TMS320C55x DSP CPU Reference Guide

Literature Number: SPRU371F
February 2004

Texas Instruments

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

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

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DSP	dsp.ti.com	Broadband	www.ti.com/broadband
Interface	interface.ti.com	Digital Control	www.ti.com/digitalcontrol
Logic	logic.ti.com	Military	www.ti.com/military
Power Mgmt	power.ti.com	Optical Networking	www.ti.com/opticalnetwork
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
		Telephony	www.ti.com/telephony
		Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless

Mailing Address: Texas Instruments
Post Office Box 655303 Dallas, Texas 75265

Preface

Read This First

About This Manual

This manual describes the central processing unit (CPU) of the TMS320C55x™ (C55x™) fixed-point digital signal processors (DSPs): the architecture, registers, and operation.

Notational Conventions

This document uses the following conventions.

- If a signal or pin is active low, it has an overbar. For example, the $\overline{\text{RESET}}$ signal is active low.
- In most cases, hexadecimal numbers are shown with the suffix h. For example, the following number is a hexadecimal 40 (decimal 64):

  40h

  Similarly, binary numbers usually are shown with the suffix b. For example, the following number is the decimal number 4 shown in binary form:

  0100b

- Bits and signals are sometimes referenced with the following notations:

Notation	Description	Example
Register(n–m)	Bits n through m of Register	AC0(15–0) represents bits 15 through 0 of the register AC0.
Bus[n:m]	Signals n through m of Bus	A[21:1] represents signals 21 through 1 of bus A.

❑ The following terms are used to name portions of data:

Term	Description	Example
LSB	Least significant bit	In AC0(15–0), bit 0 is the LSB.
MSB	Most significant bit	In AC0(15–0), bit 15 is the MSB.
LSByte	Least significant byte	In AC0(15–0), bits 7–0 are the LSByte.
MSByte	Most significant byte	In AC0(15–0), bits 15–8 are the MSByte.
LSW	Least significant word	In AC0(31–0), bits 15–0 are the LSW.
MSW	Most significant word	In AC0(31–0), bits 31–16 are the MSW.

Related Documentation From Texas Instruments

The following documents describe the C55x devices and related support tools. Copies of these documents are available on the Internet at www.ti.com. *Tip:* Enter the literature number in the search box provided at www.ti.com.

TMS320C55x Technical Overview (literature number SPRU393) introduces the TMS320C55x DSPs, the latest generation of fixed-point DSPs in the TMS320C5000™ DSP platform. Like the previous generations, this processor is optimized for high performance and low-power operation. This book describes the CPU architecture, low-power enhancements, and embedded emulation features.

TMS320C55x DSP Peripherals Overview Reference Guide (literature number SPRU317) introduces the peripherals, interfaces, and related hardware that are available on TMS320C55x DSPs.

TMS320C55x DSP Algebraic Instruction Set Reference Guide (literature number SPRU375) describes the TMS320C55x DSP algebraic instructions individually. Also includes a summary of the instruction set, a list of the instruction opcodes, and a cross-reference to the mnemonic instruction set.

TMS320C55x DSP Mnemonic Instruction Set Reference Guide (literature number SPRU374) describes the TMS320C55x DSP mnemonic instructions individually. Also includes a summary of the instruction set, a list of the instruction opcodes, and a cross-reference to the algebraic instruction set.

TMS320C55x Optimizing C/C++ Compiler User's Guide (literature number SPRU281) describes the TMS320C55x C/C++ Compiler. This C/C++ compiler accepts ISO standard C and C++ source code and produces assembly language source code for TMS320C55x devices.

TMS320C55x Assembly Language Tools User's Guide (literature number SPRU280) 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 TMS320C55x devices.

TMS320C55x DSP Programmer's Guide (literature number SPRU376) describes ways to optimize C and assembly code for the TMS320C55x DSPs and explains how to write code that uses special features and instructions of the DSPs.

Trademarks

TMS320C5000, TMS320C54x, C54x, TMS320C55x, and C55x are trademarks of Texas Instruments.

This page is intentionally left blank.

Contents

Figures

Tables

Examples

This page is intentionally left blank.

Chapter 1

CPU Architecture

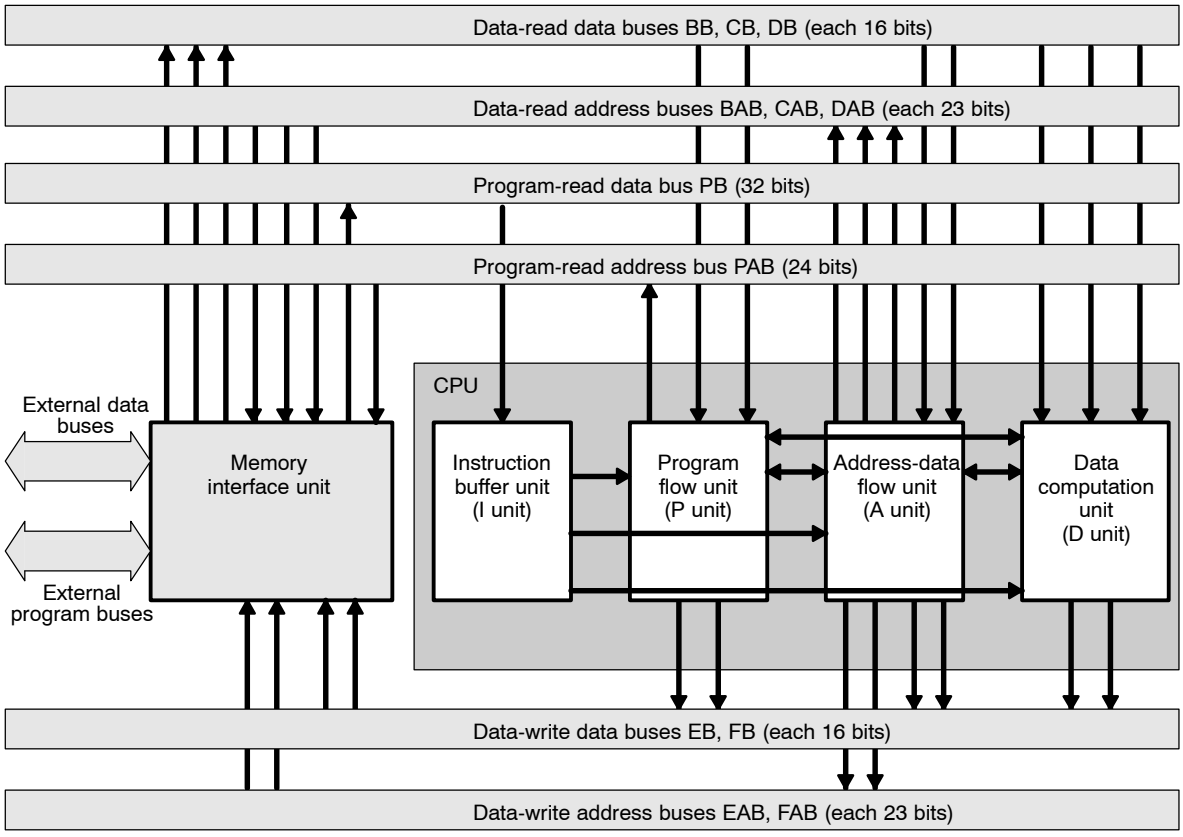
This chapter describes the CPU architecture of the TMS320C55x™ (C55x™) DSPs. It gives conceptual details about the four functional units of the CPU and about the buses that carry instructions and data. It also describes the parallel phases of the instruction pipeline and the pipeline protection mechanism (which prevents read and write operations from happening out of the intended order).

Topic **Page**

1.1 Overview of the CPU Architecture

Figure 1−1 shows a conceptual block diagram of the CPU. Sections 1.1.1 through 1.1.6 describe the buses and units represented in the figure.

Figure 1−1. CPU Diagram



1.1.1 Internal Data and Address Buses

The buses shown in Figure 1–1 are:

- **Data-Read Data Buses (BB, CB, DB).** These three buses carry 16-bit data from data space or I/O space to functional units of the CPU.

  BB only carries data from internal memory to the D unit (primarily to the dual multiply-and-accumulate (MAC) unit). BB is not connected to external memory. Specific instructions enable you to use BB, CB, and DB to read three operands at the same time.

Note:

BB and BAB are not connected to external memory. If an instruction fetches an operand using BB or BAB, the operand must be in internal memory. Inadvertent use of an external memory address generates a bus-error interrupt.

  CB and DB feed data to the P unit, the A unit, and the D unit. Instructions that read two operands at once use both CB and DB. Instructions that perform single read operations use DB.

- **Data-Read Address Buses (BAB, CAB, DAB).** These three buses carry 23-bit word data addresses to the memory interface unit, which then fetches the data from memory and transfers the requested values to the data-read data buses. All data-space addresses are generated in the A unit.

  BAB carries addresses for data that is carried from internal memory to the CPU on BB.

  CAB carries addresses for data that is carried to the CPU on CB.

  DAB carries addresses for data that is carried to the CPU on only DB or both CB and DB.

- **Program-Read Data Bus (PB).** PB carries 32 bits (4 bytes) of program code at a time to the I unit, where instructions are decoded.

- **Program-Read Address Bus (PAB).** PAB carries the 24-bit byte program address of the program code that is carried to the CPU by PB.

- **Data-Write Data Buses (EB, FB).** These two buses carry 16-bit data from functional units of the CPU to data space or I/O space.

  EB and FB receive data from the P unit, the A unit, and the D unit. Instructions that write two 16-bit values to memory at once use both EB and FB. Instructions that perform single write operations use EB.

- **Data-Write Address Buses (EAB, FAB).** These two buses carry 23-bit addresses to the memory interface unit, which then receives the values driven on the data-write data buses. All data-space addresses are generated in the A unit.

  EAB carries addresses for data that is carried to memory on only EB or both EB and FB.

  FAB carries addresses for data that is carried to memory on FB.

1.1.2 Memory Interface Unit

The memory interface mediates all data transfers between the CPU and program/data space or I/O space.

1.1.3 Instruction Buffer Unit (I Unit)

During each CPU cycle, the I unit receives 4 bytes of program code into its instruction buffer queue and decodes 1 to 6 bytes of code that were previously received in the queue. The I unit then passes data to the P unit, the A unit, and the D unit for the execution of instructions. For example, any constants that were encoded in instructions (for loading registers, providing shift counts, identifying bit numbers, etc.) are isolated in the I unit and passed to the appropriate unit.

The instruction buffer queue is emptied whenever the CPU branches to a new location.

The instruction buffer queue is loaded (but not necessarily full) for single-repeat and local-repeat operations.

1.1.4 Program Flow Unit (P Unit)

The P unit generates all program-space addresses and sends them out on PAB. It also controls the sequence of instructions by directing operations such as hardware loops, branches, and conditional execution.

1.1.5 Address-Data Flow Unit (A Unit)

The A unit contains all the logic and registers necessary to generate the data-space addresses and send them out on BAB, CAB, and DAB. It also contains a 16-bit arithmetic logic unit (ALU) that can perform arithmetical, logical, shift, and saturation operations.

1.1.6 Data Computation Unit (D Unit)

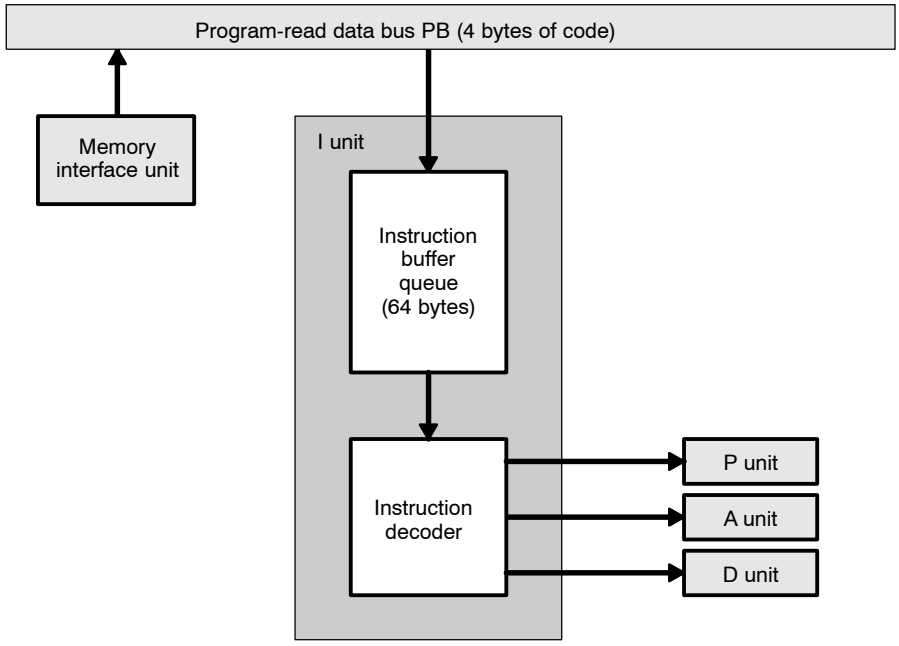
The D unit contains the primary computational units of the CPU:

- A 40-bit barrel shifter that provides a shift range of –32 to 31.
- A 40-bit arithmetic logic unit (ALU) that can perform arithmetical, logical, rounding, and saturation operations.
- A pair of multiply-and-accumulate units (MACs) that can perform a 17-bit multiplication and a 40-bit addition or subtraction in a single cycle.

1.2 Instruction Buffer Unit (I Unit)

The I unit receives program code into its instruction buffer queue and decodes instructions. The I unit then passes data to the P unit, the A unit, and the D unit for the execution of instructions. Figure 1–2 shows a conceptual block diagram of the I unit. Sections 1.2.1 and 1.2.2 describe the main parts of the I unit.

Figure 1–2. Instruction Buffer Unit (I Unit) Diagram



1.2.1 Instruction Buffer Queue

The CPU fetches 32 bits at a time from program memory. The program-read data bus (PB) carries these 32 bits from memory to the instruction buffer queue. The queue can hold up to 64 bytes of undecoded instructions. When the CPU is ready to decode instructions, 6 bytes are transferred from the queue to the instruction decoder.

In addition to helping with the pipelining of instructions, the queue enables:

- The execution of a block of code stored in the queue (local repeat instruction)
- Speculative fetching of instructions while a condition is being tested for one of the following instructions:
  - Conditional branch
  - Conditional call
  - Conditional return

1.2.2 Instruction Decoder

In the decode phase of the instruction pipeline, the instruction decoder accepts 6 bytes of program code from the instruction buffer queue and decodes those bytes. The instruction decoder:

- Identifies instruction boundaries so that it can decode 8-, 16-, 24-, 32-, 40-, and 48-bit instructions
- Determines whether the CPU has been instructed to execute two instructions in parallel.
- Sends decoded execution commands and immediate values to the program flow unit (P unit), the address-data flow unit (A unit), and the data computation unit (D unit)

Certain instructions enable writing of immediate values directly to memory or I/O space by way of a dedicated data path.

Although the decoder typically decodes no more than 6 bytes at a time, there are cases in which it decodes 7 bytes for a single instruction. An instruction listed in Table 1–1 has a 4-byte opcode and is extended by 3 bytes when the k23 absolute addressing mode is used for Smem. For details about the k23 absolute address mode, see section 6.2.2.

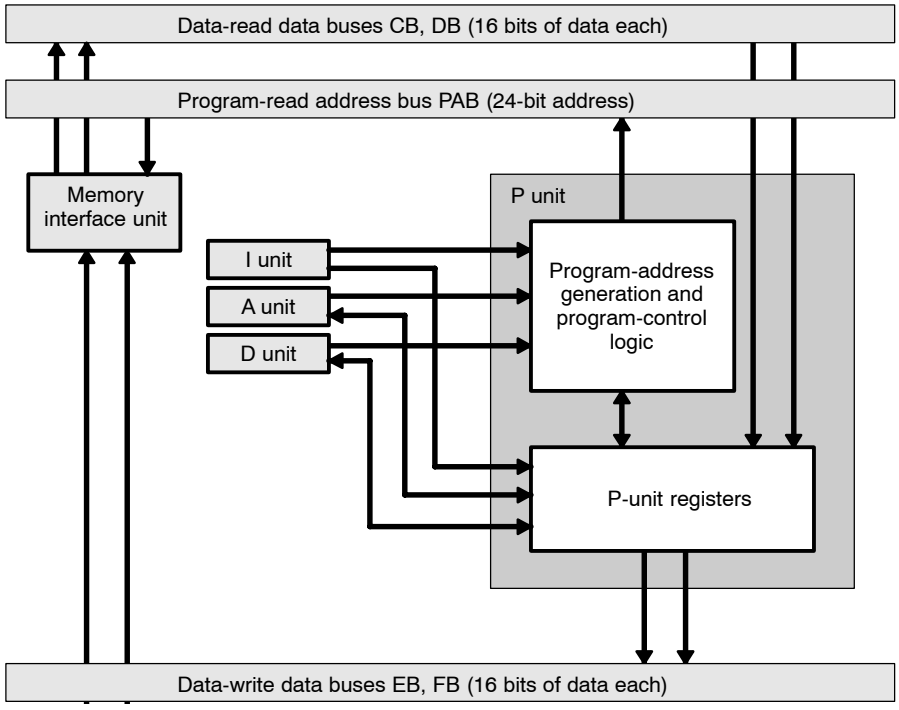
Table 1–1. 4-Byte Instructions That are Extended to 7 Bytes When the k23 Absolute Addressing Mode is Used for Smem

Instruction Syntax	Instruction Type
CMP Smem == K16, TCx	Compare memory with immediate value
BAND Smem, k16, TCx	Bitwise AND memory with immediate value
AND k16, Smem	Bitwise AND
OR k16, Smem	Bitwise OR
XOR k16, Smem	Bitwise XOR
ADD k16, Smem	Addition
MPYMK[R] [T3 =]Smem, K8, ACx	Multiply
MACMK[R] [T3 =]Smem, K8, [ACx,] ACy	Multiply and accumulate
ADD [uns(]Smem[)] << #SHIFTW, [ACx,] ACy	Addition
SUB [uns(]Smem[)] << #SHIFTW, [ACx,] ACy	Subtraction
MOV [uns(]Smem[)] << #SHIFTW, ACx	Load accumulator from memory
MOV [rnd(]HI(ACx << #SHIFTW)[)], Smem	Store accumulator content to memory
MOV [uns(] [rnd(]HI[(saturate](ACx << #SHIFTW)[)))], Smem	Store accumulator content to memory
MOV k16, Smem	Load memory with immediate value

1.3 Program Flow Unit (P Unit)

The P unit generates all program-space addresses. It also controls the sequence of instructions. Figure 1–3 shows a conceptual block diagram of the P unit. Sections 1.3.1 and 1.3.2 describe the main parts of the P unit.

Figure 1–3. Program Flow Unit (P Unit) Diagram



1.3.1 Program-Address Generation and Program-Control Logic

The program-address generation logic is responsible for generating 24-bit addresses for fetches from program memory. Normally, it generates sequential addresses. However, for instructions that require reads from nonsequential addresses, the program-address generation logic can accept immediate data from the I unit and register values from the D unit. Once an address is generated, it is carried to memory by the program-read address bus (PAB).

The program control logic accepts immediate values from the I unit and test results from the A unit or the D unit, and performs the following actions:

- ❏ Tests whether a condition is true for a conditional instruction and communicates the result to the program-address generation logic
- ❏ Initiates interrupt servicing when an interrupt is requested and properly enabled
- ❏ Controls the repetition of a single instruction preceded by a single-repeat instruction, or a block of instructions preceded by a block-repeat instruction. You can implement three levels of loops by nesting a block-repeat operation within another block-repeat operation and including a single-repeat operation in either or both of the repeated blocks. All repeat operations are interruptible.
- ❏ Manages instructions that are executed in parallel. Parallelism within the C55x DSP enables the execution of program-control instructions at the same time as data processing instructions.

1.3.2 P-Unit Registers

The P unit contains and uses the registers listed below. Access to the program flow registers is limited. You cannot read from or write to PC. You can access RETA and CFCT only with the following syntaxes:

MOV dbl(Lmem), RETA

MOV RETA, dbl(Lmem)

All the other registers can be loaded with immediate values (from the I unit) and can communicate bidirectionally with data memory, I/O space, the A-unit registers, and the D-unit registers.

Program Flow Registers

PC	Program counter
RETA	Return address register
CFCT	Control flow context register

Block-Repeat Registers

BRC0, BRC1	Block-repeat counters 0 and 1
BRS1	BRC1 save register
RSA0, RSA1	Block-repeat start address registers 0 and 1
REA0, REA1	Block-repeat end address registers 0 and 1

Single-Repeat Registers

RPTC	Single-repeat counter
CSR	Computed single-repeat register

Interrupt Registers

IFR0, IFR1	Interrupt flag registers 0 and 1
IER0, IER1	Interrupt enable registers 0 and 1
DBIER0, DBIER1	Debug interrupt enable registers 0 and 1

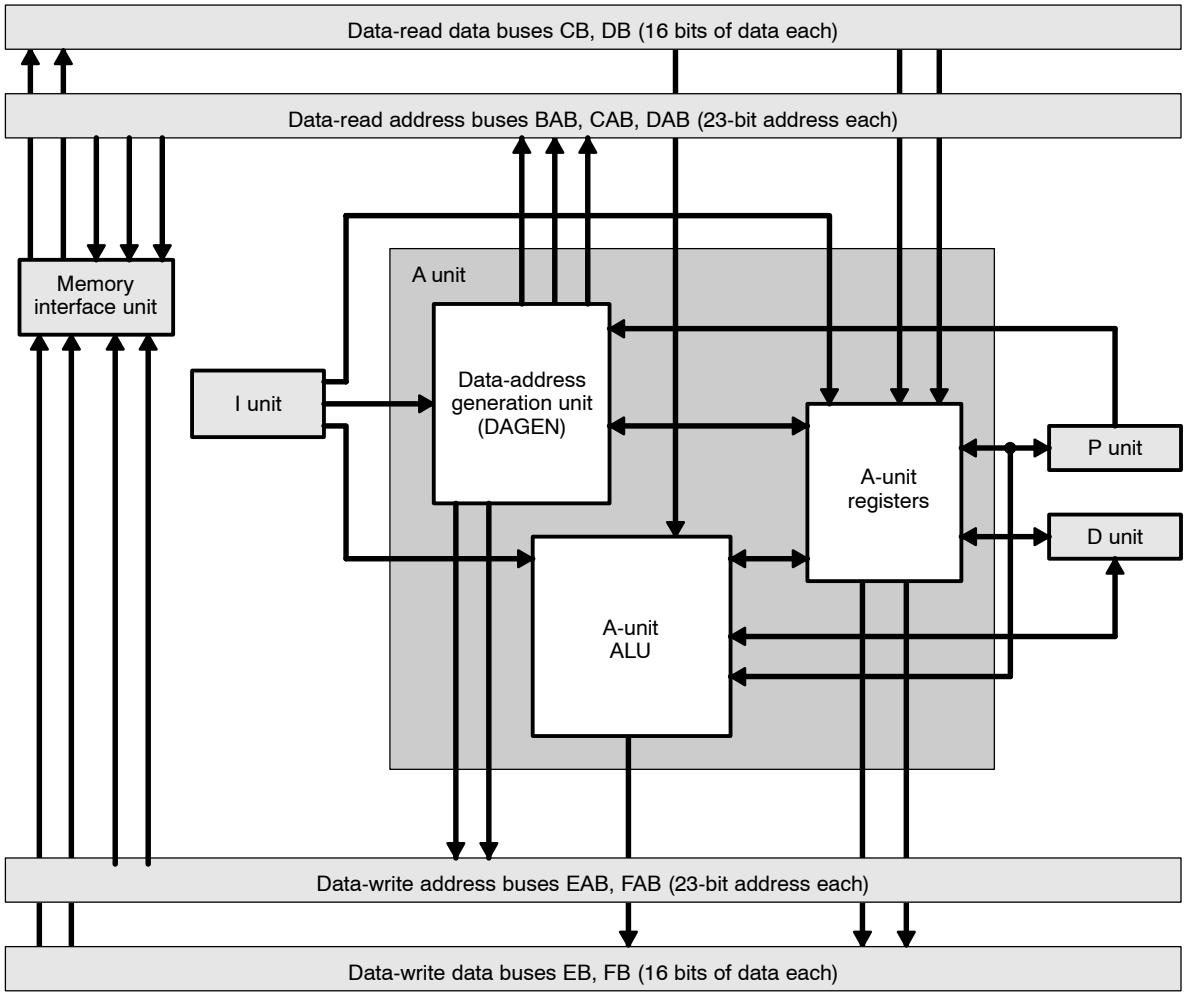
Status Registers

ST0_55–ST3_55	Status registers 0, 1, 2, and 3

1.4 Address-Data Flow Unit (A Unit)

The A unit contains all the logic and registers necessary to generate the data-space and I/O space addresses. It also contains an arithmetic logic unit (ALU) that can perform arithmetical, logical, shift, and saturation operations. Figure 1–4 shows a conceptual block diagram of the A unit. Sections 1.4.1 through 1.4.3 describe the main parts of the A unit.

Figure 1–4. Address-Data Flow Unit (A Unit) Diagram



1.4.1 Data-Address Generation Unit (DAGEN)

DAGEN generates all addresses for reads from or writes to data space and I/O space. In doing so, it can accept immediate values from the I unit and register values from the A unit. The P unit indicates to DAGEN whether to use linear or circular addressing for an instruction that uses an indirect addressing mode.

1.4.2 A-Unit Arithmetic Logic Unit (A-Unit ALU)

The A unit contains a 16-bit ALU that accepts immediate values from the I unit and communicates bidirectionally with memory, I/O space, the A-unit registers, the D-unit registers, and the P-unit registers. The A-unit ALU performs the following actions:

- ❑ Performs additions, subtractions, comparisons, Boolean logic operations, signed shifts, logical shifts, and absolute value calculations
- ❑ Tests, sets, clears, and complements A-unit register bits and memory bits
- ❑ Modifies and moves register values
- ❑ Rotates register values
- ❑ Moves certain results from the shifter to an A-unit register

1.4.3 A-Unit Registers

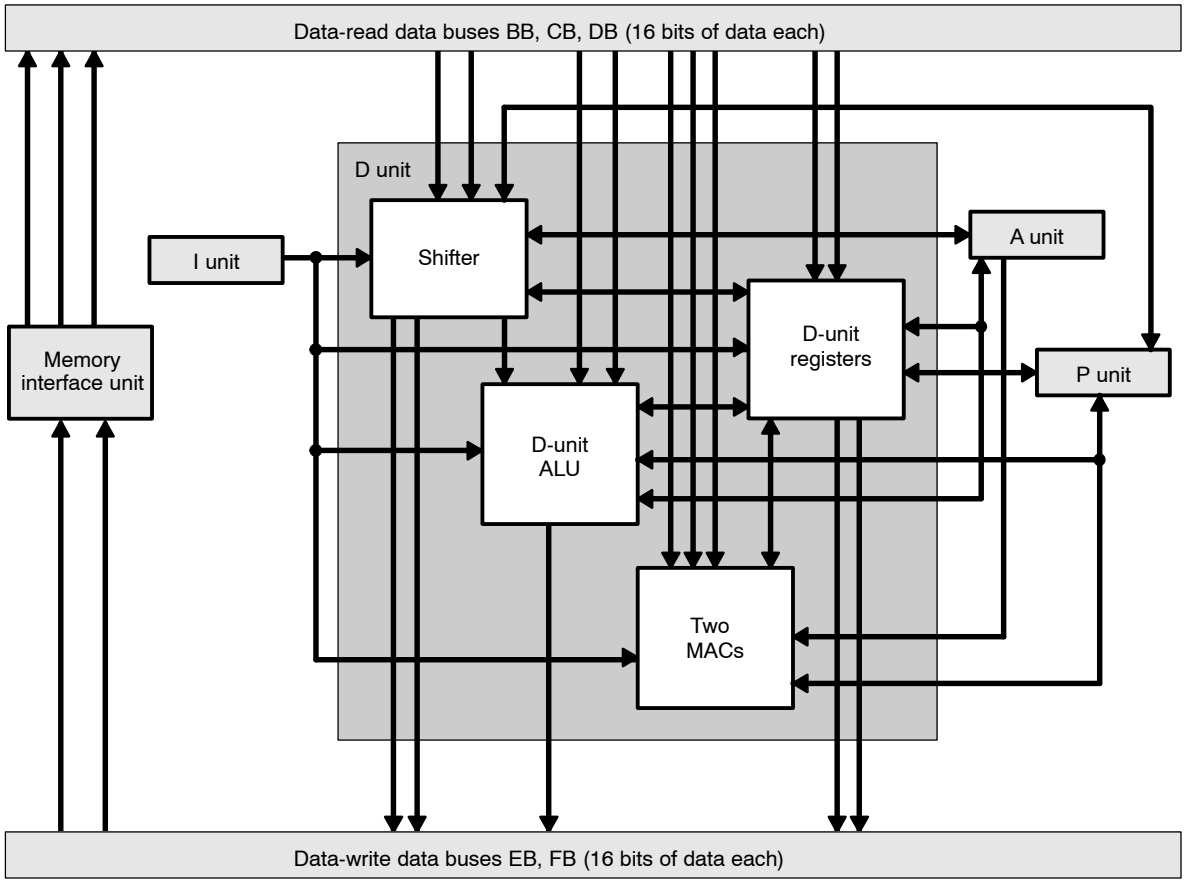
The A unit contains and uses the registers listed after this paragraph. All of these registers can accept immediate data from the I unit and can accept data from or provide data to the P-unit registers, the D-unit registers, and data memory. Within the A unit, the registers have bidirectional connections with DAGEN and the A-unit ALU.

Data Page Registers	
DPH, DP	Data page registers
PDP	Peripheral data page register
Pointers	
CDPH, CDP	Coefficient data pointer registers
SPH, SP, SSP	Stack pointer registers
XAR0–XAR7	Auxiliary registers
Circular Buffer Registers	
BK03, BK47, BKC	Circular buffer size registers
BSA01, BSA23, BSA45, BSA67, BSAC	Circular buffer start address registers
Temporary Registers	
T0–T3	Temporary registers 0, 1, 2, and 3

1.5 Data Computation Unit (D Unit)

The D unit contains the primary computational units of the CPU. Figure 1–5 shows a conceptual block diagram of the D unit. Sections 1.5.1 through 1.5.4 describe the main parts of the D unit.

Figure 1–5. Data Computation Unit (D Unit) Diagram



1.5.1 Shifter

The D-unit shifter accepts immediate values from the I unit and communicates bidirectionally with memory, I/O space, the A-unit registers, the D-unit registers, and the P-unit registers. In addition, it supplies shifted values to the D-unit ALU (as an input for further calculation) and to the A-unit ALU (as a result to be stored in an A-unit register). The shifter performs the following actions:

- Shifts 40-bit accumulator values up to 31 bits to the left or up to 32 bits to the right. The shift count can be read from one of the temporary registers (T0–T3) or it can be supplied as a constant in the instruction.
- Shifts 16-bit register, memory, or I/O-space values up to 31 bits to the left or up to 32 bits to the right. The shift count can be read from one of the temporary registers (T0–T3) or it can be supplied as a constant in the instruction.
- Shifts 16-bit immediate values up to 15 bits to the left. You supply the shift count as a constant in the instruction.
- Normalizes accumulator values
- Extracts and expands bit fields, and performs bit counting
- Rotates register values
- Rounds and/or saturates accumulator values before they are stored to data memory
- Performs additions and subtractions for some instructions that include shifting

For the C54x™-compatible mode (C54CM = 1), the overflow detection is only performed for the final operation of a calculation. For C55x-native mode (C54CM = 0), the overflow detection is performed on each operation (shifting, rounding, and addition/subtraction).

1.5.2 D-Unit Arithmetic Logic Unit (D-Unit ALU)

The CPU contains a 40-bit ALU in the D unit that accepts immediate values from the I unit and communicates bidirectionally with memory, I/O space, the A-unit registers, the D-unit registers, and the P-unit registers. In addition, it receives results from the shifter. The D-unit ALU performs the following actions:

- ❑ Performs additions, subtractions, comparisons, rounding, saturation, Boolean logic operations, and absolute value calculations
- ❑ Performs two arithmetical operations simultaneously when a dual 16-bit arithmetic instruction is executed
- ❑ Tests, sets, clears, and complements D-unit register bits
- ❑ Moves register values

1.5.3 Two Multiply-and-Accumulate Units (MACs)

Two MACs support multiplication and addition/subtraction. In a single cycle each MAC can perform a 17-bit $\times$ 17-bit multiplication (fractional or integer) and a 40-bit addition or subtraction with optional 32-/40-bit saturation. The accumulators (which are D-unit registers) receive all the results of the MACs.

The MACs accept immediate values from the I unit; accept data values from memory, I/O space, and the A-unit registers; and communicate bidirectionally with the D-unit registers and the P-unit registers. Status register bits (in the P unit) are affected by MAC operations.

Overflow detection is only performed for the final operation of a calculation.

1.5.4 D-Unit Registers

The D unit contains and uses the registers listed after this paragraph. All of these registers can accept immediate data from the I unit and can accept data from and provide data to the P-unit registers, the A-unit registers, and data memory. Within the D unit, the registers have bidirectional connections with the shifter, the D-unit ALU, and the MACs.

Accumulators

AC0–AC3	Accumulators 0, 1, 2, and 3

Transition Registers

TRN0, TRN1	Transition registers 0 and 1

1.6 Address Buses and Data Buses

The CPU is supported by one 32-bit program bus (PB), five 16-bit data buses (BB, CB, DB, EB, FB), one 24-bit address bus (PAB), and five 23-bit address buses (BAB, CAB, DAB, EAB, FAB). This parallel bus structure enables up to a 32-bit program read, three 16-bit data reads, and two 16-bit data writes per CPU clock cycle. Table 1–2 describes the functions of the 12 buses, and Table 1–3 shows which bus or buses are used for a given access type.

Table 1–2. Functions of the Address and Data Buses

Bus(es)	Width	Function
PAB	24 bits	The program-read address bus (PAB) carries a 24-bit byte address for a read from program space.
PB	32 bits	The program-read data bus (PB) carries 4 bytes (32 bits) of program code from program memory to the CPU.
CAB, DAB	23 bits each	Each of these data-read address buses carries a 23-bit word address. DAB carries an address for a read from data space or I/O space. CAB carries a second address during dual data reads (see Table 1–3).
CB, DB	16 bits each	Each of these data-read data buses carries a 16-bit data value to the CPU. DB carries a value from data space or from I/O-space. CB carries a second value during long data reads and dual data reads (see Table 1–3).
BAB	23 bits	This data-read address bus carries a 23-bit word address for a coefficient read. Many instructions that use the coefficient indirect addressing mode use BAB to reference coefficient data values (and use BB to carry the data values).
BB	16 bits	This data-read data bus carries a 16-bit coefficient data value from internal memory to the CPU. BB is not connected to external memory. Data carried by BB is addressed using BAB. Specific instructions use BB, CB, and DB to provide, in one cycle, three 16-bit operands to the CPU, using the coefficient indirect addressing mode. The operand fetched via BB must be in a memory bank other than the bank(s) accessed via CB and DB.
EAB, FAB	23 bits each	Each of these data-write address buses carries a 23-bit word address. EAB carries an address for a write to data space or I/O space. FAB carries a second address during dual data writes (see Table 1–3).
EB, FB	16 bits each	Each of these data-write data buses carries a 16-bit data value from the CPU. EB carries a value to data space or to I/O-space. FB carries a second value during long data writes and dual data writes (see Table 1–3).

Note:

In the event of a dual data write to the same address, the result is undefined.

Table 1–3. Bus Usage by Access Type

Access Type	Address Bus(es)	Data Bus(es)	Description
Instruction fetch	PAB	PB	32-bit read from program space
Single data read	DAB	DB	16-bit read from data memory
Single MMR read	DAB	DB	16-bit read from a memory-mapped register (MMR)
Single I/O read	DAB	DB	16-bit read from I/O space
Single data write	EAB	EB	16-bit write to data memory
Single MMR write	EAB	EB	16-bit write to a memory-mapped register (MMR)
Single I/O write	EAB	EB	16-bit write to I/O space
Long data read	DAB	CB, DB	32-bit read from data memory
Long MMR read	DAB	CB, DB	32-bit read from one 32-bit MMR or two adjacent 16-bit MMRs
Long data write	EAB	EB, FB	32-bit write to data memory
Long MMR write	EAB	EB, FB	32-bit write to one 32-bit MMR or two adjacent 16-bit MMRs
Dual read	CAB, DAB	CB, DB	Two simultaneous 16-bit reads from data space: ❑ The first operand read uses DAB and DB. This read can be from data memory, from an MMR, or from I/O space. ❑ The second operand read uses CAB and CB. This read must be from data memory.
Dual write	EAB, FAB	EB, FB	Two simultaneous 16-bit writes: ❑ The first operand write uses uses FAB and FB. This write must be to data memory. ❑ The second operand write uses EAB and EB. This write can be to data memory, to an MMR, or to I/O space.

Table 1–3. Bus Usage by Access Type (Continued)

Access Type	Address Bus(es)	Data Bus(es)	Description
Single data read || Single data write	DAB, EAB	DB, EB	The following two operations happen in parallel: ❑ Single data read: 16-bit read from data memory (uses DAB and DB) ❑ Single data write: 16-bit write to data memory (uses EAB and EB)
Long data read || Long data write	DAB, EAB	CB, DB, EB, FB	The following two operations happen in parallel: ❑ Long data read: 32-bit read from data memory (uses DAB, CB, and DB) ❑ Long data write: 32-bit write to data memory (uses EAB, EB, and FB)
Single data read || Coefficient data read	DAB, BAB	DB, BB	The following two operations happen in parallel: ❑ Single data read: 16-bit read from data space (uses DAB and DB) ❑ Coefficient data read: 16-bit read from **internal** memory using the coefficient indirect addressing mode (uses BAB and BB)
Dual data read || Coefficient data read	CAB, DAB, BAB	CB, DB, BB	The following two operations happen in parallel: ❑ Dual data read: Two simultaneous 16-bit reads from data space. The first operand read uses DAB and DB. The second operand read uses CAB and CB. ❑ Coefficient data read: 16-bit read from **internal** memory using the coefficient indirect addressing mode (uses BAB and BB)

1.7 Instruction Pipeline

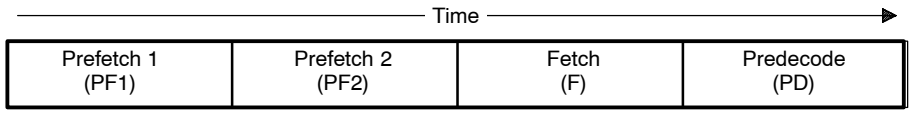
The C55x CPU uses instruction pipelining. Section 1.7.1 introduces the pipeline, and section 1.7.2 describes how the CPU prevents conflicts that might otherwise occur in the pipeline. The *TMS320C55x DSP Programmer's Guide* (literature number SPRU376) contains additional information about pipeline operation.

1.7.1 Pipeline Phases

The C55x instruction pipeline is a protected pipeline that has two decoupled segments:

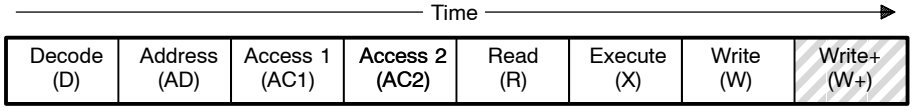
- The first segment, referred to as the *fetch pipeline*, fetches 32-bit instruction packets from memory, places them in the instruction buffer queue (IBQ), and then feeds the second pipeline segment with 48-bit instruction packets. The fetch pipeline is illustrated in Figure 1–6.
- The second segment, referred to as the *execution pipeline*, decodes instructions and performs data accesses and computations. The execution pipeline is illustrated in Figure 1–7. Table 1–4 provides examples to help you understand the activity in the key phases of the execution pipeline.

Figure 1–6. First Segment of the Pipeline (Fetch Pipeline)



Pipeline Phase	Description
PF1	Present program address to memory.
PF2	Wait for memory to respond.
F	Fetch an instruction packet from memory and place it in the IBQ.
PD	Pre-decode instructions in the IBQ (identify where instructions begin and end; identify parallel instructions).

Figure 1–7. Second Segment of the Pipeline (Execution Pipeline)



Note: Only for memory write operations

Pipeline Phase	Description
D	❑ Read six bytes from the instruction buffer queue. ❑ Decode an instruction pair or a single instruction. ❑ Dispatch instructions to the appropriate CPU functional units. ❑ Read STx_55 bits associated with data address generation: ST1_55(CPL) ST2_55(ARnLC) ST2_55(ARMS) ST2_55(CDPLC)
AD	❑ Read/modify registers involved in data address generation. For example: – ARx and T0 in *ARx+(T0) – BK03 if AR2LC = 1 – SP during pushes and pops – SSP, same as for SP if in the 32-bit stack mode ❑ Perform operations that use the A-unit ALU. For example: – Arithmetic using AADD instruction – Swapping A-unit registers with a SWAP instruction – Writing constants to A-unit registers (BKxx, BSAxx, BRCx, CSR, etc.) ❑ Decrement ARx for the conditional branch instruction that branches on ARx not zero. ❑ (Exception) Evaluate the condition of the XCC instruction (execute(AD-unit) attribute in the algebraic syntax).
AC1	For memory read operations, send addresses on the appropriate CPU address buses.
AC2	Allow one cycle for memories to respond to read requests.
R	❑ Read data from memory and MMR-addressed registers. ❑ Read A-unit registers when executing specific D-unit instructions that “prefetch” A-unit registers in the R phase rather than reading them in the X phase. ❑ Evaluate the conditions of conditional instructions. Most but not all condition evaluation is performed in the R phase. Exceptions are marked with “(Exception)” in this table.

Figure 1–7. Second Segment of the Pipeline (Execution Pipeline) (Continued)

Pipeline Phase	Description
X	❑ Read/modify registers that are not MMR-addressed. ❑ Read/modify individual register bits. ❑ Set conditions. ❑ (Exception) Evaluate the condition of the XCCPART instruction (execute(D-unit) attribute in the algebraic syntax), **unless** the instruction is conditioning a write to memory (in this case, the condition is evaluated in the R phase). ❑ (Exception) Evaluate the condition of the RPTCC instruction.
W	❑ Write data to MMR-addressed registers or to I/O space (peripheral registers). ❑ Write data to memory. From the perspective of the CPU, the write operation is finished in this pipeline phase.
W+	❑ Write data to memory. From the perspective of the memory, the write operation is finished in this pipeline phase.

Table 1–4. Examples to Illustrate Execution Pipeline Activity

Example Syntax	Pipeline Explanation
AMOV #k23, XARx	XARx is initialized with a constant in the AD phase.
MOV #k, ARx	ARx is not MMR-addressed. ARx is initialized with a constant in the X phase.
MOV #k, mmap(ARx)	ARx is MMR-addressed. ARx is initialized with a constant in the W phase.
AADD #k, ARx	With this special instruction, ARx is initialized with a constant in the AD phase.
MOV #k, *ARx+	The memory write happens in the W+ phase.
MOV *ARx+, AC0	ARx is read and updated in the AD phase. AC0 is loaded in the X phase.
ADD #k, ARx	ARx is read at the beginning of the X phase and is modified at the end of the X phase.
ADD ACy, ACx	ACx and ACy read and write activity occurs in the X phase.

Table 1–4. Examples to Illustrate Execution Pipeline Activity (Continued)

Example Syntax	Pipeline Explanation
MOV mmap(ARx), ACx	ARx is MMR-addressed and so is read in the R phase. ACx is modified in the X phase.
MOV ARx, ACx	ARx is not MMR-addressed and so is read in the X phase. ACx is modified in the X phase.
BSET CPL	The CPL bit is set in the X phase.
PUSH, POP, RET or AADD #K8, SP	SP is read and modified in the AD phase. SSP is also affected if the 32-bit stack mode is selected.
XCCPART overflow(ACx) || MOV *AR1+, AC1	The condition is evaluated in the X phase. Note: AR1 is incremented regardless of whether the condition is true.
XCCPART overflow(ACx) || MOV AC1, *AR1+	The condition is evaluated in the R phase because it conditions a write to memory. Note: AR1 is incremented regardless of whether the condition is true.
XCC overflow(ACx) || MOV *AR1+, AC1	The condition is evaluated in the AD phase. Note: AR1 is incremented only if the condition is true.

1.7.2 Pipeline Protection

Multiple instructions are executed simultaneously in the pipeline, and different instructions perform modifications to memory, I/O-space, and register values during different phases of completion. In an unprotected pipeline, this could lead to pipeline conflicts—reads and writes at the same location happening out of the intended order. However, the C55x pipeline has a mechanism that automatically protects against pipeline conflicts. The pipeline-protection mechanism adds inactive cycles between instructions that would cause conflicts.

Most pipeline-protection cycles are inserted based on two rules:

- If an instruction is supposed to write to a location but a previous instruction has not yet read from that location, extra cycles are inserted so that the read occurs first.
- If an instruction is supposed to read from a location but a previous instruction has not yet written to that location, extra cycles are inserted so that the write occurs first.

Note:

The pipeline-protection mechanism cannot prevent pipeline conflicts between two instructions that are executed in parallel.

The *TMS320C55x DSP Programmer's Guide* (literature number SPRU376) offers tips on how to minimize the number of cycles that get inserted for pipeline protection.

Chapter 2

CPU Registers

This chapter describes the main registers in a C55x DSP CPU. Section 2.1 lists the registers in alphabetical order, and section 2.2 shows the addresses for the memory-mapped registers. The other sections contain additional details about the CPU registers.

Topic **Page**

2.1 Alphabetical Summary of Registers

Table 2–1 lists the registers in alphabetical order. For more details about a particular register, see the page given in the last column of the table.

Table 2–1. Alphabetical Summary of Registers

Register Name	Description	Size	See ...
AC0–AC3	Accumulators 0 through 3	40 bits each	Page 2-9
AR0–AR7	Auxiliary registers 0 through 7	16 bits each	Page 2-12
BK03, BK47, BKC	Circular buffer size registers	16 bits each	Page 2-16
BRC0, BRC1	Block-repeat counters 0 and 1	16 bits each	Page 2-34
BRS1	BRC1 save register	16 bits	Page 2-34
BSA01, BSA23, BSA45, BSA67, BSAC	Circular buffer start address registers	16 bits each	Page 2-15
CDP	Coefficient data pointer (low part of XCDP)	16 bits	Page 2-14
CDPH	High part of XCDP	7 bits	Page 2-14
CFCT	Control-flow context register	8 bits	Page 2-21
CSR	Computed single-repeat register	16 bits	Page 2-34
DBIER0, DBIER1	Debug interrupt enable registers 0 and 1	16 bits each	Page 2-30
DP	Data page register (low part of XDP)	16 bits	Page 2-17
DPH	High part of XDP	7 bits	Page 2-17
IER0, IER1	Interrupt enable registers 0 and 1	16 bits each	Page 2-27
IFR0, IFR1	Interrupt flag registers 0 and 1	16 bits each	Page 2-24
IVPD, IVPH	Interrupt vector pointers	16 bits each	Page 2-23
PC	Program counter	24 bits	Page 2-21
PDP	Peripheral data page register	9 bits	Page 2-18
REA0, REA1	Block-repeat end address registers 0 and 1	24 bits each	Page 2-34
RETA	Return address register	24 bits	Page 2-21
RPTC	Single-repeat counter	16 bits	Page 2-34
RSA0, RSA1	Block-repeat start address registers 0 and 1	24 bits each	Page 2-34

Table 2–1. Alphabetical Summary of Registers (Continued)

Register Name	Description	Size	See ...
SP	Data stack pointer (low part of XSP)	16 bits	Page 2-18
SPH	High part of XSP and XSSP	7 bits	Page 2-18
SSP	System stack pointer (low part of XSSP)	16 bits	Page 2-18
ST0_55–ST3_55	Status registers 0 through 3	16 bits each	Page 2-37
T0–T3	Temporary registers	16 bits each	Page 2-11
TRN0, TRN1	Transition registers 0 and 1	16 bits each	Page 2-10
XAR0–XAR7	Extended auxiliary registers 0 through 7	23 bits each	Page 2-12
XCDP	Extended coefficient data pointer	23 bits	Page 2-14
XDP	Extended data page register	23 bits	Page 2-17
XSP	Extended data stack pointer	23 bits	Page 2-18
XSSP	Extended system stack pointer	23 bits	Page 2-18

2.2 Memory-Mapped Registers

Table 2–2 shows the memory-mapped registers, which are CPU registers mapped to addresses in the data space of the DSP.

Notes:

1) ST0_55, ST1_55, and ST3_55 are each accessible at two addresses. At one address, all the TMS320C55x bits are available. At the other address (the protected address), certain bits cannot be modified. The protected address is provided to support TMS320C54x™ code that writes to ST0, ST1, and PMST (the C54x™ counterpart of ST3_55).

2) T3, RSA0L, REA0L, and SP are each accessible at two addresses. For accesses using the DP direct addressing mode memory-mapped register accesses, the assembler substitutes the higher of the two addresses: T3 = 23h (not 0Eh), RSA0L = 3Dh (not 1Bh), REA0L = 3Fh (not 1Ch), SP = 4Dh (not 18h).

3) Any C55x instruction that loads BRC1 loads the same value to BRS1.

Table 2–2. Memory-Mapped Registers

Address(es)	Register	Description	Bit Range	See ...
00 0000h	IER0	Interrupt enable register 0	15–0	Page 2-27
00 0001h	IFR0	Interrupt flag register 0	15–0	Page 2-24
00 0002h (for C55x code)	ST0_55	Status register 0	15–0	Page 2-37
Note: Address 00 0002h is for native TMS320C55x code that accesses ST0_55. TMS320C54x code that was written to access ST0 should use address 00 0006h to access ST0_55.				
00 0003h (for C55x code)	ST1_55	Status register 1	15–0	Page 2-37
Note: Address 00 0003h is for native TMS320C55x code that accesses ST1_55. TMS320C54x code that was written to access ST1 should use address 00 0007h to access ST1_55.				
00 0004h (for C55x code)	ST3_55	Status register 3	15–0	Page 2-37
Note: Address 00 0004h is for native TMS320C55x code that accesses ST3_55. TMS320C54x code that was written to access the processor mode status register (PMST) should use address 00 001Dh to access ST3_55.				
00 0005h	–	Reserved (do not use this address)	–	–

Table 2–2. Memory-Mapped Registers (Continued)

Address(es)	Register	Description	Bit Range	See ...
00 0006h (for C54x code)	ST0 (ST0_55)	Status register 0	15–0	Page 2-37
Note: Address 00 0006h is the protected address of ST0_55. This address is for TMS320C54x code that was written to access ST0. Native TMS320C55x code should use address 00 0002h to access ST0_55.				
00 0007h (for C54x code)	ST1 (ST1_55)	Status register 1	15–0	Page 2-37
Note: Address 00 0007h is the protected address of ST1_55. This address is for TMS320C54x code that was written to access ST1. Native TMS320C55x code should use address 00 0003h to access ST1_55.				
00 0008h	AC0L	Accumulator 0	15–0	Page 2-9
00 0009h	AC0H		31–16	
00 000Ah	AC0G		39–32	
00 000Bh	AC1L	Accumulator 1	15–0	Page 2-9
00 000Ch	AC1H		31–16	
00 000Dh	AC1G		39–32	
00 000Eh	T3	Temporary register 3	15–0	Page 2-11
00 000Fh	TRN0	Transition register 0	15–0	Page 2-10
00 0010h	AR0	Auxiliary register 0	15–0	Page 2-12
00 0011h	AR1	Auxiliary register 1	15–0	Page 2-12
00 0012h	AR2	Auxiliary register 2	15–0	Page 2-12
00 0013h	AR3	Auxiliary register 3	15–0	Page 2-12
00 0014h	AR4	Auxiliary register 4	15–0	Page 2-12
00 0015h	AR5	Auxiliary register 5	15–0	Page 2-12
00 0016h	AR6	Auxiliary register 6	15–0	Page 2-12
00 0017h	AR7	Auxiliary register 7	15–0	Page 2-12
00 0018h	SP	Data stack pointer	15–0	Page 2-18
00 0019h	BK03	Circular buffer size register for AR0–AR3	15–0	Page 2-16
Note: In the TMS320C54x-compatible mode (C54CM = 1), BK03 is used for all the auxiliary registers. C54CM is a bit in status register 1 (ST1_55). The status registers are described beginning on page 2-37.				
00 001Ah	BRC0	Block-repeat counter 0	15–0	Page 2-34

Table 2–2. Memory-Mapped Registers (Continued)

Address(es)	Register	Description	Bit Range	See ...
00 001Bh	RSA0L	Low part of block-repeat start address register 0	15–0	Page 2-34
00 001Ch	REA0L	Low part of block-repeat end address register 0	15–0	Page 2-34
00 001Dh (for C54x code)	PMST (ST3_55)	Status register 3	15–0	Page 2-37
Note: Address 00 001Dh is the protected address of ST3_55. This address is for TMS320C54x code that was written to access the processor mode status register (PMST). Native TMS320C55x code should use address 00 0004h to access ST3_55.				
00 001Eh	XPC	This address is set aside for compatibility with TMS320C54x code that uses the program counter extension register (XPC).	7–0	–
00 001Fh	–	Reserved (do not use this address)	–	–
00 0020h	T0	Temporary register 0	15–0	Page 2-11
00 0021h	T1	Temporary register 1	15–0	Page 2-11
00 0022h	T2	Temporary register 2	15–0	Page 2-11
00 0023h	T3	Temporary register 3	15–0	Page 2-11
00 0024h	AC2L	Accumulator 2	15–0	Page 2-9
00 0025h	AC2H		31–16	
00 0026h	AC2G		39–32	
00 0027h	CDP	Coefficient data pointer	15–0	Page 2-14
00 0028h	AC3L	Accumulator 3	15–0	Page 2-9
00 0029h	AC3H		31–16	
00 002Ah	AC3G		39–32	
00 002Bh	DPH	High part of the extended data page register	6–0	Page 2-17
00 002Ch	–	Reserved (do not use these addresses)	–	–
00 002Dh	–		–	
00 002Eh	DP	Data page register	15–0	Page 2-17
00 002Fh	PDP	Peripheral data page register	8–0	Page 2-18

Table 2–2. Memory-Mapped Registers (Continued)

Address(es)	Register	Description	Bit Range	See ...
00 0030h	BK47	Circular buffer size register for AR4–AR7	15–0	Page 2-16
00 0031h	BKC	Circular buffer size register for CDP	15–0	Page 2-16
00 0032h	BSA01	Circular buffer start address register for AR0 and AR1	15–0	Page 2-15
00 0033h	BSA23	Circular buffer start address register for AR2 and AR3	15–0	Page 2-15
00 0034h	BSA45	Circular buffer start address register for AR4 and AR5	15–0	Page 2-15
00 0035h	BSA67	Circular buffer start address register for AR6 and AR7	15–0	Page 2-15
00 0036h	BSAC	Circular buffer start address register for CDP	15–0	Page 2-15
00 0037h	–	Reserved for BIOS. This location contains a 16-bit register that is used as a start-up storage location for the data table pointer necessary for BIOS operation.	–	–
00 0038h	TRN1	Transition register 1	15–0	Page 2-10
00 0039h	BRC1	Block-repeat counter 1	15–0	Page 2-34
00 003Ah	BRS1	BRC1 save register	15–0	Page 2-34
00 003Bh	CSR	Computed single-repeat register	15–0	Page 2-34
00 003Ch	RSA0H	Block-repeat start address register 0	23–16	Page 2-34
00 003Dh	RSA0L		15–0	
00 003Eh	REA0H	Block-repeat end address register 0	23–16	Page 2-34
00 003Fh	REA0L		15–0	
00 0040h	RSA1H	Block-repeat start address register 1	23–16	Page 2-34
00 0041h	RSA1L		15–0	
00 0042h	REA1H	Block-repeat end address register 1	23–16	Page 2-34
00 0043h	REA1L		15–0	
00 0044h	RPTC	Single-repeat counter	15–0	Page 2-34

Table 2–2. Memory-Mapped Registers (Continued)

Address(es)	Register	Description	Bit Range	See ...
00 0045h	IER1	Interrupt enable register 1	10–0	Page 2-27
00 0046h	IFR1	Interrupt flag register 1	10–0	Page 2-24
00 0047h	DBIER0	Debug interrupt enable register 0	15–0	Page 2-30
00 0048h	DBIER1	Debug interrupt enable register 1	10–0	Page 2-30
00 0049h	IVPD	Interrupt vector pointer for vectors 0–15 and 24–31	15–0	Page 2-23
00 004Ah	IVPH	Interrupt vector pointer for vectors 16–23	15–0	Page 2-23
00 004Bh	ST2_55	Status register 2	15–0	Page 2-37
00 004Ch	SSP	System stack pointer	15–0	Page 2-18
00 004Dh	SP	Data stack pointer	15–0	Page 2-18
00 004Eh	SPH	High part of the extended stack pointers	6–0	Page 2-18
00 004Fh	CDPH	High part of the extended coefficient data pointer	6–0	Page 2-14
00 0050h to 00 005Fh	–	Reserved (do not use these addresses)	–	–

2.3 Accumulators (AC0–AC3)

The CPU contains four 40-bit accumulators: AC0, AC1, AC2, and AC3 (see Figure 2–1). The primary function of these registers is to assist in data computation in the following parts of the D unit: the arithmetic logic unit (ALU), the multiply-and-accumulate units (MACs), and the shifter. The four accumulators are basically equivalent; however, some instructions are restricted to certain accumulator pair groupings; for example:

```
SWAP  AC0, AC2  ; Valid instruction
SWAP  AC1, AC3  ; Valid instruction
```

but,

```
SWAP  AC0, AC1  ; Invalid instruction
```

Each accumulator is partitioned into a low word (ACxL), a high word (ACxH), and eight guard bits (ACxG). You can access each of these portions individually by using addressing modes that access the memory-mapped registers.

In the TMS320C54x-compatible mode (C54CM = 1), accumulators AC0 and AC1 correspond to TMS320C54x accumulators A and B, respectively.

Figure 2–1. Accumulators

	39–32	31–16	15–0
AC0	AC0G	AC0H	AC0L
AC1	AC1G	AC1H	AC1L
AC2	AC2G	AC2H	AC2L
AC3	AC3G	AC3H	AC3L

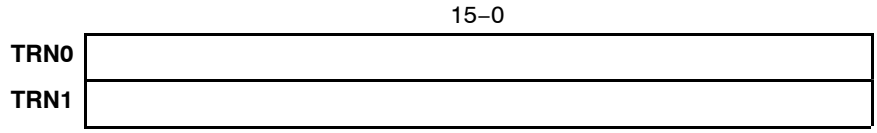
2.4 Transition Registers (TRN0, TRN1)

The two transition registers (see Figure 2–2) are used in the compare-and-select-extremum instructions:

- The syntaxes that perform two 16-bit extremum selections update TRN0 and TRN1 based on the comparison of two accumulators' high words and low words. TRN0 is updated based on the comparison of the accumulators' high words; TRN1 is updated based on the comparison of the low words.
- The syntaxes that perform a single 40-bit extremum selection update the selected transition register (TRN0 or TRN1) based on the comparison of two accumulators throughout their 40 bits.

TRN0 and TRN1 can hold transition decisions for the path to new metrics in Viterbi algorithm implementations.

Figure 2–2. Transition Registers



2.5 Temporary Registers (T0–T3)

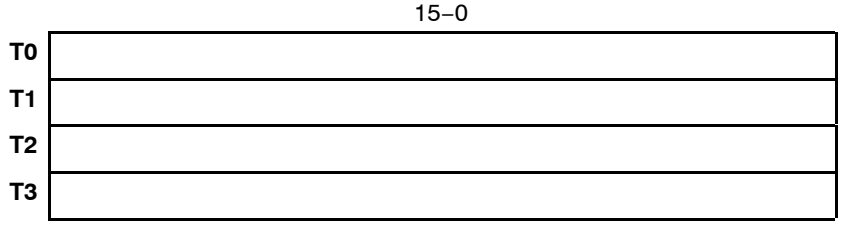
The CPU includes four 16-bit general-purpose temporary registers: T0–T3 (see Figure 2–3). Here are some of the things you can do with the temporary registers:

- Hold one of the memory multiplicands for multiply, multiply-and-accumulate, and multiply-and-subtract instructions
- Hold the shift count used in addition, subtraction, and load instructions performed in the D unit
- Keep track of more pointer values by swapping the contents of the auxiliary registers (AR0–AR7) and the temporary registers (using a swap instruction)
- Hold the transition metric of a Viterbi butterfly for dual 16-bit operations performed in the D-unit ALU

Note:

If C54CM = 1 (the TMS320C54x-compatible mode is on), T2 is tied to the ASM bits of status register ST1_55 and cannot be used as a general-purpose register. For details, see section 2.10.2.1, *ASM Bit Field of ST1_55*.

Figure 2–3. Temporary Registers



2.6 Registers Used to Address Data Space and I/O Space

This section describes the following registers:

Register(s)	Function	See ...
XAR0–XAR7 and AR0–AR7	Point to a value in data space for accesses made with indirect addressing modes	Page 2-12
XCDP and CDP	Point to a value in data space for accesses made with indirect addressing modes	Page 2-14
BSA01, BSA23, BSA45, BSA67, BSAC	Specify a circular buffer start address to be added to a pointer	Page 2-15
BK03, BK47, BKC	Specify a circular buffer size	Page 2-16
XDP and DP	Specify the start address for accesses made with the DP direct addressing mode	Page 2-17
PDP	Identify the peripheral data page for an access to I/O space	Page 2-18
XSP and SP	Point to a value on the data stack	Page 2-18
XSSP and SSP	Point to a value on the system stack	Page 2-18

2.6.1 Auxiliary Registers (XAR0–XAR7 / AR0–AR7)

The CPU includes eight extended auxiliary registers XAR0–XAR7 (see Figure 2–4 and Table 2–3). Each high part (for example, AR0H) is used to specify the 7-bit main data page for accesses to data space. Each low part (for example, AR0) can be used as:

- ❑ A 16-bit offset to the 7-bit main data page (to form a 23-bit address)
- ❑ A bit address (in instructions that access individual bits or bit pairs)
- ❑ A general-purpose register or counter
- ❑ An index to select words relative to the start address of a circular buffer (see section 6.11, *Circular Addressing)*

Figure 2–4. Extended Auxiliary Registers and Their Parts

	22–16	15–0
XAR0	AR0H	AR0
XAR1	AR1H	AR1
XAR2	AR2H	AR2
XAR3	AR3H	AR3
XAR4	AR4H	AR4
XAR5	AR5H	AR5
XAR6	AR6H	AR6
XAR7	AR7H	AR7

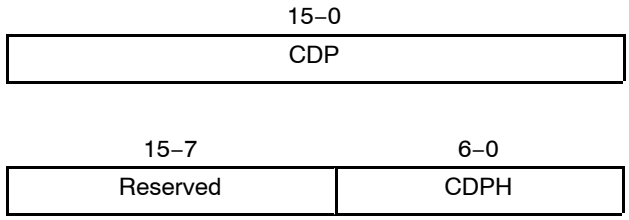
Table 2–3. Extended Auxiliary Registers and Their Parts

Register	Referred To As ...	Accessibility
XARn	Extended auxiliary register n	Accessible via dedicated instructions only. XARn is not mapped to memory.
ARn	Auxiliary register n	Accessible via dedicated instructions and as a memory-mapped register
ARnH	High part of extended auxiliary register n	Not individually accessible. To access ARnH, you must access XARn.

XAR0–XAR7 or AR0–AR7 are used in the AR indirect addressing mode and the dual AR indirect addressing mode. Basic arithmetical, logical, and shift operations can be performed on AR0–AR7 in the A-unit arithmetic logic unit (ALU). These operations can be performed in parallel with address modifications performed on the auxiliary registers in the data-address generation unit (DAGEN).

2.6.2 Coefficient Data Pointer (XCDP / CDP)

The CPU includes in its memory map a coefficient data pointer, CDP, and an associated extension register, CDPH:



The CPU can concatenate the two to form an extended CDP that is called XCDP (see Figure 2–5 and Table 2–4). The high part (CDPH) is used to specify the 7-bit main data page for accesses to data space. The low part (CDP) can be used as:

- ❑ A 16-bit offset to the 7-bit main data page (to form a 23-bit address)
- ❑ A bit address (in instructions that access individual bits or bit pairs)
- ❑ A general-purpose register or counter
- ❑ An index to select words relative to the start address of a circular buffer (see section 6.11, *Circular Addressing)*

Figure 2–5. Extended Coefficient Data Pointer and Its Parts

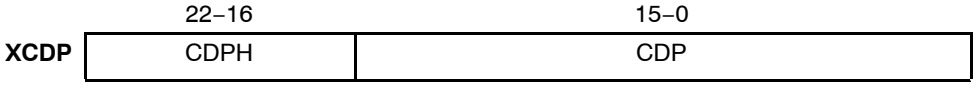


Table 2–4. Extended Coefficient Data Pointer and Its Parts

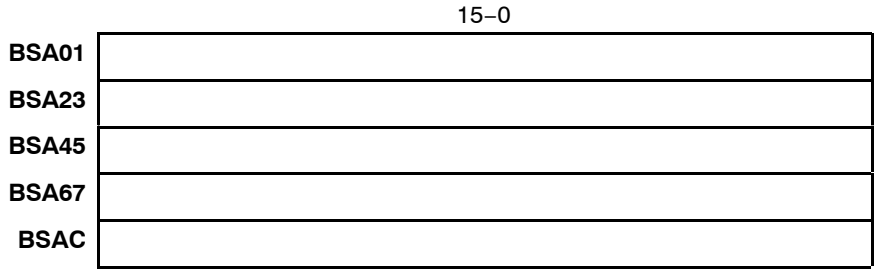
Register	Referred To As ...	Accessibility
XCDP	Extended coefficient data pointer	Accessible via dedicated instructions only. XCDP is not a register mapped to memory.
CDP	Coefficient data pointer	Accessible via dedicated instructions and as a memory-mapped register
CDPH	High part of extended coefficient data pointer	Accessible as a memory-mapped register. You can also access CDPH by accessing XCDP. There are no dedicated instructions for CDPH.

XCDP or CDP is used in the CDP indirect addressing mode and the coefficient indirect addressing mode. CDP can be used in any instruction that accesses a single data-space value; however, CDP is more advantageously used in dual multiply-and-accumulate (MAC) instructions because it provides a third, independent operand to the D-unit dual-MAC operator.

2.6.3 Circular Buffer Start Address Registers (BSA01, BSA23, BSA45, BSA67, BSAC)

The CPU includes five 16-bit circular buffer start address registers (see Figure 2–6) to enable you to define a circular buffer with a start address that is not bound by any alignment constraint.

Figure 2–6. Circular Buffer Start Address Registers



Each buffer start address register is associated with a particular pointer or pointers (see Table 2–5). A buffer start address is only added to the pointer value when the pointer is configured for circular addressing in status register ST2_55.

Table 2–5. Circular Buffer Start Address Registers and the Associated Pointers

Register	Pointer	Supplier of Main Data Page
BSA01	AR0 or AR1	AR0H for AR0 AR1H for AR1
BSA23	AR2 or AR3	AR2H for AR2 AR3H for AR3
BSA45	AR4 or AR5	AR4H for AR4 AR5H for AR5
BSA67	AR6 or AR7	AR6H for AR6 AR7H for AR7
BSAC	CDP	CDPH

As an example of using a buffer start address, consider the following instruction:

```
MOV *AR6, T2     ; Load T2 with a value from the circular
                 ; buffer of words referenced by XAR6.
```

In this example, with AR6 configured for circular addressing, the address generated is of the following form. The main data page value (AR6H) is concatenated with the sum of AR6 and its associated buffer start address (BSA67).

AR6H:(BSA67 + AR6) = XAR6 + BSA67

When you run TMS320C54x code in the compatible mode (C54CM = 1), make sure the buffer start address registers contain 0.

2.6.4 Circular Buffer Size Registers (BK03, BK47, BKC)

Three 16-bit circular buffer size registers (see Figure 2−7) specify the number of words (up to 65535) in a circular buffer. Each buffer size register is associated with a particular pointer or pointers (see Table 2−6).

Figure 2−7. Circular Buffer Size Registers

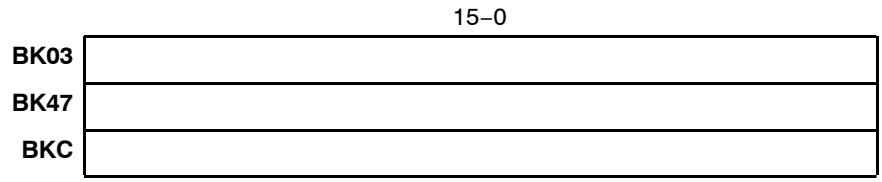


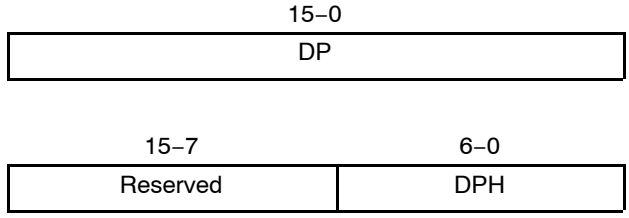
Table 2−6. Circular Buffer Size Registers and the Associated Pointers

Register	Pointer
BK03	AR0, AR1, AR2, or AR3
BK47	AR4, AR5, AR6, or AR7
BKC	CDP

In the TMS320C54x-compatible mode (C54CM = 1), BK03 is used for all the auxiliary registers and BK47 is not used.

2.6.5 Data Page Register (XDP / DP)

The CPU includes in its memory map a data page register, DP, and an associated extension register, DPH:



The CPU can concatenate the two to form an extended DP that is called XDP (see Figure 2–8 and Table 2–7). The high part (DPH) is used to specify the 7-bit main data page for accesses to data space. The low part specifies a 16-bit offset (local data page) that is concatenated with the main data page to form a 23-bit address.

Figure 2–8. Extended Data Page Register and Its Parts

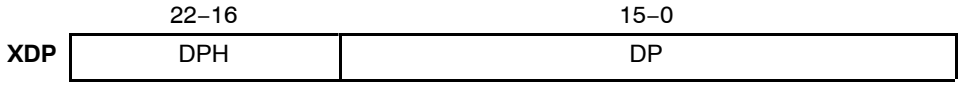


Table 2–7. Extended Data Page Register and Its Parts

Register	Referred To As ...	Accessibility
XDP	Extended data page register	Accessible via dedicated instructions only. XDP is not a register mapped to memory.
DP	Data page register	Accessible via dedicated instructions and as a memory-mapped register
DPH	High part of extended data page register	Accessible via dedicated instructions and as a memory-mapped register

In the DP direct addressing mode, XDP specifies a 23-bit address, and in the k16 absolute addressing mode, DPH is concatenated with a 16-bit immediate value to form a 23-bit address.

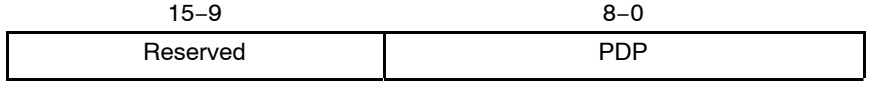
2.6.6 Peripheral Data Page Register (PDP)

For the PDP direct addressing mode, the 9-bit peripheral data page register (PDP) selects a 128-word page within the 64K-word I/O space.

As shown in Figure 2–9, PDP is a 9-bit field within a 16-bit register location. Bits 15–9 of that location are ignored by the CPU.

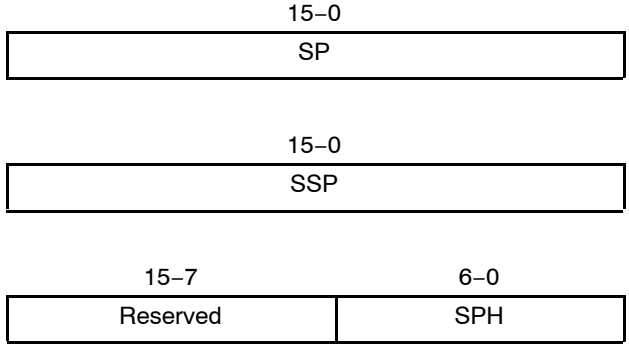
This register is accessible via dedicated instructions and as a memory-mapped register.

Figure 2–9. Peripheral Data Page Register



2.6.7 Stack Pointers (XSP / SP, XSSP / SSP)

The CPU includes in its memory map a data stack pointer (SP), a system stack pointer (SSP), and an associated extension register (SPH):



See Figure 2–10 and Table 2–8. When accessing the data stack, the CPU concatenates SPH with SP to form an extended SP that is called XSP. XSP contains the address of the value last pushed onto the data stack. SPH holds the 7-bit main data page of memory, and SP points to the specific word on that page.

Similarly, when accessing the system stack, the CPU concatenates SPH with SSP to form XSSP. XSSP contains the address of the value last pushed onto the system stack.

Figure 2–10. Extended Stack Pointers

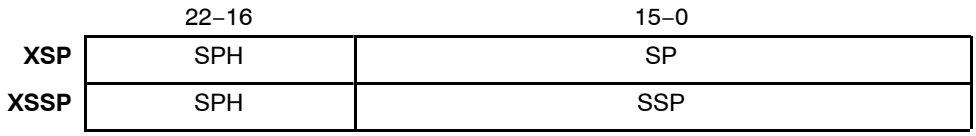


Table 2–8. Stack Pointer Registers

Register	Referred To As ...	Accessibility
XSP	Extended data stack pointer	Accessible via dedicated instructions only. XSP is not a register mapped to memory.
SP	Data stack pointer	Accessible via dedicated instructions and as a memory-mapped register
XSSP	Extended system stack pointer	Accessible via dedicated instructions only. XSSP is not a register mapped to memory.
SSP	System stack pointer	Accessible via dedicated instructions and as a memory-mapped register
SPH	High part of XSP and XSSP	Accessible as a memory-mapped register. You can also access SPH by accessing XSP or XSSP. There are no dedicated instructions for SPH. **Note:** SPH is affected by writes to XSP or XSSP.

XSP is used in the SP direct addressing mode. The instructions in Table 2–9 use and/or modify SP and SSP.

Table 2–9. Instructions That Use and/or Modify SP and SSP

Instruction Type(s)	Description
Software interrupt, software trap, software reset, call unconditionally, call conditionally	These instructions push data onto the data stack and the system stack. SP and SSP are decremented before each pair of data values is pushed.
Push	This instruction pushes data onto the data stack only. SP is decremented before the data is pushed.
Return unconditionally, return conditionally, return from interrupt	These instructions pop data from the data stack and the system stack. SP and SSP are incremented after each pair of data values is popped.
Pop	This instruction pops data from the data stack only. SP is incremented after the data is popped.

Stack pointer increments and decrements are made to SP and SSP. You cannot address the stacks across main data pages without changing the value in the extension register (SPH).

Note:

Although an increment past FFFFh or a decrement past 0000h causes the pointer value to wrap around, do not make use of this behavior; it is not supported.

2.7 Program Flow Registers (PC, RETA, CFCT)

Table 2–10 describes three registers used by the CPU to maintain proper program flow.

Table 2–10. Program Flow Registers

Register	Description
PC	Program counter. This 24-bit register holds the address of the 1 to 6 bytes of code being decoded in the I unit. When the CPU performs an interrupt or call, the current PC value (the return address) is stored, and then PC is loaded with a new address. When the CPU returns from an interrupt service routine or a called subroutine, the return address is restored to PC.
RETA	Return address register. If the selected stack configuration (see section 4.2) uses the fast-return process, RETA is a temporary holding place for the return address while a subroutine is being executed. RETA, along with CFCT, enables the efficient execution of multiple layers of subroutines. You can read from or write to RETA and CFCT as a pair with dedicated 32-bit load and store instructions.
CFCT	Control-flow context register. The CPU keeps a record of active repeat loops (the loop context). If the selected stack configuration (see section 4.2) uses the fast-return process, CFCT is a temporary holding place for the 8-bit loop context while a subroutine is being executed. CFCT, along with RETA, enables the efficient execution of multiple layers of subroutines. You can read from or write to RETA and CFCT as a pair with dedicated, 32-bit load and store instructions.

Note:

RETA and CFCT are cleared to 0 by a DSP hardware reset, and are not affected by push/pop instructions or by a software reset.

2.7.1 Context Bits Stored in CFCT

The CPU has internal bits for storing the loop context—the status (active or inactive) of repeat loops in a routine. When the CPU follows an interrupt or a call, the loop context is stored in CFCT. When the CPU returns from an interrupt or called subroutine, the loop context is restored from CFCT. In the 8-bit CFCT, the loop context bits have the form shown in Table 2–11.

Table 2–11. Form of Loop Context Bits in CFCT

Bit(s)	Description
7	This bit reflects whether a single-repeat loop is active. 0 Not active 1 Active
6	This bit reflects whether a conditional single-repeat loop is active. 0 Not active 1 Active
5–4	Reserved
3–0	This 4-bit code reflects the status of the two possible levels of block-repeat loops, the outer (level 0) loop and the inner (level 1) loop. Depending on which type of block-repeat instruction you choose, an active loop is local (all its code is repeatedly executed from within the instruction buffer queue) or external (its code is repeatedly fetched and transferred through the buffer queue to the CPU).

Block-Repeat Code	Level 0 Loop Is ...	Level 1 Loop Is ...
0	Not active	Not active
2	Active, external	Not active
3	Active, local	Not active
7	Active, external	Active, external
8	Active, external	Active, local
9	Active, local	Active, local
Other: Reserved	–	–

2.8 Registers for Managing Interrupts

This section describes the following registers:

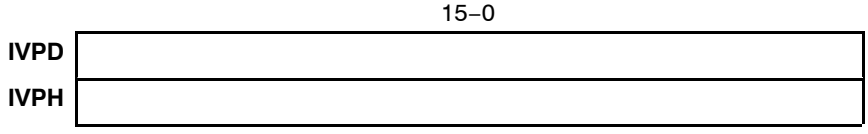
Register(s)	Function	See Section ...
IVPD	Points to interrupt vectors 0–15 and 24–31	2.8.1
IVPH	Points to interrupt vectors 16–23	2.8.1
IFR0, IFR1	Indicate which maskable interrupts have been requested	2.8.2
IER0, IER1	Enable or disable maskable interrupts	2.8.3
DBIER0, DBIER1	Configure select maskable interrupts as time-critical interrupts during debugging	2.8.4

2.8.1 Interrupt Vector Pointers (IVPD, IVPH)

Two 16-bit interrupt vector pointers IVPD and IVPH (see Figure 2–11) point to up to 32 interrupt vectors in program space. IVPD points to the 256-byte program page for interrupt vectors 0–15 and 24–31. IVPH points to the 256-byte program page for interrupt vectors 16–23.

If IVPD and IVPH have the same value, all of the interrupt vectors are in the same 256-byte program page. A DSP hardware reset loads both IVPs with FFFFh. The IVPs are not affected by a software reset instruction.

Figure 2–11. Interrupt Vector Pointers



Before you modify the IVPs, make sure that:

- Maskable interrupts are globally disabled (INTM = 1). This prevents a maskable interrupt from occurring before the IVPs are modified to point to new vectors.
- Each hardware nonmaskable interrupt has a vector and an interrupt service routine for the old IVPD value and for the new IVPD value. This prevents fetching of an illegal instruction code if a hardware nonmaskable interrupt occurs during the process of modifying the IVPD.

Table 2–12 shows how the vector addresses are formed for the different interrupt vectors. The CPU concatenates a 16-bit interrupt vector pointer with a vector number coded on 5 bits (for example, 00001 for IV1 and 10000 for IV16) and shifted left by 3 bits.

Table 2–12. Vectors and the Formation of Vector Addresses

		Vector Address		
Vector(s)	**Interrupt(s)**	**Bits 23–8**	**Bits 7–3**	**Bits 2–0**
IV0	Reset	IVPD	00000	000
IV1	Nonmaskable hardware interrupt, $\overline{\text{NMI}}$	IVPD	00001	000
IV2–IV15	Maskable interrupts	IVPD	00010 to 01111	000
IV16–IV23	Maskable interrupts	IVPH	10000 to 10111	000
IV24	Bus error interrupt (maskable), BERRINT	IVPD	11000	000
IV25	Data log interrupt (maskable), DLOGINT	IVPD	11001	000
IV26	Real-time operating system interrupt (maskable), RTOSINT	IVPD	11010	000
IV27–IV31	General-purpose software-only interrupts INT27–INT31	IVPD	11011 to 11111	000

2.8.2 Interrupt Flag Registers (IFR0, IFR1)

The 16-bit interrupt flag registers, IFR1 and IFR0, contain flag bits for all the maskable interrupts. When a maskable interrupt request reaches the CPU, the corresponding flag is set to 1 in one of the IFRs. This indicates that the interrupt is pending or waiting for acknowledgement from the CPU. Figure 2–12 is a general representation of the C55x IFRs. To see which interrupts are mapped to these bits, see the applicable C55x DSP data manual.

You can read the IFRs to identify pending interrupts, and write to the IFRs to clear pending interrupts. To clear an interrupt request (and clear its IFR bit to 0), write a 1 to the corresponding IFR bit. For example:

```
; Clear flags IF14 and IF2:
MOV #0100000000000100b, mmap(@IFR0)
```

All pending interrupts can be cleared by writing the current contents of the IFR back into the IFR. Acknowledgement of a hardware interrupt request also clears the corresponding IFR bit. A device reset clears all IFR bits.

Figure 2−12. Interrupt Flag Registers

IFR1

15				11	10	9	8
Reserved					RTOSINTF	DLOGINTF	BERRINTF
R−0					R/W1C−0	R/W1C−0	R/W1C−0
7	6	5	4	3	2	1	0
IF23	IF22	IF21	IF20	IF19	IF18	IF17	IF16
R/W1C−0	R/W1C−0	R/W1C−0	R/W1C−0	R/W1C−0	R/W1C−0	R/W1C−0	R/W1C−0

IFR0

15	14	13	12	11	10	9	8
IF15	IF14	IF13	IF12	IF11	IF10	IF9	IF8
R/W1C−0	R/W1C−0	R/W1C−0	R/W1C−0	R/W1C−0	R/W1C−0	R/W1C−0	R/W1C−0
7	6	5	4	3	2	1	0
IF7	IF6	IF5	IF4	IF3	IF2	Reserved	
R/W1C−0	R/W1C−0	R/W1C−0	R/W1C−0	R/W1C−0	R/W1C−0	R−0	

Legend: R = Read access; W1C = Writing a 1 to this bit causes the CPU to clear this bit to 0; *-n* = Value after DSP hardware reset; Reserved = A write to this bit has no effect, and the bits in this field always appear as 0s during read operations.

2.8.2.1 RTOSINTF Bit in IFR1

Bit	Name	Description	Accessibility	HW Reset
10	RTOSINTF	Interrupt flag bit for the real-time operating system interrupt, RTOSINT	Read/Write	0

When you read the RTOSINTF bit, interpret it as follows:

RTOSINTF	Description
0	RTOSINT is not pending.
1	RTOSINT is pending.

To clear this flag bit to 0 (and clear its corresponding interrupt request), write a 1 to the bit.

2.8.2.2 DLOGINTF Bit in IFR1

Bit	Name	Description	Accessibility	HW Reset
9	DLOGINTF	Interrupt flag bit for the data log interrupt, DLOGINT	Read/Write	0

When you read the DLOGINTF bit, interpret it as follows:

DLOGINTF	Description
0	DLOGINT is not pending.
1	DLOGINT is pending.

To clear this flag bit to 0 (and clear its corresponding interrupt request), write a 1 to the bit.

2.8.2.3 BERRINTF Bit in IFR1

Bit	Name	Description	Accessibility	HW Reset
8	BERRINTF	Interrupt flag bit for the bus error interrupt, BERRINT	Read/Write	0

When you read the BERRINTF bit, interpret it as follows:

BERRINTF	Description
0	BERRINT is not pending.
1	BERRINT is pending.

To clear this flag bit to 0 (and clear its corresponding interrupt request), write a 1 to the bit.

2.8.2.4 IF16–IF23 Bits in IFR1

Bit	Name	Description	Accessibility	HW Reset
0	IF16	Interrupt flag bit 16	Read/Write	0
1	IF17	Interrupt flag bit 17	Read/Write	0
2	IF18	Interrupt flag bit 18	Read/Write	0
3	IF19	Interrupt flag bit 19	Read/Write	0
4	IF20	Interrupt flag bit 20	Read/Write	0
5	IF21	Interrupt flag bit 21	Read/Write	0
6	IF22	Interrupt flag bit 22	Read/Write	0
7	IF23	Interrupt flag bit 23	Read/Write	0

When you read these bits, interpret them as follows (x is a number from 16 to 23):

IFx	Description
0	The interrupt associated with interrupt vector x is not pending.
1	The interrupt associated with interrupt vector x is pending.

To clear a flag bit to 0 (and clear its corresponding interrupt request), write a 1 to the bit.

2.8.2.5 IF2–IF15 Bits in IFR0

Bit	Name	Description	Accessibility	HW Reset
2	IF2	Interrupt flag bit 2	Read/Write	0
3	IF3	Interrupt flag bit 3	Read/Write	0
4	IF4	Interrupt flag bit 4	Read/Write	0
5	IF5	Interrupt flag bit 5	Read/Write	0
6	IF6	Interrupt flag bit 6	Read/Write	0
7	IF7	Interrupt flag bit 7	Read/Write	0
8	IF8	Interrupt flag bit 8	Read/Write	0
9	IF9	Interrupt flag bit 9	Read/Write	0
10	IF10	Interrupt flag bit 10	Read/Write	0
11	IF11	Interrupt flag bit 11	Read/Write	0
12	IF12	Interrupt flag bit 12	Read/Write	0
13	IF13	Interrupt flag bit 13	Read/Write	0
14	IF14	Interrupt flag bit 14	Read/Write	0
15	IF15	Interrupt flag bit 15	Read/Write	0

When you read these bits, interpret them as follows (x is a number from 2 to 15):

IFx	Description
0	The interrupt associated with interrupt vector x is not pending.
1	The interrupt associated with interrupt vector x is pending.

To clear a flag bit to 0 (and clear its corresponding interrupt request), write a 1 to the bit.

2.8.3 Interrupt Enable Registers (IER0, IER1)

To enable a maskable interrupt, set its corresponding bit in IER0 or IER1 to 1. To disable a maskable interrupt, clear its corresponding enable bit to 0. At a DSP hardware reset, all the IER bits are cleared to 0, disabling all the maskable interrupts. Figure 2–13 is a general representation of the C55x IERs. To see which interrupts are mapped to these bits, see the applicable C55x DSP data manual.

Note:

IER1 and IER0 are not affected by a software reset instruction. Initialize these registers before globally enabling (INTM = 0) the maskable interrupts.

Figure 2−13. Interrupt Enable Registers

IER1

15				11	10	9	8
Reserved					RTOSINTE	DLOGINTE	BERRINTE
R−0					R/W−0	R/W−0	R/W−0

7	6	5	4	3	2	1	0
IE23	IE22	IE21	IE20	IE19	IE18	IE17	IE16
R/W−0	R/W−0	R/W−0	R/W−0	R/W−0	R/W−0	R/W−0	R/W−0

IER0

15	14	13	12	11	10	9	8
IE15	IE14	IE13	IE12	IE11	IE10	IE9	IE8
R/W−0	R/W−0	R/W−0	R/W−0	R/W−0	R/W−0	R/W−0	R/W−0

7	6	5	4	3	2	1	0
IE7	IE6	IE5	IE4	IE3	IE2	Reserved	
R/W−0	R/W−0	R/W−0	R/W−0	R/W−0	R/W−0	R−0	

Legend: R = Read; W = Write; *-n* = Value after DSP hardware reset

2.8.3.1 RTOSINTE Bit in IER1

Bit	Name	Description	Accessibility	HW Reset
10	RTOSINTE	Enable bit for the real-time operating system interrupt, RTOSINT	Read/Write	0

The RTOSINTE bit enables or disables RTOSINT:

RTOSINTE	Description
0	RTOSINT is disabled.
1	RTOSINT is enabled.

2.8.3.2 DLOGINTE Bit in IER1

Bit	Name	Description	Accessibility	HW Reset
9	DLOGINTE	Enable bit for the data log interrupt, DLOGINT	Read/Write	0

The DLOGINTE bit enables or disables DLOGINT:

DLOGINTE	Description
0	DLOGINT is disabled.
1	DLOGINT is enabled.

2.8.3.3 BERRINTE Bit in IER1

Bit	Name	Description	Accessibility	HW Reset
8	BERRINTE	Enable bit for the bus error interrupt, BERRINT	Read/Write	0

The BERRINTE bit enables or disables BERRINT:

BERRINTE	Description
0	BERRINT is disabled.
1	BERRINT is enabled.

2.8.3.4 IE16–IE23 Bits in IER1

Bit	Name	Description	Accessibility	HW Reset
0	IE16	Interrupt enable bit 16	Read/Write	0
1	IE17	Interrupt enable bit 17	Read/Write	0
2	IE18	Interrupt enable bit 18	Read/Write	0
3	IE19	Interrupt enable bit 19	Read/Write	0
4	IE20	Interrupt enable bit 20	Read/Write	0
5	IE21	Interrupt enable bit 21	Read/Write	0
6	IE22	Interrupt enable bit 22	Read/Write	0
7	IE23	Interrupt enable bit 23	Read/Write	0

The functions of these bits can be summarized as follows, where x is a number from 16 to 23:

IEx	Description
0	The interrupt associated with interrupt vector x is disabled.
1	The interrupt associated with interrupt vector x is enabled.

2.8.3.5 IE2–IE15 Bits in IER0

Bit	Name	Description	Accessibility	HW Reset
2	IE2	Interrupt enable bit 2	Read/Write	0
3	IE3	Interrupt enable bit 3	Read/Write	0
4	IE4	Interrupt enable bit 4	Read/Write	0
5	IE5	Interrupt enable bit 5	Read/Write	0
6	IE6	Interrupt enable bit 6	Read/Write	0
7	IE7	Interrupt enable bit 7	Read/Write	0
8	IE8	Interrupt enable bit 8	Read/Write	0
9	IE9	Interrupt enable bit 9	Read/Write	0
10	IE10	Interrupt enable bit 10	Read/Write	0
11	IE11	Interrupt enable bit 11	Read/Write	0
12	IE12	Interrupt enable bit 12	Read/Write	0
13	IE13	Interrupt enable bit 13	Read/Write	0
14	IE14	Interrupt enable bit 14	Read/Write	0
15	IE15	Interrupt enable bit 15	Read/Write	0

The functions of these bits can be summarized as follows, where x is a number from 2 to 15:

IEx	Description
0	The interrupt associated with interrupt vector x is disabled.
1	The interrupt associated with interrupt vector x is enabled.

2.8.4 Debug Interrupt Enable Registers (DBIER0, DBIER1)

The 16-bit debug interrupt enable registers, DBIER1 and DBIER0, are used only when the CPU is *halted* in the real-time emulation mode of the debugger. If the CPU is *running* in real-time mode, the standard interrupt-handling process is used and the DBIERs are ignored.

A maskable interrupt enabled in a DBIER is defined as a *time-critical interrupt*. When the CPU is halted in the real-time mode, the only interrupts that are serviced are time-critical interrupts that are also enabled in an interrupt enable register (IER1 or IER0).

Read the DBIERs to identify time-critical interrupts. Write the DBIERs to enable or disable time-critical interrupts. To enable an interrupt, set its corresponding bit. To disable an interrupt, clear its corresponding bit. Figure 2–14 is a general representation of the C55x DBIERs. To see which interrupts are mapped to these bits, see the applicable C55x DSP data manual.

Notes:

1) DBIER1 and DBIER0 are not affected by a software reset instruction. Initialize these registers before you use the real-time emulation mode.

2) All DBIER bits are cleared to 0 by a DSP hardware reset, disabling all time-critical interrupts.

Figure 2−14. Debug Interrupt Enable Registers

DBIER1

15				11	10	9	8
Reserved					RTOSINTD	DLOGINTD	BERRINTD
R−0					R/W−0	R/W−0	R/W−0

7	6	5	4	3	2	1	0
DBIE23	DBIE22	DBIE21	DBIE20	DBIE19	DBIE18	DBIE17	DBIE16
R/W−0	R/W−0	R/W−0	R/W−0	R/W−0	R/W−0	R/W−0	R/W−0

DBIER0

15	14	13	12	11	10	9	8
DBIE15	DBIE14	DBIE13	DBIE12	DBIE11	DBIE10	DBIE9	DBIE8
R/W−0	R/W−0	R/W−0	R/W−0	R/W−0	R/W−0	R/W−0	R/W−0

7	6	5	4	3	2	1	0
DBIE7	DBIE6	DBIE5	DBIE4	DBIE3	DBIE2	Reserved	
R/W−0	R/W−0	R/W−0	R/W−0	R/W−0	R/W−0	R−0	

Legend: R = Read; W = Write; *-n* = Value after DSP hardware reset

2.8.4.1 RTOSINTD Bit in DBIER1

Bit	Name	Description	Accessibility	HW Reset
10	RTOSINTD	Debug enable bit for the real-time operating system interrupt, RTOSINT	Read/Write	0

The RTOSINTD bit enables or disables RTOSINT as a time-critical interrupt:

RTOSINTD	Description
0	RTOSINT is disabled.
1	RTOSINT is enabled. It is configured as a time-critical interrupt.

2.8.4.2 DLOGINTD Bit in DBIER1

Bit	Name	Description	Accessibility	HW Reset
9	DLOGINTD	Debug enable bit for the data log interrupt, DLOGINT	Read/Write	0

The DLOGINTD bit enables or disables DLOGINT as a time-critical interrupt:

DLOGINTD	Description
0	DLOGINT is disabled.
1	DLOGINT is enabled. It is configured as a time-critical interrupt.

2.8.4.3 BERRINTD Bit in DBIER1

Bit	Name	Description	Accessibility	HW Reset
8	BERRINTD	Debug enable bit for the bus error interrupt, BERRINT	Read/Write	0

The BERRINTD bit enables or disables BERRINT as a time-critical interrupt:

BERRINTD	Description
0	BERRINT is disabled.
1	BERRINT is enabled. It is configured as a time-critical interrupt.

2.8.4.4 DBIE16–DBIE23 Bits in DBIER1

Bit	Name	Description	Accessibility	HW Reset
0	DBIE16	Debug interrupt enable bit 16	Read/Write	0
1	DBIE17	Debug interrupt enable bit 17	Read/Write	0
2	DBIE18	Debug interrupt enable bit 18	Read/Write	0
3	DBIE19	Debug interrupt enable bit 19	Read/Write	0
4	DBIE20	Debug interrupt enable bit 20	Read/Write	0
5	DBIE21	Debug interrupt enable bit 21	Read/Write	0
6	DBIE22	Debug interrupt enable bit 22	Read/Write	0
7	DBIE23	Debug interrupt enable bit 23	Read/Write	0

The functions of these bits can be summarized as follows, where x is a number from 16 to 23:

DBIEx	Description
0	The interrupt associated with interrupt vector x is disabled.
1	The interrupt associated with interrupt vector x is enabled. The interrupt is configured as a time-critical interrupt.

2.8.4.5 DBIE2–DBIE15 Bits in DBIER0

Bit	Name	Description	Accessibility	HW Reset
2	DBIE2	Debug interrupt enable bit 2	Read/Write	0
3	DBIE3	Debug interrupt enable bit 3	Read/Write	0
4	DBIE4	Debug interrupt enable bit 4	Read/Write	0
5	DBIE5	Debug interrupt enable bit 5	Read/Write	0
6	DBIE6	Debug interrupt enable bit 6	Read/Write	0
7	DBIE7	Debug interrupt enable bit 7	Read/Write	0
8	DBIE8	Debug interrupt enable bit 8	Read/Write	0
9	DBIE9	Debug interrupt enable bit 9	Read/Write	0
10	DBIE10	Debug interrupt enable bit 10	Read/Write	0
11	DBIE11	Debug interrupt enable bit 11	Read/Write	0
12	DBIE12	Debug interrupt enable bit 12	Read/Write	0
13	DBIE13	Debug interrupt enable bit 13	Read/Write	0
14	DBIE14	Debug interrupt enable bit 14	Read/Write	0
15	DBIE15	Debug interrupt enable bit 15	Read/Write	0

The functions of these bits can be summarized as follows, where x is a number from 2 to 15:

DBIEx	Description
0	The interrupt associated with interrupt vector x is disabled.
1	The interrupt associated with interrupt vector x is enabled. The interrupt is configured as a time-critical interrupt.

2.9 Registers for Controlling Repeat Loops

This section describes registers that control the execution of repeat loops. Single-repeat registers are used for the repetition of a single instruction. Block-repeat registers are used for the repetition of one or more blocks of instructions.

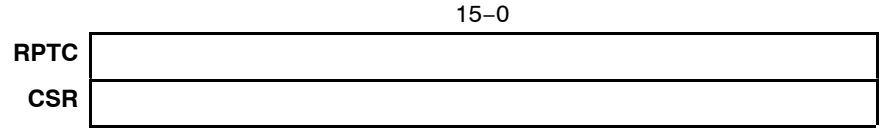
2.9.1 Single-Repeat Registers (RPTC, CSR)

The 16-bit single-repeat instruction registers RPTC and CSR enable repetition of a single-cycle instruction (or two single-cycle instructions that are executed in parallel). The number of repetitions, N, is loaded into the single-repeat counter (RPTC) before the first execution. After the first execution, the instruction is executed N more times; therefore, total execution is N+1 times.

In some syntaxes of the unconditional single-repeat instruction, you can use the computed single-repeat register (CSR) to specify the number N. The value from CSR is copied into RPTC before the first execution of the instruction or instruction pair to be repeated.

As shown in Figure 2−15, RPTC and CSR have 16 bits, enabling up to 65536 consecutive executions of an instruction (the first execution plus 65535 repetitions).

Figure 2−15. Single-Repeat Registers



2.9.2 Block-Repeat Registers (BRC0, BRC1, BRS1, RSA0, RSA1, REA0, REA1)

The block-repeat instructions enable you to form loops that repeat blocks of instructions. You can have one block-repeat loop nested inside another, creating an inner (level 1) loop and an outer (level 0) loop. Table 2−13 describes the C55x registers associated with level 0 and level 1 loops. As described in the following paragraphs, the use of these registers is affected by the C54x-compatible mode bit (C54CM), which is introduced in section 2.10.2.4.

If C54CM = 0: C55x native mode ...
The CPU keeps a record of active repeat loops when an interrupt or call is performed while the loop is active (see the description for CFCT in section 2.7, *Program Flow Registers (PC, RETA, CFCT)*). This enables the use of level 0 resources in subroutines. When the CPU decodes a block-repeat instruction, it first determines whether a loop is already being executed. If the CPU detects an active level 0 loop, it uses the level 1 loop registers; otherwise, it uses the level 0 loop registers.

If C54CM = 1: C54x-compatible mode ...
Block-repeat instructions activate the level 0 loop registers only. Level 1 loop registers are not used. Nested block-repeat operations can be implemented as on the C54x DSPs, using context saving/restoring and the block-repeat active flag (BRAF). A block-repeat instruction sets BRAF, and BRAF is cleared at the end of the block-repeat operation when BRC0 contains 0. For more details about BRAF, see section 2.10.2.2, *BRAF Bit of ST1_55*.

When a block-repeat loop begins in the C54x-compatible mode (C54CM = 1), the BRAF bit is automatically set to indicate that a loop is in progress. If your program must switch modes from C54CM = 1 to C54CM = 0, the BRAF bit must be cleared before or during the switch. There are three options:

- ❏ Wait until the loop is finished (when BRAF is cleared automatically) and then clear C54CM
- ❏ Clear BRAF (this also stops the loop) and then clear C54CM
- ❏ Clear BRAF and C54CM at the same time with an instruction that modifies status register ST1_55

Note:

Make sure the last three instructions of a level 0 loop do not write to BRC0. Likewise, make sure the last three instructions of a level 1 loop do not write to BRC1.

Table 2–13. Block-Repeat Register Descriptions

Level 0 Loop Registers		Level 1 Loop Registers (Not Used If C54CM = 1)	
Register	**Description**	**Register**	**Description**
BRC0	Block-repeat counter 0. This 16-bit register contains the number of times to repeat the instruction block after its initial execution.	BRC1	Block-repeat counter 1. This 16-bit register contains the number of times to repeat the instruction block after its initial execution.
RSA0	Block-repeat start address register 0. This 24-bit register contains the address of the first instruction in the instruction block.	RSA1	Block-repeat start address register 1. This 24-bit register contains the address of the first instruction in the instruction block.
REA0	Block-repeat end address register 0. This 24-bit register contains the address of the last instruction in the instruction block.	REA1	Block-repeat end address register 1. This 24-bit register contains the address of the last instruction in the instruction block.
		BRS1	BRC1 save register. Whenever BRC1 is loaded, BRS1 is loaded with the same value. The content of BRS1 is not modified during the execution of the level 1 loop. Each time the level 1 loop is triggered, BRC1 is reinitialized from BRS1. This feature enables initialization of BRC1 outside of the level 0 loop, reducing the time needed for each repetition.

Note: The 24-bit register values are stored in two consecutive 16-bit locations. Bits 23–16 are stored at the lower address (the eight most significant bits in this location are ignored by the CPU). Bits 15–0 are stored at the higher address. For example, RSA0(23–16) is accessible at address 00 003Ch, and RSA0(15–0) is accessible at address 00 003Dh.

2.10 Status Registers (ST0_55−ST3_55)

These four 16-bit registers (see Figure 2−16) contain control bits and flag bits. The control bits affect the operation of the C55x DSP and the flag bits reflect the current status of the DSP or indicate the results of operations.

ST0_55, ST1_55, and ST3_55 are each accessible at two addresses (see section 2.2, *Memory-Mapped Registers*). At one address, all the TMS320C55x bits are available. At the other address (the protected address), the bits highlighted in Figure 2−16 cannot be modified. The protected address is provided to support TMS320C54x code that was written to access ST0, ST1, and PMST (the C54x counterpart of ST3_55). Reserved bits are not available for use.

Notes:

1) Always write 1100b (Ch) to bits 11−8 of ST3_55.

2) Some C55x devices do not have an instruction cache; these devices do not use the CAFRZ, CAEN, and CACLR bits.

Figure 2–16. Status Registers

ST0_55

15	14	13	12	11	10	9
ACOV2†	**ACOV3**†	**TC1**†	TC2	CARRY	ACOV0	ACOV1
R/W–0	R/W–0	R/W–1	R/W–1	R/W–1	R/W–0	R/W–0

8	7	6	5	4	3	2	1	0
DP[15:7]								
R/W–0								

ST1_55

15	14	13	12	11	10	9	8
BRAF	CPL	XF	HM	INTM	**M40**†	SATD	SXMD
R/W–0	R/W–0	R/W–1	R/W–0	R/W–1	R/W–0	R/W–0	R/W–1

7	6	5	4	3	2	1	0
C16	FRCT	**C54CM**†	ASM				
R/W–0	R/W–0	R/W–1	R/W–0				

ST2_55

15	14	13	12	11	10	9	8
ARMS	Reserved		DBGM	EALLOW	RDM	Reserved	CDPLC
R/W–0	R–11b		R/W–1	R/W–0	R/W–0	R–0	R/W–0

7	6	5	4	3	2	1	0
AR7LC	AR6LC	AR5LC	AR4LC	AR3LC	AR2LC	AR1LC	AR0LC
R/W–0	R/W–0	R/W–0	R/W–0	R/W–0	R/W–0	R/W–0	R/W–0

ST3_55

15	14	13	12	11	10	9	8
CAFRZ†#	**CAEN**†#	**CACLR**†#	**HINT**†‡	Reserved (always write as 1100b)			
R/W–0	R/W–0	R/W–0	R/W–1	R/W–1100b			

7	6	5	4	3	2	1	0
CBERR†	MPNMC§	**SATA**†	Reserved	Reserved	CLKOFF	SMUL	SST
R/W–0	R/W–pins	R/W–0	R/W–0¶	R–0	R/W–0	R/W–0	R/W–0

Legend: R = Read; W = Write; *-n* = Value after DSP hardware reset

† Highlighted bit: If you write to the protected address of the status register, a write to this bit has no effect, and the bit always appears as a 0 during read operations.

‡ The HINT bit is not used for all C55x host port interfaces (HPIs). Consult the documentation for the specific C55x DSP.

§ The reset value of MPNMC may be dependent on the state of predefined pins at reset. To check this for a particular C55x DSP, see its data manual.

¶ Always write 0 to this bit

Some C55x devices do not have an instruction cache; these devices do not use bits CAFRZ, CAEN, and CACLR.

2.10.1 ST0_55 Bits

This section describes the bits of ST0_55 in alphabetical order.

2.10.1.1 ACOV0, ACOV1, ACOV2, and ACOV3 Bits of ST0_55

Each of the four accumulators has its own overflow flag in ST0_55:

Bit	Name	Description	Accessibility	HW Reset
9	ACOV1	AC1 overflow flag	Read/Write	0
10	ACOV0	AC0 overflow flag	Read/Write	0
14	ACOV3	AC3 overflow flag	Read/Write	0
15	ACOV2	AC2 overflow flag	Read/Write	0

For each of these flags:

- Overflow detection depends on the M40 bit in ST1_55:
  - M40 = 0: Overflow is detected at bit position 31.
  - M40 = 1: Overflow is detected at bit position 39.

  If you need compatibility with TMS320C54x code, make sure M40 = 0.
- ACOVx is set when an overflow occurs in ACx, where x is 0, 1, 2, or 3.
- Once an overflow occurs, ACOVx remains set until one of the following events occurs:
  - A DSP hardware or software reset is performed.
  - The CPU executes a conditional branch, call, return, or execute instruction that tests the state of ACOVx.
  - ACOVx is explicitly cleared by a status bit clear instruction. For example, you can clear ACOV1 by using the following instruction:

    ```
    BCLR ACOV1
    ```

    (To set ACOV1, use BSET ACOV1.)

2.10.1.2 CARRY Bit of ST0_55

Bit	Name	Description	Accessibility	HW Reset
11	CARRY	Carry bit	Read/Write	1

The following are main points about the carry bit:

- Carry/borrow detection depends on the M40 bit in ST1_55:
  - M40 = 0: Carry/borrow is detected with respect to bit position 31.
  - M40 = 1: Carry/borrow is detected with respect to bit position 39.

  If you need compatibility with TMS320C54x code, make sure M40 = 0.

- When an addition is performed in the D-unit arithmetic logic unit (D-unit ALU), if the addition generates a carry, CARRY is set; if no carry is generated, CARRY is cleared. There is one exception to this behavior: When the following syntax is used (shifting Smem by 16 bits), CARRY is set for a carry but is not affected if no carry is generated.

  ```
  ADD Smem <<#16, [ACx,] ACy
  ```

- When a subtraction is performed in the D-unit ALU, if the subtraction generates a borrow, CARRY is cleared; if no borrow is generated, CARRY is set. There is one exception to this behavior: When the following syntax is used (shifting Smem by 16 bits), CARRY is cleared for a borrow but is not affected if no borrow is generated.

  ```
  SUB Smem <<#16, [ACx,] ACy
  ```

- CARRY is modified by the logical shift instructions.

- For signed shift instructions and rotate instructions, you can choose whether CARRY is modified.

- The following instruction syntaxes modify CARRY to indicate particular computation results when the destination register (dst) is an accumulator:

MIN [src,] dst	Minimum comparison
MAX [src,] dst	Maximum comparison
ABS [src,] dst	Absolute value
NEG [src,] dst	Negate

- You can clear and set CARRY with the following instructions:

  ```
  BCLR CARRY   ; Clear CARRY
  BSET CARRY   ; Set CARRY
  ```

2.10.1.3 DP Bit Field of ST0_55

Bits	Name	Description	Accessibility	HW Reset
8–0	DP	Copy of the 9 most significant bits of the data page register (DP)	Read/Write	0

This 9-bit field is provided for compatibility with code transferred from the TMS320C54x DSPs. TMS320C55x DSPs have a data page register independent of ST0_55. Any change to bits 15–7 of this data page register—DP(15–7)—is reflected in the DP status bits. Any change to the DP status bits is reflected in DP(15–7). When generating addresses for the DP direct addressing mode, the CPU uses the full data page register, DP(15–0). You are not required to use the DP status bits; you can modify DP directly.

Note:

If you want to load ST0_55 but do not want the access to change the content of the data page register, use an OR or an AND operation with a mask value that does not modify the 9 least significant bits (LSBs) of ST0_55. For an OR operation, put 0s in the 9 LSBs of the mask value. For an AND operation, put 1s in the 9 LSBs of the mask value.

2.10.1.4 TC1 and TC2 Bits of ST0_55

Bit	Name	Description	Accessibility	HW Reset
12	TC2	Test/control flag 2	Read/Write	1
13	TC1	Test/control flag 1	Read/Write	1

The main function of the test/control bit is to hold the result of a test performed by specific instructions. The following are main points about the test/control bits:

- All the instructions that affect a test/control flag allow you to choose whether TC1 or TC2 is affected.
- TCx (where x = 1 or 2) or a Boolean expression of TCx can be used as a trigger in any conditional instruction.
- You can clear and set TC1 and TC2 with the following instructions:

```
BCLR TC1  ; Clear TC1
BSET TC1  ; Set TC1
BCLR TC2  ; Clear TC2
BSET TC2  ; Set TC2
```

2.10.2 ST1_55 Bits

This section describes the bits of ST1_55 in alphabetical order.

2.10.2.1 ASM Bit Field of ST1_55

Bits	Name	Description	Accessibility	HW Reset
4–0	ASM	Accumulator shift mode bits	Read/Write	00000b

ASM is not used by native TMS320C55x instructions but is available to support TMS320C54x code running on the TMS320C55x DSP. In a C54x DSP, the ASM field supplies a signed shift count for special instructions that shift an accumulator value. The C55x ASM field is used in the C54x-compatible mode (C54CM = 1).

Before reading further, it is important to know that the C55x register that contains ASM (status register ST1_55) is accessible at two addresses. One address, 00 0003h, is to be used by native C55x instructions. The other address, 00 0007h, is provided to support C54x code that accesses ST1 at 0007h.

If **C54CM = 1** (C54x-compatible mode):

- Whenever ASM is loaded by a write to address 00 0007h, the 5-bit ASM value is sign-extended to 16 bits and written to temporary register 2 (T2). Clear/set status register bit instructions do not affect this bit field. When a C54x instruction requests the CPU to shift an accumulator according to ASM, the CPU uses the shift count in T2.
- Whenever T2 is loaded, the five least significant bits are copied to ASM.
- Because T2 is tied to ASM, T2 is not available as a general-purpose data register.

If **C54CM = 0**:

- ASM is ignored. During an accumulator shift operation, the CPU reads the shift count from the temporary register (T0, T1, T2, or T3) that was specified in the C55x instruction, or from a constant embedded in the C55x instruction.
- T2 can be used as a general-purpose data register. Writing to address 00 0007h does not affect T2, and writing to T2 does not affect ASM.

2.10.2.2 BRAF Bit of ST1_55

Bit	Name	Description	Accessibility	Reset Value
15	BRAF	Block-repeat active flag	Read/Write	0

If **C54CM = 0**: BRAF is not used. The status of repeat operations is maintained automatically by the CPU (see the description for CFCT in section 2.7, *Program Flow Registers (PC, RETA, CFCT)).*

If **C54CM = 1**: Reading BRAF indicates the status of a block-repeat operation:

BRAF	Block-Repeat Activity
0	No block-repeat operation is active.
1	A block-repeat operation is active.

To stop an active block-repeat operation in the C54x-compatible mode, you can clear BRAF with the following instruction:

```
BCLR BRAF    ; Clear BRAF
```

You can set BRAF with the following instruction:

```
BSET BRAF    ; Set BRAF
```

BRAF also can be set or cleared with an instruction that modifies ST1_55.

Functionality of BRAF. A block-repeat loop begins with a block-repeat instruction such as RPTB. BRAF is set at the address phase of this block-repeat instruction to indicate that a loop is active.

Each time the last instruction of the loop enters the decode phase of the pipeline, the CPU checks the values of BRAF and the counter register (BRC0). If BRAF = 1 and BRC0 > 0, the CPU decrements BRC0 by 1 and begins the next iteration of the loop. Otherwise, the CPU stops the loop. (In either case, the last instruction completes its passage through the pipeline.)

BRAF is cleared in the following cases:

- The last instruction of the loop enters the decode phase, and BRC0 is decremented to 0. BRAF is automatically cleared one cycle later.
- An instruction writes 0 to the block-repeat counter, BRC0. BRAF is automatically cleared one cycle later.
- A far branch (FB) or far call (FCALL) instruction is executed. (BRAF is *not* cleared by the execution of other call or branch instructions, or by the execution of an INTR or TRAP instruction.)
- BRAF is manually cleared by a BCLR BRAF instruction or an instruction that modifies status register ST1_55.

BRAF is saved and restored with ST1_55 during the context switches caused by an interrupt and a return-from-interrupt instruction. BRAF is not saved when the CPU responds to a call instruction.

If a block-repeat loop is in progress and your program must switch modes from C54CM = 1 to C54CM = 0, the BRAF bit must be cleared before or during the switch. There are three options:

- ❏ Wait until the loop is finished (when BRAF is cleared automatically) and then clear C54CM.
- ❏ Clear BRAF (this also stops the loop) and then clear C54CM.
- ❏ Clear BRAF and C54CM at the same time with an instruction that modifies ST1_55.

Pipeline considerations. As already mentioned, the CPU clears BRAF one cycle after executing an instruction that clears BRC0. This modification of BRAF is not pipeline-protected. To ensure that BRAF is modified before another instruction reads BRAF, you may need to insert instructions between the instruction that clears BRC0 and the instruction that reads BRAF. For example:

```
MOV #0, mmap(BRC0)     ; Clear BRC0.
NOP                    ; Wait for BRAF to be cleared.
NOP
NOP
MOV mmap(ST1_55),  AR0    ; Read ST1_55 (including BRAF).
```

The number of instructions to insert depends on when the first instruction clears BRC0:

Pipeline Phase When BRC0 Is Cleared†	Instructions to Insert
Address (AD) phase	0
Execute (X) phase	2
Write (W) phase	3

† Consult the instruction set documentation for the active pipeline phase of a given syntax.

This pipeline issue can also affect when the loop ends. To ensure that BRAF is modified before the last instruction of the loop reaches the decode phase, you must insert 5 or 6 cycles between the instruction that clears BRAF and the last instruction:

Pipeline Phase When BRAF Is Modified†	Instructions to Insert
Execute (X) phase	5
Write (W) phase	6

† Consult the instruction set documentation for the active pipeline phase of a given syntax.

Updating BRAF prior to a return instruction (RET or RETI) is protected in the pipeline. After the return, if the next instruction reads BRAF, it reads the updated value.

2.10.2.3 C16 Bit of ST1_55

Bit	Name	Description	Accessibility	HW Reset
7	C16	Dual 16-bit arithmetic mode bit	Read/Write	0

In the TMS320C54x-compatible mode (C54CM = 1), BRAF indicates/controls the status of a block-repeat operation. In this mode, the execution of some instructions is affected by C16. C16 determines whether such an instruction is executed in a single 32-bit operation (double-precision arithmetic) or in two parallel 16-bit operations (dual 16-bit arithmetic).

If **C54CM = 1**: The arithmetic performed in the D-unit ALU depends on C16:

C16	Dual 16-Bit Mode Is ...
0	Off. For an instruction that is affected by C16, the D-unit ALU performs one 32-bit operation.
1	On. For an instruction that is affected by C16, the D-unit ALU performs two 16-bit operations in parallel.

If **C54CM = 0**: The CPU ignores C16. The instruction syntax alone determines whether dual 16-bit arithmetic or 32-bit arithmetic is used.

You can clear and set C16 with the following instructions:

```
BCLR C16     ; Clear C16
BSET C16     ; Set C16
```

2.10.2.4 C54CM Bit of ST1_55

Bit	Name	Description	Accessibility	HW Reset
5	C54CM	TMS320C54x-compatible mode bit	Read/Write	1

The C54CM bit determines whether the CPU will support code that was developed for a TMS320C54x DSP:

C54CM	C54x-Compatible Mode Is ...
0	Disabled. The CPU supports code written for a TMS320C55x (C55x) DSP.
1	Enabled. This mode must be set when you are using code that was originally developed for a TMS320C54x (C54x) DSP. All the C55x CPU resources remain available; therefore, as you translate code, you can take advantage of the additional features on the C55x to optimize your code.

Change modes with the following instructions and assembler directives:

```
BCLR C54CM      ; Clear C54CM (happens at run time)
.C54CM_off      ; Tell assembler C54CM = 0

BSET C54CM      ; Set C54CM (happens at run time)
.C54CM_on       ; Tell the assembler C54CM = 1
```

Do not modify C54CM within a block-repeat loop as shown in this example:

```
      RPBTLOCAL end1   ; Start of loop 1
         :
      BSET C54CM
         :
end1  MOV AC0, dbl(*AR3+) ; End of loop 1
```

Also, do not modify C54CM in parallel with a block-repeat instruction such as:

```
      BCLR C54CM || RPTB end2   ; Start of loop 2
         :
end2  MOV AC1, dbl(*AR4+)       ; End of loop 2
```

2.10.2.5 CPL Bit of ST1_55

Bit	Name	Description	Accessibility	HW Reset
14	CPL	Compiler mode bit	Read/Write	0

The CPL bit determines which of two direct addressing modes is active:

CPL	Direct Addressing Mode Selected
0	DP direct addressing mode. Direct accesses to data space are made relative to the data page register (DP).
1	SP direct addressing mode. Direct accesses to data space are made relative to the data stack pointer (SP). The DSP is said to be in compiler mode.

Note: Direct addresses to I/O space are always made relative to the peripheral data page register (PDP).

Change modes with the following instructions and assembler directives:

```
BCLR CPL     ; Clear CPL (happens at run time)
.CPL_off     ; Tell assembler CPL = 0

BSET CPL     ; Set CPL (happens at run time)
.CPL_on      ; Tell the assembler CPL = 1
```

2.10.2.6 FRCT Bit of ST1_55

Bit	Name	Description	Accessibility	HW Reset
6	FRCT	Fractional mode bit	Read/Write	0

The FRCT bit turns the fractional mode on or off:

FRCT	Fractional Mode Is ...
0	Off. Results of multiply operations are not shifted.
1	On. Results of multiply operations are shifted left by 1 bit for decimal point adjustment. This is required when you multiply two signed Q15 values and you need a Q31 result.

You can clear and set FRCT with the following instructions:

```
BCLR FRCT    ; Clear FRCT
BSET FRCT    ; Set FRCT
```

2.10.2.7 HM Bit of ST1_55

Bit	Name	Description	Accessibility	HW Reset
12	HM	Hold mode bit	Read/Write	0

When the external memory interface (EMIF) of the DSP receives a HOLD request, the DSP places the EMIF output pins in the high-impedance state. Depending on HM, the DSP may also stop internal program execution:

HM	Hold Mode
0	The DSP continues executing instructions from internal program memory.
1	The DSP stops executing instructions from internal program memory.

You can use the following instructions to clear and set HM:

```
BCLR HM      ; Clear HM
BSET HM      ; Set HM
```

2.10.2.8 INTM Bit of ST1_55

Bit	Name	Description	Accessibility	HW Reset
11	INTM	Interrupt mode bit	Read/Write	1

The INTM bit globally enables or disables the maskable interrupts as shown in the following table. This bit has no effect on nonmaskable interrupts (those that cannot be blocked by software).

INTM	Description
0	All unmasked interrupts are enabled.
1	All maskable interrupts are disabled.

The following are main points about the INTM bit:

- Modify the INTM bit with status bit clear and set instructions:

```
BCLR INTM    ; Clear INTM
BSET INTM    ; Set INTM
```

  The only other instructions that affect INTM are the software interrupt instruction and the software reset instruction, which set INTM before branching to the interrupt service routine.

  In CPU cores with revisions older than 2.2, there is no pipeline protection by the hardware between the INTM bit update and an interrupt jamming. Because the INTM bit is updated in the execute phase of the pipeline, an interrupt can be taken in between any of the 5 instructions following the INTM set instruction which globally disables interrupts. In CPU cores with revisions 2.2 or newer, no interrupt is taken after the INTM set instruction.

- The state of the INTM bit is automatically saved when the CPU approves an interrupt request. Specifically, the INTM bit is saved when the CPU saves ST1_55 to the data stack.

- Before executing an interrupt service routine (ISR) triggered by an INTR #5 instruction, by the RESET instruction, or by a hardware interrupt source, the CPU automatically sets the INTM bit to globally disable the maskable interrupts. The TRAP #k5 instruction does not affect the INTM bit. The ISR can re-enable the maskable interrupts by clearing the INTM bit.

- A return-from-interrupt instruction restores the INTM bit from the data stack.

- When the CPU is halted in the real-time emulation mode of the debugger, INTM is ignored and only time-critical interrupts can be serviced (see the description for the debug interrupt enable registers in section 2.8.4).

2.10.2.9 M40 Bit of ST1_55

Bit	Name	Description	Accessibility	HW Reset
10	M40	Computation mode bit for the D unit	Read/Write	0

The M40 bit selects one of two computation modes for the D unit:

M40 D-Unit Computation Mode Is ...

0 32-bit mode. In this mode:

- ❑ The sign bit is extracted from bit position 31.
- ❑ During arithmetic, the carry is determined with respect to bit position 31.
- ❑ Overflows are detected at bit position 31.
- ❑ During saturation, the saturation value is 00 7FFF FFFFh (positive overflow) or FF 8000 0000h (negative overflow).
- ❑ Accumulator comparisons versus 0 are done using bits 31–0.
- ❑ Shift or rotate operations are performed on 32-bit values.
- ❑ During left shifts or rotations of accumulators, bits shifted out are extracted from bit position 31.
- ❑ During right shifts or rotations of accumulators, bits shifted in are inserted at bit position 31.
- ❑ During signed shifts of accumulators, if SXMD = 0, 0 is copied into the accumulator's guard bits; if SXMD = 1, bit 31 is copied into the accumulator's guard bits. During any rotations or logical shifts of accumulators, the guard bits of the destination accumulator are cleared.

Note: In the TMS320C54x-compatible mode (C54CM = 1), there are some exceptions: An accumulator's sign bit is extracted from bit position 39. Accumulator comparisons versus 0 are done using bits 39–0. Signed shifts are performed as if M40 = 1.

1 40-bit mode. In this mode:

- ❑ The sign bit is extracted from bit position 39.
- ❑ During arithmetic, the carry is determined with respect to bit position 39.
- ❑ Overflows are detected at bit position 39.
- ❑ During saturation, the saturation value is 7F FFFF FFFFh (positive overflow) or 80 0000 0000h (negative overflow).
- ❑ Accumulator comparisons versus 0 are done using bits 39–0.
- ❑ Shift or rotate operations are performed on 40-bit values.
- ❑ During left shifts or rotations of accumulators, bits shifted out are extracted from bit position 39.
- ❑ During right shifts or rotations of accumulators, bits shifted in are inserted at bit position 39.

You can clear and set M40 with the following instructions:

```
BCLR M40     ; Clear M40
BSET M40     ; Set M40
```

2.10.2.10 SATD Bit of ST1_55

Bit	Name	Description	Accessibility	HW Reset
9	SATD	Saturation mode bit for the D unit	Read/Write	0

The SATD bit determines whether the CPU saturates overflow results in the D unit:

SATD	Saturation Mode in the D Unit Is ...
0	Off. No saturation is performed.
1	On. If an operation performed by the D unit results in an overflow, the result is saturated. The saturation depends on the value of the M40 bit: M40 = 0 The CPU saturates the result to 00 7FFF FFFFh (positive overflow) or FF 8000 0000h (negative overflow). M40 = 1 The CPU saturates the result to 7F FFFF FFFFh (positive overflow) or 80 0000 0000h (negative overflow).

If you want compatibility with TMS320C54x code, make sure M40 = 0.

You can clear and set SATD with the following instructions:

```
BCLR SATD    ; Clear SATD
BSET SATD    ; Set SATD
```

2.10.2.11 SXMD Bit of ST1_55

Bit	Name	Description	Accessibility	HW Reset
8	SXMD	Sign-extension mode bit for the D unit	Read/Write	1

The SXMD bit turns on or off the sign-extension mode, which affects accumulator loads, and also affects additions, subtractions, and signed shift operations that are performed in the D unit. A summary of the effects of SXMD follows.

SXMD **Sign-Extension Mode Is ...**

0 Off. When sign-extension mode is off:

- For 40-bit operations, 16-bit or smaller operands are zero extended to 40 bits.
- For the conditional subtract instruction, any 16-bit divisor produces the expected result.
- When the D-unit arithmetic logic unit (ALU) is locally configured in its dual 16-bit mode (by a dual 16-bit arithmetic instruction):
  - 16-bit values used in the higher part of the D-unit ALU are zero extended to 24 bits.
  - 16-bit accumulator halves are zero extended if they are shifted right.
- During a signed shift of an accumulator, if it is a 32-bit operation (M40 = 0), 0 is copied into the accumulator's guard bits (39–32).
- During a signed right shift of an accumulator, the shifted value is zero extended.

1 On. In this mode:

- For 40-bit operations, 16-bit or smaller operands are sign extended to 40 bits.
- For the conditional subtract instruction, the 16-bit divisor must be a positive value (its most significant bit (MSB) must be 0).
- When the D-unit ALU is locally configured in its dual 16-bit mode (by a dual 16-bit arithmetic instruction):
  - 16-bit values used in the higher part of the D-unit ALU are sign extended to 24 bits.
  - 16-bit accumulator halves are sign extended if they are shifted right.
- During a signed shift of an accumulator, if it is a 32-bit operation (M40 = 0), bit 31 is copied into the accumulator's guard bits (39–32).
- During a signed right shift of an accumulator, the shifted value is sign extended, unless the uns() expression qualifier is used to designate the accumulator value as unsigned.

SXMD is ignored during some operations:

- For unsigned operations (boolean logic operations, rotate operations, and logical shift operations), input operands are always zero extended to 40 bits, regardless of the value of SXMD.
- For operations performed in a multiply-and-accumulate unit (MAC), 16-bit input operands are sign extended to 17 bits, regardless of the value of SXMD.
- If an operand in an instruction is enclosed in the operand qualifier uns(), the operand is treated as unsigned, regardless of the value of SXMD.

You can clear and set SXMD with the following instructions:

```
BCLR SXMD    ; Clear SXMD
BSET SXMD    ; Set SXMD
```

2.10.2.12 XF Bit of ST1_55

Bit	Name	Description	Accessibility	HW Reset
13	XF	External flag	Read/Write	1

The XF bit is a general-purpose output bit. This bit is directly connected to the XF pin on those C55x devices that have an XF pin. Setting the XF bit drives the XF pin high. Clearing the XF bit drives the XF pin low. The following instructions clear and set XF:

```
BCLR XF      ; Clear XF
BSET XF      ; Set XF
```

2.10.3 ST2_55 Bits

This section describes the bits of ST2_55 in alphabetical order.

2.10.3.1 AR0LC–AR7LC Bits of ST2_55

The CPU has eight auxiliary registers, AR0–AR7. Each auxiliary register ARn (n = 0, 1, 2, 3, 4, 5, 6, or 7) has its own linear/circular configuration bit in ST2_55:

Bit	Name	Description	Accessibility	HW Reset
n	ARnLC	ARn linear/circular configuration bit	Read/Write	0

Each ARnLC bit determines whether ARn is used for linear addressing or circular addressing:

ARnLC	ARn Is Used For ...
0	Linear addressing
1	Circular addressing

For example, if AR3LC = 0, AR3 is used for linear addressing; if AR3LC = 1, AR3 is used for circular addressing.

You can clear and set the ARnLC bits with the status bit set/clear instruction. For example, the following instructions respectively clear and set AR3LC. To modify other ARnLC bits, replace the 3 with the appropriate number.

```
BCLR AR3LC      ; Clear AR3LC
BSET AR3LC      ; Set AR3LC
```

2.10.3.2 ARMS Bit of ST2_55

Bit	Name	Description	Accessibility	HW Reset
15	ARMS	AR mode switch	Read/Write	0

The ARMS bit determines the CPU mode used for the AR indirect addressing mode:

ARMS	AR Indirect Operands Available
0	DSP mode operands, which provide efficient execution of DSP intensive applications. Among these operands are those that use reverse carry propagation when adding to or subtracting from a pointer. Short-offset operands are not available.
1	Control mode operands, which enable optimized code size for control system applications. The short-offset operand *ARn(short(#k3)) is available. (Other offsets require a 2-byte extension on an instruction, and instructions with these extensions cannot be executed in parallel with other instructions.)

Change modes with the following instructions and assembler directives:

```
BCLR ARMS     ; Clear ARMS(happens at run time)
.ARMS_off     ; Tell assembler ARMS = 0

BSET ARMS     ; Set ARMS (happens at run time)
.ARMS_on      ; Tell assembler ARMS = 1
```

2.10.3.3 CDPLC Bit of ST2_55

Bit	Name	Description	Accessibility	HW Reset
8	CDPLC	CDP linear/circular configuration bit	Read/Write	0

The CDPLC bit determines whether the coefficient data pointer (CDP) is used for linear addressing or circular addressing:

CDPLC	CDP Is Used For ...
0	Linear addressing
1	Circular addressing

You can clear and set CDPLC with the following instructions:

```
BCLR CDPLC       ; Clear CDPLC
BSET CDPLC       ; Set CDPLC
```

2.10.3.4 DBGM Bit of ST2_55

Bit	Name	Description	Accessibility	HW Reset
12	DBGM	Debug mode bit	Read/Write	1

The DBGM bit provides the capability to block debug events during time-critical portions of a program:

DBGM	Debug Events Are ...
0	Enabled.
1	Disabled. The emulator cannot access memory or registers. Software breakpoints still cause the CPU to halt, but hardware breakpoints or halt requests are ignored.

The following are main points about the DBGM bit:

- For pipeline protection, the DBGM bit can only be modified by status bit clear and set instructions:

```
BCLR DBGM    ; Clear DBGM
BSET DBGM    ; Set DBGM
```

  Writes to ST2_55 do not affect DBGM.

- The state of the DBGM bit is automatically saved when the CPU approves an interrupt request or fetches the INTR #k5, TRAP #k5, or RESET instruction. Specifically, the DBGM bit is saved when the CPU saves ST2_55 to the data stack.

- Before executing an interrupt service routine (ISR) triggered by the INTR #k5, TRAP #k5, or RESET instruction, or by a hardware interrupt source, the CPU automatically sets the DBGM bit to disable debug events. The ISR can reenable debug events by clearing the DBGM bit.

- A return-from-interrupt instruction restores the DBGM bit from the data stack.

2.10.3.5 EALLOW Bit of ST2_55

Bit	Name	Description	Accessibility	HW Reset
11	EALLOW	Emulation access enable bit	Read/Write	0

The EALLOW bit enables or disables write access to non-CPU emulation registers:

EALLOW	Write Access To Non-CPU Emulation Registers Is ...
0	Disabled
1	Enabled

The following are main points about the EALLOW bit:

- The state of the EALLOW bit is automatically saved when the CPU approves an interrupt request or fetches the INTR #k5, TRAP #k5, or RESET instruction. Specifically, the EALLOW bit is saved when the CPU saves ST2_55 to the data stack.

- Before executing an interrupt service routine (ISR) triggered by the INTR #k5, TRAP #k5, or RESET instruction, or by a hardware interrupt source, the CPU automatically clears the EALLOW bit to prevent accesses to the emulation registers. The ISR can re-enable access by setting the EALLOW bit:

  ```
  BSET EALLOW
  ```

  (To clear EALLOW, you can use BCLR EALLOW.)

- A return-from-interrupt instruction restores the EALLOW bit from the data stack.

2.10.3.6 RDM Bit of ST2_55

Bit	Name	Description	Accessibility	HW Reset
10	RDM	Rounding mode bit	Read/Write	0

Certain instructions executed in the D unit allow you to indicate whether an operand is to be rounded. The type of rounding performed depends on the value of the RDM bit:

RDM	Rounding Mode Selected
0	Round to the infinite. The CPU adds 8000h (2 raised to the 15th power) to the 40-bit operand. Then the CPU clears bits 15 through 0 to generate a rounded result in a 24- or 16-bit representation. For a 24-bit representation, only bits 39 through 16 of the result are meaningful. For a 16-bit representation, only bits 31 through 16 of the result are meaningful.

1	Round to the nearest. The rounding depends on bits 15 through 0 of the 40-bit operand, as shown by the following if statements. The rounded result is in a 24-bit representation (in bits 39 through 16) or a 16-bit representation (in bits 31 through 16).
	If (0 =< bits 15–0 < 8000h)
	CPU clears bits 15–0
	If (8000h < bits 15–0 < 10000h)
	CPU adds 8000h to the operand and then clears bits 15–0
	If (bits 15–0 == 8000h)
	If bits 31–16 contain an odd value
	CPU adds 8000h to the operand and then clears bits 15–0

If you need compatibility with TMS320C54x code, make sure RDM = 0 and C54CM = 1. When C54CM = 1 (C54x-compatible mode enabled), the following instructions do not clear bits 15–0 of the result after the rounding:

SATR [ACx,] ACy	Saturate with rounding
RND [ACx,] ACy	Round
LMS Xmem,Ymem,ACx,ACy	Least mean square

You can clear and set RDM with the following instructions:

```
BCLR RDM     ; Clear RDM
BSET RDM     ; Set RDM
```

2.10.4 ST3_55 Bits

This section describes the bits of ST3_55 in alphabetical order.

2.10.4.1 CACLR Bit of ST3_55

Bit	Name	Description	Accessibility	HW Reset
13	CACLR	Cache clear bit	Read/Write	0

To clear (or flush) the instruction cache (invalidate all the lines of its data arrays), set the CACLR bit. You can set CACLR using the following instruction:

```
BSET CACLR      ; Set CACLR
```

Once set, CACLR remains 1 until the flush process is complete, at which time CACLR is automatically reset to 0. Therefore, you can poll CACLR to get the status:

CACLR	The Flush Process Is...
0	Complete
1	Not complete. All cache blocks are invalid. The number of cycles required to flush the cache depends on the memory architecture. When the cache is flushed, the content of the prefetch queue in the instruction buffer unit is automatically flushed.

2.10.4.2 CAEN Bit of ST3_55

Bit	Name	Description	Accessibility	HW Reset
14	CAEN	Cache enable bit	Read/Write	0

The CAEN bit enables or disables the program cache:

CAEN	Cache Is ...
0	Disabled. The cache controller never receives a program request. All program requests are handled either by the internal memory or the external memory, depending on the address decoded.
1	Enabled. Program code is fetched from the cache, from the internal memory, or from the external memory, depending on the address decoded.

Some important notes:

- When the cache is disabled by clearing the CAEN bit, the content of the instruction buffer queue in the I unit is automatically flushed.
- You can clear and set CAEN using the following instructions

```
BCLR CAEN    ; Clear CAEN
BSET CAEN    ; Set CAEN
```

2.10.4.3 CAFRZ Bit of ST3_55

Bit	Name	Description	Accessibility	HW Reset
15	CAFRZ	Cache freeze bit	Read/Write	0

CAFRZ enables you to lock the instruction cache, so that its contents are not updated on a cache miss but are still available for cache hits. The contents of the cache remain undisturbed until CAFRZ is cleared. The role of CAFRZ is summarized as follows:

CAFRZ	Description
0	The cache is in its default operating mode.
1	The cache is frozen (the cache content is locked).

You can clear and set CAFRZ using the following instructions:

```
BCLR CAFRZ      ; Clear CAFRZ
BSET CAFRZ      ; Set CAFRZ
```

2.10.4.4 CBERR Bit of ST3_55

Bit	Name	Description	Accessibility	HW Reset
7	CBERR	CPU bus error flag	Read/Write (Can only write 0)	0

The CBERR bit is set when an internal bus error is detected. This error causes the CPU to set the bus error interrupt flag (BERRINTF) in interrupt flag register 1 (IFR1). Some important points follow:

- ❏ Writing a 1 to the CBERR bit has no effect. This bit is 1 only if an internal bus error has occurred.
- ❏ The interrupt service routine for the bus error interrupt (BERRINT) must clear the CBERR bit before it returns control to the interrupted program code:

```
BCLR CBERR      ; Clear CBERR
```

The CBERR bit can be summarized as follows:

CBERR	Description
0	The flag has been cleared by your program or by a reset.
1	An internal bus error has been detected.

Note:

When a bus error occurs, the functionality of the instruction that caused the error, and of any instruction executed in parallel, can not be assured.

2.10.4.5 CLKOFF Bit of ST3_55

Bit	Name	Description	Accessibility	HW Reset
2	CLKOFF	CLKOUT disable bit	Read/Write	0

When CLKOFF = 0, the CLKOUT pin is enabled; the associated clock signal appears on the pin. When CLKOFF = 1, the CLKOUT pin is disabled.

You can clear and set CLKOFF with the following instructions:

```
BCLR CLKOFF     ; Clear CLKOFF
BSET CLKOFF     ; Set CLKOFF
```

2.10.4.6 HINT Bit of ST3_55

Bit	Name	Description	Accessibility	HW Reset
12	HINT	Host interrupt bit	Read/Write	1

Use the HINT bit to send an interrupt request to a host processor by way of the host port interface. You produce an active-low interrupt pulse by clearing and then setting the HINT bit:

```
BCLR HINT    ; Clear HINT
BSET HINT    ; Set HINT
```

Note:

The HINT bit is not used for all C55x host port interfaces (HPIs). Consult the documentation for the specific C55x DSP.

2.10.4.7 MPNMC Bit of ST3_55

Bit	Name	Description	Accessibility	HW Reset
6	MPNMC	Microprocessor/ microcomputer mode bit	Read/Write	May be dependent on the state of predefined pins at reset. To check this for a particular C55x DSP, see its data manual.

The MPNMC bit enables or disables the on-chip ROM:

MPNMC	Mode
0	Microcomputer mode. The on-chip ROM is enabled; it is addressable in program space.
1	Microprocessor mode. The on-chip ROM is disabled; it is not in the program-space map.

Some important notes:

- The reset value of the MPNMC bit may be dependent on the state of predefined pins at reset. To check this for a particular C55x DSP, see its data manual.
- The software reset instruction does not affect the MPNMC bit.
- You can clear and set MPNMC using the following instructions:

```
BCLR MPNMC      ; Clear MPNMC
BSET MPNMC      ; Set MPNMC
```

- An instruction that changes the MPNMC bit must not be followed too closely by a branch instruction. Otherwise, the CPU may use the old MPNMC value and, as a result, fetch the next instruction from the incorrect memory location. The minimum number of instruction cycles needed to separate an MPNMC-update instruction and a branch instruction depends on the type of branch instruction used. Table 2–14 divides branch instructions into three categories, and Table 2–15 shows the minimum number of separation cycles needed for each category.

Table 2–14. Categories of Branch Instructions

Category I	Category II	Category III
B L7	RET (when slow return selected)	B ACx
B L16	RETCC (when slow return selected)	CALL ACx
B P24	RETI (when slow return selected)	
BCC I4, cond		
BCC L8, cond		
BCC L16, cond		
BCC P24, cond		
CALL L16		
CALL P24		
CALLCC L16, cond		
CALLCC P24, cond		
RET (when fast return selected)		
RETCC (when fast return selected)		
RETI (when fast return selected)		

Table 2–15. Minimum Number of Instruction Cycles Required Between an MPNMC-Update Instruction and a Branch Instruction

	Cycles Required Before Subsequent Branch Instruction		
MPNMC-Update Instruction	**Category I**	**Category II**	**Category III**
One of the following instructions: BSET MPNMC BSET k4, ST3_55 BCLR MPNMC BCLR k4, ST3_55	4	0	0
An instruction that changes MPNMC when writing to the memory-mapped address for ST3_55	5	1	0

Consider the following example in which the BSET instruction changes MPNMC. Table 2–14 specifies CALL as a category I branch instruction. Table 2–15 indicates that 4 cycles are needed between the BSET MPNMC instruction and a category I branch instruction. In this example, the 4 cycles are provided by inserting four NOP (no operation) instructions. Other instructions could be placed here instead.

```
BSET  MPNMC
NOP
NOP
NOP
NOP
CALL #SubroutineA
```

2.10.4.8 SATA Bit of ST3_55

Bit	Name	Description	Accessibility	HW Reset
5	SATA	Saturation mode bit for the A unit	Read/Write	0

The SATA bit determines whether the CPU saturates overflow results of the A-unit arithmetic logic unit (A-unit ALU):

SATA	Saturation Mode in the A Unit Is ...
0	Off. No saturation is performed.
1	On. If a calculation in the A-unit ALU results in an overflow, the result is saturated to 7FFFh (for overflow in the positive direction) or 8000h (for overflow in the negative direction).

You can clear and set SATA with the following instructions:

```
BCLR SATA    ; Clear SATA
BSET SATA    ; Set SATA
```

2.10.4.9 SMUL Bit of ST3_55

Bit	Name	Description	Accessibility	HW Reset
1	SMUL	Saturation-on-multiplication mode bit	Read/Write	0

The SMUL bit turns the saturation-on-multiplication mode on or off:

SMUL	Saturation-On-Multiplication Mode Is ...
0	Off
1	On. When SMUL = 1, FRCT = 1, and SATD = 1, the result of 18000h × 18000h is saturated to 7FFF FFFFh (regardless of the value of the M40 bit). This forces the product of the two negative numbers to be a positive number.

For multiply-and-accumulate/subtract instructions, the saturation is performed after the multiplication and before the addition/subtraction.

You can clear and set SMUL with the following instructions:

```
BCLR SMUL    ; Clear SMUL
BSET SMUL    ; Set SMUL
```

2.10.4.10 SST Bit of ST3_55

Bit	Name	Description	Accessibility	Reset Value
0	SST	Saturate-on-store mode bit	Read/Write	0

In the TMS320C54x-compatible mode (C54CM = 1), the execution of some accumulator-store instructions is affected by SST. When SST is 1, the 40-bit accumulator value is saturated to a 32-bit value before the store operation. If the accumulator value is shifted, the CPU performs the saturation after the shift.

If **C54CM = 1**: SST turns the saturation-on-store mode on or off.

SST	Saturation-On-Store Mode Is ...
0	Off
1	On. For an instruction that is affected by SST, the CPU saturates a shifted or unshifted accumulator value before storing it. The saturation depends on the value of the sign-extension mode bit (SXMD): SXMD = 0 The 40-bit value is treated as unsigned. If the 40-bit value is greater than 00 7FFF FFFFh, the CPU produces the 32-bit result 7FFF FFFFh. SXMD = 1 The 40-bit value is treated as signed. If the 40-bit value is less than 00 8000 0000h, the CPU produces the 32-bit result 8000 0000h. If the 40-bit value is greater than 00 7FFF FFFFh, the CPU produces 7FFF FFFFh.

If **C54CM = 0**: The CPU ignores SST. The instruction syntax alone determines whether saturation occurs.

You can clear and set SST with the following instructions:

```
BCLR SST     ; Clear SST
BSET SST     ; Set SST
```

Chapter 3

Memory and I/O Space

The C55x DSP provides access to a unified data/program space and an I/O space. Data-space addresses are used to access general-purpose memory and to access the memory-mapped CPU registers. Program-space addresses are used by the CPU to read instructions from memory. I/O space is available for two-way communication with peripherals. An on-chip boot loader provides ways to help load code and data into internal memory.

Topic **Page**

3.1 Memory Map

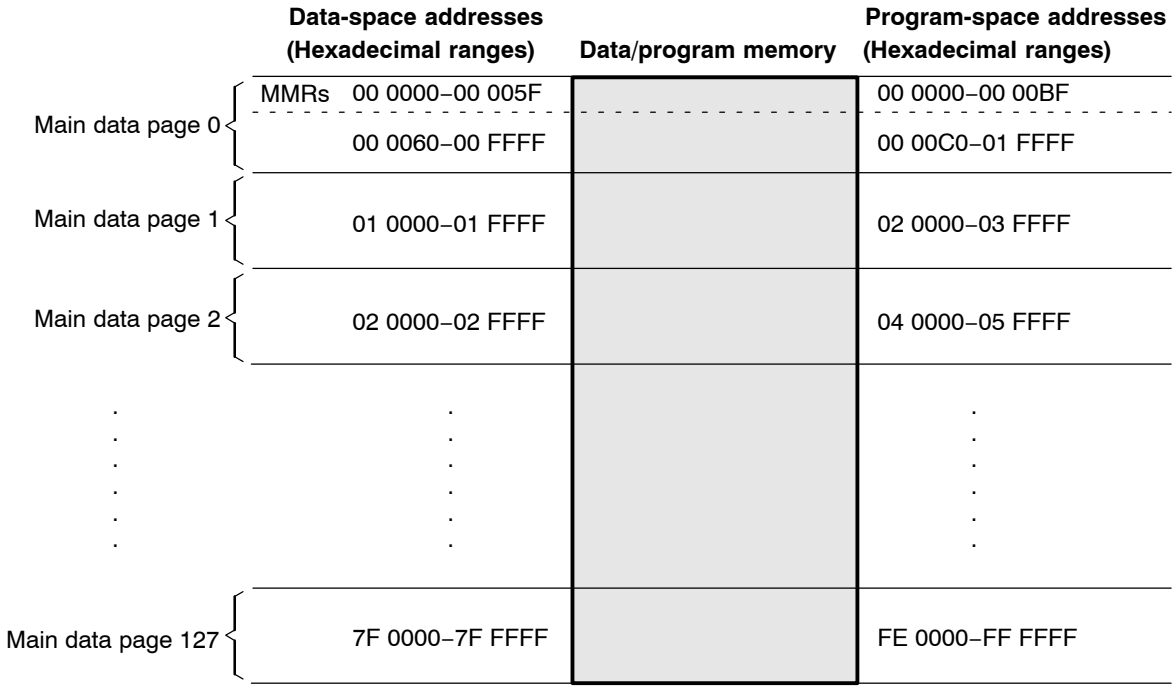
All 16M bytes of memory are addressable as program space or data space (see Figure 3–1). When the CPU uses program space to read program code from memory, it uses 24-bit addresses to reference bytes. When your program accesses data space, it uses 23-bit addresses to reference 16-bit words. In both cases, the address buses carry 24-bit values, but during a data-space access, the least significant bit on the address bus is forced to 0.

Data space is divided into 128 main data pages (0 through 127). Each main data page has 64K addresses. An instruction that references a main data page concatenates a 7-bit main data page value with a 16-bit offset.

On data page 0, the first 96 addresses (00 0000h–00 005Fh) are reserved for the memory-mapped registers (MMRs). There is a corresponding block of 192 addresses (00 0000h–00 00BFh) in program space. It is recommended that you do not store program code to these addresses.

To determine how the addresses are divided between internal memory and external memory, and for the details regarding internal memory, see the data manual for your C55x DSP.

Figure 3–1. Memory Map



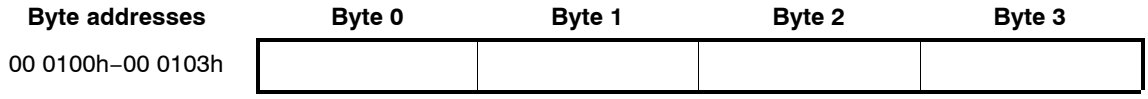
3.2 Program Space

The CPU accesses program space only when reading instructions from program memory. The CPU uses byte addresses (see section 3.2.1) to fetch instructions of varying sizes (see section 3.2.2). Instruction fetches are aligned to even-address 32-bit boundaries (see section 3.2.3).

3.2.1 Byte Addresses (24 Bits)

When the CPU fetches instructions from program memory, it uses **byte addresses**, which are addresses assigned to individual bytes. These addresses are 24 bits wide. Figure 3–2 shows a row of 32-bit-wide memory. Each byte is assigned an address. For example, byte 0 is at address 00 0100h and byte 2 is at address 00 0102h.

Figure 3–2. Example of Byte Addresses for 32-Bit-Wide Program Memory



3.2.2 Instruction Organization in Program Space

The DSP supports 8-, 16-, 24-, 32-, 40-, and 48-bit instructions. Figure 3–3 provides an example of how instructions are organized in program space. Five instructions of varying sizes have been stored in 32-bit-wide memory. The address for each instruction is the address of its most significant byte (the opcode). No code is stored in the shaded bytes.

Figure 3–3. Example of Instruction Organization in Program Space

Instruction	Size	Address
A	24 bits	00 0101h
B	16 bits	00 0104h
C	32 bits	00 0106h
D	8 bits	00 010Ah
E	24 bits	00 010Bh

Byte addresses	Byte 0	Byte 1	Byte 2	Byte 3
00 0100h–00 0103h		A(23–16)	A(15–8)	A(7–0)
00 0104h–00 0107h	B(15–8)	B(7–0)	C(31–24)	C(23–16)
00 0108h–00 010Bh	C(15–8)	C(7–0)	D(7–0)	E(23–16)
00 010Ch–00 010Fh	E(15–8)	E(7–0)		

3.2.3 Alignment of Fetches From Program Space

You do not have to align instructions as you store them in program memory, but the instruction fetches are aligned to even-address 32-bit boundaries. During an instruction fetch, the CPU reads 32 bits from an address whose two least significant bits (LSBs) are 0s. In other words, the least significant digit of a hexadecimal fetch address is always 0h, 4h, 8h, or Ch.

When the CPU executes a discontinuity, the address written to the program counter (PC) might not be the same as the fetch address. The PC address and the fetch address are the same only if the two LSBs of the PC address are 0s. Consider the following assembly code segment, which calls a subroutine:

```
CALL #SubroutineB
```

Suppose the first instruction of the subroutine is instruction C at byte address 00 0106h, as shown in Figure 3–3. The PC contains 00 0106h, but the program-read address bus (PAB) carries the byte address at the previous 32-bit boundary, 00 0104h. The CPU reads 4-byte packets of code beginning at address 00 0104h. Instruction C is the first instruction executed.

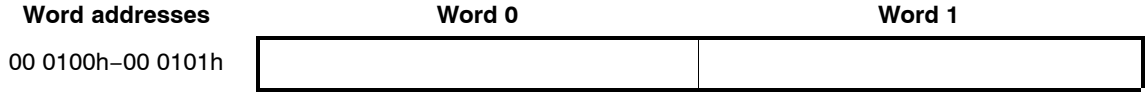
3.3 Data Space

When programs read from or write to memory or registers, the accesses are made to data space. The CPU uses word addresses (see section 3.3.1) to read or write 8-bit, 16-bit, or 32-bit values (see section 3.3.2). The address that needs to be generated for a particular value depends on how it its stored within the word boundaries in data space (see section 3.3.3).

3.3.1 Word Addresses (23 Bits)

When the CPU accesses data space, it uses word addresses, which are addresses assigned to individual 16-bit words. These addresses are 23 bits wide. Figure 3–4 shows a row of 32-bit-wide memory. Each word is assigned an address. Word 0 is at address 00 0100h and word 1 is at address 00 0101h.

Figure 3–4. Example of Word Addresses for 32-Bit-Wide Data Memory



The address buses are 24-bit buses. When the CPU reads from or writes to data space, the 23-bit address is concatenated with a trailing 0. For example, suppose an instruction reads a word at the 23-bit address 00 0102h. The appropriate data-read address bus carries the 24-bit value 00 0204h:

Word address: 000 0000 0000 0001 0000 0010

Data-read address bus: 0000 0000 0000 0010 0000 0100

3.3.2 Data Types

The instruction set handles the following data types:

byte 8 bits

word 16 bits

long word 32 bits

Dedicated instruction syntaxes (see Table 3–1) allow you to select high or low bytes of particular words. The byte-load instructions read bytes and load them into registers. The bytes that are read are zero extended (if the uns() operand qualifier is used) or sign extended before being stored. The byte-store instructions store the 8 least significant bits of a register to a specified byte in memory.

Note:

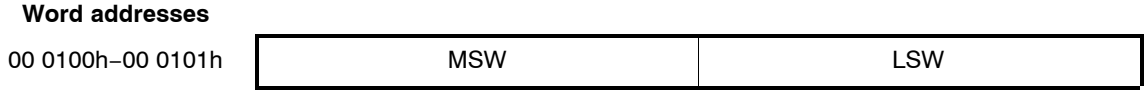
In data space, the CPU uses 23-bit addresses to access words. To access a byte, the CPU must manipulate the word that contains the byte.

Table 3–1. Byte Load and Byte Store Instructions

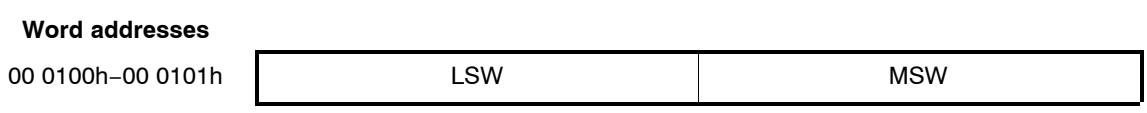
Operation	Instruction Syntax	Byte Accessed
Byte load (Accumulator, Auxiliary, or Temporary Register Load instructions)	MOV [uns(]high_byte(Smem)[)], dst	Smem(15–8)
	MOV [uns(]low_byte(Smem)[)], dst	Smem(7–0)
	MOV high_byte(Smem) << #SHIFTW, ACx	Smem(15–8)
	MOV low_byte(Smem) << #SHIFTW, ACx	Smem(7–0)
Byte store (Accumulator, Auxiliary, or Temporary Register Store instructions)	MOV src, high_byte(Smem)	Smem(15–8)
	MOV src, low_byte(Smem)	Smem(7–0)

When the CPU accesses long words, the address used for the access is the address of the most significant word (MSW) of the 32-bit value. The address of the least significant word (LSW) depends on the address of the MSW:

- If the address of the MSW is even, the LSW is accessed at the next address. For example:



- If the address of the MSW is odd, the LSW is accessed at the previous address. For example:



Given the address of the MSW (LSW), complement its least significant bit to find the address of the LSW (MSW).

3.3.3 Data Organization in Data Space

Figure 3–5 provides an example of how data are organized in data space. Seven data values of varying sizes have been stored in 32-bit-wide memory. No data value is stored in the shaded byte at address 00 0100h. Important points about the example are:

- To access a long word, you must reference its most significant word (MSW). C is accessed at address 00 0102h. D is accessed at address 00 0105h.
- Word addresses are also used to access bytes in data space. For example, the address 00 0107h is used for both F (high byte) and G (low byte). Special byte instructions indicate whether the high byte or low byte is accessed.

Figure 3–5. Example of Data Organization in Data Space

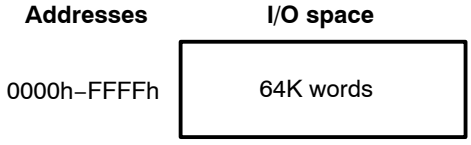
Data Value	Data Type	Address
A	Byte	00 0100h (low byte)
B	Word	00 0101h
C	Long Word	00 0102h
D	Long Word	00 0105h
E	Word	00 0106h
F	Byte	00 0107h (high byte)
G	Byte	00 0107h (low byte)

Word addresses	Word 0		Word 1	
00 0100h–00 0101h		A	B	
00 0102h–00 0103h	MSW of C (bits 31–16)		LSW of C (bits 15–0)	
00 0104h–00 0105h	LSW of D (bits 15–0)		MSW of D (bits 31–16)	
00 0106h–00 0107h	E		F	G

3.4 I/O Space

I/O space is separate from data/program space and is available only for accessing registers of the peripherals on the DSP. The word addresses in I/O space are 16 bits wide, enabling access to 64K locations (see Figure 3–6).

Figure 3–6. I/O Space



The CPU uses the data-read address bus DAB for reads and data-write address bus EAB for writes. When the CPU reads from or writes to I/O space, the 16-bit address is concatenated with leading 0s. For example, suppose an instruction reads a word at the 16-bit address 0102h. DAB carries the 24-bit value 00 0102h.

Note:

Although an increment past FFFFh or a decrement past 0000h causes the I/O address to wrap around, do not make use of this behavior; it is not supported.

3.5 Boot Loader

An on-chip boot loader provides options for transferring code and data from an external source to the RAM inside the C55x DSP at power up/reset. For a list of boot options for a particular C55x DSP and for a description on how to select the desired option, see the data manual for that DSP. To learn how the C55x hex conversion utility can help you with boot loading, see the *TMS320C55x Assembly Language Tools User's Guide* (literature number SPRU280).

Chapter 4

Stack Operation

This chapter introduces the two stacks located on each C55x DSP. It also explains how they relate to each other and how they are used by the CPU during automatic context switching (saving important register values before executing a subroutine and restoring those values when the subroutine is done).

Topic **Page**

4.1 Data Stack and System Stack

The CPU supports two 16-bit software stacks known as the data stack and the system stack. Figure 4–1 and Table 4–1 describe the registers used for the stack pointers. For an access to the data stack, the CPU concatenates SPH with SP to form XSP. XSP contains the 23-bit address of the value last pushed onto the data stack. SPH holds the 7-bit main data page of memory, and SP points to the specific word on that page. The CPU decrements SP before pushing a value onto the stack and increments SP after popping a value off the stack. SPH is not modified during stack operations.

Similarly, when accessing the system stack, the CPU concatenates SPH with SSP to form XSSP. XSSP contains the address of the value last pushed onto the system stack. The CPU decrements SSP before pushing a value onto the system stack and increments SSP after popping a value off the system stack. Again, SPH is not modified during stack operations.

As described in section 4.2, *Stack Configurations*, SSP can either be linked with or independent of SP. If you select the 32-bit stack configuration, operations that modify SP will modify SSP in the same way. If you select one of the dual 16-bit stack configurations, SSP is independent of SP; SSP will be modified only during automatic context switching (see section 4.4).

Figure 4–1. Extended Stack Pointers

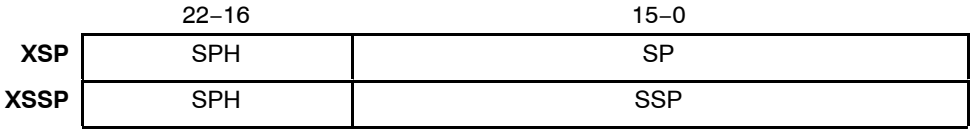


Table 4−1. Stack Pointer Registers

Register	Referred To As ...	Accessibility
XSP	Extended data stack pointer	Accessible via dedicated instructions only. XSP is not a register mapped to memory.
SP	Data stack pointer	Accessible via dedicated instructions and as a memory-mapped register
XSSP	Extended system stack pointer	Accessible via dedicated instructions only. XSSP is not a register mapped to memory.
SSP	System stack pointer	Accessible via dedicated instructions and as a memory-mapped register
SPH	High part of XSP and XSSP	Accessible as a memory-mapped register. You can also access SPH by accessing XSP or XSSP. There are no dedicated instructions for SPH. **Note:** SPH is not affected by writes to XSP or XSSP.

4.2 Stack Configurations

The TMS320C55x DSP provides three possible stack configurations, which are described in Table 4–2. Notice that one configuration features a fast-return process, and the others use a slow-return process. Section 4.3 explains the difference between the two processes.

Select one of the three stack configurations by placing the appropriate value in bits 29 and 28 of the 32-bit reset vector location (see Table 4–2). This can be done in C55x assembly code as part of the *.ivec* assembler directive. This directive is described in the *TMS320C55x Assembly Language User's Guide* (literature number SPRU280). The 24 least-significant bits in the reset vector location must be the start address for the reset interrupt service routine (ISR).

Table 4–2. Stack Configurations

Stack Configuration	Description	Reset Vector Value† (Binary)
Dual 16-bit stack with fast return	The data stack and the system stack are independent: When you access the data stack, the data stack pointer (SP) is modified, but the system stack pointer (SSP) is not. The registers RETA and CFCT are used to implement a fast return (see Figure 4–3 on page 4-7).	XX00 XXXX:(24-bit ISR address)
Dual 16-bit stack with slow return	The data stack and the system stack are independent: When you access the data stack, SP is modified, but SSP is not. RETA and CFCT are not used (see Figure 4–2 on page 4-6).	XX01 XXXX:(24-bit ISR address)
32-bit stack with slow return	The data stack and the system stack act as a single 32-bit stack: When you access the data stack, SP and SSP are modified by the same increment. RETA and CFCT are not used (see Figure 4–2 on page 4-6). **Note:** If you modify SP via its memory-mapped location, SSP is not automatically updated. In this case, you must also modify SSP to keep the two pointers aligned.	XX10 XXXX:(24-bit ISR address)
–	Reserved. Do not set bits 29 and 28 both to 1.	XX11 XXXX:(24-bit ISR address) (This is an illegal value.)

† A bit shown as an X may be 0 or 1.

4.3 Fast Return Versus Slow Return

The difference between the fast-return process and the slow-return process is how the CPU saves and restores the value of two internal registers: the program counter (PC) and a loop context register.

PC holds the 24-bit address of the 1 to 6 bytes of code being decoded in the I unit. When the CPU performs an interrupt or call, the current PC value (the return address) is stored, and then PC is loaded with the start address of the interrupt service routine or called routine. When the CPU returns from the routine, the return address is transferred back to PC, so that the interrupted program sequence can continue as before.

An 8-bit loop context register keeps a record of active repeat loops (the loop context). When the CPU performs an interrupt or call, the current loop context is stored, and then the 8-bit register is cleared to create a new context for the subroutine. When the CPU returns from the subroutine, the loop context is transferred back to the 8-bit register.

In the slow-return process, the return address and the loop context are stored to the stacks (in memory). When the CPU returns from a subroutine, the speed at which these values are restored is dependent on the speed of the memory accesses.

In the fast-return process, the return address and the loop context are saved to registers, so that these values can always be restored quickly. These special registers are the return address register (RETA) and the control-flow context register (CFCT). You can read from or write to RETA and CFCT as a pair with dedicated, 32-bit load and store instructions.

Figure 4–2 (slow return) and Figure 4–3 (fast return) show examples of how the return address and the loop context are handled within several layers of routines. In these figures, Routine 0 is the highest level routine, Routine 1 is nested inside Routine 0, and Routine 2 is nested inside Routine 1.

Figure 4–2. Return Address and Loop Context Passing During Slow-Return Process

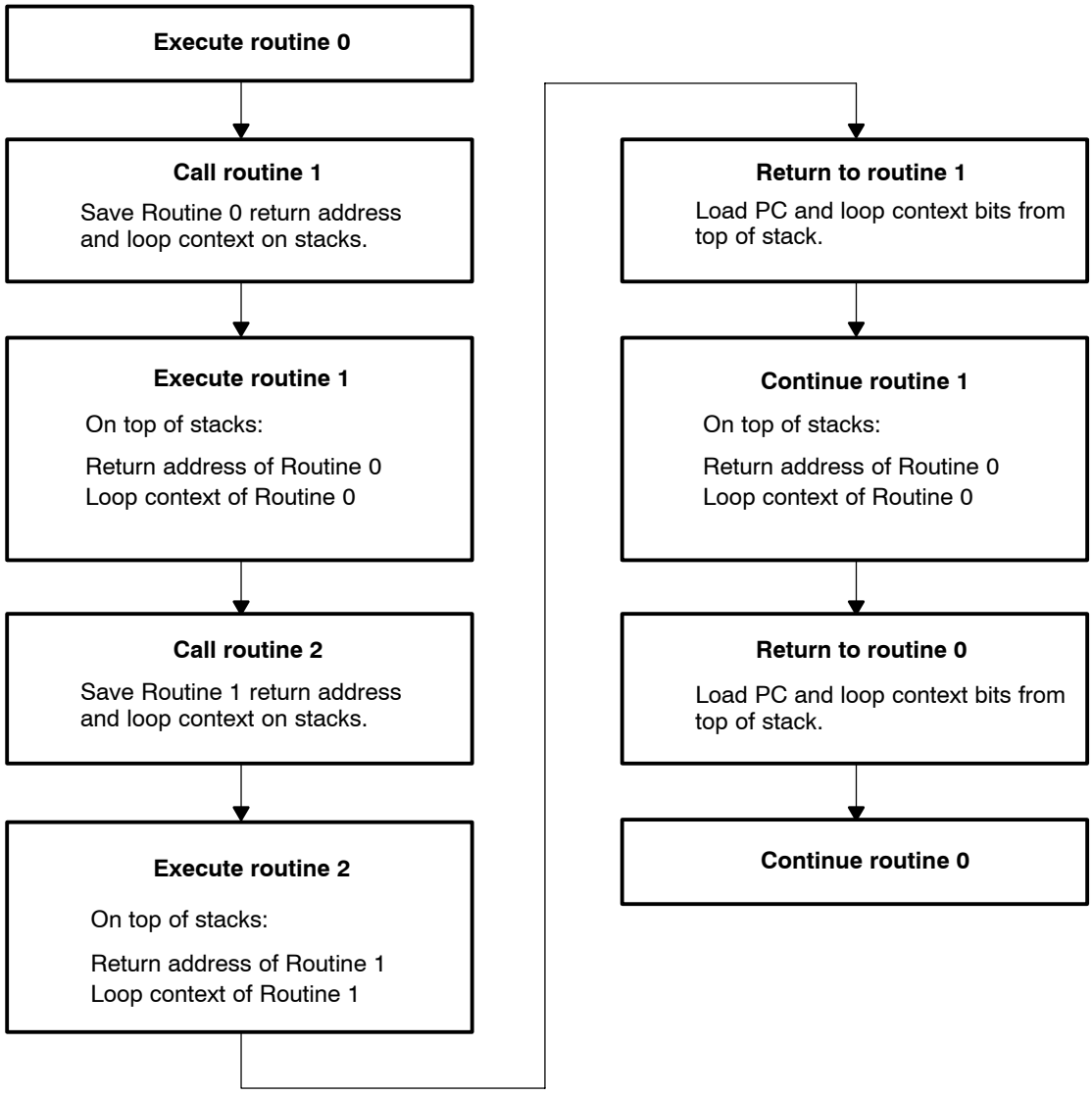
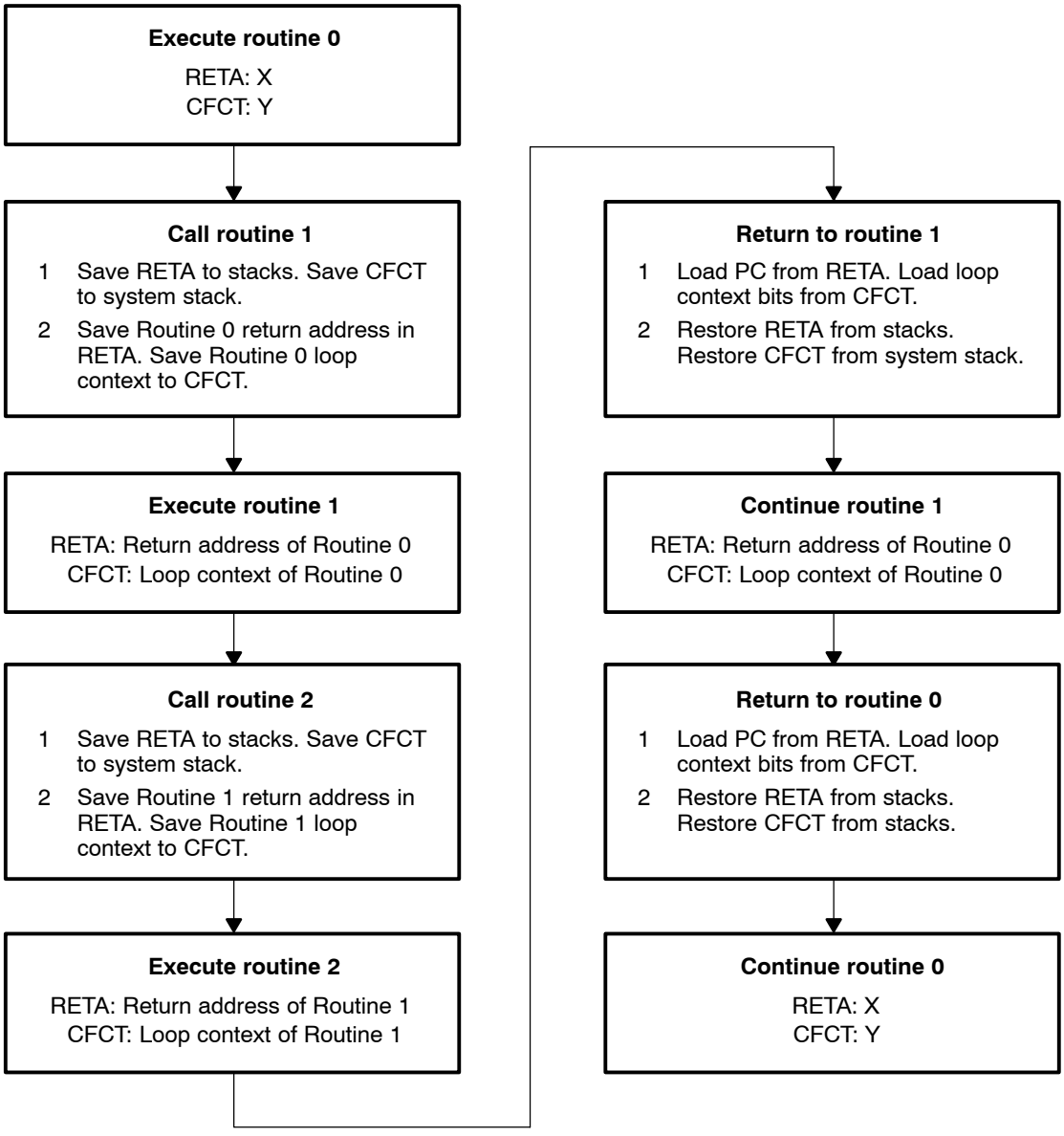


Figure 4–3. Use of RETA and CFCT in Fast-Return Process



4.4 Automatic Context Switching

Before beginning an interrupt service routine (ISR) or a called routine, the CPU automatically saves certain values. The CPU can use these values to re-establish the context of the interrupted program sequence when the subroutine is done.

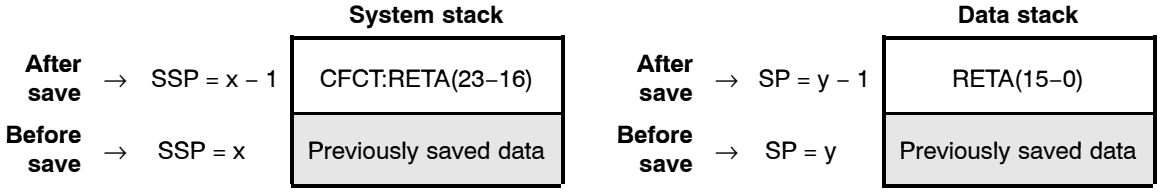
Whether responding to an interrupt or a call, the CPU saves the return address and the loop context bits. The return address, taken from the program counter (PC), is the address of the instruction to be executed when the CPU returns from the subroutine. The loop context bits are a record of the type and status of repeat loops that were active when the interrupt or call occurred. When responding to an interrupt, the CPU additionally saves status registers 0, 1, and 2 and the debug status register (DBSTAT). DBSTAT is a DSP register that holds debug context information used during emulation.

If the selected stack configuration (see section 4.2) uses the fast-return process, RETA is used as a temporary storage place for the return address, and CFCT is used as a temporary storage place for the loop context bits. If the selected stack configuration uses the slow-return process, the return address and the loop context bits are saved to and restored from the stack.

4.4.1 Fast-Return Context Switching for Calls

Before beginning a called routine, the CPU automatically:

1) Saves CFCT and RETA to the system stack and the data stack in parallel. For each stack, the CPU decrements the stack pointer (SSP or SP) by 1 before the write to the stack:



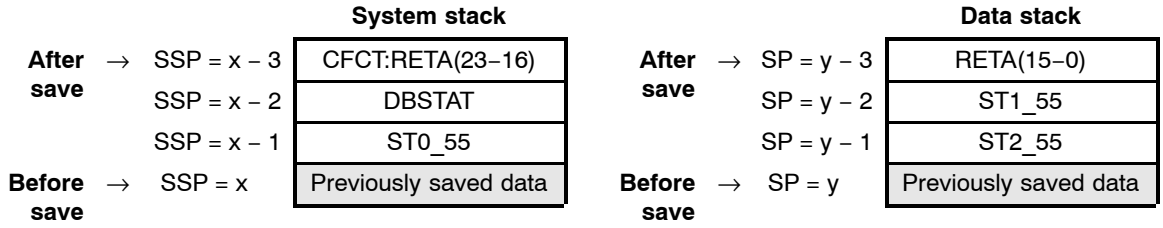
2) Saves the return address to RETA and saves loop context flags in CFCT:

A return instruction at the end of a subroutine forces the CPU to restore values in the opposite order. First, the CPU transfers the return address from RETA to PC and restores its loop context flags from CFCT. Second, the CPU reads the CFCT and RETA values from the stacks in parallel. For each stack, the CPU increments the stack pointer (SSP or SP) by 1 after the read from the stack.

4.4.2 Fast-Return Context Switching for Interrupts

Before beginning an interrupt service routine (ISR), the CPU automatically:

1) Saves registers to the system stack and the data stack in parallel. For each stack, the CPU decrements the stack pointer (SSP or SP) by 1 before each write to the stack:



Note:

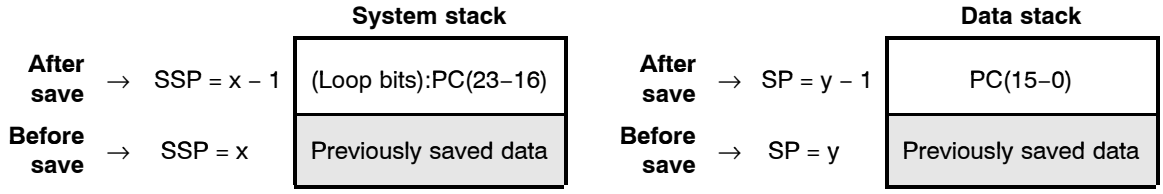
DBSTAT (the debug status register) holds debug context information used during emulation. Make sure the ISR does not modify the value that will be returned to DBSTAT.

2) Saves the return address (from PC) to RETA and saves loop context flags in CFCT:

A return-from-interrupt instruction at the end of an ISR forces the CPU to restore values in the opposite order. First, the CPU transfers the return address from RETA to PC and restores its loop context flags from CFCT. Second, the CPU reads the values from the stacks in parallel. For each stack, the CPU increments the stack pointer (SSP or SP) by 1 after each read from the stack.

4.4.3 Slow-Return Context Switching for Calls

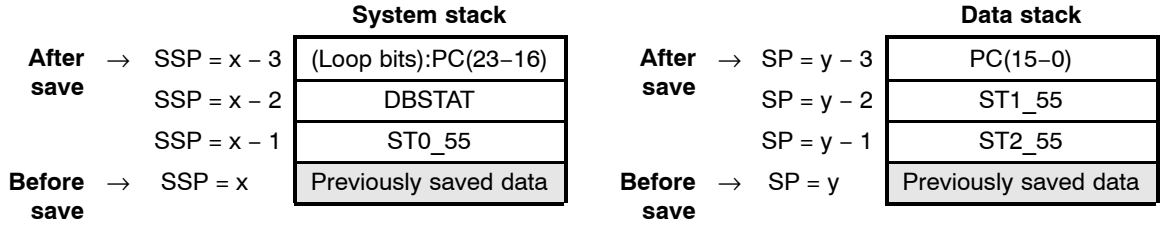
Before beginning a called routine, the CPU automatically saves the return address (from PC) and the loop context bits to the system stack and the data stack in parallel. For each stack, the CPU decrements the stack pointer (SSP or SP) by 1 before the write to the stack:



A return instruction at the end of a subroutine forces the CPU to restore the return address and the loop context from the stack. For each stack, the CPU increments the stack pointer (SSP or SP) by 1 after the read from the stack.

4.4.4 Slow-Return Context Switching for Interrupts

Before beginning an interrupt service routine (ISR), the CPU automatically saves registers to the system stack and the data stack in parallel. For each stack, the CPU decrements the stack pointer (SSP or SP) by 1 before each write to the stack:



Note:

DBSTAT (the debug status register) holds debug context information used during emulation. Make sure the ISR does not modify the value that will be returned to DBSTAT.

A return-from-interrupt instruction at the end of an ISR forces the CPU to restore values in the opposite order. First, the CPU restores the return address and the loop context bits from the stack. Second, the CPU reads the other values from the stacks in parallel. For each stack, the CPU increments the stack pointer (SSP or SP) by 1 after each read from the stack.

Chapter 5

Interrupts and Reset Operations

This chapter describes the available interrupts of the C55x DSP, how some of them can be blocked through software, and how all of them are handled by the CPU. This chapter also explains the automatic effects of two types of reset operations, one initiated by hardware and one initiated by software.

Topic **Page**

5.1 Introduction to the Interrupts

Interrupts are hardware- or software-driven signals that cause the DSP to suspend its current program sequence and execute another task called an interrupt service routine (ISR). The TMS320C55x (C55x) DSP supports 32 ISRs. Some of the ISRs can be triggered by software or hardware; others can be triggered only by software. When the CPU receives multiple hardware interrupt requests at the same time, the CPU services them according to a predefined priority ranking (see section 5.2, *Interrupt Vectors and Priorities*).

All C55x interrupts, whether hardware or software, can be placed in one of two categories. Maskable interrupts can be blocked (masked) through software. Nonmaskable interrupts cannot be blocked. All software interrupts are nonmaskable.

The DSP handles interrupts in four main phases:

1) Receive the interrupt request. Software or hardware requests a suspension of the current program sequence.

2) Acknowledge the interrupt request. The CPU must acknowledge the request. If the interrupt is maskable, certain conditions must be met for acknowledgment. For nonmaskable interrupts, acknowledgment is immediate.

3) Prepare for the interrupt service routine. The main tasks performed by the CPU are:

   - ❏ Complete execution of the current instruction and flush from the pipeline any instructions that have not reached the decode phase.
   - ❏ Automatically store certain register values to the data stack and the system stack (see the description of automatic context switching in section 4.4).
   - ❏ Fetch the interrupt vector that you store at a preset vector address. The interrupt vector points to the interrupt service routine.

4) Execute the interrupt service routine. The CPU executes the ISR that you have written. The ISR is concluded with a return-from-interrupt instruction, which automatically restores the register values that were automatically saved (see the description of automatic context switching in section 4.4).

Notes:

1) External interrupts must occur at least 3 cycles after the CPU exits reset or they will not be recognized.

2) All interrupts (maskable and nonmaskable) are disabled following a hardware reset, regardless of the setting of the INTM bit and the IER0 and IER1 registers. Interrupts remain disabled until the stack pointers are initialized by a software write to each pointer (the SP and SSP registers). After stack initialization, the INTM bit and the IER0 and IER1 registers determine interrupt enabling.

5.2 Interrupt Vectors and Priorities

The TMS320C55x DSP supports 32 interrupt service routines (ISRs). After receiving and acknowledging an interrupt request, the CPU generates an interrupt vector address. At the vector address, the CPU fetches the vector that points to the corresponding ISR. When multiple hardware interrupts occur simultaneously, the CPU services them one at a time, according to their predefined hardware interrupt priorities. Table 5–1 shows the vectors sorted by ISR number. Table 5–2 shows the vectors sorted by priority. Both tables show only a general representation of the C55x vectors. To determine which interrupt corresponds to each of the vectors, see the data manual for your C55x DSP.

You must write the desired interrupt vectors (ISR start address) at the vector addresses. Each interrupt vector must contain 8 bytes. Byte 0 of the reset vector contains the setting for the stack mode. Byte 0 of the remaining vectors is ignored. Bytes 1–3 encode the 24-bit byte address of the interrupt service routine (ISR). Bytes 4–7 must be filled with NOP instructions.

Vector pointers IVPD and IVPH point to up to 32 interrupt vectors in program space. IVPD points to the 256-byte program page for interrupt vectors 0–15 and 24–31. IVPH points to the 256-byte program page for interrupt vectors 16–23. The details about these pointers are in section 2.8.1.

Table 5–1. Interrupt Vectors Sorted by ISR Number

ISR Number	Hardware Interrupt Priority	Vector Name	Vector Address (Byte Address)	This ISR Is For ...
0	1 (highest)	RESETIV(IV0)	IVPD:0h	Reset (hardware and software)
1	2	NMIV (IV1)	IVPD:8h	Hardware nonmaskable interrupt ($\overline{\text{NMI}}$) or software interrupt 1
2	4	IV2	IVPD:10h	Hardware or software interrupt
3	6	IV3	IVPD:18h	Hardware or software interrupt
4	7	IV4	IVPD:20h	Hardware or software interrupt
5	8	IV5	IVPD:28h	Hardware or software interrupt
6	10	IV6	IVPD:30h	Hardware or software interrupt
7	11	IV7	IVPD:38h	Hardware or software interrupt
8	12	IV8	IVPD:40h	Hardware or software interrupt
9	14	IV9	IVPD:48h	Hardware or software interrupt

Table 5–1. Interrupt Vectors Sorted by ISR Number (Continued)

ISR Number	Hardware Interrupt Priority	Vector Name	Vector Address (Byte Address)	This ISR Is For ...
10	15	IV10	IVPD:50h	Hardware or software interrupt
11	16	IV11	IVPD:58h	Hardware or software interrupt
12	18	IV12	IVPD:60h	Hardware or software interrupt
13	19	IV13	IVPD:68h	Hardware or software interrupt
14	22	IV14	IVPD:70h	Hardware or software interrupt
15	23	IV15	IVPD:78h	Hardware or software interrupt
16	5	IV16	IVPH:80h	Hardware or software interrupt
17	9	IV17	IVPH:88h	Hardware or software interrupt
18	13	IV18	IVPH:90h	Hardware or software interrupt
19	17	IV19	IVPH:98h	Hardware or software interrupt
20	20	IV20	IVPH:A0h	Hardware or software interrupt
21	21	IV21	IVPH:A8h	Hardware or software interrupt
22	24	IV22	IVPH:B0h	Hardware or software interrupt
23	25	IV23	IVPH:B8h	Hardware or software interrupt
24	3	BERRIV (IV24)	IVPD:C0h	Bus error interrupt or software interrupt
25	26	DLOGIV (IV25)	IVPD:C8h	Data log interrupt or software interrupt
26	27 (lowest)	RTOSIV (IV26)	IVPD:D0h	Real-time operating system interrupt or software interrupt
27	–	IV27	IVPD:D8h	Reserved
28	–	IV28	IVPD:E0h	Reserved
29	–	IV29	IVPD:E8h	Reserved
30	–	SIV30	IVPD:F0h	Software (only) interrupt
31	–	SIV31	IVPD:F8h	Software (only) interrupt 31

Table 5–2. Interrupt Vectors Sorted by Priority

ISR Number	Hardware Interrupt Priority	Vector Name	Vector Address (Byte Address)	This ISR Is For(..
0	1 (highest)	RESETIV(IV0)	IVPD:0h	Reset (hardware and software)
1	2	NMIV (IV1)	IVPD:8h	Hardware nonmaskable interrupt ($\overline{\text{NMI}}$) or software interrupt 1
24	3	BERRIV (IV24)	IVPD:C0h	Bus error interrupt or software interrupt
2	4	IV2	IVPD:10h	Hardware or software interrupt
16	5	IV16	IVPH:80h	Hardware or software interrupt
3	6	IV3	IVPD:18h	Hardware or software interrupt
4	7	IV4	IVPD:20h	Hardware or software interrupt
5	8	IV5	IVPD:28h	Hardware or software interrupt
17	9	IV17	IVPH:88h	Hardware or software interrupt
6	10	IV6	IVPD:30h	Hardware or software interrupt
7	11	IV7	IVPD:38h	Hardware or software interrupt
8	12	IV8	IVPD:40h	Hardware or software interrupt
18	13	IV18	IVPH:90h	Hardware or software interrupt
9	14	IV9	IVPD:48h	Hardware or software interrupt
10	15	IV10	IVPD:50h	Hardware or software interrupt
11	16	IV11	IVPD:58h	Hardware or software interrupt
19	17	IV19	IVPH:98h	Hardware or software interrupt
12	18	IV12	IVPD:60h	Hardware or software interrupt
13	19	IV13	IVPD:68h	Hardware or software interrupt
20	20	IV20	IVPH:A0h	Hardware or software interrupt
21	21	IV21	IVPH:A8h	Hardware or software interrupt
14	22	IV14	IVPD:70h	Hardware or software interrupt
15	23	IV15	IVPD:78h	Hardware or software interrupt
22	24	IV22	IVPH:B0h	Hardware or software interrupt

Table 5–2. Interrupt Vectors Sorted by Priority (Continued)

ISR Number	Hardware Interrupt Priority	Vector Name	Vector Address (Byte Address)	This ISR Is For(..
23	25	IV23	IVPH:B8h	Hardware or software interrupt
25	26	DLOGIV (IV25)	IVPD:C8h	Data log interrupt or software interrupt
26	27 (lowest)	RTOSIV (IV26)	IVPD:D0h	Real-time operating system interrupt or software interrupt
27	–	IV27	IVPD:D8h	Reserved
28	–	IV28	IVPD:E0h	Reserved
29	–	IV29	IVPD:E8h	Reserved
30	–	SIV30	IVPD:F0h	Software (only) interrupt
31	–	SIV31	IVPD:F8h	Software (only) interrupt 31

5.3 Maskable Interrupts

Maskable interrupts can be blocked (masked) or enabled (unmasked) through software. All of the TMS320C55x maskable interrupts are hardware interrupts:

Interrupt	Description
Interrupts associated with interrupt vectors 2–23	Each of these 22 interrupts is triggered at a pin or by a peripheral of the DSP.
BERRINT	Bus error interrupt. This interrupt is triggered when a system bus error is transmitted to the CPU or when a bus error occurs in the CPU.
DLOGINT	Data log interrupt. DLOGINT is triggered by the DSP at the end of a data log transfer. You can use the DLOGINT interrupt service routine (ISR) to start the next data log transfer.
RTOSINT	Real-time operating system interrupt. RTOSINT can be triggered by a hardware breakpoint or watchpoint. You can use the RTOSINT ISR to begin a data log transfer in response to an emulation condition.

Whenever a maskable interrupt is requested by hardware, the corresponding interrupt flag is set in one of the interrupt flag registers (see the description of IFR0 and IFR1 in section 2.8.2). Once the flag is set, the interrupt is not serviced unless it is properly enabled (see section 5.3.1, *Bit and Registers Used to Enable Maskable Interrupts)*.

The ISRs for the maskable interrupts can also be executed by software (see the discussion on nonmaskable interrupts in section 5.4).

Note:

When a bus error occurs, the functionality of the instruction that caused the error, and of any instruction executed in parallel, cannot be assured.

5.3.1 Bit and Registers Used to Enable Maskable Interrupts

The following bit and registers are used to enable the maskable interrupts:

Bit/Registers	Description
INTM	Interrupt mode bit. This bit globally enables/disables the maskable interrupts. (See the description of INTM in section 2.10.2.8.)
IER0 and IER1	Interrupt enable registers. Each maskable interrupt has an enable bit in one of these two registers. (See the description of IER0 and IER1 in section 2.8.3.)
DBIER0 and DBIER1	Debug interrupt enable registers. Each maskable interrupt can be defined as time-critical by a bit in one of these two registers. (See the description of DBIER0 and DBIER1 in section 2.8.4.)

As shown in the next two sections, the roles of INTM, the IER bit, and the DBIER bit depend on the operating condition of the DSP.

5.3.2 Standard Process Flow for Maskable Interrupts

The flow chart in Figure 5–1 provides a conceptual model of the standard process for handling maskable interrupts. Table 5–3 describes each of the steps in the flow chart. When the CPU is halted in the real-time emulation mode, only time-critical interrupts can be serviced, and the process is different (see section 5.3.3).

Figure 5−1. Standard Process Flow for Maskable Interrupts

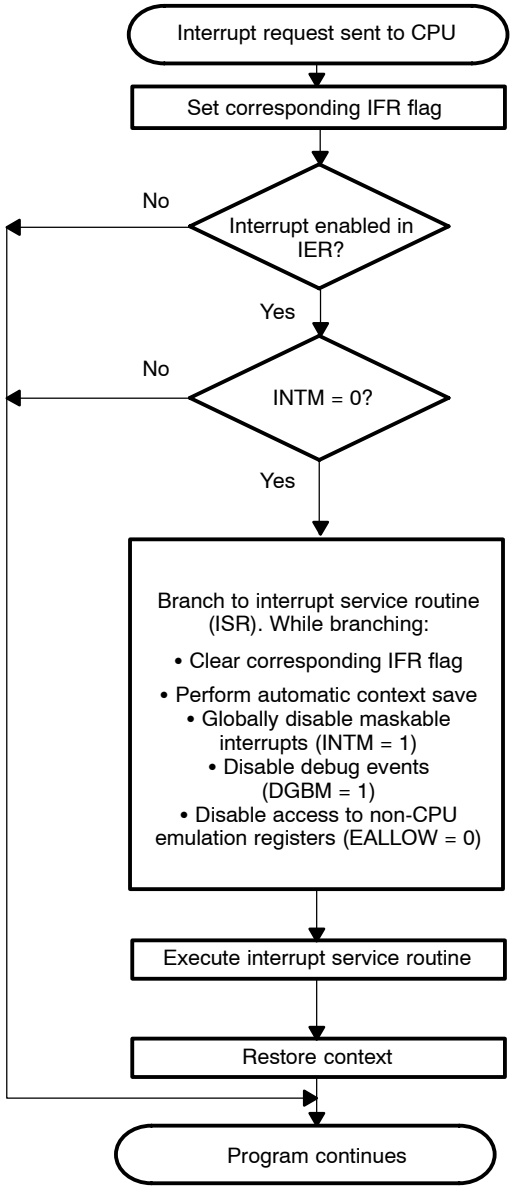


Table 5–3. Steps in the Standard Process Flow for Maskable Interrupts

Step	Description
Interrupt request sent to CPU	The CPU receives a maskable interrupt request.
Set corresponding IFR flag	When the CPU detects a valid maskable interrupt request, it sets and latches the corresponding flag in one of the interrupt flag registers (IFR0 or IFR1). This flag stays latched until the interrupt is acknowledged or until the flag is cleared by software or by a DSP hardware reset. (See the description of IFR0 and IFR1 in section 2.8.2.)
Interrupt enabled in IER?	The CPU cannot acknowledge the interrupt unless the corresponding enable bit is 1 in one of the interrupt enable registers (IER0 or IER1). (See the description of IER0 and IER1 in section 2.8.3.)
INTM = 0?	The CPU cannot acknowledge the interrupt unless the interrupt mode bit (INTM) is 0. That is, interrupts must be globally enabled. (See the description of INTM in section 2.10.2.8.)
Branch to interrupt service routine	The CPU follows the interrupt vector to the interrupt service routine. While branching, the CPU performs the following actions: ❑ It completes instructions that have already made it to the decode phase of the pipeline. Other instructions are flushed from the pipeline. ❑ It clears the corresponding flag in IFR0 or IFR1, to indicate that the interrupt has been acknowledged. ❑ It saves certain registers values automatically, to record important mode and status information about the interrupted program sequence (see the description of automatic context switching in section 4.4). ❑ It creates a fresh context for the ISR by forcing INTM = 1 (globally disables interrupts), DBGM = 1 (disables debug events), and EALLOW = 0 (disables access to non-CPU emulation registers).
Execute interrupt service routine	The CPU executes the interrupt service routine (ISR) that you have written for the acknowledged interrupt. Some registers values were saved automatically during the branch to the ISR. A return-from-interrupt instruction at the end of your ISR forces an automatic context restore operation (see the description of automatic context switching in section 4.4) to restore these register values. If the ISR shares other registers with the interrupted program sequence, the ISR must save other register values at the beginning of the ISR and restore these values before returning to the interrupted program sequence.
Program continues	If the interrupt request is not properly enabled, the CPU ignores the request, and the program continues uninterrupted. If the interrupt is properly enabled, its interrupt service routine is executed, and then the program continues from the point where it was interrupted.

5.3.3 Process Flow for Time-Critical Interrupts

The flow chart in Figure 5–2 and the descriptions in Table 5–4 provide a conceptual model of how time-critical interrupts are handled. When the CPU is halted in the real-time emulation mode, the only maskable interrupts that can be serviced are the time-critical interrupts. In all other cases, the CPU uses the standard process flow that is described in section 5.3.2.

Figure 5–2. Process Flow for Time-Critical Interrupts

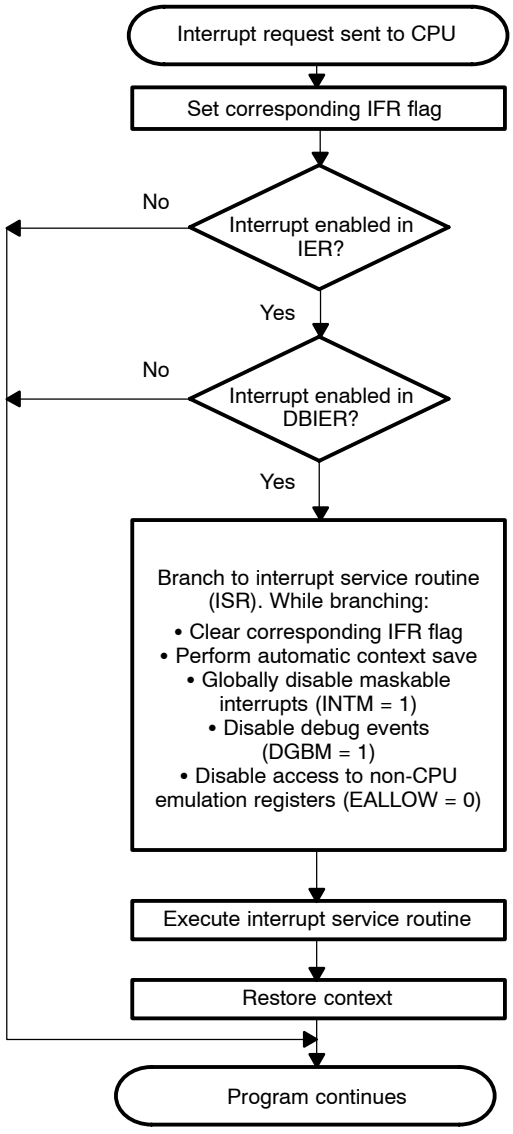


Table 5–4. Steps in the Process Flow for Time-Critical Interrupts

Step	Description
Interrupt request sent to CPU	The CPU receives a maskable interrupt request.
Set corresponding IFR flag	When the CPU detects a valid maskable interrupt request, it sets and latches the corresponding flag in one of the interrupt flag registers (IFR0 or IFR1). This flag stays latched until the interrupt is acknowledged or until the flag is cleared by software or by a DSP hardware reset. (See the description of IFR0 and IFR1 in section 2.8.2.)
Interrupt enabled in IER?	The CPU cannot acknowledge the interrupt unless the corresponding enable bit is 1 in one of the interrupt enable registers (IER0 or IER1). (See the description of IER0 and IER1 in section 2.8.3.)
Interrupt enabled in DBIER?	The CPU cannot acknowledge the interrupt unless the corresponding enable bit is 1 in one of the debug interrupt enable registers (DBIER0 or DBIER1). (See the description of DBIER0 and DBIER1 in section 2.8.4.)
Branch to interrupt service routine	The CPU follows the interrupt vector to the interrupt service routine. While branching, the CPU performs the following actions: ❑ It completes instructions that have already made it to the decode phase of the pipeline. Other instructions are flushed from the pipeline. ❑ It clears the corresponding flag in IFR0 or IFR1, to indicate that the interrupt has been acknowledged. ❑ It saves certain registers values automatically, to record important mode and status information about the interrupted program sequence (see the description of automatic context switching in section 4.4). ❑ It creates a fresh context for the ISR by forcing INTM = 1 (globally disables interrupts), DBGM = 1 (disables debug events), and EALLOW = 0 (disables access to non-CPU emulation registers).
Execute interrupt service routine	The CPU executes the interrupt service routine (ISR) that you have written for the acknowledged interrupt. Some registers values were saved automatically during the branch to the ISR. A return-from-interrupt instruction at the end of your ISR forces an automatic context restore operation (see the description of automatic context switching in section 4.4) to restore these register values. If the ISR shares other registers with the interrupted program sequence, the ISR must save other register values at the beginning of the ISR and restore these values before returning to the interrupted program sequence.
Program continues	If the interrupt request is not properly enabled, the CPU ignores the request, and the program continues uninterrupted. If the interrupt is properly enabled, its interrupt service routine is executed, and then the program continues from the point where it was interrupted.

5.4 Nonmaskable Interrupts

When the CPU receives a nonmaskable interrupt request, the CPU acknowledges it unconditionally and immediately branches to the corresponding interrupt service routine (ISR). The nonmaskable interrupts are:

- The hardware interrupt $\overline{\text{RESET}}$. If you drive the $\overline{\text{RESET}}$ pin low, you initiate a DSP hardware reset plus an interrupt that forces execution of the reset ISR. (Specific effects of a DSP hardware reset are described in section 5.5.1.)
- The hardware interrupt $\overline{\text{NMI}}$. If you drive the $\overline{\text{NMI}}$ pin low, you force the CPU to execute the corresponding ISR. $\overline{\text{NMI}}$ provides a general-purpose, hardware method to interrupt the DSP unconditionally.
- All software interrupts, which are initiated by one of the following instructions.

Instruction	Description
INTR #k5	You can initiate any of the 32 ISRs with this instruction. The variable k5 is a 5-bit number from 0 to 31. Before executing the ISR, the CPU performs an automatic context save (to save important register values) and sets the INTM bit (to globally disable maskable interrupts).
TRAP #k5	This instruction performs the same function as INTR #k5, except that it does **not** affect the INTM bit.
RESET	This instruction performs a software reset operation, which is a subset of the hardware reset operation, and then forces the CPU to execute the reset ISR. (Specific effects of a software reset are described in section 5.5.2.)

5.4.1 Standard Process Flow for Nonmaskable Interrupts

The following flow chart provides a conceptual model of the standard process for handling nonmaskable interrupts.

Note:

If the interrupt was initiated by a TRAP instruction, the INTM bit is not affected during the branch to the interrupt service routine.

Figure 5–3. Standard Process Flow for Nonmaskable Interrupts

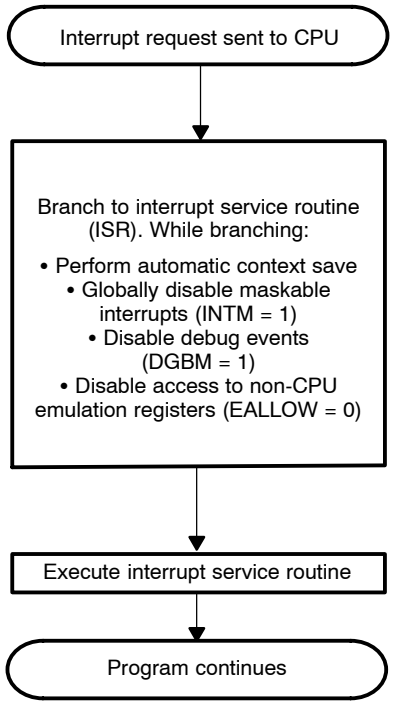


Table 5–5. Steps in the Standard Process Flow for Nonmaskable Interrupts

Step	Description
Interrupt request sent to CPU	The CPU receives a nonmaskable interrupt request.
Branch to interrupt service routine	The CPU follows the interrupt vector to the interrupt service routine. While branching, the CPU performs the following actions: ❑ It completes instructions that have already made it to the decode phase of the pipeline. Other instructions are flushed from the pipeline. ❑ It saves certain registers values automatically, to record important mode and status information about the interrupted program sequence (see the description of automatic context switching in section 4.4). ❑ It creates a fresh context for the ISR by forcing INTM = 1 (globally disables interrupts), DBGM = 1 (disables debug events), and EALLOW = 0 (disables access to non-CPU emulation registers).
Execute interrupt service routine	The CPU executes the interrupt service routine (ISR) that you have written for the acknowledged interrupt. Some registers values were saved automatically during the branch to the ISR. A return-from-interrupt instruction at the end of your ISR forces an automatic context restore operation (see the description of automatic context switching in section 4.4) to restore these register values. If the ISR shares other registers with the interrupted program sequence, the ISR must save other register values at the beginning of the ISR and restore these values before returning to the interrupted program sequence.
Program continues	After the interrupt service routine is executed, the program continues from the point where it was interrupted.

5.5 DSP Reset

This section covers DSP hardware and software reset operations. Table 5–6 summarizes the effects of the hardware and software reset on the DSP registers.

Section 5.5.1 describes the DSP hardware reset, and section 5.5.2 describes the DSP software reset. Table 5–6 summarizes the effects of both types of reset on the CPU registers.

Note:

A hardware reset loads the interrupt vector pointer called IVPD with FFFFh and, thus, forces the CPU to fetch the reset vector from program address FF FF00h. During a software reset, IVPD remains unchanged; the CPU fetches the reset vector using the current IVPD value.

Table 5–6. Effects of a Reset on CPU Registers

		Reset Value		
Register	**Bit(s)**	**H/W**	**S/W**	**Comments**
AC0–AC3	all	0	0	
BK03, BK47, BKC	all	0	0	
BRC0, BRC1	all	0	0	
BRS1	all	0	0	
BSA01	all	0	†	
BSA23	all	0	†	
BSA45	all	0	†	
BSA67	all	0	†	
BSAC	all	0	†	
CFCT	all	0	†	The content of any active loops is cleared.
CSR	all	0	0	
DBIER0, DBIER1	all	0	†	All time-critical interrupts are disabled.

† Not affected by a software reset

Table 5–6. Effects of a Reset on CPU Registers (Continued)

Register	Bit(s)	Reset Value H/W	Reset Value S/W	Comments
IER0 IER1	all	0	0	All maskable interrupts are disabled.
IFR0 IFR1	all	0	0	All pending interrupts are cleared.
IVPD	all	FFFFh	†	The vectors reference by IVPD are in the 256-byte program page that begins with address FFF FF00h.
IVPH	all	FFFFh	†	The vectors referenced by IVPH are in the same 256-byte program page as the vectors referenced by IVPD.
PC	all	0	0	
PDP	all	0	0	
REA0, REA1	all	0	0	
RETA	all	0	†	Any return address stored in RETA is cleared.
RPTC	all	0	0	
RSA0, RSA1	all	0	0	
ST0_55	0–8: DP	0	0	Data page 0 is selected. Flags in ST0_55 are cleared.
	9: ACOV1	0	0	
	10: ACOV0	0	0	
	11: C	1	1	
	12: TC2	1	1	
	13: TC1	1	1	
	14: ACOV3	0	0	
	15: ACOV2	0	0	

† Not affected by a software reset

Table 5–6. Effects of a Reset on CPU Registers (Continued)

Register	Bit(s)	Reset Value H/W	Reset Value S/W	Comments
ST1_55	0–4: ASM	0	0	Instructions affected by ASM use a shift count of 0 (no shift). When ASM is cleared, register T2 is also cleared. This is due to the relationship between ASM and T2 when C54CM = 1 (see the description for ASM in section 2.10.2.1).
	5: C54CM	1	1	The TMS320C54x-compatible mode is on.
	6: FRCT	0	0	Results of multiply operations are not shifted.
	7: C16	0	0	The dual 16-bit mode is off. For an instruction that is affected by C16, the D-unit ALU performs one 32-bit operation rather than two parallel 16-bit operations.
	8: SXMD	1	1	The sign-extension mode is on.
	9: SATD	0	0	The CPU does not saturate overflow results in the D unit.
	10: M40	0	0	The 32-bit (rather than 40-bit) computation mode is selected for the D unit.
	11: INTM	1	1	Maskable interrupts are globally disabled.
	12: HM	0	0	When an active $\overline{\text{HOLD}}$ signal forces the DSP to place its external interface in the high-impedance state, the DSP continues executing code from internal memory.
	13: XF	1	1	Pin XF is driven high.
	14: CPL	0	0	The DP (rather than SP) direct addressing mode is selected. Direct accesses to data space are made relative to the data page register (DP).
	15: BRAF	0	0	This flag is cleared. (BRAF indicates/controls the status of a block-repeat operation.)
ST2_55	0: AR0LC	0	0	AR0 is used for linear addressing (rather than circular addressing).
	1: AR1LC	0	0	AR1 is used for linear addressing.
	2: AR2LC	0	0	AR2 is used for linear addressing.
	3: AR3LC	0	0	AR3 is used for linear addressing.
	4: AR4LC	0	0	AR4 is used for linear addressing.
	5: AR5LC	0	0	AR5 is used for linear addressing.
	6: AR6LC	0	0	AR6 is used for linear addressing.

† Not affected by a software reset

Table 5–6. Effects of a Reset on CPU Registers (Continued)

Register	Bit(s)	Reset Value H/W	Reset Value S/W	Comments
ST2_55 (continued)	7: AR7LC	0	0	AR7 is used for linear addressing.
	8: CDPLC	0	0	CDP is used for linear addressing.
	9: Reserved	0	0	
	10: RDM	0	0	When an instruction specifies that an operand should be rounded, the CPU uses rounding to the infinite (rather than rounding to the nearest).
	11: EALLOW	0	0	A program cannot write to the non-CPU emulation registers.
	12: DBGM	1	1	Debug events are disabled.
	13–14: Reserved	11b	11b	
	15: ARMS	0	0	When you use the AR indirect addressing mode, the DSP mode (rather than control mode) operands are available.
ST3_55	0: SST	0	†	In the TMS320C54x-compatible mode (C54CM = 1), the execution of some accumulator-store instructions is affected by SST. When SST is 0, the 40-bit accumulator value is not saturated to a 32-bit value before the store operation.
	1: SMUL	0	†	The results of multiplications will not be saturated.
	2: CLKOFF	0	†	The output of the CLKOUT pin is enabled; it reflects the CLKOUT clock signal.
	3–4: Reserved	0	†	
	5: SATA	0	0	The CPU does not saturate overflow results in the A unit.
	6: MPNMC	pins	†	The reset value of MPNMC may be dependent on the state of predefined pins at reset. To check this for a particular C55x DSP, see its data manual.
	7: CBERR	0	†	This flag is cleared. (CBERR indicates when an internal bus error is detected.)
	11–8: Reserved	1100b	†	

† Not affected by a software reset

Table 5–6. Effects of a Reset on CPU Registers (Continued)

Register	Bit(s)	Reset Value H/W	Reset Value S/W	Comments
ST3_55 (continued)	12: HINT	1	†	The signal used to interrupt the host processor is at the high level.
	13: CACLR	0	†	This bit is cleared. (CACLR is used to start and then check the status of an instruction cache flush.)
	14: CAEN	0	†	The program cache is disabled.
	15: CAFRZ	0	†	The cache is not frozen.
T0	all	0	0	
T1	all	0	0	
T2	all	0	0	T2 is cleared because the ASM field of ST1_55 is cleared. This is due to the relationship between ASM and T2 when C54CM = 1 (see the description for ASM in section 2.10.2.1).
T3	all	0	0	
TRN0, TRN1	all	0	0	
XAR0	all (AR0H:AR0)	0	†	
XAR1	all (AR1H:AR1)	0	†	
XAR2	all (AR2H:AR2)	0	†	
XAR3	all (AR3H:AR3)	0	†	
XAR4	all (AR4H:AR4)	0	†	
XAR5	all (AR5H:AR5)	0	†	
XAR6	all (AR6H:AR6)	0	†	
XAR7	all (AR7H:AR7)	0	†	
XCDP	all (DPH:DP)	0	0	
XDP	all (DPH:DP)	0	†	
XSP	all (SPH:SP)	0	†	
XSSP	all (SPH:SSP)	0	0	

† Not affected by a software reset

5.5.1 DSP Hardware Reset

When asserted, the DSP reset signal places the DSP into a known state. As part of a hardware reset, all current operations are aborted, the instruction pipeline is emptied, and CPU registers are reset. Then the CPU executes the reset interrupt service routine (see the standard process flow for nonmaskable interrupts in section 5.4.1). When reading the reset interrupt vector, the CPU uses bits 29 and 28 of the 32-bit reset vector location to determine which stack configuration to use (see section 4.2).

Table 5−6 summarizes the effects of a DSP hardware reset on DSP registers. A software reset (see section 5.5.2) performs a subset of these register modifications.

The $\overline{\text{RESET}}$ pin must be asserted for certain number of clock cycles (refer to applicable data manual). If the $\overline{\text{RESET}}$ pin is asserted and deasserted while the DSP is stopped for emulation purposes, the reset is ignored.

Notes:

1) External interrupts must occur at least 3 cycles after the CPU exits reset or they will not be recognized.

2) All interrupts (maskable and nonmaskable) are disabled following a hardware reset. Interrupts remain disabled until the stack pointers are initialized by a software write to each pointer (the SP and SSP registers). After stack initialization, the INTM bit and the IER0 and IER1 registers determine interrupt enabling.

5.5.2 Software Reset

A software reset is the reset operation initiated by the software reset instruction. A software reset only affects IFR0, IFR1, ST0_55, ST1_55, and ST2_55; all other registers are unaffected. The software reset values shown in Table 5−6 are the same as those forced by a DSP hardware reset.

When reading the reset interrupt vector, the CPU uses bits 29 and 28 of the 32-bit reset vector location to determine which stack configuration to use (see section 4.2).

Chapter 6

Addressing Modes

This chapter describes the modes available for addressing the data space (including CPU registers) and the I/O space of the C55x DSPs.

Topic **Page**

6.1 Introduction to the Addressing Modes

The TMS320C55x DSP supports three types of addressing modes that enable flexible access to data memory, to memory-mapped registers, to register bits, and to I/O space:

- ❑ Absolute addressing modes (see section 6.2) enable you to reference a location by supplying all or part of an address as a constant in an instruction.
- ❑ Direct addressing modes (see section 6.3) enable you to reference a location using an address offset.
- ❑ Indirect addressing modes (see section 6.4) enable you to reference a location using a pointer.

Note:

Be aware that certain instructions cannot perform two operations in the same cycle, unless DARAM or two separate blocks of SARAM are used.

Each addressing mode provides one or more types of operands. An instruction that supports an addressing-mode operand has one of the syntax elements described in Table 6–1.

Table 6–1. Syntax Elements That Support the Addressing Modes

Syntax Element(s)	Description
Smem	When an instruction syntax contains Smem, that instruction can access a single word (16 bits) of data from data memory, from I/O space, or from a memory-mapped register. As you write the instruction, replace Smem with a compatible addressing-mode operand.
Lmem	When an instruction syntax contains Lmem, that instruction can access a long word (32 bits) of data from data memory or from memory-mapped registers. As you write the instruction, replace Lmem with a compatible addressing-mode operand.
Xmem and Ymem	When an instruction contains Xmem and Ymem, that instruction can perform two simultaneous 16-bit accesses to data memory. As you write the instruction, replace Xmem and Ymem with compatible operands.
Cmem	When an instruction contains Cmem, that instruction can access a single word (16 bits) of data from data memory. As you write the instruction, replace Cmem with a compatible operand.
Baddr	When an instruction contains Baddr, that instruction can access one or two bits in an accumulator (AC0–AC3), an auxiliary register (AR0–AR7), or a temporary register (T0–T3). Only the register bit test/set/clear/complement instructions support Baddr. As you write one of these instructions, replace Baddr with a compatible operand.

6.2 Absolute Addressing Modes

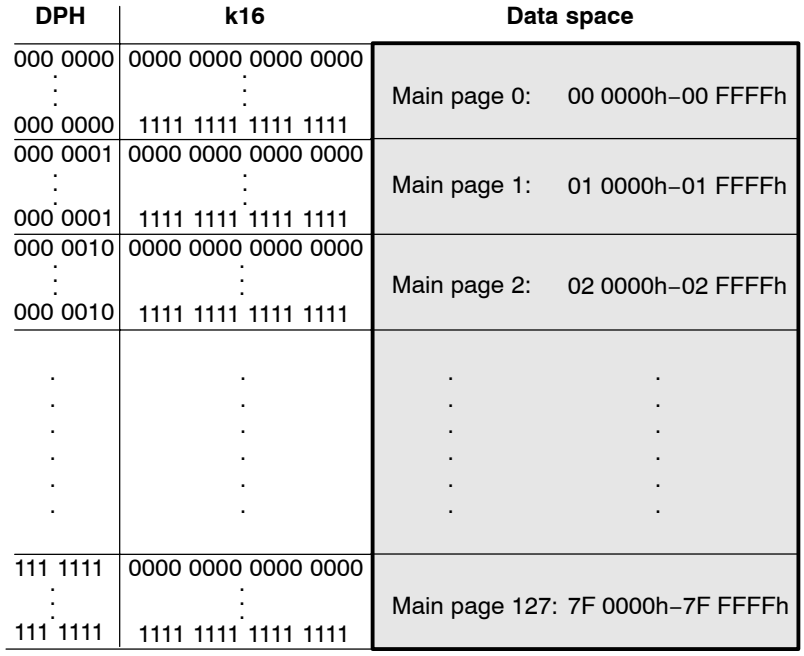
Three absolute addressing modes are available:

Addressing Mode	Description	See ...
k16 absolute	This mode uses the 7-bit register called DPH (high part of the extended data page register) and a 16-bit unsigned constant to form a 23-bit data-space address. This mode can be used to access a memory location or a memory-mapped register.	Page 6-4
k23 absolute	This mode enables you to specify a full address as a 23-bit unsigned constant. This mode can be used to access a memory location or a memory-mapped register.	Page 6-5
I/O absolute	This mode enables you to specify an I/O address as a 16-bit unsigned constant. This mode is for accessing a location in I/O space.	Page 6-6

6.2.1 k16 Absolute Addressing Mode

The k16 absolute addressing mode uses the operand *abs16(#k16), where k16 is a 16-bit unsigned constant. Figure 6−1 shows how DPH (the high part of the extended data page register) and k16 are concatenated to form a 23-bit data-space address. When an instruction uses this addressing mode, the constant is encoded in a 2-byte extension to the instruction. Because of the extension, an instruction using this mode cannot be executed in parallel with another instruction.

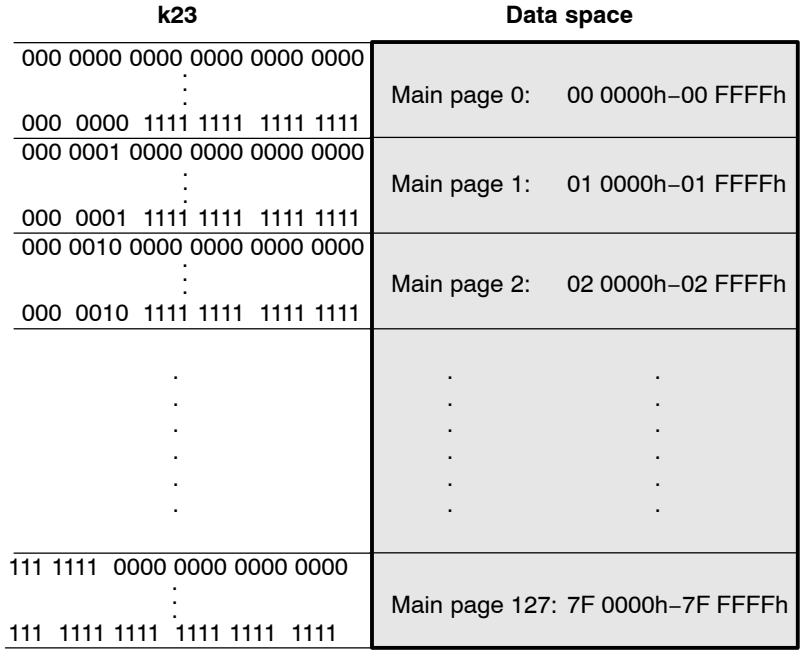
Figure 6−1. k16 Absolute Addressing Mode



6.2.2 k23 Absolute Addressing Mode

The k23 absolute addressing mode uses the operand *(#k23), where k23 is a 23-bit unsigned constant. Figure 6−2 shows how data space is addressed using k23. An instruction using this addressing mode encodes the constant as a 3-byte extension to the instruction (the most significant bit of this 3-byte extension is discarded). Because of the extension, an instruction using this mode cannot be executed in parallel with another instruction.

Figure 6–2. k23 Absolute Addressing Mode

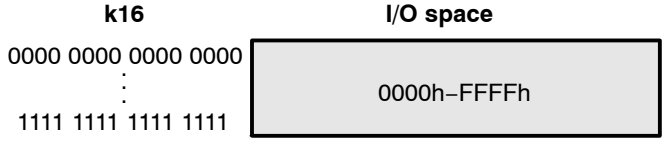


6.2.3 I/O Absolute Addressing Mode

If you use the algebraic instruction set, the I/O absolute addressing mode uses the operand *port(#k16), where k16 is a 16-bit unsigned constant. If you use the mnemonic instruction set, the I/O absolute addressing capability is provided by the port() operand qualifier. Enclose a 16-bit unsigned constant in the parentheses of the port() qualifier: port(#k16) (there is no preceding asterisk, *, in this case).

Figure 6–3 shows how k16 is used to address I/O space. When an instruction uses this addressing mode, the constant is encoded in a 2-byte extension to the instruction. Because of the extension, an instruction using this mode cannot be executed in parallel with another instruction.

Figure 6–3. I/O Absolute Addressing Mode



6.3 Direct Addressing Modes

The following direct addressing modes are available:

Addressing Mode	Description	See ...
DP direct	This mode uses the main data page specified by DPH (the high part of the extended data page register) in conjunction with the data page register (DP). This mode is used to access a memory location or a memory-mapped register.	Page 6-8
SP direct	This mode uses the main data page specified by SPH (the high part of the extended stack pointers) in conjunction with the data stack pointer (SP). Use this mode to access stack values in data memory.	Page 6-10
Register-bit direct	This mode uses an offset to specify a bit address. This mode is used to access one register bit or two adjacent register bits.	Page 6-11
PDP direct	This mode uses the peripheral data page register (PDP) and an offset to specify an I/O address. This mode is used to access a location in I/O space.	Page 6-12

The DP direct and SP direct addressing modes are mutually exclusive. The mode selected depends on the CPL bit in status register ST1_55:

CPL	Addressing Mode Selected
0	DP direct addressing mode
1	SP direct addressing mode

The register-bit and PDP direct addressing modes are independent of the CPL bit.

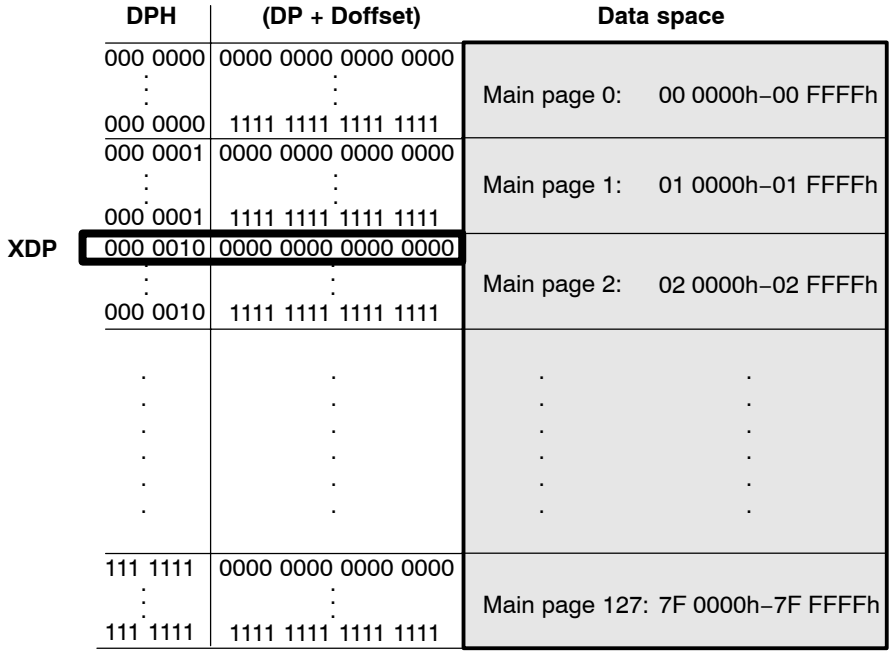
6.3.1 DP Direct Addressing Mode

Figure 6–4 shows the components of a 23-bit address in the DP direct addressing mode. The 7 most significant bits are taken from the register called DPH, and they select one of the 128 main data pages (0 through 127). The 16 least significant bits are the sum of two values:

- The value in the data page register (DP). DP identifies the start address of a 128-word local data page within the main data page. This start address can be any address within the selected main data page.
- A 7-bit offset (Doffset) calculated by the assembler. The calculation depends on whether you are accessing data memory or a memory-mapped register (using the mmap() qualifier). For details on the calculation, see section 6.3.1.1.

The concatenation of DPH and DP is called the extended data page register (XDP). You can load DPH and DP individually, or you can use an instruction that loads XDP.

Figure 6–4. DP Direct Addressing Mode



6.3.1.1 How the Assembler Calculates Doffset for the DP Direct Addressing Mode

Table 6–2 explains how the assembler calculates the value Doffset for the two types of DP direct accesses. Examples follow the table.

Table 6–2. Calculation of Doffset for the DP Direct Addressing Mode

Access Made To...	Doffset Calculation	Description
Data memory	Doffset = (Daddr – .dp) & 7Fh	Daddr is the 16-bit local address for the read or write operation; .dp is a value you assign with the .dp assembler directive (.dp generally matches DP); the symbol & indicates a bitwise AND operation.
A memory-mapped register, using the mmap() qualifier	Doffset = Daddr & 7Fh	Daddr is the 16-bit local address for the read or write operation; the symbol & indicates a bitwise AND operation. The .dp value is not used. The mmap() qualifier forces the CPU to behave as if the data page is 0.

Note:

A *local address* is an address within a main data page. It is represented by the 16 LSBs of a 23-bit data space address. For example, local address 0005h exists on every main data page.

The following code example uses DP direct addressing to access data memory:

```
AMOV #03FFF0h, XDP      ; Main data page is 03. For run-time, DP is FFF0h.
.dp #0FFF0h             ; For assembly time, .dp is FFF0h.
MOV @0FFF4h, T2         ; Load T2 with the value at local address FFF4h.
```

The assembler calculates Doffset:

Doffset = (Daddr – .dp) & 7Fh = (FFF4h – FFF0h) & 7Fh = 04h

Doffset is encoded in the instruction MOV @0FFF4h, T2. At run time, the 23-bit data-space address is generated:

23-bit address = DPH:(DP + Doffset) = 03:(FFF0h + 0004h) = 03 FFF4h

The following code example uses DP direct addressing to access a memory-mapped register (MMR):

```
MOV mmap(@AR0), T2      ; Load T2 with the value in AR0.
                        ; mmap() qualifier indicates access to MMR.
                        ; AR0 is mapped to address 000010h in data space
```

The assembler calculates Doffset:

Doffset = Daddr & 7Fh = 0010h & 7Fh = 10h

Doffset is encoded in the instruction MOV mmap(@AR0), T2. At run time, the 23-bit data-space address is generated (recall that for register keywords the CPU behaves as if DPH = DP = 0):

23-bit address = DPH:(DP + Doffset) = 00:(0000h + 0010h) = 00 0010h

Note:

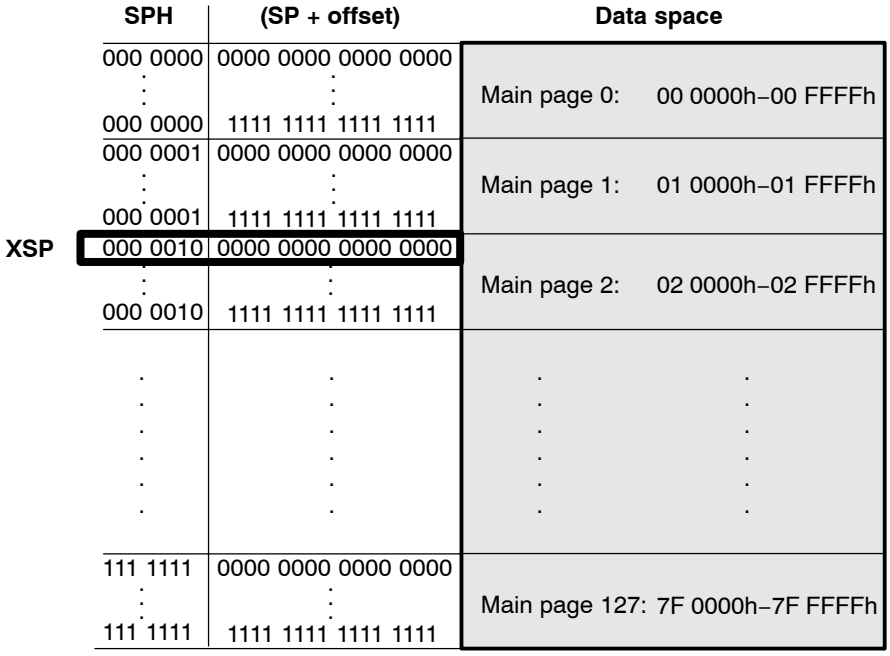
If you use mnemonic instructions, mmap() encloses the qualified operand. If you use algebraic instructions, mmap() is an instruction qualifier that is placed in parallel with the instruction that performs a memory-mapped register access.

6.3.2 SP Direct Addressing Mode

When an instruction uses the SP direct addressing mode, 23-bit addresses are formed as shown in Figure 6–5. The 7 most significant bits are supplied by the register called SPH. The 16 least significant bits are the sum of the SP value and a 7-bit offset that you specify in the instruction. The offset can be a value from 0 to 127. The concatenation of SPH and SP is called the extended data stack pointer (XSP). You can load SPH and SP individually, or you can use an instruction that loads XSP.

On the first main data page, addresses 00 0000h–00 005Fh are reserved for the memory-mapped registers. If any of your data stack is in main data page 0, make sure it uses only addresses 00 0060h–00 FFFFh on that page.

Figure 6–5. SP Direct Addressing Mode

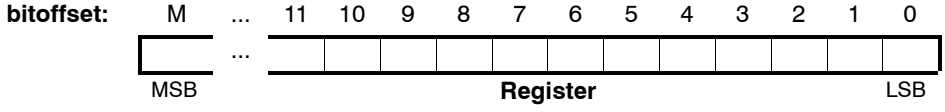


6.3.3 Register-Bit Direct Addressing Mode

In the register-bit direct addressing mode, the offset you supply in the operand, @bitoffset, is an offset from the least significant bit (LSB) of the register (see Figure 6–6). For example, if bitoffset is 0, you are addressing the least significant bit (LSB) of a register. If bitoffset is 3, you are addressing bit 3 of the register.

Only the register bit test/set/clear/complement instructions support this mode. These instructions enable you to access bits in the following registers only: the accumulators (AC0–AC3), the auxiliary registers (AR0–AR7), and the temporary registers (T0–T3).

Figure 6–6. Register-Bit Direct Addressing Mode



Note: Bit address M is 39 or 15, depending on the size of the register.

6.3.4 PDP Direct Addressing Mode

When an instruction uses the PDP direct addressing mode, 16-bit I/O addresses are formed as shown in Figure 6–7. The 9-bit peripheral data page register (PDP) selects one of the 512 peripheral data pages (0 through 511). Each page has 128 words (0 to 127). You select a particular word by specifying a 7-bit offset (Poffset) in the instruction. For example, to access the first word on a page, use an offset of 0.

Figure 6–7. PDP Direct Addressing Mode

PDP	Poffset	I/O space (64K)
0000 0000 0 ⋮ 0000 0000 0	000 0000 ⋮ 111 1111	Peripheral page 0: 0000h–007Fh
0000 0000 1 ⋮ 0000 0000 1	000 0000 ⋮ 111 1111	Peripheral page 1: 0080h–00FFh
0000 0001 0 ⋮ 0000 0001 0	000 0000 ⋮ 111 1111	Peripheral page 2: 0100h–017Fh
. . . . . .	. . . . . .	. . . . . .
1111 1111 1 ⋮ 1111 1111 1	000 0000 ⋮ 111 1111	Peripheral page 511: FF80h–FFFFh

6.4 Indirect Addressing Modes

The CPU supports the following indirect addressing modes. You may use these modes for linear addressing or circular addressing.

Addressing Mode	Description	See Section ...
AR indirect	This mode uses one of eight auxiliary registers (AR0–AR7) to point to data. The way the CPU uses the auxiliary register to generate an address depends on whether you are accessing data space (memory or memory-mapped registers), individual register bits, or I/O space.	6.4.1
Dual AR indirect	This mode uses the same address generation process as the AR indirect addressing mode. This mode is used with an instruction that accesses two or more data-memory locations at the same time.	6.4.2
CDP indirect	This mode uses the coefficient data pointer (CDP) to point to data. The way the CPU uses CDP to generate an address depends on whether you are accessing data space (memory or memory-mapped registers), individual register bits, or I/O space.	6.4.3
Coefficient indirect	This mode uses the same address-generation process as the CDP indirect addressing mode. This mode is available to support instructions that can access a coefficient in data memory at the same time they access two other data-memory values using the dual AR indirect addressing mode.	6.4.4

6.4.1 AR Indirect Addressing Mode

This mode uses an auxiliary register ARn (n = 0, 1, 2, 3, 4, 5, 6, or 7) to point to data. As shown in Table 6–3, the way the CPU uses ARn to generate an address depends on the access type.

Table 6–3. Use of an Auxiliary Register (ARn) in the AR Indirect Addressing Mode

For An Access To ...	ARn Contains ...
Data space (memory or registers)	The 16 least significant bits (LSBs) of a 23-bit address. The 7 most significant bits (MSBs) are supplied by ARnH, which is the high part of extended auxiliary register XARn. See section 6.4.1.1.
A register bit (or bit pair)	A bit number. See section 6.4.1.2.
I/O space	A 16-bit I/O address. See section 6.4.1.3.

6.4.1.1 AR Indirect Accesses of Data Space

Figure 6–8 shows how the CPU generates data-space addresses for the AR indirect addressing mode. (Note that both data memory and memory-mapped registers are mapped to data space.) For a given access, auxiliary register n (n = 0, 1, 2, 3, 4, 5, 6, or 7) provides the 16 least significant bits, and an associated register, ARnH, provides the 7 most significant bits. The concatenation of ARnH and ARn is called extended auxiliary register n (XARn). For accesses to data space, use an instruction that loads XARn; ARn can be individually loaded, but ARnH cannot.

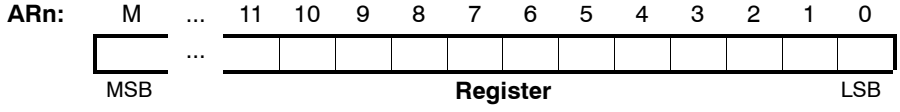
Figure 6–8. Accessing Data Space With the AR Indirect Addressing Mode

6.4.1.2 AR Indirect Accesses of Register Bits

When the AR indirect addressing mode is used to access a register bit, the selected 16-bit auxiliary register, ARn, contains a bit number (see Figure 6–9). For example, if AR2 contains 0, AR2 points to bit 0, the least significant bit (LSB) of the register.

Only the register bit test/set/clear/complement instructions support AR indirect accesses to register bits. These instructions enable you to access bits in the following registers only: the accumulators (AC0–AC3), the auxiliary registers (AR0–AR7), and the temporary registers (T0–T3).

Figure 6–9. Accessing Register Bit(s) With the AR Indirect Addressing Mode

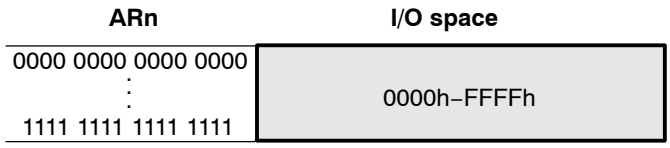


Note: Bit address M is 39 or 15, depending on the size of the register.

6.4.1.3 AR Indirect Accesses of I/O Space

Words in I/O space are accessed at 16-bit addresses. When the AR indirect addressing mode is used to access I/O space, the selected 16-bit auxiliary register, ARn, contains the complete I/O address.

Figure 6−10. Accessing I/O Space With the AR Indirect Addressing Mode



6.4.1.4 AR Indirect Operands

The types of addressing-mode operands available for this mode depend on the ARMS bit of status register ST2_55:

ARMS	DSP Mode or Control Mode?
0	DSP mode. The CPU can use the list of DSP mode operands (Table 6−4), which provide efficient execution of DSP-intensive applications.
1	Control mode. The CPU can use the list of control mode operands (Table 6−5), which enable optimized code size for control system applications.

Table 6−4 introduces the DSP mode operands available for the AR indirect addressing mode. Table 6−5 introduces the control mode operands. Table 6−6 summarizes all of the AR indirect operands based on whether they modify the auxiliary register and whether the modification occurs before or after the address for the instruction is generated. When using the tables, keep in mind that:

- Both pointer modification and address generation are linear or circular according to the pointer configuration in status register ST2_55. The content of the appropriate 16-bit buffer start address register (BSA01, BSA23, BSA45, or BSA67) is added only if circular addressing is activated for the chosen pointer.

- Increments and decrements are made to the 16-bit pointer only. You cannot address data across main data pages without changing the value in the extension register (ARnH). To change ARnH, you must write to the full 23−bit register, XARn.

Note:

Although an increment past FFFFh or a decrement past 0000h causes the pointer value to wrap around, do not make use of this behavior; it is not supported. Also, during circular addressing, the BSAxx addition must not increment the address beyond FFFFh.

Table 6−4. DSP Mode (ARMS = 0) Operands for the AR Indirect Addressing Mode

Operand	Pointer Modification	Supported Access Types
*ARn	ARn is not modified.	Data-memory (Smem, Lmem) Memory-mapped register (Smem, Lmem) Register bit (Baddr) I/O-space (Smem)
*ARn+	ARn is incremented after the address is generated.†‡	Data-memory (Smem, Lmem) Memory-mapped register (Smem, Lmem) Register bit (Baddr) I/O-space (Smem)
*ARn−	ARn is decremented after the address is generated.§¶	Data-memory (Smem, Lmem) Memory-mapped register (Smem, Lmem) Register bit (Baddr) I/O-space (Smem)

† If 16-bit/1-bit operation: ARn = ARn + 1; 1-bit operation: register bit access that reads or modifies a single bit in a register
‡ If 32-bit/2-bit operation: ARn = ARn + 2; 2-bit operation: register bit access that reads or modifies a register bit-pair
§ If 16-bit/1-bit operation: ARn = ARn − 1; 1-bit operation: register bit access that reads or modifies a single bit in a register
¶ If 32-bit/2-bit operation: ARn = ARn − 2; 2-bit operation: register bit access that reads or modifies a register bit-pair

Table 6–4. DSP Mode (ARMS = 0) Operands for the AR Indirect Addressing Mode (Continued)

Operand	Pointer Modification	Supported Access Types
*+ARn	ARn is incremented before the address is generated. †‡	Data-memory (Smem, Lmem) Memory-mapped register (Smem, Lmem) Register bit (Baddr) I/O-space (Smem)
*–ARn	ARn is decremented before the address is generated.§¶	Data-memory (Smem, Lmem) Memory-mapped register (Smem, Lmem) Register bit (Baddr) I/O-space (Smem)
*(ARn + T0/AR0)	The 16-bit signed constant in T0 or AR0 is added to ARn after the address is generated: If C54CM = 0: ARn = ARn + T0 If C54CM = 1: ARn = ARn + AR0	Data-memory (Smem, Lmem) Memory-mapped register (Smem, Lmem) Register bit (Baddr) I/O-space (Smem)
*(ARn – T0/AR0)	The 16-bit signed constant in T0 or AR0 is subtracted from ARn after the address is generated: If C54CM = 0: ARn = ARn – T0 If C54CM = 1: ARn = ARn – AR0	Data-memory (Smem, Lmem) Memory-mapped register (Smem, Lmem) Register bit (Baddr) I/O-space (Smem)

† If 16-bit/1-bit operation: ARn = ARn + 1; 1-bit operation: register bit access that reads or modifies a single bit in a register
‡ If 32-bit/2-bit operation: ARn = ARn + 2; 2-bit operation: register bit access that reads or modifies a register bit-pair
§ If 16-bit/1-bit operation: ARn = ARn – 1; 1-bit operation: register bit access that reads or modifies a single bit in a register
¶ If 32-bit/2-bit operation: ARn = ARn – 2; 2-bit operation: register bit access that reads or modifies a register bit-pair

Table 6–4. DSP Mode (ARMS = 0) Operands for the AR Indirect Addressing Mode (Continued)

Operand	Pointer Modification	Supported Access Types
*ARn(T0/AR0)	ARn is not modified. ARn is used as a base pointer. The 16-bit signed constant in T0 or AR0 is used as an offset from that base pointer: If C54CM = 0, T0 is used If C54CM = 1, AR0 is used	Data-memory (Smem, Lmem) Memory-mapped register (Smem, Lmem) Register bit (Baddr) I/O-space (Smem)
*(ARn + T0B/AR0B)	The 16-bit signed constant in T0 or AR0 is added to ARn after the address is generated: If C54CM = 0: ARn = ARn + T0 If C54CM = 1: ARn = ARn + AR0 (Either addition is done with reverse carry propagation to create bit-reverse addressing) Note: When this bit-reverse operand is used, ARn cannot be used as a circular pointer. If ARn is configured in ST2_55 for circular addressing, the corresponding buffer start address register value (BSAxx) is added to ARn, but ARn is not modified so as to remain inside a circular buffer.	Data-memory (Smem, Lmem) Memory-mapped register (Smem, Lmem) Register bit (Baddr) I/O-space (Smem)
*(ARn – T0B/AR0B)	The 16-bit signed constant in T0 or AR0 is subtracted from ARn after the address is generated: If C54CM = 0: ARn = ARn – T0 If C54CM = 1: ARn = ARn – AR0 (Either subtraction is done with reverse carry propagation.) Note: When this bit-reverse operand is used, ARn cannot be used as a circular pointer. If ARn is configured in ST2_55 for circular addressing, the corresponding buffer start address register value (BSAxx) is added to ARn, but ARn is not modified so as to remain inside a circular buffer.	Data-memory (Smem, Lmem) Memory-mapped register (Smem, Lmem) Register bit (Baddr) I/O-space (Smem)

† If 16-bit/1-bit operation: ARn = ARn + 1; 1-bit operation: register bit access that reads or modifies a single bit in a register
‡ If 32-bit/2-bit operation: ARn = ARn + 2; 2-bit operation: register bit access that reads or modifies a register bit-pair
§ If 16-bit/1-bit operation: ARn = ARn – 1; 1-bit operation: register bit access that reads or modifies a single bit in a register
¶ If 32-bit/2-bit operation: ARn = ARn – 2; 2-bit operation: register bit access that reads or modifies a register bit-pair

Table 6–4. DSP Mode (ARMS = 0) Operands for the AR Indirect Addressing Mode (Continued)

Operand	Pointer Modification	Supported Access Types
*(ARn + T1)	The 16-bit signed constant in T1 is added to ARn after the address is generated: ARn = ARn + T1	Data-memory (Smem, Lmem) Memory-mapped register (Smem, Lmem) Register bit (Baddr) I/O-space (Smem)
*(ARn – T1)	The 16-bit signed constant in T1 is subtracted from ARn after the address is generated: ARn = ARn – T1	Data-memory (Smem, Lmem) Memory-mapped register (Smem, Lmem) Register bit (Baddr) I/O-space (Smem)
*ARn(T1)	ARn is not modified. ARn is used as a base pointer. The 16-bit signed constant in T1 is used as an offset from that base pointer.	Data-memory (Smem, Lmem) Memory-mapped register (Smem, Lmem) Register bit (Baddr) I/O-space (Smem)

† If 16-bit/1-bit operation: ARn = ARn + 1; 1-bit operation: register bit access that reads or modifies a single bit in a register
‡ If 32-bit/2-bit operation: ARn = ARn + 2; 2-bit operation: register bit access that reads or modifies a register bit-pair
§ If 16-bit/1-bit operation: ARn = ARn – 1; 1-bit operation: register bit access that reads or modifies a single bit in a register
¶ If 32-bit/2-bit operation: ARn = ARn – 2; 2-bit operation: register bit access that reads or modifies a register bit-pair

Table 6–4. DSP Mode (ARMS = 0) Operands for the AR Indirect Addressing Mode (Continued)

Operand	Pointer Modification	Supported Access Types
*ARn(#K16)	ARn is not modified. ARn is used as a base pointer. The 16-bit signed constant (K16) is used as an offset from that base pointer. Note: When an instruction uses this operand, the constant is encoded in a 2-byte extension to the instruction. Because of the extension, an instruction using this operand cannot be executed in parallel with another instruction.	Data-memory (Smem, Lmem) Memory-mapped register (Smem, Lmem) Register bit (Baddr)
*+ARn(#K16)	The 16-bit signed constant (K16) is added to ARn before the address is generated. Note: When an instruction uses this operand, the constant is encoded in a 2-byte extension to the instruction. Because of the extension, an instruction using this operand cannot be executed in parallel with another instruction.	Data-memory (Smem, Lmem) Memory-mapped register (Smem, Lmem) Register bit (Baddr)

† If 16-bit/1-bit operation: ARn = ARn + 1; 1-bit operation: register bit access that reads or modifies a single bit in a register
‡ If 32-bit/2-bit operation: ARn = ARn + 2; 2-bit operation: register bit access that reads or modifies a register bit-pair
§ If 16-bit/1-bit operation: ARn = ARn – 1; 1-bit operation: register bit access that reads or modifies a single bit in a register
¶ If 32-bit/2-bit operation: ARn = ARn – 2; 2-bit operation: register bit access that reads or modifies a register bit-pair

Table 6–5. Control Mode (ARMS = 1) Operands for the AR Indirect Addressing Mode

Operand	Pointer Modification	Supported Access Types
*ARn	ARn is not modified.	Data-memory (Smem, Lmem) Memory-mapped register (Smem, Lmem) Register bit (Baddr) I/O-space (Smem)
*ARn+	ARn is incremented after the address is generated.†‡	Data-memory (Smem, Lmem) Memory-mapped register (Smem, Lmem) Register bit (Baddr) I/O-space (Smem)
*ARn–	ARn is decremented after the address is generated.§¶ If 16-bit/1-bit operation: ARn = ARn – 1 If 32-bit/2-bit operation: ARn = ARn – 2	Data-memory (Smem, Lmem) Memory-mapped register Smem, Lmem) Register bit (Baddr) I/O-space (Smem)
*(ARn + T0/AR0)	The 16-bit signed constant in T0 or AR0 is added to ARn after the address is generated: If C54CM = 0: ARn = ARn + T0 If C54CM = 1: ARn = ARn + AR0	Data-memory (Smem, Lmem) Memory-mapped register (Smem, Lmem) Register bit (Baddr) I/O-space (Smem)

† If 16-bit/1-bit operation: ARn = ARn +1; If 32-bit/2-bit operation: ARn = ARn +2

‡ 1-bit operation: register bit access that reads or modifies a single bit in a register; 2-bit operation: register bit access that reads or modifies a register bit-pair

§ If 16-bit/1-bit operation: ARn = ARn – 1; 1-bit operation: register bit access that reads or modifies a single bit in a register

¶ If 32-bit/2-bit operation: ARn = ARn – 2; 2-bit operation: register bit access that reads or modifies a register bit-pair

Table 6–5. Control Mode (ARMS = 1) Operands for the AR Indirect Addressing Mode (Continued)

Operand	Pointer Modification	Supported Access Types
*(ARn – T0/AR0)	The 16-bit signed constant in T0 or AR0 is subtracted from ARn after the address is generated: If C54CM = 0: ARn = ARn – T0 If C54CM = 1: ARn = ARn – AR0	Data-memory (Smem, Lmem) Memory-mapped register (Smem, Lmem) Register bit (Baddr) I/O-space (Smem)
*ARn(T0/AR0)	ARn is not modified. ARn is used as a base pointer. The 16-bit signed constant in T0 or AR0 is used as an offset from that base pointer: If C54CM = 0, T0 is used If C54CM = 1, AR0 is used	Data-memory (Smem, Lmem) Memory-mapped register (Smem, Lmem) Register bit (Baddr) I/O-space (Smem)
*ARn(#K16)	ARn is not modified. ARn is used as a base pointer. The 16-bit signed constant (K16) is used as an offset from that base pointer. Note: When an instruction uses this operand, the constant is encoded in a 2-byte extension to the instruction. Because of the extension, an instruction using this operand cannot be executed in parallel with another instruction.	Data-memory (Smem, Lmem) Memory-mapped register (Smem, Lmem) Register bit (Baddr)

† If 16-bit/1-bit operation: ARn = ARn +1; If 32-bit/2-bit operation: ARn = ARn +2

‡ 1-bit operation: register bit access that reads or modifies a single bit in a register; 2-bit operation: register bit access that reads or modifies a register bit-pair

§ If 16-bit/1-bit operation: ARn = ARn – 1; 1-bit operation: register bit access that reads or modifies a single bit in a register

¶ If 32-bit/2-bit operation: ARn = ARn – 2; 2-bit operation: register bit access that reads or modifies a register bit-pair

Table 6–5. Control Mode (ARMS = 1) Operands for the AR Indirect Addressing Mode (Continued)

Operand	Pointer Modification	Supported Access Types
*+ARn(#K16)	The 16-bit signed constant (K16) is added to ARn before the address is generated: ARn = ARn + K16 Note: When an instruction uses this operand, the constant is encoded in a 2-byte extension to the instruction. Because of the extension, an instruction using this operand cannot be executed in parallel with another instruction.	Data-memory (Smem, Lmem) Memory-mapped register (Smem, Lmem) Register bit (Baddr)
*ARn(short(#k3))	ARn is not modified. ARn is used as a base pointer. The 3-bit unsigned constant (k3) is used as an offset from that base pointer. k3 is in the range 1 to 7.	Data-memory (Smem, Lmem) Memory-mapped register (Smem, Lmem) Register bit (Baddr) I/O-space (Smem)

† If 16-bit/1-bit operation: ARn = ARn +1; If 32-bit/2-bit operation: ARn = ARn +2

‡ 1-bit operation: register bit access that reads or modifies a single bit in a register; 2-bit operation: register bit access that reads or modifies a register bit-pair

§ If 16-bit/1-bit operation: ARn = ARn – 1; 1-bit operation: register bit access that reads or modifies a single bit in a register

¶ If 32-bit/2-bit operation: ARn = ARn – 2; 2-bit operation: register bit access that reads or modifies a register bit-pair

Table 6–6. Summary of Indirect Operands

Not Modified	Post Modified	Premodified
AR Indirect, DSP Mode (ARMS = 0)		
*ARn	*ARn+	*+ARn
*ARn(T0/AR0)	*ARn–	*–ARn
*ARn(T1)	*(ARn + T0/AR0)	*+ARn(#K16)
*ARn(#K16)	*(ARn – T0/AR0)	
	*(ARn + T0B/AR0B)	
	*(ARn – T0B/AR0B)	
	*(ARn + T1)	
	*(ARn – T1)	
AR Indirect, Control Mode (ARMS = 1)		
*ARn	*ARn+	*+ARn(#K16)
*ARn(T0/AR0)	*ARn–	
*ARn(#K16)	*(ARn + T0/AR0)	
*ARn(short(#k3))	*(ARn – T0/AR0)	
	*(ARn + T0B/AR0B)	
	*(ARn – T0B/AR0B)	
	*(ARn + T1)	
	*(ARn – T1)	

6.4.2 Dual AR Indirect Addressing Mode

The dual AR indirect addressing mode enables you to make two data-memory accesses through the eight auxiliary registers, AR0–AR7. As with single AR indirect accesses to data space (see section 6.4.1.1), the CPU uses an extended auxiliary register to create each 23-bit address. You can use linear addressing or circular addressing for each of the two accesses.

You may use the dual AR indirect addressing mode for:

- Executing an instruction that makes two 16-bit data-memory accesses. In this case, the two data-memory operands are designated in the instruction syntax as Xmem and Ymem. For example:

  ADD Xmem, Ymem, ACx

- Executing two instructions in parallel. In this case, both instructions must each access a single memory value, designated in the instruction syntaxes as Smem or Lmem. For example:

  MOV Smem, dst
  || AND Smem, src, dst

  The operand of the first instruction is treated as an Xmem operand, and the operand of the second instruction is treated as a Ymem operand.

The available dual AR indirect operands are a subset of the AR indirect operands. The ARMS status bit does not affect the set of dual AR indirect operands available.

Note:

The assembler rejects code in which dual operands use the same auxiliary register with two different auxiliary register modifications. You can use the same ARn for both operands only if one of the operands does not modify ARn.

6.4.2.1 Dual AR Indirect Operands

Table 6–7 introduces the operands available for the dual AR indirect addressing mode. Note that:

- Both pointer modification and address generation are linear or circular according to the pointer configuration in status register ST2_55. The content of the appropriate 16-bit buffer start address register (BSA01, BSA23, BSA45, or BSA67) is added only if circular addressing is activated for the chosen pointer.

- Increments and decrements are made to the 16-bit pointer only. You cannot address data across main data pages without changing the value in the extension register (ARnH). To change ARnH, you must write to the full 23–bit register, XARn.

Note:

Although an increment past FFFFh or a decrement past 0000h causes the pointer value to wrap around, do not make use of this behavior; it is not supported. Also, during circular addressing, the BSAxx addition must not increment the address beyond FFFFh.

Table 6–7. Dual AR Indirect Operands

Operand	Pointer Modification	Supported Access Type
*ARn	ARn is not modified.	Data-memory (Smem, Lmem, Xmem, Ymem)
*ARn+	ARn is incremented after the address is generated.†‡	Data-memory (Smem, Lmem, Xmem, Ymem)
*ARn–	ARn is decremented after the address is generated.§¶	Data-memory (Smem, Lmem, Xmem, Ymem)
*(ARn + T0/AR0)	The 16-bit signed constant in T0 or AR0 is added to ARn after the address is generated: If C54CM = 0: ARn = ARn + T0 If C54CM = 1: ARn = ARn + AR0	Data-memory (Smem, Lmem, Xmem, Ymem)
*(ARn – T0/AR0)	The 16-bit signed constant in T0 or AR0 is subtracted from ARn after the address is generated: If C54CM = 0: ARn = ARn – T0 If C54CM = 1: ARn = ARn – AR0	Data-memory (Smem, Lmem, Xmem, Ymem)
*ARn(T0/AR0)	ARn is not modified. ARn is used as a base pointer. The 16-bit signed constant in T0 or AR0 is used as an offset from that base pointer: If C54CM = 0, T0 is used If C54CM = 1, AR0 is used	Data-memory (Smem, Lmem, Xmem, Ymem)
*(ARn + T1)	The 16-bit signed constant in T1 is added to ARn after the address is generated: ARn = ARn + T1	Data-memory (Smem, Lmem, Xmem, Ymem)
*(ARn – T1)	The 16-bit signed constant in T1 is subtracted from ARn after the address is generated: ARn = ARn – T1	Data-memory (Smem, Lmem, Xmem, Ymem)

† If 16-bit operation: ARn = ARn + 1
‡ If 32-bit operation: ARn = ARn + 2
§ If 16-bit operation: ARn = ARn – 1
¶ If 32-bit operation: ARn = ARn – 2

6.4.3 CDP Indirect Addressing Mode

This mode uses the coefficient data pointer (CDP) to point to data. As shown in Table 6–8, the way the CPU uses CDP to generate an address depends on the access type.

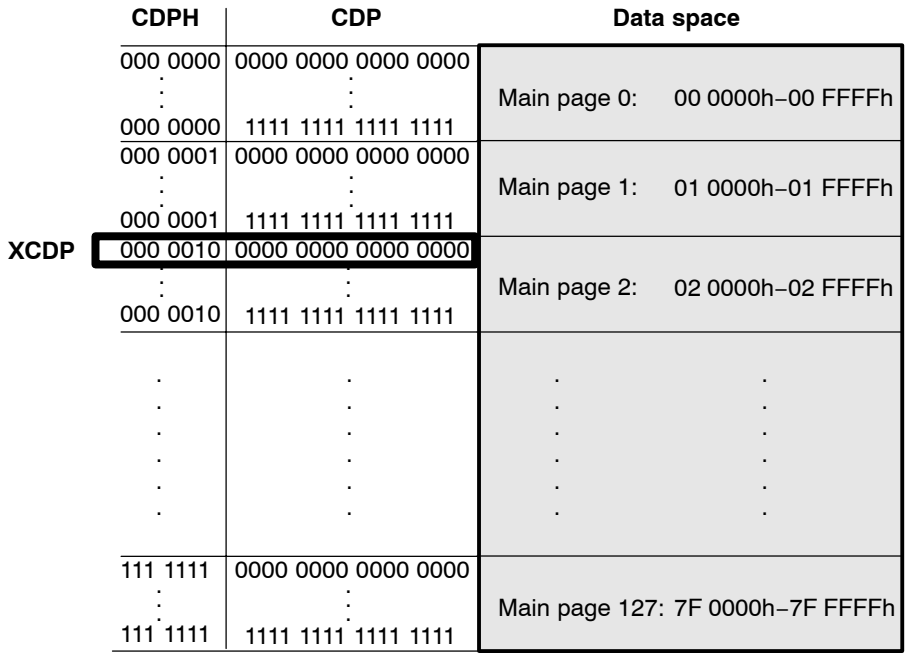
Table 6–8. Use of the Coefficient Data Pointer (CDP) in the CDP Indirect Addressing Mode

For An Access To ...	CDP Contains ...
Data space (memory or registers)	The 16 least significant bits (LSBs) of a 23-bit address. The 7 most significant bits (MSBs) are supplied by CDPH, the high part of the extended coefficient data pointer (XCDP). See section 6.4.3.1.
A register bit (or bit pair)	A bit number. See section 6.4.3.2.
I/O space	A 16-bit I/O address. See section 6.4.3.3.

6.4.3.1 CDP Indirect Accesses of Data Space

Figure 6–11 shows how the CPU generates data-space addresses for the CDP indirect addressing mode. (Note that both data memory and memory-mapped registers are mapped to data space.) CDPH provides the 7 most significant bits, and the coefficient data pointer (CDP) provides the 16 least significant bits. The concatenation of CDPH and CDP is called the extended coefficient data pointer (XCDP).

Figure 6−11. Accessing Data Space With the CDP Indirect Addressing Mode

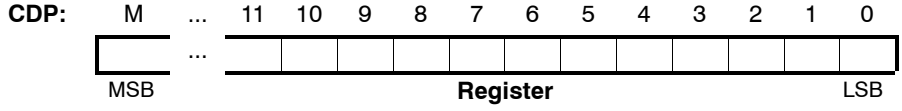


6.4.3.2 CDP Indirect Accesses of Register Bits

When the CDP indirect addressing mode is used to access a register bit, CDP contains the bit number. For example, if CDP contains 0, it points to bit 0, the least significant bit (LSB) of the register.

Only the register bit test/set/clear/complement instructions support CDP indirect accesses to register bits. These instructions enable you to access bits in the following registers only: the accumulators (AC0–AC3), the auxiliary registers (AR0–AR7), and the temporary registers (T0–T3).

Figure 6−12. Accessing Register Bits With the CDP Indirect Addressing Mode

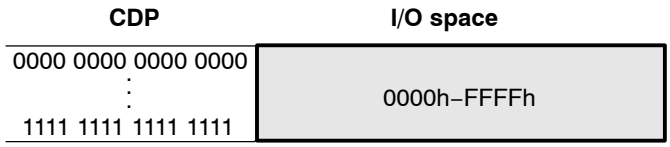


Note: Bit address M is 39 or 15, depending on the size of the register.

6.4.3.3 CDP Indirect Accesses of I/O Space

Words in I/O space are accessed at 16-bit addresses. When the CDP indirect addressing mode is used to access I/O space, the 16-bit CDP contains the complete I/O address.

Figure 6−13. Accessing I/O Space With the CDP Indirect Addressing Mode



6.4.3.4 CDP Indirect Operands

Table 6−9 introduces the operands available for the CDP indirect addressing mode. Note that:

- Both pointer modification and address generation are linear or circular according to the pointer configuration in status register ST2_55. The content of the 16-bit buffer start address register BSAC is added only if circular addressing is activated for CDP.
- Increments and decrements are made to the 16-bit pointer only. You cannot address data across main data pages without changing the value in the extension register (CDPH).

Note:

Although an increment past FFFFh or a decrement past 0000h causes the pointer value to wrap around, do not make use of this behavior; it is not supported. Also, during circular addressing, the BSAC addition must not increment the address beyond FFFFh.

Table 6–9. CDP Indirect Operands

Operand	Pointer Modification	Supported Access Types
*CDP	CDP is not modified.	Data-memory (Smem, Lmem) Memory-mapped register (Smem, Lmem) Register-bit (Baddr) I/O-space (Smem)
*CDP+	CDP is incremented after the address is generated.†‡	Data-memory (Smem, Lmem) Memory-mapped register (Smem, Lmem) Register-bit (Baddr) I/O-space (Smem)
*CDP–	CDP is decremented after the address is generated.§¶	Data-memory (Smem, Lmem) Memory-mapped register (Smem, Lmem) Register-bit (Baddr) I/O-space (Smem)
*CDP(#K16)	CDP is not modified. CDP is used as a base pointer. The 16-bit signed constant (K16) is used as an offset from that base pointer. Note: When an instruction uses this operand, the constant is encoded in a 2-byte extension to the instruction. Because of the extension, an instruction using this operand cannot be executed in parallel with another instruction.	Data-memory (Smem, Lmem) Memory-mapped register (Smem, Lmem) Register-bit (Baddr)

† If 16-bit/1-bit operation: CDP = CDP + 1; 1-bit operation: register bit access that reads or modifies a single bit in a register
‡ If 32-bit/2-bit operation: CDP = CDP + 2; 2-bit operation: register bit access that reads or modifies a register bit pair
§ If 16-bit/1-bit operation: CDP = CDP – 1; 1-bit operation: register bit access that reads or modifies a single bit in a register
¶ If 32-bit/2-bit operation: CDP = CDP – 2; 2-bit operation: register bit access that reads or modifies a register bit pair

Table 6–9. CDP Indirect Operands (Continued)

Operand	Pointer Modification	Supported Access Types
*+CDP(#K16)	The 16-bit signed constant (K16) is added to CDP before the address is generated: CDP = CDP + K16 Note: When an instruction uses this operand, the constant is encoded in a 2-byte extension to the instruction. Because of the extension, an instruction using this operand cannot be executed in parallel with another instruction.	Data-memory (Smem, Lmem) Memory-mapped register (Smem, Lmem) Register-bit (Baddr)

† If 16-bit/1-bit operation: CDP = CDP + 1; 1-bit operation: register bit access that reads or modifies a single bit in a register
‡ If 32-bit/2-bit operation: CDP = CDP + 2; 2-bit operation: register bit access that reads or modifies a register bit pair
§ If 16-bit/1-bit operation: CDP = CDP – 1; 1-bit operation: register bit access that reads or modifies a single bit in a register
¶ If 32-bit/2-bit operation: CDP = CDP – 2; 2-bit operation: register bit access that reads or modifies a register bit pair

6.4.4 Coefficient Indirect Addressing Mode

This mode uses the same address-generation process as the CDP indirect addressing mode for data-space accesses. The coefficient indirect addressing mode is supported by select memory move/initialization syntaxes and by the following arithmetical instructions:

- ❑ Finite impulse response filter
- ❑ Multiply
- ❑ Multiply and accumulate
- ❑ Multiply and subtract
- ❑ Dual multiply [and accumulate/subtract]

Instructions using the coefficient indirect addressing mode to access data are mainly instructions performing operations with three memory operands per cycle. Two of these operands (Xmem and Ymem) are accessed with the dual AR indirect addressing mode. The third operand (Cmem) is accessed with the coefficient indirect addressing mode. The Cmem operand is carried on the BB bus of the CPU (see section 1.6, *Address Buses and Data Buses*).

Consider the following instruction syntax. In one cycle, two multiplications can be performed in parallel. One memory operand (Cmem) is common to both multiplications, while dual AR indirect operands (Xmem and Ymem) are used for the other values in the multiplication.

MPY Xmem, Cmem, ACx
:: MPY Ymem, Cmem, ACy

To access three memory values (as in the above example) in a single cycle, the value referenced by Cmem must be located in a memory bank different from the one containing the Xmem and Ymem values.

6.4.4.1 Important Facts About the BB Bus

Keep the following facts about the BB bus in mind as you use the coefficient indirect addressing mode:

- Although the following instructions access Cmem operands, they do not use the BB bus to fetch the 16-bit or 32-bit Cmem operand.

Instruction Syntax	Description of Cmem Access	Bus Used to Access Cmem
MOV Cmem, Smem	16-bit read from Cmem	DB
MOV Smem, Cmem	16-bit write to Cmem	EB
MOV Cmem, dbl(Lmem)	32-bit read from Cmem	CB for most significant word (MSW) DB for least significant word (LSW)
MOV dbl(Lmem), Cmem	32-bit write to Cmem	FB for MSW EB for LSW

- The BB bus is not connected to external memory. If a Cmem operand is accessed via BB, the operand must be in internal memory.

6.4.4.2 Coefficient Indirect Operands

Table 6–10 introduces the operands available for the coefficient indirect addressing mode. Note that:

- Both pointer modification and address generation are linear or circular according to the pointer configuration in status register ST2_55. The content of the 16-bit buffer start address register BSAC is added only if circular addressing is activated for CDP.

- Increments and decrements are made to the 16-bit pointer only. You cannot address data across main data pages without changing the value in the extension register (CDPH).

Notes:

1) Although an increment past FFFFh or a decrement past 0000h causes the pointer value to wrap around, do not make use of this behavior; it is not supported. Also, during circular addressing, the BSAC addition must not increment the address beyond FFFFh.

2) If you use algebraic instructions, you must enclose each coefficient indirect operand in the syntax element coef() (see section 6.4.4.3). If you use mnemonic instructions, you can use the operands as shown in Table 6–10.

Table 6–10. Coefficient Indirect Operands

Operand	Pointer Modification	Supported Access Type
*CDP	CDP is not modified.	Data-memory
*CDP+	CDP is incremented after the address is generated: If 16-bit operation: CDP = CDP + 1 If 32-bit operation: CDP = CDP + 2	Data-memory
*CDP–	CDP is decremented after the address is generated: If 16-bit operation: CDP = CDP – 1 If 32-bit operation: CDP = CDP – 2	Data-memory
*(CDP + T0/AR0)	The 16-bit signed constant in T0 or AR0 is added to CDP after the address is generated: If C54CM = 0: CDP = CDP + T0 If C54CM = 1: CDP = CDP + AR0	Data-memory

6.4.4.3 coef() Required for Coefficient Indirect Operands in Algebraic Instructions

If you use algebraic instructions, you must enclose each coefficient indirect (Cmem) operand in the coef() syntax element. For example, suppose you are given this algebraic instruction syntax:

ACx = ACx + (Smem * Cmem)

Assume ACx = AC0 and Smem = *AR0, and assume that for Cmem you want to use *CDP. The instruction is written as follows:

```
AC0 = AC0 + (*AR0 * coef(*CDP))
```

6.5 Addressing Data Memory

Absolute, direct, and indirect addressing can be used to address values in data memory:

Addressing Type	See ...
Absolute	This page
Direct	Page 6-37
Indirect	Page 6-38

6.5.1 Addressing Data Memory With Absolute Addressing Modes

The k16 absolute operand *abs16(#k16) or the k23 absolute operand *(#k23) can be used to access data memory in any instruction with one of these syntax elements:

Smem Indicates one word (16 bits) of data

Lmem Indicates two adjacent words (32 bits) of data

Table 6–11 and Table 6–12 provide examples of how to use these operands.

Note:

Because of a multi-byte extension, an instruction using *abs16(#k16) or *(#k23) cannot be executed in parallel with another instruction.

- When an instruction uses *abs16(#k16), the constant, k16, is encoded in a 2-byte extension to the instruction.
- When an instruction uses *(#k23), the constant, k23, is encoded in a 3-byte extension to the instruction.

*Table 6–11. *abs16(#k16) Used for Data-Memory Access*

Example Syntax	Example Instruction	Address(es) Generated For DPH = 3	Description
MOV Smem, dst	MOV *abs16(#2002h), T2	DPH:k16 = 03 2002h	The CPU loads the value at address 03 2002h into T2.
MOV dbl(Lmem), pair(TAx)	MOV dbl(*abs16(#2002h)), pair(T2)	DPH:k16 = 03 2002h (DPH:k16) + 1 = 03 2003h	The CPU reads the values at addresses 03 2002h and 03 2003h and copies them into T2 and T3, respectively.

*Table 6−12. *(#k23) Used for Data-Memory Access*

Example Syntax	Example Instruction	Address(es) Generated	Description
MOV Smem, dst	MOV *(#032002h), T2	k23 = 03 2002h	The CPU loads the value at address 03 2002h into T2.
MOV dbl(Lmem), pair(TAx)	MOV dbl(*(#032002h)), pair(T2)	k23 = 03 2002h k23 + 1 = 03 2003h	The CPU reads the values at address 03 2002h and address 03 2003h and copies them into T2 and T3, respectively.

6.5.2 Addressing Data Memory With Direct Addressing Modes

You can use direct addressing modes to access data memory in any instruction with one of these syntax elements:

Smem Indicates one word (16 bits) of data

Lmem Indicates two adjacent words (32 bits) of data

When the CPL bit is 0, you can use the DP direct operand (@Daddr). When the CPL bit is 1, you can use the SP direct operand *SP(offset). Table 6−13 and Table 6−14 provide examples of using these operands to access data memory.

For the DP direct addressing mode, the assembler calculates Doffset for the address as follows:

Doffset = (Daddr − .dp) & 7Fh

where .dp is a value assigned by the .dp assembler directive, and & indicates a bitwise AND operation. For examples of using the .dp directive and of the Doffset calculation, see section 6.3.1.1. As shown in Table 6−13, when DP = .dp, Doffset is equal to Daddr (in this case, both are 0005h).

Table 6–13. @Daddr Used for Data-Memory Access

Example Syntax	Example Instruction	Address(es) Generated For DPH = 3, DP = .dp = 0	Description
MOV Smem, dst	MOV @0005h, T2	DPH:(DP + Doffset) = 03:(0000h + 0005h)= 03 0005h	The CPU loads the value at address 03 0005h into T2.
MOV dbl(Lmem), pair(TAx)	MOV dbl(@0005h), pair(T2)	DPH:(DP + Doffset) = 03 0005h DPH:(DP + Doffset – 1) = 03 0004h	The CPU reads the values at addresses 03 0005h and 03 0004h and loads these values into T2 and T3, respectively. The second word is read from the preceding even address in accordance with the alignment rule for long words.

*Table 6–14. *SP(offset) Used for Data-Memory Access*

Example Syntax	Example Instruction	Address(es) Generated For SP = FF00h and SPH = 0	Description
MOV Smem, dst	MOV *SP(5), T2	SPH:(SP + offset) = 00 FF05h	The CPU loads the value at address 00 FF05h into T2.
MOV dbl(Lmem), pair(TAx)	MOV dbl(*SP(5)), pair(T2)	SPH:(SP + offset) = 00 FF05h SPH:(SP + offset – 1) = 00 FF04h	The CPU reads the values at addresses 00 FF05h and 00 FF04h, and loads these values into T2 and T3, respectively. The second word is read from the preceding even address in accordance with the alignment rule for long words.

6.5.3 Addressing Data Memory With Indirect Addressing Modes

When you want to use indirect operands to access data memory, it is important to know which operands you can use for a given instruction. Each instruction syntax that supports indirect accesses to data memory includes one of the syntax elements shown in Table 6–15. The last column of the table shows which indirect operands can be used in place of the syntax element(s) for accesses to data memory. As explained in section 6.4.1.4, the AR indirect

operands available depend on whether DSP mode or control mode is selected by the ARMS bit. After you have found an operand in Table 6–15, you can find details about that operand in the sections that follow the table. For details about the alignment of long words in data memory, see section 3.3.2, *Data Types.*

Table 6–15. Choosing an Indirect Operand for a Data Memory Access

Syntax Element	Description	Available Indirect Operands
Smem or Lmem	Smem indicates one word (16 bits) of data. Lmem indicates two consecutive words (32 bits) of data.	AR indirect addressing mode: DSP mode (ARMS = 0): Control mode (ARMS = 1): *ARn *ARn *ARn+ *ARn+ *ARn– *ARn– *+ARn *–ARn *(ARn + T0/AR0) *(ARn + T0/AR0) *(ARn – T0/AR0) *(ARn – T0/AR0) *ARn(T0/AR0) *ARn(T0/AR0) *(ARn + T0B/AR0B) *(ARn – T0B/AR0B) *(ARn + T1) *(ARn – T1) *ARn(T1) *ARn(#K16) *ARn(#K16) *+ARn(#K16) *+ARn(#K16) *ARn(short(#k3)) CDP indirect addressing mode: *CDP *CDP+ *CDP– *CDP(#k16) *+CDP(#k16)
Xmem or Ymem	One word (16 bits) of data	Dual AR indirect addressing mode: *ARn *ARn+ *ARn– *(ARn + T0/AR0) *(ARn – T0/AR0) *ARn(T0/AR0) *(ARn + T1) *(ARn – T1)
Cmem	One word (16 bits) of data	Coefficient indirect addressing mode: *CDP *CDP+ *CDP– *(CDP + T0/AR0)

6.5.3.1 *ARn Used for Data-Memory Access

Operand	Description
*ARn	Address generated: ARnH:([BSAyy +] ARn) ARn is not modified.

Example Syntax	Example Instruction	Address(es) Generated (Linear Addressing)	Description
MOV Smem, dst	MOV *AR4, T2	AR4H:AR4 = XAR4	The CPU reads the value at address XAR4 and loads it into T2. AR4 is not modified.
MOV dbl(Lmem), pair(TAx)	MOV dbl(*AR4), pair(T2)	**First address:** XAR4 **Second address:** If XAR4 is even XAR4 + 1 If XAR4 is odd XAR4 – 1	The CPU reads the values at address XAR4 and the following or preceding address, and loads them into T2 and T3, respectively. AR4 is not modified. For details about the alignment of long words in data memory, see *Data Types* on page 3-5.

6.5.3.2 *ARn+ Used for Data-Memory Access

Operand	Description
*ARn+	1) Address generated: ARnH:([BSAyy +] ARn) 2) ARn modified: If 16-bit operation: ARn = ARn + 1 If 32-bit operation: ARn = ARn + 2

Example Syntax	Example Instruction	Address(es) Generated (Linear Addressing)	Description
MOV Smem, dst	MOV *AR4+, T2	AR4H:AR4 = XAR4	The CPU reads the value at address XAR4 and loads it into T2. After being used for the address, AR4 is incremented by 1.
MOV dbl(Lmem), pair(TAx)	MOV dbl(*AR4+), pair(T2)	**First address:** XAR4 **Second address:** If XAR4 is even XAR4 + 1 If XAR4 is odd XAR4 – 1	The CPU reads the values at address XAR4 and the following or preceding address, and loads them into T2 and T3, respectively. After being used for the addresses, AR4 is incremented by 2. For details about the alignment of long words in data memory, see *Data Types* on page 3-5.

6.5.3.3 *ARn– Used for Data-Memory Access

Operand	Description
*ARn–	1) Address generated: ARnH:([BSAyy +] ARn) 2) ARn modified: If 16-bit operation: ARn = ARn – 1 If 32-bit operation: ARn = ARn – 2

Example Syntax	Example Instruction	Address(es) Generated (Linear Addressing)	Description
MOV Smem, dst	MOV *AR4–, T2	AR4H:AR4 = XAR4	The CPU reads the value at address XAR4 and loads it into T2. After being used for the address, AR4 is decremented by 1.
MOV dbl(Lmem), pair(TAx)	MOV dbl(*AR4–), pair(T2)	**First address:** XAR4 **Second address:** If XAR4 is even XAR4 + 1 If XAR4 is odd XAR4 – 1	The CPU reads the values at address XAR4 and the following or preceding address, and loads them into T2 and T3, respectively. After being used for the addresses, AR4 is decremented by 2. For details about the alignment of long words in data memory, see *Data Types* on page 3-5.

6.5.3.4 *+ARn Used for Data-Memory Access

Operand	Description
*+ARn	1) ARn modified: If 16-bit operation: ARn = ARn + 1 If 32-bit operation: ARn = ARn + 2 2) Address generated: If 16-bit operation: ARnH:([BSAyy +] ARn + 1) If 32-bit operation: ARnH:([BSAyy +] ARn + 2)

Example Syntax	Example Instruction	Address(es) Generated (Linear Addressing)	Description
MOV Smem, dst	MOV *+AR4, T2	AR4H:(AR4 + 1) = XAR4 + 1	Before being used for the address, AR4 is incremented by 1. The CPU reads the value at address XAR4 + 1 and loads it into T2.
MOV dbl(Lmem), pair(TAx)	MOV dbl(*+AR4), pair(T2)	**First address:** AR4H:(AR4 + 2) = XAR4 + 2 **Second address:** If XAR4 + 2 is even (XAR4 + 2) + 1 If XAR4 + 2 is odd (XAR4 + 2) – 1	Before being used for the addresses, AR4 is incremented by 2. The CPU reads the values at address XAR4 + 2 and the following or preceding address, and loads them into T2 and T3, respectively. For details about the alignment of long words in data memory, see *Data Types* on page 3-5.

6.5.3.5 *–ARn Used for Data-Memory Access

Operand	Description
*–ARn	1) ARn modified: If 16-bit operation: ARn = ARn – 1 If 32-bit operation: ARn = ARn – 2 2) Address generated: If 16-bit operation: ARnH:([BSAyy +] ARn – 1) If 32-bit operation: ARnH:([BSAyy +] ARn – 2)

Example Syntax	Example Instruction	Address(es) Generated (Linear Addressing)	Description
MOV Smem, dst	MOV *–AR4, T2	AR4H:(AR4 – 1) = XAR4 – 1	Before being used for the address, AR4 is decremented by 1. The CPU reads the value at address XAR4 – 1 and loads it into T2.
MOV dbl(Lmem), pair(TAx)	MOV dbl(*–AR4), pair(T2)	**First address:** AR4H:(AR4 – 2) = XAR4 – 2 **Second address:** If XAR4 – 2 is even (XAR4 – 2) + 1 If XAR4 – 2 is odd (XAR4 – 2) – 1	Before being used for the addresses, AR4 is decremented by 2. The CPU reads the values at address XAR4 – 2 and the following or preceding address and loads them into T2 and T3, respectively. For details about the alignment of long words in data memory, see *Data Types* on page 3-5.

6.5.3.6 *(ARn + T0/AR0) Used for Data-Memory Access

C54CM = 0		C54CM = 1	
Operand	**Description**	**Operand**	**Description**
*(ARn + T0)	1) Address generated: ARnH:([BSAyy +] ARn) 2) ARn modified: ARn = ARn + T0	*(ARn + AR0)	1) Address generated: ARnH:([BSAyy +] ARn) 2) ARn modified: ARn = ARn + AR0

Example Syntax	Example Instruction	Address(es) Generated (Linear Addressing)	Description
MOV Smem, dst	MOV *(AR4 + T0), T2	AR4H:AR4 = XAR4	The CPU reads the value at address XAR4 and loads it into T2. After being used for the address, AR4 is incremented by the number in T0.
MOV dbl(Lmem), pair(TAx)	MOV dbl(*(AR4 + T0)), pair(T2)	**First address:** XAR4 **Second address:** If XAR4 is even XAR4 + 1 If XAR4 is odd XAR4 – 1	The CPU reads the values at address XAR4 and the following or preceding address and loads them into T2 and T3, respectively. After being used for the addresses, AR4 is incremented by the number in T0. For details about the alignment of long words in data memory, see *Data Types* on page 3-5.

6.5.3.7 *(ARn – T0/AR0) Used for Data-Memory Access

C54CM = 0		C54CM = 1	
Operand	**Description**	**Operand**	**Description**
*(ARn – T0)	1) Address generated: ARnH:([BSAyy +] ARn) 2) ARn modified: ARn = ARn – T0	*(ARn – AR0)	1) Address generated: ARnH:([BSAyy +] ARn) 2) ARn modified: ARn = ARn – AR0

Example Syntax	Example Instruction	Address(es) Generated (Linear Addressing)	Description
MOV Smem, dst	MOV *(AR4 – T0), T2	AR4H:AR4 = XAR4	The CPU reads the value at address XAR4 and loads it into T2. After being used for the address, AR4 is decremented by the number in T0.
MOV dbl(Lmem), pair(TAx)	MOV dbl(*(AR4 – T0)), pair(T2)	**First address:** XAR4 **Second address:** If XAR4 is even XAR4 + 1 If XAR4 is odd XAR4 – 1	The CPU reads the values at address XAR4 and the following or preceding address, and loads them into T2 and T3, respectively. After being used for the addresses, AR4 is decremented by the number in T0. For details about the alignment of long words in data memory, see *Data Types* on page 3-5.

6.5.3.8 *ARn(T0/AR0) Used for Data-Memory Access

C54CM = 0		C54CM = 1	
Operand	**Description**	**Operand**	**Description**
*ARn(T0)	Address generated: ARnH:([BSAyy +] ARn + T0) ARn is not modified. ARn is used as a base pointer. T0 is used as an offset from that base pointer.	*ARn(AR0)	Address generated: ARnH:([BSAyy +] ARn + AR0) ARn is not modified. ARn is used as a base pointer. AR0 is used as an offset from that base pointer.

Example Syntax	Example Instruction	Address(es) Generated (Linear Addressing)	Description
MOV Smem, dst	MOV *AR4(T0), T2	AR4H:(AR4 + T0) = XAR4 + T0	The CPU reads the value at address XAR4 + T0 and loads it into T2. AR4 is not modified.
MOV dbl(Lmem), pair(TAx)	MOV dbl(*AR4(T0)), pair(T2)	**First address:** XAR4 + T0 **Second address:** If XAR4 + T0 is even (XAR4 + T0) + 1 If XAR4 + T0 is odd (XAR4 + T0) – 1	The CPU reads the values at address XAR4 + T0 and the following or preceding address, and loads them into T2 and T3, respectively. AR4 is not modified. For details about the alignment of long words in data memory, see *Data Types* on page 3-5.

6.5.3.9 *(ARn + T0B/AR0B) Used for Data-Memory Access

C54CM = 0		C54CM = 1	
Operand	**Description**	**Operand**	**Description**
*(ARn + T0B)	1) Address generated: ARnH:([BSAyy +] ARn) 2) ARn modified: ARn = ARn + T0 (done with reverse carry propagation) See **Note** about circular addressing restriction.	*(ARn + AR0B)	1) Address generated: ARnH:([BSAyy +] ARn) 2) ARn modified: ARn = ARn + AR0 (done with reverse carry propagation) See **Note** about circular addressing restriction.

Note: When this bit-reverse operand is used, ARn cannot be used as a circular pointer. If ARn is configured in ST2_55 for circular addressing, the corresponding buffer start address register value (BSAyy) is added to ARn, but ARn is not modified so as to remain inside a circular buffer.

Example Syntax	Example Instruction	Address(es) Generated (Linear Addressing)	Description
MOV Smem, dst	MOV *(AR4 + T0B), T2	AR4H:AR4 = XAR4	The CPU reads the value at address XAR4 and loads it into T2. After being used for the address, AR4 is incremented by the number in T0. Reverse carry propagation is used during the addition.
MOV dbl(Lmem), pair(TAx)	MOV dbl(*(AR4 + T0B)), pair(T2)	**First address:** XAR4 **Second address:** If XAR4 is even XAR4 + 1 If XAR4 is odd XAR4 – 1	The CPU reads the values at address XAR4 and the following or preceding address and loads them into T2 and T3, respectively. After being used for the address, AR4 is incremented by the number in T0. Reverse carry propagation is used during the addition. For details about the alignment of long words in data memory, see *Data Types* on page 3-5.

6.5.3.10 *(ARn – T0B/AR0B) Used for Data-Memory Access

C54CM = 0		C54CM = 1	
Operand	**Description**	**Operand**	**Description**
*(ARn – T0B)	1) Address generated: ARnH:([BSAyy +] ARn) 2) ARn modified: ARn = ARn – T0 (done with reverse carry propagation) See **Note** about circular addressing restriction.	*(ARn – AR0B)	1) Address generated: ARnH:([BSAyy +] ARn) 2) ARn modified: ARn = ARn – AR0 (done with reverse carry propagation) See **Note** about circular addressing restriction.

Note: When this bit-reverse operand is used, ARn cannot be used as a circular pointer. If ARn is configured in ST2_55 for circular addressing, the corresponding buffer start address register value (BSAyy) is added to ARn, but ARn is not modified so as to remain inside a circular buffer.

Example Syntax	Example Instruction	Address(es) Generated (Linear Addressing)	Description
MOV Smem, dst	MOV *(AR4 – T0B), T2	AR4H:AR4 = XAR4	The CPU loads the value at address XAR4 into T2. After being used for the address, AR4 is decremented by the number in T0. Reverse carry propagation is used during the subtraction.
MOV dbl(Lmem), pair(TAx)	MOV dbl(*(AR4 – T0B)), pair(T2)	**First address:** XAR4 **Second address:** If XAR4 is even XAR4 + 1 If XAR4 is odd XAR4 – 1	The CPU reads the values at address XAR4 and the following or preceding address and loads them into T2 and T3, respectively. After being used for the addresses, AR4 is decremented by the number in T0. Reverse carry propagation is used during the subtraction. For details about the alignment of long words in data memory, see *Data Types* on page 3-5.

6.5.3.11 *(ARn + T1) Used for Data-Memory Access

Operand	Description
*(ARn + T1)	1) Address generated: ARnH: [BSAyy +] ARn 2) ARn modified: ARn = ARn + T1

Example Syntax	Example Instruction	Address(es) Generated (Linear Addressing)	Description
MOV Smem, dst	MOV *(AR7 + T1), AR3	AR7H:AR7 = XAR7	The CPU reads the value at address XAR7 and loads it into AR3. After being used for the address, AR7 is incremented by the number in T1.
MOV dbl(Lmem), pair(TAx)	MOV dbl(*(AR5 + T1)), pair(AR2)	**First address:** AR5H:AR5 = XAR5 **Second address:** If XAR5 is even XAR5 + 1 If XAR5 is odd XAR5 – 1	The CPU reads the values at address XAR5 and the following or preceding address, and loads them into AR2 and AR3, respectively. After being used for the addresses, AR5 is incremented by the number in T1. For details about the alignment of long words in data memory, see *Data Types* on page 3-5.

6.5.3.12 *(ARn – T1) Used for Data-Memory Access

Operand	Description
*(ARn – T1)	1) Address generated: ARnH:([BSAyy +] ARn) 2) ARn modified: ARn = ARn – T1

Example Syntax	Example Instruction	Address(es) Generated (Linear Addressing)	Description
MOV Smem, dst	MOV *(AR7 – T1), AR3	AR7H:AR7 = XAR7	The CPU reads the value at address XAR7 and loads it into AR3. After being used for the address, AR7 is decremented by the number in T1.
MOV dbl(Lmem), pair(TAx)	MOV dbl(*(AR5 – T1)), pair(AR2)	**First address:** AR5H:AR5 = XAR5 **Second address:** If XAR5 is even XAR5 + 1 If XAR5 is odd XAR5 – 1	The CPU reads the values at address XAR5 and the following or preceding address and loads them into AR2 and AR3, respectively. After being used for the addresses, AR5 is decremented by the number in T1. For details about the alignment of long words in data memory, see *Data Types* on page 3-5.

6.5.3.13 *ARn(T1) Used for Data-Memory Access

Operand	Description
*ARn(T1)	Address generated: ARnH:([BSAyy +] ARn + T1) ARn is not modified. ARn is used as a base pointer. T1 is used as an offset from that base pointer.

Example Syntax	Example Instruction	Address(es) Generated (Linear Addressing)	Description
MOV Smem, dst	MOV *AR7(T1), AR3	AR7H:(AR7 + T1) = XAR7 + T1	The CPU reads the value at address XAR7 + T1 and loads it into AR3. AR7 is not modified.
MOV dbl(Lmem), pair(TAx)	MOV dbl(*AR5(T1)), pair(AR2)	**First address:** AR5H:(AR5 + T1) = XAR5 + T1 **Second address:** If XAR5 + T1 is even (XAR5 + T1) + 1 If XAR5 + T1 is odd (XAR5 + T1) – 1	The CPU reads the values at address XAR5 + T1 and the following or preceding address, and loads them into AR2 and AR3, respectively. AR5 is not modified. For details about the alignment of long words in data memory, see *Data Types* on page 3-5.

6.5.3.14 *ARn(#K16) Used for Data-Memory Access

Operand	Description
*ARn(#K16)	Address generated: ARnH:([BSAyy +] ARn + K16) ARn is not modified. ARn is used as a base pointer. The 16-bit signed constant (K16) is used as an offset from that base pointer.

Note: When an instruction uses this operand, the constant, K16, is encoded in a 2-byte extension to the instruction. Because of the extension, an instruction using this operand cannot be executed in parallel with another instruction.

Example Syntax	Example Instruction	Address(es) Generated (Linear Addressing)	Description
MOV Smem, dst	MOV *AR7(#8), AR3	AR7H:(AR7 + 8) = XAR7 + 8	The CPU reads the value at address XAR7 + 8 and loads it into AR3. AR7 is not modified.
MOV dbl(Lmem), pair(TAx)	MOV dbl(*AR5(#20)), pair(AR2)	**First address:** AR5H:(AR5 + 20) = XAR5 + 20 **Second address:** If XAR5 + 20 is even (XAR5 + 20) + 1 If XAR5 + 20 is odd (XAR5 + 20) – 1	The CPU reads the values at address XAR5 + 20 and the following or preceding address, and loads them into AR2 and AR3, respectively. AR5 is not modified. For details about the alignment of long words in data memory, see *Data Types* on page 3-5.

6.5.3.15 *+ARn(#K16) Used for Data-Memory Access

Operand	Description
*+ARn(#K16)	1) ARn modified: ARn = ARn + K16 2) Address generated: ARnH:([BSAyy +] ARn + K16)

Note: When an instruction uses this operand, the constant, K16, is encoded in a 2-byte extension to the instruction. Because of the extension, an instruction using this operand cannot be executed in parallel with another instruction.

Example Syntax	Example Instruction	Address(es) Generated (Linear Addressing)	Description
MOV Smem, dst	MOV *+AR7(#8), AR3	AR7H:(AR7 + 8) = XAR7 + 8	Before AR7 is used for the address, the constant is added to AR7. The CPU reads the value at address XAR7 + 8 and loads it into AR3.
MOV dbl(Lmem), pair(TAx)	MOV dbl(*+AR5(#20)), pair(AR2)	**First address:** AR5H:(AR5 + 20) = XAR5 + 20 **Second address:** If XAR5 + 20 is even (XAR5 + 20) + 1 If XAR5 + 20 is odd (XAR5 + 20) − 1	Before AR5 is used for the addresses, the constant is added to AR5. The CPU reads the values at address XAR5 + 20 and the following or preceding address, and loads them into AR2 and AR3, respectively. For details about the alignment of long words in data memory, see *Data Types* on page 3-5.

6.5.3.16 *ARn(short(#k3)) Used for Data-Memory Access

Operand	Description
*ARn(short(#k3))	Address generated: ARnH:([BSAyy +] ARn + k3) ARn is not modified. ARn is used as a base pointer. The 3-bit unsigned constant (k3) is used as an offset from that base pointer. k3 can be a number from 1 to 7.

Example Syntax	Example Instruction	Address(es) Generated (Linear Addressing)	Description
MOV Smem, dst	MOV *AR7(short(#1)), AR3	AR7H:(AR7 + 1) = XAR7 + 1	The CPU reads the value at address XAR7 + 1 and loads it into AR3. AR7 is not modified.
MOV dbl(Lmem), pair(TAx)	MOV dbl(*AR5(short(#7))), pair(AR2)	**First address:** AR5H:(AR5 + 7) = XAR5 + 7 **Second address:** If XAR5 + 7 is even (XAR5 + 7) + 1 If XAR5 + 7 is odd (XAR5 + 7) − 1	The CPU reads the values at address XAR5 + 7 and the following or preceding address, and loads them into AR2 and AR3, respectively. AR5 is not modified. For details about the alignment of long words in data memory, see *Data Types* on page 3-5.

6.5.3.17 *CDP Used for Data-Memory Access

Operand	Description
*CDP	Address generated: CDPH:[BSAC +] CDP CDP is not modified.

Example Syntax	Example Instruction	Address(es) Generated (Linear Addressing)	Description
MOV Smem, dst	MOV *CDP, T2	CDPH:CDP = XCDP	The CPU reads the value at address XCDP and loads it into T2. CDP is not modified.
MOV dbl(Lmem), pair(TAx)	MOV dbl(*CDP), pair(T2)	**First address:** XCDP **Second address:** If XCDP is even XCDP + 1 If XCDP is odd XCDP – 1	The CPU reads the values at address XCDP and the following or preceding address, and loads them into T2 and T3, respectively. CDP is not modified. For details about the alignment of long words in data memory, see *Data Types* on page 3-5.
MPY Xmem, Cmem, ACx :: MPY Ymem, Cmem, ACy	MPY *AR0, *CDP, AC0 :: MPY *AR1, *CDP, AC1	CDPH:CDP = XCDP	The CPU multiplies the value at address XAR0 by the coefficient at address XCDP and stores the result to AC0. At the same time, the CPU multiplies the value at address XAR1 by the same coefficient and stores the result to AC1. CDP is not modified.
MOV dbl(Lmem), Cmem	MOV dbl(*AR7), *CDP	**First address:** XCDP **Second address:** If XCDP is even XCDP + 1 If XCDP is odd XCDP – 1	The CPU reads the values at addresses XAR7 and XAR7 ± 1. Then it writes the values to addresses XCDP and XCDP ± 1, respectively. CDP is not modified.

Note: If you use algebraic instructions, you must enclose each coefficient indirect (Cmem) operand in the syntax element coef(). For an example, see section 6.4.4.3.

6.5.3.18 *CDP+ Used for Data-Memory Access

Operand	Description
*CDP+	1) Address generated: CDPH:[BSAC +] CDP 2) CDP modified: If 16-bit operation: CDP = CDP + 1 If 32-bit operation: CDP = CDP + 2

Example Syntax	Example Instruction	Address(es) Generated (Linear Addressing)	Description
MOV Smem, dst	MOV *CDP+, T2	CDPH:CDP = XCDP	The CPU reads the value at address XCDP and loads it into T2. After being used for the address, CDP is incremented by 1.
MOV dbl(Lmem), pair(TAx)	MOV dbl(*CDP+), pair(T2)	**First address:** XCDP **Second address:** If XCDP is even XCDP + 1 If XCDP is odd XCDP – 1	The CPU reads the values at address XCDP and the following or preceding address, and loads them into T2 and T3, respectively. After being used for the addresses, CDP is incremented by 2. For details about the alignment of long words in data memory, see *Data Types* on page 3-5.
MPY Xmem, Cmem, ACx :: MPY Ymem, Cmem, ACy	MPY *AR0, *CDP+, AC0 :: MPY *AR1, *CDP+, AC1	CDPH:CDP = XCDP	The CPU multiplies the value at address XAR0 by the coefficient at address XCDP and stores the result to AC0. At the same time, the CPU multiplies the value at address XAR1 by the same coefficient and stores the result to AC1. After being used for the address, CDP is incremented by 1.
MOV dbl(Lmem), Cmem	MOV dbl(*AR7), *CDP+	**First address:** XCDP **Second address:** If XCDP is even XCDP + 1 If XCDP is odd XCDP – 1	The CPU reads the values at addresses XAR7 and XAR7 ± 1. Then it writes the values to addresses XCDP and XCDP ± 1, respectively. After being used for the addresses, CDP is incremented by 2.

Note: If you use algebraic instructions, you must enclose each coefficient indirect (Cmem) operand in the syntax element coef(). For an example, see section 6.4.4.3.

6.5.3.19 *CDP– Used for Data-Memory Access

Operand	Description
*CDP–	1) Address generated: CDPH:[BSAC +] CDP 2) CDP modified: If 16-bit operation: CDP = CDP – 1 If 32-bit operation: CDP = CDP – 2

Example Syntax	Example Instruction	Address(es) Generated (Linear Addressing)	Description
MOV Smem, dst	MOV *CDP–, T2	CDPH:CDP = XCDP	The CPU reads the value at address XCDP and loads it into T2. After being used for the address, CDP is decremented by 1.
MOV dbl(Lmem), pair(TAx)	MOV dbl(*CDP–), pair(T2)	**First address:** XCDP **Second address:** If XCDP is even XCDP + 1 If XCDP is odd XCDP – 1	The CPU reads the values at address XCDP and the following or preceding address, and loads them into T2 and T3, respectively. After being used for the addresses, CDP is decremented by 2. For details about the alignment of long words in data memory, see *Data Types* on page 3-5.
MPY Xmem, Cmem, ACx :: MPY Ymem, Cmem, ACy	MPY *AR0, *CDP–, AC0 :: MPY *AR1, *CDP–, AC1	CDPH:CDP = XCDP	The CPU multiplies the value at address XAR0 by the coefficient at address XCDP and stores the result to AC0. At the same time, the CPU multiplies the value at address XAR1 by the same coefficient and stores the result to AC1. After being used for the address, CDP is decremented by 1.
MOV dbl(Lmem), Cmem	MOV dbl(*AR7), *CDP–	**First address:** XCDP **Second address:** If XCDP is even XCDP + 1 If XCDP is odd XCDP – 1	The CPU reads the values at addresses XAR7 and XAR7 ± 1. Then it writes the values to addresses XCDP and XCDP ± 1, respectively. After being used for the addresses, CDP is decremented by 2.

Note: If you use algebraic instructions, you must enclose each coefficient indirect (Cmem) operand in the syntax element coef(). For an example, see section 6.4.4.3.

6.5.3.20 *CDP(#K16) Used for Data-Memory Access

Operand	Description
*CDP(#K16)	Address generated: CDPH:([BSAC +] CDP + K16) CDP is not modified. CDP is used as a base pointer. The 16-bit signed constant (K16) is used as an offset from that base pointer.

Note: When an instruction uses this operand, the constant, K16, is encoded in a 2-byte extension to the instruction. Because of the extension, an instruction using this operand cannot be executed in parallel with another instruction.

Example Syntax	Example Instruction	Address(es) Generated (Linear Addressing)	Description
MOV Smem, dst	MOV *CDP(#8), AR3	CDPH:(CDP + 8) = XCDP + 8	The CPU reads the value at address XCDP + 8 and loads it into AR3. CDP is not modified.
MOV dbl(Lmem), pair(TAx)	MOV dbl(*CDP(#20)), pair(AR2)	**First address:** CDPH:(CDP + 20) = XCDP + 20 **Second address:** If XCDP + 20 is even (XCDP + 20) + 1 If XCDP + 20 is odd (XCDP + 20) − 1	The CPU reads the values at address XCDP + 20 and the following or preceding address, and loads them into AR2 and AR3, respectively. CDP is not modified. For details about the alignment of long words in data memory, see *Data Types* on page 3-5.

6.5.3.21 *+CDP(#K16) Used for Data-Memory Access

Operand	Description
*+CDP(#K16)	1) CDP modified: CDP = CDP + K16 2) Address generated: CDPH:([BSAC +] CDP + K16)

Note: When an instruction uses this operand, the constant, K16, is encoded in a 2-byte extension to the instruction. Because of the extension, an instruction using this operand cannot be executed in parallel with another instruction.

Example Syntax	Example Instruction	Address(es) Generated (Linear Addressing)	Description
MOV Smem, dst	MOV *+CDP(#8), AR3	CDPH:(CDP + 8) = XCDP + 8	Before CDP is used for the address, the constant is added to CDP. The CPU reads the value at address XCDP + 8 and loads it into AR3.
MOV dbl(Lmem), pair(TAx)	MOV dbl(*+CDP(#20)), pair(AR2)	**First address:** CDPH:(CDP + 20) = XCDP + 20 **Second address:** If XCDP + 20 is even (XCDP + 20) + 1 If XCDP + 20 is odd (XCDP + 20) – 1	Before CDP is used for the addresses, the constant is added to CDP. The CPU reads the values at address XCDP + 20 and the following or preceding address, and loads them into AR2 and AR3, respectively. For details about the alignment of long words in data memory, see *Data Types* on page 3-5.

6.5.3.22 *(CDP + T0/AR0) Used for Data-Memory Access

C54CM = 0		C54CM = 1	
Operand	**Description**	**Operand**	**Description**
*(CDP + T0)	1) Address generated: CDPH:[BSAC +] CDP 2) CDP = CDP + T0	*(CDP + AR0)	1) Address generated: CDPH:[BSAC +] CDP 2) CDP = CDP + AR0

Example Syntax	Example Instruction	Address(es) Generated (Linear Addressing)	Description
MPY Xmem, Cmem, ACx :: MPY Ymem, Cmem, ACy	MPY *AR0, *(CDP + T0), AC0 :: MPY *AR1, *(CDP + T0), AC1	CDPH:CDP = XCDP	The CPU multiplies the value at address XAR0 by the coefficient at address XCDP and stores the result to AC0. At the same time, the CPU multiplies the value at address XAR1 by the same coefficient and stores the result to AC1. After being used for the address, CDP is incremented by the number in T0.
MOV dbl(Lmem), Cmem	MOV dbl(*AR7), *(CDP + T0)	**First address:** XCDP **Second address:** If XCDP is even XCDP + 1 If XCDP is odd XCDP − 1	The CPU reads the values at addresses XAR7 and XAR7 ± 1. Then it writes the values to addresses XCDP and XCDP ± 1, respectively. After being used for the addresses, CDP is incremented by the number in T0. For details about the alignment of long words in data memory, see *Data Types* on page 3-5.

Note: If you use algebraic instructions, you must enclose each coefficient indirect (Cmem) operand in the syntax element coef(). For an example, see section 6.4.4.3.

6.6 Addressing Memory-Mapped Registers

Absolute, direct, and indirect addressing can be used to address memory-mapped registers (MMRs):

Addressing Type	See ...
Absolute	This page
Direct	Page 6-61
Indirect	Page 6-63

There are some restrictions on accesses to memory-mapped registers. See section 6.7 for the details.

6.6.1 Addressing MMRs With the k16 and k23 Absolute Addressing Modes

The k16 absolute operand *abs16(#k16) and the k23 absolute operand *(#k23) can be used to access memory-mapped registers in any instruction with one of these syntax elements:

Smem Indicates one word (16 bits) of data

Lmem Indicates two adjacent words (32 bits) of data

See the examples in Table 6–16 and Table 6–17. Because the memory-mapped registers are on main data page 0, to access them with *abs16(#k16), you must first make sure DPH = 0.

Note:

Because of a multi-byte extension, an instruction using *abs16(#k16) or *(#k23) cannot be executed in parallel with another instruction.

- When an instruction uses *abs16(#k16), the constant, k16, is encoded in a 2-byte extension to the instruction.
- When an instruction uses *(#k23), the constant, k23, is encoded in a 3-byte extension to the instruction.

*Table 6–16. *abs16(#k16) Used for Memory-Mapped Register Access*

Example Syntax	Example Instruction	Address(es) Generated (DPH must be 0)	Description
MOV Smem, dst	MOV *abs16(#AR2), T2	DPH:k16 = 00 0012h	AR2 is at address 00 0012h. The CPU loads the content of AR2 into T2.
MOV dbl(Lmem), pair(TAx)	MOV dbl(*abs16(#AR2)), pair(T2)	**First Address:** DPH:k16= 00 0012h **Second address:** (DPH:k16) + 1 = 00 0013h	AR2 and AR3 are at addresses 00 0012h and 00 0013h, respectively. The CPU reads the contents of AR2 and AR3 and copies them into T2 and T3, respectively.

*Table 6–17. *(#k23) Used for Memory-Mapped Register Access*

Example Syntax	Example Instruction	Address(es) Generated	Description
MOV Smem, dst	MOV *(#AC0L), T2	k23 = 00 0008h	AC0L represents the address of the 16 LSBs of AC0. The CPU loads the content of AC0(15–0) into T2.
MOV dbl(Lmem), pair(TAx)	MOV dbl(*(#AC0L)), pair(T2)	**First Address:** k23 = 00 0008h **Second address:** k23 + 1 = 00 0009h	AC0L represents the address of the 16 LSBs of AC0. The CPU loads the content of AC0(15–0) into T2 and the content of AC0(31–16) into T3.

6.6.2 Addressing MMRs With the DP Direct Addressing Mode

When the CPL bit is 0, you can use the DP direct operand (@Daddr) to access a memory-mapped register or registers (MMRs) if an instruction has one of these syntax elements:

Smem Indicates one word (16 bits) of data

Lmem Indicates two adjacent words (32 bits) of data

Use the mmap() qualifier to indicate that the access is to a memory-mapped register rather than to a data-memory location. If you use algebraic instructions, mmap() is an instruction qualifier placed in parallel with the instruction that performs a memory-mapped register access. If you use mnemonic instructions, mmap() encloses the qualified operand.

The mmap() qualifier forces the address-generation unit (DAGEN) to act as if:

DPH = 0 Accesses are made to main data page 0.

CPL = 0 Accesses are made relative to the DP.

DP = 0 Accesses are made relative to a local address of 0000h.

Table 6–18 provides examples of using @Daddr and mmap() to access registers. For the address, the assembler calculates Doffset as follows:

Doffset = Daddr & 7Fh

where & indicates a bitwise AND operation. Note that Daddr is supplied by a reference to AC0L (the 16 least significant bits of AC0). AC0L is mapped to address 00 0008h in data space.

Table 6–18. @Daddr Used for Memory-Mapped Register Access

Example Syntax	Example Instruction	Address(es) Generated (DPH = DP = 0)	Description
MOV Smem, dst	MOV mmap(@AC0L), AR2	DPH:(DP + Doffset) = 00:(0000h + 0008h) = 00 0008h	AC0L represents the address of the 16 LSBs of AC0. The CPU copies the content of AC0(15–0) into AR2.
MOV dbl(Lmem), pair(TAx)	MOV dbl(mmap(@AC0L)), pair(AR2)	**First address:** DPH:(DP + Doffset) = 00 0008h **Second address:** DPH:(DP + Doffset + 1) = 00 0009h	AC0L represents the address of the 16 LSBs of AC0. The CPU copies the content of AC0(15–0) into AR2 and copies the content of AC0(31–16) into AR3.

6.6.3 Addressing MMRs With Indirect Addressing Modes

You can use indirect operands to access a memory-mapped register or registers if an instruction has one of these syntax elements:

Smem Indicates one word (16 bits) of data

Xmem Indicates one word (16 bits) of read data through DB bus

Ymem Indicates one word (16 bits) of write data through EB bus

Lmem Indicates two adjacent words (32 bits) of data

You must first make sure the pointer contains the correct data-space address. For example, if XAR6 is the pointer and you want to access the low word of AC0, you can use the following instruction to initialize XAR6:

AMOV #AC0L, XAR6

where AC0L is a keyword that represents the address of the low word of AC0. Similarly, when CDP is the pointer, you can use:

AMOV #AC0L, XCDP

Figure 6−14 lists the indirect operands that support accesses of memory-mapped registers, and the sections following the table provide details and examples for the operands.

Figure 6−14. Indirect Operands for Memory-Mapped Register Accesses

AR indirect addressing mode:

DSP mode (ARMS = 0):	Control mode (ARMS = 1):
*ARn	*ARn
*ARn+	*ARn+
*ARn−	*ARn−
*+ARn	
*−ARn	
*(ARn + T0/AR0)	*(ARn + T0/AR0)
*(ARn − T0/AR0)	*(ARn − T0/AR0)
*ARn(T0/AR0)	*ARn(T0/AR0)
*(ARn + T0B/AR0B)	
*(ARn − T0B/AR0B)	
*(ARn + T1)	
*(ARn − T1)	
*ARn(T1)	
*ARn(#K16)	*ARn(#K16)
*+ARn(#K16)	*+ARn(#K16)
	*ARn(short(#k3))

CDP indirect addressing mode:

*CDP
*CDP+
*CDP−
*CDP(#k16)
*+CDP(#k16)

6.6.3.1 *ARn Used for Memory-Mapped Register Access

Operand	Description
*ARn	Address generated: ARnH:([BSAyy +] ARn) ARn is not modified.

Example Syntax	Example Instruction	Address(es) Generated (Linear Addressing)	Description
MOV Smem, dst	MOV *AR6, T2 (AR6 references a register.)	AR6H:AR6 = XAR6	The CPU reads the value at address XAR6 and loads it into T2. AR6 is not modified.
MOV dbl(Lmem), pair(TAx)	MOV dbl(*AR6), pair(T2) (AR6 references a register.)	**First address:** XAR6 **Second address:** If XAR6 is even XAR6 + 1 If XAR6 is odd XAR6 – 1	The CPU reads the values at address XAR6 and the following or preceding address, and loads them into T2 and T3, respectively. AR6 is not modified. For details about the alignment of long words in data memory, see *Data Types* on page 3-5.

6.6.3.2 *ARn+ Used for Memory-Mapped Register Access

Operand	Description
*ARn+	1) Address generated: ARnH:([BSAyy +] ARn) 2) ARn modified: If 16-bit operation: ARn = ARn + 1 If 32-bit operation: ARn = ARn + 2

Example Syntax	Example Instruction	Address(es) Generated (Linear Addressing)	Description
MOV Smem, dst	MOV *AR6+, T2 (AR6 references a register.)	AR6H:AR6 = XAR6	The CPU reads the value at address XAR6 and loads it into T2. After being used for the address, AR6 is incremented by 1.
MOV dbl(Lmem), pair(TAx)	MOV dbl(*AR6+), pair(T2) (AR6 references a register.)	**First address:** XAR6 **Second address:** If XAR6 is even XAR6 + 1 If XAR6 is odd XAR6 – 1	The CPU reads the values at address XAR6 and the following or preceding address, and loads them into T2 and T3, respectively. After being used for the addresses, AR6 is incremented by 2. For details about the alignment of long words in data memory, see *Data Types* on page 3-5.

6.6.3.3 *ARn– Used for Memory-Mapped Register Access

Operand	Description
*ARn–	1) Address generated: ARnH:([BSAyy +] ARn) 2) ARn modified: If 16-bit operation: ARn = ARn – 1 If 32-bit operation: ARn = ARn – 2

Example Syntax	Example Instruction	Address(es) Generated (Linear Addressing)	Description
MOV Smem, dst	MOV *AR6–, T2 (AR6 references a register.)	AR6H:AR6 = XAR6	The CPU reads the value at address XAR6 and loads it into T2. After being used for the address, AR6 is decremented by 1.
MOV dbl(Lmem), pair(TAx)	MOV dbl(*AR6–), pair(T2) (AR6 references a register.)	**First address:** XAR6 **Second address:** If XAR6 is even XAR6 + 1 If XAR6 is odd XAR6 – 1	The CPU reads the values at address XAR6 and the following or preceding address, and loads them into T2 and T3, respectively. After being used for the addresses, AR6 is decremented by 2. For details about the alignment of long words in data memory, see *Data Types* on page 3-5.

6.6.3.4 *+ARn Used for Memory-Mapped Register Access

Operand	Description
*+ARn	1) ARn modified: If 16-bit operation: ARn = ARn + 1 If 32-bit operation: ARn = ARn + 2 2) Address generated: If 16-bit operation: ARnH:([BSAyy +] ARn + 1) If 32-bit operation: ARnH:([BSAyy +] ARn + 2)

Example Syntax	Example Instruction	Address(es) Generated (Linear Addressing)	Description
MOV Smem, dst	MOV *+AR6, T2 (AR6 references a register.)	AR6H:(AR6 + 1) = XAR6 + 1	Before being used for the address, AR6 is incremented by 1. The CPU reads the value at address XAR6 + 1 and loads it into T2.
MOV dbl(Lmem), pair(TAx)	MOV dbl(*+AR6), pair(T2) (AR6 references a register.)	**First address:** AR6H:(AR6 + 2) = XAR6 + 2 **Second address:** If XAR6 + 2 is even (XAR6 + 2) + 1 If XAR6 + 2 is odd (XAR6 + 2) – 1	Before being used for the addresses, AR6 is incremented by 2. The CPU reads the values at address XAR6 + 2 and the following or preceding address, and loads them into T2 and T3, respectively. For details about the alignment of long words in data memory, see *Data Types* on page 3-5.

6.6.3.5 *–ARn Used for Memory-Mapped Register Access

Operand	Description
*–ARn	1) ARn modified: If 16-bit operation: ARn = ARn – 1 If 32-bit operation: ARn = ARn – 2 2) Address generated: If 16-bit operation: ARnH:([BSAyy +] ARn – 1) If 32-bit operation: ARnH:([BSAyy +] ARn – 2)

Example Syntax	Example Instruction	Address(es) Generated (Linear Addressing)	Description
MOV Smem, dst	MOV *–AR6, T2 (AR6 references a register.)	AR6H:(AR6 – 1) = XAR6 – 1	Before being used for the address, AR6 is decremented by 1. The CPU reads the value at address XAR6 – 1 and loads it into T2.
MOV dbl(Lmem), pair(TAx)	MOV dbl(*–AR6), pair(T2) (AR6 references a register.)	**First address:** AR6H:(AR6 – 2) = XAR6 – 2 **Second address:** If XAR6 – 2 is even (XAR6 – 2) + 1 If XAR6 – 2 is odd (XAR6 – 2) – 1	Before being used for the addresses, AR6 is decremented by 2. The CPU reads the values at address XAR6 – 2 and the following or preceding address, and loads them into T2 and T3, respectively. For details about the alignment of long words in data memory, see *Data Types* on page 3-5.

6.6.3.6 *(ARn + T0/AR0) Used for Memory-Mapped Register Access

	C54CM = 0		C54CM = 1
Operand	**Description**	**Operand**	**Description**
*(ARn + T0)	1) Address generated: ARnH:([BSAyy +] ARn) 2) ARn modified: ARn = ARn + T0	*(ARn + AR0)	1) Address generated: ARnH:([BSAyy +] ARn) 2) ARn modified: ARn = ARn + AR0

Example Syntax	Example Instruction	Address(es) Generated (Linear Addressing)	Description
MOV Smem, dst	MOV *(AR6 + T0), T2 (AR6 references a register.)	AR6H:AR6 = XAR6	The CPU reads the value at address XAR6 and loads it into T2. After being used for the address, AR6 is incremented by the number in T0.
MOV dbl(Lmem), pair(TAx)	MOV dbl(*(AR6 + T0)), pair(T2) (AR6 references a register.)	**First address:** XAR6 **Second address:** If XAR6 is even XAR6 + 1 If XAR6 is odd XAR6 – 1	The CPU reads the values at address XAR6 and the following or preceding address, and loads them into T2 and T3, respectively. After being used for the addresses, AR6 is incremented by the number in T0. For details about the alignment of long words in data memory, see *Data Types* on page 3-5.

6.6.3.7 *(ARn – T0/AR0) Used for Memory-Mapped Register Access

C54CM = 0		C54CM = 1	
Operand	**Description**	**Operand**	**Description**
*(ARn – T0)	1) Address generated: ARnH:([BSAyy +] ARn) 2) ARn modified: ARn = ARn – T0	*(ARn – AR0)	1) Address generated: ARnH:([BSAyy +] ARn) 2) ARn modified: ARn = ARn – AR0

Example Syntax	Example Instruction	Address(es) Generated (Linear Addressing)	Description
MOV Smem, dst	MOV *(AR6 – T0), T2 (AR6 references a register.)	AR6H:AR6 = XAR6	The CPU reads the value at address XAR6 and loads it into T2. After being used for the address, AR6 is decremented by the number in T0.
MOV dbl(Lmem), pair(TAx)	MOV dbl(*(AR6 – T0)), pair(T2) (AR6 references a register.)	**First address:** XAR6 **Second address:** If XAR6 is even XAR6 + 1 If XAR6 is odd XAR6 – 1	The CPU reads the values at address XAR6 and the following or preceding address, and loads them into T2 and T3, respectively. After being used for the addresses, AR6 is decremented by the number in T0. For details about the alignment of long words in data memory, see *Data Types* on page 3-5.

6.6.3.8 *ARn(T0/AR0) Used for Memory-Mapped Register Access

C54CM = 0		C54CM = 1	
Operand	**Description**	**Operand**	**Description**
*ARn(T0)	Address generated: ARnH:([BSAyy +] ARn + T0) ARn is not modified. ARn is used as a base pointer. T0 is used as an offset from that base pointer.	*ARn(AR0)	Address generated: ARnH:([BSAyy +] ARn + AR0) ARn is not modified. ARn is used as a base pointer. AR0 is used as an offset from that base pointer.

Example Syntax	Example Instruction	Address(es) Generated (Linear Addressing)	Description
MOV Smem, dst	MOV *AR6(T0), T2 (AR6 references a register.)	AR6H:(AR6 + T0) = XAR6 + T0	The CPU reads the value at address XAR6 + T0 and loads it into T2. AR6 is not modified.
MOV dbl(Lmem), pair(TAx)	MOV dbl(*AR6(T0)), pair(T2) (AR6 references a register.)	**First address:** XAR6 + T0 **Second address:** If XAR6 + T0 is even (XAR6 + T0) + 1 If XAR6 + T0 is odd (XAR6 + T0) – 1	The CPU reads the values at address XAR6 + T0 and the following or preceding address, and loads them into T2 and T3, respectively. AR6 is not modified. For details about the alignment of long words in data memory, see *Data Types* on page 3-5.

6.6.3.9 *(ARn + T0B/AR0B) Used for Memory-Mapped Register Access

C54CM = 0		C54CM = 1	
Operand	**Description**	**Operand**	**Description**
*(ARn + T0B)	1) Address generated: ARnH:([BSAyy +] ARn) 2) ARn modified: ARn = ARn + T0 (done with reverse carry propagation) See **Note** about circular addressing restriction.	*(ARn + AR0B)	1) Address generated: ARnH:([BSAyy +] ARn) 2) ARn modified: ARn = ARn + AR0 (done with reverse carry propagation) See **Note** about circular addressing restriction.

Note: When this bit-reverse operand is used, ARn cannot be used as a circular pointer. If ARn is configured in ST2_55 for circular addressing, the corresponding buffer start address register value (BSAyy) is added to ARn, but ARn is not modified so as to remain inside a circular buffer.

Example Syntax	Example Instruction	Address(es) Generated (Linear Addressing)	Description
MOV Smem, dst	MOV *(AR6 + T0B), T2 (AR6 references a register.)	AR6H:AR6 = XAR6	The CPU reads the value at address XAR6 and loads it into T2. After being used for the address, AR6 is incremented by the number in T0. Reverse carry propagation is used during the addition.
MOV dbl(Lmem), pair(TAx)	MOV dbl(*(AR6 + T0B)), pair(T2) (AR6 references a register.)	**First address:** XAR6 **Second address:** If XAR6 is even XAR6 + 1 If XAR6 is odd XAR6 – 1	The CPU reads the values at address XAR6 and the following or preceding address, and loads them into T2 and T3, respectively. After being used for the address, AR6 is incremented by the number in T0. Reverse carry propagation is used during the addition. For details about the alignment of long words in data memory, see *Data Types* on page 3-5.

6.6.3.10 *(ARn – T0B/AR0B) Used for Memory-Mapped Register Access

C54CM = 0		C54CM = 1	
Operand	**Description**	**Operand**	**Description**
*(ARn – T0B)	1) Address generated: ARnH:([BSAyy +] ARn) 2) ARn modified: ARn = ARn – T0 (done with reverse carry propagation) See **Note** about circular addressing restriction.	*(ARn – AR0B)	1) Address generated: ARnH:([BSAyy +] ARn) 2) ARn modified: ARn = ARn – AR0 (done with reverse carry propagation) See **Note** about circular addressing restriction.

Note: When this bit-reverse operand is used, ARn cannot be used as a circular pointer. If ARn is configured in ST2_55 for circular addressing, the corresponding buffer start address register value (BSAyy) is added to ARn, but ARn is not modified so as to remain inside a circular buffer.

Example Syntax	Example Instruction	Address(es) Generated (Linear Addressing)	Description
MOV Smem, dst	MOV *(AR6 – T0B), T2 (AR6 references a register.)	AR6H:AR6 = XAR6	The CPU reads the value at address XAR6 and loads it into T2. After being used for the address, AR6 is decremented by the number in T0. Reverse carry propagation is used during the subtraction.
MOV dbl(Lmem), pair(TAx)	MOV dbl(*(AR6 – T0B)), pair(T2) (AR6 references a register.)	**First address:** XAR6 **Second address:** If XAR6 is even XAR6 + 1 If XAR6 is odd XAR6 – 1	The CPU reads the values at address XAR6 and the following or preceding address, and loads them into T2 and T3, respectively. After being used for the addresses, AR6 is decremented by the number in T0. Reverse carry propagation is used during the subtraction. For details about the alignment of long words in data memory, see *Data Types* on page 3-5.

6.6.3.11 *(ARn + T1) Used for Memory-Mapped Register Access

Operand	Description
*(ARn + T1)	1) Address generated: ARnH: [BSAyy +] ARn 2) ARn modified: ARn = ARn + T1

Example Syntax	Example Instruction	Address(es) Generated (Linear Addressing)	Description
MOV Smem, dst	MOV *(AR7 + T1), AR3 (AR7 references a register.)	AR7H:AR7 = XAR7	The CPU reads the value at address XAR7 and loads it into AR3. After being used for the address, AR7 is incremented by the number in T1.
MOV dbl(Lmem), pair(TAx)	MOV dbl(*(AR5 + T1)), pair(AR2) (AR5 references a register.)	**First address:** AR5H:AR5 = XAR5 **Second address:** If XAR5 is even XAR5 + 1 If XAR5 is odd XAR5 – 1	The CPU reads the values at address XAR5 and the following or preceding address, and loads them into AR2 and AR3, respectively. After being used for the addresses, AR5 is incremented by the number in T1. For details about the alignment of long words in data memory, see *Data Types* on page 3-5.

6.6.3.12 *(ARn – T1) Used for Memory-Mapped Register Access

Operand	Description
*(ARn – T1)	1) Address generated: ARnH:([BSAyy +] ARn) 2) ARn modified: ARn = ARn – T1

Example Syntax	Example Instruction	Address(es) Generated (Linear Addressing)	Description
MOV Smem, dst	MOV *(AR7 – T1), AR3 (AR7 references a register.)	AR7H:AR7 = XAR7	The CPU reads the value at address XAR7 and loads it into AR3. After being used for the address, AR7 is decremented by the number in T1.
MOV dbl(Lmem), pair(TAx)	MOV dbl(*(AR5 – T1)), pair(AR2) (AR5 references a register.)	**First address:** AR5H:AR5 = XAR5 **Second address:** If XAR5 is even XAR5 + 1 If XAR5 is odd XAR5 – 1	The CPU reads the values at address XAR5 and the following or preceding address, and loads them into AR2 and AR3, respectively. After being used for the addresses, AR5 is decremented by the number in T1. For details about the alignment of long words in data memory, see *Data Types* on page 3-5.

6.6.3.13 *ARn(T1) Used for Memory-Mapped Register Access

Operand	Description
*ARn(T1)	Address generated: ARnH:([BSAyy +] ARn + T1) ARn is not modified. ARn is used as a base pointer. T1 is used as an offset from that base pointer.

Example Syntax	Example Instruction	Address(es) Generated (Linear Addressing)	Description
MOV Smem, dst	MOV *AR7(T1), AR3 (AR7 references a register.)	AR7H:(AR7 + T1) = XAR7 + T1	The CPU reads the value at address XAR7 + T1 and loads it into AR3. AR7 is not modified.
MOV dbl(Lmem), pair(TAx)	MOV dbl(*AR5(T1)), pair(AR2) (AR5 references a register.)	**First address:** AR5H:(AR5 + T1) = XAR5 + T1 **Second address:** If XAR5 + T1 is even (XAR5+ T1) + 1 If XAR5 + T1 is odd (XAR5+ T1) – 1	The CPU reads the values at address XAR5 + T1 and the following or preceding address, and loads them into AR2 and AR3, respectively. AR5 is not modified. For details about the alignment of long words in data memory, see *Data Types* on page 3-5.

6.6.3.14 *ARn(#K16) Used for Memory-Mapped Register Access

Operand	Description
*ARn(#K16)	Address generated: ARnH:([BSAyy +] ARn + K16) ARn is not modified. ARn is used as a base pointer. The 16-bit signed constant (K16) is used as an offset from that base pointer.

Note: When an instruction uses this operand, the constant, K16, is encoded in a 2-byte extension to the instruction. Because of the extension, an instruction using this operand cannot be executed in parallel with another instruction.

Example Syntax	Example Instruction	Address(es) Generated (Linear Addressing)	Description
MOV Smem, dst	MOV *AR7(#9), AR3 (AR7 references a register.)	AR7H:(AR7 + 9) = XAR7 + 9	The CPU reads the value at address XAR7 + 9 and loads it into AR3. AR7 is not modified.
MOV dbl(Lmem), pair(TAx)	MOV dbl(*AR5(#40)), pair(AR2) (AR5 references a register.)	**First address:** AR5H:(AR5 + 40) = XAR5 + 40 **Second address:** If XAR5 + 40 is even (XAR5+ 40) + 1 If XAR5 + 40 is odd (XAR5+ 40) – 1	The CPU reads the values at address XAR5 + 40 and the following or preceding address, and loads them into AR2 and AR3, respectively. AR5 is not modified. For details about the alignment of long words in data memory, see *Data Types* on page 3-5.

6.6.3.15 *+ARn(#K16) Used for Memory-Mapped Register Access

Operand	Description
*+ARn(#K16)	1) ARn modified: ARn = ARn + K16 2) Address generated: ARnH:([BSAyy +] ARn + K16)

Note: When an instruction uses this operand, the constant, K16, is encoded in a 2-byte extension to the instruction. Because of the extension, an instruction using this operand cannot be executed in parallel with another instruction.

Example Syntax	Example Instruction	Address(es) Generated (Linear Addressing)	Description
MOV Smem, dst	MOV *+AR7(#9), AR3 (AR7 references a register.)	AR7H:(AR7 + 9) = XAR7 + 9	Before AR7 is used for the address, the constant is added to AR7. The CPU reads the value at address XAR7 + 9 and loads it into AR3.
MOV dbl(Lmem), pair(TAx)	MOV dbl(*+AR5(#40)), pair(AR2) (AR5 references a register.)	**First address:** AR5H:(AR5 + 40) = XAR5 + 40 **Second address:** If XAR5 + 40 is even (XAR5 + 40) + 1 If XAR5 + 40 is odd (XAR5 + 40) – 1	Before AR5 is used for the addresses, the constant is added to AR5. The CPU reads the values at address XAR5 + 40 and the following or preceding address, and loads them into AR2 and AR3, respectively. For details about the alignment of long words in data memory, see *Data Types* on page 3-5.

6.6.3.16 *ARn(short(#k3)) Used for Memory-Mapped Register Access

Operand	Description
*ARn(short(#k3))	Address generated: ARnH:([BSAyy +] ARn + k3) ARn is not modified. ARn is used as a base pointer. The 3-bit unsigned constant (k3) is used as an offset from that base pointer. k3 can be a number from 1 to 7.

Example Syntax	Example Instruction	Address(es) Generated (Linear Addressing)	Description
MOV Smem, dst	MOV *AR7(short(#2)), AR3 (AR7 references a register.)	AR7H:(AR7 + 2) = XAR7 + 2	The CPU reads the value at address XAR7 + 2 and loads it into AR3. AR7 is not modified.
MOV dbl(Lmem), pair(TAx)	MOV dbl(*AR5(short(#7))), pair(AR2) (AR5 references a register.)	**First address:** AR5H:(AR5 + 7) = XAR5 + 7 **Second address:** If XAR5 + 7 is even (XAR5+ 7) + 1 If XAR5 + 7 is odd (XAR5+ 7) – 1	The CPU reads the values at address XAR5 + 7 and the following or preceding address, and loads them into AR2 and AR3, respectively. AR5 is not modified. For details about the alignment of long words in data memory, see *Data Types* on page 3-5.

6.6.3.17 *CDP Used for Memory-Mapped Register Access

Operand	Description
*CDP	Address generated: CDPH:[BSAC +] CDP CDP is not modified.

Example Syntax	Example Instruction	Address(es) Generated (Linear Addressing)	Description
MOV Smem, dst	MOV *CDP, T2 (CDP references a register.)	CDPH:CDP = XCDP	The CPU reads the value at address XCDP and loads it into T2. CDP is not modified.
MOV dbl(Lmem), pair(TAx)	MOV dbl(*CDP), pair(T2) (CDP references a register.)	**First address:** XCDP **Second address:** If XCDP is even XCDP + 1 If XCDP is odd XCDP – 1	The CPU reads the values at address XCDP and the following or preceding address, and loads them into T2 and T3, respectively. CDP is not modified. For details about the alignment of long words in data memory, see *Data Types* on page 3-5.

6.6.3.18 *CDP+ Used for Memory-Mapped Register Access

Operand	Description
*CDP+	1) Address generated: CDPH:[BSAC +] CDP 2) CDP modified: If 16-bit operation: CDP = CDP + 1 If 32-bit operation: CDP = CDP + 2

Example Syntax	Example Instruction	Address(es) Generated (Linear Addressing)	Description
MOV Smem, dst	MOV *CDP+, T2 (CDP references a register.)	CDPH:CDP = XCDP	The CPU reads the value at address XCDP and loads it into T2. After being used for the address, CDP is incremented by 1.
MOV dbl(Lmem), pair(TAx)	MOV dbl(*CDP+), pair(T2) (CDP references a register.)	**First address:** XCDP **Second address:** If XCDP is even XCDP + 1 If XCDP is odd XCDP – 1	The CPU reads the values at address XCDP and the following or preceding address, and loads them into T2 and T3, respectively. After being used for the addresses, CDP is incremented by 2. For details about the alignment of long words in data memory, see *Data Types* on page 3-5.

6.6.3.19 *CDP– Used for Memory-Mapped Register Access

Operand	Description
*CDP–	1) Address generated: CDPH:[BSAC +] CDP 2) CDP modified: If 16-bit operation: CDP = CDP – 1 If 32-bit operation: CDP = CDP – 2

Example Syntax	Example Instruction	Address(es) Generated (Linear Addressing)	Description
MOV Smem, dst	MOV *CDP–, T2 (CDP references a register.)	CDPH:CDP = XCDP	The CPU reads the value at address XCDP and loads it into T2. After being used for the address, CDP is decremented by 1.
MOV dbl(Lmem), pair(TAx)	MOV dbl(*CDP–), pair(T2) (CDP references a register.)	**First address:** XCDP **Second address:** If XCDP is even XCDP + 1 If XCDP is odd XCDP – 1	The CPU reads the values at address XCDP and the following or preceding address, and loads them into T2 and T3, respectively. After being used for the addresses, CDP is decremented by 2. For details about the alignment of long words in data memory, see *Data Types* on page 3-5.

6.6.3.20 *CDP(#K16) Used for Memory-Mapped Register Access

Operand	Description
*CDP(#K16)	Address generated: CDPH:([BSAC +] CDP + K16) CDP is not modified. CDP is used as a base pointer. The 16-bit signed constant (K16) is used as an offset from that base pointer.

Note: When an instruction uses this operand, the constant, K16, is encoded in a 2-byte extension to the instruction. Because of the extension, an instruction using this operand cannot be executed in parallel with another instruction.

Example Syntax	Example Instruction	Address(es) Generated (Linear Addressing)	Description
MOV Smem, dst	MOV *CDP(#9), AR3 (CDP references a register.)	CDPH:(CDP + 9) = XCDP + 9	The CPU reads the value at address XCDP + 9 and loads it into AR3. CDP is not modified.
MOV dbl(Lmem), pair(TAx)	MOV dbl(*CDP(#40)), pair(AR2) (CDP references a register.)	**First address:** CDPH:(CDP + 40) = XCDP + 40 **Second address:** If XCDP + 40 is even (XCDP+ 40) + 1 If XCDP + 40 is odd (XCDP+ 40) – 1	The CPU reads the values at address XCDP + 40 and the following or preceding address, and loads them into AR2 and AR3, respectively. CDP is not modified. For details about the alignment of long words in data memory, see *Data Types* on page 3-5.

6.6.3.21 *+CDP(#K16) Used for Memory-Mapped Register Access

Operand	Description
*+CDP(#K16)	1) CDP modified: CDP = CDP + K16 2) Address generated: CDPH:([BSAC +] CDP + K16)

Note: When an instruction uses this operand, the constant, K16, is encoded in a 2-byte extension to the instruction. Because of the extension, an instruction using this operand cannot be executed in parallel with another instruction.

Example Syntax	Example Instruction	Address(es) Generated (Linear Addressing)	Description
MOV Smem, dst	MOV *+CDP(#9), AR3 (CDP references a register.)	CDPH:(CDP + 9) = XCDP + 9	Before CDP is used for the address, the constant is added to CDP. The CPU reads the value at address XCDP + 9 and loads it into AR3.
MOV dbl(Lmem), pair(TAx)	MOV dbl(*+CDP(#40)), pair(AR2) (CDP references a register.)	**First address:** CDPH:(CDP + 40) = XCDP + 40 **Second address:** If XCDP + 40 is even (XCDP + 40) + 1 If XCDP + 40 is odd (XCDP + 40) – 1	Before CDP is used for the addresses, the constant is added to CDP. The CPU reads the values at address XCDP + 40 and the following or preceding address, and loads them into AR2 and AR3, respectively. For details about the alignment of long words in data memory, see *Data Types* on page 3-5.

6.7 Restrictions on Accesses to Memory-Mapped Registers

In the instruction syntaxes listed in Table 6−19, Smem cannot reference a memory-mapped register (MMR). No instruction can access a byte within a memory-mapped register. If Smem is an MMR in one of the following syntaxes, the DSP sends a hardware bus-error interrupt (BERRINT) request to the CPU.

Table 6−19. Instructon Syntaxes in Which the Smem Operand Cannot be an MMR Reference

Syntax in Which Smem Cannot be an MMR Reference	Instruction Type
MOV [uns(]high_byte(Smem)[)], dst	Accumulator, Auxiliary, or Temporary Register Load
MOV [uns(]low_byte(Smem)[)], dst	
MOV high_byte(Smem) << #SHIFTW, ACx	
MOV low_byte(Smem) << #SHIFTW, ACx	
MOV src, high_byte(Smem)	Accumulator, Auxiliary, or Temporary Register Store
MOV src, low_byte(Smem)	

In the cases listed in Table 6−20, a memory-mapped register (MMR) cannot be referenced; otherwise, the DSP sends a hardware bus-error interrupt (BERRINT) request to the CPU.

Table 6−20. Cases in Which an MMR Reference Cannot be Made

Cases in Which an MMR Reference Cannot be Made	Note
Xmem of dual write	FAB bus cannot access MMRs.
Ymem of dual read	CAB bus cannot access MMRs.
Cmem of coefficient read	BAB bus cannot access MMRs.
Stack access	By call/return/intr/trap/push/pop

When a bus-error occurs, the functionality of the instruction that caused the error, and of any instruction executed in parallel, cannot be assured.

6.8 Addressing Register Bits

Direct addressing (see section 6.8.1) and indirect addressing (see section 6.8.2) can be used to address individual register bits or pairs of register bits. None of the absolute addressing modes support accesses to register bits. Any access way for I/O space can not be combined with these addressing register bits.

6.8.1 Addressing Register Bits With the Register-Bit Direct Addressing Mode

You can use the register-bit direct operand @bitoffset to access a register bit if an instruction has the following syntax element:

Baddr Indicates the address of one bit of data. Only the register bit test/set/clear/complement instructions support Baddr, and these instructions enable you to access bits in the following registers only: the accumulators (AC0–AC3), the auxiliary registers (AR0–AR7), and the temporary registers (T0–T3).

Table 6–21 provides examples of using @bitoffset to access register bits.

Table 6–21. @bitoffset Used for Register-Bit Access

Example Syntax	Example Instruction	Bit Address(es) Generated	Description
BSET Baddr, src	BSET @0, AC3	0	The CPU sets bit 0 of AC3.
BTSTP Baddr, src	BTSTP @30, AC3	30 and 31	The CPU tests bits 30 and 31 of AC3. It copies the content of AC0(30) into the TC1 bit of status register ST0_55, and it copies the content of AC0(31) into the TC2 bit of ST0_55.

6.8.2 Addressing Register Bits With Indirect Addressing Modes

You can use indirect operands to access register bits if an instruction has the following syntax element:

Baddr Indicates the address of one bit of data. Only the register bit test/set/clear/complement instructions support Baddr, and these instructions enable you to access bits in the following registers only: the accumulators (AC0–AC3), the auxiliary registers (AR0–AR7), and the temporary registers (T0–T3).

You must first make sure the pointer contains the correct bit number. For example, if AR6 is the pointer and you want to access bit 15 of a register, you could use the following instruction to initialize AR6:

```
MOV #15, AR6
```

Figure 6−15 lists the indirect operands that support accesses of register bits, and the sections following the table provide details and examples for the operands.

Figure 6−15. Indirect Operands for Register-Bit Accesses

AR indirect addressing mode:

DSP mode (ARMS = 0):	Control mode (ARMS = 1):
*ARn	*ARn
*ARn+	*ARn+
*ARn−	*ARn−
*+ARn	
*−ARn	
*(ARn + T0/AR0)	*(ARn + T0/AR0)
*(ARn − T0/AR0)	*(ARn − T0/AR0)
*ARn(T0/AR0)	*ARn(T0/AR0)
*(ARn + T0B/AR0B)	
*(ARn − T0B/AR0B)	
*(ARn + T1)	
*(ARn − T1)	
*ARn(T1)	
*ARn(#K16)	*ARn(#K16)
*+ARn(#K16)	*+ARn(#K16)
	*ARn(short(#k3))

CDP indirect addressing mode:

*CDP
*CDP+
*CDP−
*CDP(#k16)
*+CDP(#k16)

6.8.2.1 *ARn Used for Register-Bit Access

Operand	Description
*ARn	Bit address generated: [BSAyy +] ARn ARn is not modified.

Example Syntax	Example Instruction	Address(es) Generated (Linear Addressing)	Description
BSET Baddr, src	BSET *AR2, AC3	AR2 Assume AR2 = 0.	The CPU sets bit 0 of AC3. AR2 is not modified.
BTSTP Baddr, src	BTSTP *AR5, AC3	AR5 AR5 + 1 Assume AR5 = 30.	The CPU tests bits 30 and 31 of AC3. It copies the content of AC3(30) into the TC1 bit of status register ST0_55, and it copies the content of AC3(31) into the TC2 bit of ST0_55. AR5 is not modified.

6.8.2.2 *ARn+ Used for Register-Bit Access

Operand	Description
*ARn+	1) Bit address generated: [BSAyy +] ARn 2) ARn modified: If 1-bit operation: ARn = ARn + 1 If 2-bit operation: ARn = ARn + 2

Example Syntax	Example Instruction	Address(es) Generated (Linear Addressing)	Description
BSET Baddr, src	BSET *AR2+, AC3	AR2 Assume AR2 = 0.	The CPU sets bit 0 of AC3. After being used for the address. AR2 is incremented by 1.
BTSTP Baddr, src	BTSTP *AR5+, AC3	AR5 AR5 + 1 Assume AR5 = 30.	The CPU tests bits 30 and 31 of AC3. It copies the content of AC3(30) into the TC1 bit of status register ST0_55, and it copies the content of AC3(31) into the TC2 bit of ST0_55. After being used for the addresses. AR5 is incremented by 2.

6.8.2.3 *ARn– Used for Register-Bit Access

Operand	Description
*ARn–	1) Bit address generated: [BSAyy +] ARn 2) ARn modified: If 1-bit operation: ARn = ARn – 1 If 2-bit operation: ARn = ARn – 2

Example Syntax	Example Instruction	Address(es) Generated (Linear Addressing)	Description
BSET Baddr, src	BSET *AR2–, AC3	AR2 Assume AR2 = 0.	The CPU sets bit 0 of AC3. After being used for the address, AR2 is decremented by 1.
BTSTP Baddr, src	BTSTP *AR5–, AC3	AR5 AR5 + 1 Assume AR5 = 30.	The CPU tests bits 30 and 31 of AC3. It copies the content of AC3(30) into the TC1 bit of status register ST0_55, and it copies the content of AC3(31) into the TC2 bit of ST0_55. After being used for the addresses, AR5 is decremented by 2.

6.8.2.4 *+ARn Used for Register-Bit Access

Operand	Description
*+ARn	1) ARn modified: If 1-bit operation: ARn = ARn + 1 If 2-bit operation: ARn = ARn + 2 2) Bit address generated: If 1-bit operation: [BSAyy +] ARn + 1 If 2-bit operation: [BSAyy +] ARn + 2

Example Syntax	Example Instruction	Address(es) Generated (Linear Addressing)	Description
BSET Baddr, src	BSET *+AR2, AC3	(AR2 + 1) Assume (AR2 + 1) = 1.	Before being used for the address, AR2 is incremented by 1. The CPU sets bit 1 of AC3.
BTSTP Baddr, src	BTSTP *+AR5, AC3	(AR5 + 2) (AR5 + 2) + 1 Assume (AR5 + 2) = 30.	Before being used for the addresses, AR5 is incremented by 2. The CPU tests bits 30 and 31 of AC3. It copies the content of AC3(30) into the TC1 bit of status register ST0_55, and it copies the content of AC3(31) into the TC2 bit of ST0_55.

6.8.2.5 *–ARn Used for Register-Bit Access

Operand	Description
*–ARn	1) ARn modified: If 1-bit operation: ARn = ARn – 1 If 2-bit operation: ARn = ARn – 2 2) Bit address generated: If 1-bit operation: [BSAyy +] ARn – 1 If 2-bit operation: [BSAyy +] ARn – 2

Example Syntax	Example Instruction	Address(es) Generated (Linear Addressing)	Description
BSET Baddr, src	BSET *–AR2, AC3	(AR2 – 1) Assume (AR2 – 1) = 1.	Before being used for the address, AR2 is decremented by 1. The CPU sets bit 1 of AC3.
BTSTP Baddr, src	BTSTP *–AR5, AC3	(AR5 – 2) (AR5 – 2) + 1 Assume (AR5 – 2) = 30.	Before being used for the addresses, AR5 is decremented by 2. The CPU tests bits 30 and 31 of AC3. It copies the content of AC3(30) into the TC1 bit of status register ST0_55, and it copies the content of AC3(31) into the TC2 bit of ST0_55.

6.8.2.6 *(ARn + T0/AR0) Used for Register-Bit Access

C54CM = 0		C54CM = 1	
Operand	**Description**	**Operand**	**Description**
*(ARn + T0)	1) Bit address generated: [BSAyy +] ARn 2) ARn modified: ARn = ARn + T0	*(ARn + AR0)	1) Bit address generated: [BSAyy +] ARn 2) ARn modified: ARn = ARn + AR0

Example Syntax	Example Instruction	Address(es) Generated (Linear Addressing)	Description
BSET Baddr, src	BSET *(AR2 + T0), AC3	AR2 Assume AR2 = 0.	The CPU sets bit 0 of AC3. After being used for the address, AR2 is incremented by the number in T0.
BTSTP Baddr, src	BTSTP *(AR5 + T0), AC3	AR5 AR5 + 1 Assume AR5 = 30.	The CPU tests bits 30 and 31 of AC3. It copies the content of AC3(30) into the TC1 bit of status register ST0_55, and it copies the content of AC3(31) into the TC2 bit of ST0_55. After being used for the addresses, AR5 is incremented by the number in T0.

6.8.2.7 *(ARn – T0/AR0) Used for Register-Bit Access

C54CM = 0		C54CM = 1	
Operand	**Description**	**Operand**	**Description**
*(ARn – T0)	1) Bit address generated: [BSAyy +] ARn 2) ARn modified: ARn = ARn – T0	*(ARn – AR0)	1) Bit address generated: [BSAyy +] ARn 2) ARn modified: ARn = ARn – AR0

Example Syntax	Example Instruction	Address(es) Generated (Linear Addressing)	Description
BSET Baddr, src	BSET *(AR2 – T0), AC3	AR2 Assume AR2 = 0.	The CPU sets bit 0 of AC3. After being used for the address, AR2 is decremented by the number in T0.
BTSTP Baddr, src	BTSTP *(AR5 – T0), AC3	AR5 AR5 + 1 Assume AR5 = 30.	The CPU tests bits 30 and 31 of AC3. It copies the content of AC3(30) into the TC1 bit of status register ST0_55, and it copies the content of AC3(31) into the TC2 bit of ST0_55. After being used for the addresses, AR5 is decremented by the number in T0.

6.8.2.8 *ARn(T0/AR0) Used for Register-Bit Access

C54CM = 0		C54CM = 1	
Operand	**Description**	**Operand**	**Description**
*ARn(T0)	Bit address generated: [BSAyy +] ARn + T0 ARn is not modified. ARn is used as a base pointer. T0 is used as an offset from that base pointer.	*ARn(AR0)	Bit address generated: [BSAyy +] ARn + AR0 ARn is not modified. ARn is used as a base pointer. AR0 is used as an offset from that base pointer.

Example Syntax	Example Instruction	Address(es) Generated (Linear Addressing)	Description
BSET Baddr, src	BSET *AR2(T0), AC3	(AR2 + T0) Assume AR2 = 0 and T0 = 15.	The CPU sets bit 15 of AC3. AR2 is not modified.
BTSTP Baddr, src	BTSTP *AR5(T0), AC3	(AR5 + T0) (AR5 + T0) + 1 Assume AR5 = 25 and T0 = 5.	The CPU tests bits 30 and 31 of AC3. It copies the content of AC3(30) into the TC1 bit of status register ST0_55, and it copies the content of AC3(31) into the TC2 bit of ST0_55. AR5 is not modified.

6.8.2.9 *(ARn + T0B/AR0B) Used for Register-Bit Access

C54CM = 0		C54CM = 1	
Operand	**Description**	**Operand**	**Description**
*(ARn + T0B)	1) Bit address generated: [BSAyy +] ARn 2) ARn modified: ARn = ARn + T0 (done with reverse carry propagation) See **Note** about circular addressing restriction.	*(ARn + AR0B)	1) Bit address generated: [BSAyy +] ARn 2) ARn modified: ARn = ARn + AR0 (done with reverse carry propagation) See **Note** about circular addressing restriction.

Note: When this bit-reverse operand is used, ARn cannot be used as a circular pointer. If ARn is configured in ST2_55 for circular addressing, the corresponding buffer start address register value (BSAyy) is added to ARn, but ARn is not modified so as to remain inside a circular buffer.

Example Syntax	Example Instruction	Address(es) Generated (Linear Addressing)	Description
BSET Baddr, src	BSET *(AR2 + T0B), AC3	AR2 Assume AR2 = 0.	The CPU sets bit 0 of AC3. After being used for the address, AR2 is incremented by the number T0. Reverse carry propagation is used during the addition.
BTSTP Baddr, src	BTSTP *(AR5 + T0B), AC3	AR5 AR5 + 1 Assume AR5 = 30.	The CPU tests bits 30 and 31 of AC3. It copies the content of AC3(30) into the TC1 bit of status register ST0_55, and it copies the content of AC3(31) into the TC2 bit of ST0_55. After being used for the addresses, AR5 is incremented by the number T0. Reverse carry propagation is used during the addition.

6.8.2.10 *(ARn – T0B/AR0B) Used for Register-Bit Access

C54CM = 0		C54CM = 1	
Operand	**Description**	**Operand**	**Description**
*(ARn – T0B)	1) Bit address generated: [BSAyy +] ARn 2) ARn modified: ARn = ARn – T0 (done with reverse carry propagation) See **Note** about circular addressing restriction.	*(ARn – AR0B)	1) Bit address generated: [BSAyy +] ARn 2) ARn modified: ARn = ARn – AR0 (done with reverse carry propagation) See **Note** about circular addressing restriction.

Note: When this bit-reverse operand is used, ARn cannot be used as a circular pointer. If ARn is configured in ST2_55 for circular addressing, the corresponding buffer start address register value (BSAyy) is added to ARn, but ARn is not modified so as to remain inside a circular buffer.

Example Syntax	Example Instruction	Address(es) Generated (Linear Addressing)	Description
BSET Baddr, src	BSET *(AR2 – T0B), AC3	AR2 Assume AR2 = 0.	The CPU sets bit 0 of AC3. After being used for the address, AR2 is decremented by the number in T0. Reverse carry propagation is used during the subtraction.
BTSTP Baddr, src	BTSTP *(AR5 – T0B), AC3	AR5 AR5 + 1 Assume AR5 = 30.	The CPU tests bits 30 and 31 of AC3. It copies the content of AC3(30) into the TC1 bit of status register ST0_55, and it copies the content of AC3(31) into the TC2 bit of ST0_55. After being used for the addresses, AR5 is decremented by the number in T0. Reverse carry propagation is used during the subtraction.

6.8.2.11 *(ARn + T1) Used for Register-Bit Access

Operand	Description
*(ARn + T1)	1) Bit address generated: [BSAyy +] ARn 2) ARn modified: ARn = ARn + T1

Example Syntax	Example Instruction	Address(es) Generated (Linear Addressing)	Description
BCLR Baddr, src	BCLR *(AR4 + T1), AC2	AR4 Assume AR4 = 0.	The CPU clears bit 0 of AC2. After being used for the address, AR4 is incremented by the number in T1.
BTSTP Baddr, src	BTSTP *(AR1 + T1), AC2	AR1 AR1 + 1 Assume AR1 = 30.	The CPU tests bits 30 and 31 of AC2. It copies the content of AC2(30) into the TC1 bit of status register ST0_55, and it copies the content of AC2(31) into the TC2 bit of ST0_55. After being used for the addresses, AR1 is incremented by the number in T1.

6.8.2.12 *(ARn – T1) Used for Register-Bit Access

Operand	Description
*(ARn – T1)	1) Bit address generated: [BSAyy +] ARn 2) ARn modified: ARn = ARn – T1

Example Syntax	Example Instruction	Address(es) Generated (Linear Addressing)	Description
BCLR Baddr, src	BCLR *(AR4 – T1), AC2	AR4 Assume AR4 = 0.	The CPU clears bit 0 of AC2. After being used for the address, AR4 is decremented by the number in T1.
BTSTP Baddr, src	BTSTP *(AR1 – T1), AC2	AR1 AR1 + 1 Assume AR1 = 30.	The CPU tests bits 30 and 31 of AC2. It copies the content of AC2(30) into the TC1 bit of status register ST0_55, and it copies the content of AC2(31) into the TC2 bit of ST0_55. After being used for the addresses, AR1 is decremented by the number in T1.

6.8.2.13 *ARn(T1) Used for Register-Bit Access

Operand	Description
*ARn(T1)	Bit address generated: [BSAyy +] ARn + T1 ARn is not modified. ARn is used as a base pointer. T1 is used as an offset from that base pointer.

Example Syntax	Example Instruction	Address(es) Generated (Linear Addressing)	Description
BCLR Baddr, src	BCLR *AR4(T1), AC2	(AR4 + T1) Assume AR4 = 0 and T1 = 15.	The CPU clears bit 15 of AC2. AR4 is not modified.
BTSTP Baddr, src	BTSTP *AR1(T1), AC2	(AR1 + T1) (AR1+ T1) + 1 Assume AR1 = 25 and T1 = 5.	The CPU tests bits 30 and 31 of AC2. It copies the content of AC2(30) into the TC1 bit of status register ST0_55, and it copies the content of AC2(31) into the TC2 bit of ST0_55. AR1 is not modified.

6.8.2.14 *ARn(#K16) Used for Register-Bit Access

Operand	Description
*ARn(#K16)	Bit address generated: [BSAyy +] ARn + K16 ARn is not modified. ARn is used as a base pointer. The 16-bit signed constant (K16) is used as an offset from that base pointer.

Note: When an instruction uses this operand, the constant, K16, is encoded in a 2-byte extension to the instruction. Because of the extension, an instruction using this operand cannot be executed in parallel with another instruction.

Example Syntax	Example Instruction	Address(es) Generated (Linear Addressing)	Description
BCLR Baddr, src	BCLR *AR4(#31), AC2	(AR4 + 31) Assume AR4 = 0.	The CPU clears bit 31 of AC2. AR4 is not modified.
BTSTP Baddr, src	BTSTP *AR1(#5), AC2	(AR1 + 5) (AR1+ 5) + 1 Assume AR1 = 16.	The CPU tests bits 21 and 22 of AC2. It copies the content of AC2(21) into the TC1 bit of status register ST0_55, and it copies the content of AC2(22) into the TC2 bit of ST0_55. AR1 is not modified.

6.8.2.15 *+ARn(#K16) Used for Register-Bit Access

Operand	Description
*+ARn(#K16)	1) ARn modified: ARn = ARn + K16 2) Bit address generated: [BSAyy +] ARn + K16

Note: When an instruction uses this operand, the constant, K16, is encoded in a 2-byte extension to the instruction. Because of the extension, an instruction using this operand cannot be executed in parallel with another instruction.

Example Syntax	Example Instruction	Address(es) Generated (Linear Addressing)	Description
BCLR Baddr, src	BCLR *+AR4(#31), AC2	(AR4 + 31) Assume AR4 = 0.	Before AR4 is used for the address, the constant is added to AR4. The CPU clears bit 31 of AC2.
BTSTP Baddr, src	BTSTP *+AR1(#5), AC2	(AR1 + 5) (AR1 + 5) + 1 Assume AR1 = 16.	Before AR1 is used for the addresses, the constant is added to AR1. The CPU tests bits 21 and 22 of AC2. It copies the content of AC2(21) into the TC1 bit of status register ST0_55, and it copies the content of AC2(22) into the TC2 bit of ST0_55.

6.8.2.16 *ARn(short(#k3)) Used for Register-Bit Access

Operand	Description
*ARn(short(#k3))	Bit address generated: [BSAyy +] ARn + k3 ARn is not modified. ARn is used as a base pointer. The 3-bit unsigned constant (k3) is used as an offset from that base pointer. k3 can be a number from 1 to 7.

Example Syntax	Example Instruction	Address(es) Generated (Linear Addressing)	Description
BCLR Baddr, src	BCLR *AR4(short(#3)), AC2	(AR4 + 3) Assume AR4 = 0.	The CPU clears bit 3 of AC2. AR4 is not modified.
BTSTP Baddr, src	BTSTP *AR1(short(#5)), AC2	(AR1 + 5) (AR1+ 5) + 1 Assume AR1 = 16.	The CPU tests bits 21 and 22 of AC2. It copies the content of AC2(21) into the TC1 bit of status register ST0_55, and it copies the content of AC2(22) into the TC2 bit of ST0_55. AR1 is not modified.

6.8.2.17 *CDP Used for Register-Bit Access

Operand	Description
*CDP	Bit address generated: [BSAC +] CDP CDP is not modified.

Example Syntax	Example Instruction	Address(es) Generated (Linear Addressing)	Description
BSET Baddr, src	BSET *CDP, AC3	CDP Assume CDP = 0.	The CPU sets bit 0 of AC3. CDP is not modified.
BTSTP Baddr, src	BTSTP *CDP, AC3	CDP CDP + 1 Assume CDP = 30.	The CPU tests bits 30 and 31 of AC3. It copies the content of AC3(30) into the TC1 bit of status register ST0_55, and it copies the content of AC3(31) into the TC2 bit of ST0_55. CDP is not modified.

6.8.2.18 *CDP+ Used for Register-Bit Access

Operand	Description
*CDP+	1) Bit address generated: [BSAC +] CDP 2) CDP modified: If 1-bit operation: CDP = CDP + 1 If 2-bit operation: CDP = CDP + 2

Example Syntax	Example Instruction	Address(es) Generated (Linear Addressing)	Description
BSET Baddr, src	BSET *CDP+, AC3	CDP Assume CDP = 0.	The CPU sets bit 0 of AC3. After being used for the address, CDP is incremented by 1.
BTSTP Baddr, src	BTSTP *CDP+, AC3	CDP CDP + 1 Assume CDP = 30.	The CPU tests bits 30 and 31 of AC3. It copies the content of AC3(30) into the TC1 bit of status register ST0_55, and it copies the content of AC3(31) into the TC2 bit of ST0_55. After being used for the addresses, CDP is incremented by 2.

6.8.2.19 *CDP– Used for Register-Bit Access

Operand	Description
*CDP–	1) Bit address generated: [BSAC +] CDP 2) CDP modified: If 1-bit operation: CDP = CDP – 1 If 2-bit operation: CDP = CDP – 2

Example Syntax	Example Instruction	Address(es) Generated (Linear Addressing)	Description
BSET Baddr, src	BSET *CDP–, AC3	CDP Assume CDP = 0.	The CPU sets bit 0 of AC3. After being used for the address, CDP is decremented by 1.
BTSTP Baddr, src	BTSTP *CDP–, AC3	CDP CDP + 1 Assume CDP = 30.	The CPU tests bits 30 and 31 of AC3. It copies the content of AC3(30) into the TC1 bit of status register ST0_55, and it copies the content of AC3(31) into the TC2 bit of ST0_55. After being used for the addresses, CDP is decremented by 2.

6.8.2.20 *CDP(#K16) Used for Register-Bit Access

Operand	Description
*CDP(#K16)	Bit address generated: [BSAC +] CDP + K16 CDP is not modified. CDP is used as a base pointer. The 16-bit signed constant (K16) is used as an offset from that base pointer.

Note: When an instruction uses this operand, the constant, K16, is encoded in a 2-byte extension to the instruction. Because of the extension, an instruction using this operand cannot be executed in parallel with another instruction.

Example Syntax	Example Instruction	Address(es) Generated (Linear Addressing)	Description
BCLR Baddr, src	BCLR *CDP(#31), AC2	(CDP + 31) Assume CDP = 0.	The CPU clears bit 31 of AC2. CDP is not modified.
BTSTP Baddr, src	BTSTP *CDP(#5), AC2	(CDP + 5) (CDP+ 5) + 1 Assume CDP = 16.	The CPU tests bits 21 and 22 of AC2. It copies the content of AC2(21) into the TC1 bit of status register ST0_55, and it copies the content of AC2(22) into the TC2 bit of ST0_55. CDP is not modified.

6.8.2.21 *+CDP(#K16) Used for Register-Bit Access

Operand	Description
*+CDP(#K16)	1) CDP modified: CDP = CDP + K16 2) Bit address generated: [BSAC +] CDP + K16

Note: When an instruction uses this operand, the constant, K16, is encoded in a 2-byte extension to the instruction. Because of the extension, an instruction using this operand cannot be executed in parallel with another instruction.

Example Syntax	Example Instruction	Address(es) Generated (Linear Addressing)	Description
BCLR Baddr, src	BCLR *+CDP(#31), AC2	(CDP + 31) Assume CDP = 0.	Before CDP is used for the address, the constant is added to CDP. The CPU clears bit 31 of AC2.
BTSTP Baddr, src	BTSTP *+CDP(#5), AC2	(CDP + 5) (CDP + 5) + 1 Assume CDP = 16.	Before CDP is used for the addresses, the constant is added to CDP. The CPU tests bits 21 and 22 of AC2. It copies the content of AC2(21) into the TC1 bit of status register ST0_55, and it copies the content of AC2(22) into the TC2 bit of ST0_55.

6.9 Addressing I/O Space

Absolute, direct, and indirect addressing can be used to address the peripheral registers in I/O space:

Addressing Type	See ...
Absolute	This page
Direct	Page 6-98
Indirect	Page 6-98

There are some restrictions on accesses to I/O space. See section 6.10 for the details.

6.9.1 Addressing I/O Space With the I/O Absolute Addressing Mode

The defined operand *port(#k16) of the algebraic syntax or the qualified operand port(#k16) of the mnemonic syntax can be used to access only I/O space. You can use this operand in any instruction with the following syntax element:

Smem Indicates one word (16 bits) of data

Table 6–22 provides examples of using *port(#k16) or port(#k16) to access locations in I/O space.

Note:

When an instruction uses *port(#k16) or port(#k16), the unsigned 16-bit constant, k16, is encoded in a 2-byte extension to the instruction. Because of the extension, an instruction using this operand cannot be executed in parallel with another instruction.

*Table 6–22. *port(#k16) or port(#k16) Used for I/O-Space Access*

Example Syntax	Example Instruction	Address(es) Generated	Description
dst = Smem or MOV Smem, dst	AR2 = *port(#2) or MOV port(#2), AR2	k16 = 0002h	The CPU loads the value at I/O address 0002h into AR2.
Smem = src or MOV src, Smem	*port(#0F000h) = AR0 or MOV AR0, port(#0F000h)	k16 = F000h	The CPU stores the content of AR0 to the location at I/O address F000h.

6.9.2 Addressing I/O Space With the PDP Direct Addressing Mode

You can use the PDP direct operand @Poffset to access I/O space if an instruction has the following syntax element:

Smem Indicates one word (16 bits) of data

You must use a port() qualifier to indicate that you are accessing an I/O-space location rather than a data-memory location. If you use algebraic instructions, you place the readport() instruction qualifier or the writeport() instruction qualifier in parallel with the instruction that performs the I/O-space access. If you use mnemonic instructions, port() must enclose the qualified read or write operand.

Table 6–23 provides examples of using @Poffset and the port() qualifier to access I/O space. The 9-bit peripheral data page (PDP) value is concatenated with the 7 bits of Poffset.

Table 6–23. @Poffset Used for I/O-Space Access

Example Syntax	Example Instruction	Address Generated For PDP = 511	Description
MOV Smem, dst	MOV port(@0), T2	PDP:Poffset = FF80h	An offset of 0 indicates the top of the current peripheral data page. The CPU copies the value at the top of peripheral data page 511 (address FF80h) and loads it into T2.
MOV src, Smem	MOV T2, port(@127)	PDP:Poffset = FFFFh	An offset of 127 indicates the bottom of the current peripheral data page. The CPU copies the content of T2 and writes it to the bottom of peripheral data page 511 (address FFFFh).

6.9.3 Addressing I/O Space With Indirect Addressing Modes

You can use an indirect operand to access I/O space if an instruction has the following syntax element:

Smem Indicates one word (16 bits) of data

You must use a port() qualifier to indicate that you are accessing an I/O-space location rather than a data-memory location. If you use algebraic instructions, you place the readport() instruction qualifier or the writeport() instruction qualifier in parallel with the instruction that performs the I/O-space access. If you use mnemonic instructions, port() must enclose the qualified read or write operand.

Figure 6–16 lists the indirect operands that support accesses to I/O space, and the sections following the table provide details and examples for the operands. These examples use mnemonic instructions.

Figure 6−16. Indirect Operands for I/O-Space Accesses

AR indirect addressing mode:

DSP mode (ARMS = 0):	Control mode (ARMS = 1):
*ARn	*ARn
*ARn+	*ARn+
*ARn−	*ARn−
*+ARn	
*−ARn	
*(ARn + T0/AR0)	*(ARn + T0/AR0)
*(ARn − T0/AR0)	*(ARn − T0/AR0)
*ARn(T0/AR0)	*ARn(T0/AR0)
*(ARn + T0B/AR0B)	
*(ARn − T0B/AR0B)	
*(ARn + T1)	
*(ARn − T1)	
*ARn(T1)	
	*ARn(short(#k3))

CDP indirect addressing mode:

*CDP
*CDP+
*CDP−

6.9.3.1 *ARn Used for I/O-Space Access

Operand	Description
*ARn	I/O address generated: [BSAyy +] ARn ARn is not modified.

Example Syntax	Example Instruction	Address Generated (Linear Addressing)	Description
MOV Smem, dst	MOV port(*AR4), T2	AR4 Assume AR4 = FF80h.	The CPU reads the value at I/O address FF80h and loads it into T2. AR4 is not modified.
MOV src, Smem	MOV T2, port(*AR5)	AR5 Assume AR5 = FFFFh.	The CPU reads the content of T2 and writes it to I/O address FFFFh. AR5 is not modified.

6.9.3.2 *ARn+ Used for I/O-Space Access

Operand	Description
*ARn+	1) I/O address generated: [BSAyy +] ARn 2) ARn modified: ARn = ARn + 1

Example Syntax	Example Instruction	Address Generated (Linear Addressing)	Description
MOV Smem, dst	MOV port(*AR4+), T2	AR4 Assume AR4 = FF80h.	The CPU reads the value at I/O address FF80h and loads it into T2. After being used for the address, AR4 is incremented by 1.
MOV src, Smem	MOV T2, port(*AR5+)	AR5 Assume AR5 = FFFFh.	The CPU reads the content of T2 and writes it to I/O address FFFFh. After being used for the address, AR5 is incremented by 1.

6.9.3.3 *ARn– Used for I/O-Space Access

Operand	Description
*ARn–	1) I/O address generated: [BSAyy +] ARn 2) ARn modified: ARn = ARn – 1

Example Syntax	Example Instruction	Address Generated (Linear Addressing)	Description
MOV Smem, dst	MOV port(*AR4–), T2	AR4 Assume AR4 = FF80h.	The CPU reads the value at I/O address FF80h and loads it into T2. After being used for the address, AR4 is decremented by 1.
MOV src, Smem	MOV T2, port(*AR5–)	AR5 Assume AR5 = FFFFh.	The CPU reads the content of T2 and writes it to I/O address FFFFh. After being used for the address, AR5 is decremented by 1.

6.9.3.4 *+ARn Used for I/O-Space Access

Operand	Description
*+ARn	1) ARn modified: ARn = ARn + 1 2) I/O address generated: [BSAyy +] ARn + 1

Example Syntax	Example Instruction	Address Generated (Linear Addressing)	Description
MOV Smem, dst	MOV port(*+AR4), T2	(AR4 + 1) Assume AR4 = FF7Fh.	Before being used for the address, AR4 is incremented by 1. The CPU reads the value at I/O address FF80h and loads it into T2.
MOV src, Smem	MOV T2, port(*+AR5)	(AR5 + 1) Assume AR5 = FFFEh.	Before being used for the address, AR5 is incremented by 1. The CPU reads the content of T2 and writes it to I/O address FFFFh.

6.9.3.5 *–ARn Used for I/O-Space Access

Operand	Description
*–ARn	1) ARn modified: ARn = ARn – 1 2) I/O address generated: [BSAyy +] ARn – 1

Example Syntax	Example Instruction	Address Generated (Linear Addressing)	Description
MOV Smem, dst	MOV port(*–AR4), T2	(AR4 – 1) Assume AR4 = FF80h.	Before being used for the address, AR4 is decremented by 1. The CPU reads the value at I/O address FF7Fh and loads it into T2.
MOV src, Smem	MOV T2, port(*–AR5)	(AR5 – 1) Assume AR5 = FFFFh.	Before being used for the address, AR5 is decremented by 1. The CPU reads the content of T2 and writes it to I/O address FFFEh.

6.9.3.6 *(ARn + T0/AR0) Used for I/O-Space Access

C54CM = 0		C54CM = 1	
Operand	**Description**	**Operand**	**Description**
*(ARn + T0)	1) I/O address generated: [BSAyy +] ARn 2) ARn modified: ARn = ARn + T0	*(ARn + AR0)	1) I/O address generated: [BSAyy +] ARn 2) ARn modified: ARn = ARn + AR0

Example Syntax	Example Instruction	Address Generated (Linear Addressing)	Description
MOV Smem, dst	MOV port(*(AR4 + T0)), T2	AR4 Assume AR4 = FF80h.	The CPU reads the value at I/O address FF80h and loads it into T2. After being used for the address, AR4 is incremented by the number in T0.
MOV src, Smem	MOV T2, port(*(AR5 + T0))	AR5 Assume AR5 = FFFFh.	The CPU reads the content of T2 and writes it to I/O address FFFFh. After being used for the address, AR5 is incremented by the number in T0.

6.9.3.7 *(ARn – T0/AR0) Used for I/O-Space Access

C54CM = 0		C54CM = 1	
Operand	**Description**	**Operand**	**Description**
*(ARn – T0)	1) I/O address generated: [BSAyy +] ARn 2) ARn modified: ARn = ARn – T0	*(ARn – AR0)	1) I/O address generated: [BSAyy +] ARn 2) ARn modified: ARn = ARn – AR0

Example Syntax	Example Instruction	Address Generated (Linear Addressing)	Description
MOV Smem, dst	MOV port(*(AR4 – T0)), T2	AR4 Assume AR4 = FF80h.	The CPU reads the value at I/O address FF80h and loads it into T2. After being used for the address, AR4 is decremented by the number in T0.
MOV src, Smem	MOV T2, port(*(AR5 – T0))	AR5 Assume AR5 = FFFFh.	The CPU reads the content of T2 and writes it to I/O address FFFFh. After being used for the address, AR5 is decremented by the number in T0.

6.9.3.8 *ARn(T0/AR0) Used for I/O-Space Access

C54CM = 0		C54CM = 1	
Operand	**Description**	**Operand**	**Description**
*ARn(T0)	I/O address generated: [BSAyy +] ARn + T0 ARn is not modified. ARn is used as a base pointer. T0 is used as an offset from that base pointer.	*ARn(AR0)	I/O address generated: [BSAyy +] ARn + AR0 ARn is not modified. ARn is used as a base pointer. AR0 is used as an offset from that base pointer.

Example Syntax	Example Instruction	Address Generated (Linear Addressing)	Description
MOV Smem, dst	MOV port(*AR4(T0)), T2	(AR4 + T0) Assume AR4 = FF7Dh and T0 = 3.	The CPU reads the value at I/O address FF80h and loads it into T2. AR4 is not modified.
MOV src, Smem	MOV T2, port(*AR5(T0))	(AR5 + T0) Assume AR5 = FFFAh and T0 = 5.	The CPU reads the content of T2 and writes it to I/O address FFFFh. AR5 is not modified.

6.9.3.9 *(ARn + T0B/AR0B) Used for I/O-Space Access

C54CM = 0		C54CM = 1	
Operand	**Description**	**Operand**	**Description**
*(ARn + T0B)	1) I/O address generated: [BSAyy +] ARn 2) ARn modified: ARn = ARn + T0 (done with reverse carry propagation) See **Note** about circular addressing restriction.	*(ARn + AR0B)	1) I/O address generated: [BSAyy +] ARn 2) ARn modified: ARn = ARn + AR0 (done with reverse carry propagation) See **Note** about circular addressing restriction.

Note: When this bit-reverse operand is used, ARn cannot be used as a circular pointer. If ARn is configured in ST2_55 for circular addressing, the corresponding buffer start address register value (BSAyy) is added to ARn, but ARn is not modified so as to remain inside a circular buffer.

Example Syntax	Example Instruction	Address Generated (Linear Addressing)	Description
MOV Smem, dst	MOV port(*(AR4 + T0B)), T2	AR4 Assume AR4 = FF80h.	The CPU reads the value at I/O address FF80h and loads it into T2. After being used for the address, AR4 is incremented by the number in T0. Reverse carry propagation is used during the addition.
MOV src, Smem	MOV T2, port(*(AR5 + T0B))	AR5 Assume AR5 = FFFFh.	The CPU reads the content of T2 and writes it to I/O address FFFFh. After being used for the address, AR5 is incremented by the number in T0. Reverse carry propagation is used during the addition.

6.9.3.10 *(ARn – T0B/AR0B) Used for I/O-Space Access

C54CM = 0		C54CM = 1	
Operand	**Description**	**Operand**	**Description**
*(ARn – T0B)	1) I/O address generated: [BSAyy +] ARn 2) ARn modified: ARn = ARn – T0 (done with reverse carry propagation) See **Note** about circular addressing restriction.	*(ARn – AR0B)	1) I/O address generated: [BSAyy +] ARn 2) ARn modified: ARn = ARn – AR0 (done with reverse carry propagation) See **Note** about circular addressing restriction.

Note: When this bit-reverse operand is used, ARn cannot be used as a circular pointer. If ARn is configured in ST2_55 for circular addressing, the corresponding buffer start address register value (BSAyy) is added to ARn, but ARn is not modified so as to remain inside a circular buffer.

Example Syntax	Example Instruction	Address Generated (Linear Addressing)	Description
MOV Smem, dst	MOV port(*(AR4 – T0B)), T2	AR4 Assume AR4 = FF80h.	The CPU reads the value at I/O address FF80h and loads it into T2. After being used for the address, AR4 is decremented by the number in T0. Reverse carry propagation is used during the subtraction.
MOV src, Smem	MOV T2, port(*(AR5 – T0B))	AR5 Assume AR5 = FFFFh.	The CPU reads the content of T2 and writes it to I/O address FFFFh. After being used for the address, AR5 is decremented by the number in T0. Reverse carry propagation is used during the subtraction.

6.9.3.11 *(ARn + T1) Used for I/O-Space Access

Operand	Description
*(ARn + T1)	1) I/O address generated: [BSAyy +] ARn 2) ARn modified: ARn = ARn + T1

Example Syntax	Example Instruction	Address Generated (Linear Addressing)	Description
MOV Smem, dst	MOV port(*(AR7 + T1)), AR3	AR7 Assume AR7 = FF80h.	The CPU reads the value at I/O address FF80h and loads it into AR3. After being used for the address, AR7 is incremented by the number in T1.
MOV src, Smem	MOV AR4, port(*(AR5 + T1))	AR5 Assume AR5 = FFFFh.	The CPU reads the content of AR4 and writes it to I/O address FFFFh. After being used for the address, AR5 is incremented by the number in T1.

6.9.3.12 *(ARn – T1) Used for I/O-Space Access

Operand	Description
*(ARn – T1)	1) I/O address generated: [BSAyy +] ARn 2) ARn modified: ARn = ARn – T1

Example Syntax	Example Instruction	Address Generated (Linear Addressing)	Description
MOV Smem, dst	MOV port(*(AR7 – T1)), AR3	AR7 Assume AR7 = FF80h.	The CPU reads the value at I/O address FF80h and loads it into AR3. After being used for the address, AR7 is decremented by the number in T1.
MOV src, Smem	MOV AR4, port(*(AR5 – T1))	AR5 Assume AR5 = FFFFh.	The CPU reads the content of AR4 and writes it to I/O address FFFFh. After being used for the address, AR5 is decremented by the number in T1.

6.9.3.13 *ARn(T1) Used for I/O-Space Access

Operand	Description
*ARn(T1)	I/O address generated: [BSAyy +] ARn + T1 ARn is not modified. ARn is used as a base pointer. T1 is used as an offset from that base pointer.

Example Syntax	Example Instruction	Address Generated (Linear Addressing)	Description
MOV Smem, dst	MOV port(*AR7(T1)), AR3	(AR7 + T1) Assume AR7 = FF7Dh and T1 = 3.	The CPU reads the value at I/O address FF80h and loads it into AR3. AR7 is not modified.
MOV src, Smem	MOV AR4, port(*AR5(T1))	(AR5 + T1) Assume AR5 = FFFAh and T1 = 5.	The CPU reads the content of AR4 and writes it to I/O address FFFFh. AR5 is not modified.

6.9.3.14 *ARn(short(#k3)) Used for I/O-Space Access

Operand	Description
*ARn(short(#k3))	Address generated: [BSAyy +] ARn + k3 ARn is not modified. ARn is used as a base pointer. The 3-bit unsigned constant (k3) is used as an offset from that base pointer. k3 can be a number from 1 to 7.

Example Syntax	Example Instruction	Address Generated (Linear Addressing)	Description
MOV Smem, dst	MOV port(*AR7(short(#4))), AR3	(AR7 + 4) Assume AR7 = FF70h.	The CPU reads the value at I/O address FF74h and loads it into AR3. AR7 is not modified.
MOV src, Smem	MOV AR4, port(*AR5(short(#7)))	(AR5 + 7) Assume AR5 = FFF0h.	The CPU reads the content of AR4 and writes it to I/O address FFF7h. AR5 is not modified.

6.9.3.15 *CDP Used for I/O-Space Access

Operand	Description
*CDP	I/O address generated: [BSAC +] CDP CDP is not modified.

Example Syntax	Example Instruction	Address Generated (Linear Addressing)	Description
MOV Smem, dst	MOV port(*CDP), T2	CDP Assume CDP = FF80h.	The CPU reads the value at I/O address FF80h and loads it into T2. CDP is not modified.
MOV src, Smem	MOV T2, port(*CDP)	CDP Assume CDP = FFFFh.	The CPU reads the content of T2 and writes it to I/O address FFFFh. CDP is not modified.

6.9.3.16 *CDP+ Used for I/O-Space Access

Operand	Description
*CDP+	1) I/O address generated: [BSAC +] CDP 2) CDP modified: CDP = CDP + 1

Example Syntax	Example Instruction	Address Generated (Linear Addressing)	Description
MOV Smem, dst	MOV port(*CDP+), T2	CDP Assume CDP = FF80h.	The CPU reads the value at I/O address FF80h and loads it into T2. After being used for the address, CDP is incremented by 1.
MOV src, Smem	MOV T2, port(*CDP+)	CDP Assume CDP = FFFFh.	The CPU reads the content of T2 and writes it to I/O address FFFFh. After being used for the address, CDP is incremented by 1.

6.9.3.17 *CDP– Used for I/O-Space Access

Operand	Description
*CDP–	1) I/O address generated: [BSAC +] CDP 2) CDP modified: CDP = CDP – 1

Example Syntax	Example Instruction	Address Generated (Linear Addressing)	Description
MOV Smem, dst	MOV port(*CDP–), T2	CDP Assume CDP = FF80h.	The CPU reads the value at I/O address FF80h and loads it into T2. After being used for the address, CDP is decremented by 1.
MOV src, Smem	MOV T2, port(*CDP–)	CDP Assume CDP = FFFFh.	The CPU reads the content of T2 and writes it to I/O address FFFFh. After being used for the address, CDP is decremented by 1.

6.10 Restrictions on Accesses to I/O Space

The following indirect operands **cannot** be used for accesses to I/O space. An instruction using one of these operands requires a 2-byte extension for the constant. This extension would prevent the use of the port() qualifier needed to indicate an I/O-space access.

Table 6–24. Indirect Operands That do not Support Accesses to I/O Space

Operand That Does not Support I/O-Space Accesses	Pointer Modification
*ARn(#K16)	ARn is not modified. ARn is used as a base pointer. The 16-bit signed constant (K16) is used as an offset from that base pointer.
*+ARn(#K16)	The 16-bit signed constant (K16) is added to ARn before the address is generated.
*CDP(#K16)	CDP is not modified. CDP is used as a base pointer. The 16-bit signed constant (K16) is used as an offset from that base pointer.
*+CDP(#K16)	The 16-bit signed constant (K16) is added to CDP before the address is generated.

Also, the delay operation cannot be used for accesses to I/O space. Therefore, the syntaxes listed in Table 6–25 do not support I/O-space accesses.

Table 6–25. Instruction Syntaxes That do not Support Accesses to I/O Space

Syntax That Does not Support I/O-Space Accesses	Instruction Type
DELAY Smem	Memory Delay
MACM[R]Z [T3 =] Smem, Cmem, ACx	Multiply and Accumulate with delay

Any illegal access to I/O space generates a hardware bus-error interrupt (BERRINT) to be handled by the CPU.

6.11 Circular Addressing

Circular addressing can be used with any of the indirect addressing modes. Each of the eight auxiliary registers (AR0–AR7) and the coefficient data pointer (CDP) can be independently configured to be linearly or circularly modified as they act as pointers to data or to register bits. This configuration is done with a bit in status register ST2_55 (see Table 6–26). To choose circular modification, set the bit.

The size of a circular buffer is defined by one of three registers—BK03, BK47, or BKC (see Table 6–26). The buffer size register defines the number of words in a buffer of words, or it defines the number of bits in a buffer of bits within a register.

For a buffer of words in data space, the buffer must be placed in one of the 128 main data pages of data space. Each address within the buffer has 23 bits, and the 7 MSBs are the main data page. Set the main data page in CDPH or ARnH, where n is the number of the auxiliary register. CDPH can be loaded individually, but ARnH must be loaded via its extended auxiliary register. For example, to load AR0H, you must load XAR0, which is the concatenation AR0H:AR0. Within the main data page, the start of the buffer is defined by the value you load into the appropriate 16-bit buffer start address register (see Table 6–26). The value you load into the pointer (ARn or CDP) acts as an index, selecting words relative to the start address.

For a buffer of bits, the buffer start address register defines the reference bit, and the pointer selects bits relative to the position of that reference bit. You need only load ARn or CDP; you do not have to load XARn or XCDP.

Table 6–26. Pointers and the Associated Bits and Registers for Circular Addressing

Pointer	Linear/Circular Configuration Bit	Supplier of Main Data Page	Buffer Start Address Register	Buffer Size Register
AR0	ST2_55(0) = AR0LC	AR0H	BSA01	BK03
AR1	ST2_55(1) = AR1LC	AR1H	BSA01	BK03
AR2	ST2_55(2) = AR2LC	AR2H	BSA23	BK03
AR3	ST2_55(3) = AR3LC	AR3H	BSA23	BK03
AR4	ST2_55(4) = AR4LC	AR4H	BSA45	BK47
AR5	ST2_55(5) = AR5LC	AR5H	BSA45	BK47
AR6	ST2_55(6) = AR6LC	AR6H	BSA67	BK47
AR7	ST2_55(7) = AR7LC	AR7H	BSA67	BK47
CDP	ST2_55(8) = CDPLC	CDPH	BSAC	BKC

6.11.1 Configuring AR0–AR7 and CDP for Circular Addressing

Each auxiliary register ARn has its own linear/circular configuration bit in ST2_55:

ARnLC	ARn Is Used For ...
0	Linear addressing
1	Circular addressing

The CDPLC bit in status register ST2_55 configures the DSP to use CDP for linear addressing or circular addressing:

CDPLC	CDP Is Used For ...
0	Linear addressing
1	Circular addressing

You can use the circular addressing instruction qualifier if you want every pointer used by the instruction to be modified circularly. If you are using mnemonic instructions, just add .CR to the end of the instruction mnemonic (for example, ADD.CR). If you are using algebraic instructions, add the circular() qualifier in parallel with the instruction (instruction || circular()). The circular addressing instruction qualifier overrides the linear/circular configuration in ST2_55.

6.11.2 Circular Buffer Implementation

As an example of how to set up a circular buffer, consider this procedure for a circular buffer of words in data memory:

1) Initialize the appropriate buffer size register (BK03, BK47, or BKC). For example, for a buffer of size 8, load the BK register with 8.

2) Initialize the appropriate configuration bit in ST2_55 to choose circular modification for the selected pointer.

3) Initialize the appropriate extended register (XARn or XCDP) to select a main data page (in the 7 most significant bits). For example, if auxiliary register 3 (AR3) is the circular pointer, load extended auxiliary register 3 (XAR3). If CDP is the circular pointer, load XCDP.

4) Initialize the appropriate buffer start address register (BSA01, BSA23, BSA45, BSA67, or BSAC). The main data page, in XARn(22–16) or XCDP(22–16), concatenated with the content of the BSA register defines the 23-bit start address of the buffer.

5) Load the selected pointer, ARn or CDP, with a value from **0 to (buffer size – 1)**. For example, if you are using AR1 and the buffer size is 8, load AR1 with a value less than or equal to 7.

   A circular buffer of size R must start on an N-bit boundary, where N is the smallest integer that satisfies the relationship, $2^N > R$. For example, for a buffer size R = 8, N is 4. In this case, the top of the circular buffer is the address generated when the 4 LSBs of the pointer (ARn or CDP) are 0s. When the address incrementing leads beyond the buffer, the 4 LSBs of the pointer are forced to 0s.

If you are using indirect addressing operands with offsets, make sure that the absolute value of each offset is less than or equal to (buffer size – 1). Likewise, if the circular pointer is to be incremented or decremented by a programmed amount (supplied by a constant or by T0, AR0, or T1), make sure the absolute value of that amount is less than or equal to (buffer size – 1).

After the initialization, you have a 23-bit address of the following form:

ARnH:(BSAxx + ARn),
or
CDPH:(BSAC + CDP)

Increments and decrements are made to the 16-bit pointer (ARn or CDP) only. You cannot address data across main data pages without changing the value in the corresponding extension register (ARnH or CDPH).

Note:

Although an increment past FFFFh or a decrement past 0000h causes the pointer value to wrap around, do not make use of this behavior; it is not supported. Also, the BSAxx or BSAC addition must not increment the address beyond FFFFh.

Example 6–1 demonstrates initializing and then accessing a circular buffer.

Example 6–1. Initializing and Accessing a C55x Circular Buffer

```
MOV #3, BK03               ; Circular buffer size is 3 words
BSET AR1LC                 ; AR1 is configured to be modified circularly
AMOV #010000h, XAR1        ; Circular buffer is in main data page 01
MOV #0A02h, BSA01          ; Circular buffer start address is 010A02h
MOV #0000h, AR1            ; Index (in AR1) is 0000h

MOV *AR1+, AC0             ; AC0 loaded from 010A02h + (AR1) = 010A02h,
                           ; and then AR1 = 0001h
MOV *AR1+, AC0             ; AC0 loaded from 010A02h + (AR1) = 010A03h,
                           ; and then AR1 = 0002h
MOV *AR1+, AC0             ; AC0 loaded from 010A02h + (AR1) = 010A04h,
                           ; and then AR1 = 0000h
MOV *AR1+, AC0             ; AC0 loaded from 010A02h + (AR1) = 010A02h,
                           ; and then AR1 = 0001h
```

6.11.3 TMS320C54x Compatibility

In the TMS320C54x-compatible mode (when the C54CM bit is 1), the circular buffer size register BK03 is used with all the auxiliary registers, and BK47 is not used. The TMS320C55x device enables you to emulate TMS320C54x circular buffer management by following these programming rules:

- ❏ Initialize BK03 with the desired buffer size.
- ❏ Initialize the appropriate configuration bit in ST2_55 to set circular activity for the selected pointer.
- ❏ Initialize the appropriate extended auxiliary register (XARn) with the main data page in the seven most significant bits (ARnH).
- ❏ Initialize the pointer (ARn or CDP) to set the start address.
- ❏ Initialize the appropriate buffer start address register to 0, so that it has no effect.

If you are using indirect addressing operands with offsets, make sure that the absolute value of each offset is less than or equal to (buffer size – 1). Likewise, if the circular pointer is to be incremented or decremented by a programmed amount (supplied by a constant or by T0, AR0, or T1), make sure the absolute value of that amount is less than or equal to (buffer size – 1).

The code in Example 6–2 emulates a C54x circular buffer.

Example 6–2. Emulating a C54x Circular Buffer

```
MOV #3, BK03                ; Circular buffer size is 3 words
BSET AR1LC                  ; AR1 is configured to be modified circularly
AMOV #010000h, XAR1         ; Circular buffer is in main data page 01
MOV #0A01h, AR1             ; Circular buffer start address is 010A00h
MOV #0h, BSA01              ; BSA01 is 0, so that it has no effect

MOV *AR1+, AC0              ; AC0 loaded from 010A01h, and then AR1 = 0A02h
MOV *AR1+, AC0              ; AC0 loaded from 010A02h, and then AR1 = 0A00h
MOV *AR1+, AC0              ; AC0 loaded from 010A00h, and then AR1 = 0A01h
```

This circular buffer implementation has the disadvantage that it requires the alignment of the circular buffer on an 8-word address boundary. To remove this constraint, you can initialize BSA01 with an offset, as shown in Example 6–3.

Example 6–3. Avoiding the Alignment Constraint in Example 6–2

```
MOV #3, BK03                ; Circular buffer size is 3 words
BSET AR1LC                  ; AR0 is configured to be modified circularly
AMOV #010000h, XAR1         ; Circular buffer is in main data page 01
MOV #0A01h, AR1             ; Circular buffer start address is 010A00h
MOV #2h, BSA01              ; Add an offset of 2 to the buffer start address,
                            ; so that the effective start address is 010A02h

MOV *AR1+, AC0              ; AC0 loaded from 010A01h + 2h = 010A03h,
                            ; and then AR1 = 0A02h
MOV *AR1+, AC0              ; AC0 loaded from 010A02h + 2h = 010A04h,
                            ; and then AR1 = 0A00h
MOV *AR1+, AC0              ; AC0 loaded from 010A00h + 2h = 010A02h,
                            ; and then AR1 = 0A01h
```

Appendix A

Revision History

This document was revised to SPRU371F from SPRU371E, which was dated November 2003. Notable changes made since the last revision are listed in Table A–1.

Table A–1. Revision History

Page	Additions/Modifications/Deletions
Global	Notes were added in appropriate sections of this document to highlight the following important information: If a 16-bit address or pointer is incremented past FFFFh or decremented past 0000h, the address or pointer value wraps around. However, this behavior must not be used; it is not supported. Also, during circular addressing, the BSAxx or BSAC addition must not increment the address beyond FFFFh.
2-35	The following Note was added to section 2.9.2*, Block-Repeat Registers (BRC0, BRC1, BRS1, RSA0, RSA1, REA0, REA1):* Make sure the last three instructions of a level 0 loop do not write to BRC0. Likewise, make sure the last three instructions of a level 1 loop do not write to BRC1.
2-54	Section 2.10.3.4, *DBGM Bit of ST2_55*, was revised to indicate that writes to ST2_55 do not affect DBGM.
Index-1	The index was revised.

This page is intentionally left blank.

Index

A

B

C

D

K

L

M

N

O

P

R

S