TMS320C62xx CPU and Instruction Set Reference Guide

Literature Number: SPRU189B
Manufacturing Part Number: D426008-9761 revision A
July 1997

TEXAS INSTRUMENTS

IMPORTANT NOTICE

Texas Instruments (TI) reserves the right to make changes to its products or to discontinue any semiconductor product or service without notice, and advises its customers to obtain the latest version of relevant information to verify, before placing orders, that the information being relied on is current.

TI warrants performance of its semiconductor products and related software 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.

Certain applications using semiconductor products may involve potential risks of death, personal injury, or severe property or environmental damage ("Critical Applications").

TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, INTENDED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT APPLICATIONS, DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS.

Inclusion of TI products in such applications is understood to be fully at the risk of the customer. Use of TI products in such applications requires the written approval of an appropriate TI officer. Questions concerning potential risk applications should be directed to TI through a local SC sales office.

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

TI assumes no liability for applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does TI 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.

Preface

Read This First

About This Manual

This reference guide describes the CPU architecture, pipeline, instruction set, and interrupts for the TMS320C62xx digital signal processors (DSPs). Unless otherwise specified, all references to the ’C6x refer to the TMS320C6x generation of DSPs and ’C62xx refer to the TMS320C62xx DSPs in the ’C6x generation.

How to Use This Manual

Use this manual as a reference for the architecture of the TMS320C62xx CPU. First-time readers should read Chapter 1 for general information about TI DSPs, the features of the TMS320C62xx, and the applications for which the TMS320C62xx is best suited.

Read chapters 2, 4, and 5 to grasp the concepts of the architecture. Chapter 3 contains detailed information about each instruction and is best used as reference material; however, you may want to read sections 3.1 through 3.8 for general information about the instruction set and to understand the instruction descriptions, then browse through Chapter 3 to familiarize yourself with the instructions.

The following table gives chapter references for specific information:

If you are looking for information about:	Turn to these chapters:
Addressing modes	Chapter 3, *Instruction Set*
Conditional operations	Chapter 3, *Instruction Set*
Control registers	Chapter 2, *CPU Overview*
CPU architecture	Chapter 2, *CPU Overview*
CPU data paths	Chapter 2, *CPU Overview*
Delay slots	Chapter 3, *Instruction Set* Chapter 4, *Pipeline Operation*

If you are looking for information about:	Turn to these chapters:
General-purpose register files	Chapter 2, *CPU Overview*
Instruction set	Chapter 3, *Instruction Set*
Interrupt control registers	Chapter 5, *Interrupts*
Interrupts	Chapter 5, *Interrupts*
Parallel operations	Chapter 3, *Instruction Set*
Pipeline operation	Chapter 4, *Pipeline Operation*
Pipeline phases	Chapter 4, *Pipeline Operation*
Reset	Chapter 5, *Interrupts*

If you are interested in topics that are not listed here, check *Related Documentation From Texas Instruments*, page v, for brief descriptions of other 'C62xx-related books that are available.

Notational Conventions

This document uses the following conventions:

- Program listings and program examples are shown in a `special font`. Here is a sample program listing:

```
LDW .D1    *A0,A1
ADD .L1    A1,A2,A3
NOP        3
MPY .M1    A1,A4,A5
```

- To help you easily recognize instructions and parameters throughout the book, instructions are in **bold face** and parameters are in *italics* (except in program listings).

- In instruction syntaxes, portions of a syntax that are in **bold** should be entered as shown; portions of a syntax that are in *italics* describe the *type* of information that should be entered. Here is an example of an instruction:

  MPY *src1,src2,dst*

  MPY is the instruction mnemonic. When you use **MPY**, you must supply two source operands (*src1* and *src2*) and a destination operand (*dst*) of appropriate types as defined in Chapter 3, *Instruction Set*.

  Although the instruction mnemonic (**MPY** in this example) is in capital letters, the 'C6x assembler *is not case sensitive* — it can assemble mnemonics entered in either upper or lower case.

- Square brackets, [and], and parentheses, (and), are used to identify optional items. If you use an optional item, you must specify the information within brackets or parentheses; however, you do not enter the brackets or parentheses themselves. Here is an example of an instruction that has optional items.

  [*label*] **EXTU** (*.unit*) *src2, csta, cstb, dst*

  The **EXTU** instruction is shown with a label and several parameters. The [*label*] and the parameter (*.unit*) are optional. The parameters *src2, csta, cstb,* and *dst* are not optional.

- Throughout this book MSB means *most significant bit* and LSB means *least significant bit*.

Related Documentation From Texas Instruments

The following books describe the TMS320C6x 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.

TMS320C62xx Technical Brief (literature number SPRU197) gives an introduction to the ’C62xx digital signal processor, development tools, and third-party support.

TMS320C62xx Peripherals Reference Guide (literature number SPRU190) describes common peripherals available on the TMS320C62xx 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.

TMS320C62xx Programmer’s Guide (literature number SPRU198) describes ways to optimize C and assembly code and includes application program examples.

TMS320C6x 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 ’C6x generation of devices.

TMS320C6x Optimizing C Compiler User’s Guide (literature number SPRU187) describes the ’C6x C compiler. This C compiler accepts ANSI standard C source code and produces assembly language source code for the ’C6x generation of devices. This book also describes the assembly optimizer, which helps you optimize your assembly code.

TMS320C6x C Source Debugger User's Guide (literature number SPRU188) tells you how to invoke the 'C6x simulator and emulator versions of the C source debugger interface. This book discusses various aspects of the debugger, including command entry, code execution, data management, breakpoints, profiling, and analysis.

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.

TMS320C6201 Digital Signal Processor Data Sheet (literature number SPRS051) describes the features of the TMS320C6xx and provides pinouts, electrical specifications, and timings for the device.

Trademarks

If You Need Assistance . . .

World-Wide Web Sites

TI Online	http://www.ti.com
Semiconductor Product Information Center (PIC)	http://www.ti.com/sc/docs/pic/home.htm
DSP Solutions	http://www.ti.com/dsps
320 Hotline On-line™	http://www.ti.com/sc/docs/dsps/support.htm

North America, South America, Central America

Product Information Center (PIC)	(972) 644-5580		
TI Literature Response Center U.S.A.	(800) 477-8924		
Software Registration/Upgrades	(214) 638-0333	Fax: (214) 638-7742	
U.S.A. Factory Repair/Hardware Upgrades	(281) 274-2285		
U.S. Technical Training Organization	(972) 644-5580		
DSP Hotline	(281) 274-2320	Fax: (281) 274-2324	Email: dsph@ti.com
DSP Modem BBS	(281) 274-2323		

DSP Internet BBS via anonymous ftp to ftp://ftp.ti.com/pub/tms320bbs

Europe, Middle East, Africa

European Product Information Center (EPIC) Hotlines:

Multi-Language Support	+33 1 30 70 11 69	Fax: +33 1 30 70 10 32	Email: epic@ti.com
Deutsch	+49 8161 80 33 11 or +33 1 30 70 11 68		
English	+33 1 30 70 11 65		
Francais	+33 1 30 70 11 64		
Italiano	+33 1 30 70 11 67		
EPIC Modem BBS	+33 1 30 70 11 99		
European Factory Repair	+33 4 93 22 25 40		
Europe Customer Training Helpline		Fax: +49 81 61 80 40 10	

Asia-Pacific

Literature Response Center	+852 2 956 7288	Fax: +852 2 956 2200
Hong Kong DSP Hotline	+852 2 956 7268	Fax: +852 2 956 1002
Korea DSP Hotline	+82 2 551 2804	Fax: +82 2 551 2828
Korea DSP Modem BBS	+82 2 551 2914	
Singapore DSP Hotline		Fax: +65 390 7179
Taiwan DSP Hotline	+886 2 377 1450	Fax: +886 2 377 2718
Taiwan DSP Modem BBS	+886 2 376 2592	

Taiwan DSP Internet BBS via anonymous ftp to ftp://dsp.ee.tit.edu.tw/pub/TI/

Japan

Product Information Center	+0120-81-0026 (in Japan)	Fax: +0120-81-0036 (in Japan)
	+03-3457-0972 or (INTL) 813-3457-0972	Fax: +03-3457-1259 or (INTL) 813-3457-1259
DSP Hotline	+03-3769-8735 or (INTL) 813-3769-8735	Fax: +03-3457-7071 or (INTL) 813-3457-7071
DSP BBS via Nifty-Serve	Type “Go TIASP”	

Documentation

When making suggestions or reporting errors in documentation, please include the following information that is on the title page: the full title of the book, the publication date, and the literature number.

Mail: Texas Instruments Incorporated
Technical Documentation Services, MS 702
P.O. Box 1443
Houston, Texas 77251-1443

Email: comments@books.sc.ti.com

Note: When calling a Literature Response Center to order documentation, please specify the literature number of the book.

Contents

Figures

Tables

Examples

Chapter 1

Introduction

The TMS320C6x generation of digital signal processors is part of the TMS320 family of digital signal processors (DSPs). The TMS320C62xx devices are fixed-point DSPs in the TMS320C6x generation. The TMS320C62xx is the first DSP to use the VelociTI™ architecture, a high-performance, advanced VLIW (very long instruction word) architecture, making the 'C62xx an excellent choice for multichannel and multifunction applications.

The 'C62xx's VelociTI architecture makes it the first off-the-shelf DSP to use advanced VLIW to achieve high performance through increased instruction-level parallelism. A traditional VLIW architecture consists of multiple execution units running in parallel, performing multiple instructions during a single clock cycle. Parallelism is the key to extremely high performance, taking these DSPs well beyond the performance capabilities of traditional superscalar designs. VelociTI is a highly deterministic architecture, having few restrictions on how or when instructions are fetched, executed, or stored. It is this architectural flexibility that is key to the breakthrough efficiency levels of the 'C6x compiler. VelociTI's advanced features include:

- Instruction packing: reduced code size
- All instructions can operate conditionally: flexibility of code
- Variable-width instructions: flexibility of data types
- Fully pipelined branches: zero-overhead branching

Topic **Page**

1.1 TMS320 Family Overview

The TMS320 family consists of fixed-point, floating-point, and multiprocessor digital signal processors (DSPs). TMS320 DSPs have an architecture designed specifically for real-time signal processing.

1.1.1 History of TMS320 DSPs

In 1982, Texas Instruments introduced the TMS32010—the first fixed-point DSP in the TMS320 family. Before the end of the year, *Electronic Products* magazine awarded the TMS32010 the title “Product of the Year”. Today, the TMS320 family consists of many generations: ’C1x, ’C2x, ’C2xx, ’C5x, and ’C54x fixed-point DSPs; ’C3x and ’C4x floating-point DSPs, and ’C8x multiprocessor DSPs. Now there is a new generation of DSPs, the TMS320C6x generation, with performance and features that are reflective of Texas Instruments commitment to lead the world in DSP solutions.

1.1.2 Typical Applications for the TMS320 Family

Table 1–1 lists some typical applications for the TMS320 family of DSPs. The TMS320 DSPs offer adaptable approaches to traditional signal-processing problems. They also support complex applications that often require multiple operations to be performed simultaneously.

Table 1–1. Typical Applications for the TMS320 DSPs

Automotive	**Consumer**	**Control**
Adaptive ride control	Digital radios/TVs	Disk drive control
Antiskid brakes	Educational toys	Engine control
Cellular telephones	Music synthesizers	Laser printer control
Digital radios	Pagers	Motor control
Engine control	Power tools	Robotics control
Global positioning	Radar detectors	Servo control
Navigation	Solid-state answering machines	
Vibration analysis		
Voice commands		
General Purpose	**Graphics/Imaging**	**Industrial**
Adaptive filtering	3-D rotation	Numeric control
Convolution	Animation/digital maps	Power-line monitoring
Correlation	Homomorphic processing	Robotics
Digital filtering	Image compression/transmission	Security access
Fast Fourier transforms	Image enhancement	
Hilbert transforms	Pattern recognition	
Waveform generation	Robot vision	
Windowing	Workstations	
Instrumentation	**Medical**	**Military**
Digital filtering	Diagnostic equipment	Image processing
Function generation	Fetal monitoring	Missile guidance
Pattern matching	Hearing aids	Navigation
Phase-locked loops	Patient monitoring	Radar processing
Seismic processing	Prosthetics	Radio frequency modems
Spectrum analysis	Ultrasound equipment	Secure communications
Transient analysis		Sonar processing
Telecommunications		**Voice/Speech**
1200- to 56 600-bps modems	Faxing	Speaker verification
Adaptive equalizers	Future terminals	Speech enhancement
ADPCM transcoders	Line repeaters	Speech recognition
Base stations	Personal communications systems (PCS)	Speech synthesis
Cellular telephones	Personal digital assistants (PDA)	Speech vocoding
Channel multiplexing	Speaker phones	Text-to-speech
Data encryption	Spread spectrum communications	Voice mail
Digital PBXs	Digital subscriber loop (xDSL)	
Digital speech interpolation (DSI)	Video conferencing	
DTMF encoding/decoding	X.25 packet switching	
Echo cancellation		

1.2 Overview of the TMS320C6x Generation of Digital Signal Processors

With a performance of up to 1600 million instructions per second (MIPS) and an efficient C compiler, the TMS320C6x DSPs give system architects unlimited possibilities to differentiate their products. High performance, ease of use, and affordable pricing make the TMS320C6x generation the ideal solution for multichannel, multifunction applications, such as:

- Pooled modems
- Wireless base stations
- Remote access servers (RAS)
- Digital subscriber loop (DSL) systems
- Cable modems
- Multichannel telephony systems

The TMS320C6x generation is also an ideal solution for exciting new applications; for example:

- Personalized home security with face and hand/fingerprint recognition
- Advanced cruise control with GPS navigation and accident avoidance
- Remote medical diagnostics

1.3 Features and Options of the TMS320C62xx

At 200 MHz, the ’C62xx devices operate at a 5-ns cycle time, executing up to eight 32-bit instructions every cycle. The device’s core CPU consists of 32 general-purpose registers of 32-bit word length and eight functional units:

- Two multipliers
- Six ALUs

The ’C62xx has a complete set of optimized development tools, including an efficient C compiler, an assembly optimizer for simplified assembly-language programming and scheduling, and a Windows™ based debugger interface for visibility into source code execution characteristics. A hardware emulation board, compatible with the TI XDS510™ emulator interface, is also available. This tool complies with IEEE Standard 1149.1–1990, IEEE Standard Test Access Port and Boundary-Scan Architecture.

Features of the ’C62xx include:

- Advanced VLIW CPU with eight functional units, including two multipliers and six arithmetic units
  - Executes up to eight instructions per cycle for up to ten times the performance of typical DSPs
  - Allows designers to develop highly effective RISC-like code for fast development time
- Instruction packing
  - Gives code size equivalence for eight instructions executed serially or in parallel
  - Reduces code size, program fetches, and power consumption.
- All instructions execute conditionally.
  - Reduces costly branching
  - Increases parallelism for higher sustained performance
- Code executes as programmed on independent functional units.
  - Industry’s most efficient C compiler on DSP benchmark suite
  - Industry’s first assembly optimizer for fast development and improved parallelization
- 8/16/32-bit data support, providing efficient memory support for a variety of applications
- 40-bit arithmetic options add extra precision for vocoders and other computationally intensive applications
- Saturation and normalization provide support for key arithmetic operations.
- Field manipulation and instruction extract, set, clear, and bit counting support common operation found in control and data manipulation applications.

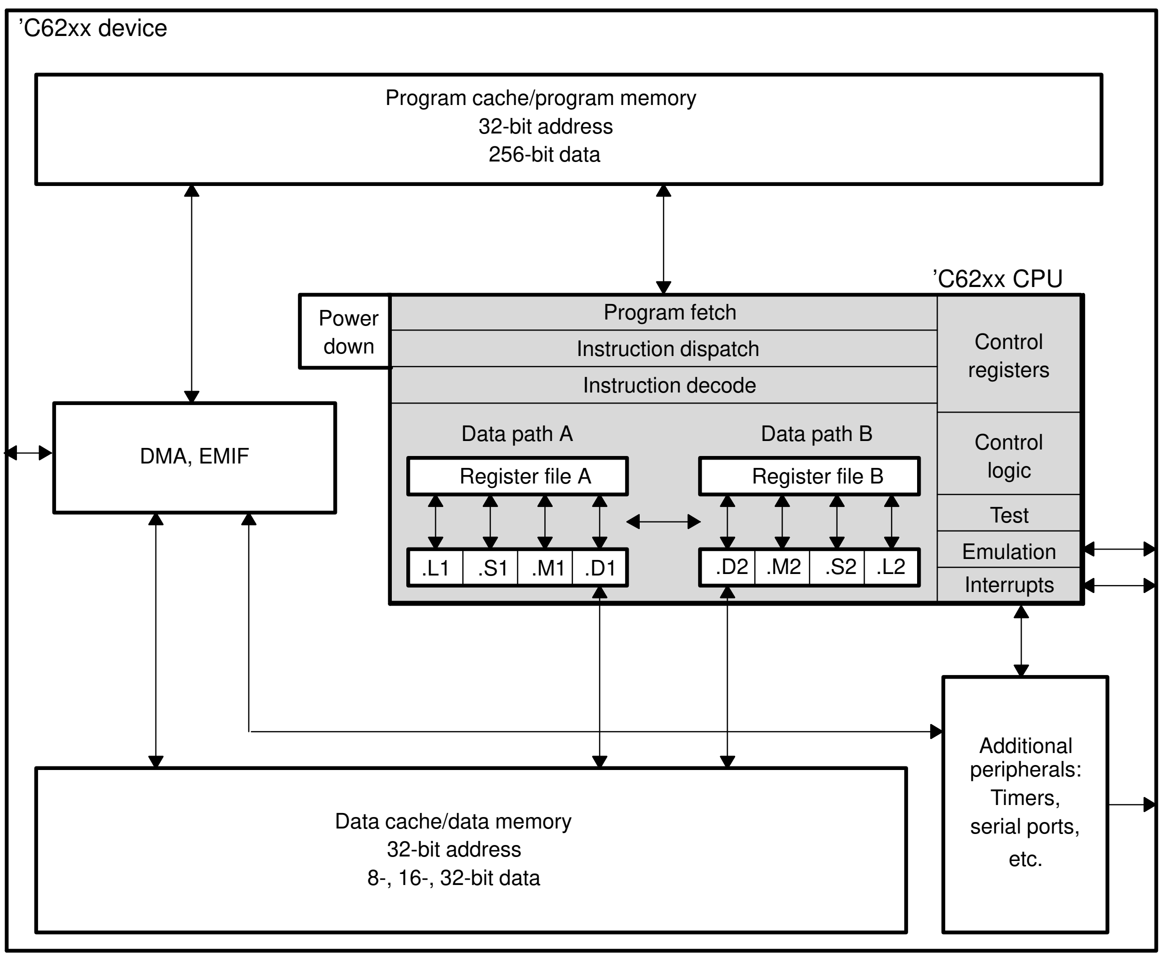
A variety of memory and peripheral options are available for the ’C62xx:

- Large on-chip RAM for fast algorithm execution
- 32-bit external memory interface supports SDRAM, SBSRAM, SRAM, and other asynchronous memories, for a broad range of external memory requirements and maximum system performance
- 16-bit host port for host to access ’C62xx memory and peripherals
- Multichannel DMA controller
- Multichannel serial port(s)
- 32-bit timer(s)

1.4 TMS320C62xx Architecture

Figure 1–1 is the block diagram for the TMS320C62xx DSPs. ’C62xx DSPs are based on the ’C62xx CPU, shown in the right center of the figure. ’C62xx devices come with program memory, which, on some devices, can be used as a program cache. The devices also have varying sizes of data memory. Peripherals such as a direct memory access (DMA) controller, power-down logic, and external memory interface (EMIF) usually come with the CPU, while peripherals such as serial ports and host ports are on only certain devices. Check the data sheet for your device to determine the specific peripheral configurations you have.

Figure 1–1. TMS320C62xx Block Diagram



1.4.1 Central Processing Unit (CPU)

The ’C62xx CPU, shaded in Figure 1–1, is common to all the ’C62xx devices. The CPU contains:

- Program fetch unit
- Instruction dispatch unit
- Instruction decode unit
- Two data paths, each with four functional units
- 32 32-bit registers
- Control registers
- Control logic
- Test, emulation, and interrupt logic

The program fetch, instruction dispatch, and instruction decode units can deliver up to eight 32-bit instructions to the functional units every CPU clock cycle. The processing of instructions occurs in each of the two data paths (A and B), each of which contains four functional units (.L, .S, .M, and .D) and 16 32-bit general-purpose registers. The data paths are described in more detail in section 2.1. A control register file provides the means to configure and control various processor operations. To understand how instructions are fetched, dispatched, decoded, and executed in the data path, see Chapter 4, *TMS320C62xx Pipeline*.

1.4.2 Internal Memory

The ’C62xx has a 32-bit, byte-addressable address space. Internal (on-chip) memory is organized in separate data and program spaces. When off-chip memory is used, these spaces are unified on most devices to a single memory space via the external memory interface (EMIF).

The ’C62xx has two internal ports to access data memory, each with 32 bits of data and a 32-bit byte address reach. The ’C62xx has a single port to access program memory, with an instruction-fetch width of 256 bits and a 30-bit word address, equivalent to a 32-bit byte address.

1.4.3 Peripherals

The following peripheral modules can complement the CPU on the ’C62xx DSPs. Your particular device has a subset of these peripherals but may not have all of them.

- Serial ports
- Timers
- External memory interface (EMIF) that supports synchronous and asynchronous SRAM and synchronous DRAM
- DMA controller
- Host port
- Power-down logic that can halt CPU activity, peripheral activity, and PLL activity to reduce power consumption

Chapter 2

CPU Data Paths and Control

This chapter provides an overview of the ’C62xx architecture. It focuses on the CPU, providing information about the data paths and control registers.

Topic **Page**

2.1 CPU Data Paths

Figure 2–2 on page 2-4 shows the ’C62xx CPU data paths, which 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 paths (LD1 and LD2)
- Two store-to-memory paths (ST1 and ST2)
- Two register file cross paths (1X and 2X)

2.1.1 General-Purpose Register Files

There are two general-purpose register files (A and B) in the ’C62xx data paths. 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.

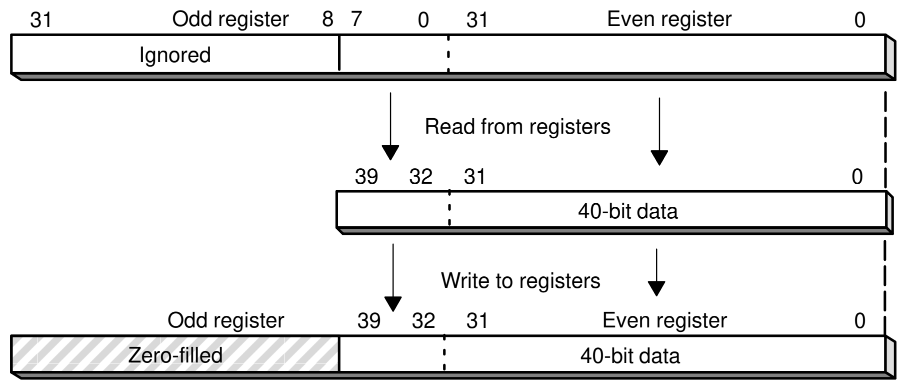
The general-purpose register file supports 32- and 40-bit data. Thirty-two-bit data can be contained in any general-purpose register. Forty-bit data is contained across two registers; the 32 LSBs of the data are placed in an even register and the remaining 8 MSBs are placed in the 8 LSBs of the next upper register (which is always an odd register). There are 16 valid register pairs for 40-bit data as shown in Table 2–1. In assembly language syntax, the register pairs are denoted by a colon between the register names, and the odd register is specified first.

Table 2–1. Long (40-Bit) Register Pairs

Register File	
A	**B**
A1:A0	B1:B0
A3:A2	B3:B2
A5:A4	B5:B4
A7:A6	B7:B6
A9:A8	B9:B8
A11:A10	B11:B10
A13:A12	B13:B12
A15:A14	B15:B14

Figure 2–1 illustrates the register storage scheme for 40-bit data. Operations requiring a long input ignore the 24 MSBs of the odd register. Operations producing a long result zero-fill the 24 MSBs of the odd register. The even register is encoded in the opcode.

Figure 2–1. Storage Scheme for 40-Bit Data in a Register Pair



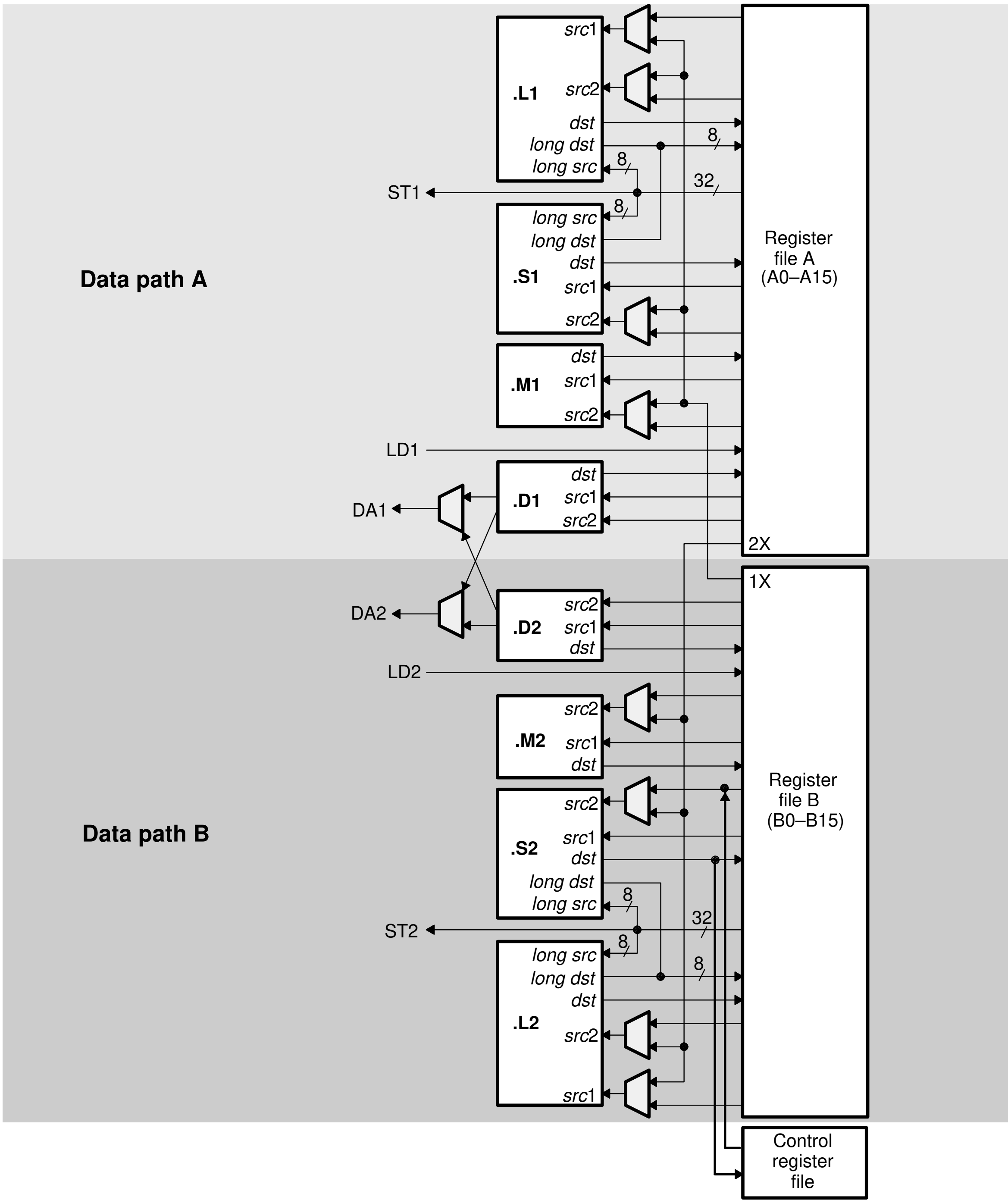
2.1.2 Functional Units

The eight functional units in the ’C62xx 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 are described in Table 2–2.

Table 2–2. Functional Units and Operations Performed

Functional Unit	Operations
.L Unit (.L1,.L2)	32/40-bit arithmetic and compare operations Leftmost 1 or 0 bit counting for 32 bits Normalization count for 32 and 40 bits 32-bit logical 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 the control register file (.S2 only)
.M Unit (.M1, .M2)	16 × 16 bit multiply operations
.D Unit (.D1, .D2)	32-bit add, subtract, linear and circular address calculation Loads and stores with a 5-bit constant offset Loads and stores with 15-bit constant offset (.D2 only)

Figure 2–2. TMS320C62xx CPU Data Paths



Most data lines in the CPU support 32-bit operands, and some support long (40-bit) operands. Each functional unit has its own 32-bit write port into a general-purpose register file. All units ending in 1 (for example, .L1) write to register file A and all units ending in 2 write to register file B. Each functional unit has two 32-bit read ports for source operands *src*1 and *src*2. Four units (.L1, .L2, .S1, .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 every cycle.

2.1.3 Register File Cross 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. The register files are 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 data path A's functional units to read their source from register file B and the 2X cross path allows data path B's functional units to read their source from register file A.

Six of the functional units have access to the opposite side's register file via a cross path. The .M1, .M2, .S1, and .S2 units' *src2* inputs are multiplex-selectable between the cross path and the same side register file. The .L1 and .L2 units' *src1* and *src2* inputs are also multiplex-selectable between the cross path and the same side register file.

There are only two cross paths in the 'C62xx CPU, 1X and 2X. This limits one source read from each data path's opposite register file per cycle, or two cross-path source reads per cycle.

2.1.4 Memory, Load, and Store Paths

There are two 32-bit paths for loading data from memory to the register file: one (LD1) for register file A, and one (LD2) for register file B. There are also two 32-bit paths, ST1 and ST2, for storing register values to memory from each register file. The store paths are shared with the .L and .S long read paths.

2.1.5 Data Address Paths

Locate the .D1 and .D2 units in Figure 2–2. The data address paths (DA1 and DA2) coming out of the .D units allow data addresses generated from one register file to support loads and stores to memory from the other register file.

2.2 Control Register File

Locate the control register file in Figure 2–2. One unit (.S2) can read from and write to the control register file. Table 2–3 lists the control registers contained in the control register file and describes each. If more information is available on a control register, the table lists where to look for that information. Each control register is accessed by the **MVC** instruction. See the **MVC** instruction description in Chapter 3, *Instruction Set*, for information on how to use this instruction.

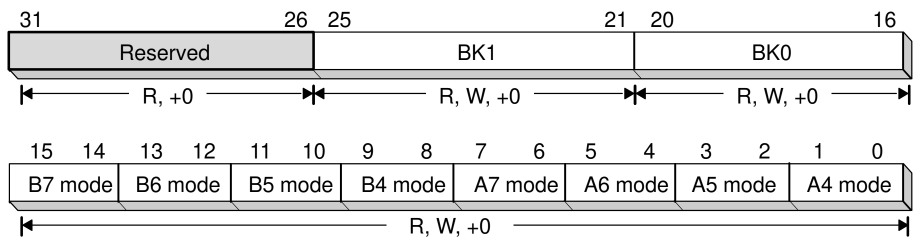
Table 2–3. Control Registers

Register			
Abbreviation	**Name**	**Description**	**Page No.**
AMR	Addressing mode register	Specifies whether to use linear or circular addressing for one of eight registers; also contains sizes for circular addressing	2-7
CSR	Control status register	Contains the global interrupt enable bit, cache control bits, and other miscellaneous control and status bits	2-9
IFR	Interrupt flag register	Displays status of interrupts	5-14
ISR	Interrupt set register	Allows you to set pending interrupts manually	5-14
ICR	Interrupt clear register	Allows you to clear pending interrupts manually	5-14
IER	Interrupt enable register	Allows enabling/disabling of individual interrupts	5-13
ISTP	Interrupt service table pointer	Points to the beginning of the interrupt service table	5-8
IRP	Interrupt return pointer	Contains the address to be used to return from a maskable interrupt	5-16
NRP	Nonmaskable interrupt return pointer	Contains the address to be used to return from a nonmaskable interrupt	5-16
PCE1	Program counter	Contains the address of the fetch packet that contains the execute packet in the E1 pipeline stage	

2.3 Addressing Mode Register (AMR)

Figure 2–3 shows the AMR. Eight registers (A4–A7, B4–B7) can perform circular addressing. For each of these registers, the AMR specifies the addressing mode. A 2-bit field for each register is used to select the address modification mode: linear (the default) or circular mode. With circular addressing, the field also specifies which BK (block size) field to use for a circular buffer. In addition, the buffer must be aligned on a byte boundary equal to the block size. The mode select field encoding is shown in Table 2–4.

Figure 2–3. Addressing Mode Register (AMR)



Legend: R Readable by the **MVC** instruction
W Writeable by the **MVC** instruction
+0 Value is zero after reset

Table 2–4. Addressing Mode Field Encoding

Mode	Description
0 0	Linear modification (default at reset)
0 1	Circular addressing using the BK0 field
1 0	Circular addressing using the BK1 field
1 1	Reserved

The reserved portion of AMR is always 0. The AMR is initialized to zero at reset.

The block size fields, BK0 and BK1, contain 5-bit values used in calculating block sizes for circular addressing.

Block size (in bytes) = $2^{(N+1)}$
where N is the 5-bit value in BK0 or BK1

Table 2–5 shows block size calculations for all 32 possibilities.

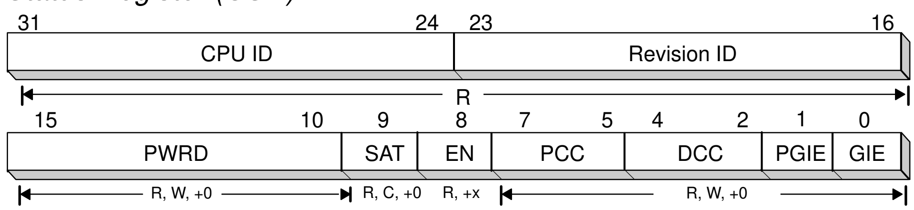
Table 2–5. Block Size Calculations

N	Block Size	N	Block Size
00000	2	10000	131 072
00001	4	10001	262 144
00010	8	10010	524 288
00011	16	10011	1 048 576
00100	32	10100	2 097 152
00101	64	10101	4 194 304
00110	128	10110	8 388 608
00111	256	10111	16 777 216
01000	512	11000	33 554 432
01001	1 024	11001	67 108 864
01010	2 048	11010	134 217 728
01011	4 096	11011	268 435 456
01100	8 192	11100	536 870 912
01101	16 384	11101	1 073 741 824
01110	32 768	11110	2 147 483 648
01111	65 536	11111	4 294 967 296

2.4 Control Status Register (CSR)

The CSR, shown in Figure 2–4, contains control and status bits. The functions of the fields in the CSR are shown in Table 2–6. For the EN, PWRD, PCC, and DCC fields, see your data sheet to see if your device supports the options that these fields control, and see the *TMS320C62xx Peripherals Reference Guide* for more information on these options.

Figure 2–4. Control Status Register (CSR)



Legend: R Readable by the **MVC** instruction
W Writeable by the **MVC** instruction
+x Value undefined after reset
+0 Value is zero after reset
C Clearable using the **MVC** instruction

Table 2–6. Control Status Register: Fields, Read/Write Status, and Function

Bit Position	Width	Field Name	Function
31-24	8	CPU ID	CPU ID; defines which CPU
23-16	8	Rev ID	Revision ID; defines silicon revision of the CPU
15-10	6	PWRD	Control power down modes; the values are always read as zero.†
9	1	SAT	The saturate bit, set when any unit performs a saturate, can be cleared only by the **MVC** instruction and can be set only by a functional unit. The set by a functional unit has priority over a clear (by the **MVC** instruction) if they occur on the same cycle. The saturate bit is set one full cycle (one delay slot) after a saturate occurs.
8	1	EN	Endian bit: 1 = little endian, 0 = big endian †
7-5	3	PCC	Program cache control mode†
4-2	3	DCC	Data cache control mode†
1	1	PGIE	Previous GIE (global interrupt enable); saves GIE when an interrupt is taken
0	1	GIE	Global interrupt enable; enables (1) or disables (0) all interrupts except the reset interrupt and NMI (nonmaskable interrupt)

† See the *TMS320C62xx Peripherals Reference Guide* for more information.

Chapter 3

Instruction Set

This chapter describes the assembly language instructions and addressing modes for the ’C62xx digital signal processor. Also described are parallel operations, conditional operations, and resource constraints.

Topic **Page**

3.1 Instruction Operation and Execution Notations

Table 3–1 explains the symbols used in this chapter.

Table 3–1. Instruction Operation and Execution Notations

Symbol	Meaning
int	32-bit register value
long	40-bit register value
creg	3-bit field specifying a conditional register
*cst*n	n-bit constant field
LSBn	n least significant bits
MSBn	n most significant bits
$\rightarrow$	Assignment
+	Addition
–	Subtraction
$\times$	Multiplication
+a	Perform 2s-complement addition using the addressing mode defined by the AMR
–a	Perform 2s-complement subtraction using the addressing mode defined by the AMR
and	Bitwise AND
or	Bitwise OR
xor	Bitwise exclusive OR
not	Bitwise logical complement
$b_{y..z}$	Selection of bits y through z of bit string b
<<	Shift left
>>s	Shift right with sign extension
>>z	Shift right with a zero fill
x clear *b,e*	Clear a field in x, specified by b (beginning bit) and e (ending bit)

Table 3–1. Instruction Operation and Execution Notations (Continued)

Symbol	**Meaning**
x exts *l,r*	Extract and sign-extend a field in x, specified by l (shift left value) and r (shift right value)
x extu *l,r*	Extract an unsigned field in x, specified by l (shift left value) and r (shift right value)
+s	Perform 2s-complement addition and saturate the result to the result size if an overflow occurs
–s	Perform 2s-complement subtraction and saturate the result to the result size if an overflow occurs
x set *b,e*	Set field in x to all 1s, specified by b (beginning bit) and e (ending bit)
abs(x)	Absolute value of x
lmb0(x)	Leftmost 0 bit search of x
lmb1(x)	Leftmost 1 bit search of x
norm(x)	Leftmost nonredundant sign bit of x
R	Any general-purpose register
cond	Check for either *creg* equal to zero or *creg* not equal to zero
nop	No operation

3.2 Mapping Between Instructions and Functional Units

Table 3–2 and Table 3–3 define the mapping between instructions and functional units.

Table 3–2. Instruction to Functional Unit Mapping

.L Unit	.M Unit	.S Unit	.D Unit
ABS	MPY	ADD	ADD
ADD	SMPY	ADDK	ADDA
AND		ADD2	LD mem
CMPEQ		AND	LD mem (15-bit offset)‡
CMPGT		B disp	MV
CMPGTU		B IRP†	NEG
CMPLT		B NRP†	ST mem
CMPLTU		B reg	ST mem (15-bit offset)‡
LMBD		CLR	SUB
MV		EXT	SUBA
NEG		EXTU	ZERO
NORM		MVC†	
NOT		MV	
OR		MVK	
SADD		MVKH	
SAT		NEG	
SSUB		NOT	
SUB		OR	
SUBC		SET	
XOR		SHL	
ZERO		SHR	
		SHRU	
		SSHL	
		SUB	
		SUB2	
		XOR	
		ZERO	

† .S2 only
‡ .D2 only

Table 3–3. Functional Unit to Instruction Mapping

	’C62xx Functional Units			
Instruction	**.L Unit**	**.M Unit**	**.S Unit**	**.D Unit**
ABS	✓			
ADD	✓		✓	✓
ADDA				✓
ADDK			✓	
ADD2			✓	
AND	✓		✓	
B			✓	
B IRP			✓†	
B NRP			✓†	
B reg			✓†	
CLR			✓	
CMPEQ	✓			
CMPGT	✓			
CMPGTU	✓			
CMPLT	✓			
CMPLTU	✓			
EXT			✓	
EXTU			✓	
IDLE				
LD mem				✓
LD mem (15-bit offset)				✓‡
LMBD	✓			
MPY		✓		
MVC†			✓	
MV	✓		✓	✓
MVK			✓	

† .S2 only
‡ .D2 only

Table 3–3. Functional Unit to Instruction Mapping (Continued)

	’C62xx Functional Units			
Instruction	**.L Unit**	**.M Unit**	**.S Unit**	**.D Unit**
MVKH			✓	
NEG	✓		✓	✓
NOP				
NORM	✓			
NOT	✓		✓	
OR	✓		✓	
SADD	✓			
SAT	✓			
SET			✓	
SHL			✓	
SHR			✓	
SHRU			✓	
SMPY		✓		
SSHL			✓	
SSUB	✓			
ST mem				✓
ST mem (15-bit offset)				✓‡
SUB	✓		✓	✓
SUBA				✓
SUBC	✓			
SUB2			✓	
SWI				
XOR	✓		✓	
ZERO	✓		✓	✓

† .S2 only
‡ .D2 only

3.3 TMS320C62xx Opcode Map

The 'C62xx opcode map is shown in Figure 3–1. Refer to Table 3–1 and the instruction descriptions in this chapter for explanations of the field syntaxes and values.

Figure 3–1. TMS320C62xx Opcode Map

Operations on the .L unit

31–29	28	27–23	22–18	17–13	12	11–5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1/cst*	x	*op*	1	1	0	*s*	*p*
3		5	5	5		7					

Operations on the .M unit

31–29	28	27–23	22–18	17–13	12	11–7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1/cst*	x	*op*	0	0	0	0	0	*s*	*p*
3		5	5	5		5							

Operations on the .D unit

31–29	28	27–23	22–18	17–13	12–7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1/cst*	*op*	1	0	0	0	0	*s*	*p*
3		5	5	5	6							

Load/store with 15-bit offset (on the .D unit)

31–29	28	27–23	22–8	7	6–4	3	2	1	0
creg	*z*	*dst/src*	*ucst15*	y	*ld/st*	1	1	*s*	*p*
3		5	15		3				

Load/store baseR + offsetR/cst on the .D unit

31–29	28	27–23	22–18	17–13	12–9	8	7	6–4	3	2	1	0
creg	*z*	*dst/src*	*baseR*	*offsetR/ucst5*	*mode*	*r*	y	*ld/st*	0	1	*s*	*p*
3		5	5	5	4			3				

Operations on the .S unit

31–29	28	27–23	22–18	17–13	12	11–6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1/cst*	x	*op*	1	0	0	0	*s*	*p*
3		5	5	5		6						

ADDK on the .S unit

31–29	28	27–23	22–7	6	5	4	3	2	1	0
creg	*z*	*dst*	*cst*	1	0	1	0	0	*s*	*p*
3		5	16							

Figure 3–1. TMS320C62xx Opcode Map (Continued)

Field operations (immediate forms) on the .S unit

31 29	28	27 23	22 18	17 13	12 8	7 6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*csta*	*cstb*	*op*	0	0	1	0	*s*	*p*
3		5	5	5	5	2						

MVK and MVKH on the .S unit

31 29	28	27 23	22 7	6	5	4	3	2	1	0
creg	z	*dst*	*dst*	*h*	1	0	1	0	*s*	*p*
3		5	16							

Bcond disp on the .S unit

31 29	28	27 7	6	5	4	3	2	1	0
creg	*z*	*cst*	0	0	1	0	0	*s*	*p*
3		21							

IDLE

31 18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	*s*	*p*
14																		1

NOP

31 18	17	16 13	12 1	0
Reserved	0	*src*	0 0 0 0 0 0 0 0 0 0 0 0	*p*
14		4		1

3.4 Delay Slots

The execution of instructions can be defined in terms of delay slots. The numbered delay slots is equivalent to the number of extra cycles required before a result is available for reading after the source operands are read. For a single-cycle type instruction (such as **ADD**), source operands read in cycle *i* produce a result that can be read in cycle $i + 1$. For a multiply instruction (**MPY**), source operands read in cycle *i* produce a result that can be read in cycle $i + 2$. Table 3–4 shows the number of delay slots associated with each type of instruction.

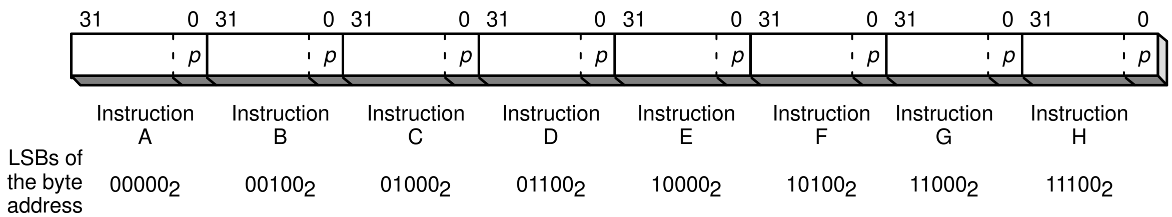
Table 3–4. Delay Slot Summary

Instruction Type	Delay Slots
NOP (no execution pipeline operation)	0
Store	0
Single cycle	0
Multiply	1
Load (LD) (address modification occurs in E1)	4
Branch (The cycle when the target enters E1)	5

3.5 Parallel Operations

Instructions are always fetched eight at a time. This constitutes a *fetch packet*. The basic format of a fetch packet is shown in Figure 3–2. The execution grouping of the fetch packet is specified by the *p*-bit (bit 0) of each instruction. Fetch packets are aligned on 256-bit (8-word) boundaries.

Figure 3–2. Basic Format of a Fetch Packet



The *p*-bit controls the parallel execution of instructions. The *p*-bits are scanned from left to right (lower to higher address). If the *p*-bit of instruction *i* is 1, then instruction $i + 1$ is to be executed in parallel with (in the the same cycle as) instruction *i*. If the *p*-bit of instruction *i* is 0, then instruction $i + 1$ is executed in the cycle after instruction *i*. All instructions executing in parallel constitute an *execute packet*. An execute packet can contain up to eight instructions. Each instruction in an execute packet must use a different functional unit.

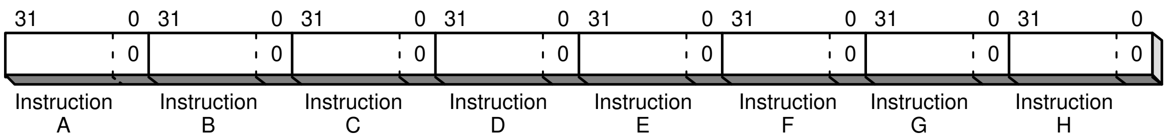
An execute packet cannot cross an 8-word boundary. Therefore, the last *p*-bit in a fetch packet is always set to 0, and each fetch packet starts a new execute packet. There are three types of *p*-bit patterns for fetch packets. These three *p*-bit patterns result in the following execution sequences for the eight instructions:

- ❑ Fully serial
- ❑ Fully parallel
- ❑ Partially serial

Example 3–1 through Example 3–3 illustrate the conversion of a *p*-bit sequence into a cycle-by-cycle execution stream of instructions.

Example 3–1. Fully Serial p-Bit Pattern in a Fetch Packet

This *p*-bit pattern:



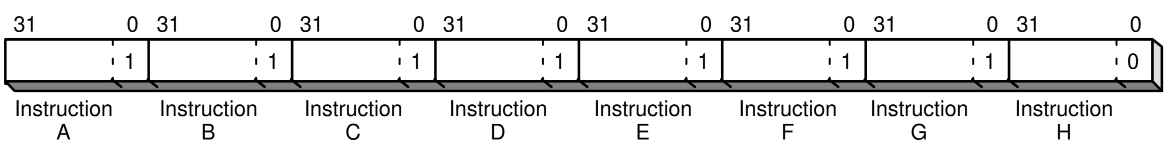
results in this execution sequence:

Cycle/Execute Packet	Instructions
1	A
2	B
3	C
4	D
5	E
6	F
7	G
8	H

The eight instructions are executed sequentially.

Example 3–2. Fully Parallel p-Bit Pattern in a Fetch Packet

This *p*-bit pattern:



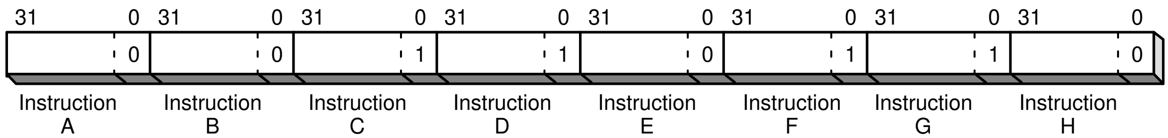
results in this execution sequence:

Cycle/Execute Packet	Instructions							
1	A	B	C	D	E	F	G	H

All eight instructions are executed in parallel.

Example 3–3. Partially Serial p-Bit Pattern in a Fetch Packet

This *p*-bit pattern:



results in this execution sequence:

Cycle/Execute Packet	Instructions		
1	A		
2	B		
3	C	D	E
4	F	G	H

Note: Instructions C, D, and E do not use any of the same functional units, cross paths, or other data path resources. This is also true for instructions F, G, and H.

3.5.1 Example Parallel Code

The || characters signify that an instruction is to execute in parallel with the previous instruction. The code for the fetch packet in Example 3–3 would be represented as this:

```
   instruction A

   instruction B

   instruction C
|| instruction D
|| instruction E

   instruction F
|| instruction G
|| instruction H
```

3.5.2 Branching Into the Middle of an Execute Packet

If a branch into the middle of an execution packet occurs, all instructions at lower addresses are ignored. In Example 3–3, if a branch to the address containing instruction D occurs, then only D and E execute. Even though instruction C is in the same execute packet, it is ignored. Instructions A and B are also ignored because they are in earlier execute packets.

3.6 Conditional Operations

All instructions can be conditional. The condition is controlled by a 3-bit opcode field (*creg*) that specifies the condition register tested, and a 1-bit field (*z*) that specifies a test for zero or nonzero. The four MSBs of every opcode are *creg* and *z*. The register is tested at the beginning of the E1 pipeline stage for all instructions. For more information on the pipeline, see Chapter 4, *Pipeline Operation*. If $z = 1$, the test is for equality with zero. If $z = 0$, the test is for nonzero. The case of $creg = 0$ and $z = 0$ is treated as always true to allow instructions to be executed unconditionally. The *creg* field is encoded in the instruction opcode as shown in Table 3–5.

Table 3–5. Registers That Can Be Tested by Conditional Operations

Conditional Register		*creg*			*z*
	Bit	***31***	***30***	***29***	***28***
Unconditional		0	0	0	0
Reserved		0	0	0	1
B0		0	0	1	*z*
B1		0	1	0	*z*
B2		0	1	1	*z*
A1		1	0	0	*z*
A2		1	0	1	*z*
Reserved		1	1	x	x

Note: x can be any value in reserved cases.

Conditional instructions are represented by using square brackets, [ ], surrounding the condition register. The following execute packet contains two **ADD** instructions in parallel. The first **ADD** is conditional on B0 being nonzero. The second **ADD** is conditional on B0 being zero. The character ! indicates the ‘not’ of the condition.

```
   [B0]  ADD   .L1   A1,A2,A3
|| [!B0] ADD   .L2   B1,B2,B3
```

The above instructions are mutually exclusive. This means that only one will execute.

If they are scheduled in parallel, mutually exclusive instructions are constrained as described in section 3.7. If mutually exclusive instructions share any resources as described in section 3.7, they cannot be scheduled in parallel (put in the same execute packet), even though only one will execute.

3.7 Resource Constraints

No two instructions within the same execute packet can use the same resources. Also, no two instructions can write to the same register during the same cycle. The following sections, 3.7.1 to 3.7.5, describe each of the resources an instruction can use.

3.7.1 Constraints on Instructions Using the Same Functional Unit

Two instructions using the same functional unit cannot be issued in the same execute packet.

The following execute packet is invalid:

```
      ADD .S1   A0, A1, A2 ; \ .S1 is used for
||    SHR .S1   A3, 15, A4 ; / both instructions
```

The following execute packet is valid:

```
      ADD .L1   A0, A1, A2 ; \ Two different functional
||    SHR .S1   A3, 15, A4 ; / units are used
```

3.7.2 Constraints on Cross Paths (1X and 2X)

One unit (either a .S, .L, or .M unit) per data path, per execute packet, can read a source operand from its opposite register file via the cross paths (1X and 2X). For example, .S1 can read both of an instruction's operands from the A register file, or it can read one operand from the B register file using the 1X cross path and the other from the A register file. This is denoted by an X following the unit name in the instruction syntax.

Two instructions using the same cross path between register files cannot be issued in the same execute packet, because there is only one path from A to B and one path from B to A.

The following execute packet is invalid:

```
      ADD.L1X   A0,B1,A1 ; \ 1X cross path is used
||    MPY.M1X   A4,B4,A5 ; / for both instructions
```

The following execute packet is valid:

```
      ADD.L1X   A0,B1,A1 ; \ Instructions use the 1X and
||    MPY.M2X   B4,A4,B2 ; / 2X cross paths
```

The operand will come from a register file opposite of the destination if the x bit in the instruction field is set (shown in the opcode map located in section 3.3).

3.7.3 Constraints on Loads and Stores

Loads and stores can use an address pointer from one register file while loading to or storing from the other register file. Two loads and/or stores using an address pointer from the same register file cannot be issued in the same execute packet.

The following execute packet is invalid:

```
   LDW.D1   *A0,A1 ; \ Address registers from the same
|| LDW.D1   *A2,B2 ; / register file
```

The following execute packet is valid:

```
   LDW.D1   *A0,A1 ; \ Address registers from different
|| LDW.D2   *B0,B2 ; / register files
```

Two loads and/or stores loading to and/or storing from the same register file cannot be issued in the same execute packet.

The following execute packet is invalid:

```
   LDW.D1   *A4,A5 ; \ Loading to and storing from the
|| STW.D2   A6,*B4 ; / same register file
```

The following execute packet is valid:

```
   LDW.D1   *A4,B5 ; \ Loading to, and storing from
|| STW.D2   A6,*B4 ; / different register files
```

3.7.4 Constraints on Long (40-Bit) Data

Because the .S and .L units share a read register port for long source operands and a write register port for long results, only one long result may be issued per register file in an execute packet. See section 2.1.1, on page 2-2 for the order for long pairs.

The following execute packet is invalid:

```
   ADD.L1   A5:A4,A1,A3:A2  ; \ Two long writes
|| SHL.S1   A8,A9,A7:A6     ; / on A register file
```

The following execute packet is valid:

```
   ADD.L1   A5:A4,A1,A3:A2   ; \ One long write for
|| SHL.S2   B8,B9,B7:B6      ; / each register file
```

Because the .L and .S units share their long read port with the store port, operations that read a long value cannot be issued on the .L and/or .S units in the same execute packet as a store.

The following execute packet is invalid:

```
   ADD.L1    A5:A4,A1,A3:A2   ; \ Long read operation and a
|| STW.D1    A8,*A9           ; / store
```

The following execute packet is valid:

```
   ADD.L1    A4, A1, A3:A2    ; \ No long read with
|| STW.D1    A8,*A9           ; / with the store
```

3.7.5 Constraints on Register Reads

More than four reads of the same register cannot occur on the same cycle. Conditional registers are not included in this count.

The following code sequence is invalid:

```
         MPY    .M1 A1,A1,A4 ; five reads of register A1
||       ADD    .L1 A1,A1,A5
||       SUB    .D1 A1,A2,A3
```

This code sequence is valid:

```
         MPY    .M1 A1,A1,A4 ; only four reads of A1
|| [A1]  ADD    .L1 A0,A1,A5
||       SUB    .D1 A1,A2,A3
```

3.7.6 Constraints on Register Writes

Multiple writes to the same register on the same cycle can occur if instructions with different latencies writing to the same register are issued on different cycles. For example, an **MPY** issued on cycle *i* followed by an **ADD** on cycle *i* + 1 cannot write to the same register because both instructions write a result on cycle *i* + 1. Therefore, the following code sequence is invalid:

```
MPY    .M1 A0,A1,A2
ADD    .L1 A4,A5,A2
```

Figure 3–3 shows different multiple-write conflicts. For example, **ADD** and **SUB** in execute packet L1 write to the same register. This conflict is easily detectable.

MPY in packet L2 and **ADD** in packet L3 might both write to B2 simultaneously; however, if a branch instruction causes the execute packet after L2 to be something other than L3, a conflict would not occur. Thus, the potential conflict in L2 and L3 might not be detected by the assembler. The instructions in L4 do not constitute a write conflict because they are mutually exclusive. In contrast, because it is not obvious that the instructions in L5 are mutually exclusive, the assembler cannot determine a conflict. If the pipeline does receive commands to perform multiple writes to the same register, the result is undefined.

Figure 3–3. Examples of the Detectability of Write Conflicts by the Assembler

```
L1:       ADD.L2   B5,B6,B7 ; \ detectable, conflict
||        SUB.S2   B8,B9,B7 ; /

L2:       MPY.M2   B0,B1,B2 ; \ not detectable

L3:       ADD.L2   B3,B4,B2 ; /

L4:[!B0]  ADD.L2   B5,B6,B7 ; \ detectable, no conflict
|| [B0]   SUB.S2   B8,B9,B7 ; /

L5:[!B1]  ADD.L2   B5,B6,B7 ; \ not detectable
|| [B0]   SUB.S2   B8,B9,B7 ; /
```

3.8 Addressing Modes

The addressing modes on the ’C62xx are linear, circular using BK0, and circular using BK1. The mode is specified by the addressing mode register, or AMR (defined in Chapter 2).

Eight registers can perform circular addressing. A4-A7 are used by the .D1 unit and B4-B7 are used by the .D2 unit. No other units can perform circular addressing. All registers can perform linear addressing. **LD(B)(H)(W)**, **ST(B)(H)(W)**, **ADDA(B)(H)(W)**, and **SUBA(B)(H)(W)** instructions all use the AMR to determine what type of address calculations are performed for these registers.

3.8.1 Linear Addressing Mode

LD/ST instructions

Linear mode simply shifts the *offsetR/cst* operand to the left by 2, 1, or 0 for word, half-word, or byte access, respectively, and then performs an add or a subtract to *baseR* (depending on the operation specified).

ADDA/SUBA instructions

Linear mode simply shifts the *src1/cst* operand to the left by 2, 1, or 0 for word, halfword, or byte data sizes, respectively, and then performs the add or subtract specified.

3.8.2 Circular Addressing Mode

The BK0 and BK1 fields in the AMR specify block sizes for circular addressing. See section 2.3 for more information on the AMR.

LD/ST Instructions

After shifting *offsetR/cst* to the left by 2, 1, or 0 for **LDW**, **LDH**, or **LDB**, respectively, an add or subtract is performed with the carry/borrow inhibited between bits N and N + 1. Bits N + 1 to 31 of *baseR* remain unchanged. All other carries/borrows propagate as usual. If you specify an *offsetR/cst* greater than the circular buffer size, $2^{(N+1)}$, the effective *offsetR/cst* is modulo the circular buffer size (see Example 3–4). The circular buffer size in the AMR is not scaled; for example: a block size of 4 is 4 bytes, not 4 × data size (byte, halfword, word). So, to perform circular addressing on an array of 8 words, a size of 32 should be specified, or N = 4. Example 3–4 shows a **LDW** performed with register A4 in circular mode and BK0 = 4, so the buffer size is 32 bytes, 16 halfwords, or 8 words. The value put in the AMR for this example is 0004 0001h.

Example 3–4. LDW in Circular Mode

```
LDW    .D1    *++A4[9],A1
```

	Before LDW		1 cycle after LDW		5 cycles after LDW
A4	0000 0100h	A4	0000 0104h	A4	XXXX XXXXh
A1	XXXX XXXXh	A1	XXXX XXXXh	A1	0000 0100h
mem 104h	1234 5678h	mem 104h	XXXX XXXXh	mem 104h	XXXX XXXXh

Note: 9h words is 24h bytes. 24h bytes is 4 bytes beyond the 32-byte (20h) boundary 100h–11Fh; thus, it is wrapped around to 104h.

ADDA/SUBA Instructions

After shifting *src*1/*cst* to the left by 2, 1, or 0 for **ADDAW**, **ADDAH**, or **ADDAB**, respectively, an add or a subtract is performed with the carry/borrow inhibited between bits N and N + 1. Bits N + 1 to 31 (inclusive) of *src*2 remain unchanged. All other carries/borrows propagate as usual. If you specify *src*1 greater than the circular buffer size, $2^{(N+1)}$, the effective *offsetR/cst* is modulo the circular buffer size (see Example 3–5). The circular buffer size in the AMR is not scaled; for example, a block size of 4 is 4 bytes, not 4 × data size (byte, half-word, word). So, to perform circular addressing on an array of 8 words, a size of 32 should be specified, or N = 4. Example 3–5 shows an **ADDAH** performed with register A4 in circular mode and BK0 = 4, so the buffer size is 32 bytes, 16 halfwords, or 8 words. The value put in the AMR for this example is 0004 0001h.

Example 3–5. ADDAH in Circular Mode

```
ADDAH     .D1    A4,A1,A1
```

	Before ADDAH		1 cycle after ADDAH
A4	0000 0100h	A4	0000 0106h
A1	0000 0013h	A1	XXXX XXXXh

Note: 13h halfwords is 26h bytes. 26h bytes is 6 bytes beyond the 32-byte (20h) boundary 100h–11Fh; thus, it is wrapped around to 106h.

3.8.3 Syntax for Load/Store Address Generation

The ’C62xx CPU has a load/store architecture, which means that the only way to access data in memory is with a load or store instruction. Table 3–6 shows the syntax of an indirect address to a memory location.

Table 3–6. Indirect Address Generation for Load/Store

Addressing Type	**No Modification of Address Register**	**Preincrement or Decrement of Address Register**	**Postincrement Decrement of Address Register**
Register indirect	*R	*++R *– –R	*R++ *R– –
Register relative	*+R[*ucst5*] *–R[*ucst5*]	*++R[*ucst5*] *– –R[*ucst5*]	*R++[*ucst5*] *R– –[*ucst5*]
Base + index	*+R[*offsetR*] *–R[*offsetR*]	*++R[*offsetR*] *– –R[*offsetR*]	*R++[*offsetR*] *R– –[*offsetR*]

3.9 Individual Instruction Descriptions

This section gives detailed information on the instruction set for the 'C62xx. Each instruction presents the following information:

- Assembler syntax
- Functional units
- Operands
- Opcode
- Description
- Execution
- Instruction type
- Delay slots
- Examples

An example instruction description is provided before the actual instruction descriptions (starting with the ABS instruction description). The example is provided to familiarize you with how the instructions are described. It includes and describes information pulled from the ADD instruction description, giving basic information about what each description category provides and where to go for more information.

Syntax

EXAMPLE (.unit) *src*, *dst*
.unit = .L1, .L2, .S1, .S2, .D1, .D2

src and *dst* indicate source and destination, respectively. The (.unit) dictates which functional unit the instruction is mapped to (.L1, .L2, .S1, .S2, .M1, .M2, .D1, or .D2).

A table is provided for each instruction that gives the opcode map fields, units the instruction is mapped to, types of operands, and the opcode.

The opcode map breaks down the various bit fields that make up each instruction. These fields are illustrated in section 3.3.

There are instructions that can be executed on more than one functional unit. Table 3–7 shows how this situation is documented for the **ADD** instruction. This instruction has three opcode map fields: *src1*, *src2*, and *dst*. In the seventh row, the operands have the types *cst5*, *long*, and *long* for *src1*, *src2*, and *dst*, respectively. The ordering of these fields implies *cst5* + *long* → *long*, where + represents the operation being performed by the **ADD**. This operation can be done on .L1 or .L2 (both are specified in the unit column). The s in front of each operand signifies that *src*1 (s*cst5*), *src*2 (slong), and *dst* (slong) are all signed values.

In the third row, *src*1, *src*2, and *dst* are int, int, and long, respectively. The u in front of each operand signifies that all operands are unsigned. Any operand that begins with x can be read from a register file that is different from the destination register file. The operand comes from the register file opposite the destination if the x bit in the instruction field is set (shown in the opcode map located in section 3.3).

Table 3–7. Relationships Between Operands, Operand Size, Signed/Unsigned, Functional Units, and Opfields for Example Instruction (ADD Instruction)

Opcode map field used...	For operand type...	Unit	Opfield	Mnemonic
src1 *src2* *dst*	sint xsint sint	.L1, .L2	0000011	ADD
src1 *src2* *dst*	sint xsint slong	.L1, .L2	0100011	ADD
src1 *src2* *dst*	uint xunit ulong	.L1, .L2	0101011	ADDU
src1 *src2* *dst*	xsint slong slong	.L1, .L2	0100001	ADD
src1 *src2* *dst*	xuint ulong ulong	.L1, .L2	0101001	ADDU
src1 *src2* *dst*	*scst*5 xsint sint	.L1, .L2	0000010	ADD
src1 *src2* *dst*	*scst*5 slong slong	.L1, .L2	0100000	ADD
src1 *src2* *dst*	sint xsint sint	.S1, .S2	000111	ADD
src1 *src2* *dst*	*scst*5 xsint sint	.S1, .S2	000110	ADD
src2 *src1* *dst*	sint sint sint	.D1, .D2	010000	ADD
src2 *src1* *dst*	sint *ucst*5 sint	.D1, .D2	010010	ADD