time we will use the new `_amemd8_const` memory intrinsic available in compiler release 4.1 to have the compiler access 64-bit values. As we mentioned earlier, the **DOTP2** instruction is doing two 16 x 16 multiplies that use two 32-bit values. Since we are bringing in the data as 64-bits we need to specify which 32-bit values the **DOTP2** instructions are operating on. We can do this by using the `_lo` and `_hi` intrinsics. The `_lo` intrinsic specifies the lower 32-bits of a 64-bit value and the `_hi` intrinsic specifies the upper 32 bits of a 64-bit value. Finally, since we are doing four 16 x 16 multiplies per clock cycle, we only need to perform this loop 10 times instead of 20 times in our previous example.

Our C code for the DotP now looks like the following:

```
int DotP(const short * restrict m, const short * restrict n, int count)
{
    int i;
    int sum= 0;

    m3_m2 = _hi(_amemd8_const(&m[i]));
    m1_m0 = _lo(_amemd8_const(&m[i]));

    n3_n2 = _hi(_amemd8_const(&n[i]));
    n1_n0 = _lo(_amemd8_const(&n[i]));

    count >> 2; /* count is divided by two if using same main
                function to call this subroutine*/
    for (i=0; i < count; i++)
    {
        sum += _dotp2(m3_m2, n3_n2) +
              _dotp2(m1_m0, n1_n0);
    }
    return sum;
}
```

Here is the output of the compiler for the loop kernel for our second intrinsic example above.

The compiler options used were:

```
-k -mv6400 -o2 -mt -mi -mx -mw
```

```
loop:      ; PIPED LOOP KERNEL
[ B0]      SUB      .L2      B0,1,B0      ; decrement running sum counter
|| [!B0]    ADD      .S2      B8,B6,B6      ; running sum 0
|| [!B0]    ADD      .L1      A7,A6,A6      ; running sum 1
||          DOTP2     .M2X     B4,A4,B8      ; 2 16x16 multiplies + add ;prod 0
||          DOTP2     .M1X     B5,A5,A7      ; 2 16x16 multiplies + add ;prod 1
|| [ A0]    BDEC      .S1      loop,A0      ; branch to loop & decrement loop count
||          LDDW      .D1T1    *A3++,A5:A4    ; load a 64-bit value
||          LDDW      .D2T2    *B7++,B5:B4    ; load a 64-bit value
```

The compiler has created a single cycle loop with four 16 x 16 multiplies occurring and two results produced every loop iteration. This represents a four-fold improvement from our original implementation.

The previous two code examples used intrinsics to improve the performance of the C code. The use of intrinsics is not always necessary to achieve single cycle loop performance for the sum of products example. If the compiler is provided with enough information about the loop count and about the alignment and scope of the pointer variables, single cycle throughput can be achieved for this algorithm without the use of intrinsics. For more information on compiler optimization, please see the C6000 Compiler Optimization Tutorial at the URL <http://www.ti.com/>.

Image Processing Kernel Code Examples

Appendix B contains code examples that come from the application benchmarks section and commented on in the architectural overview. These examples are meant to highlight some of the key extensions to the VelociTI architecture. For this reason they are coded in linear assembly to illustrate the functionality of the particular instructions. Linear assembly allows us to write assembly code with C variable names and without having to specify register allocation.

B.1 Threshold Example

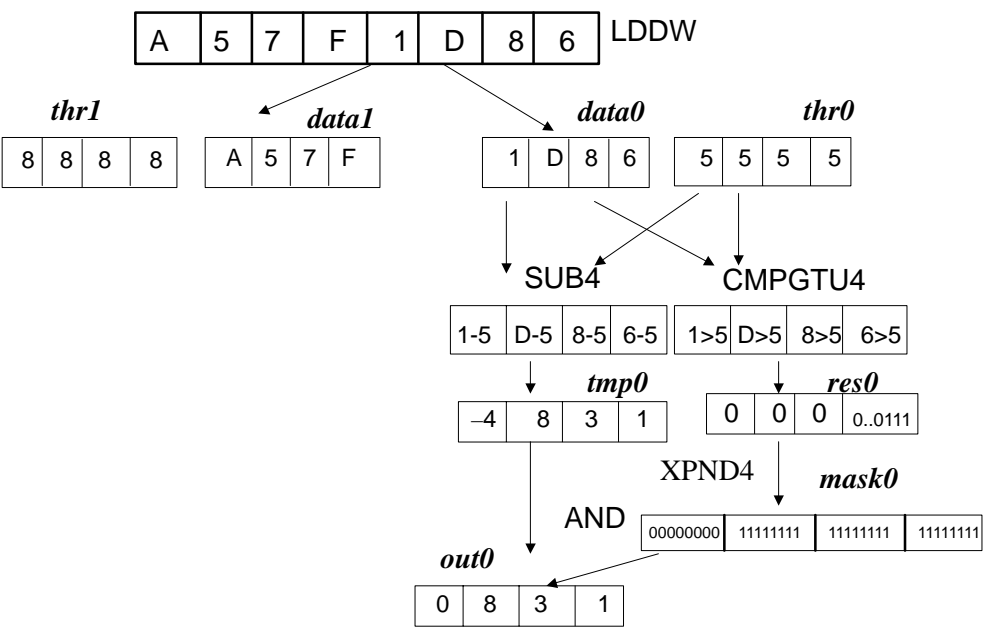
The code fragment below illustrates a thresholding example. An input data value is compared to a reference value. If the input value is less than the threshold, the corresponding output is set to zero. Otherwise, the output value is equal to the input data minus the threshold. This is a form of clamping. Other threshold algorithms can be implemented in a similar manner.

Two load double word instructions, **LDDW**, are used to load in the sixteen 8-bit pixel values. The threshold value has been loaded into each of the bytes contained in registers *thr0* and *thr1*. The **CMPTU4** instruction is used to compare four input pixel values with the threshold value at the same time. Each of the four comparisons will generate a 1-bit result. A 1 if the input value is greater than the threshold and a 0 if the input is less than the threshold. These four results are stored in the 4 LSBs of the register *res0*.

The **XPND4** instruction is used to expand the results in *res0* where each bit will be replicated to fill an entire byte's worth of data creating the mask, *mask0*. The threshold is subtracted from the input data using a **SUB4** instruction, creating *tmp0*. The *mask0* is then ANDed with *tmp0* to produce the output, *out0*. This is then repeated for each set of four pixels.

Figure B-1 is a graphical interpretation of the data flow in the algorithm described above. These operations would be repeated for each set of four pixels.

Figure B-1. Threshold Example



```

LDDW *input_data_ptr++(8), data0:data1      ; Load eight 8-bit input data values
                                              ; and post increment pointer by 8
                                              ; bytes
CMPGTU4 data0, thr0, res0                    ; Compare four input pixels with
                                              ; threshold value
XPND4   res0, mask0                          ; Expand bit to byte in a mask
SUB4    data0, thr0, tmp0                    ; Subtract threshold from input data
AND     mask0, tmp0, out0                    ; AND mask with the subtracted value

CMPGTU4 data1, thr0, res1                    ; Compare second set of four input
                                              ; pixels with threshold value
XPND4   res1, mask1                          ; Expand bit to byte in a mask
SUB4    data1, thr1, tmp1                    ; Subtract threshold from input data
AND     mask1, tmp1, out1                    ; AND mask with the subtracted value

STDW    out0:out1, *output_data_ptr++        ; Store eight 8-bit values to memory
LDDW    *input_data_ptr++(8), data2:data3    ; Load next eight 8-bit input data
                                              ; values and post increment pointer
                                              ; by 8 bytes
CMPGTU4 data2, thr1, res2                    ; Compare third set of four input
                                              ; pixels to threshold value
XPND4   res2, mask2                          ; Expand bit to byte in a mask
SUB4    data2, thr2, tmp2                    ; Subtract threshold from input data
AND     mask2, tmp2, out2                    ; AND mask with the subtracted value

CMPGTU4 data3, thr1, res3                    ; Compare last set of four input
                                              ; pixels with threshold value
XPND4   res3, mask3                          ; Expand bit to byte in a mask
SUB4    data3, thr3, tmp3                    ; Subtract threshold from input data
AND     mask3, tmp3, out3                    ; AND mask with the subtracted value

STDW    out2:out3, *output_data_ptr++        ; Store eight 8-bit values to memory

```

B.2 Motion Estimation Example

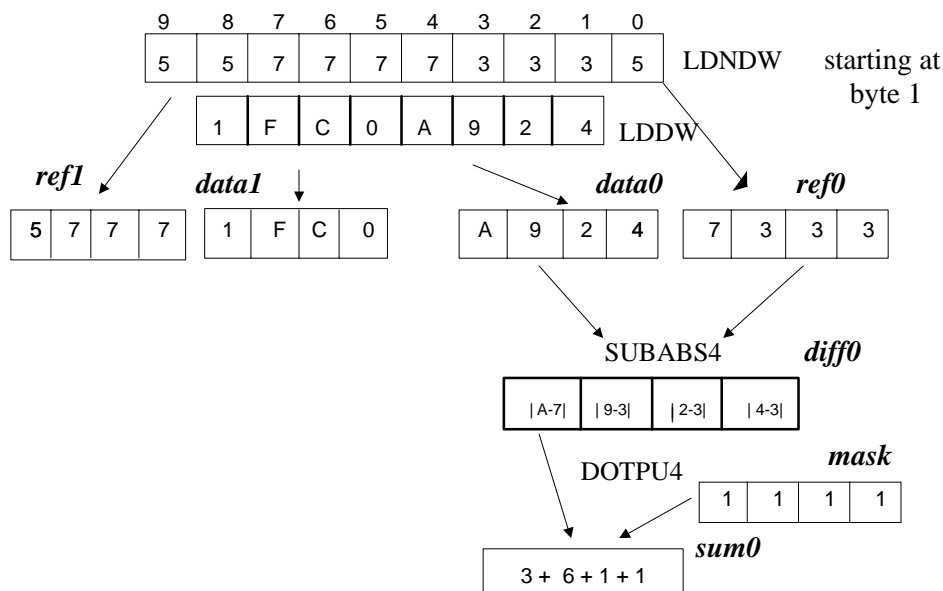
This example illustrates one row of processing for an 8x8 minimum absolute difference (MAD) computation. Multiple instances of this code block are used in a loop to achieve computation of the overall MAD value.

One non-aligned load double word instruction, **LDNDW**, is used to load in the eight 8-bit pixel reference values. A non-aligned load is used because in the MAD computation the reference data block can start on any pixel (byte) boundary. Typically, you are stepping the motion search across a general region of reference data so the required data alignment of the reference data can change from one loop iteration to the next. The input data is loaded using a load double word instruction, **LDDW**, because it is assumed to be aligned on double word boundaries. Next the **SUBABS4** instruction is used to take the absolute difference between the input data and the reference values. Each **SUBABS4** instruction processes four 8-bit pixels.

We now need to sum together the absolute difference values. The four values in one 32-bit register may be summed together by using a **DOTPU4** instruction as shown. The masks in the **DOTPU4** operations are pre-loaded with each byte value containing the value +1. Therefore, multiplying each byte value with one and adding them together allows us to sum the four values in the 32-bit register.

We then use **ADD** instructions to sum the results together and the final result is added to a quantity *my_mad* that is the running MAD value that is carried over from row to row and provides the final numerical output. Figure B-2 is a graphical interpretation of the data flow in the algorithm described above. These operations would be repeated for each set of four pixels.

Figure B-2. Motion Estimation Example



Example B–1. One row of 8x8 MAD calculation

; row 1

```

LDNDW  *ref_ptr++(8), ref0:ref1      ; Load eight 8-bit reference values
                                           ; and post increment pointer by 8
                                           ; bytes
ADD     ref_ptr, offset, ref_ptr      ; Add offset to move reference
                                           ; data pointer to next row

LDDW    *input_data_ptr++, data0:data1 ; Load eight 8-bit input data values
                                           ; and post increment pointer

SUBABS4 ref0, data0, diff0             ; get absolute difference value
                                           ; for first four bytes
SUBABS4 ref1, data1, diff1             ; get absolute difference value
                                           ; for next four bytes

DOTPU4  diff0, mask0, sum0             ; add absolute differences of the
                                           ; first four bytes
DOTPU4  diff1, mask0, sum1             ; add absolute differences of the
                                           ; next four bytes
ADD     sum0, sum1, mad_r1             ; add result of absolute differences
                                           ; of the eight bytes

ADD     my_mad, mad_r1, my_mad         ; add this result to running sum
                                           ; my_mad

```

Glossary

A

address: The location of program code or data stored; an individually accessible memory location.

ALU: See *arithmetic logic unit*.

arithmetic logic unit (ALU): The hardware of the CPU that performs arithmetic and logic functions.

C

central processing unit (CPU): The unit that coordinates the functions of a processor.

circular addressing: An address mode in which a finite set of addresses is reused by linking the largest address back to the smallest address.

clock cycles: A periodic or sequence of events based on the input from the external clock.

code: A set of instructions written to perform a task; a computer program or part of a program.

compiler: A computer program that translates programs in a high-level language into their assembly-language equivalents.

CPU: See *central processing unit*.

crosspath: A link between register files to provide communication between the CPU units.

D

double word: A set of 64 bits that is stored, addressed, transmitted, or operated on as a unit.

E

execute packet: A group of instructions that execute in parallel.

F

fixed-point processor: A processor which does arithmetic operations using integer arithmetic with no exponents.

floating-point processor: A processor capable of handling floating-point arithmetic where real operands are represented using exponents.

M

million instructions per second (MIPS): A measure of the execution speed of a computer.

P

parallelism: Sequencing events to occur simultaneously. Parallelism is achieved in a CPU by using instruction pipelining.

pipeline: A method of executing instructions in which the output of one process serves as the input to another, much like an assembly line. These processes become the stages or phases of the pipeline.

pipeline processing: A technique that provides simultaneous, or parallel, processing within the computer. It refers to overlapping operations by moving data or instructions into a conceptual pipe with all stages of the pipe processing simultaneously.

program fetch unit: The CPU hardware that retrieves program instructions.

R

register: A small area of high speed memory, located within a processor or electronic device, that is used for temporarily storing data or instructions. Each register is given a name, contains a few bytes of information, and is referenced by programs.

reduced-instruction set computer (RISC): A computer whose instruction set and related decode mechanism are much simpler than those of microprogrammed complex instruction set computers. The result is a higher instruction throughput and a faster real-time interrupt service response from a smaller, cost-effective chip.

S

saturation: A state where any further input no longer results in the expected output.

shifter: A hardware unit that shifts bits in a word to the left or to the right.

W

word: A set of 32 bits that is stored, addressed, transmitted, or operated on as a unit.

Related Documents

The following books describe the TMS320C6000 generation and related support tools. To obtain a copy of any of these TI documents, call the Texas Instruments Literature Response Center at (800) 477–8924. When ordering, please identify the book by its title and literature number.

TMS320C6000 CPU and Instruction Set Reference Guide (literature number SPRU189) describes the CPU architecture, pipeline, instruction set, and interrupts for the TMS320C6000 digital signal processors (DSPs).

TMS320C6201 Digital Signal Processor Data Sheet (literature number SPRS051) describes the features of the TMS320C6201 and provides pinouts, electrical specifications, and timings for the device.

TMS320C6202/C6202B/C6203/C6204 Digital Signal Processor Data Sheet (literature number SPRS104) describes the features of the TMS320C6202/03/04 fixed-point DSP and provides pinouts, electrical specifications, and timings for the device.

TMS320C6205 Digital Signal Processor Data Sheet (literature number SPRS106) describes the features of the TMS320C6205 fixed-point DSP and provides pinouts, electrical specifications, and timings for the device.

TMS320C6211/C6711 Digital Signal Processor Data Sheet (literature number SPRS073) describes the features of the TMS320C6211 fixed-point DSP and the TMS320C6711 floating-point DSP and provides pinouts, electrical specifications, and timings for the device.

TMS320C6712 Digital Signal Processor Data Sheet (literature number SPRS148) describes the features of the TMS320C6712 floating-point DSP and provides pinouts, electrical specifications, and timings for the device.

TMS320C6701 Digital Signal Processor Data Sheet (literature number SPRS067) describes the features of the TMS320C6701 floating-point DSP and provides pinouts, electrical specifications, and timings for the device.

TMS320C6000 Peripherals Reference Guide (literature number SPRU190) describes common peripherals available on the TMS320C6000 digital signal processors. This book includes information on the internal data and program memories, the external memory interface (EMIF), the host port, serial ports, direct memory access (DMA), clocking and phase-locked loop (PLL), and the power-down modes.

TMS320C6000 Programmer's Guide (literature number SPRU198) describes ways to optimize C and assembly code for the TMS320C6000 DSPs and includes application program examples.

TMS320C6000 Assembly Language Tools User's Guide (literature number SPRU186) describes the assembly language tools (assembler, linker, and other tools used to develop assembly language code), assembler directives, macros, common object file format, and symbolic debugging directives for the C6000 generation of devices.

TMS320C6000 Optimizing C Compiler User's Guide (literature number SPRU187) describes the C6000 C compiler and the assembly optimizer. This C compiler accepts ANSI standard C source code and produces assembly language source code for the C6000 generation of devices. The assembly optimizer helps you optimize your assembly code.

TMS320 Third-Party Support Reference Guide (literature number SPRU052) alphabetically lists over 100 third parties that provide various products that serve the family of TMS320 digital signal processors. A myriad of products and applications are offered—software and hardware development tools, speech recognition, image processing, noise cancellation, modems, etc.

Trademarks

TI, XDS510, VelociTI, and 320 Hotline On-line are trademarks of Texas Instruments Incorporated.

Windows and Windows NT are registered trademarks of Microsoft Corporation.