TMS320C28x CPU and Instruction Set Reference Guide

User's Guide

Literature Number: SPRU430F
August 2001–Revised April 2015

Contents

List of Figures

List of Tables

Read This First

About This Manual

This manual describes the central processing unit (CPU) and the assembly language instructions of the TMS320C28x 32-bit fixed-point CPU. It also describes emulation features available on these devices. A summary of the chapters and appendices follows.

Chapter 1 —Architectural Overview

This chapter introduces the C2800 CPU that is at the heart of each TMS320C28x device. The chapter includes a memory map and a high-level description of the memory interface that connects the core with memory and peripheral devices.

Chapter 2 —Central Processing Unit

This chapter describes the architecture, registers, and primary functions of the CPU. The chapter includes detailed descriptions of the flag and control bits in the most important CPU registers, status registers ST0 and ST1.

Chapter 3 —Interrupts and Reset

This chapter describes the interrupts and how they are handled by the CPU. The chapter also explains the effects of a reset on the CPU and includes discussion of the automatic context save performed by the CPU prior to servicing an interrupt.

Chapter 4 —Pipeline

This chapter describes the phases and operation of the instruction pipeline. The chapter is primarily for readers interested in increasing the efficiency of their programs by preventing pipeline delays.

Chapter 5 —Addressing Modes

This chapter explains the modes by which the assembly language instructions accept data and access register and memory locations. The chapter includes a description of how addressing-mode information is encoded in op-codes.

Chapter 6 —Assembly Language Instructions

This chapter provides summaries of the instruction set and detailed descriptions (including examples) for the instructions. The chapter includes an ex- planation of how 32-bit accesses are aligned to even addresses.

Chapter 7 —Emulation Features

This chapter describes the TMS320C28x emulation features that can be used with only a JTAG port and two additional emulation pins.

Appendix A —Register Quick Reference

This appendix is a concise central resource for information about the status and control registers of the CPU. The chapter includes figures that summarize the bit fields of the registers.

Appendix B —C2xLP and C28x Architectureal Differences

This appendix describes the differences in the architecture of the C2xLP and the C28x.

Appendix C —Migration From C2xLP

This appendix explains how to migrate code from the C2xLP to the C28x.

Appendix D —C2xLP Instruction Set Compatibility

This appendix describes the instruction set compatibility with the C2xLP.

Appendix E —Migration From C27x to C28x

Migration From C27x to C28x

Appendix F —Glossary

This appendix explains abbreviations, acronyms, and special terminology used throughout this document.

Notational Conventions

This document uses the following conventions:

- The device number TMS320C28x is very often abbreviated as ’28x.
- Program examples are shown in a special typeface. Here is a sample line of program code:

```
PUSH IER
```

- Portions of an instruction syntax that are in **bold** should be entered as shown; portions of a syntax that are in _italics_ are variables indicating information that should be entered. Here is an example of an instruction syntax:
  MOV AR_x_, *__−SP__[_6bit_]
  MOV is the instruction mnemonic. This instruction has two operands, indicated by **AR**x and *__−SP__[_6bit_]. Where the variable x appears, you type a value from 0 to 5; where the 6bit appears, you type a 6-bit constant. The rest of the instruction, including the square brackets, must be entered as shown.
- When braces or brackets enclose an operand, as in {operand}, the operand is optional. If you use an optional operand, you specify the information within the braces; you do not enter the braces themselves. In the following syntax, the operand << _shift_ is optional:
  MOV ACC, *__−SP__[_6bit_] {<< _shift_}
  MOV ACC, *__−SP__[_6bit_] {<< _shift_}
  For example, you could use either of the following instructions:

```
MOV ACC, *−SP[5]
MOV ACC, *−SP[5]<< 4
```

- In most cases, hexadecimal numbers are shown with a subscript of 16. For example, the hexadecimal number 40 would be shown as 40_{16}. An exception to this rule is a hexadecimal number in a code example; these hexadecimal numbers have the suffix h. For example, the number 40 in the following code is a hexadecimal 40.

```
MOVB AR0,#40h
```

Similarly, binary numbers usually are shown with a subscript of 2. For example, the binary number 4 would be shown as 0100_2. Binary numbers in example code have the suffix b. For example, the following code uses a binary 4.

```
MOVB AR0,#0100b
```

- Bus signals and bits are sometimes represented with the following notations:

Notation	Description	Example
Bus(n:m)	Signals n through m of bus	PRDB(31:0) represents the 32 signals of the program-read data bus (PRDB).
Register(n:m)	Bits n through m of register	T(3:0) represents the 4 least significant bits of the T register.
Register(n)	Bit n of register	IER(4) represents bit 4 of the interrupt enable register (IER).

- Concatenated values are represented with the following notation:

Notation	Description	Example
x:y	x concatenated with y	AR1:AR0 is the concatenation of the 16-bit registers AR1 and AR0. AR0 is the low word. AR1 is the high word.

- If a signal is from an active-low pin, the name of the signal is qualified with an overbar (for example, $\overline{\text{INT1}}$). If a signal is from an active-high pin or from hardware inside the the device (in which case, the polarity is irrelevant), the name of the signal is left unqualified (for example, DLOGINT).

Related Documentaiton from Texas Instruments

The following books describe the TMS320C28x DSP and related support tools. The documents are available for downloading on the Texas Instruments website (www.ti.com).

TMS320F2801, TMS320F2806, TMS320F2808 Digital Signal Processors — (SPRS230) data sheet contains the pinout, signal descriptions, as well as electrical and timing specifications for the F280x devices.

TMS320C28x Assembly Language Tools User's Guide — (lSPRU513) describes the assembly language tools (assembler and other tools used to develop assembly language code), assembler directives, macros, common object file format, and symbolic debugging directives for the TMS320C28x™ device.

TMS320C28x Optimizing C Compiler User's Guide — (SPRU514) describes the TMS320C28x™ C/C++ compiler. This compiler accepts ANSI standard C/C++ source code and produces TMS320™ DSP assembly language source code for the TMS320C28x device.

TMS320F2810, TMS320F2811, TMS320F2812, TMS320C2810, TMS320C2811, and TMS320C2812 Digital Signal Processors — (SPRS174) data sheet contains the electrical and timing specifications for these devices, as well as signal descriptions and pinouts for all of the available packages.

TMS320x28xx, 28xxx DSP Peripherals Reference Guide — (SPRU566) describes all the peripherals available for TMS320x28xx and TMS320x28xxx devices.

TMS320C28x Floating Point Unit and Instruction Set Reference Guide — (SPRUEO2) describes the CPU architecture, pipeline, instruction set, and interrupts of the C28x floating−point DSP.

Architectural Overview

The TMS320C28xTM is one of several fixed-point CPUs in the TMS320 family. The C28x™ is source-code and object-code compatible with the C27x™. In addition, much of the code written for the C2xLP CPU can be reassembled to run on a C28x device.

The C2xLP CPU is used in all TMS320F24xx and TMS320C20x devices and their derivatives. This document refers to C2xLP as a generic name for the CPU used in these devices.

This chapter provides an overview of the architectural structure and components of the C28x CPU.

Topic **Page**

1.1 Introduction to the CPU

The CPU is a low-cost 32-bit fixed-point processor. This device draws from the best features of digital signal processing; reduced instruction set computing (RISC); and microcontroller architectures, firmware, and tool sets. The CPU features include a modified Harvard architecture and circular addressing. The RISC features are single-cycle instruction execution, register-to-register operations, and modified Harvard architecture (usable in Von Neumann mode). The microcontroller features include ease of use through an intuitive instruction set, byte packing and unpacking, and bit manipulation.

The modified Harvard architecture of the CPU enables instruction and data fetches to be performed in parallel. The CPU can read instructions and data while it writes data simultaneously to maintain the single-cycle instruction operation across the pipeline. The CPU does this over six separate address/data buses.

1.1.1 Compatibility With Other TMS320 CPUs

The C28x CPU features compatibility modes that minimize the migration effort from the C27x and C2xLP CPUs. The operating mode of the device is determined by a combination of the OBJMODE and AMODE bits in status register 1 (ST1) as shown in Table 1-1. The OBJMODE bit allows you to select between code compiled for a C28x (OBJMODE == 1) and code compiled for a C27x (OBJMODE == 0). The AMODE bit allows you to select between C28x/C27x instruction addressing modes (AMODE == 0) and C2xLP compatible instruction addressing modes (AMODE == 1).

Table 1-1. Compatibility Modes

	OBJMODE	AMODE
C28x Mode	1	0
C2xLP Source-compatible Mode	1	1
C27x Object-compatible Mode[1]	0	0

[1] The C28x is in C27x-compatible mode at reset.

- C28x Mode: In C28x mode, you can take advantage of all the C28x native features, addressing modes, and instructions. To operate in C28x mode from reset, your code must first set the OBJMODE bit by using the "C28OBJ" (or "M SETC OBJMODE"M) instruction. This book assumes you are operating in C28x mode unless stated otherwise.
- C2xLP Source-Compatible Mode: C2xLP source-compatible mode al- lows you to run C2xLP source code which has been reassembled using the C28x code-generation tools. For more information on operating in this mode and migration from a C2xLP CPU, see Appendix C, Appendix D, Appendix E.
- C27x Object-Compatible Mode: At reset, the C28x CPU operates in C27x object-compatible mode. In this mode, the C28x is 100% object-code and cycle-count compatible with the C27x CPU. For detailed information on operating in C27x object-compatible mode and migrating from the C27x, see Appendix F.

1.1.2 Switching to C28x Mode From Reset

At reset, the C28x CPU is in C27x Object-Compatible Mode (OBJMODE == 0, AMODE == 0) and is 100% compatible with the C27x CPU. To take advan- tage of the enhanced C28x instruction set, you must instead operate the de- vice in C28x mode. To do this, after a reset your code must first set the OBJMODE bit in ST1 by using the "M C28OBJ"M (or "M SETC OBJMODE"M) instruction.

1.2 Components of the CPU

As shown in Figure 1-1, the CPU contains:

- A CPU for generating data- and program-memory addresses; decoding and executing instructions; performing arithmetic, logical, and shift operations; and controlling data transfers among CPU registers, data memory, and program memory
- Emulation logic for monitoring and controlling various parts and functionalities of the DSP and for testing device operation
- Signals for interfacing with memory and peripherals, clocking and controlling the CPU and the emulation logic, showing the status of the CPU and the emulation logic, and using interrupts

The CPU does not contain memory, a clock generator, or peripheral devices. For information about interfacing to these items, see the *C28x Peripheral User's Guide* (SPRU566) and the data sheet that corresponds to your DSP.

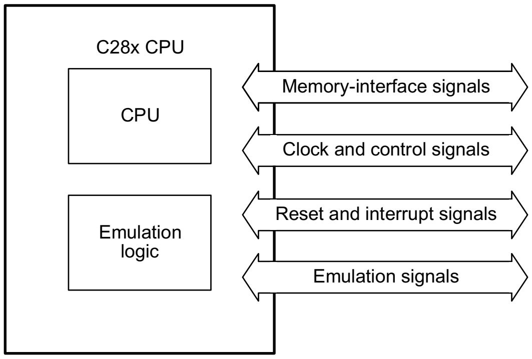


Figure 1-1. High-Level Conceptual Diagram of the CPU

1.2.1 *Central Processing Unit (CPU)*

The CPU is discussed in more detail in Chapter 2, but following is a list of its major features:

- Protected pipeline. The CPU implements an 8-phase pipeline that prevents a write to and a read from the same location from occurring out of order.
- Independent register space. The CPU contains registers that are not mapped to data space. These registers function as system-control registers, math registers, and data pointers. The system-control registers are accessed by special instructions. The other registers are accessed by special instructions or by a special addressing mode (register addressing mode).
- Arithmetic logic unit (ALU). The 32-bit ALU performs 2s-complement arithmetic and Boolean logic operations.
- Address register arithmetic unit (ARAU). The ARAU generates data- memory addresses and increments or decrements pointers in parallel with ALU operations.
- Barrel shifter. This shifter performs all left and right shifts of data. It can shift data to the left by up to 16 bits and to the right by up to 16 bits.
- Multiplier. The multiplier performs 32-bit × 32-bit 2s-complement multiplication with a 64-bit result. The multiplication can be performed with two signed numbers, two unsigned numbers, or one signed number and one unsigned number.

1.2.2 Emulation Logic

The emulation logic includes the following features. For more details about these features, see Chapter 7, *Emulation Features*.

- Debug-and-test direct memory access (DT-DMA). A debug host can gain direct access to the content of registers and memory by taking control of the memory interface during unused cycles of the instruction pipeline.
- Data logging. The emulation logic enables application-initiated transfers of memory contents between the C28x and a debug host.
- A counter for performance benchmarking
- Multiple debug events. Any of the following *debug events* can cause a break in program execution:
  - A breakpoint initiated by the ESTOP0 or ESTOP1 instruction
  - An access to a specified program-space or data-space location
  - A request from the debug host or other hardware

  When a debug event causes the C28x to enter the debug-halt state, the event is called a *break event*.
- Real-time mode of operation. When the C28x is in this mode and a break event occurs, the main body of program code comes to a halt, but time-critical interrupts can still be serviced.

1.2.3 Signals

The CPU has four main types of signals:

- Memory-interface signals. These signals transfer data among the CPU, memory, and peripherals; indicate program-memory accesses and data-memory accesses; and differentiate between accesses of different sizes (16-bit or 32-bit).
- Clock and control signals. These provide clocking for the CPU and the emulation logic, and they are used to control and monitor the CPU.
- Reset and interrupt signals. These are used for generating a hardware reset and interrupts, and for monitoring the status of interrupts.
- Emulation signals. These signals are used for testing and debugging.

1.3 Memory Map

The C28x uses 32-bit data addresses and 22-bit program addresses. This allows for a total address reach of 4G words (1 word = 16 bits) in data space and 4M words in program space. Memory blocks on all C28x designs are uniformly mapped to both program and data space. Figure 1-2 shows a high-level view of how addresses are allocated in program space and data space.

The memory map in Figure 1-2 has been divided into the following segments:

- On-chip program/data
- Reserved
- CPU interrupt vectors

For specific details about each of the map segments, see the data sheet for your device. See Appendix C for more information on the C2xLP compatible memory space.

1.3.1 CPU Interrupt Vectors

Sixty-four addresses in program space are set aside for a table of 32 CPU interrupt vectors. The CPU vectors can be mapped to the top or bottom of program space by way of the VMAP bit. For more information about the CPU vectors, see Section 3.2.

For devices with a peripheral interrupt expansion (PIE) block, the interrupt vectors will reside in the PIE vector table and this memory can be used as program memory.

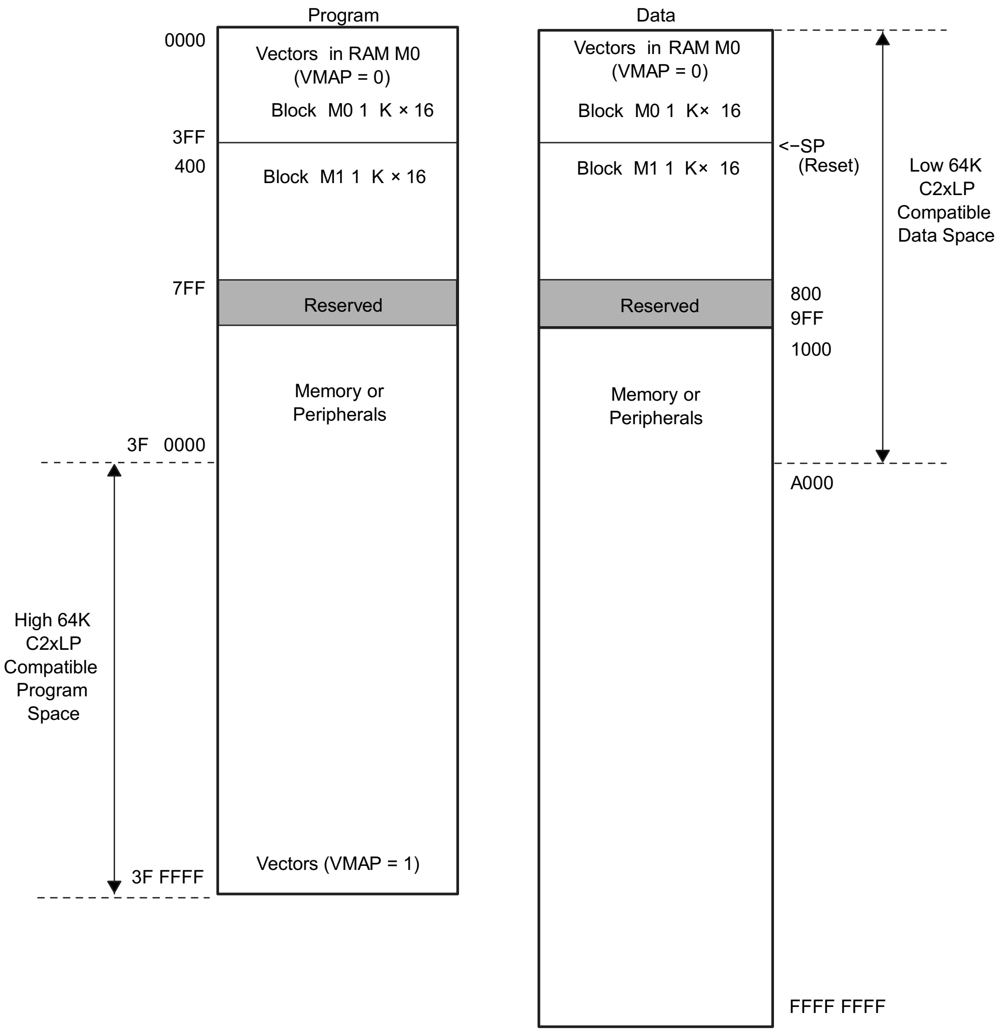


Figure 1-2. TMS320C28x High-Level Memory Map

See the data sheet for your specific device for details of the exact memory map.

1.4 Memory Interface

The C28x memory map is accessible outside the CPU by the memory interface, which connects the CPU logic to memories, peripherals, or other interfaces. The memory interface includes separate buses for program space and data space. This means an instruction can be fetched from program memory while data memory is being accessed.

The interface also includes signals that indicate the type of read or write being requested by the CPU. These signals can select a specified memory block or peripheral for a given bus transaction. In addition to 16-bit and 32-bit accesses, the C28x supports special byte-access instructions which can access the least significant byte (LSByte) or most significant byte (MSByte) of an addressed word. Strobe signals indicate when such an access is occurring on a data bus.

1.4.1 Address and Data Buses

The memory interface has three address buses:

PAB — *Program address bus.* The PAB carries addresses for reads and writes from program space. PAB is a 22-bit bus.

DRAB — *Data-read address bus.* The 32-bit DRAB carries addresses for reads from data space.

DWAB — *Data-write address bus.* The 32-bit DWAB carries addresses for writes to data space.

The memory interface also has three data buses:

PRDB — *Program-read data bus.*

The PRDB carries instructions or data during reads from program space. PRDB is a 32-bit bus.

DRDB — *Data-read data bus.* The DRDB carries data during reads from data space. PRDB is a 32-bit bus.

DWDB — *Data-/Program-write data bus.* The 32-bit DWDB carries data during writes to data space or program space.

Table 1-2 summarizes how these buses are used during accesses.

Table 1-2. Summary of Bus Use During Data-Space and Program-Space Accesses

Access Type	Address Bus	Data Bus
Read from program space	PAB	PRDB
Read from data space	DRAB	DRDB
Write to program space	PAB	DWDB
Write to data space	DWAB	DWDB

A program-space read and a program-space write cannot happen simultaneously because both use the PAB. Similarly, a program-space write and a data-space write cannot happen simultaneously because both use the DWDB. Transactions that use different buses can happen simultaneously. For example, the CPU can read from program space (using PAB and PRDB), read from data space (using DRAB and DRDB), and write to data space (using DWAB and DWDB) at the same time.

1.4.2 Special Bus Operations

Typically, PAB and PRDB are used only for reading instructions from program space, and DWDB is used only for writing data to data space. However, the instructions in Table 1-3 are exceptions to this behavior. For more details about using these instructions, see Chapter 6, *Assembly Language Instructions*.

Table 1-3. Special Bus Operations

Instruction	Special Bus Operation
PREAD	This instruction reads a data value rather than an instruction from program space. It then transfers that value to data space or a register. For the read from program space, the CPU places the source address on the program address bus (PAB), sets the appropriate program space select signals, and reads the data value from the program-read data bus (PRDB).
PWRITE	This instruction writes a data value to program space. The value is read from from data space or a register. For the write to program space, the CPU places the destination address on the program address bus (PAB), sets the appropriate program-space select signals, and writes the data value to the data-/program-write data bus (DWDB).
MAC DMAC QMACL IMACL XMAC XMACD	As part of their operation, these instructions multiply two data values, one of which is read from program space. For the read from program space, the CPU places the program-space source address on the program address bus (PAB), sets the appropriate program-space select signals, and reads the program data value from the program read data bus (PRDB).

1.4.3 Alignment of 32-Bit Accesses to Even Addresses

The C28x CPU expects memory wrappers or peripheral-interface logic to align any 32-bit read or write to an even address. If the address-generation logic generates an odd address, the CPU must begin reading or writing at the previous even address. This alignment does not affect the address values generated by the address-generation logic.

Most instruction fetches from program space are performed as 32-bit read operations and are aligned accordingly. However, alignment of instruction fetches are effectively invisible to a programmer. When instructions are stored to program space, they do not have to be aligned to even addresses. Instruction boundaries are decoded within the CPU.

You need to be concerned with alignment when using instructions that perform 32-bit reads from or writes to data space.

Chapter 2

SPRU430F–August 2001–Revised April 2015

Central Processing Unit

The central processing unit (CPU) is responsible for controlling the flow of a program and the processing of instructions. It performs arithmetic, Boolean-logic, multiply, and shift operations. When performing signed math, the CPU uses 2s-complement notation. This chapter describes the architecture, registers, and primary functions of the CPU.

Topic **Page**

2.1 CPU Architecture

All C28x devices contain a central processing unit (CPU), emulation logic, and signals for interfacing with memory and peripherals. Included with these signals are three address buses and three data buses. Figure 2-1 shows the major blocks and data paths of the C28x CPU. It does not reflect the actual silicon implementation. The shaded buses are memory-interface buses that are external to the CPU. The operand bus supplies the values for multiplier, shifter, and ALU operations, and the result bus carries the results to registers and memory. The main blocks of the CPU are:

- **Program and data control logic.** This logic stores a queue of instructions that have been fetched from program memory.
- **Real-Time emulation and visibility**
- **Address register arithmetic unit (ARAU).** The ARAU generates addresses for values that must be fetched from data memory. For a data read, it places the address on the data-read address bus (DRAB); for a data write, it loads the data-write address bus (DWAB). The ARAU also increments or decrements the stack pointer (SP) and the auxiliary registers (XAR0, XAR1, XAR2, XAR3, XAR4, XAR5, XAR6, and XAR7).
- **Atomic arithmetic logic unit (ALU).** The 32-bit ALU performs 2s-complement arithmetic and Boolean logic operations. Before doing its calculations, the ALU accepts data from registers, from data memory, or from the program control logic. The ALU saves results to a register or to data memory.
- **Prefetch queue and instruction decode**
- **Address generators for program and data**
- **Fixed-point MPY/ALU**. The multiplier performs 32-bit × 32-bit 2s-complement multiplication with a 64-bit result. In conjunction with the multiplier, the '28xx uses the 32-bit multiplicand register (XT), the 32-bit product register (P), and the 32-bit accumulator (ACC). The XT register supplies one of the values to be multiplied. The result of the multiplication can be sent to the P register or to ACC.
- **Interrupt processing**

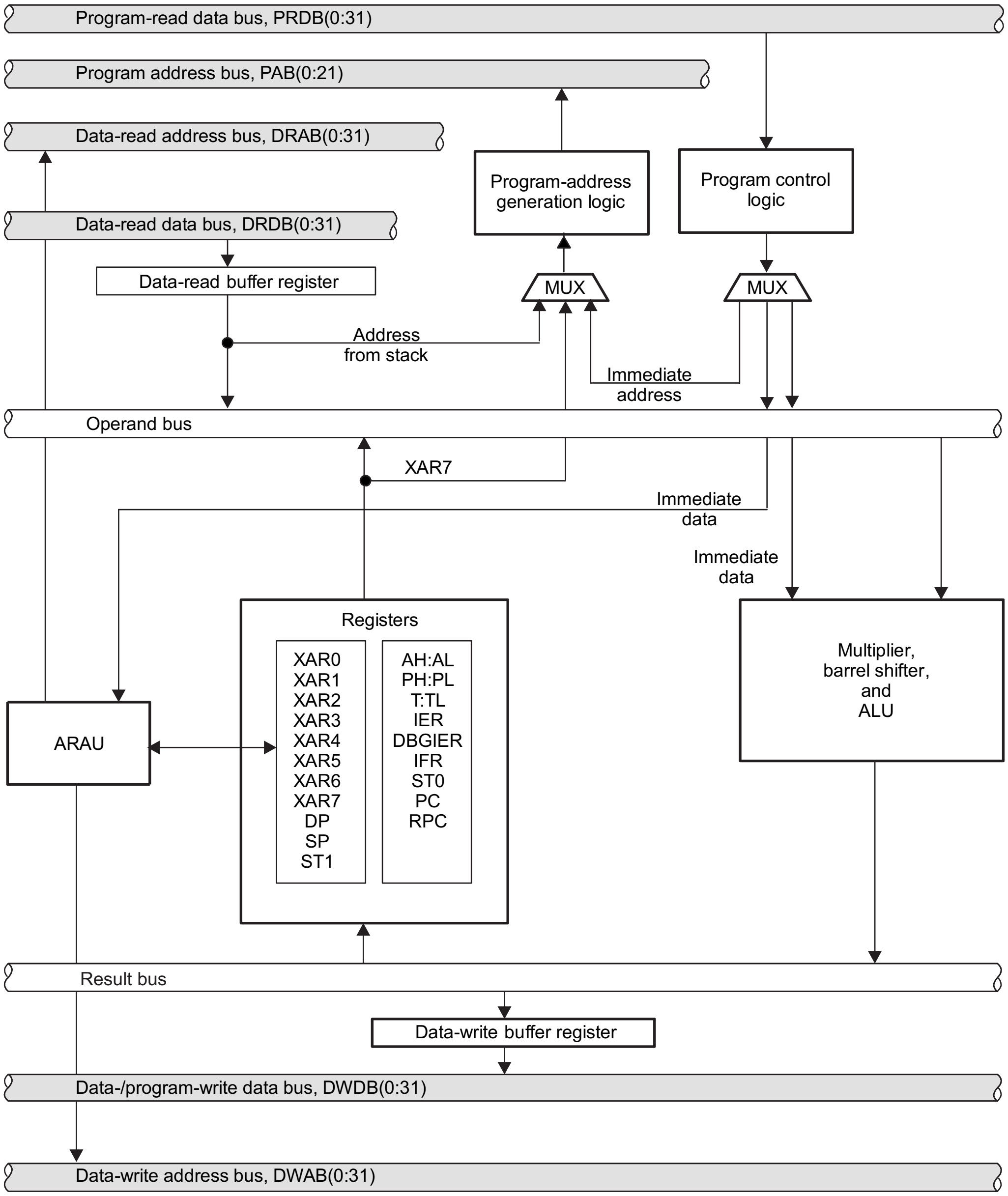

Program-read data bus, PRDB(0:31)
Program address bus, PAB(0:21)
Data-read address bus, DRAB(0:31)
Data-read data bus, DRDB(0:31)
Program-address generation logic
Program control logic
Data-read buffer register
MUX
MUX
Address from stack
Immediate address
Operand bus
XAR7
Immediate data
Immediate data
Registers
XAR0
XAR1
XAR2
XAR3
XAR4
XAR5
XAR6
XAR7
DP
SP
ST1
AH:AL
PH:PL
T:TL
IER
DBGIER
IFR
ST0
PC
RPC
ARAU
Multiplier, barrel shifter, and ALU
Result bus
Data-write buffer register
Data-/program-write data bus, DWDB(0:31)
Data-write address bus, DWAB(0:31)

Figure 2-1. Conceptual Block Diagram of the CPU

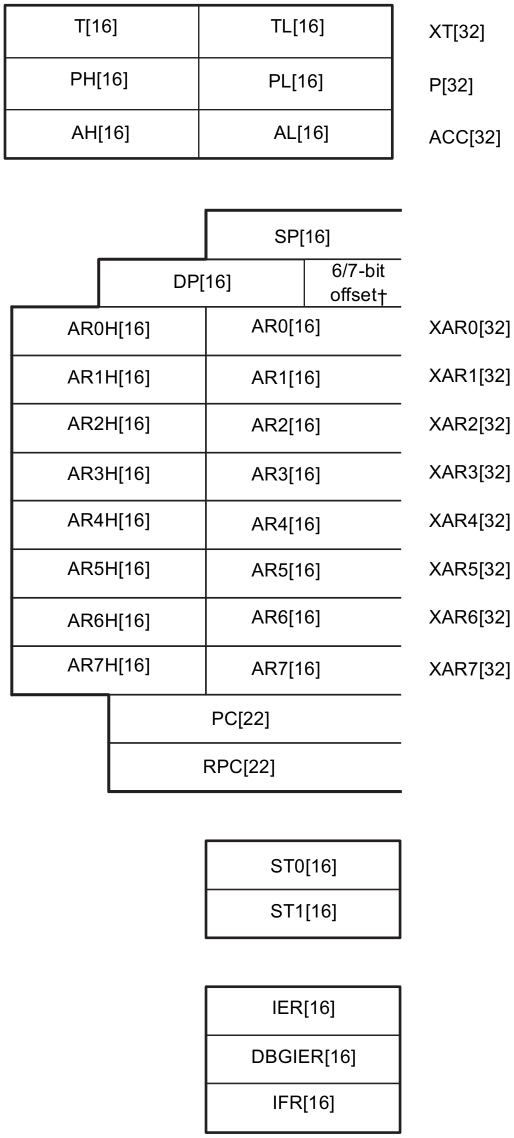
2.2 CPU Registers

Table 2-1 lists the main CPU registers and their values after reset. Section 2.2.1through Section 2.2.10 describe the registers in more detail. Figure 2-2 shows the registers.

Table 2-1. CPU Register Summary

Register	Size	Description	Value After Reset
ACC	32 bits	Accumulator	0x00000000
AH	16 bits	High half of ACC	0x0000
AL	16 bits	Low half of ACC	0x0000
XAR0	16 bits	Auxiliary register 0	0x00000000
XAR1	32 bits	Auxiliary register 1	0x00000000
XAR2	32 bits	Auxiliary register 2	0x00000000
XAR3	32 bits	Auxiliary register 3	0x00000000
XAR4	32 bits	Auxiliary register 4	0x00000000
XAR5	32 bits	Auxiliary register 5	0x00000000
XAR6	32 bits	Auxiliary register 6	0x00000000
XAR7	32 bits	Auxiliary register 7	0x00000000
AR0	16 bits	Low half of XAR0	0x0000
AR1	16 bits	Low half of XAR1	0x0000
AR2	16 bits	Low half of XAR2	0x0000
AR3	16 bits	Low half of XAR3	0x0000
AR4	16 bits	Low half of XAR4	0x0000
AR5	16 bits	Low half of XAR5	0x0000
AR6	16 bits	Low half of XAR6	0x0000
AR7	16 bits	Low half of XAR7	0x0000
DP	16 bits	Data-page pointer	0x0000
IFR	16 bits	Interrupt flag register	0x0000
IER	16 bits	Interrupt enable register	0x0000 (INT1 to INT14, DLOGINT, RTOSINT disabled)
DBGIER	16 bits	Debug interrupt enable register	0x0000 (INT1 to INT14, DLOGINT, RTOSINT disabled)
P	32 bits	Product register	0x00000000
PH	16 bits	High half of P	0x0000
PL	16 bits	Low half of P	0x0000
PC	22 bits	Program counter	0x3F FFC0
RPC	22 bits	Return program counte	0x00000000
SP	16 bits	Stack pointer	0x0400
ST0	16 bits	Status register 0	0x0000
ST1	16 bits	Status register 1	0x080B[1]
XT	32 bits	Multiplicand register	0x00000000
T	16 bits	High half of XT	0x0000
TL	16 bits	Low half of XT	0x0000

[1] Reset value shown is for devices without the VMAP signal and MOM1MAP signal pinned out. On these devices both of these signals are tied high internal to the device.



(1) A 6-bit offset is used when operating in C28x mode or C27x object-compatible mode.

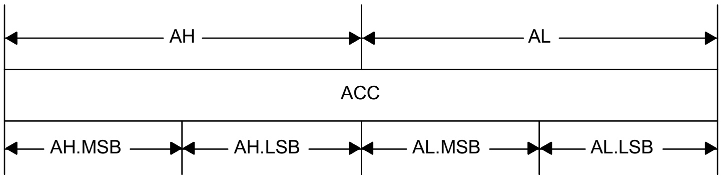
(2) A 7-bit offset is used when operating in C2xLP source-compatible mode. The least significant bit of the DP is ignored when operating in this mode.

Figure 2-2. C28x Registers

2.2.1 Accumulator (ACC, AH, AL)

The accumulator (ACC) is the main working register for the device. It is the destination for all ALU operations except those which operate directly on memory or registers. ACC supports single-cycle move, add, subtract, and compare operations from 32-bit-wide data memory. It can also accept the 32-bit result of a multiplication operation.

The halves and quarters of the ACC can also be accessed (see Figure 2-3). ACC can be treated as two independent 16-bit registers: AH (high 16 bits) and AL (low 16 bits). The bytes within AH and AL can also be accessed independently. Special byte-move instructions load and store the most significant byte or least significant byte of AH or AL. This enables efficient byte packing and unpacking.



AH = ACC (31:16)
AH.MSB = ACC (31:24)
AH.LSB = ACC (23:16)

AL = ACC (15:0)
AL.MSB = ACC (15:8)
AL.LSB = ACC (7:0)

Figure 2-3. Individually Accessible Portions of the Accumulator

The accumulator has the following associated status bits. For the details on these bits, see Section 2.3.

- Overflow mode bit (OVM)
- Sign-extension mode bit (SXM)
- Test/control flag bit (TC)
- Carry bit (C)
- Zero flag bit (Z)
- Negative flag bit (N)
- Latched overflow flag bit (V)
- Overflow counter bits (OVC)

Table 2-2 shows the ways to shift the content of AH, AL, or ACC.

Table 2-2. Available Operations for Shifting Values in the Accumulator

Register	Shift Direction	Shift Type	Instruction
ACC	Left	Logical	LSL or LSLL
		Rotation	ROL
	Right	Arithmetic	SFR with SXM = 1 or ASRL
		Logical	SFR with SXM = 0 or LSRL
		Rotation	ROR
AH or AL	Left	Logical	LSL
	Right	Arithmetic	ASR
		Logical	LSR

2.2.2 Multiplicand Register (XT)

The multiplicand register (XT register) is used primarily to store a 32-bit signed integer value prior to a 32-bit multiply operation.

The lower 16-bit portion of the XT register is referred to as the TL register. This register can be loaded with a signed 16-bit value that is automatically sign-extended to fill the 32-bit XT register.

The upper 16-bit portion of the XT register is referred to as the T register. The T register is mainly used to store a 16-bit integer value prior to a 16-bit multiply operation.

The T register is also used to specify the shift value for some shift operations. In this case, only a portion of the T register is used, depending on the instruction.

For example:

```
ASR     AX, T     performs an arithmetic shift right based on the four least significant bits
                        of T: T(3:0) = 0...15
ASRL    ACC, T    performs an arithmetic shift right by the five least significant bits of T:
                        T(4:0) 0...31
```

For these operations, the most significant bits of T are ignored.

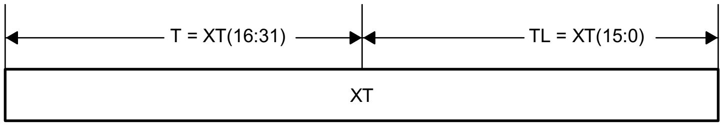


Figure 2-4. Individually Accessible Halves of the XT Register

2.2.3 Product Register (P, PH, PL)

The product register (P register) is typically used to hold the 32-bit result of a multiplication. It can also be loaded directly from a 16- or 32-bit data-memory location, a 16-bit constant, the 32-bit ACC, or a 16-bit or a 32-bit addressable CPU register. The P register can be treated as a 32-bit register or as two independent 16-bit registers: PH (high 16 bits) and PL (low 16 bits); see Figure 2-5.

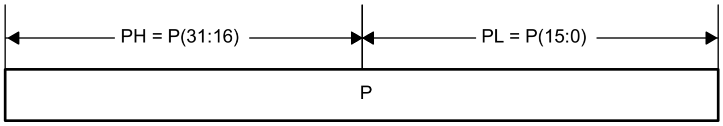


Figure 2-5. Individually Accessible Halves of the P Register

When some instructions access P, PH, or PL, all 32-bits are copied to the ALU- shifter block, where the barrel shifter may perform a left shift, a right shift, or no shift. The action of the shifter for these instructions is determined by the product shift mode (PM) bits in status register ST0. Table 2-3 shows the possible PM values and the corresponding product shift modes. When the barrel shifter performs a left shift, the low order bits are filled with zeros. When the shifter performs a right shift, the P register value is sign extended. Instructions that use PH or PL as operands ignore the product shift mode.

For a complete list of instructions affected by PM bits, see Table 2-6.

Table 2-3. Product Shift Modes

PM Value	Product Shift Mode
000_2	Left shift by 1
001_2	No shift 7
010_2	Right shift by 1
011_2	Right shift by 2
100_2	Right shift by 3
101_2	Right shift by 4 (if AMODE = 1, left 4)
110_2	Right shift by 5
111_2	Right shift by 6

2.2.4 Data Page Pointer (DP)

In the direct addressing modes, data memory is addressed in blocks of 64 words called *data pages*. The lower 4M words of data memory consists of 65 536 data pages labeled 0 through 65 535, as shown in Figure 2-6. In DP direct addressing mode, the 16-bit data page pointer (DP) holds the current data page number. You change the data page by loading the DP with a new number. For information about the direct addressing modes, see Section 5.4.

Data page	Offset	Data memory
00 0000 0000 0000 00 ⋮ 00 0000 0000 0000 00	00 0000 ⋮ 11 1111	Page 0: 0000 0000−0000 003F
00 0000 0000 0000 01 ⋮ 00 0000 0000 0000 01	00 0000 ⋮ 11 1111	Page 1: 0000 0040−0000 007F
00 0000 0000 0000 10 ⋮ 00 0000 0000 0000 10	00 0000 ⋮ 11 1111	Page 2: 0000 0080−0000 00BF
⋮	⋮	⋮ ⋮
11 111 1 1111 111 1 11 ⋮ 11 111 1 111 1 111 1 11	00 0000 ⋮ 11 1111	Page 65 535: 003F FFC0−003F FFFF

Figure 2-6. Pages of Data Memory

Data memory above 4M words is not accessible using the DP.

When operating in C2xLP source-compatible mode, a 7-bit offset is used and the least significant bit of the DP register is ignored. See Appendix C for more details.

2.2.5 Stack Pointer (SP)

The stack pointer (SP) enables the use of a software stack in data memory. The stack pointer has only 16 bits and can only address the low 64K of data space (see Figure 2-7). When the SP is used, the upper six bits of the 32-bit address are forced to 0. (For information about addressing modes that use the SP, see Section 5.5). After reset, SP points to address 00000400_{16}.

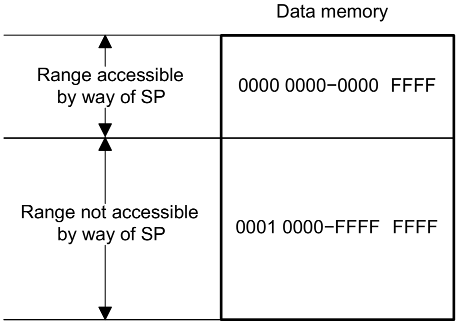

Data memory
Range accessible by way of SP
0000 0000−0000 FFFF
Range not accessible by way of SP
0001 0000−FFFF FFFF

Figure 2-7. Address Reach of the Stack Pointer

The operation of the stack is as follows:

- The stack grows from low memory to high memory.
- The SP always points to the next empty location in the stack.
- At reset, the SP is initialized, so that it points to address $0000\ 0400_{16}$.
- When 32-bit values are saved to the stack, the least significant 16 bits are saved first, and the most significant 16 bits are saved to the next higher address (little endian format).
- When 32-bit operations read or write a 32-bit value, the C28x CPU expects the memory wrapper or peripheral-interface logic to align that read or write to an even address. For example, if the SP contains the odd address $0000\ 0083_{16}$, a 32-bit read operation reads from addresses $0000\ 0082_{16}$ and 0000
- The SP overflows if its value is increased beyond FFFF16 or decreased below 0000_{16}. When the SP increases past $FFFF_{16}$, it counts forward from 0000_{16}. For example, if SP = $FFFE_{16}$ and an instruction adds 3 to the SP, the result is 0001_{16}. When the SP decreases past 000016, it counts backward from $FFFF_{16}$. For example, if SP = 0002_{16} and an instruction subtracts 4 from SP, the result is $FFFE_{16}$.
- When values are being saved to the stack, the SP is not forced to align with even or odd addresses. Alignment is forced by the memory wrapper or peripheral-interface logic.

2.2.6 Auxiliary Registers (XAR0-XAR7, AR0-AR7)

The CPU provides eight 32-bit registers that can be used as pointers to memory or as general-purpose registers (see Section 5.6). The auxiliary registers are: XAR0, XAR1, XAR2, XAR3, XAR4, XAR5, XAR6, and XAR7.

Many instructions allow you to access the 16 LSBs of XAR0-XAR7. As shown in Figure 2-8, the 16 LSBs of the auxiliary registers are referred to as AR0-AR7. AR0-AR7 can be used as general purpose registers for loop control and for efficient 16-bit comparisons.

When accessing AR0-AR7, the upper 16 bits of the register (known as AR0H-AR7H) may or may not be modified, depending on the instruction used (see Chapter 6 for information on the behavior of particular instructions). AR0H-AR7H are accessed only as part of XAR0-XAR7 and are not individually accessible.

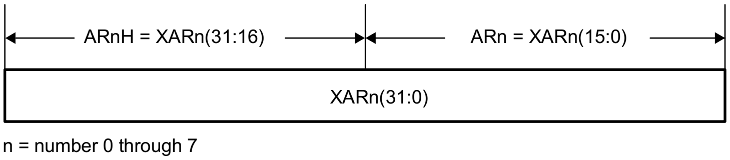


Figure 2-8. XAR0 - XAR7 Registers

For ACC operations, all 32 bits are valid ([@XARn]). For 16-bit operations, the lower 16 bits are used and upper 16 bits are ignored ([@ARn]).

XAR0 - XAR7 can also be used by some instructions to point to any value in program memory; see Section 5.6.

Many instructions allow you to access the 16 least significant bits (LSBs) of XAR0-XAR7. As shown in Figure 2-9, 16 LSBs of XAR0-XAR7 are known as one auxiliary register of AR0-AR7.

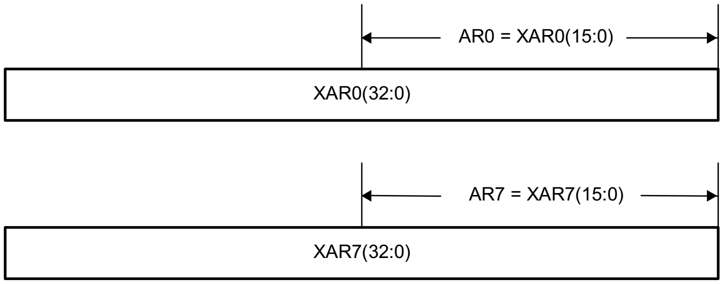


Figure 2-9. XAR0 - XAR7

2.2.7 Program Counter (PC)

When the pipeline is full, the 22-bit program counter (PC) always points to the instruction that is currently being processed â€” the instruction that has just reached the decode 2 phase of the pipeline. Once an instruction reaches this phase of the pipeline, it cannot be flushed from the pipeline by an interrupt. It is executed before the interrupt is taken. The pipeline is discussed in Chapter 4.

2.2.8 Return Program Counter (RPC)

When a call operation is performed using the LCR instruction, the return address is saved in the RPC register and the old value in the RPC is saved on the stack (in two 16-bit operations). When a return operation is performed using the LRETR instruction, the return address is read from the RPC register and the value on the stack is written into the RPC register (in two 16-bit operations). Other call instructions do not use the RPC register. For more information, see the instructions in Chapter 6.

2.2.9 Status Registers (ST0, ST1)

The C28x has two status registers, ST0 and ST1, which contain various flag bits and control bits. These registers can be stored into and loaded from data memory, enabling the status of the machine to be saved and restored for subroutines.

The status bits have been organized according to when the bit values are modified in the pipeline. Bits in ST0 are modified in the execute phase of the pipeline; bits in ST1 are modified in the decode 2 phase. (For details about the pipeline, see Chapter 4.) The status bits are described in detail in Section 2.3 and Section 2.4. Also, ST0 and ST1 are included in Appendix A.

2.2.10 Interrupt-Control Registers (IFR, IER, DBGIER)

The C28x CPU has three registers dedicated to the control of interrupts:

- Interrupt flag register (IFR)
- Interrupt enable register (IER)
- Debug interrupt enable register (DBGIER)

These registers handle interrupts at the CPU level. Devices with a peripheral interrupt expansion (PIE) block will have additional interrupt control as part of the PIE module.

The IFR contains flag bits for maskable interrupts (those that can be enabled and disabled with software). When one of these flags is set, by hardware or software, the corresponding interrupt will be serviced if it is enabled. You enable or disable a maskable interrupt with its corresponding bit in the IER. The DBGIER indicates the time-critical interrupts that will be serviced (if enabled) while the DSP is in real-time emulation mode and the CPU is halted.

The C28x CPU interrupts and the interrupt-control registers are described in detail in Section 3.1. Also, the IFR, IER, and DBGIER are included in Appendix A.

2.3 Status Register ST0

The following figure shows the bit fields of status register (ST0). All of these bit fields are modified in the execute phase of the pipeline. Detailed descriptions of these bits follow the figure.

Table 2-4. Bit Fields of Status Register (ST0)

15 10	9 7	6	5	4	3	2	1	0
OVC/OVCU	PM	V	N	Z	C	TC	OVM	SXM
R/W-00 0000	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0

LEGEND: R/W = Read/Write; R = Read only; -*n* = value after reset

OVC/OVCU (Bits 15-10) — Overflow counter.

The overflow counter behaves differently for signed and unsigned operations.

For signed operations, the overflow counter is a 6-bit signed counter with a range of -32 to 31. When overflow mode is off (OVM = 0), ACC overflows normally, and OVC keeps track of overflows. When overflow mode is on (OVM = 1) and an overflow occurs in ACC, the OVC is not affected. Instead, the CPU automatically fills ACC with a positive or negative saturation value (see the description for OVM).

When ACC overflows in the positive direction (from 7FFF $FFFF^{16}$ to 8000 0000_{16}), the OVC is incremented by 1. When ACC overflows in the negative direction (from 8000 0000_{16} to 7FFF $FFFF_{16}$) the OVC is decremented by 1. The increment or decrement is performed as the overflow affects the V flag.

For unsigned operations (OVCU), the counter increments for ADD when a Carry is generated and decrements for a SUB when a Borrow is generated (similar to a carry counter).

If OVC increments past its most positive value, 31, the counter wraps around to -32. If OVC decrements past its most negative value, -32, the counter wraps around to 31. At reset, OVC is cleared.

OVC is not affected by overflows in registers other than ACC and is not affected by compare instructions (CMP and CMPL). The table that follows explains how OVC may be affected by the saturate accumulator (SAT ACC) instruction.

Table 2-5 lists the instructions affecting OVC/OVCU. See the instruction set in Chapter 6 for a complete description of each instruction.

Table 2-5. Instructions That Affect OVC/OVCU

Signed Addition Instructions	Effect on OVC/OVCU
ADD ACC,loc16 << shift	if(OVM == 0) Inc OVC on +ve signed overflow
ADD ACC,#16bit << shift	
ADD ACC,loc16 << T	
ADD loc16,#16bitSigned	
ADDB ACC,#8bit	
ADDCL ACC,loc32	
ADDCU ACC,loc16	
ADDL ACC,loc32	
ADDL loc32,ACC	
ADDU ACC,loc16	

Table 2-5. Instructions That Affect OVC/OVCU (continued)

Signed Addition Instructions	Effect on OVC/OVCU
DMAC ACC:P,loc32,*XAR7/++	
INC loc16	
MAC P,loc16,*XAR7/++	
MAC P,loc16,0:pma	
MOVA T,loc16	
MOVAD T,loc16	
MPYA P,loc16,#16bit	
MPYA P,T,loc16	
QMACL P,loc32,*XAR7/++	
QMPYAL P,XT,loc32	
SQRA loc16	
XMAC P,loc16,*(pma)	
XMACD P,loc16,*(pma)	
Signed Subtraction Instructions	**Effect on OVC/OVCU**
DEC loc16 MOVS T,loc16	if(OVM == 0) Dec OVC on -ve signed overflow
Signed Addition Instructions	**Effect on OVC/OVCU**
MPYS P,T,loc16	
QMPYSL P,XT,loc32	
SBBU ACC,loc16	
SQRS loc16	
SUB ACC,#16bit << shift	
SUB ACC,loc16 << shift	
SUB ACC,loc16 << T	
SUBB ACC,#8bit	
SUBBL ACC,loc32	
SUBL ACC,loc32	
SUBL loc32,ACC	
SUBRL loc32,ACC	
SUBU ACC,loc16	
SUBUL ACC,loc32	
SUBUL P,loc32	
Unsigned Instructions	**Effect on OVC/OVCU**
ADDUL ACC,loc32	Inc OVC/OVCU on unsigned carry
ADDUL P,loc32	
IMPYAL P,XT,loc32	
IMACL P,loc32,*XAR7/++	
Misc Instructions	**Effect on OVC/OVCU**
SAT ACC	if(OVC > 0) Saturate +ve if(OVC < 0) Saturate -ve OVC = 0
SAT64 ACC:P	
ZAPA	OVC = 0
ZAP OVC	
MOV OVC,loc16	OVC = [loc16(15:10)]
Signed Addition Instructions	**Effect on OVC/OVCU**
MOVU OVC,loc16	OVC = [loc16(5:0)]
Condition	**Operation Performed by SAT ACC Instruction**
OVC = 0	Leave ACC and OVC unchanged.

Table 2-5. Instructions That Affect OVC/OVCU (continued)

Signed Addition Instructions	Effect on OVC/OVCU
OVC > 0	Saturate ACC in the positive direction (fill ACC with 7FFF FFFF16) and clear OVC.
OVC < 0	Saturate ACC in the negative direction (fill ACC with 8000 000016) and clear OVC.

PM (Bits 9-7) —Product shift mode bits.

This 3-bit value determines the shift mode for any output operation from the product (P) register. The shift modes are shown in the following table. The output can be to the ALU or to memory. All instructions that are affected by the product shift mode will sign extend the P register value during a right shift operation. At reset, PM is cleared (left shift by 1 bit is the default).

PM is summarized as follows:

000 = Left shift by 1. During the shift the low-order bit is zero filled. At reset this mode is selected.

001 = No shift

010 = Right shift by 1. During the shift the lower bits are lost and the shifted value is sign extended.

011 = Right shift by 2. During the shift the lower bits are lost and the shifted value is sign extended.
100 = Right shift by 3. During the shift the lower bits are lost and the shifted value is sign extended.

101 = Right shift by 4. During the shift the lower bits are lost and the shifted value is sign extended. **Note** if AMODE = 1 then 101 is a left shift by 4.

110 = Right shift by 5. During the shift the lower bits are lost and the shifted value is sign extended.

111 = Right shift by 6. During the shift the lower bits are lost and the shifted value is sign extended.

Note: For performing unsigned arithmetic, you must use a product shift of 0 (PM = 001) to avoid sign extension and generation of incorrect results.

Table 2-6 lists instructions that are affected by the PM bits. See the instruction set in Chapter 6 for a complete description of each instruction.

Table 2-6. Instructions Affected by the PM Bits

Instruction	Effect of PM
CMPL ACC,P << PM	flags set on(ACC - P << PM)
DMAC ACC:P,loc32,*XAR7/++	ACC = ACC + MSW*MSW << PM P = P + LSW*LSW << PM
IMACL P,loc32,*XAR7/++	P = ([loc32] * Prog[*XAR7/++]) << PM
IMPYAL P,XT,loc32	P = (XT * [loc32]) << PM
IMPYL P,XT,loc32	P = (XT *[loc32]) << PM
IMPYSL P,XT,loc32	ACC = ACC - P unsigned > > P = (XT * [loc32]) << PM
IMPYXUL P,XT,loc32	P = (XT sign * [loc32]uns) << PM
MAC P,loc16,*XAR7/++	ACC = ACC + P << PM
MAC P,loc16,0:pma	ACC = ACC + P << PM
MOV loc16,P	[loc16] = low(P << PM)
MOVA T,loc16	ACC = ACC + P << PM
MOVAD T,loc16	ACC = ACC + P << PM
MOVH loc16,P	[loc16] = high(P << PM)
MOVP T,loc16	ACC = P << PM
MOVS T,loc16	ACC = ACC - P << PM
MPYA P,loc16,#16bit	ACC = ACC + P << PM
MPYA P,T,loc16	ACC = ACC + P << PM
MPYS P,T,loc16	ACC = ACC - P << PM
QMACL P,loc32,*XAR7	ACC = ACC + P << PM
QMACL P,loc32,*XAR7++	ACC = ACC + P << PM
QMPYAL P,XT,loc32	ACC = ACC + P << PM

Table 2-6. Instructions Affected by the PM Bits (continued)

Instruction	Effect of PM
QMPYSL P,XT,loc32	ACC = ACC - P << PM
SQRA loc16	ACC = ACC + P << PM
SQRS loc16	ACC = ACC - P << PM
XMAC P,loc16,*(pma)	ACC = ACC + P << PM
XMACD P,loc16,*(pma)	ACC = ACC + P << PM

V (Bit 6) — Overflow flag.

If the result of an operation causes an overflow in the register holding the result, V is set and latched. If no overflow occurs, V is not modified. Once V is latched, it remains set until it is cleared by reset or by a conditional branch instruction that tests V. Such a conditional branch clears V regardless of whether the tested condition (V = 0 or V = 1) is true.

An overflow occurs in ACC (and V is set) if the result of an addition or subtraction does not fit within the signed numerical range -231 to (+231 - 1), or 8000 0000_{16} to 7FFF FFFF_{16}.

An overflow occurs in AH, AL, or another 16-bit register or data-memory location if the result of an addition or subtraction does not fit within the signed numerical range -2^{15} to $(+2^{15} - 1)$, or 8000_{16} to $\mathrm{7FFF}_{16}$.

The instructions CMP, CMPB and CMPL do not affect the state of the V flag. Table 2-7 lists the instructions that are affected by V flag. See Chapter 6 for more details on instructions.

V can be summarized as follows:

0 = V has been cleared.

1 = An overflow has been detected, or V has been set.

Table 2-7. Instructions Affected by V flag

ABS ACC	if(ACC == 0x8000 0000) V = 1
ABSTC ACC	if(ACC == 0x8000 0000) V = 1
ADD ACC,#16bit << shift	V = 1 on signed overflow
ADD ACC,loc16 << shift	V = 1 on signed overflow
ADD ACC,loc16 << T	V = 1 on signed overflow
ADD AX,loc16	V = 1 on signed overflow
ADD loc16,#16bitSigned	V = 1 on signed overflow
ADD loc16,AX	V = 1 on signed overflow
ADDB ACC,#8bit	V = 1 on signed overflow
ADDB AX,#8bitSigned	V = 1 on signed overflow
ADDCL ACC,loc32	V = 1 on signed overflow
ADDCU ACC,loc16	V = 1 on signed overflow
ADDL ACC,loc32	V = 1 on signed overflow
ADDL loc32,ACC	V = 1 on signed overflow
ADDU ACC,loc16	V = 1 on signed overflow
ADDUL ACC,loc32	V = 1 on signed overflow
ADDUL P,loc32	V = 1 on signed overflow
B 16bitOff,COND	V = 0 if tested
BF 16bitOff,COND	V = 0 if tested
DEC loc16	V = 1 on signed overflow
DMAC ACC:P,loc33,*XAR77/++	V = 1 on signed overflow
IMACL P,loc32,*XAR77/++	V = 1 on signed overflow
DMAC ACC:P,loc32,*XAR7/++	V = 1 on signed overflow
IMACL P,loc32,*XAR7/++	V = 1 on signed overflow
IMPYAL P,XT,loc32	V = 1 on signed overflow

Table 2-7. Instructions Affected by V flag (continued)

IMPYSL P,XT,loc32	V = 1 on signed overflow
INC loc16	V = 1 on signed overflow
MAC P,loc16,*XAR7/++	V = 1 on signed overflow
MAC P,loc16,0:pma	V = 1 on signed overflow
MAX AX,loc16	if((AX - [loc16]) < 0) V = 1
MAXL ACC,loc32	if((ACC - [loc32]) < 0) V = 1
MIN AX,loc16	if((AX - [loc16]) > 0) V = 1
MINL ACC,loc32	if((ACC - [loc32]) > 0) V = 1
MOV loc16,AX,COND	V = 0 if tested
MOVA T,loc16	V = 1 on signed overflow
MOVAD T,loc16	V = 1 on signed overflow
MOVB loc16,#8bit,COND	V = 0 if tested
MOVL loc32,ACC,COND	V = 0 if tested
MOVS T,loc16	V = 1 on signed overflow
MPYA P,loc16,#16bit	V = 1 on signed overflow
MPYA P,T,loc16	V = 1 on signed overflow
MPYS P,T,loc16	V = 1 on signed overflow
NEG ACC	if(ACC == ox8000 0000) V = 1
NEGAX	if(AX == 0x8000) V = 1
NEG64 ACC:P	if(ACC:P == 0x80....00) V = 1
NEGTC ACC	if(TC == 1)
	if(ACC == 0x8000 0000) V = 1
QMACL P,loc32,*XAR7/++	V = 1 on signed overflow
QMPYAL P,XT,loc32	V = 1 on signed overflow
QMPYSL P,XT,loc32	V = 1 on signed overflow
SAT ACC	if(OVC == 0) V = 0 else V = 1
SAT64 ACC:P	if(Oâ€‹VC == 0) V = 0 else V = 1
SB 8bitOff,COND	V = 0 if tested
SBBU ACC,loc16	V = 1 on signed overflow
SQRA loc16	V = 1 on signed overflow
SQRS loc16	V = 1 on signed overflow
SUB ACC,#16bit << shift	V = 1 on signed overflow
SUB ACC,loc16 << shift	V = 1 on signed overflow
SUB ACC,loc16 << T	V = 1 on signed overflow
SUB AX,loc16	V = 1 on signed overflow
SUB loc16,AX	V = 1 on signed overflow
SUBB ACC,#8bit	V = 1 on signed overflow
SUBBL ACC,loc32	V = 1 on signed overflow
SUBL ACC,loc32	V = 1 on signed overflow
SUBL loc32,ACC	V = 1 on signed overflow
SUBR loc16,AX	V = 1 on signed overflow
SUBRL loc32,ACC	V = 1 on signed overflow
SUBU ACC,loc16	V = 1 on signed overflow
SUBUL ACC,loc32	V = 1 on signed overflow
SUBUL P,loc32	V = 1 on signed overflow
XB pma,COND	V = 0 if tested
XCALL pma,COND	V = 0 if tested
XMAC P,loc16,*(pma)	V = 1 on signed

Table 2-7. Instructions Affected by V flag (continued)

XMACD P,loc16,*(pma)	V = 1 on signed overflow
XRETC COND	V = 0 if tested

N (Bit 5) — Negative flag.

During certain operations, N is set if the result of the operation is a negative number or cleared if the result is a positive number. At reset, N is cleared.

Results in ACC are tested for the negative condition. Bit 31 of ACC is the sign bit. If bit 31 is a 0, ACC is positive; if bit 31 is a 1, ACC is negative. N is set if a result in ACC is negative or cleared if a result is positive.

Results in AH, AL, and other 16-bit registers or data-memory locations are also tested for the negative condition. In these cases bit 15 of the value is the sign bit (1 indicates negative,

0 indicates positive). N is set if the value is negative or cleared if the value is positive.

The TEST ACC instruction sets N if the value in ACC is negative. Otherwise the instruction clears N.

As shown in Table 2-8, under overflow conditions, the way the N flag is set for compare operations is different from the way it is set for addition or subtraction operations. For addition or subtraction operations, the N flag is set to match the most significant bit of the truncated result. For compare operations, the N flag assumes infinite precision. This applies to operations whose result is loaded to ACC, AH, AL, another register, or a data-memory location.

Table 2-8. Negative Flag Under Overflow Conditions

A[1]	B[1]	(A - B)	Subtraction	Compare[2]
Pos	Neg	Neg (due to overflow in positive direction)	N = 1	N = 0
Neg	Pos	Pos (due to overflow in negative direction)	N = 0	N = 1

[1] For 32-bit data: Pos = Positive nummber from 0000 0000_{16} to 7FFF FFFF_{16}. Neg=Negative number from $8000\text{-}0000_{16}$ to FFFF-FFFF_{16}. For 16-bit data: Pos = Positive number from 0000_{16} to 7FFF_{16}. Neg = Negative number from 8000_{16} to FFFF_{16}.

[2] The compare instructions are CMP, CMPB, CMPL, MIN, MAX, MINL, and MAXL.

N can be summarized as follows:

0 = The tested number is positive, or N has been cleared.

1 = The tested number is negative, or N has been set.

Z (Bit 4) — Zero flag.

Z is set if the result of certain operations is 0 or is cleared if the result is nonzero. This applies to results that are loaded into ACC, AH, AL, another register, or a data-memory location. At reset, Z is cleared.

The TEST ACC instruction sets Z if the value in ACC is 0. Otherwise, it clears Z.

Z can be summarized as follows:

0 = The tested number is nonzero, or Z has been cleared.

1 = The tested number is 0, or Z has been set.

C (Bit 3) — Carry bit.

This bit indicates when an addition or increment generates a carry or when a subtraction, compare, or decrement generates a borrow. It is also affected by rotate operations on ACC and barrel shifts on ACC, AH, and AL.

During additions/increments, C is set if the addition generates a carry; otherwise C is cleared. There is one exception: If you are using the ADD instruction with a shift of 16, the ADD instruction can set C but cannot clear C.

During subtractions/decrements/compares, C is cleared if the subtraction generates a carry; otherwise C is set. There is one exception: if you are using the SUB instruction with a shift of 16, the SUB instruction can clear C but cannot set C.

This bit can be individually set and cleared by the SETC C instruction and CLRC C instruction, respectively. At reset, C is cleared.

C can be summarized as follows:

0 = A subtraction generated a borrow, an addition did not generate a carry, or C has been cleared. *Exception:* An ADD instruction with a shift of 16 cannot clear C.

1 = An addition generated a carry, a subtraction did not generate a borrow, or C has been set. *Exception:* A SUB instruction with a shift of 16 cannot set C.

Table 2-9 lists the bits that are affected by the C bit. For more information on instructions, see Chapter 6.

Table 2-9. Bits Affected by the C Bit

Instruction	Affect of or Affect on C
ABS ACC	C = 0
ABSTC ACC	C = 0
ADD ACC,#16bit << shift	C = 1 on carry else C = 0

ADD ACC,loc16 << shift	if(shift == 16)
	C = 1 on carry
	if(shift != 16)
	C = 1 on carry else C = 0

ADD ACC,loc16 << shift	C = 1 on carry else C = 0
ADD ACC,loc16 << T	C = 1 on carry else C = 0
ADD AX,loc16	C = 1 on carry else C = 0
ADD loc16,#16bitSigned	C = 1 on carry else C = 0
ADD loc16,AX	C = 1 on carry else C = 0
ADDB ACC,#8bit	C = 1 on carry else C = 0
ADDB AX,#8bitSigned	C = 1 on carry else C = 0

ADDCL ACC,loc32	ACC = ACC + [loc32] + C
	C = 1 on carry else C = 0

ADDCU ACC,loc16	ACC = ACC + [loc16] + C
	C = 1 on carry else C = 0

ADDL ACC,loc32	C = 1 on carry else C = 0
ADDL loc32,ACC	C = 1 on carry else C = 0
ADDU ACC,loc16	C = 1 on carry else C = 0
ADDUL ACC,loc32	C = 1 on carry else C = 0
ADDUL P,loc32	C = 1 on carry else C = 0
ASR AX,1..16	C = AX(bit(shift-1))
ASR AX,T	if(T == 0) C = 0 else C =
	AX(bit(T-1))
ASR64 ACC:P,1..16	C = P(bit(shift-1))
ASR64 ACC:P,T	if(T == 0) C = 0 else C =
	P(bit(T-1))
ASRL ACC,T	if(T == 0) C = 0 else C =
	ACC(bit(T-1))
B 16bitOff,COND	C bit used as test condition
BF 16bitOff,COND	C bit used as test condition
CLRC C	C = 0
CMP AX,loc16	C = 0 on borrow else C = 1
CMP loc16,#16bitSigned	for([loc16] - 16bitSigned) C = 0
	on borrow else C = 1
CMPB AX,#8bit	C = 0 on borrow else C = 1
CMPL ACC,loc32	for(ACC - [loc32]) C = 0 on borrow
	else C = 1

Table 2-9. Bits Affected by the C Bit (continued)

Instruction	Affect of or Affect on C
CMPL ACC,P << PM	for(ACC - P << PM) C = 0 on
	borrow else C = 1
DEC loc16+	C = 0 on borrow else C = 1
DMAC ACC:P,loc32,*XAR7/++	C = 1 on carry else C = 0
IMACL P,loc32,*XAR7/++	C = 1 on carry else C = 0
IMPYAL P,XT,loc32	C = 1 on carry else C = 0
IMPYSL P,XT,loc32	C = 0 on borrow else C = 1
INC loc16	C = 1 on carry else C = 0
LSL ACC,1..16	C = ACC(bit(32-shift))
LSL ACC,T	if(T == 0) C = 0 else C =
	ACC(bit(32-T))
LSL AX,1..16	C = AX(bit(16-shift))
LSL AX,T	if(T == 0) C = 0 else C =
	AX(bit(16-T))
LSL64 ACC:P,1..16	C = ACC(bit(32-shift))
LSL64 ACC:P,T	if(T == 0) C = 0 else C =
	ACC(bit(32-T))
LSLL ACC,T	if(T == 0) C = 0 else C =
	ACC(bit(32-T))
LSR AX,1..16	C = AX(bit(shift-1))
LSR AX,T	if(T == 0) C = 0 else C =
	AX(bit(T-1))
LSR64 ACC:P,1..16	C = P(bit(shift-1))
LSR64 ACC:P,T	if(T == 0) C = 0 else C =
	P(bit(T-1))
LSRL ACC,T	if(T == 0) C = 0 else C =
	ACC(bit(T-1))
MAC P,loc16,*XAR7/++	C = 1 on carry else C = 0
MAC P,loc16,0:pma	C = 1 on carry else C = 0
MAX AX,loc16	for(AX - [loc16]) C = 0 on borrow
	else C = 1
MAXL ACC,loc32	for(ACC - [loc32]) C = 0 on borrow
	else C = 1
MIN AX,loc16	for(AX - [loc16]) C = 0 on borrow
	else C = 1
MINL ACC,loc32	for(ACC - [loc32]) C = 0 on borrow
	else C = 1
MOV loc16,AX,COND	C bit used as test condition
MOVA T,loc16	C = 1 on carry else C = 0
MOVAD T,loc16	C = 1 on carry else C = 0
MOVB loc16,#8bit,COND	C bit used as test condition
MOVL loc32,ACC,COND	C bit used as test condition
MOVS T,loc16	C = 0 on borrow else C = 1
MPYA P,loc16,#16bit	C = 1 on carry else C = 0
MPYA P,T,loc16	C = 1 on carry else C = 0
MPYS P,T,loc16	C = 0 on borrow else C = 1
NEG ACC	if(ACC == 0) C = 1 else C = 0

Table 2-9. Bits Affected by the C Bit (continued)

Instruction	Affect of or Affect on C
NEG AX	if(AX == 0) C = 1 else C = 0
NEG64 ACC:P	if(ACC:P == 0) C = 1 else C = 0
NEGTC ACC	if(TC == 1) if(ACC == 0) C = 1
	else C = 0
QMACL P,loc32,*XAR7/++	C = 1 on carry else C = 0
QMPYAL P,XT,loc32	C = 1 on carry else C = 0
QMPYSL P,XT,loc32	C = 0 on borrow else C = 1
ROL ACC	C <- (ACC << 1) <- C(before)
ROR ACC	C(before) -(ACC >1) -C
SAT ACC	C = 0
SAT64 ACC:P	C = 0
SB 8bitOff,COND	C bit used as test condition
SBBU ACC,loc16	ACC = ACC - ([loc16] + ~C) C = 0 on
	borrow else C = 1
SETC C	C = 1
SFR ACC,1..16	C = ACC(bit(shift-1))
SFR ACC,T	if(T == 0) C = 0 else C =
	ACC(bit(T-1))
SQRA loc16	C = 1 on carry else C = 0
SQRS loc16	C = 0 on borrow else C = 1
SUB ACC,#16bit << shift	C = 0 on borrow else C = 1
SUB ACC,loc16 << shift	if(shift == 16) C = 0 on borrow
	if(shift != 16) C = 0 on borrow
	else C = 1
SUB ACC,loc16 << T	C = 0 on borrow else C = 1
SUB AX,loc16	C = 0 on borrow else C = 1
SUB loc16,AX	C = 0 on borrow else C = 1
SUBB ACC,#8bit	C = 0 on borrow else C = 1
SUBBL ACC,loc32	ACC = ACC - ([loc32] + ~C) C = 0 on
	borrow else C = 1
SUBCU ACC,loc16	for(ACC - [loc16]<<15) C = 0
	on borrow else C = 1
SUBCUL ACC,loc32	for(ACC<<1 + P(31) - [loc32]) C = 0
	on borrow else C = 1
SUBL ACC,loc32	C = 0 on borrow else C = 1
SUBL loc32,ACC	C = 0 on borrow else C = 1
SUBR loc16,AX	C = 0 on borrow else C = 1
SUBRL loc32,ACC	C = 0 on borrow else C = 1
SUBU ACC,loc16	C = 0 on borrow else C = 1
SUBUL ACC,loc32	C = 0 on borrow else C = 1
SUBUL P,loc32	C = 0 on borrow else C = 1
XB pma,COND	C bit used as test condition
XCALL pma,COND	C bit used as test condition
XMAC P,loc16,*(pma)	C = 1 on carry else C = 0
XMACD P,loc16,*(pma)	C = 1 on carry else C = 0
XRETC COND	C bit used as test condition

TC (Bit 2) — Test/control flag.

This bit shows the result of a test performed by either the TBIT (test bit) instruction or the NORM (normalize) instruction.

The TBIT instruction tests a specified bit. When TBIT is executed, the TC bit is set if the tested bit is 1 or cleared if the tested bit is 0.

When a NORM instruction is executed, TC is modified as follows: If ACC holds 0, TC is set. If ACC does not hold 0, the CPU calculates the exclusive-OR of ACC bits 31 and 30, and then loads TC with the result.

This bit can be individually set and cleared by the SETC TC instruction and CLRC TC instruction, respectively. At reset, TC is cleared.

Table 2-10 lists the instructions that affect the TC bit. See the instruction set in Chapter 6 for a complete description of each instruction.

Table 2-10. Instructions That Affect the TC Bit

Instruction	Affect on the TC bit
ABSTC ACC	if(ACC < 0) TC = TC ^ 1
B 16bitOff,COND	TC bit used as test condition
BF 16bitOff,COND	TC bit used as test condition
CLRC TC	TC = 0
CMPR 0/1/2/3	TC = 0 0: if(AR(ARP) == AR0) TC = 1 1: if(AR(ARP) < AR0) TC = 1 2: if(AR(ARP) > AR0) TC = 1 3: if(AR(ARP) != AR0) TC = 1
CSB ACC	TC = N flag
MOV loc16,AX,COND	TC bit used as test condition
MOVB loc16,#8bit,COND	TC bit used as test condition
MOVL loc32,ACC,COND	TC bit used as test condition
NEGTC ACC	TC bit used as test condition

NORM ACC,XARn++/--
NORM ACC,*ind
	if(ACC |= 0)
	TC = ACC(31) ^ ACC(30)
	else
	TC = 1

SB 8bitOff,COND	TC bit used as test condition
SBF 8bitOff,TC/NTC	TC bit used as test condition
SETC TC	TC = 1
TBIT loc16,#bit	TC = [loc16(bit)]
TBIT loc16,T	TC = [loc16(15-T)]
TCLR loc16,#bit	TC = [loc16(bit)]
TSET loc16,#bit	TC = [loc16(bit)]
XB pma,COND	TC bit used as test condition
XCALL pma,COND	TC bit used as test condition
XRETC COND	TC bit used as test condition

OVM (Bit 1) — Overflow mode bit.

When ACC accepts the result of an addition or subtraction and the result causes an overflow, OVM determines how the CPU handles the overflow as follows:

0 = Results overflow normally in ACC. The OVC reflects the overflow

1 = ACC is filled with either its most positive or most negative value as follows:
If ACC overflows in the positive direction (from 7FFF $FFFF_{16}$ to 8000 0000_{16}), ACC is then filled with 7FFF $FFFF_{16}$.
If ACC overflows in the negative direction (from 8000 0000_{16} to 7FFF $FFFF_{16}$), ACC is then filled with 8000 0000_{16}.

This bit can be individually set and cleared by the SETC OVM instruction and CLRC OVM instruction, respectively. At reset, OVM is cleared.

SXM (Bit 0) — Sign-extension mode bit.

SXM affects the MOV, ADD, and SUB instructions that use a 16-bit value in an operation on the 32-bit accumulator. When the 16-bit value is loaded into (MOV), added to (ADD), or subtracted from (SUB) the accumulator, SXM determines whether the 16-bit value is sign extended during the operation as follows:

0 = Sign extension is suppressed. (The 16-bit value is treated as unsigned.)

1 = Sign extension is enabled. (The 16-bit value is treated as signed.)

For example:

```
ADD ACC, loc16 << shift
 if SXM = 0, do not sign extend loc16 before adding to the 32-bit ACC.
 if SXM = 1, sign extend loc16 before adding to the 32-bit ACC.
```

SXM also determines whether the accumulator is sign extended when it is shifted right by the SFR instruction. SXM does not affect instructions that shift the product register value; all right shifts of the product register value use sign extension.

This bit can be individually set and cleared by the SETC SXM instruction and CLRC SXM instruction, respectively. At reset, SXM is cleared. Table 2-11 lists the instructions that are affected by SXM. See Chapter 6 for more details on instructions.

Table 2-11. Instructions Affected by SXM

Instruction	Description
ADD ACC,#16bit << shift	Affected By SXM
ADD ACC,loc16 << shift	Affected By SXM
ADD ACC,loc16 << T	Affected By SXM
CLRC SXM	SXM = 0
MOV ACC,#16bit << shift	Affected By SXM
MOV ACC,loc16 << shift	Affected By SXM
MOV ACC,loc16 << T	Affected By SXM
SETC SXM	SXM = 1
SFR ACC,1..16	Affected By SXM
SFR ACC,T	Affected By SXM
SUB ACC,#16bit << shift	Affected By SXM
SUB ACC,loc16 << shift	Affected By SXM
SUB ACC,loc16 << T	Affected By SXM

2.4 Status Register ST1

The following figure shows the bit fields of status register ST1. All of these bit fields are modified in the decode 2 phase of the pipeline. Detailed descriptions of these bits follow the figure.

Table 2-12. Bit Fields of Status Register 1 (ST1)

15		13	12	11	10	9	8
ARP			XF	MOM1MAP	Reserved	OBJMODE	AMODE
R/W-000			R/W-0	R/W-1	R/W-0	R/W-0	R/W-0

7	6	5	4	3	2	1	0
IDLESTAT	EALLOW	LOOP	SPA	VMAP	PAGE0	DBGM	INTM
R-0	R/W-0	R-0	R/W-0	R/W-1	R/W-0	R/W-1	R/W-1

LEGEND: R/W = Read/Write; R = Read only; -*n* = value after reset

ARP (Bits 15-13) — Auxiliary register pointer.

This 3-bit field points to the current auxiliary register. This is one of the 32-bit auxiliary registers (XAR0-XAR7). The mapping of ARP values to auxiliary registers is as follows:

000 = XAR0 (selected at reset)

001 = XAR1

010 = XAR2

011 = XAR3

100 = XAR4

101 = XAR5

110 = XAR6

111 = XAR7

XF (Bit 12) —XF status bit.

This bit reflects the current state of the XFS output signal, which is compatible to the C2XLP CPU. This bit is set by the "M SETC XF"M instruction. This bit is cleared by the "M CLRC XF"M instruction. The pipeline is not flushed when setting or clearing this bit using the given instructions. This bit can be saved and restored by interrupts and when restoring the ST1 register. This bit is set to 0 on reset.

NOTE: Use of the XFS signal requires an external pin that is only present on TMS320x2801x devices.

M0M1MAP (Bit 11) —M0 and M1 mapping mode bit.

The M0M1MAP bit should always remain set to 1 in the C28x object mode. This is the default value at reset. The M0M1MAP bit may be set low when operating in C27x-compatible mode. The effect of this bit, when low, is to swap the location of blocks M0 and M1 only in program space and to set the stack pointer default reset value to 0x000. C28x mode users should never set this bit to 0.

Reserved (Bit 10) —Reserved.

This bit is reserved. Writes to this bit have no effect.

OBJMODE (Bit 9) — Object compatibility mode bit.

This mode is used to select between C27x object mode (OBJMODE == 0) and C28x object mode (OBJMODE == 1) compatibility. This bit is set by the "M C28OBJ"M (or "M SETC OBJMODE"M) instructions. This bit is cleared by the "M C27OBJ"M (or "M CLRC OBJMODE"M) instructions. The pipeline is flushed when setting or clearing this bit using the given instructions. This bit is saved and restored by interrupts and when restoring the ST1 register. This bit is set to 0 on reset.

AMODE (Bit 8) — Address mode bit.

This mode, in conjunction with the PAGE0 mode bit, is used to select the appropriate addressing mode decodes. This bit is set by the "LPADDR"M ("M SETC AMODE"M) instructions. This bit is cleared by the "M C28ADDR"M (or "M CLRC AMODE"M) instructions. The pipeline is not flushed when setting or clearing this bit using the given instructions. This bit is saved and restored by interrupts and when restoring the ST1 register. This bit is set to 0 on reset.

Note: Setting PAGE0 = AMODE = 1 will generate an illegal instruction trap ONLY for instructions that decode a memory or register addressing mode field (loc16 or loc32).

IDLESTAT (Bit 7) — IDLE status bit.

This read-only bit is set when the IDLE instruction is executed. It is cleared by any one of the following events:

- An interrupt is serviced.
- An interrupt is not serviced but takes the CPU out of the IDLE state.
- A valid instruction enters the instruction register (the register that holds the instruction currently being decoded).
- A device reset occurs.

When the CPU services an interrupt, the current value of IDLESTAT is saved on the stack (when ST1 is saved on the stack), and then IDLESTAT is cleared. Upon return from the interrupt, IDLESTAT is not restored from the stack.

EALLOW (Bit 6) — Emulation access enable bit.

This bit, when set, enables access to emulation and other protected registers. Set this bit by using the EALLOW instruction and clear this bit by using the EDIS instruction. See the data sheet for a particular device to determine the registers that are protected.

When the CPU services an interrupt, the current value of EALLOW is saved on the stack (when ST1 is saved on the stack), and then EALLOW is cleared. Therefore, at the start of an interrupt service routine (ISR), access to protected registers is disabled. If the ISR must access protected registers, it must include an EALLOW instruction. At the end of the ISR, EALLOW can be restored by the IRET instruction.

LOOP (Bit 5)— Loop instruction status bit.

LOOP is set when a loop instruction (LOOPNZ or LOOPZ) reaches the decode 2 phase of the pipeline. The loop instruction does not end until a specified condition is met. When the condition is met, LOOP is cleared. LOOP is a read-only bit; it is not affected by any instruction except a loop instruction.

When the CPU services an interrupt, the current value of LOOP is saved on the stack (when ST1 is saved on the stack), and then LOOP is cleared. Upon return from the interrupt, LOOP is not restored from the stack.

SPA (Bit 4) — Stack pointer alignment bit.

SPA indicates whether the CPU has previously aligned the stack pointer to an even address by the ASP instruction:

- 0: The stack pointer has not been aligned to an even address.
- 1: The stack pointer has been aligned to an even address.

When the ASP (align stack pointer) instruction is executed, if the stack pointer (SP) points to an odd address, SP is incremented by 1 so that it points to an even address, and SPA is set. If SP already points to an even address, SP is not changed, but SPA is cleared. When the NASP (unalign stack pointer) instruction is executed, if SPA is 1, SP is decremented by 1 and SPA is cleared. If SPA is 0, SP is not changed.

At reset, SPA is cleared.

VMAP (Bit 3) — Vector map bit.

VMAP determines whether the CPU interrupt vectors (including the reset vector) are mapped to the lowest or highest addresses in program memory:

- 0: CPU interrupt vectors are mapped to the bottom of program memory, addresses 00 0000_{16}-00 $003F_{16}$.

- 1: CPU interrupt vectors are mapped to the top of program memory, addresses 3F FFC0_{16}-3F FFFF_{16}.

On C28x designs, the VMAP signal is tied high internally, forcing the VMAP bit to be set high on a reset.

This bit can be individually set and cleared by the SETC VMAP instruction and CLRC VMAP instruction, respectively.

PAGE0 (Bit 2) — PAGE0 addressing mode configuration bit.

PAGE0 selects between two mutually-exclusive addressing modes: PAGE0 direct addressing mode and PAGE0 stack addressing mode. Selection of the modes is as follows:

- 0: PAGE0 stack addressing mode
- 1: PAGE0 direct addressing mode

Note: Illegal Instruction Trap Setting PAGE0 = AMODE = 1 will generate an illegal instruction trap.

PAGE0 = 1 is included for compatibility with the C27x. the recommended operating mode for C28x is PAGE0 = 0.

This bit can be individually set and cleared by the SETC PAGE0 instruction and

CLRC PAGE0 instruction, respectively. At reset, the PAGE0 bit is cleared (PAGE0 stack addressing mode is selected).

For details about the above addressing modes, see Chapter 5, *Addressing Modes*.

DBGM (Bit 1) — Debug enable mask bit.

When DBGM is set, the emulator cannot accesss memory or registers in real time. The debugger cannot update its windows.

In the real-time emulation mode, if DBGM = 1, the CPU ignores halt requests or hardware breakpoints until DBGM is cleared. DBGM does not prevent the CPU from halting at a software breakpoint. One effect of this may be seen in real-time emulation mode.

If you single-step an instruction in real time emulation mode and that instruction sets DBGM, the CPU continues to execute instructions until DBGM is cleared.

When you give the TI debugger the REALTIME command (to enter real-time mode), DBGM is forced to 0. Having DBGM = 0 ensures that debug and test direct memory accesses (DT-DMAs) are allowed; memory and register values can be passed to the host processor for updating debugger windows.

Before the CPU executes an interrupt service routine (ISR), it sets DBGM. When DBGM = 1, halt requests from the host processor and hardware breakpoints are ignored. If you want to single-step through or set breakpoints in a non-time-critical ISR, you must add a CLRC DBGM instruction at the beginning of the ISR.

DBGM is primarily used in emulation to block debug events in time-critical portions of program code. DBGM enables or disables debug events as follows:

- 0: Debug events are enabled.
- 1: Debug events are disabled.

When the CPU services an interrupt, the current value of DBGM is saved on the stack (when ST1 is saved on the stack), and then DBGM is set. Upon return from the interrupt, DBGM is restored from the stack.

This bit can be individually set and cleared by the SETC DBGM instruction and CLRC DBGM instruction, respectively. DBGM is also set automatically during interrupt operations. At reset, DBGM is set. Executing the ABORTI (abort interrupt) instruction also sets DBGM.

INTM (Bit 0) —Interrupt global mask bit.

This bit globally enables or disables all maskable CPU interrupts (those that can be blocked by software):

- 0: Maskable interrupts are globally enabled. To be acknowledged by the CPU, a maskable interrupt must also be locally enabled by the interrupt enable register (IER).

- 1: Maskable interrupts are globally disabled. Even if a maskable interrupt is locally enabled by the IER, it is not acknowledged by the CPU.

INTM has no effect on the nonmaskable interrupts, including a hardware reset or the hardware interrupt NMI. In addition, when the CPU is halted in real-time emulation mode, an interrupt enabled by the IER and the DBGIER will be serviced even if INTM is set to disable maskable interrupts.

When the CPU services an interrupt, the current value of INTM is saved on the stack (when ST1 is saved on the stack), and then INTM is set. Upon return from the interrupt, INTM is restored from the stack.

This bit can be individually set and cleared by the SETC INTM instruction and CLRC INTM instruction, respectively. At reset, INTM is set. The value in INTM does not cause modification to the interrupt flag register (IFR), the interrupt enable register (IER), or the debug interrupt enable register (DBGIER).

2.5 Program Flow

The program control logic and program-address generation logic work together to provide proper program flow. Normally, the flow of a program is sequential: the CPU executes instructions at consecutive program-memory addresses. At times, a discontinuity is required; that is, a program must branch to a nonsequential address and then execute instructions sequentially at that new location. For this purpose, the '28x supports interrupts, branches, calls, returns, and repeats.

Proper program flow also requires smooth flow at the instruction level. To meet this need, the '28x has a protected pipeline and an instruction-fetch mechanism that attempts to keep the pipeline full.

2.5.1 Interrupts

Interrupts are hardware or software-driven events that cause the CPU to suspend its current program sequence and execute a subroutine called an interrupt service routine. Interrupts are described in detail in Section 3.1.

2.5.2 Branches, Calls, and Returns

Branches, calls, and returns break the sequential flow of instructions by transferring control to another location in program memory. A branch only transfers control to the new location. A call also saves the return address (the address of the instruction following the call). Called subroutines or interrupt service routines are each concluded with a return instruction, which takes the return address from the stack or from XAR7 or RPC and places it into the program counter (PC).

The following branch instructions are conditional: B, BANZ, BAR, BF, SB, SBF, XBANZ, XCALL, and XRETC. They are executed only if a certain specified or predefined condition is met. For detailed descriptions of these instructions, see Chapter 6.

2.5.3 Repeating a Single Instruction

The repeat (RPT) instruction allows the execution of a single instruction (N + 1) times, where N is specified as an operand of the RPT instruction. The instruction is executed once and then repeated N times. When RPT is executed, the repeat counter (RPTC) is loaded with N. RPTC is then decremented every time the repeated instruction is executed, until RPTC equals 0. For a description of RPT and a list of repeatable instructions, see Chapter 6.

2.5.4 Instruction Pipeline

Each instruction passes through eight independent phases that form an instruction pipeline. At any given time, up to eight instructions may be active, each in a different phase of completion. Not all reads and writes happen in the same phases, but a pipeline-protection mechanism stalls instructions as needed to ensure that reads and writes to the same location happen in the order in which they are programmed.

To maximize pipeline efficiency, an instruction-fetch mechanism attempts to keep the pipeline full. Its role is to fill an instruction-fetch queue, which holds instructions in preparation for decoding and execution. The instruction-fetch mechanism fetches 32-bits at a time from program memory; it fetches one 32-bit instruction or two 16-bit instructions.

The instruction-fetch mechanism uses three program-address counters: the program counter (PC), the instruction counter (IC), and the fetch counter (FC). When the pipeline is full the PC will always point to the instruction in its decode 2 pipeline phase. The IC points to the next instruction to be processed. When the PC points to a 1-word instruction, IC = (PC+1); when the PC points to a 2-word instruction, IC = (PC+2). The value in the FC is the address from which the next fetch is to be made.

The pipeline and the instruction-fetch mechanism are described in more detail in Chapter 4.

2.6 Multiply Operations

The C28x features a hardware multiplier that can perform 16-bit × 16-bit or 32-bit × 32-bit fixed-point multiplication. This functionality is enhanced by 16-bit × 16-bit multiply and accumulate (MAC), 32 × 32 MAC, and 16-bit × 16-bit dual MAC (DMAC) instructions. This section describes the components involved in each type of multiplication.

2.6.1 16-bit × 16-bit Multiplication

The C28x multiplier can perform a 16-bit × 16-bit multiplication to produce a signed or unsigned 32-bit product. Figure 2-10 shows the CPU components involved in this multiplication.

The multiplier accepts two 16-bit inputs:

- One input is from the upper 16 bits of the multiplicand register (T). Most 16 × 16 multiplication instructions require that you load T from a datamemory location or a register before you execute the instruction. However, the MAC and some versions of the MPY and MPYA instructions load T for you before the multiplication.
- The other input is from one of the following:
  - A data-memory location or a register (depending on which you specify in the multiply instruction).
  - An instruction opcode. Some C28x multiply instructions allow you to include a constant as an operation.

After the value has been multiplied by the second value, the 32-bit result is stored in one of two places, depending on the particular multiply instruction: the 32-bit product register (P) or the 32-bit accumulator (ACC).

One special 16-bit × 16-bit multiplication instruction takes two 32-bit input values as its operands. This instruction is the 16 × 16 DMAC instruction, which performs dual 16 × 16 MAC operations in one instruction. In this case, the ACC contains the result of multiplying and adding the upper word of the 32-bit operands. The P register contains the result of multiplying and adding the results of the lower word of the 32-bit operands.

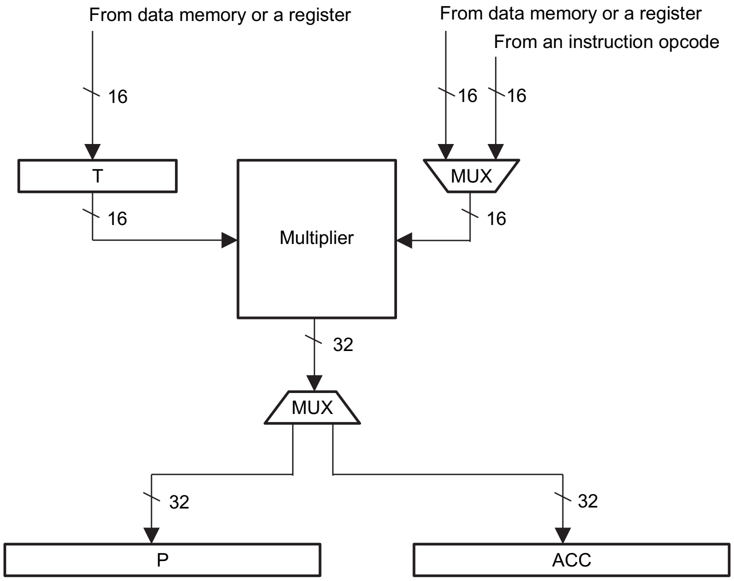

From data memory or a register
From data memory or a register
From an instruction opcode
16
16
16
T
MUX
16
16
Multiplier
32
MUX
32
32
P
ACC

Figure 2-10. Conceptual Diagram of Components Involved in 16 X16-Bit Multiplication

2.6.2 32-Bit × 32-Bit Multiplication

The C28x multiplier can also perform 32-bit by 32-bit multiplication. Figure 2-11 shows the CPU components involved n this multiplication. In this case, the multiplier accepts two 32-bit inputs:

- The first input is from one of the following:
  - A program memory location. Some C28x 32 × 32 multiply MAC-type instructions such as IMACL and QMACL take one data value directly from memory using the program-address bus.
  - The 32-bit multiplicand register (XT). Most 32 × 32-bit multiplication instructions require that you load XT from data memory or a register before you execute the instruction.
- A data-memory location or a register (depending on which you specify in the multiply instruction).

After the two values have ben multiplied, 32 bits of the 64-bit result are stored in the product register (P). You can control which half is stored (upper 32 bits or lower 32 Bits) and whether the multiplication is signed or unsigned by the instruction used.

If you need support for larger data values, the 32 X 32 multiplication instructions can be combined to implement 32 × 32 = 64-bit or 64 × 64 = 128-bit math.

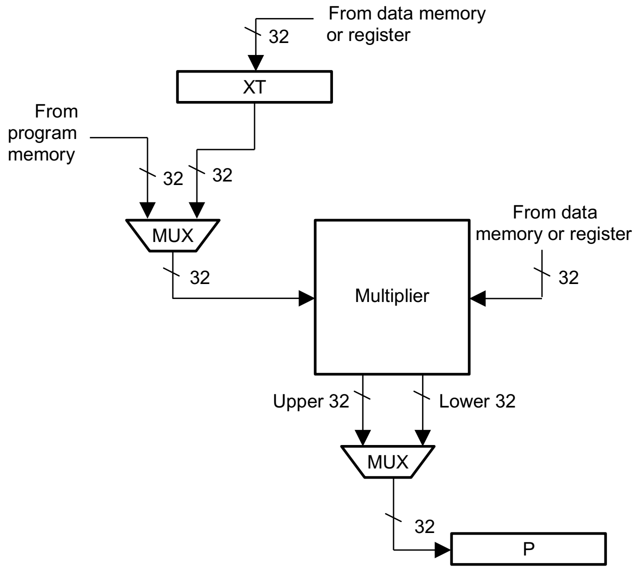


Figure 2-11. Conceptual Diagram of Components Involved in 32 X 32-Bit Multiplication

2.7 Shift Operations

The shifter holds 64 bits and accepts either a 16-bit, 32-bit, or 64-bit input value. When the input value has 16 bits, the value is loaded into the 16 least significant bits (LSBs) of the shifter. When the input value has 32 bits, the value is loaded into the 32 LSBs of the shifter. Depending on the instruction that uses the shifter, the output of the shifter may be all of its 64 bits or just its 16 LSBs.

When a value is shifted *right* by an amount N, the N LSBs of the value are lost and the bits to the left of the value are filled with all 0s or all 1s. If sign extension is specified, the bits to the left are filled with copies of the sign bit. If sign extension is not specified, the bits to the left are filled with 0s, or zero filled.

When a value is shifted *left* by an amount N, the bits to the right of the shifted value are zero filled. If the value has 16 bits and sign extension is specified, the bits to the left are filled with copies of the sign bit. If the value has 16 bits and sign extension is not specified, the bits to the left are zero filled. If the value has 32 bits, the N MSBs of the value are lost, and sign extension is irrelevant.

The figure below lists the instructions that use the shifter and provides an illustration of the corresponding shifter operation. The table uses the following graphical symbols:

Shift left	This symbol represents the 32-bit shifter. The text inside the box indicates the direction of the shift.
0	This symbol indicates zero filling.
Sign	This symbol indicates sign extending.
SXM 0/Sign	This symbol indicates that the MSBs of the shifter depend on the sign-extension mode bit (SXM). If SXM = 0, the MSBs are zero filled after the shift. If SXM = 1, the MSBs are filled with the sign of the shifted value.
C	This symbol indicates the carry bit (C).

Figure 2-12.

Table 2-13. Shift Operations

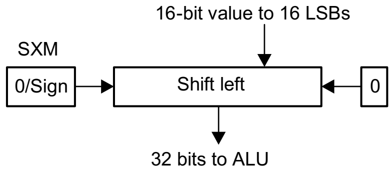
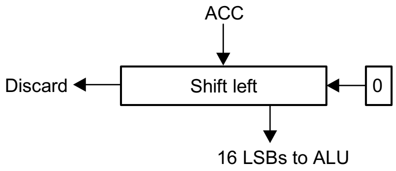
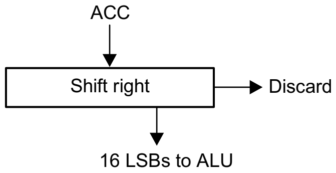
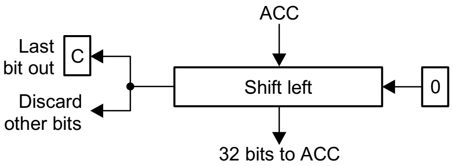
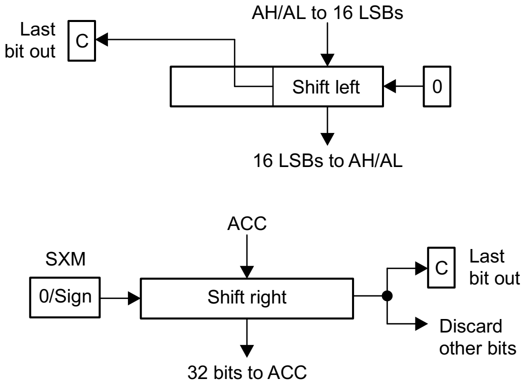
Operation Type	Illustration
Left shift of 16-bit value for ACC operation Syntaxes: ADD ACC, *loc16 << 0...16* ADD ACC, # *16Bit << 0...15* ADD ACC, *loc16* <<T SUB ACC, *loc16 << 0...16* SUB ACC, # *16Bit << 0 ...15* SUB ACC, *loc16<< T* MOV ACC, *loc16 << 0...16* MOV ACC, # *16Bit << 0...15* MOV ACC , *loc16, << T*	 16-bit value to 16 LSBs SXM 0/Sign Shift left 0 32 bits to ALU **Figure 2-13. ADD Command**
Store 16 LSBs of left-shifted ACC Syntax: MOV *loc16*, ACC << *1...8*	 **Figure 2-14. MOV Command**
Store 16 MSBs of left-shifted ACC. Syntax: MOVH *loc16*, ACC << *1...8* **Note:** This instruction performs a single right shift by (16- *shift1*), where *shift1* is a value from 0 to 8.	 **Figure 2-15. MOVH Command**
Logical left shift of ACC. The last bit to be shifted out fills the carry bit (C) Syntaxes: LSL ACC, *1...16* LSL ACC, T (shift = T(3:0)) LSL ACC, T (shift = T(4:0)) **Note:** If T(3:0) = 0 or T(4:0) = 0, indicating a shift of 0, C is cleared.	 ACC Last bit out C Shift left 0 Discard other bits 32 bits to ACC **Figure 2-16. LSL Command**
Logical left shift of AH or AL. The last AH/AL to 16 LSBs bit to be shifted out fills the carry bit (C). Syntaxes: LSL A *X*, *1...16* LSL A *X*, T (shift = T(3:0)) **Note:** If T(3:0) = 0, indicating a shift of 0, C is cleared. **Right shift of ACC.** If SXM = 0, a logical shift is performed. If SXM = 1, an arithmetic shift is performed. The last bit to be shifted out fills the carry bit (C). Syntaxes: SFR ACC, *1...16* SFR ACC, T **Note:** If T(3:0) = 0, indicating a shift of 0, C is cleared.	 Last bit out C AH/AL to 16 LSBs Shift left 0 16 LSBs to AH/AL ACC SXM 0/Sign Shift right C Last bit out Discard other bits 32 bits to ACC **Figure 2-17. LSL and SFR Commands**

Table 2-13. Shift Operations (continued)

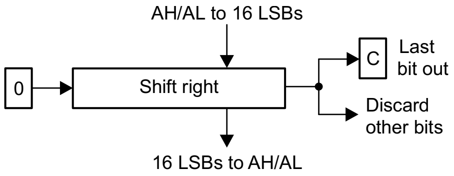
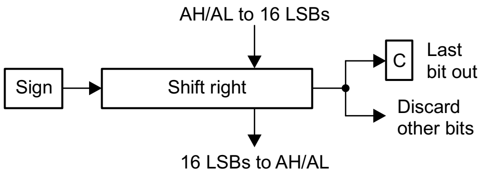
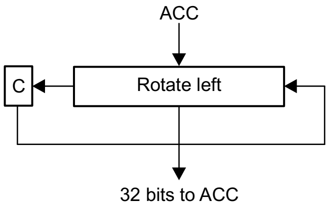
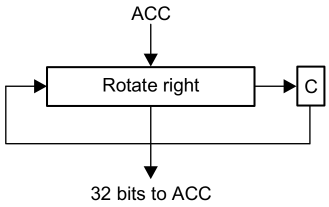
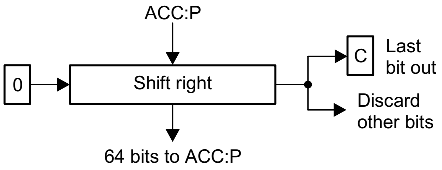
Operation Type	Illustration
Logical right shift of AH or AL. The last bit to be shifted out fills the carry bit (C). Syntaxes: LSR A *X*, *shift* LSR A *X*, T (shift = T(3:0)) ARLACC, T (shift = T(4:0) **Note:** If T(4:0) = 0, indicating a shift of 0, C is cleared.	 AH/AL to 16 LSBs 0 Shift right C Last bit out Discard other bits 16 LSBs to AH/AL **Figure 2-18. LSR Command**
Arithmetic right shift of AH or AL. The last bit to be shifted out fills the carry bit (C). Syntaxes: ASR A *X*, *shift* ASR A *X*, T **Note:** If T(4:0) = 0, indicating a shift of 0, C is cleared.	 AH/AL to 16 LSBs Sign Shift right C Last bit out Discard other bits 16 LSBs to AH/AL **Figure 2-19. ASR Command**
Rotate ACC left by 1 bit. Bit 31 of ACC fills the carry bit (C). C fills bit 0 of ACC. Syntax: ROL ACC	 **Figure 2-20. ROL Command**
Rotate ACC right by 1 bit. Bit 0 of ACC fills the carry bit (C). C fills bit 31 of ACC. Syntax: ROR ACC	 **Figure 2-21. ROR Command**
Logical right shift of ACC:P. Syntaxes LSR64 ACC:P, 1...16 LSR64, ACC:P, T shift = T(5:0)	 ACC:P 0 Shift right C Last bit out Discard other bits 64 bits to ACC:P **Figure 2-22. LSR64 Command**

Table 2-13. Shift Operations (continued)

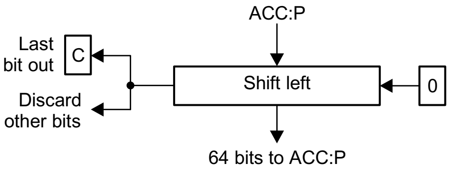
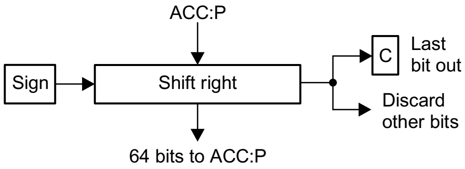
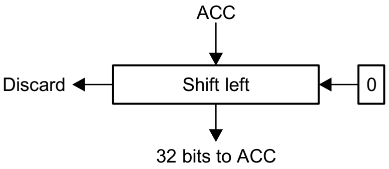
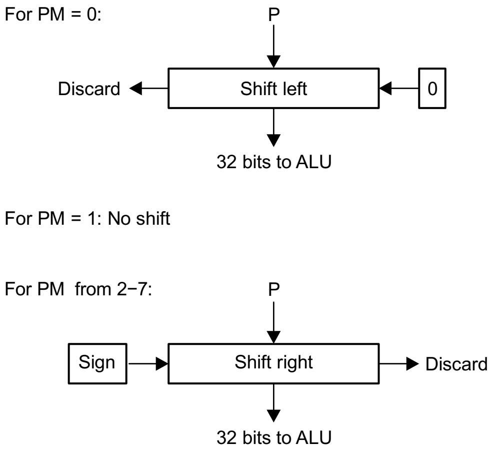
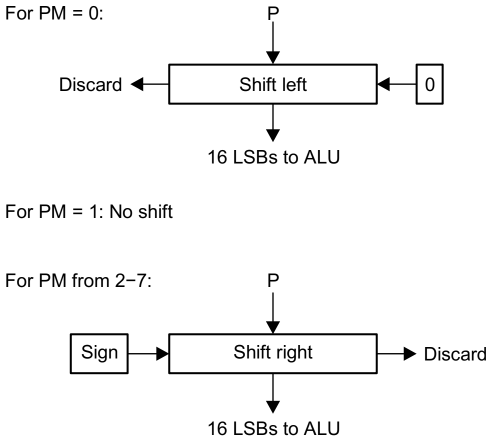
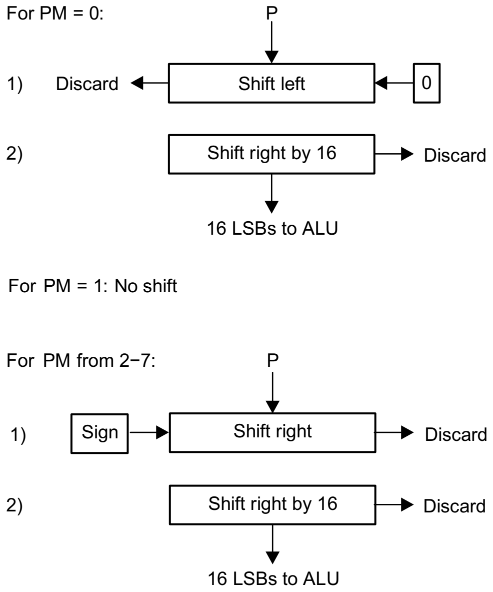
Operation Type	Illustration
Logical left shift of ACC:P. Syntaxes: LSL64 ACC:P, 1...16 LSL64 ACC:P, T shift = T(5:0)	 ACC:P Last bit out C Shift left 0 Discard other bits 64 bits to ACC:P **Figure 2-23. LSL64 Command**
Arithmetic right shift of ACC:P. Syntaxes: ASR64 ACC:P, 1...16 ASR64, ACC:P, T shift = T(5:0)	 ACC:P Sign Shift right C Last bit out Discard other bits 64 bits to ACC:P **Figure 2-24. ASR64 Command**
Conditional shift of ACC by 1 bit. Syntaxes: NORM ACC, *aux*++ NORM ACC, *aux*- - SUBCU ACC, *loc*	 **Figure 2-25. NORM and SUBCU Command**
Shift of P as per PM bits. Syntaxes: ADD ACC, P SUB ACC, P CMP ACC, P MAC P, *loc*, 0: *pmem* MOV ACC, P MOVA T, *loc* MOVP T, *loc* MOVS T, *loc* MPYA P, *loc*, # *16BitSigned* MPYA P, T, *loc* MPYS P, T, *loc*	 For PM = 0: P Discard Shift left 0 32 bits to ALU For PM = 1: No shift For PM from 2−7: P Sign Shift right Discard 32 bits to ALU **Figure 2-26. Shift Operations**

Table 2-13. Shift Operations (continued)

Operation Type	Illustration
Store 16 LSBs of shifted P. P is shifted as per the PM bits. The 16 LSBs of shifter are stored. Syntax: MOV *loc16*, P	 For PM = 0: P Discard Shift left 0 16 LSBs to ALU For PM = 1: No shift For PM from 2−7: P Sign Shift right Discard 16 LSBs to ALU **Figure 2-27. MOV Command**
Store 16 MSBs of shifted P. P is shifted as per the PM bits. The result is shifted right by 16 so that its 16 MSBs are in the **For PM = 0:** P 16 LSBs of the shifter. 16 LSBs of shifter are stored. Syntax: MOVH *loc16*, P	 For PM = 0: P 1) Discard Shift left 0 2) Shift right by 16 Discard 16 LSBs to ALU For PM = 1: No shift For PM from 2−7: P 1) Sign Shift right Discard 2) Shift right by 16 Discard 16 LSBs to ALU **Figure 2-28. MOV Command**

Chapter 3

SPRU430F–August 2001–Revised April 2015

CPU Interrupts and Reset

This chapter describes the available CPU interrupts and how they are handled by the CPU. It also explains how to control those interrupts that can be controlled through software. Finally, it describes how a hardware reset affects the CPU.

Topic Page

3.1 CPU Interrupts Overview

Interrupts are hardware- or software-driven signals that cause the C28x CPU to suspend its current program sequence and execute a subroutine. Typically, interrupts are generated by peripherals or hardware devices that need to give data to or take data from the C28x (for example, A/D and D/A converters and other processors). Interrupts can also signal that a particular event has taken place (for example, a timer has finished counting).

On the C28x, interrupts can be triggered by software (the INTR, OR IFR, or TRAP instruction) or by hardware (a pin, an external peripheral, or on-chip peripheral/logic). If hardware interrupts are triggered at the same time, the C28x services them according to a set priority ranking.

Some 28x devices include a peripheral interrupt expansion (PIE) module that multiplexes interrupts from a number of peripherals into a single CPU interrupt. The PIE module provides additional control before an interrupt reaches the C28x CPU. See the *TMS320C8x System and Interrupts Reference Guide* (SPRU078) for more details.

At the CPU level, each of the C28x interrupts, whether hardware or software, can be placed in one of the following two categories:

- **Maskable interrupts.** These are interrupts that can be blocked (masked) or enabled (unmasked) through software.
- **Nonmaskable interrupts.** These interrupts cannot be blocked. The C28x will immediately approve this type of interrupt and branch to the corresponding subroutine. All software-initiated interrupts are in this category.

The C28x handles interrupts in four main phases:

1. **Receive the interrupt request.** Suspension of the current program sequence must be requested by a software interrupt (from program code) or a hardware interrupt (from a pin or an on-chip device).
2. **Approve the interrupt.** The C28x must approve the interrupt request. If the interrupt is maskable, certain conditions must be met in order for the C28x to approve it. For nonmaskable hardware interrupts and for software interrupts, approval is immediate.
3. **Prepare for the interrupt service routine and save register values.** The main tasks performed in this phase are:
   - Complete execution of the current instruction and flush from the pipeline any instructions that have not reached the decode 2 phase.
   - Automatically save most of the current program context by saving the following registers to the stack: ST0, T, AL, AH, PL, PH, AR0, AR1, DP, ST1, DBGSTAT, PC, and IER.
   - Fetch the interrupt vector and load it into the program counter (PC). For devices with a PIE module, the vector fetched will depend on the setting of the PIE enable and flag registers.
4. **Execute the interrupt service routine.** The C28x branches to its corresponding subroutine called an interrupt service routine (ISR). The C28x branches to the address (vector) you store at a predetermined vector location and executes the ISR you have written.

3.2 CPU Interrupt Vectors and Priorities

The C28x supports 32 CPU interrupt vectors, including the reset vector. Each vector is a 22-bit address that is the start address for the corresponding interrupt service routine (ISR). Each vector is stored in 32 bits at two consecutive addresses. The location at the lower address holds the 16 least significant bits (LSBs) of the vector. The location at the higher address holds the 6 most significant bits (MSBs) right-justified. When an interrupt is approved, the 22-bit vector is fetched, and the 10 MSBs at the higher address are ignored.

The C28x supports 32 CPU interrupt vectors, including the reset vector. Each vector is a 22-bit address that is the start address for the corresponding interrupt service routine (ISR). Each vector is stored in 32 bits at two consecutive addresses. The location at the lower address holds the 16 least significant bits (LSBs) of the vector. The location at the higher address holds the 6 most significant bits (MSBs) right-justified. When an interrupt is approved, the 22-bit vector is fetched, and the 10 MSBs at the higher address are ignored.

For devices with a PIE module, this table is re-mapped and expanded into the PIE vector table.

Table 3-1 lists the available CPU interrupt vectors and their locations. The addresses are shown in hexadecimal form. The table also shows the priority of each of the hardware interrupts.

Table 3-1. Interrupt Vectors and Priorities

Vector	Absolute Address (hexadecimal)		Hardware Priority	Description
	VMAP = 0	VMAP = 1[1]		
RESET	00 0000	3F FFC0	1 (highest)	Reset
INT1	00 0002	3F FFC2	5	Maskable Interrupt 1
INT2	00 0004	3F FFC4	6	Maskable interrupt 2
INT3	00 0006	3F FFC6	7	Maskable interrupt 3
INT4	00 0008	3F FFC8	8	Maskable interrupt 4
INT5	00 000A	3F FFCA	9	Maskable interrupt 5
INT6	00 000C	3F FFCC	10	Maskable interrupt 6
INT7	00 000E	3F FFCE	11	Maskable interrupt 7
INT8	00 0010	3F FFD0	12	Maskable interrupt 8
INT9	00 0012	3F FFD2	13	Maskable interrupt 9
INT10	00 0014	3F FFD4	14	Maskable interrupt 10
INT11	00 0016	3F FFD6	15	Maskable interrupt 11
INT12	00 0018	3F FFD8	16	Maskable interrupt 12
INT13	00 001A	3F FFDA	17	Maskable interrupt 13
INT14	00 001C	3F FFDC	18	Maskable interrupt 14
DLOGINT[2]	00 001E	3F FFDE	19 (lowest)	Maskable data log interrupt
RTOSINT [2]	00 0020	3F FFE0	4	Maskable real-time operating system interrupt
Reserved	00 0022	3F FFE2	2	Reserved
NMI	00 0024	3F FFE4	3	Nonmaskable interrupt
ILLEGAL	00 0026	3F FFE6		Illegal-instruction trap
USER1	00 0028	3F FFE8		User-defined software interrupt
USER2	00 002A	3F FFEA		User defined software interrupt
USER3	00 002C	3F FFEC		User-defined software interrupt
USER4	00 002E	3F FFEE		User-defined software interrupt
USER5	00 0030	3F FFF0		User-defined software interrupt
USER6	00 0032	3F FFF2		User-defined software interrupt
USER7	00 0034	3F FFF4		User-defined software interrupt
USER8	00 0036	3F FFF6		User-defined software interrupt
USER9	00 0038	3F FFF8		User-defined software interrupt
USER10	00 003A	3F FFFA		User-defined software interrupt
USER11	00 003C	3F FFFC		User-defined software interrupt
USER12	00 003E	3F FFFE		User-defined software interrupt

[1] For C28x catalog devices, VMAP = 1 at reset.

[2] Interrupts DLOGINT and RTOSINT are generated by the emulation logic internal to the CPU.

The vector table can be mapped to the top or bottom of program space, depending on the value of the vector map bit (VMAP) in status register ST1. If the VMAP bit is 0, the vectors are mapped beginning at address 00 000016. If the VMAP bit is 1, the vectors are mapped beginning at address 3F FFC016. Table 3-1 lists the absolute addresses for VMAP = 0 and VMAP = 1.

The VMAP bit can be set by the SETC VMAP instruction and cleared by the CLRC VMAP instruction. The reset value of VMAP is 1.

3.3 Maskable Interrupts: INT1–INT14, DLOGINT, and RTOSINT

INT1–INT14 are 14 general-purpose interrupts. DLOGINT (the data log interrupt) and RTOSINT (the real-time operating system interrupt) are available for emulation purposes. These interrupts are supported by three dedicated registers: the CPU interrupt flag register (IFR), the CPU interrupt enable register (IER), and the CPU debug interrupt enable register (DBGIER).

The 16-bit IFR contains flag bits that indicate which of the corresponding interrupts are pending (waiting for approval from the CPU). The external input lines INT1–INT14 are sampled at every CPU clock cycle. If an interrupt signal is recognized, the corresponding bit in the IFR is set and latched. For DLOGINT or RTOSINT, a signal sent by the CPU on-chip analysis logic causes the corresponding flag bit to be set and latched. You can set one or more of the IFR bits at the same time by using the OR IFR instruction. More details about the IFR are given in Section 3.3.1. The on-chip analysis resources are introduced in Chapter 7.

The interrupt enable register (IER) and the debug interrupt enable register (DBGIER) each contain bits for individually enabling or disabling the maskable interrupts. To enable one of the interrupts in the IER, you set the corresponding bit in the IER; to enable the same interrupt in the DBGIER, you set the corresponding bit in the DBGIER. The DBGIER indicates which interrupts can be serviced when the CPU is in the real-time emulation mode. The IER and the DBGIER are discussed more in Section 3.3.2. Real-time mode is discussed in Section 7.4.2.

The maskable interrupts also share bit 0 in status register ST1. This bit, the interrupt global mask bit (INTM), is used to globally enable or globally disable these interrupts. When INTM = 0, these interrupts are globally enabled. When INTM = 1, these interrupts are globally disabled. You can set and clear INTM with the SETC INTM and CLRC INTM instructions, respectively. ST1 is described in Section 2.4.

After a flag has been latched in the IFR, the corresponding interrupt is not serviced until it is appropriately enabled by two of the following: the IER, the DBGIER, and the INTM bit. As shown in Table 3-2, the requirements for enabling the maskable interrupts depend on the interrupt-handling process used. In the standard process, which occurs in most circumstances, the DBGIER is ignored. When the C28x is in real-time emulation mode and the CPU is halted, a different process is used. In this special case, the DBGIER is used and the INTM bit is ignored. (If the DSP is in real-time mode and the CPU is running, the standard interrupt-handling process applies.)

Once an interrupt has been requested and properly enabled, the CPU prepares for and then executes the corresponding interrupt service routine. For a detailed description of this process, see Section 3.4.

Table 3-2. Requirements for Enabling a Maskable Interrupt

Interrupt-Handling Process	Interrupt Enabled If ...
Standard	INTM = 0 and bit in IER is 1
DSP in real-time mode and CPU halted	Bit in IER is 1 and bit in DBGIER is 1

As an example of varying interrupt-enable requirements, suppose you want interrupt INT5 enabled. This corresponds to bit 4 in the IER and bit 4 in the DBGIER. Usually, INT5 is enabled if INTM = 0 and IER(4) = 1. In real-time emulation mode with the CPU halted, INT5 is enabled if IER(4) = 1 and DBGIER(4) = 1.

3.3.1 CPU Interrupt Flag Register (IFR)

Figure 3-1 shows the IFR. If a maskable interrupt is pending (waiting for approval from the CPU), the corresponding IFR bit is 1; otherwise, the IFR bit is 0. To identify pending interrupts, use the PUSH IFR instruction and then test the value on the stack. Use the OR IFR instruction to set IFR bits, and use the AND IFR instruction to clear pending interrupts. When a hardware interrupt is serviced, or when an INTR instruction is executed, the corresponding IFR bit is cleared. All pending interrupts are cleared by the AND IFR, #0 instruction or by a hardware reset.

NOTE: When an interrupt is requested by the TRAP instruction, if the corresponding IFR bit is set, the CPU does not clear it automatically. If an application requires that the IFR bit be cleared, the bit must be cleared in the interrupt service routine.

15	14	13	12	11	10	9	8
RTOSINT	DLOGINT	INT14	INT13	INT12	INT11	INT10	INT9
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
INT8	INT7	INT6	INT5	INT4	INT3	INT2	INT1
R/W−0	R/W−0	R/W−0	R/W−0	R/W−0	R/W−0	R/W−0	R/W−0

Note: R = Read access; W = Write access; value following dash (−) is value after reset.

Figure 3-1. Interrupt Flag Register (IFR)

Bits 15 and 14 of the IFR correspond to the interrupts RTOSINT and DLOGINT:

Table 3-3. RTOSINT Real-time Operating System Interrupt Flag

RTOSINT	**Real-time operating system interrupt flag**	
Bit 15	RTOSINT = 0	RTOSINT is not pending.
	RTOSINT = 1	RTOSINT is pending.
DLOGINT	**Data log interrupt flag**	
Bit 14	DLOGINT = 0	DLOGINT is not pending.
	DLOGINT = 1	DLOGINT is pending.

For bits INT1-INT14, the following general description applies:

INTx	**Interrupt x flag (x = 1, 2, 3, ..., or 14)**	
Bit (x-1)	INTx = 0	INTx is not pending.
	INTx = 1	INTx is pending.

3.3.2 CPU Interrupt Enable Register (IER) and CPU Debug Interrupt Enable Register (DBGIER)

Figure 3-2 shows the IER. To enable an interrupt, set its corresponding bit to 1. To disable an interrupt, clear its corresponding bit to 0. Two syntaxes of the MOV instruction allow you to read from the IER and write to the IER. In addition, the OR IER instruction enables you to set IER bits, and the AND IER instruction enables you to clear IER bits. When a hardware interrupt is serviced, or when an INTR instruction is executed, the corresponding IER bit is cleared. At reset, all the IER bits are cleared to 0, disabling all the corresponding interrupts.

NOTE: When an interrupt is requested by the TRAP instruction, if the corresponding IER bit is set, the CPU does not clear it automatically. If an application requires that the IER bit be cleared, the bit must be cleared in the interrupt service routine.

NOTE: If an IFR b it is set in the same cycle that the corresponding IER bit is cleared, the interrupt will not be serviced until the IER bit is set again.

15	14	13	12	11	10	9	8
RTOSINT	DLOGINT	INT14	INT13	INT12	INT11	INT10	INT9
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
INT8	INT7	INT6	INT5	INT4	INT3	INT2	INT1
R/W−0	R/W−0	R/W−0	R/W−0	R/W−0	R/W−0	R/W−0	R/W−0

Note: R = Read access; W = Write access; value following dash (−) is value after reset.

Figure 3-2. Interrupt Enable Register (IER)

NOTE: When using the AND IER and OR IER instructions, make sure that they do not modify the state of bit 15 (RTOSINT) unless a real-time operating system is present.

Bits 15 and 14 of the IER enable or disable the interrupts RTOSINT and

DLOGINT:

RTOSINT	**Real-time operating system interrupt enable bit**	
Bit 15	RTOSINT = 0	RTOSINT is disabled.
	RTOSINT = 1	RTOSINT is enabled.
DLOGINT	**Data log interrupt enable bit**	
Bit 14	DLOGINT = 0	DLOGINT is disabled.
	DLOGINT = 1	DLOGINT is enabled.

For bits INT1-INT14, the following general description applies:

INTx	**Interrupt x enable bit (x = 1, 2, 3, ..., or 14)**	
Bit (x-1)	INTx = 0	INTx is disabled.
	INTx = 1	INTx is enabled.

Figure 3-3 shows the DBGIER, which is used only when the CPU is halted in real-time emulation mode. An interrupt enabled in the DBGIER is defined as a *time-critical interrupt*. When the CPU is halted in real-time mode, the only interrupts that are serviced are time-critical interrupts that are also enabled in the IER. If the CPU is running in real-time emulation mode, the standard interrupt-handling process is used and the DBGIER is ignored.

As with the IER, you can read the DBGIER to identify enabled or disabled interrupts and write to the DBGIER to enable or disable interrupts. To enable an interrupt, set its corresponding bit to 1. To disable an interrupt, set its corresponding bit to 0. Use the PUSH DBGIER instruction to read from the DBGIER and the POP DBGIER instruction to write to the DBGIER. At reset, all the DBGIER bits are set to 0.

15	14	13	12	11	10	9	8
RTOSINT	DLOGINT	INT14	INT13	INT12	INT11	INT10	INT9
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
INT8	INT7	INT6	INT5	INT4	INT3	INT2	INT1
R/W−0	R/W−0	R/W−0	R/W−0	R/W−0	R/W−0	R/W−0	R/W−0

Note: R = Read access; W = Write access; value following dash (−) is value after reset.

Figure 3-3. Debug Interrupt Enable Register (DBGIER)

Bits 15 and 14 of the DBGIER enable or disable the interrupts RTOSINT and DLOGINT:

RTOSINT	**Real-time operating system interrupt debug enable bit**	
Bit 15	RTOSINT = 0	RTOSINT is disabled.
	RTOSINT = 1	RTOSINT is enabled.
DLOGINT	**Data log interrupt debug enable bit**	
Bit 14	DLOGINT = 0	DLOGINT is disabled.
	DLOGINT = 1	DLOGINT is enabled.

For bits INT1-INT14, the following general description applies:

INTx	**Interrupt x debug enable bit (x = 1, 2, 3, ..., or 14)**	
Bit (x-1)	INTx = 0	INTx is disabled.
	INTx = 1	INTx is enabled.

3.4 Standard Operation for Maskable Interrupts

The flow chart in Figure 3-4 shows the standard process for handling interrupts. Section 7.4.2 contains information on handling interrupts when the DSP is in real-time mode and the CPU is halted. When more than one interrupt is requested at the same time, the C28x services them one after another according to their set priority ranking. See the priorities in Table 3-1.

Figure 3-4 is not meant to be an exact representation of how an interrupt is handled. It is a conceptual model of the important events.

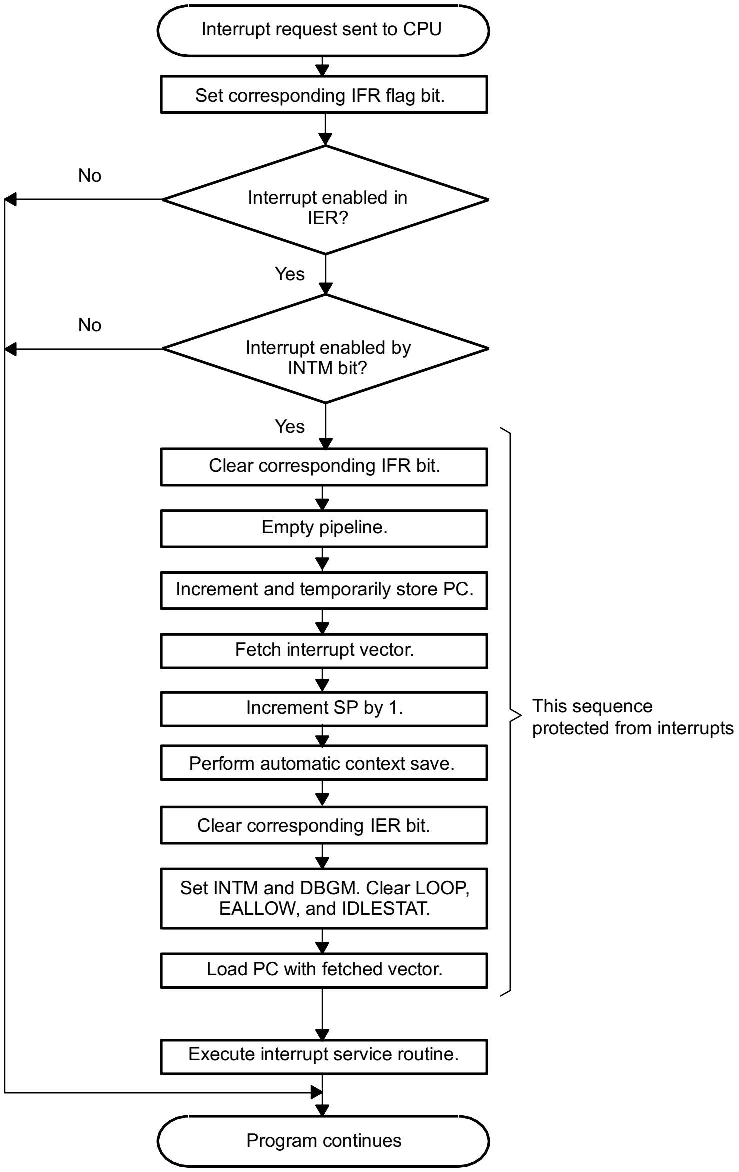

Interrupt request sent to CPU
Set corresponding IFR flag bit.
No
Interrupt enabled in IER?
Yes
No
Interrupt enabled by INTM bit?
Yes
Clear corresponding IFR bit.
Empty pipeline.
Increment and temporarily store PC.
Fetch interrupt vector.
Increment SP by 1.
Perform automatic context save.
Clear corresponding IER bit.
Set INTM and DBGM. Clear LOOP, EALLOW, and IDLESTAT.
Load PC with fetched vector.
This sequence protected from interrupts
Execute interrupt service routine.
Program continues

Figure 3-4. Standard Operation for CPU Maskable Interrupts

What following list explains the steps shown in Table 3-5:

1. **Interrupt request sent to CPU.** One of the following events occurs:
   - One of the pins INT1-INT14 is driven low by an external event, peripheral or PIE interrupt request..
   - The CPU emulation logic sends to the CPU a signal for DLOGINT or RTOSINT.
   - One of the interrupts INT1-INT14, DLOGINT, and RTOSINT is initiated by way of the OR IFR instruction.
2. **Set corresponding IFR flag bit.** When the CPU detects a valid interrupt in step 1, it sets and latches the corresponding flag in the interrupt flag register (IFR). This flag stays latched even if the interrupt is not approved by the CPU in step 3. The IFR is explained in detail in Section 3.3.1.

3. **Is the interrupt enabled in IER? Is the interrupt enabled by INTM bit?** The CPU approves the interrupt only if the following conditions are true:
   - The corresponding bit in the IER is 1.
   - The INTM bit in ST1 is 0. Once an interrupt has been enabled and then approved by the CPU, no other interrupts can be serviced until the CPU has begun executing the interrupt service routine for the approved interrupt (step 13). The IER is described in Section 3.3.2. ST1 is described in Section 2.4.
4. **Clear corresponding IFR bit.** Immediately after the interrupt is approved, its IFR bit is cleared. If the interrupt signal is kept low, the IFR register bit will be set again. However, the interrupt is not immediately serviced again. The CPU blocks new hardware interrupts until the interrupt service routine (ISR) begins. In addition, the IER bit is cleared (in step 10) before the ISR begins; therefore, an interrupt from the same source cannot disturb the ISR until the IER bit is set again by the ISR.
5. **Empty the pipeline.** The CPU completes any instructions that have reached or passed their decode 2 phase in the instruction pipeline. Any instructions that have not reached this phase are flushed from the pipeline.
6. **Increment and temporarily store PC.** The PC is incremented by 1 or 2, depending on the size of the current instruction. The result is the *return address*, which is temporarily saved in an internal hold register. During the automatic context save (step 9), the return address is pushed onto the stack.
7. **Fetch interrupt vector.** The PC is filled with the address of the appropriate interrupt vector, and the vector is fetched from that location. To determine which vector address has been assigned to each of the interrupts, see Section 3.2, *Interrupt Vectors* or, if your device uses a PIE module, see the System and Interrupts Reference Guide for your specific device.
8. **Increment SP by 1.** The stack pointer (SP) is incremented by 1 in preparation for the automatic context save (step 9). During the automatic context save, the CPU performs 32-bit accesses, and the CPU expects 32-bit accesses to be aligned to even addresses by the memory wrapper. Incrementing SP by 1 ensures that the first 32-bit access does not overwrite the previous stack value.
9. **Perform automatic context save.** A number of register values are saved automatically to the stack. These registers are saved in pairs; each pair is saved in a single 32-bit operation. At the end of each 32-bit save operation, the SP is incremented by 2. Table 3-4 shows the register pairs and the order in which they are saved. The CPU expects all 32-bit saves to be even-word aligned by the memory wrapper. As shown in the table, the SP is not affected by this alignment.

Table 3-4. Register Pairs Saved and SP Positions for Context Saves

Save Operation [1]	Register Pairs	Bit 0 of Storage Address SP Starts at Odd Address	SP Starts at Even Address
		1 SP position before step 8	1
1st	STO	0	0 SP position before step 8
	T	1	1
2nd	AL	0	0
	AH	1	1
3rd	PL [2]	0	0
	PH	1	1
4th	AR0	0	0
	AR1	1	1
5th	ST1	0	1
	DP	1	1
6th	IER	0	0
	DBGSTAT [3]	1	1
7th	Return address (low half)	0	0
	Return address (high half)	1	1
		0 SP position after save	0
		1	1 SP position after save

[1] All registers are saved as pairs, as shown.

[2] The P register is saved with 0 shift (CPU ignores current state of the product shift mode bits, PM, in status register 0).

[3] The DBGSTAT register contains special emulation information.

1. **Clear corresponding IER bit.** After the IER register is saved on the stack in step 9, the CPU clears the IER bit that corresponds to the interrupt being handled. This prevents reentry into the same interrupt. If you want to nest occurrences of the interrupt, have the ISR set that IER bit again.
2. **Set INTM and DBGM. Clear LOOP, EALLOW, and IDLESTAT.** All these bits are in status register ST1. By setting INTM to 1, the CPU prevents maskable interrupts from disturbing the ISR. If you wish to nest interrupts, have the ISR clear the INTM bit. By setting DBGM to 1, the CPU prevents debug events from disturbing time-critical code in the ISR. If you do not want debug events blocked, have the ISR clear DBGM. The CPU clears LOOP, EALLOW, and IDLESTAT so that the ISR operates within a new context.
3. **Load PC with fetched vector.** The PC is loaded with the interrupt vector that was fetched in step 7. The vector forces program control to the ISR.
4. **Execute interrupt service routine.** Here is where the CPU executes the program code you have prepared to handle the interrupt. A typical ISR is shown in Example 3-1.
   Although a number of register values are saved automatically in step 10, if the ISR uses other registers, you may need to save the contents of these registers at the beginning of the ISR. These values must then be restored before the return from the ISR. The ISR in Example 3-1 saves and restores auxiliary registers AR1H:AR0H, XAR2-XAR7, and the temporary register XT.
   If you want the ISR to inform a peripheral that the interrupt is being serviced, you can use the IACK instruction to send an interrupt acknowledge signal. The IACK instruction accepts a 16-bit constant as an operand. For a detailed description of the IACK instruction, see Chapter 6, *C28x Assembly Language Instructions*.
5. **Program continues.** If the interrupt is not approved by the CPU, the interrupt is ignored, and the program continues uninterrupted. If the interrupt is approved, its interrupt service routine is executed and the program continues where it left off (at the return address).

Example 3-1. Typical ISR

```
C28x Full Context Save/Restore

INTX:  ;                   8 cycles

       PUSH                AR1H:AR0H     ; 32-bit
       PUSH                XAR2          ; 32-bit
       PUSH                XAR3          ; 32-bit
       PUSH                XAR4          ; 32-bit
       PUSH                XAR5          ; 32-bit
       PUSH                XAR6          ; 32-bit
       PUSH                XAR7          ; 32-bit
       PUSH                XT            ; 32-bit

       ; +8 = 16 cycles
       .
       .
       POP                 XT
       POP                 XAR7

       POP                 XAR6
       POP                 XAR5
       POP                 XAR4
       POP                 XAR3
       POP                 XAR2
       POP                 XAR1H:AR0H
       IRET

       ; 16 cycles
```

3.5 Nonmaskable Interrupts

Nonmaskable interrupts cannot be blocked by any of the enable bits (the INTM bit, the DBGM bit, and enable bits in the IFR, IER, or DBGIER). The C28x immediately approves this type of interrupt and branches to the corresponding interrupt service routine. There is one exception to this rule: When the CPU is halted in stop mode (an emulation mode), no interrupts are serviced. Stop mode is described in Section 7.4.1.

The C28x nonmaskable interrupts include:

- Software interrupts (the INTR and TRAP instructions).
- Hardware interrupt NMI
- Illegal-instruction trap
- Hardware reset interrupt (RS)

The software interrupt instructions and NMI are described in this section. The illegal-instruction trap and reset are described in Section 3.6 and Section 3.7, respectively.

3.5.1 INTR Instruction

You can use the INTR instruction to initiate one of the following interrupts by name: INT1-INT14, DLOGINT, RTOSINT and NMI. For example, you can execute the interrupt service routine for INT1 by using the following instruction:

```
INTR INT1
```

Once an interrupt is initiated by the INTR instruction, how it is handled depends on which interrupt is specified:

- **INT1-INT14, DLOGINT, and RTOSINT.** These maskable interrupts have corresponding flag bits in the IFR. When a request for one of these interrupts is received at an external pin, the corresponding IFR bit is set and the interrupt must be enabled to be serviced. In contrast, when one of these interrupts is initiated by the INTR instruction, the IFR flag is not set, and the interrupt is serviced regardless of the value of any enable bits. However, in other respects, the INTR instruction and the hardware request are the same. For example, both clear the IFR bit that corresponds to the requested interrupt. For more details, see Section 3.4.
- **NMI.** Because this interrupt is nonmaskable, a hardware request at a pin and a software request with the INTR instruction lead to the same events. These events are identical to those that take place during a TRAP instruction (see Section 3.5.2).

Chapter 6, *C28x Assembly Language Instructions*, contains a detailed description of the INTR instruction.

3.5.2 TRAP Instruction

You can use the TRAP instruction to initiate any interrupt, including one of the user-defined software interrupts (see USER1-USER12 in Table 3-1. The TRAP instruction refers to one of the 32 interrupts by a number from 0 to 31. For example, you can execute the interrupt service routine for INT1 by using the following instruction: TRAP #1

Regardless of whether the interrupt has bits set in the IFR and IER, neither the IFR nor the IER is affected by this instruction. Figure 3-5 shows a functional flow chart for an interrupt initiated by the TRAP instruction. For more details about the TRAP instruction, see Chapter 6, *C28x Assembly Language Instructions*

NOTE: The TRAP #0 instruction does not initiate a full reset. It only forces execution of the interrupt service routine that corresponds to the RESET interrupt vector.

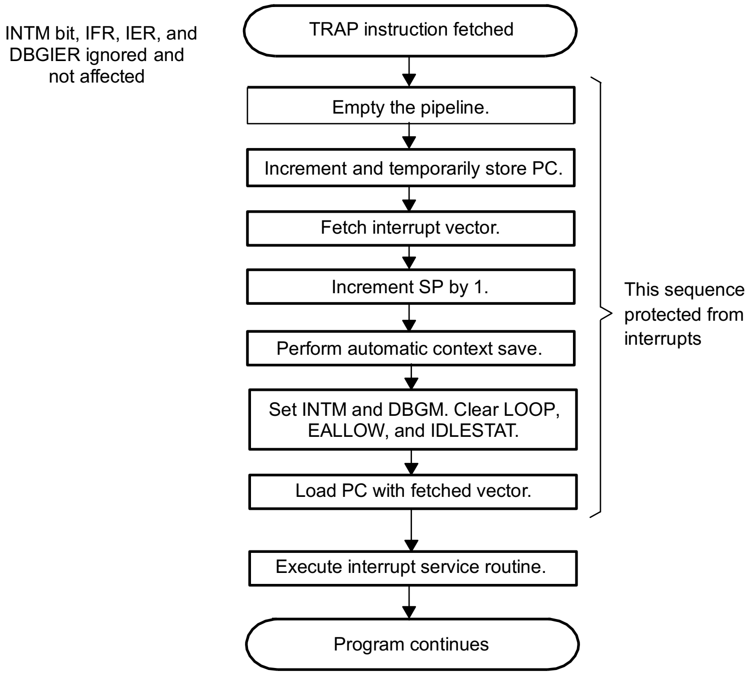

INTM bit, IFR, IER, and DBGIER ignored and not affected
TRAP instruction fetched
Empty the pipeline.
Increment and temporarily store PC.
Fetch interrupt vector.
Increment SP by 1.
Perform automatic context save.
Set INTM and DBGM. Clear LOOP, EALLOW, and IDLESTAT.
Load PC with fetched vector.
Execute interrupt service routine.
Program continues
This sequence protected from interrupts

Figure 3-5. Functional Flow Chart for an Interrupt Initiated by the TRAP Instruction

The following lists explains the steps shown in Figure 3-5:

1. **TRAP instruction fetched.** The CPU fetches the TRAP instruction from program memory. The desired interrupt vector has been specified as an operand and is now encoded in the instruction word. At this stage, no other interrupts can be serviced until the CPU begins executing the interrupt service routine (step 9).
2. **Empty the pipeline.** The CPU completes any instructions that have reached or passed the decode 2 phase of the pipeline. Any instructions that have not reached this phase are flushed from the pipeline.
3. **Increment and temporarily store PC.** The PC is incremented by 1. This value is the *return address*, which is temporarily saved in an internal hold register. During the automatic context save (step 6), the return address is pushed onto the stack.
4. **Fetch interrupt vector.** The PC is set to point to the appropriate vector location (based on the VMAP bit and the interrupt), and the vector located at the PC address is loaded into the PC. (To determine which vector address has been assigned to each of the interrupts, see Section 3.2, *Interrupt Vectors*.)
5. **Increment SP by 1.** The stack pointer (SP) is incremented by 1 in preparation for the automatic context save (step 6). During the automatic context save, the CPU performs 32-bit accesses, which are aligned to even addresses. Incrementing SP by 1 ensures that the first 32-bit access will not overwrite the previous stack value.
6. **Perform automatic context save.** A number of register values are saved automatically to the stack. These registers are saved in pairs; each pair is saved in a single 32-bit operation. At the end of each 32-bit operation, the SP is incremented by 2. Table 3-3 shows the register pairs and the order in which they are saved. All 32-bit saves are even-word aligned. As shown in the table, the SP is not affected by this alignment.

Table 3-5. Register Pairs Saved and SP Positions for Context Saves

Save Operation[(1)]	Register Pairs	Bit 0 of Storage Address SP Starts at Odd Address	SP Starts at Even Address
1st	ST0	0	0 ← SP position before step 5
	T	1	1
2nd	AL	0	0
	AH	1	1
3rd	PL [(2)]	0	0
	PH	1	1
4th	AR0	0	0
	AR1	1	1
5th	ST1	0	0
	DP	1	1
6th	IER	0	0
	DBGSTAT [(3)]	1	1
7th	Return address (low half)	0	0
	Return address (high half)	1	1
		0 SP position after save	0
		1	1 SP position after save

[(1)] All registers are saved as pairs, as shown.

[(2)] The P register is saved with 0 shift (CPU ignores current state of the product shift mode bits, PM, in status register 0).

[(3)] The DBGSTAT register contains special emulation information.

1. **Set INTM and DBGM. Clear LOOP, EALLOW, and IDLESTAT.** All these bits are in status register ST1 (described in Section 2.4). By setting INTM to 1, the CPU prevents maskable interrupts from disturbing the ISR. If you wish to nest interrupts, have the ISR clear the INTM bit. By setting DBGM to 1, the CPU prevents debug events from disturbing time critical code in the ISR. If you do not want debug events blocked, have the ISR clear DBGM.
   The CPU clears LOOP, EALLOW, and IDLESTAT so that the ISR operates within a new context.
2. **Load PC with fetched vector.** The PC is loaded with the interrupt vector that was fetched in step 4. The vector forces program control to the ISR.
3. **Execute interrupt service routine.** The CPU executes the program code you have prepared to handle the interrupt. You may wish to have the interrupt service routine (ISR) save register values in addition to those saved in step 6. A typical ISR is shown in Example 3-1.
   If you want the ISR to inform external hardware that the interrupt is being serviced, you can use the IACK instruction to send an interrupt acknowledge signal. The IACK instruction accepts a 16-bit constant as an operand and drives this 16-bit value on the 16 least significant lines of the data-write bus, DWDB(15:0). For a detailed description of the IACK instruction, see Chapter 6, *C28x Assembly Language Instructions* .
4. **Program continues.** After the interrupt service routine is completed, the program continues where it left off (at the return address).

3.5.3 Hardware Interrupt NMI

An interrupt can be requested by way the NMI input pin, which must be driven low to initiate the interrupt. Although NMI cannot be masked, there are some debug execution states in which NMI is not serviced (see Section 7.4, *Execution Control Modes*). For more details on real-time mode, see Section 7.4.2. Once a valid request is detected on the NMI pin, the CPU handles the interrupt in the same manner as shown for the TRAP instruction (see Section 3.5.2).

3.6 Illegal-Instruction Trap

Any one of the following events causes an illegal-instruction trap:

- An invalid instruction is decoded (this includes invalid addressing modes).
- The opcode value 000016 is decoded. This opcode corresponds to the ITRAP0 instruction.
- The opcode value FFFF16 is decoded. This opcode corresponds to the ITRAP1 instruction.
- A 32-bit operation attempts to use the [@SP] register addressing mode.
- Address mode setting AMODE=1 and PAGE0=1

An illegal-instruction trap cannot be blocked, not even during emulation. Once initiated, an illegal-instruction trap operates the same as a TRAP #19 instruction. The handling of an interrupt initiated by the TRAP instruction is described in Section 3.5.2. As part of its operation, the illegal-instruction trap saves the return address on the stack. Thus, you can detect the offending address by examining this saved value. For more information about the TRAP instruction, see Chapter 6, *C28x Assembly Language Instructions*.

3.7 Hardware Reset (RS)

When asserted, the reset input signal (RS) places the CPU into a known state. As part of a hardware reset, all current operations are aborted, the pipeline is flushed, and the CPU registers are reset as shown in Table 3-5. Then the RESET interrupt vector is fetched and the corresponding interrupt service routine is executed. For the reset condition of signals, see the data sheet for your particular C28x DSP. Also see the your data sheet for specific information on the process for resetting your DSP. Although RS cannot be masked, there are some debug execution states in which RS is not serviced (see Section 7.4, *Execution Control Modes*).

Table 3-6. Registers After Reset

Register	Bit(s)	Value After Reset	Comments
ACC	all	0000 0000_{16}	
XAR0	all	0000 0000_{16}	
XAR1	all	0000 0000_{16}	
XAR2	all	0000 0000_{16}	
XAR3	all	0000 0000_{16}	
XAR4	all	0000 0000_{16}	
XAR5	all	0000 0000_{16}	
XAR6	all	0000 0000_{16}	
XAR7	all	0000 0000_{16}	
DP	all	0000_{16}	DP points to data page 0.
IFR	16 bits	0000_{16}	There are no pending interrupts. All interrupts pending at the time of reset have been cleared.
IER	16 bits	0000_{16}	Maskable interrupts are disabled in the IER.
DBGIER	all	0000_{16}	Maskable interrupts are disabled in the DBGIER.
P	all	0000 0000_{16}	
PC	all	3F $FFC0_{16}$	PC is loaded with the reset interrupt vector at program-space address 00 0000_{16} or 3F $FFC0_{16}$.
RPC	all	0000_{16}	
SP	all	SP = 0x400	SP points to address 0400.
ST0	0: SXM	0	Sign extension is suppressed.
	1: OVM	0	Overflow mode is off.
	2: TC	0	
	3: C	0	
	4: Z	0	
	5: N	0	

Table 3-6. Registers After Reset (continued)

Register	Bit(s)	Value After Reset	Comments
	6: V	0	
	7-9: PM	000 $_2$	The product shift mode is set to left-shift-by-1.
	10-15: OVC	00 0000 $_2$	
ST1	0: INTM	1	Maskable interrupts are globally disabled. They cannot be serviced unless the C28x is in real-time mode with the CPU halted.
	1: DBGM	1	Emulation accesses and events are disabled.
	2: PAGE0	0	PAGE0 stack addressing mode is enabled. PAGE0 direct addressing mode is disabled.
	3: VMAP	1	The interrupt vectors are mapped to program-memory addresses 3F FFC0$_{16}$-3F FFFF$_{16}$.
	4: SPA	0	
	5: LOOP	0	
	6: EALLOW	0	Access to emulation registers is disabled.
	7: IDLESTAT	0	
	8: AMODE	0	C27x/C28x addressing mode
	9: OBJMODE	0	C27x object mode
	10: Reserved	0	
	11: M0M1MAP	1	
	12: XF	0	XFS output signal is low
	13-15: ARP	000 $_2$	ARP points to AR0.
XT	all	0000 0000 $_{32}$	

Chapter 4

SPRU430F–August 2001–Revised April 2015

Pipeline

This chapter explains the operation of the C28x instruction pipeline. The pipeline contains hardware that prevents reads and writes at the same register or data-memory location from happening out of order. However, you can increase the efficiency of your programs if you take into account the operation of the pipeline. In addition, you should be aware of two types of pipeline conflicts the pipeline does not protect against and how you can avoid them (see Section 4.4).

For more information about the instructions shown in examples throughout this chapter, see *C28x Assembly Language Instructions*.

Topic **Page**

4.1 Pipelining of Instructions

When executing a program, the C28x CPU performs these basic operations:

- Fetches instructions from program memory
- Decodes instructions
- Reads data values from memory or from CPU registers
- Executes instructions
- Writes results to memory or to CPU registers

For efficiency, the C28x performs these operations in eight independent phases. Reads from memory are designed to be pipelined in two stages, which correspond to the two pipeline phases used by the CPU for each memory-read operation. At any time, there can be up to eight instructions being carried out, each in a different phase of completion. Following are descriptions of the eight phases in the order they occur. The address and data buses mentioned in these descriptions are introduced in the *Memory Interface* chapter, Address and Data Buses section.

Fetch 1 (F1) In the fetch 1 (F1) phase, the CPU drives a program-memory ad- dress on the 22-bit program address bus, PAB(21:0).

Fetch 2 (F2) In the fetch 2 (F2) phase, the CPU reads from program memory by way of the program-read data bus, PRDB (31:0), and loads the instruction(s) into an instruction-fetch queue.

Decode 1 (D1) The C28x supports both 32-bit and 16-bit instructions and an instruction can be aligned to an even or odd address. The decode 1 (D1) hardware identifies instruction boundaries in the instruction-fetch queue and determines the size of the next instruction to be executed. It also determines whether the instruction is a legal instruction.

Decode 2 (D2) The decode 2 (D2) hardware requests an instruction from the instruction-fetch queue. The requested instruction is loaded into the instruction register, where decoding is completed. Once an instruction reaches the D2 phase, it runs to completion before any interrupts are taken. In this pipeline phase, the following tasks are performed:

- If data is to be read from memory, the CPU generates the source address or addresses.
- If data is to be written to memory, the CPU generates the destination address.
- The address register arithmetic unit (ARAU) performs any required modifications to the stack pointer (SP) or to an auxiliary register and/or the auxiliary register pointer (ARP).
- If a program-flow discontinuity (such as a branch or an illegal-instruction trap) is required, it is taken.

Read 1 (R1) If data is to be read from memory, the read 1 (R1) hardware drives the address(es) on the appropriate address bus(es).

Read 2 (R2) If data was addressed in the R1 phase, the read 2 (R2) hardware fetches that data by way of the appropriate data bus(es).

Execute (E) In the execute (E) phase, the CPU performs all multiplier, shifter, and ALU operations. This includes all the prime arithmetic and logic operations involving the accumulator and product register. For operations that involve reading a value, modifying it, and writing it back to the original location, the modification (typically an arithmetic or a logical operation) is performed during the E phase of the pipeline. Any CPU register values used by the multiplier, shifter, and ALU are read from the registers at the beginning of the E phase. A result that is to be written to a CPU register is written to the register at the end of the E phase.

Write (W) If a transferred value or result is to be written to memory, the write occurs in the write (W) phase. The CPU drives the destination address, the appropriate write strobes, and the data to be written. The actual storing, which takes at least one more clock cycle, is handled by memory wrappers or peripheral interface logic and is not visible as a part of the CPU pipeline.

Although every instruction passes through the eight phases, not every phase is active for a given instruction. Some instructions complete their operations in the decode 2 phase, others in the execute phase, and still others in the write phase. For example, instructions that do not read from memory perform no operations in the read phases, and instructions that do not write to memory perform no operation in the write phase.

Because different instructions perform modifications to memory and registers during different phases of their completion, an unprotected pipeline could lead to reads and writes at the same location happening out of the intended order. The CPU automatically adds inactive cycles to ensure that these reads and writes happen as intended. For more details about pipeline protection, see Section 4.3.

4.1.1 Decoupled Pipeline Segments

The fetch 1 through decode 1 (F1−D1) hardware acts independently of the decode 2 through write (D2−W) hardware. This allows the CPU to continue fetching instructions when the D2−W phases are halted. It also allows fetched instructions to continue through their D2−W phases when fetching of new instructions is delayed. Events that cause portions of the pipeline to halt are described in Section 4.2.2.

Instructions in their fetch 1, fetch 2, and decode 1 phases are discarded if an interrupt or other program-flow discontinuity occurs. An instruction that reaches its decode 2 phase always runs to completion before any program-flow discontinuity is taken.

4.1.2 Instruction-Fetch Mechanism

Certain branch instructions perform prefetching. The first few instructions of the branch destination will be fetched but not allowed to reach D2 until it is known whether the discontinuity will be taken. The instruction-fetch mechanism is the hardware for the F1 and F2 pipeline phases. During the F1 phase, the mechanism drives an address on the program address bus (PAB). During the F2 phase, it reads from the program-read data bus (PRDB). While an instruction is read from program memory in the F2 phase, the address for the next fetch is placed on the program address bus (during the next F1 phase).

The instruction-fetch mechanism contains an instruction-fetch queue of four 32-bit registers. During the F2 phase, the fetched instruction is added to the queue, which behaves like a first-in, first-out (FIFO) buffer. The first instruction in the queue is the first to be executed. The instruction-fetch mechanism performs 32-bit fetches until the queue is full. When a program-flow discontinuity (such as a branch or an interrupt) occurs, the queue is emptied. When the instruction at the bottom of the queue reaches its D2 phase, that instruction is passed to the instruction register for further decoding.

4.1.3 Address Counters FC, IC, and PC

Three program-address counters are involved in the fetching and execution of instructions:

- **Fetch counter (FC).** The fetch counter contains the address that is driven on the program address bus (PAB) in the F1 pipeline phase. The CPU continually increments the FC until the queue is full or the queue is emptied by a program-flow discontinuity. Generally, the FC holds an even address and is incremented by 2, to accommodate 32-bit fetches. The only exception to this is when the code after a discontinuity begins at an odd address. In this case, the FC holds the odd address. After performing a16-bit fetch at the odd address, the CPU increments the FC by 1 and resumes 32-bit fetching at even addresses
- **Instruction counter (IC).** After the D1 hardware determines the instruction size (16-bit or 32-bit), it fills the instruction counter (IC) with the address of the next instruction to undergo D2 decoding. On an interrupt or call operation, the IC value represents the return address, which is saved to the stack, to auxiliary register XAR7, or to RPC.
- **Program counter (PC).** When a new address is loaded into the IC, the previous IC value is loaded into the PC. The program counter (PC) always contains the address of the instruction that has reached its D2 phase.

Figure 4-1 shows the relationship between the pipeline and the address counters. Instruction 1 has reached its D2 phase (it has been passed to the instruction register). The PC points to the address from which instruction 1 was taken (00 0050$_{16}$). Instruction 2 has reached its D1 phase and will be executed next (assuming no program-flow discontinuity flushes the instruction-fetch queue). The IC points to the address from which instruction 2 was taken (00 0051$_{16}$). Instruction 3 is in its F2 phase. It has been transferred to the instruction-fetch queue but has not been decoded. Instructions 4 and 5 are each in their F1 phase. The FC address (00 0054$_{16}$) is being driven on the PAB. During the next 32-bit fetch, Instructions 4 and 5 will be transferred from addresses 00 0054$_{16}$ and 00 0055$_{16}$ to the queue.

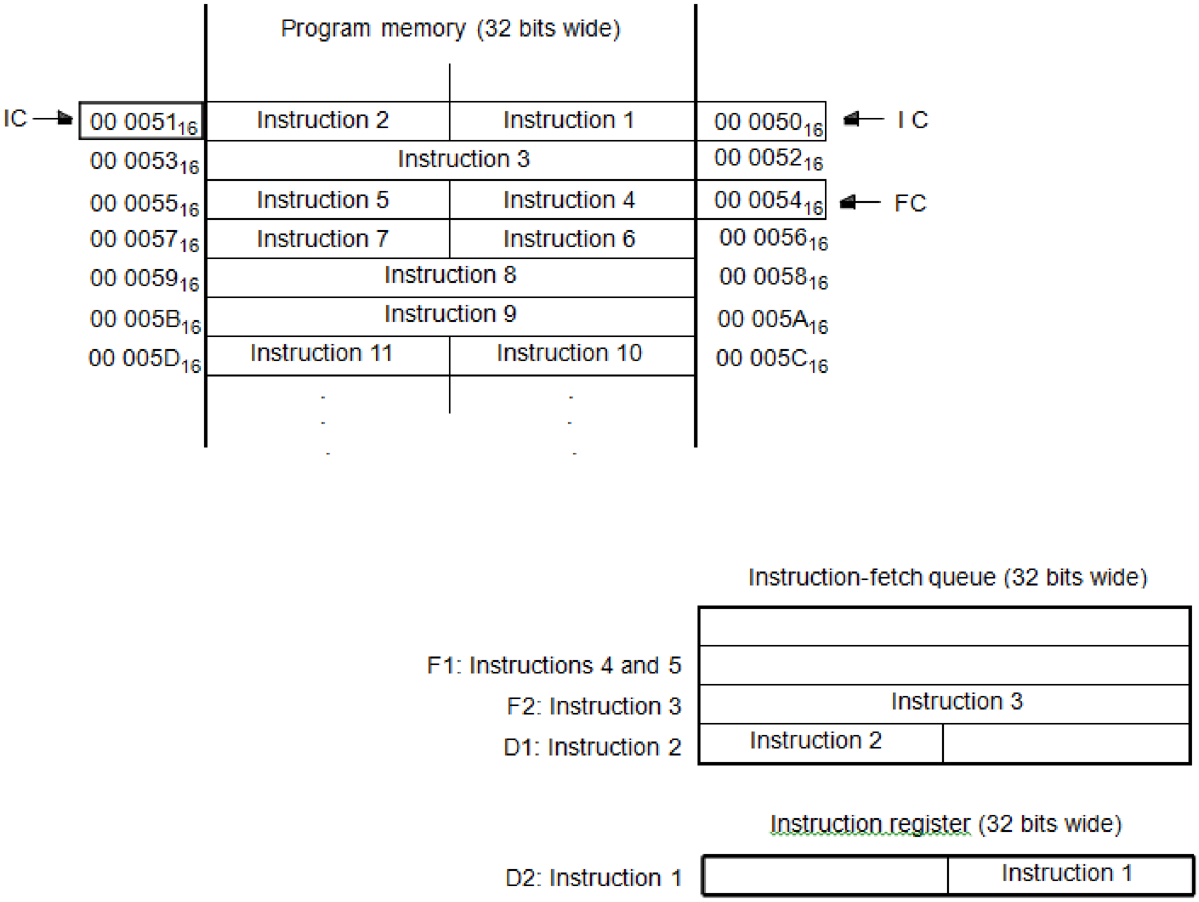


Figure 4-1. Relationship Between Pipeline and Address Counters FC, IC, and PC

The remainder of this document refers almost exclusively to the PC. The FC and the IC are visible in only limited ways. For example, when a call is executed or an interrupt is initiated, the IC value is saved to the stack or to auxiliary register XAR7.

4.2 Visualizing Pipeline Activity

Consider Example 4−2, which lists eight instructions, I1−I8, and shows a diagram of the pipeline activity for those instructions. The F1 column shows addresses and the F2 column shows the instruction opcodes read at those addresses. During an instruction fetch, 32 bits are read, 16 bits from the specified address and 16 bits from the following address. The D1 column shows instructions being isolated in the instruction-fetch queue, and the D2 column indicates address generation and modification of address registers. The Instruction column shows the instructions that have reached the D2 phase. The R1 column shows addresses, and the R2 column shows the data values being read from those addresses. In the E column, the diagram shows results being written to the low half of the accumulator (AL). In the W column, address and a data values are driven simultaneously on the appropriate memory buses. For example, in the last active W phase of the diagram, the address 00 020516 is driven on the data-write address bus (DWAB), and the data value 123416 is driven on the data-write data bus (DWDB).

The highlighted blocks in Section 4.2.1 indicate the path taken by the instruction ADD AL,*AR0++. That path can be summarized as follows:

Phase	Activity Shown
F1	Drive address 00 004216 on the program address bus (PAB).
F2	Read the opcodes F347 and F348 from addresses 00 0042$_{16}$ and 00 0043$_{16}$, respectively .
D1	Isolate F348 in the instruction-fetch queue.
D2	Use XAR0 = 0066$_{16}$ to generate source address 0000 0066$_{16}$ and then increment XAR0 to 0067$_{16}$.
R1	Drive address 00 0066$_{16}$ on the data-read data bus (DRDB).
R2	Read the data value 1 from address 0000 0066$_{16}$.
E	Add 1 to content of AL (1230$_{16}$) and store result (1231$_{16}$) to AL.
W	No activity

4.2.1 *Example 4-2: Diagraming Pipeline Activity*

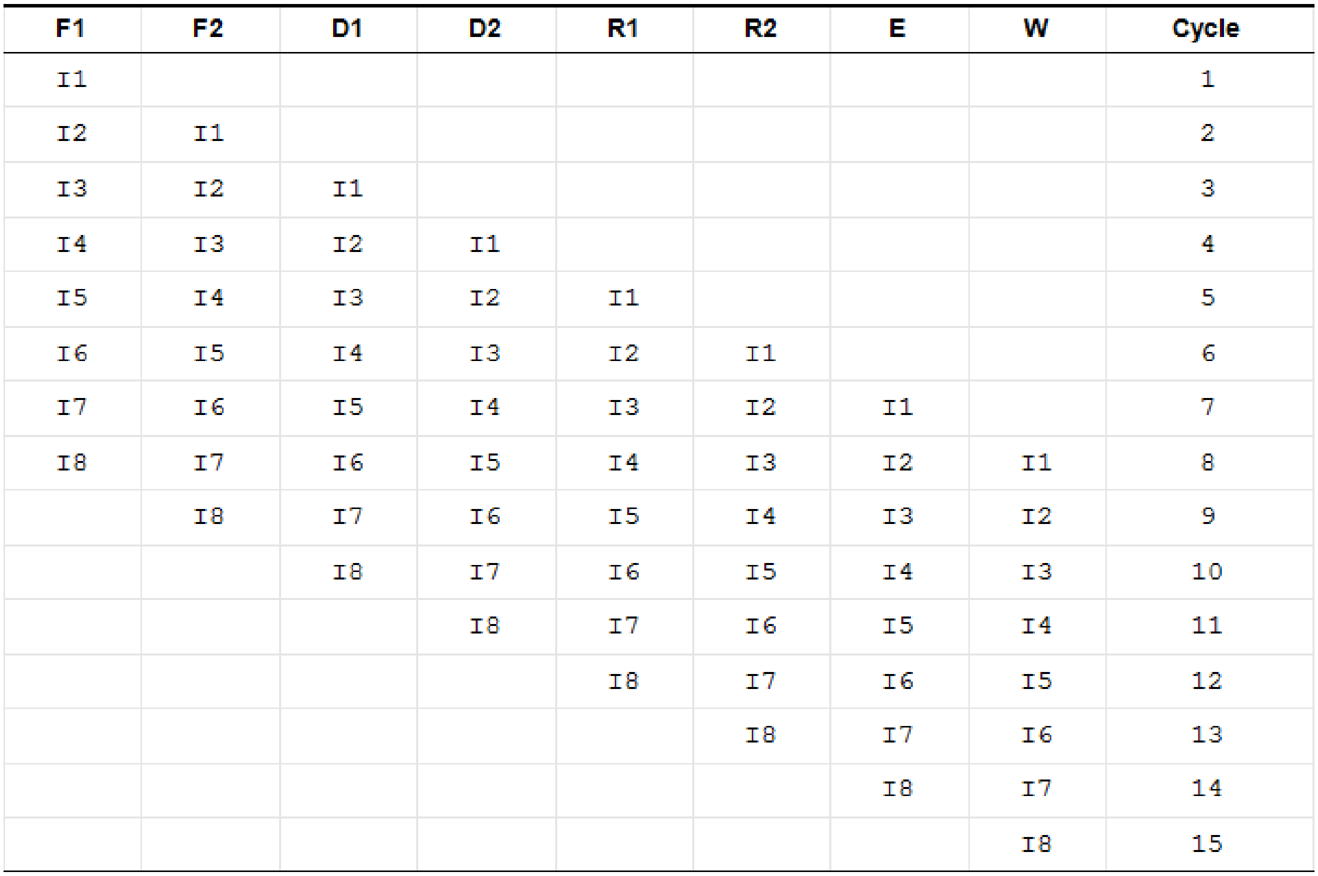
```
Address    Opcode  Instruction                                            Initial Values
00 0040    F345    I1: MOV DP,#VarA     ; DP = page that has VarA.         VarA address=00 0203
00 0041    F346    I2: MOV AL,@VarA     ; Move content of VarA to AL.      VarA=1230
00 0042    F347    I3: MOVB AR0,#VarB   ; AR0 points to VarB.              VarB address=00 0066
00 0043    F348    I4: ADD AL,*XAR0++   ; Add content of VarB to           VarB=0001

                                        ; AL, and add 1 to XAR0.           (VarB + 1)=0003
00 0044    F349    I5: MOV @VarC,AL     ; Replace content of VarC          (VarB + 2)=0005
                                        ; with content of AL.               VarC address=00 0204
00 0045    F34A    I6: ADD AL,*XAR0++   ; Add content of (VarB + 1)         VarD address=00 0205
                                        ; to AL, and add 1 to XAR0.
00 0046    F34B    I7: MOV @VarD,AL     ; Replace content of VarD
                                        ; with content of AL.
00 0047    F34C    I8: ADD AL,*XAR0      ; Add content of (VarB + 2)
                                         ; to AL.
```

F1	F2	D1	Instruction	D2	R1	R2	E	W
00 0040								
00 0042		F345						
	F348:F347	F346	I1: MOV DP,#VarA	DP = 8				
00 0044		F347	I2: MOV AL,@VarA	Generate VarA address	–			
	F34A: F349	F348	I3: MOVBXAR0,#VarB	XAR0 = 66	00 0203	–		
00 0046		F349	I4: ADD AL,*XAR0++	XAR0 = 67	–	1230	–	
	F34C:F34B	F34A	I5: MOV @VarC,AL	Generate VarC address	00 0066	–	AL = 1230	–
		F34B	I6: ADD AL,*XAR0++	XAR0 = 68	–	0001	–	–
		F34C	I7: MOV @VarD,AL	Generate VarD address	00 0067	–	AL = 1231	–
			I8: ADD AL,*XAR0	XAR0 = 68	–	0003	–	–
					00 0068	–	AL = 1234	00 0204 1231
						0005	–	–
							AL = 1239	00 0205 1234

NOTE: The opcodes shown in the F2 and D1 columns were chosen for illustrative purposes; they are not the actual opcodes of the instructions shown.

The pipeline activity in Section 4.2.1 can also be represented by the simplified diagram in Section 4.2.2. This type of diagram is useful if your focus is on the path of each instruction rather than on specific pipeline events. In cycle 8, the pipeline is full: there is an instruction in every pipeline phase. Also, the effective execution time for each of these instructions is one cycle. Some instructions finish their activity at the D2 phase, some at the E phase, and some at the W phase. However, if you choose one phase as a reference, you can see that each instruction is in that phase for one cycle.

4.2.2 Example 4-3 : Simplified Diagram of Pipeline Activity



F1	F2	D1	D2	R1	R2	E	W	Cycle
I1								1
I2	I1							2
I3	I2	I1						3
I4	I3	I2	I1					4
I5	I4	I3	I2	I1				5
I6	I5	I4	I3	I2	I1			6
I7	I6	I5	I4	I3	I2	I1		7
I8	I7	I6	I5	I4	I3	I2	I1	8
	I8	I7	I6	I5	I4	I3	I2	9
		I8	I7	I6	I5	I4	I3	10
			I8	I7	I6	I5	I4	11
				I8	I7	I6	I5	12
					I8	I7	I6	13
						I8	I7	14
							I8	15

4.3 Freezes in Pipeline Activity

This section describes the two causes for freezes in pipeline activity:

- Wait states
- An instruction-not-available condition

4.3.1 Wait States

When the CPU requests a read from or write to a memory device or peripheral device, that device may take more time to finish the data transfer than the CPU allots by default. Each device must use one of the CPU ready signals to insert wait states into the data transfer when it needs more time. The CPU has three independent sets of ready signals: one set for reads from and writes to program space, a second set for reads from data space, and a third set for writes to data space. Wait-state requests freeze a portion of the pipeline if they are received during the F1, R1, or W phase of an instruction:

- **Wait states in the F1 phase.** The instruction-fetch mechanism halts until the wait states are completed. This halt effectively freezes activity for instructions in their F1, F2, and D1 phases. However, because the F1−D1 hardware and the D2−W hardware are decoupled, instructions that are in their D2−W phases continue to execute.
- **Wait states in the R1 phase.** All D2−W activities of the pipeline freeze. This is necessary because subsequent instructions can depend on the data-read taking place. Instruction fetching continues until the instruction-fetch queue is full or a wait-state request is received during an F1 phase.
- **Wait states in the W phase.** All D2−W activity in the pipeline freezes. This is necessary because subsequent instructions may depend on the write operation happening first. Instruction fetching continues until the instruction-fetch queue is full or a wait-state request is received during an F1 phase.

4.3.2 Instruction-Not-Available Condition

The D2 hardware requests an instruction from the instruction-fetch queue. If a new instruction has been fetched and has completed its D1 phase, the instruction is loaded into the instruction register for more decoding. However, if a new instruction is not waiting in the queue, an instruction-not-available condition exists. Activity in the F1−D1 hardware continues. However, the activity in the D2−W hardware ceases until a new instruction is available.

One time that an instruction-not-available condition will occur is when the first instruction after a discontinuity is at an odd address and has 32 bits. A *discontinuity* is a break in sequential program flow, generally caused by a branch, a call, a return, or an interrupt. When a discontinuity occurs, the instruction-fetch queue is emptied, and the CPU branches to a specified address. If the specified address is an odd address, a 16-bit fetch is performed at the odd address, followed by 32-bit fetches at subsequent even addresses. Thus, if the first instruction after a discontinuity is at an odd address and has 32 bits, two fetches are required to get the entire instruction. The D2−W hardware ceases until the instruction is ready to enter the D2 phase.

To avoid the delay where possible, you can begin each block of code with one or two (preferably two) 16-bit instructions:

```
FunctionA:
     16-bit instruction    ; First instruction
     16-bit instruction    ; Second instruction
     32-bit instruction    ; 32-bit instructions can start here
     .
     .
     .
```

If you choose to use a 32-bit instruction as the first instruction of a function or subroutine, you can prevent a pipeline delay only by making sure the instruction begins at an even address.

4.4 Pipeline Protection

Instructions are being executed in parallel in the pipeline, and different instructions perform modifications to memory and registers 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 C28x pipeline has a mechanism that automatically protects against pipeline conflicts. There are two types of pipeline conflicts that can occur on the C28x:

- Conflicts during reads and writes to the same data-space location
- Register conflicts

The pipeline prevents these conflicts by adding inactive cycles between instructions that would cause the conflicts. Sections 4.6.1 and Section 4.6.2 explain the circumstances under which these pipeline-protection cycles are added and tells how to avoid them, so that you can reduce the number of inactive cycles in your programs.

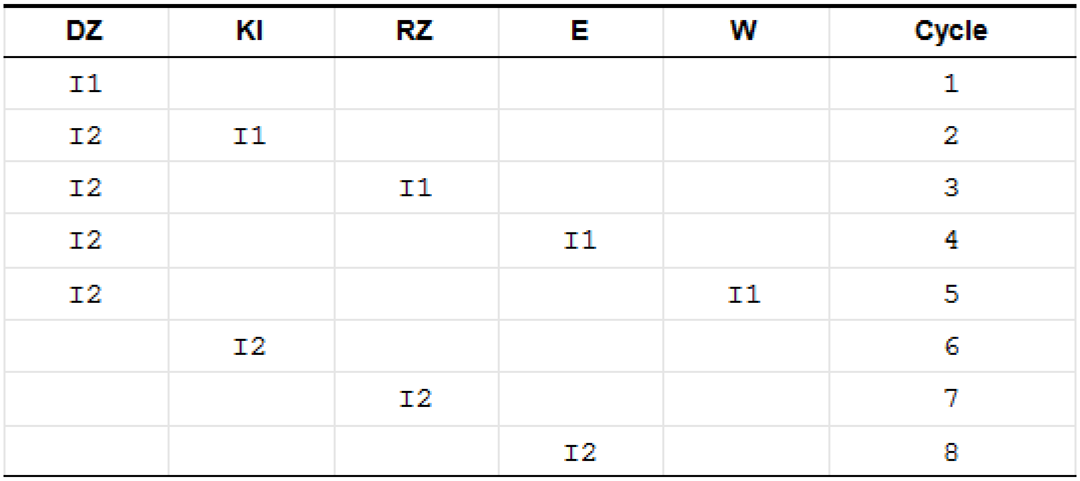
4.4.1 Protection During Reads and Writes to the Same Data-Space Location

Consider two instructions, A and B. Instruction A writes a value to a memory location during its W phase. Instruction B must read that value from the same location during its R1 and R2 phases. Because the instructions are being executed in parallel, it is possible that the R1 phase of instruction B could occur before the W phase of instruction A. Without pipeline protection, instruction B could read too early and fetch the wrong value. The C28x pipeline prevents that read by holding instruction B in its D2 phase until instruction A is finished writing.

Section 4.4.1.1 shows a conflict between two instructions that are accessing the same data-memory location. The pipeline activity shown is for an *unprotected* pipeline. For convenience, the F1−D1 phases are not shown. I1 writes to VarA during cycle 5. Data memory completes the store in cycle 6. I2 should not read the data-memory location any sooner than cycle 7. However, I2 performs the read during cycle 4 (three cycles too early). To prevent this kind of conflict, the pipeline-protection mechanism would hold I2 in the D2 phase for 3 cycles. During these *pipeline-protection* cycles, no new operations occur.

4.4.1.1 Example 4-4: Conflict Between a Read From and a Write to Same Memory Location

```
I1:    MOV @VarA,AL ; Write AL to data−memory location
I2:    MOV AH,@VarA ; Read same location, store value in AH
```



DZ	KI	RZ	E	W	Cycle
I1					1
I2	I1				2
I2		I1			3
I2			I1		4
I2				I1	5
	I2				6
		I2			7
			I2		8

You can reduce or eliminate these types of pipeline-protection cycles if you can take other instructions in your program and insert them between the instructions that conflict. Of course, the inserted instructions must not cause conflicts of their own or cause improper execution of the instructions that follow them. For example, the code in Section 4.4.1.1 could be improved by moving a CLRC instruction to the position between the MOV instructions (assume that the instructions following CLRC SXM operate correctly with SXM = 0):

```
I1:  MOV  @VarA,AL ; Write AL to data−memory location
     CLRC SXM       ; SXM = 0 (sign extension off)

I2:  MOV  AH,@VarA ; Read same location, store value in AH
```

Inserting the CLRC instruction between I1 and I2 reduces the number of pipeline-protection cycles to two. Inserting two more instructions would remove the need for pipeline protection. As a general rule, if a read operation occurs within three instructions from a write operation to the same memory location, the pipeline protection mechanism adds at least one inactive cycle.

4.4.2 Protection Against Register Conflicts

All reads from and writes to CPU registers occur in either the D2 phase or the E phase of an instruction. A register conflict arises when an instruction attempts to read and/or modify the content of a register (in the D2 phase) before a previous instruction has written to that register (in the E phase).

The pipeline-protection mechanism resolves register conflicts by holding the later instruction in its D2 phase for as many cycles as needed (one to three). You do not have to consider register conflicts unless you wish to achieve maximum pipeline efficiency. If you choose to reduce the number of pipeline-protection cycles, you can identify the pipeline phases in which registers are accessed and try to move conflicting instructions away from each other.

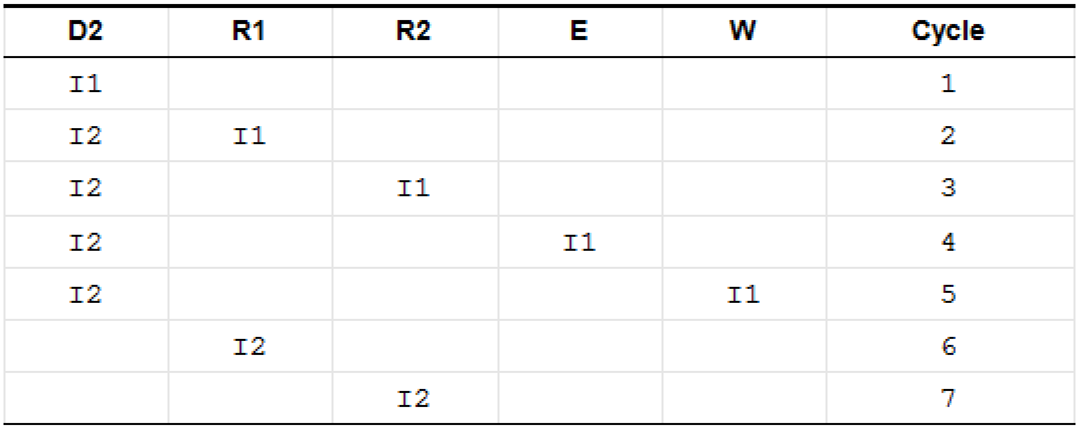
Generally, a register conflict involves one of the address registers:

- 16-bit auxiliary registers AR0−AR7
- 32-bit auxiliary registers XAR0−XAR7
- 16-bit data page pointer (DP)
- 16-bit stack pointer (SP)

Example 4−5 shows a register conflict involving auxiliary register XAR0. The pipeline activity shown is for an unprotected pipeline, and for convenience, the F1−D1 phases are not shown. I1 writes to XAR0 at the end of cycle 4. I2 should not attempt to read XAR0 until cycle 5. However, I2 reads XAR0 (to generate an address) during cycle 2. To prevent this conflict, the pipeline-protection mechanism would hold I2 in the D2 phase for three cycles. During these cycles, no new operations occur.

4.4.2.1 Example 4-5: Register Conflict

```
I1: MOVB AR0,@7    ; Load AR0 with the value addressed by
                   ; the operand @7 and clear the upper
                   ; half of XAR0.
I2: MOV AH,*XAR0   ; Load AH with the value pointed to by
                   ; XAR0.
```



D2	R1	R2	E	W	Cycle
I1					1
I2	I1				2
I2		I1			3
I2			I1		4
I2				I1	5
	I2				6
		I2			7

You can reduce or eliminate pipeline-protection cycles due to a register conflict by inserting other instructions between the instructions that cause the conflict. For example, the code in Section 4.4.2.1 could be improved by moving two other instructions from elsewhere in the program (assume that the instructions following SETC SXM operate correctly with PM = 1 and SXM = 1):

```
I1: MOVB AR0,@7    ; Load AR0 with the value addressed by
                   ; the operand @7 and clear the upper
                   ; half of XAR0.
    SPM 0          ; PM = 1 (no product shift)
    SETC SXM       ; SXM = 1 (sign extension on)

I2: MOV AH,*XAR0   ; Load AH with the value pointed to by
                   ; AR0.
```

Inserting the SPM and SETC instructions reduces the number of pipeline-protection cycles to one. Inserting one more instruction would remove the need for pipeline protection. As a general rule, if a read operation occurs within three instructions from a write operation to the same register, the pipeline-protection mechanism adds at least one inactive cycle.

4.4.3 Protection Against Interrupts

Instructions for enabling and disabling interrupts via IER and INTM always take effect before the next instruction is processed. These instructions take up multiple cycles in the pipeline to prevent any following instructions from reaching the D2 stage before IER and INTM are modified.

4.5 Avoiding Unprotected Operations

This section describes pipeline conflicts that the pipeline-protection mechanism does not protect against. These conflicts are avoidable, and this section offers suggestions for avoiding them.

4.5.1 Unprotected Program-Space Reads and Writes

The pipeline protects only register and data-space reads and writes. It does not protect the program-space reads done by the PREAD and MAC instructions or the program-space write done by the PWRITE instruction. Be careful with these instructions when using them to access a memory block that is shared by data space and program space.

As an example, suppose a memory location can be accessed at address 00 0D5016 in program space and address 0000 0D5016 in data space. Consider the following lines of code:

```
; XAR7 = 000D50 in program space
; Data1 = 000D50 in data space
ADD   @Data1,AH    ; Store AH to data−memory location
                   ; Data1.

PREAD @AR1,*XAR7   ; Load AR1 from program−memory
                   ; location given by XAR7.
```

The operands @Data1 and *XAR7 are referencing the same location, but the pipeline cannot interpret this fact. The PREAD instruction reads from the memory location (in the R2 phase) before the ADD writes to the memory location (in the W phase).

However, the PREAD is not necessary in this program. Because the location can be accessed by an instruction that reads from data space, you can use another instruction, such as a MOV instruction:

```
ADD    @Data1,AH    ;    Store AH to memory location Data1.
MOV    AR1,*XAR7    ;    Load AR1 from memory location
                    ;    given by    XAR7.
```

4.5.2 An Access to One Location That Affects Another Location

If an access to one location affects another location, you may need to correct your program to prevent a pipeline conflict. You only need to be concerned about this kind of pipeline conflict if you are addressing a location outside of a protected address range. (See Section 4.5.3). Consider the following example:

```
     MOV @DataA,#4      ; This write to DataA causes a
                        ; peripheral to clear bit 15 of DataB.

$10: TBIT @DataB,#15    ; Test bit 15 of DataB.
     SB    $10,NTC      ; Loop until bit 15 is set.
```

This program causes a misread. The TBIT instruction reads bit 15 (in the R2 phase) before the MOV instruction writes to bit 15 (in the W phase). If the TBIT instruction reads a 1, the code prematurely ends the loop. Because DataA and DataB reference different data-memory locations, the pipeline does not identify this conflict.

However, you can correct this type of error by inserting two or more NOP (no operation) instructions to allow for the delay between the write to DataA and the change to bit 15 of DataB. For example, if a 2-cycle delay is sufficient, you can fix the previous code as follows:

```
     MOV @DataA,#4       ; This write to DataA causes a
                         ; peripheral to clear bit 15 of DataB.

     NOP                 ; Delay by 1 cycle.
     NOP                 ; Delay by 1 cycle.
$10: TBIT @DataB,#15     ; Test bit 15 of DataB.
     SB    $10,NTC      ; Loop until bit 15 is set.
```

4.5.3 Write Followed By Read Protection Mode

The CPU contains a write followed by read protection mode to ensure that any read operation that follows a write operation within a protected address range is executed as written by delaying the read operation until the write is initiated.

See your device data sheet for device-specific information about which memory region is write-followed-by-read protected.

The PROTSTART(15:0) and PROTRANGE(15:0) input signals set the protection range. The PROTRANGE(15:0) value is a binary multiple with the smallest block size being 64 words, and the largest being 4M words (64 words, 128 words, 256 words ...1M words, 2M words, 4M words). The PROTSTART address must always be a multiple of the chosen range. For example, if a 4K block size is selected, then the start address must be a multiple of 4K.

The ENPROT signal enables this feature (when set high), it disables this feature (when set low)

All of the above signals are latched on every cycle. The above signals are connected to registers and can be changed within the application program.

The above mechanism only works for reads that follow writes to the protected area. Reads and write sequences to unprotected areas are not affected, as shown in the following examples.

```
Example 1: write protected_area
           write protected_area
           write protected_area
                                      <- pipe protection (3 cycles)

           read non_protected_area

Example 2: write protected_area
           write protected_area
           write protected_area
                                      <- no pipe protection invoked

           read non_protected_area
                                      <- pipe protection (2 cycles)
           read protected_area
           read protected_area

Example 3: write non_protected_area
           write non_protected_area
           write non_protected_area
                                      <- no pipe protection invoked
           read protectd_area
```

C28x Addressing Modes

This chapter describes the addressing modes of the C28x and provides examples.

Topic **Page**

5.1 Type of Addressing Modes

The C28x CPU supports four basic types of addressing modes:

- Direct Addressing Mode

  DP (data page pointer): In this mode, the 16-bit DP register behaves like a fixed page pointer. The instruction supplies a 6-bit or 7-bit offset field, which is concatenated with the value in the DP register. This type of addressing is useful for accessing fixed address data structures, such as peripheral registers and global or static variables in C/C++.
- Stack Addressing Mode

  SP (stack pointer): In this mode, the 16-bit SP pointer is used to access information on the software stack. The software stack grows from low to high memory on the C28x and the stack pointer always points to the next empty location. The instruction supplies a 6-bit offset field that is subtracted from the current stack pointer value for accessing data on the stack or the stack pointer can be post-incremented or pre-decremented when pushing and popping data from the stack, respectively.
- Indirect Addressing Mode

  XAR0 to XAR7 (auxiliary register pointers): In this mode, the 32-bit XARn registers behave as generic data pointers. The instruction can direct to post-increment, pre/post-decrement, or index from the current register contents with either a 3-bit immediate offset field or with the contents of another 16-bit register.
- Register Addressing Mode

  In this mode, another register can be the source or destination operand of an access. This enables register-to-register operations in the C28x architecture.

On most C28x instructions, an 8-bit field in the instruction op-code selects the addressing mode to use and what modification to make to that mode. In the C28x instruction set, this field is referred to as:

- loc16

  Selects Direct/Stack/Indirect/Register addressing mode for 16-bit data access.
- loc32

  Selects Direct/Stack/Indirect/Register addressing mode for 32-bit data access.

An example C28x instruction description, which uses the above, would be:

- ADD — AL,loc16

  Take the 16-bit contents of AL register, add the contents of 16-bit location specified by the ”loc16” field and store the contents in AL register.
- ADDL — loc32,ACC

  Take the 32-bit contents of the location pointed to by the ”loc32” field, add the contents of the 32-bit ACC register, and store the result back into the location specified by the ”loc32” field.

Other types of addressing modes supported are:

- Data/Program/IO Space Immediate Addressing Modes:

  In this mode, the address of the memory operand is embedded in the instruction.
- Program Space Indirect Addressing Modes:

  Some instructions can access a memory operand located in program space using an indirect pointer. Since memory is unified on the C28x CPU, this enables the reading of two operands in a single cycle.

Only a small number of instructions use the above modes and typically they are in combination with the ”loc16/loc32” modes.

The following sections contain detailed descriptions of the addressing modes with example instructions. For more information about the instructions shown in examples throughout this chapter, see Chapter 6, *Assembly Language Instructions*.

5.2 Addressing Modes Select Bit (AMODE)

To accommodate various types of addressing modes, an addressing mode bit (AMODE) selects the decoding of the 8-bit field (loc16/loc32). This bit is found in Status Register 1 (ST1). The addressing modes have been broadly classified as follows:

- AMODE = 0

  This is the default mode on reset and is the mode used by the C28x C/C++ compiler. This mode is not fully compatible to the C2xLP CPU addressing modes. The data page pointer offset is 6-bits (it is 7-bits on the C2xLP) and not all of the indirect addressing modes are supported.

- AMODE = 1

  This mode contains addressing modes that are fully compatible to the C2xLP device. The data page pointer offset is increased to 7-bits and all of the indirect addressing modes available on the C2xLP are supported.

The available addressing modes, for the ”loc16” or ”loc32” field, are summarized in Table 5-1.

Table 5-1. Addressing Modes for “loc16” or “loc32”

AMODE = 0		AMODE = 1	
8-Bit Decode	**”loc16/loc32” Syntax**	**8-Bit Decode**	**”loc16/loc32” Syntax**
Direct Addressing Modes (DP):			
0 0 III III	@6bit	0 I III III	@@7bit
Stack Addressing Modes (SP):			
0 1 III III 1 0 111 101 1 0 111 110	*-SP[6bit] *SP++ *--SP	1 0 111 101 1 0 111 110	*SP++ *--SP
C28x Indirect Addressing Modes (XAR0 to XAR7):			
1 0 000 AAA 1 0 001 AAA 1 0 010 AAA 1 0 011 AAA 1 1 III AAA	*XARn++ *--XARn *+XARn[AR0] *+XARn[AR1] *+XARn[3bit]	1 0 000 AAA 1 0 001 AAA 1 0 010 AAA 1 0 011 AAA	*XARn++ *--XARn *+XARn[AR0] *+XARn[AR1]
C2xLP Indirect Addressing Modes (ARP, XAR0 to XAR7):			
1 0 111 000 1 0 111 001 1 0 111 010 1 0 111 011 1 0 111 100 1 0 101 110 1 0 101 111 1 0 110 RRR	* *++ *-- *0++ *0-- *BR0++ *BR0-- *,ARPn	1 0 111 000 1 0 111 001 1 0 111 010 1 0 111 011 1 0 111 100 1 0 101 110 1 0 101 111 1 0 110 RRR 1 1 000 RRR 1 1 001 RRR 1 1 010 RRR 1 1 011 RRR 1 1 100 RRR 1 1 101 RRR	* *++ *-- *0++ *0-- *BR0++ *BR0-- *,ARPn *++,ARPn *--,ARPn *0++,ARPn *0--,ARPn *BR0++,ARPn *BR0--,ARPn
Circular Indirect Addressing Modes (XAR6, XAR1):			
1 0 111 111	*AR6%++	1 0 111 111	*+XAR6[AR1%++]
32-Bit Register Addressing Modes (XAR0 to XAR7, ACC, P, XT):			
1 0 100 AAA 1 0 101 001 1 0 101 011 1 0 101 100	@XARn @ACC @P @XT	1 0 100 AAA 1 0 101 001 1 0 101 011 1 0 101 100	@XARn @ACC @P @XT
16-Bit Register Addressing Modes (AR0 to AR7, AH, AL, PH, PL, TH, SP):			

Table 5-1. Addressing Modes for “loc16” or “loc32” (continued)

AMODE = 0		AMODE = 1	
8-Bit Decode	**”loc16/loc32” Syntax**	**8-Bit Decode**	**”loc16/loc32” Syntax**
`1 0 100 AAA`	`@ARn`	`1 0 100 AAA`	`@ARn`
`1 0 101 000`	`@AH`	`1 0 101 000`	`@AH`
`1 0 101 001`	`@AL`	`1 0 101 001`	`@AL`
`1 0 101 010`	`@PH`	`1 0 101 010`	`@PH`
`1 0 101 011`	`@PL`	`1 0 101 011`	`@PL`
`1 0 101 100`	`@TH`	`1 0 101 100`	`@TH`
`1 0 101 101`	`@SP`	`1 0 101 101`	`@SP`

In the ”C28x Indirect” addressing modes, the auxiliary register pointer used in the addressing mode is implicitly specified. In the ”C2xLP Indirect” addressing modes, a 3-bit pointer called the auxiliary register pointer (ARP) is used to s lect which of the auxiliary registers is currently used and which pointer is used in the next operation.

The examples below illustrate the differences between the ”C28x Indirect” and ”C2xLP Indirect” addressing modes:

- ADD — AL,*XAR4++

  Read the contents of 16-bit memory location pointed to by register XAR4, add the contents to AL register. Post-increment the contents of XAR4 by 1.

- ADD — AL,*++

  Assume ARP pointer in ST1 contains the value 4. Read the contents of 16-bit memory location pointed to by register XAR4, add the contents to AL register. Post-increment the contents of XAR4 by 1.

- ADD — AL,*++,ARP5

  Assume ARP pointer in ST1 contains the value 4. Read the contents of 16-bit memory location pointed to by register XAR4, add the contents to AL register. Post-increment the contents of XAR4 by 1. Set the ARP pointer to 5. Now it points to XAR5.

On the C28x instruction syntax, the destination operand is always on the left and the source operands are always on the right.

5.3 Assembler/Compiler Tracking of AMODE Bit

The compiler will always assume the addressing mode is set to AMODE = 0 and therefore will only use addressing modes that are valid for AMODE = 0. The assembler can be instructed, via the command line options, to default to either AMODE = 0 or AMODE = 1. The command line options are:

–v28	Assumes AMODE = 0 (C28x addressing modes).
−v28 −m20	Assumes AMODE = 1 (full C2xLP compatible addressing modes.

Additionally, the assembler allows directives to be embedded within a file to instruct the assembler to override the default mode and change syntax checking to the new address mode setting:

.c28_amode	Tells assembler that any code that follows assumes AMODE = 0 (C28x addressing modes).
.lp_amode	Tells assembler that any code that follows assumes AMODE = 1 (full C2xLP compatible addressing modes)

The above directives cannot be nested. The above directives can be used as follows within an assembly program:

```
; File assembled using "−v28" option (assume AMODE = 0):
   .           ; This section of code can only use AMODE = 0
               ; addressing modes

   .
   .
   .
   .
SETC AMODE    ; Change to AMODE = 1
.lp_amode     ; Tell assembler to check for AMODE = 1 syntax
   .          ; This section of code can only use AMODE = 1
              ; addressing modes
   .
   .
   .
   .

CLRC AMODE     ; Revert back to AMODE = 0
.c28_amode     ; Tell assembler to check for AMODE = 1 syntax
   .           ; This section of code can only use AMODE = 0
               ; addressing modes
   .
   .
   .
   .
; End of file.
```

5.4 Direct Addressing Modes (DP)

AMODE	”loc16/loc32” Syntax	Description
0	@6bit	32bitDataAddr(31:22) = 0 32bitDataAddr(21:6) = DP(15:0) 32bitDataAddr(5:0) = 6bit **Note:** The 6-bit offset value is concatenated with the 16-bit DP register. The offset value enables 0 to 63 words to be addressed relative to the current DP register value.

Example (s):

```
MOVW    DP,#VarA        ; Load DP pointer with page value containing VarA
ADD     AL,@VarA        ; Add memory location VarA to register AL
MOV     @VarB,AL        ; Store AL into memory location VarB
                        ; VarB is located in the same 64−word page as VarA
MOVW    DP,#VarC        ; Load DP pointer with page value containing VarC
SUB     AL,@VarC        ; Subtract memory location VarC from register AL
MOV     @VarD,AL        ; Store AL into memory location VarD
                        ; VarC is located in the same 64−word page as VarD
                        ; VarC & D are in different pages than VarA & B
```

AMODE	”loc16/loc32” Syntax	Description
0	@@7bit	32bitDataAddr(31:22) = 0 32bitDataAddr(21:7) = DP(15:1) 32bitDataAddr(6:0) = 7bit **Note:** The 7-bit offset value is concatenated with the 15-bit DP register. Bit 0 of DP register is ignored and is not affected by the operation. The offset value enables 0 to 127 words to be addressed relative to the current DP register value.

Example (s):

```
SETC AMODE              ; Make sure AMODE = 1
.lp_amode               ; Tell assembler that AMODE = 1
MOVW    DP,#VarA        ; Load DP pointer with page value containing VarA
ADD     AL,@@VarA       ; Add memory location VarA to register AL
MOV     @@VarB,AL       ; Store AL into memory location VarB
                        ; VarB is located in the same 128−word page as VarA
MOVW    DP,#VarC        ; Load DP pointer with page value containing VarC
SUB     AL,@@VarC       ; Subtract memory location VarC from register AL
MOV     @@VarD,AL       ; Store AL into memory location VarD
                        ; VarC is located in the same 128−word page as VarD
                        ; VarC & D are in different pages than VarA & B
```

Note: The direct addressing mode can access only the lower 4M of data address space on the C28x device.

5.5 Stack Addressing Modes (SP)

AMODE	”loc16/loc32” Syntax	Description

0	`*-SP [6bit]`	`32bitDataAddr(31:16) = 0x0000`
		`32bitDataAddr(15:0) = SP - 6bit`
		Note: The 6-bit offset value is subtracted from the current 16-bit SP register value. The offset value enables 0 to 63 words to be addressed relative to the current SP register value.

Example (s):

```
        ADD     AL,*-SP[5]      ; Add 16-bit contents from stack location
                                ; -5 words from top of stack to AL register
        MOV     *-SP[8],AL      ; Store 16-bit AL register to stack location
                                ; -8 words from top of stack
        ADDL    ACC,*-SP[12]    ; Add 32-bit contents from stack location
                                ; −12 words from top of stack to ACC register.
        MOVL    *-Sp[34],ACC    ; Store 32-bit ACC register to stack location
                                ; -34 words from top of stack
```

AMODE	”loc16/loc32” Syntax	Description

X	`*SP++`	`32bitDataAddr(31:16) = 0x0000`
		`32bitDataAddr(15:0) = SP`
		if(loc16), SP = SP + 1
		if(loc32), SP = SP + 2

Example (s):

```
        MOV     *SP++,AL        ; Push contents of 16-bit AL register onto top
                                ; of stack
        MOVL    *Sp++,P         ; Push contents of 32-bit P register onto top
                                ; of stack
```

AMODE	”loc16/loc32” Syntax	Description

X	`*--SP`	`if(loc16), SP = SP - 1`
		`if(loc32), SP = SP - 2`
		`32bitDataAddr(31:16) = 0x0000`
		`32bitDataAddr(15:0) = SP`

Example (s):

```
      ADD       AL, *--SP       ; Pop contents from top of stack and add to 16-bit
                                ; Al register
      MOVL      ACC, *--Sp      ; Pop contents from top of stack and store in
                                ; 32-bit ACC register
```

Note: This addressing mode can only access the lower 64K of data address space on the C28x device.

5.6 Indirect Addressing Modes

This section includes indirect addressing modes for the 28x and 2xLP devices. It also includes circular indirect addressing modes.

5.6.1 C28x Indirect Addressing Modes (XAR0 to XAR7)

AMODE	”loc16/loc32” Syntax	Description

X	*XARn++	ARP = n
		32bitDataAddr(31:0) = XARn
		if(loc16), XARn = XARn + 1
		if(loc32), XARn = XARn + 2

Example (s):

```
     MOVL   XAR2, #Array1    ; Load XAR2 with start address of Array1
     MOVL   XAR3, #Array2    ; Load XAR3 with start address of Array2
     MOV    @AR0, #N-1       ; Load AR0 with loop count N
Loop:
     MOVL   ACC,*XAR2++      ; Load ACC with location pointed to by XAR2,
                             ; post−increment XAR2
     MOVL   *XAR3++,ACC      ; Store ACC into location pointed to by XAR3,
                             ; post−increment XAR3
     BANZ   Loop,AR0−−       ; Loop until AR0 == 0, post−decrement AR0
```

AMODE	”loc16/loc32” Syntax	Description

X	*--XARn	ARP = n
		if(loc16), XARn = XARn + 1
		if(loc32), XARn = XARn + 2
		32bitDataAddr(31:0) = XARn

Example (s):

```
     MOVL   XAR2,#Array1+N*2  ; Load XAR2 with start address of Array1
     MOVL   XAR3,#Array2+N*2  ; Load XAR3 with start address of Array2
     MOV    @AR0, #N-1        ; Load AR0 with loop count N
Loop:
     MOVL   ACC,*--XAR2       ; Pre-decrement XAR2,
                              ; load ACC with location pointed to by XAR2
     MOVL   *--XAR3,ACC       ; Pre-decrement XAR3,
                              ; store ACC into location pointed to by XAR3,
     BANZ   Loop,AR0−−        ; Loop until AR0 == 0, post−decrement AR0
```

AMODE	”loc16/loc32” Syntax	Description
X	`*+XARn[AR0]`	ARP = n 32bitDataAddr(31:0) = XARn + AR0 **Note:** The lower 16-bits of XAR0 are added to the selected 32-bit register. Upper 16-bits of XAR0 are ignored. AR0 is treated as an unsigned 16-bit value. Overflow into the upper 16-bits of XARn can occur.

Example (s):

```
MOVW   DP,#Array1Ptr    ; Point to Array1 Pointer location
MOVL   XAR2,@Array1Ptr  ; Load XAR2 with pointer to Array1
MOVB   XAR0,#16         ; AR0 = 16, AR0H = 0
MOVB   XAR1,#68         ; AR1 = 68, AR1H = 0
MOVL   ACC,*+XAR2[AR0]  ; ; Swap contents of location Array1[16]
MOVL   P,*+XAR2[AR1]    ; ; with the contents of location Array1[68]
MOVL   *+XAR2[AR1],ACC  ; ;
MOVL   *+XAR2[AR0],P    ; ;
```

AMODE	”loc16/loc32” Syntax	Description
X	`*+XARn[AR1]`	ARP = n 32bitDataAddr(31:0) = XARn + AR1 **Note:** The lower 16-bits of XAR0 are added to the selected 32-bit register. Upper 16-bits of XAR0 are ignored. AR0 is treated as an unsigned 16-bit value. Overflow into the upper 16-bits of XARn can occur.

Example (s):

```
MOVW   DP,#Array1Ptr    ; Point to Array1 Pointer location
MOVL   XAR2,@Array1Ptr  ; Load XAR2 with pointer to Array1
MOVB   XAR0,#16         ; AR0 = 16, AR0H = 0
MOVB   XAR1,#68         ; AR1 = 68, AR1H = 0
MOVL   ACC,*+XAR2[AR0]  ; ; Swap contents of location Array1[16]
MOVL   P,*+XAR2[AR1]    ; ; with the contents of location Array1[68]
MOVL   *+XAR2[AR1],ACC  ; ;
MOVL   *+XAR2[AR0],P    ; ;
```

AMODE	”loc16/loc32” Syntax	Description
X	`*+XARn[3bit]`	ARP = n 32bitDataAddr(31:0) = XARn +3bit **Note:** The immediate value is treated as an unsigned 3-bit value.

Example (s):

```
MOVW   DP,#Array1Ptr    ; Point to Array1 Pointer location
MOVL   XAR2,@Array1Ptr  ; Load XAR2 with pointer to Array1
MOVL   ACC,*+XAR2[2]    ; ; Swap contents of location Array1[2]
MOVL   P,*+XAR2[5]      ; ; with the contents of location Array1[5]
MOVL   *+XAR2[5],ACC    ; ;
MOVL   *+XAR2[2],P      ; ;
```

Note: The assembler also accepts ”*XARn” as an addressing mode. This is the same encoding as the ”*+XARn[0]” mode.

5.6.2 C2xLP Indirect Addressing Modes (ARP, XAR0 to XAR7)

AMODE	”loc16/loc32” Syntax	Description
X	*	32bitDataAddr(31:0) = XAR(ARP) **Note:** The XARn register used is the register pointed to by the current value in the ARP pointer. ARP = 0, points to XAR0, ARP = 1, points to XAR1 and so on.

Example (s):

```
MOVZ    DP,#RegAPtr       ; Load DP with page address containing RegAPtr
MOVZ    AR2,@RegAPtr      ; Load AR2 with contents of RegAPtr, AR2H = 0
MOVZ    AR3,@RegAPtr      ; Load AR3 with contents of RegBPtr, AR3H = 0
                          ; RegAPtr and RegBPtr are located in the same 128 word data page,
                          ; Both are located in the low 64K of data memory space.
NOP     *,ARP2            ; Set ARP pointer to point to XAR2
MOV     *,#0x0404         ; Store 0x0404 into location pointed by XAR2
NOP     *,ARP3            ; Set ARP pointer to point to XAR3
MOV     *,#0x8000         ; Store 0x8000 into location pointed by XAR3,
```

AMODE	”loc16/loc32” Syntax	Description

X	*,XARn	32bitDataAddr(31:0) = XAR(ARP)
		ARP = n

Example (s):

```
MOVZ    DP,#RegAPtr       ; Load DP with page address containing RegAPtr
MOVZ    AR2,@RegAPtr      ; Load AR2 with contents of RegAPtr, AR2H = 0
MOVZ    AR3,@RegAPtr      ; Load AR3 with contents of RegBPtr, AR3H = 0
                          ; RegAPtr and RegBPtr are located in the same 128 word data page,
                          ; Both are located in the low 64K of data memory space.
NOP     *,ARP2            ; Set ARP pointer to point to XAR2
MOV     *,#0x0404,ARP3    ; Store 0x0404 into location pointed by XAR2
                          ; Set ARP pointer to point to XAR3
MOV     *,#0x8000         ; Store 0x8000 into location pointed by XAR3,
```

AMODE	”loc16/loc32” Syntax	Description

X	`*++`	`32bitDataAddr(31:0) = XARn (ARP)`
		`if(loc16), XAR(ARP) = XAR(ARP) + 1`
		`if(loc32), XAR(ARP) = XAR(ARP) + 2`

Example (s):

```
        MOVL    XAR2, #Array1      ; Load XAR2 with start address of Array1
        MOVL    XAR3, #Array2      ; Load XA32 with start address of Array2
        MOV     @AR0,#N-1          ; ALoad AR0 with loop count N
```

Loop:

```
        NOP     *,ARP2             ; Set ARP pointer to point to XAR2
        MOVL    ACC,*++            ; Load ACC with location pointed to by XAR2,
                                   ; post-increment XAR2
        NOP     *,ARP3             ; Set ARP pointer to point to XAR3
        MOVL    *++,ACC            ; Store ACC into location pointed to by XAR3,
                                   ; post-increment XAR3
        NOP     *,ARP0             ; Set ARP pointer to point to XAR0
        XBANZ   Loop,*--           ; Loop until AR0 == 0, post-decrement AR0
```

AMODE	”loc16/loc32” Syntax	Description

X	`*++,ARPn`	`32bitDataAddr(31:0) = XARn + AR1`
		`if(loc16), XAR(ARP) = XAR(ARP) + 1`
		`if(loc32), XAR(ARP) = XAR(ARP) + 2`

Example (s):

```
        MOVL    XAR2,#Array1       ; Load XAR2 with start address of Array1
        MOVL    XAR2,#Array2       ; Load XAR3 with start address of Array2
        MOV     @AR0,#N-1          ; Load AR0 with loop count N
        NOP     *,ARP2             ; Set ARP pointer to point to XAR2
        SETC    AMODE              ; Make sure AMODE = 1
        .lp_amode                  ; Tell assembler that AMODE = 1
```

Loop:

```
        MOVL    ACC,*++,ARP3       ; Load ACC with location pointed to by XAR2
                                   ; post-increment XAR2, set ARP to point to XAR3
        MOVL    *++,ACC,ARP0       ; Store ACC into location pointed to by XAR3,
                                   ; post-increment XAR3, set ARP to point to XAR0
        XBANZ   Loop, *--,ARP2     ; Loop until AR0 == 0, post-decrement AR0,
                                   ; set ARP pointer to point to XAR2
```

AMODE	”loc16/loc32” Syntax	Description

X	`*--`	`32bitDataAddr(31:0) = XAR(ARp)`
		`if(loc16), XAR(ARP) = XAR(ARP) + 1`
		`if(loc32), XAR(ARP) = XAR(ARP) + 2`

Example (s):

```
     MOVL   XAR2,#Array1+(n-1)*2    ; Load XAR2 with ends address of Array1
     MOVL   XAR3,#Array2+(n-1)*2    ; Load XAR3 with end address of Array2
     MOV    @AR0,#N-1               ; Load AR0 with loop count N
```

Loop:

```
     NOP    *,ARP2                  ; Set ARP pointer to point to XAR2
     MOVL   ACC,*--                 ; Load ACC with location pointed to by XAR2
                                    ; post−increment XAR2
     NOP    *,ARP3                  ; Set ARP pointer to point to XAR3
     MOVL   *--,ACC                 ; Store ACC into location pointed to by XAR3,
                                    ; post−increment XAR3
     NOP    *,ARP0                  ; Set ARP pointer to point to XAR0
     XBANZ  Loop, *--               ; Loop until AR0 == 0, post−decrement AR0
```

AMODE	”loc16/loc32” Syntax	Description

1	`*--,ARPn`	`32bitDataAddr(31:0) = XAR(ARp)`
		`if(loc16), XAR(ARP) = XAR(ARP) + 1`
		`if(loc32), XAR(ARP) = XAR(ARP) + 2`
		ARP=n

Example (s):

```
     MOVL   XAR2,#Array1+(n-1)*2   ; Load XAR2 with ends address of Array1
     MOVL   XAR3,#Array2+(n-1)*2   ; Load XAR3 with end address of Array2
     MOV    @AR0,#N-1              ; Load AR0 with loop count N
     NOP    *,ARP2                 ; Set ARP pointer to point to XAR2
     SETC   AMODE                  ; Make sure AMODE = 1
     .lp_amode                     ; Tell assembler that AMODE = 1
```

Loop:

```
     MOVL   ACC,*--,ARP3           ; Load ACC with location pointed to by XAR2
                                   ; post−increment XAR2, set ARP to point to XAR3
     MOVL   *--,ACC,ARP0           ; Store ACC into location pointed to by XAR3,
                                   ; post−increment XAR3, set ARP to point to XAR0
     XBANZ  Loop, *--,ARP2         ; Loop until AR0 == 0, post−decrement AR0
                                   ; set ARP pointer to point to XAR2
```

AMODE	”loc16/loc32” Syntax	Description

X	`*++`	`32bitDataAddr(31:0) = XAR(ARP)`
		`XAR(ARP)=XAR(ARP)+AR0`
		Note: The lower 16-bits of XAR0 are added to the selected 32-bit register. Upper 16-bits of XAR0 ignored. AR0 is treated as an unsigned 16-bit value. Overflow into the upper 16-bits of XAR(ARP) can occur.

Example (s):

```
MOVL   XAR2,#Array1        ; Load XAR2 with end address of Array1
MOVL   XAR3,#Array2        ; Load XAR3 with end address of Array2
MOV    @AR0,#4             ; Load AR0 to copy every fourth value from Array1 to Array2
MOV    @AR1m #N-1          ; Load AR1 with loop count N
```

Loop:

```
NOP    *,ARP2              ; Set ARP pointer to point to XAR2
MOVL   ACC,*0++            ; Load ACC with location pointed to by XAR2,
                           ; post-increment XAR2 by AR0
NOP    *,ARP3              ; Set ARP pointer to point to XAR3
MOVL   *++,ACC             ; Store ACC with location pointed to by XAR3, post−increment XAR3
NOP    *,ARP1              ; Set ARP pointer to point to XAR1
XBANZ  Loop, *--           ; Loop until AR0 == 0, post−decrement AR1
```

AMODE	”loc16/loc32” Syntax	Description

1	`*++,ARPn`	`32bitDataAddr(31:0) = XAR(ARP)`
		`XAR(ARP)=XAR(ARP)+AR0`
		`ARP=n`
		Note: The lower 16-bits of XAR0 are added to the selected 32-bit register. Upper 16-bits of XAR0 ignored. AR0 is treated as an unsigned 16-bit value. Overflow into the upper 16-bits of XAR(ARP) can occur.

Example (s):

```
MOVL   XAR2,#Array1        ; Load XAR2 with end address of Array1
MOVL   XAR3,#Array2        ; Load XAR3 with end address of Array2
MOV    @AR0,#4             ; Set AR0 to copy every fourth value from Array1 to Array2
MOV    @AR1n #N-1          ; Load AR1 with loop count N
NOP    *,ARP2              ; Set ARP pointer to point to XAR2
SETC   AMODE               ; Make sure AMODE = 1
.lp_amode                  ; Tell assembler that AMODE = 1
```

Loop:

```
MOVL   ACC,*0++,ARP3       ; Load ACC with location pointed to by XAR2,
                           ; post-increment XAR2 by AR0, set ARP pointer
MOVL   *++,ACC,ARP1        ; Store ACC with location pointed to by XAR3,
                           ; post−increment XAR3, set ARP pointer to point tp XAR1
XBANZ  Loop, *--,ARP2      ; Loop until AR0 == 0, post−decrement AR1,
                           ; set ARP to point to XAR2
```

AMODE	”loc16/loc32” Syntax	Description

X	`*0--`	`32bitDataAddr(31:0) = XAR(ARP)`
		`XAR(ARP)=XAR(ARP)+AR0`
		Note: The lower 16-bits of XAR0 are added to the selected 32-bit register. Upper 16-bits of XAR0 ignored. AR0 is treated as an unsigned 16-bit value. Overflow into the upper 16-bits of XAR(ARP) can occur.

Example (s):

```
      MOVL   XAR2,#Array1+(N-1)*8    ; Load XAR2 with end address of Array1
      MOVL   XAR3,#Array2+(N-1)*2    ; Load XAR3 with end address of Array2
      MOV    @AR0,#4                 ; Set AR0 to copy every fourth value from Array1 to Array2
      MOV    @AR1n #N-1              ; Load AR1 with loop count N
Loop:
      NOP    *,ARP2                  ; Set ARP pointer to point to XAR2
      MOVL   ACC,*0--                ; Load ACC with location pointed to by XAR2,
                                     ; post-increment XAR2 by AR0
      NOP    *,ARP3                  ; Set ARP pointer to point to XAR3
      MOVL   *--,ACC                 ; Store ACC with location pointed to by XAR3,
                                     ; post−increment XAR3
      NOP    *,ARP1                  ; Set ARP pointer to point to XAR1
      XBANZ  Loop, *--               ; Loop until AR1 == 0, post−decrement AR1
```

AMODE	”loc16/loc32” Syntax	Description

1	`*0--,ARPn`	`32bitDataAddr(31:0) = XAR(ARP)`
		`XAR(ARP)=XAR(ARP)+AR0`
		`ARP=n`
		Note: The lower 16-bits of XAR0 are added to the selected 32-bit register. Upper 16-bits of XAR0 ignored. AR0 is treated as an unsigned 16-bit value. Overflow into the upper 16-bits of XAR(ARP) can occur.

Example (s):

```
      MOVL   XAR2,#Array1+(N-1)*8    ; Load XAR2 with end address of Array1
      MOVL   XAR3,#Array2+(N-1)*2    ; Load XAR3 with end address of Array2
      MOV    @AR0,#4                 ; Set AR0 to copy every fourth value from Array1 to Array2
      MOV    @AR1n #N-1              ; Load AR1 with loop count N
      NOP    *,ARP2                  ; Set ARP pointer to point to XAR2
      SETC   AMODE                   ; Make sure AMODE = 1
      .lp_amode                      ; Tell assembler that AMODE = 1
Loop:
      MOVL   ACC,*0--,ARP3           ; Load ACC with location pointed to by XAR2,
                                     ; post-increment XAR2 by AR0, set ARP pointer to XAR3
      MOVL   *++,ACC,ARP1            ; Store ACC with location pointed to by XAR3,
                                     ; post−increment XAR3, set ARP pointer to point tp XAR1
      XBANZ  Loop, *--,ARP2          ; Loop until AR0 == 0, post−decrement AR1,
                                     ; set ARP to point to XAR2
```

AMODE	”loc16/loc32” Syntax	Description
X	*BR0++	32bitDataAddr(31:0) = XAR(ARP) XAR(ARP)(15:0)= AR(ARP)rcadd AR0 XAR(ARP)(31:16) = unchanged **Note:** The lower 16-bits of XAR0 are reverse carry added (rcadd) to the lower 16-bits of the selected register. Upper 16-bits of XAR0 ignored. Upper 16-bits of the selected register unchanged by the operation.

Example (s):

```
; Transfer contents of Array1 to Array2 in bit reverse order:
     MOVL   XAR2,#Array1          ; Load XAR2 with end address of Array1
     MOVL   XAR3,#Array2          ; Load XAR3 with end address of Array2
     MOV    @AR0,#N               ; Load AR0 with size of array,
                                  ; N must be a multiple of 2 (2, 4, 8, 16, ...)
     MOV    @AR1 #N-1             ; Load AR1 with loop count N
Loop:
     NOP    *,ARP2                ; Set ARP pointer to point to XAR2
     MOVL   ACC,*0++              ; Load ACC with location pointed to by XAR2, post-
                                    increment XAR2
     NOP    *,ARP3                ; Set ARP pointer to point to XAR3
     MOVL   *BR0++,ACC            ; Store ACC with location pointed to by XAR3,
                                  ; post−increment XAR3 with AR0 reverse carry add
     NOP    *,ARP1                ; Set ARP pointer to point to XAR1
     XBANZ  Loop, *--             ; Loop until AR1 == 0, post−decrement AR1
```

AMODE	”loc16/loc32” Syntax	Description
1	*BR0++,ARPn	32bitDataAddr(31:0) = XAR(ARP) XAR(ARP)(15:0)= AR(ARP) rcadd AR0 XAR(ARP)(31:16) = unchanged ARP=n **Note:** The lower 16-bits of XAR0 are reverse carry added (rcadd) to the lower 16-bits of the selected register. Upper 16-bits of XAR0 ignored. Upper 16-bits of the selected register unchanged by the operation.

Example (s):

```
; Transfer contents of Array1 to Array2 in bit reverse order:
     MOVL   XAR2,#Array1          ; Load XAR2 with start address of Array1
     MOVL   XAR3,#Array2          ; Load XAR3 with start address of Array2
     MOV    @AR0 #N               ; Load AR0 with size of array,
                                  ; N must be a multiple of 2 (2, 4, 8, 16,...)
     MOV    @AR1, #N-1            ; Load AR1 with loop count N
     NOP    *,ARP2                ; Set ARP pointer to point to XAR2
     SETC   AMODE                 ; Make sure AMODE = 1
     .lp_amode                    ; Tell assembler that AMODE = 1
```

Loop:

```
     MOVL   ACC,*++,ARP3          ; Load ACC with location pointed to by XAR2,
                                  ; post-increment XAR2 by AR0, set ARP pointer to XAR3
     MOVL   *++,ACC,ARP1          ; Store ACC with location pointed to by XAR3,
                                  ; post−increment XAR3, set ARP pointer to point tp XAR1
     MOVL   *BR0++,ACC,ARP1       ; Store ACC with location pointed to by XAR3,
                                  ; post−increment XAR3 with AR0 reverse carry
                                  ; add, set ARP pointer to point to XAR1
     XBANZ  Loop, *--,ARP2        ; Loop until AR1 == 0, post−decrement AR1,
                                  ; set ARP to point to XAR2
```

AMODE	”loc16/loc32” Syntax	Description

X	`*BR0--`	Address Generation:
		`32bitDataAddr(31:0)=XAR(ARP)`
		`XAR(ARP)(15:0)= AR(ARP)rbsub AR0` **{see note [1]}**
		`XAR(ARP)(31:16) = unchanged`
		Note: The lower 16-bits of XAR0 are reverse carry added (rbsub) from the lower 16-bits of the selected register. Upper 16-bits of XAR0 ignored. Upper 16-bits of the selected register unchanged by the operation.

Example (s):

```
; Transfer contents of Array1 to Array2 in bit reverse order:
     MOVL    XAR2,#Array1+(N-1)*2   ; Load XAR2 with end address of Array1
     MOVL    XAR3,#Array2+(N-1)*2   ; Load XAR3 with end address of Array2
     MOV     @AR0,#N                ; Load AR0 with size of array,
                                    ; N must be a multiple of 2 (2, 4, 8, 16,...)
     MOV     @AR1 #N-1              ; Load AR1 with loop count N
Loop:
     NOP     *,ARP2                 ; Set ARP pointer to point to XAR2
     MOVL    ACC,*0--               ; Load ACC with location pointed to by XAR2,
                                    ; post-increment XAR2
     NOP     *,ARP3                 ; Set ARP pointer to point to XAR3
     MOVL    *BR0--,ACC             ; Store ACC with location pointed to by XAR3,
                                    ; post-increment XAR3 with AR0 reverse carry add
     NOP     *,ARP1                 ; Set ARP pointer to point to XAR1
     XBANZ   Loop, *--              ; Loop until AR1 == 0, post-decrement AR1
```

AMODE	”loc16/loc32” Syntax	Description

1	`*BR0--,ARPn`	`32bitDataAddr(31:0) = XAR(ARP)`
		`XAR(ARP)(15:0)= AR(ARP) rbsub AR0`
		`XAR(ARP)(31:16) =  unchanged`
		`ARP=n`
		Note: The lower 16-bits of XAR0 are reverse carry added (rbsub) from the lower 16-bits of the selected register. Upper 16-bits of XAR0 ignored. Upper 16-bits of the selected register unchanged by the operation.

Example (s):

```
; Transfer contents of Array1 to Array2 in bit reverse order:
     MOVL   XAR2,#Array1+(N-1)*2  ; Load XAR2 with start address of Array1
     MOVL   XAR3,#Array2+(N-1)*2  ; Load XAR3 with start address of Array2
     MOV    @AR0 #N               ; Load AR0 with size of array,
                                  ; N must be a multiple of 2 (2, 4, 8, 16,...)
     MOV    @AR1, #N-1            ; Load AR1 with loop count N
     NOP    *,ARP2                ; Set ARP pointer to point to XAR2
     SETC   AMODE                 ; Make sure AMODE = 1
     .lp_amode                    ; Tell assembler that AMODE = 1
Loop:
     MOVL   ACC,*--,ARP3          ; Load ACC with location pointed to by XAR2,
                                  ; post-increment XAR2, set ARP pointer to point to XAR3
     MOVL   *++,ACC,ARP1          ; Store ACC with location pointed to by XAR3,
                                  ; post-increment XAR3, set ARP pointer to point tp XAR1
     MOVL   *BR0--,ACC,ARP1       ; Store ACC with location pointed to by XAR3,
                                  ; post-increment XAR3 with AR0 reverse borrow substract,
                                  ; set ARP pointer to point to XAR1
     XBANZ  Loop, *--,ARP2        ; Loop until AR1 == 0, post-decrement AR1,
                                  ; set ARP to point to XAR2
```

Reverse carry addition or reverse carry subtraction is used to implement bit-reversed addressing as used in the re−ordering of data elements in FFT algorithms. Typically, AR0 is initialized with the (FFT size) /2. The value of AR0 is then added or subtracted, with reverse carry addition or subtraction, to generate the bit reversed address

Reverse Carry Addition Example Is Shown Below (FFT size = 16):

```
XAR(ARP)(15:0) = 0000 0000 0000 0000
+ AR0          = 0000 0000 0000 1000
-------------    -------------------
XAR(ARP)(15:0) = 0000 0000 0000 1000
+ AR0          = 0000 0000 0000 1000
-------------    -------------------
XAR(ARP)(15:0) = 0000 0000 0000 0100
+ AR0          = 0000 0000 0000 1000
-------------    -------------------
XAR(ARP)(15:0) = 0000 0000 0000 1100
+ AR0          = 0000 0000 0000 1000
-------------    -------------------
XAR(ARP)(15:0) = 0000 0000 0000 0010
+ AR0          = 0000 0000 0000 1000
-------------    -------------------
XAR(ARP)(15:0) = 0000 0000 0000 1010
......
```

Reverse Borrow Subtraction Example Is Shown Below (FFT size = 16):

```
XAR(ARP)(15:0) = 0000 0000 0000 0000
- AR0          = 0000 0000 0000 1000
-------------    -------------------
XAR(ARP)(15:0) = 0000 0000 0000 1111
- AR0          = 0000 0000 0000 1000
-------------    -------------------
XAR(ARP)(15:0) = 0000 0000 0000 0111
- AR0          = 0000 0000 0000 1000
-------------    -------------------
XAR(ARP)(15:0) = 0000 0000 0000 1011
- AR0          = 0000 0000 0000 1000
-------------    -------------------
XAR(ARP)(15:0) = 0000 0000 0000 0011
- AR0          = 0000 0000 0000 1000
-------------    -------------------
XAR(ARP)(15:0) = 0000 0000 0000 1101
......
```

On the C28x, the bit reversed addressing is restricted to block size < 64K. This is OK since most FFT implementations are much less than this.

5.6.3 Circular Indirect Addressing Modes (XAR6, XAR1)

AMODE	”loc16/loc32” Syntax	Description

0	*AR6%++	32bitDataAddr(31:0) = XAR6
		if(XAR6(7:0) == XAR1(7:0))
		{
		XAR6(7:0) = 0x00
		XAR6(15:8) = unchanged
		}
		else
		{
		if(16-bit data), XAR6(15:0) =+ 1
		if(32-bit data), XAR6(15:0) =+ 2
		}
		XAR6(31:16) = unchanged
		ARP = 6

As seen in Figure 5-1, buffer size is determined by the 8 LSBs of AR1 or AR1[7:0]. Specifically, the buffer size is AR1[7:0] +1. When AR1[7:0] is 255, then the buffer size is at its maximum size of 256 words.

XAR6 points to the current address in the buffer. The top of the buffer must be at an address where the 8 LSBs are all 0s.

If one of the instructions accessing the circular buffer performs a 32-bit opera- tion, make sure XAR6 and AR1 are both even before the buffer is accessed.

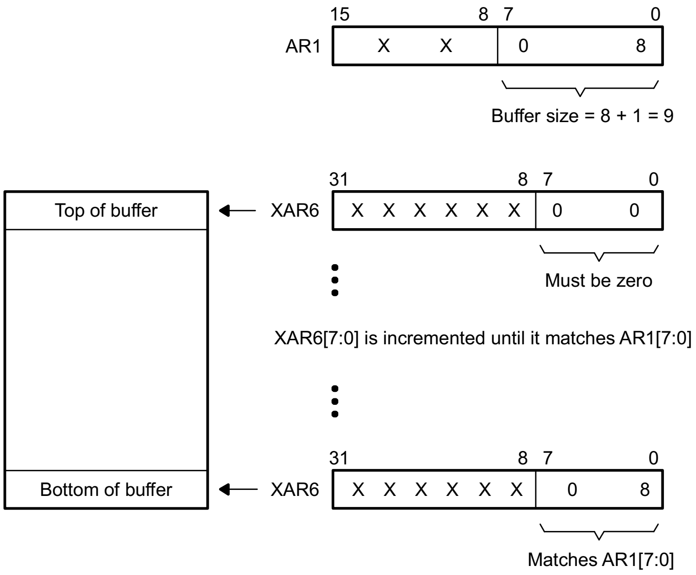


Figure 5-1. Circular Buffer with AMODE = 0

```
Example (s):
; Calculate FIR filter (X[N] = data array, C[N] = coefficient array):
     MOVW    DP,#Xpointer                ; Load DP with page address of Xpointer
     MOVL    XAR6,@Xpointer              ; Load XAR6 with current X pointer
     MOVL    XAR7,#C                     ; Load XAR7 with start address of C array
     MOV     @AR1,#N                     ; Load AR1 with size of data array N,
     SPM     −4                          ; Set product shift mode to  ">> 4","
     ZAPA                                ; Zero ACC, P, OVC
     RPT     #N−1                        ; Repeat next instruction N timeses
   ||QMACL P,*AR6%++,*XAR7++             ; ACC = ACC + P >> 4,
                                         ; P = (*AR6%++ * *XAR7++) >> 32
     ADDL    ACC,P << PM                 ; Final accumulate
     MOVL    @Xpointer,XAR6              ; Store XAR6 into current X pointer
     MOVL    @Sum,ACC                    ; Store result into sum
```

AMODE	”loc16/loc32” Syntax	Description

1	`*+XAR6[AR1%++]`	32bitDataAddr(31:0) = XAR6 + AR1
		if(XAR1(15:0) == XAR1(31:16))
		{
		XAR1(15:0) = 0x0000
		XAR6(15:8) = unchanged
		}
		else
		{
		if(16-bit data), XAR1(15:0) =+ 1
		if(32-bit data), XAR1(15:0) =+ 2
		}
		XAR1(31:16) = unchanged
		ARP = 6
		Note: With this addressing mode, there is no circular buffer alignment requirements.

As seen in Figure 5-2, buffer size is determined by the upper 16 bits of XAR1 or XAR1[31:16]. Specifically, the size is XAR1[31:16] + 1.

XAR6 points to the top of the buffer.

The current address in the buffer is pointed to by XAR6 with an offset of XAR1[15:0].

If the instructions that access the circular buffer perform 32-bit operations, make sure XAR6 and XAR1[31:16] are even.

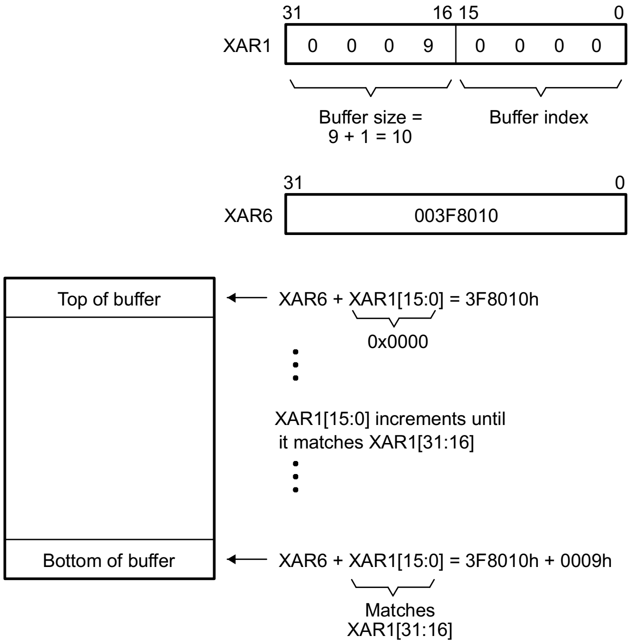


Figure 5-2. Circular Buffer with AMODE = 1

```
Example (s):
; Calculate FIR filter (X[N] = data array, C[N] = coefficient array):
     MOVW    DP,#Xindex                         ; Load DP with page address of Xindex
     MOVL    XAR6,#X                            ; Load XAR6 with start address of X array
     MOV     @AH,#N                             ; Load AH with size of data array X (N)
     MOV    AL,@Xindex                          ; Load AL with current circular index
     MOVL    XAR1,@ACC                          ; Load parameters into XAR1
     MOVL    XAR7,#C                            ; Load XAR7 with start address of C array
     SPM     -4                                 ; Set product shift mode to  ">> 4","
     ZAPA                                       ; Zero ACC, P, OVC
     RPT     #N-1                               ; Repeat next instruction N timeses
   ||QMACL P,*XAR6[AR1%++],*XAR7++              ; ACC = ACC + P >> 4,
                                                ; P = (*AR6%++ * *XAR7++) >> 32
     ADDL    ACC,P << PM                        ; Final accumulate
     MOV    @Xindex,XAR1                        ; Store XAR6 into current X index
     MOVL    @Sum,ACC                           ; Store result into sum
```

5.7 Register Addressing Modes

This section includes register addressing modes for 32-bit and 16-bit registers.

5.7.1 32-Bit Register Addressing Modes

AMODE	”loc32” Syntax	Description
X	@ACC	Access contents of 32-bit ACC register. When the ”@ACC” register is the destination operand, this may affect the Z,N,V,C,OVC flags.

Example (s):

```
MOVL   XAR6,@ACC       ; Load XAR6 with contents of ACC
MOVL   @ACC,XT         ; Load ACC with contents of XT register
ADDL   ACC,@ACC        ; ACC = ACC + ACC
```

AMODE	”loc32” Syntax	Description
X	@P	Access contents of 32-bit ACC register.

Example (s):

```
MOVL   XAR6,@AP        ; Load XAR6 with contents of P
MOVL   @P,XT           ; Load P with contents of XT register
ADDL   ACC,@AP         ; P = ACC + P
```

AMODE	”loc32” Syntax	Description
X	@XT	Access contents of 32-bit XT register.

Example (s):

```
MOVL   XAR6,@XT        ; Load XAR6 with contents of XT
MOVL   @P,XT           ; Load P with contents of XT register
ADDL   ACC,@XT         ; ACC = ACC + XT
```

AMODE	”loc32” Syntax	Description
X	@XARn	Access contents of 32-bit XARn registers.

Example (s):

```
MOVL   XAR6,@XR2       ; Load XAR6 with contents of XAR2
MOVL   P,@XAR2         ; Load P with contents of XAR2 register
ADDL   ACC,@XAR2       ; ACC = ACC + XAR2
```

Note: When writing assembly code, the “@” symbol in front of the register is optional. For example: “MOVL ACC,@P” or “MOVL ACC,P”. The disassembler will use the @ to indicate operands that are “loc16” or “loc32”. For example, MOVL ACC, @P is the MOVL ACC, loc32 instruction and MOVL @ACC, P is the MOVL loc32, P instruction.

5.7.2 16-Bit Register Addressing Modes

AMODE	”loc16” Syntax	Description
X	@AL	Access contents of 16-bit AL register. AH register contents are un-affected. When the ”@AL” register is the destination operand, this may affect the Z,N,V,C,OVC flags.

Example (s):

```
MOV    PH,@AL            ; Load PH with contents of AL
ADD    AH,@AL            ; AH = AH + AL
MOV    T,@AL             ; Load P with contents of AL
```

AMODE	”loc16” Syntax	Description
X	@AH	Access contents of 16-bit Ah register. Al register contents are unaffected. When the ”@AH” register is the destination operand, this may affect the Z,N,V,C,OVC flags.

Example (s):

```
MOV    PH,@AH            ; Load PH with contents of AH
ADD    AL,@AH            ; Al = Al + Ah
MOV    T,@AH             ; Load t with contents of AH
```

AMODE	”loc16” Syntax	Description
X	@PL	Access contents of 16-bit PL register. PH register contents are unaffected.

Example (s):

```
MOV    PH,@AL            ; Load PH with contents of PL
ADD    AH,@AL            ; AL = AL + PL
MOV    T,@AL             ; Load T with contents of PL
```

AMODE	”loc16” Syntax	Description
X	@PH	Access contents of 16-bit PH register. PL register contents are unaffected.

Example (s):

```
MOV    PL,@PH            ; Load PL with contents of PH
ADD    AL,@PH            ; AL = AL + PH
MOV    T,@PH             ; Load T with contents of PH
```

AMODE	”loc16” Syntax	Description
X	@TH	Access contents of 16-bit TH register. TL register contents are unaffected.

Example (s):

```
MOV     PL,@T          ; Load PL with contents of T
ADD     AL,@T          ; AL = AL + TH
MOVZ    AR4,@T         ; Load AR4 with contents of T, AR4H = 0
```

AMODE	”loc16” Syntax	Description
X	@TH	Access contents of 16-bit SP register

Example (s):

```
MOVZ    AR4,@SP        ; Load AR4 with contents of SP, AR4H = 0
MOV     AL,@SP         ; Load AL with contents of SP
MOV     @SP,AH         ; Load SP with contents of AH
```

AMODE	”loc16” Syntax	Description
X	@ARn	Access contents of 16-bit AR0 to AR7 registerS. AR0H to AR7H register contents are unaffected.

Example (s):

```
MOVZ    AR4,@AR2       ; Load AR4 with contents of AR2, AR4H = 0
MOV     AL,@AR3        ; Load AL with contents of AR3
MOV     @AR5,AH        ; Load AR5 with contents of AH, AR5H = unchanged
```

5.8 Data/Program/IO Space Immediate Addressing Modes

Syntax	Description

*(0:16bit)	`32BitDataAddr(31:16) = 0`
	`32BitDataAddr(15:0) = 16-bit immediate value`
	Note: If instruction is repeated, the address is post−incremented on each iteration. This addressing mode can only access the low 64K of data space.

```
Instructions that use this addressing mode:
MOV             loc16,*(0:16bit)   ; [loc16] = [0:16bit]
MOV             *(0:16bit),loc16   ; [loc16] = [0:16bit]
```

Syntax	Description

*(PA)	`32BitDataAddr(31:16) = 0`
	`32BitDataAddr(15:0) = PA 16-bit immediate value`
	Note: If instruction is repeated, the address is post−incremented on each iteration. The I/O strobe signal is toggled when accessing I/O space with this addressing mode. The data space address lines are used for accessing I/O space.

```
Instructions that use this addressing mode:
OUT             (PA),*loc16        ; IOspace[0:pa]= [loc16]
UOUT            *(0:16bit),loc16   ; IOspace[0:pa]= [loc16](unprotected)
IN              loc16,*(PA)        ; [loc16] = IOspace[0:PA]
```

Syntax	Description

0:pma	`22BitProgAddr(21:16) = 0`
	`22BitProgAddr(15:0) = pma 16-bit immediate value`
	Note: If instruction is repeated, the address is post−incremented on each iteration. This addressing mode can only access the low 64K of program space.

```
Instructions that use this addressing mode:
MAC             p,loc16,0:pma     ; ACC = ACC + P << PM,
                                  ; P = [loc16] * ProgSpace[0:pma]
```

Syntax	Description

*(0:16bit)	`22BitProgAddr(21:16) = 0x3F`
	`22BitProgAddr(15:0) = pma 16-bit immediate value`
	Note: If instruction is repeated, the address is post−incremented on each iteration. This addressing mode can only access the upper 64K of program space.

```
Instructions that use this addressing mode:
XPREAD          loc16, *(pma)      ; [loc16] = ProgSpace [0x3f:pma]
XMAC            P,loc16*(pma)      ; ACC = ACC + P << PM,
                                   ; P = [loc16] * ProgSpace[0x3F:pma]
XMACD           P,loc16*(pma)      ; ACC = ACC + P << PM,
                                   ; P = [loc16] * ProgSpace[0x3F:pma]
                                   ; [loc16+1] = [loc16]
```

5.9 Program Space Indirect Addressing Modes

Syntax	Description
*AL	22BitProgAddr(21:16) = 0x3F 22BitProgAddr(15:0) = AL **Note:** If instruction is repeated, the address in AL is copied to a shadow register and the value post−incremented on each iteration. The AL register is not modified. This addressing mode can only access the upper 64K of program space.

```
Instructions that use this addressing mode:
XPREAD       loc16,*AL             ; [loc16] = ProgSpace [0x3F:AL]
XPWRITE      *AL,loc16             ; ProgSpace [0x3F:AL] = [loc16]
```

Syntax	Description
*XAR7	22BitProgAddr(21:0) = XAR7 **Note:** If instruction is repeated, only in the XPREAD and XPWRITE instructions, is the address contained in XAR7 copied to a shadow register and the value post−incremented on each iteration. The XAR7 register is not modified. For all other instructions, the address is not incremented even when repeated.

```
Instructions that use this addressing mode:
MAC          P,loc16,*XAR7         ; ACC = ACC + P << PM,
                                   ; P = [loc16] * ProgSpace[*XAR7]
DMAC         ACC:P,loc32,*XAR7     ; ACC = ([loc32].MSW * ProgSpace[*XAR7].MSW) >> PM,
                                   ; P = ([loc32].LSW * ProgSpace[*XAR7].MSW) >> PM
QMACL        P,loc32,*XAR7         ; ACC = ACC + P >> PM,
                                   ; P = ([loc32] * ProgSpace[*XAR7] >> 32
IMACL        P,loc32,*XAR7         ; ACC = ACC + P,
                                   ; P = ([loc32] * ProgSpace[*XAR7] << PM
PREAD        loc16,*XAR7           ; [loc16] = ProgSpace[*XAR7]
PWRITE       *XAR7,loc16           ; ProgSpace[*XAR7] = [loc16]
```

Syntax	Description

*XAR7++	22BitProgAddr(21:0) = XAR7,
	if(16-bit operation) XAR7 = XAR7 + 1,
	if(32-bit operation) XAR7 = XAR7 + 2
	Note: If instruction is repeated, the address is post−incremented as normal.

```
Instructions that use this addressing mode:
MAC          P,loc16,*XAR7++       ; ACC = ACC + P << PM,
                                   ; P = [loc16] * ProgSpace[*XAR7++]
DMAC         ACC:P,loc32,*XAR7++   ; ACC = ([loc32].MSW * ProgSpace[*XAR7++].MSW) >> PM,
                                   ; P = ([loc32].LSW * ProgSpace[*XAR7].MSW) >> PM
QMACL        P,loc32,*XAR7++       ; ACC = ACC + P >> PM,
                                   ; P = ([loc32] * ProgSpace[*XAR7++] >> 32
IMACL        P,loc32,*XAR7++       ; ACC = ACC + P,
                                   ; P = ([loc32] * ProgSpace[*XAR7++] << PM
```

5.10 Byte Addressing Modes

Syntax	Description

*+XARn[AR0]
*+XARn[AR1]
*+XARn[3bit]
	`32BitDataAddr(31:0) = XARn + Offset (Offset = AR0/AR1/3bit)`
	`if(offset == Even Value)`
	`  Access LSByte Of 16-bit Memory Location;`
	`  Leave    MSByte untouched;`
	`if( Offset == Odd Value )`
	`  Access MSByte Of 16-bit Memory Location;`
	`  Leave    LSByte untouched;`
	Note: For all other addressing modes, only the LSByte of the addressed location is accessed, the MSByte is left untouched.

```
Instructions that use this addressing mode:
MOVB    AX.LSB,loc16    ; if( address mode == *+XARn[AR0/AR1/3bit] )
;    if( offset == even )
;    AX.LSB = [loc16].LSB;
;    AX.MSB = 0x00;
;    if( offset == odd )
;    AX.LSB = [loc16].MSB;
;    AX.MSB = 0x00;
; else
;    AX.LSB = [loc16].LSB;
;    AX.MSB = 0x00;
MOVB    AX.MSB,loc16    ; if( address mode == *+XARn[AR0/AR1/3bit] )
;    if( offset == even )
;    AX.LSB = untouched;
;    AX.MSB = [loc16].LSB;
;    if( offset == odd )
;    AX.LSB = untouched;
;    AX.MSB = [loc16].MSB;
; else
;    AX.LSB = untouched;
;    AX.MSB = [loc16].LSB;
MOVB    loc16,AX.LSB    ; if( address mode == *+XARn[AR0/AR1/3bit] )
;    if( offset == even )
;    [loc16].LSB = AX.LSB
;    [loc16].MSB = untouched;
;    if( offset == odd )
;    [loc16].LSB = untouched;
;    [loc16].MSB = AX.LSB;
; else
;    [loc16].LSB = AX.LSB;
;    [loc16].MSB = untouched;
MOVB    loc16,AX.MSB    ; if( address mode == *+XARn[AR0/AR1/3bit] )
;    if( offset == even )
;    [loc16].LSB = AX.MSB
;    [loc16].MSB = untouched;
;    if( offset == odd )
;    [loc16].LSB = untouched;
;    [loc16].MSB = AX.MSB;
; else
;    [loc16].LSB = AX.MSB;
;    [loc16].MSB = untouched;
```

5.11 Alignment of 32-Bit Operations

All 32-bit reads and writes to memory are aligned at the memory interface to an even address boundary with the least significant word of the 32-bit data aligned to the even address. The output of the address generation unit does not force alignment, hence pointer values retain their values.

For example:

```
MOVB    AR0,#5 ; AR0 = 5
MOVL    *AR0,ACC    ; AL –> address 0x000004
        ; AH –> address 0x000005
        ; AR0 = 5
```

The programmer must take the above into account when generating addresses that are not aligned to an even boundary.

The 32-bit operands are stored in the following order; low order bits, 0 to 15, followed by the high order bits, 16 to 31, on the next highest 16-bit address increment (little-endian format).

Chapter 6

SPRU430F–August 2001–Revised April 2015

C28x Assembly Language Instructions

This chapter presents summaries of the instruction set, defines special symbols and notations used, and describes each instruction in detail in alphabetical order.

6.1 Summary of Instructions

The instructions are listed alphabetically, preceded by a summary.

Table 6-1. Summary of Instructions

Title **Page**

Table 6-1. Summary of Instructions (continued)

Table 6-1. Summary of Instructions (continued)

Table 6-1. Summary of Instructions (continued)

Table 6-1. Summary of Instructions (continued)

Table 6-1. Summary of Instructions (continued)

Table 6-1. Summary of Instructions (continued)

6.2 C28x Assembly Language Instructions by Function

NOTE: The examples in this chapter assume that the device is already operating in C28x Mode (Objmode = = 1, AMODE = = 0). To put the device into C28x mode following a reset, you must first set the Objmode bit in ST1 by executing the "C28OBJ"M (or "SETC Objmode"M) instruction.

NOTE: Cycle Counts assume the instruction is executed from zero-wait (single-cycle) memory and there are no pipeline stalls.

Table 6-2. Instruction Set Summary (Organized by Function)

Symbol	Description
XARn	XAR0 to XAR7 registers
ARn, ARm	Lower 16-bits of XAR0 to XAR7 registers
ARnH	Upper 16-bits of XAR0 to XAR7 registers
ARPn	3-bit auxiliary register pointer, ARP0 to ARP7 ARP0 points to XAR0 and ARP7 points to XAR7
AR(ARP)	Lower 16-bits of auxiliary register pointed to by ARP
XAR(ARP)	Auxiliary registers pointed to by ARP
AX	Accumulator high (AH) and low (AL) registers
#	Immediate operand
PM	Product shift mode (+4,1,0,-1,-2,-3,-4,-5,-6) PC Program counter
~	Bitwise compliment
[loc16]	Contents of 16-bit location
0:[loc16]	Contents of 16-bit location, zero extended
S:[loc16]	Contents of 16-bit location, sign extended
[loc32]	Contents of 32-bit location
0:[loc32]	Contents of 32-bit location, zero extended
S:[loc32]	Contents of 32-bit location, sign extended
7bit	7-bit immediate value
0:7bit	7-bit immediate value, zero extended
S:7bit	7-bit immediate value, sign extended
8bit	8-bit immediate value
0:8bit	8-bit immediate value, zero extended
S:8bit	8-bit immediate value, sign extended
10bit	10-bit immediate value
0:10bit	10-bit immediate value, zero extended
16bit	16-bit immediate value
0:16bit	16-bit immediate value, zero extended
S:16bit	16-bit immediate value, sign extended
22bit	22-bit immediate value
0:22bit	22-bit immediate value, zero extended
LSb	Least Significant bit
LSB	Least Significant Byte
LSW	Least Significant Word
MSB	Most Significant Byte
MSb	Most Significant bit
MSW	Most Significant Word
OBJ	Objmode bit state for which instruction is valid

Table 6-2. Instruction Set Summary (Organized by Function) (continued)

Symbol	Description
N	Repeat count (N = 0,1,2,3,4,5,6,7,....)
{ }	Optional field
=	Assignment
==	Equivalent to

6.3 Register Operations

NOTE: The examples in this chapter assume that the device is already operating in C28x Mode (Objmode == 1, AMODE == 0). To put the device into C28x mode following a reset, you must first set the Objmode bit in ST1 by executing the "C28OBJ"M (or "SETC Objmode"M) instruction.

NOTE: Cycle Counts assume the instruction is executed from zero-wait (single-cycle) memory and there are no pipeline stalls.

Table 6-3. Register Operations

Mnemonic		Description
XARn Register Operations (XAR0-XAR7)		
ADDB	XARn,#7bit	Add 7-bit constant to auxiliary register
ADRK	#8bit	Add 8-bit constant to current auxiliary register
CMPR	0/1/2/3	Compare auxiliary registers
MOV	AR6/7,loc16	Load auxiliary register
MOV	loc16,ARn	Store 16-bit auxiliary register
MOV	XARn,PC	Save the current program counter
MOVB	AR6/7,#8bit	Load auxiliary register with an 8-bit constant
MOVB	XARn,#8bit	Load auxiliary register with 8-bit value
MOVL	loc32,XARn	Store 32-bit auxiliary register
MOVL	XARn,loc32	Load 32-bit auxiliary register
MOVL	XARn,#22bit	Load 32-bit auxiliary register with constant value
MOVZ	ARn,loc16	Load lower half of XARn and clear upper half
SBRK	#8bit	Subtract 8-bit constant from current auxiliary register
SUBB	XARn,#7bit	Subtract 7-bit constant from auxiliary register
DP Register Operations		
MOV	DP,#10bit	Load data-page pointer
MOVW	DP,#16bit	Load the entire data page
MOVZ	DP,#10bit	Load data page and clear high bits
SP Register Operations		
ADDB	SP,#7bit	Add 7-bit constant to stack pointer
POP	ACC	Pop ACC register from stack
POP	AR1:AR0	Pop AR1 & AR0 registers from stack
POP	AR1H:AR0H	Pop AR1H & AR0H registers from stack
POP	AR3:AR2	Pop AR3 & AR2 registers from stack
POP	AR5:AR4	Pop AR5 & AR4 registers from stack
POP	DBGIER	Pop DBGIER register from stack
POP	DP:ST1	Pop DP & ST1 registers on stack
POP	DP	Pop DP register from stack
POP	IFR	Pop IFR register from stack
POP	loc16	Pop â€M loc16â€M data from stack
POP	P	Pop P register from stack
POP	RPC	Pop RPC register from stack
POP	ST0	Pop ST0 register from stack
POP	ST1	Pop ST1 register from stack
POP	T:ST0	Pop T & ST0 registers from stack
POP	XT	Pop XT register from stack

Table 6-3. Register Operations (continued)

Mnemonic		Description
POP	XARn	Pop auxiliary register from stack
PUSH	ACC	Push ACC register on stack
PUSH	ARn:ARn	Push ARn & ARn registers on stack
PUSH	AR1H:AR0H	Push AR1H & AR0H registers on stack
PUSH	DBGIER	Push DBGIER register on stack
PUSH	DP:ST1	Push DP & ST1 registers on stack
PUSH	DP	Push DP register on stack
PUSH	IFR	Push IFR register on stack
PUSH	loc16	Push â€M loc16â€M data on stack
PUSH	P	Push P register on stack
PUSH	RPC	Push RPC register on stack
PUSH	ST0	Push ST0 register on stack
PUSH	ST1	Push ST1 register on stack
PUSH	T:ST0	Push T & ST0 registers on stack
PUSH	XT	Push XT register on stack
PUSH	XARn	Push auxiliary register on stack
SUBB	SP,#7bit	Subtract 7-bit constant from the stack pointer
AX Register Operations (AH, AL)		
ADD	AX,loc16	Add value to AX
ADD	loc16,AX	Add AX to specified location
ADDB	AX,#8bit	Add 8-bit constant to AX
AND	AX,loc16,#16bit	Bitwise AND
AND	AX,loc16	Bitwise AND
AND	loc16,AX	Bitwise AND
ANDB	AX,#8bit	Bitwise AND 8-bit value
ASR	AX,1..16	Arithmetic shift right
ASR	AX,T	Arithmetic shift right by T(3:0) = 0...15
CMP	AX,loc16	Compare
CMPB	AX,#8bit	Compare 8-bit value
FLIP	AX	Flip order of bits in AX register
LSL	AX,1..16	Logical shift left
LSL	AX,T	Logical shift left by T(3:0) = 0...15
LSR	AX,1..16	Logical shift right
LSR	AX,T	Logical shift right by T(3:0) = 0..15
MAX	AX,loc16	Find the maximum
MIN	AX,loc16	Find the minimum
MOV	AX,loc16	Load AX
MOV	loc16,AX	Store AX
MOV	loc16,AX,COND	Store AX register conditionally
MOVB	AX,#8bit	Load AX with 8-bit constant
MOVB	AX.LSB,loc16	Load LSB of AX reg, MSB = 0x00
MOVB	AX.MSB,loc16	Load MSB of AX reg, LSB = unchanged
MOVB	loc16,AX.LSB	Store LSB of AX reg
MOVB	loc16,AX.MSB	Store MSB of AX reg
NEG	AX	Negate AX register
NOT	AX	Complement AX register
OR	AX,loc16	Bitwise OR

Table 6-3. Register Operations (continued)

Mnemonic		Description
OR	loc16,AX	Bitwise OR
ORB	AX,#8bit	Bitwise OR 8-bit value
SUB	AX,loc16	Subtract specified location from AX
SUB	loc16,AX	Subtract AX from specified location
SUBR	loc16,AX	Reverse-subtract specified location from AX
SXTB	AX	Sign extend LSB of AX reg into MSB
XOR	AX,loc16	Bitwise exclusive OR
XORB	AX,#8bit	Bitwise exclusive OR 8-bit value
XOR	loc16,AX	Bitwise exclusive OR
16-Bit ACC Register Operations		
ADD	ACC,loc16 {<< 0..16}	Add value to accumulator
ADD	ACC,#16bit {<< 0..15}	Add value to accumulator
ADD	ACC,loc16 << T	Add shifted value to accumulator
ADDB	ACC,#8bit	Add 8-bit constant to accumulator
ADDCU	ACC,loc16	Add unsigned value plus carry to accumulator
ADDU	ACC,loc16	Add unsigned value to accumulator
AND	ACC,loc16	Bitwise AND
AND	ACC,#16bit{<< 0..16}	Bitwise AND
MOV	ACC,loc16 {<< 0..16}	Load accumulator with shift
MOV	ACC,#16bit {<< 0..15}	Load accumulator with shift
MOV	loc16,ACC << 1..8	Save low word of shifted accumulator
MOV	ACC,loc16 << T	Load accumulator with shift
MOVB	ACC,#8bit	Load accumulator with 8-bit value
MOVH	loc16,ACC << 1..8	Save high word of shifted accumulator
MOVU	ACC,loc16	Load accumulator with unsigned word
SUB	ACC,loc16 << T	Subtract shifted value from accumulator
SUB	ACC,loc16 {<< 0..16}	Subtract shifted value from accumulator
SUB	ACC,#16bit {<< 0..15}	Subtract shifted value from accumulator
SUBB	ACC,#8bit	Subtract 8-bit value
SBBU	ACC,loc16	Subtract unsigned value plus inverse borrow
SUBU	ACC,loc16	Subtract unsigned 16-bit value
OR	ACC,loc16	Bitwise OR
OR	ACC,#16bit {<< 0..16}	Bitwise OR
XOR	ACC,loc16	Bitwise exclusive OR
XOR	ACC,#16bit {<< 0..16}	Bitwise exclusive OR
ZALR	ACC,loc16	Zero AL and load AH with rounding
32-Bit ACC Register Operations		
ABS	ACC	Absolute value of accumulator
ABSTC	ACC	Absolute value of accumulator and load TC
ADDL	ACC,loc32	Add 32-bit value to accumulator
ADDL	loc32,ACC	Add accumulator to specified location
ADDCL	ACC,loc32	Add 32-bit value plus carry to accumulator
ADDUL	ACC,loc32	Add 32-bit unsigned value to accumulator
ADDL	ACC,P << PM	Add shifted P to accumulator
ASRL	ACC,T	Arithmetic shift right of accumulator by T(4:0)
CMPL	ACC,loc32	Compare 32-bit value
CMPL	ACC,P << PM	Compare 32-bit value

Table 6-3. Register Operations (continued)

Mnemonic		Description
CSB	ACC	Count sign bits
LSL	ACC,1..16	Logical shift left 1 to 16 places
LSL	ACC,T	Logical shift left by T(3:0) = 0...15
LSRL	ACC,T	Logical shift right by T(4:0)
LSLL	ACC,T	Logical shift left by T(4:0)
MAXL	ACC,loc32	Find the 32-bit maximum
MINL	ACC,loc32	Find the 32-bit minimum
MOVL	ACC,loc32	Load accumulator with 32 bits
MOVL	loc32,ACC	Store 32-bit accumulator
MOVL	P,ACC	Load P from the accumulator
MOVL	ACC,P << PM	Load the accumulator with shifted P
MOVL	loc32,ACC,COND	Store ACC conditionally
NORM	ACC,XARn++/--	Normalize ACC and modify selected auxiliary register.
NORM	ACC,*ind	C2XLP compatible Normalize ACC operation
NEG	ACC	Negate ACC
NEGTC	ACC	If TC is equivalent to 1, negate ACC
NOT	ACC	Complement ACC
ROL	ACC	Rotate ACC left
ROR	ACC	Rotate ACC right
SAT	ACC	Saturate ACC based on OVC value
SFR	ACC,1..16	Shift accumulator right by 1 to 16 places
SFR	ACC,T	Shift accumulator right by T(3:0) = 0...15
SUBBL	ACC,loc32	Subtract 32-bit value plus inverse borrow
SUBCU	ACC,loc16	Subtract conditional 16-bit value
SUBCUL	ACC,loc32	Subtract conditional 32-bit value
SUBL	ACC,loc32	Subtract 32-bit value
SUBL	loc32,ACC	Subtract 32-bit value
SUBL	ACC,P << PM	Subtract 32-bit value
SUBRL	loc32,ACC	Reverse-subtract specified location from ACC
SUBUL	ACC,loc32	Subtract unsigned 32-bit value
TEST	ACC	Test for accumulator equal to zero
64-Bit ACC:P Register Operations		
ASR64	ACC:P,#1..16	Arithmetic shift right of 64-bit value
ASR64	ACC:P,T	Arithmetic shift right of 64-bit value by T(5:0)
CMP64	ACC:P	Compare 64-bit value
LSL64	ACC:P,1..16	Logical shift left 1 to 16 places
LSL64	ACC:P,T	64-bit logical shift left by T(5:0)
LSR64	ACC:P,#1..16	64-bit logical shift right by 1 to 16 places
LSR64	ACC:P,T	64-bit logical shift right by T(5:0)
NEG64	ACC:P	Negate ACC:P
SAT64	ACC:P	Saturate ACC:P based on OVC value
P or XT Register Operations (P, PH, PL, XT, T, TL)		
ADDUL	P,loc32	Add 32-bit unsigned value to P
MAXCUL	P,loc32	Conditionally find the unsigned maximum
MINCUL	P,loc32	Conditionally find the unsigned minimum
MOV	PH,loc16	Load the high half of the P register
MOV	PL,loc16	Load the low half of the P register

Table 6-3. Register Operations (continued)

Mnemonic		Description
MOV	loc16,P	Store lower half of shifted P register
MOV	T,loc16	Load the upper half of the XT register
MOV	loc16,T	Store the T register
MOV	TL,#0	Clear the lower half of the XT register
MOVA	T,loc16	Load the T register and add the previous product
MOVAD	T,loc16	Load T register
MOVDL	XT,loc32	Store XT and load new XT
MOVH	loc16,P	Save the high word of the P register
MOVL	P,loc32	Load the P register
MOVL	loc32,P	Store the P register
MOVL	XT,loc32	Load the XT register
MOVL	loc32,XT	Store the XT register
MOVP	T,loc16	Load the T register and store P in the accumulator
MOVS	T,loc16	Load T and subtract P from the accumulator
MOVX	TL,loc16	Load lower half of XT with sign extension
SUBUL	P,loc32	Subtract unsigned 32-bit value
16x16 Multiply Operations		
DMAC	ACC:P,loc32,*XAR7/++	16-bit dual multiply and accumulate
MAC	P,loc16,0:pma	Multiply and accumulate
MAC	P,loc16,*XAR7/++	Multiply and Accumulate
MPY	P,T,loc16	16 X 16 multiply
MPY	P,loc16,#16bit	16 X 16-bit multiply
MPY	ACC,T,loc16	16 X 16-bit multiply
MPY	ACC,loc16,#16bit	16 X 16-bit multiply
MPYA	P,loc16,#16bit	16 X 16-bit multiply and add previous product
MPYA	P,T,loc16	16 X 16-bit multiply and add previous product
MPYB	P,T,#8bit	Multiply signed value by unsigned 8-bit constant
MPYS	P,T,loc16	16 X 16-bit multiply and subtract
MPYB	ACC,T,#8bit	Multiply by 8-bit constant
MPYU	ACC,T,loc16	16 X 16-bit unsigned multiply
MPYU	P,T,loc16	Unsigned 16 X 16 multiply
MPYXU	P,T,loc16	Multiply signed value by unsigned value
MPYXU	ACC,T,loc16	Multiply signed value by unsigned value
SQRA	loc16	Square value and add P to accumulator
SQRS	loc16	Square value and subtract from accumulator
XMAC	P,loc16,*(pma)	C2xLP source-compatible multiply and accumulate
XMACD	P,loc16,*(pma)	C2xLP source-compatible multiply and accumulate with data move
32x32 Multiply Operations		
IMACL	P,loc32,*XAR7/++	Signed 32 X 32-bit multiply and accumulate (lower half)
IMPYAL	P,XT,loc32	Signed 32-bit multiply (lower half) and add previous P
IMPYL	P,XT,loc32	Signed 32 X 32-bit multiply (lower half)
IMPYL	ACC,XT,loc32	Signed 32 X 32-bit multiply (lower half)
IMPYSL	P,XT,loc32	Signed 32-bit multiply (lower half) and subtract P
IMPYXUL	P,XT,loc32	Signed 32 X unsigned 32-bit multiply (lower half)
QMACL	P,loc32,*XAR7/++	Signed 32 X 32-bit multiply and accumulate (upper half)
QMPYAL	P,XT,loc32	Signed 32-bit multiply (upper half) and add previous P
QMPYL	ACC,XT,loc32	Signed 32 X 32-bit multiply (upper half)

Table 6-3. Register Operations (continued)

Mnemonic		Description
QMPYL	P,XT,loc32	Signed 32 X 32-bit multiply (upper half)
QMPYSL	P,XT,loc32	Signed 32-bit multiply (upper half) and subtract previous P
QMPYUL	P,XT,loc32	Unsigned 32 X 32-bit multiply (upper half)
QMPYXUL	P,XT,loc32	Signed 32 X unsigned 32-bit multiply (upper half)
Direct Memory Operations		
ADD	loc16,#16bitSigned	Add constant to specified location
AND	loc16,#16bitSigned	Bitwise AND
CMP	loc16,#16bitSigned	Compare
DEC	loc16	Decrement by 1
DMOV	loc16	Data move contents of 16-bit location
INC	loc16	Increment by 1
MOV	*(0:16bit),loc16	Move value
MOV	loc16,*(0:16bit)	Move value
MOV	loc16,#16bit	Save 16-bit constant
MOV	loc16,#0	Clear 16-bit location
MOVB	loc16,#8bit,COND	Store byte conditionally
OR	loc16,#16bit	Bitwise OR
TBIT	loc16,#bit	Test bit
TBIT	loc16,T	Test bit specified by T register
TCLR	loc16,#bit	Test and clear specified bit
TSET	loc16,#bit	Test and set specified bit
XOR	loc16,#16bit	Bitwise exclusive OR
IO Space Operations		
IN	loc16,*(PA)	Input data from port
OUT	*(PA),loc16	Output data to port
UOUT	*(PA),loc16	Unprotected output data to I/O port
Program Space Operations		
PREAD	loc16,*XAR7	Read from program memory
PWRITE	*XAR7,loc16	Write to program memory
XPREAD	loc16,*AL	C2xLP source-compatible program read
XPREAD	loc16,*(pma)	C2xLP source-compatible program read
XPWRITE	*AL,loc16	C2xLP source-compatible program write
Branch/Call/Return Operations		
B	16bitOff,COND	Conditional branch
BANZ	16bitOff,ARn--	Branch if auxiliary register not equal to zero
BAR	16bOf,ARn,ARn,EQ/NEQ	Branch on auxiliary register comparison
BF	16bitOff,COND	Branch fast
FFC	XAR7,22bitAddr	Fast function call
IRET		Interrupt return
LB	22bitAddr	Long branch
LB	*XAR7	Long indirect branch
LC	22bitAddr	Long call immediate
LC	*XAR7	Long indirect call
LCR	22bitAddr	Long call using RPC
LCR	*XARn	Long indirect call using RPC
LOOPZ	loc16,#16bit	Loop while zero
LOOPNZ	loc16,#16bit	Loop while not zero

Table 6-3. Register Operations (continued)

Mnemonic		Description
LRET		Long return
LRETE		Long return and enable interrupts
LRETR		Long return using RPC
RPT	#8bit/loc16	Repeat next instruction
SB	8bitOff,COND	Short conditional branch
SBF	8bitOff,EQ/NEQ/TC/NTC	Short fast conditional branch
XB	pma	C2XLP source-compatible branch
XB	pma,COND	C2XLP source-compatible conditional branch
XB	pma,*,ARPn	C2XLP source-compatible branch function call
XB	*AL	C2XLP source-compatible function call
XBANZ	pma,*ind{,ARPn}	C2XLP source-compatible branch if ARn is not zero
XCALL	pma	C2XLP source-compatible call
XCALL	pma,COND	C2XLP source-compatible conditional call
XCALL	pma,*,ARPn	C2XLP source-compatible call with ARP modification
XCALL	*AL	C2XLP source-compatible indirect call
XRET		Alias for XRETC UNC
XRETC	COND	C2XLP source-compatible conditional return
Interrupt Register Operations		
AND	IER,#16bit	Bitwise AND to disable specified CPU interrupts
AND	IFR,#16bit	Bitwise AND to clear pending CPU interrupts
IACK	#16bit	Interrupt acknowledge
INTR	INT1/../INT14 NMI EMUINT DLOGINT RTOSINT	Emulate hardware interrupts
MOV	IER,loc16	Load the interrupt-enable register
MOV	loc16,IER	Store interrupt enable register
OR	IER,#16bit	Bitwise OR
OR	IFR,#16bit	Bitwise OR
TRAP	#0..31	Software trap
Status Register Operations (ST0, ST1)		
CLRC	Mode	Clear status bits
CLRC	XF	Clear the XF status bit and output signal
CLRC	AMODE	Clear the AMODE bit
C28ADDR		Clear the AMODE status bit
CLRC	Objmode	Clear the Objmode bit
C27OBJ		Clear the Objmode bit
CLRC	M0M1MAP	Clear the M0M1MAP bit
C27MAP		Set the M0M1MAP bit
CLRC	OVC	Clear OVC bits
ZAP	OVC	Clear overflow counter
DINT		Disable maskable interrupts (set INTM bit)
EINT		Enable maskable interrupt (clear INTM bit)
MOV	PM,AX	Load product shift mode bits PM = AX(2:0)
MOV	OVC,loc16	Load the overflow counter
MOVU	OVC,loc16	Load overflow counter with unsigned value
MOV	loc16,OVC	Store the overflow counter
MOVU	loc16,OVC	Store the unsigned overflow counter

Table 6-3. Register Operations (continued)

Mnemonic		Description
SETC	Mode	Set multiple status bits
SETC	XF	Set XF bit and output signal
SETC	M0M1MAP	Set the M0M1MAP bit
C28MAP		Set M0M1MAP bit
SETC	Objmode	Set Objmode bit
C28OBJ		Set the Objmode bit
SETC	AMODE	Set AMODE bit
LPADDR		Alias for SETC AMODE
SPM	PM	Set product shift mode bits
Miscellaneous Operations		
ABORTI		Abort interrupt
ASP		Align stack pointer
EALLOW		Enable access to protected space
IDLE		Put processor in IDLE mode
NASP		Un-align stack pointer
NOP	{*ind}	No operation with optional indirect address modification
ZAPA		Zero accumulator P register and OVC
EDIS		Disable access to protected space
ESTOP0		Emulation Stop 0
ESTOP1		Emulation Stop 1

ABORTI *Abort Interrupt*

Syntax Options ABORTI

Opcode `0000 0000 0000 0001`

Objmode X

RPT –

CYC 2

Operands None

Description This instruction is available for emulation purposes. Generally, a program uses the IRET instruction to return from an interrupt. The IRET instruction restores all of the values that were saved to the stack during the automatic context save. In restoring status register ST1 and the debug status register (DBGSTAT), IRET restores the debug context that was present before the interrupt.

In some target applications, you might have interrupts that must not be returned from by the IRET instruction. Not using IRET can cause a problem for the emulation logic, because the emulation logic assumes that the original debug context will be restored. The abort interrupt (ABORTI) instruction is provided as a means to indicate that the debug context will not be restored and the debug logic needs to be reset to its default state. As part of its operation, the ABORTI instruction:

- Sets the DBGM bit in ST1. This disables debug events.
- Modifies select bits in the DBGSTAT register. This effect is a resetting of the debug context. If the CPU was in the debug-halt state before the interrupt occurred, the CPU does not halt when the interrupt is aborted.

The ABORTI instruction does not modify the DBGIER, the IER, the INTM bit or any analysis registers (for example, registers used for breakpoints, watch points, and data logging).

Flags and Modes

Flags and Modes	Description
DBGM	The DBGM bit is set.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

ABS ACC *Absolute Value of Accumulator*

Syntax Options ABS ACC

Opcode `1111 1111 0101 0110`

Objmode X

RPT –

CYC 1

Operands **ACC** – Accumulator register

Description The content of the ACC register is replaced with its absolute value:

```
if(ACC = 0x8000 0000)
  V = 1;
    If (OVM = 1)
     ACC = 0x7FFF FFFF;
    else
     ACC = 0x8000 0000;
 else
    if(ACC < 0)
     ACC = −ACC;
```

Flags and Modes

Flags and Modes	Description
N	After the operation, the N flag is set if bit 31 of the ACC is 1, else N is cleared.
Z	After the operation, the Z flag is set if the ACC is zero, else Z is cleared.
C	C is cleared by this operation.
V	If (ACC = 0x8000 0000) at the start of the operation, this is considered an overflow value and V is set. Otherwise, V is not affected.
OVM	If (ACC = 0x8000 0000) at the start of the operation, this is considered an overflow value, and the ACC value after the operation depends on the state of OVM: If OVM is cleared, ACC will be filled with 0x8000 0000. If OVM is set ACC will be saturated to 0x7FFF FFFF.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Take absolute value of VarA, make sure value is saturated:
MOVL ACC,@VarA     ; Load ACC with contents of VarA
SETC OVM           ; Turn overflow mode on
ABS ACC            ; Absolute of ACC and saturate
MOVL @VarA,ACC     ; Store result into VarA
```

ABSTC ACC *Absolute Value of Accumulator and Load TC*

Syntax Options ABSTC ACC

Opcode `0101 0110 0101 1111`

Objmode 1

RPT –

CYC 1

Operands **ACC** – Accumulator register

Description Replace the content of the ACC register with its absolute value and load the test control (TC) bit with the sign bit XORed with the previous value of the test control bit:

```
if(ACC = 0x8000 0000)
  {
  If (OVM = 1)
    ACC = 0x7FFF FFFF;
  else
    ACC = 0x8000 0000;
  V = 1;
  TC = TC XOR 1;
  {
else
  {
  if(ACC < 0)
    ACC = −ACC;
    TC = TC XOR 1;
  }
C = 0;
```

Flags and Modes

Flags and Modes	Description
N	After the operation, the N flag is set if bit 31 of the ACC is 1, else N is cleared.
Z	After the operation, the Z flag is set if the ACC is zero, else Z is cleared.
C	The C flag bit is cleared.
V	If (ACC = 0x8000 0000) at the start of the operation, this is considered an overflow value and V is set; otherwise, V is not affected.
TC	If ACC < 0) at the start of the operation, then TC = TC XOR 1; otherwise, TC is not affected.
OVM	If at the start of the operation, ACC = 0x8000 0000, then this is considered an overflow value and the ACC value after the operation depends on OVM. If OVM is cleared and TC == 1, ACC will be filled with 0x8000 0000. If OVM is set and TC = 1, ACC will be saturated to 0x7FFF FFFF.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Calculate signed: Quot16 = Num16/Den16, Rem16 = Num16%Den16
CLRC TC               ; Clear TC flag, used as sign flag
MOV ACC,@Den16 << 16 ; AH = Den16, AL = 0
ABSTC ACC             ; Take abs value, TC = sign ^ TC
MOV T,@AH             ; Temp save Den16 in T register
MOV ACC,@Num16 << 16 ; AH = Num16, AL = 0
ABSTC ACC             ; Take abs value, TC = sign ^ TC
MOVU ACC,@AH          ; AH = 0, AL = Num16
RPT #15               ; Repeat operation 16 times
||SUBCU ACC,@T        ; Conditional subtract with Den16
MOV @Rem16,AH         ; Store remainder in Rem16
MOV ACC,@AL << 16     ; AH = Quot16, AL = 0
NEGTC ACC             ; Negate if TC = 1
MOV @Quot16,AH        ; Store quotient in Quot16
```

ADD ACC,#16bit<<#0..15 *Add Value to Accumulator*

Syntax Options ADD ACC,#16bit<<#0..15

Opcode

```
1111 1111 0001 SHFT
CCCC CCCC CCCC CCCC
```

Objmode X

RPT –

CYC 1

Operands

ACC – Accumulator register

#16bit – 16-bit immediate constant value

#0..15 – Shift value (default is "<<#0" if no value specified)

Description

Add the left shifted 16-bit immediate constant value to the ACC register. The shifted value is sign extended if sign extension mode is turned on (SXM = 1) else the shifted value is zero extended (SXM = 0). The lower bits of the shifted value are zero filled:

```
if(SXM = 1) // sign extension mode enabled
  ACC = ACC + S:16bit << shift value;
else        // sign extension mode disabled
  ACC = ACC + 0:16bit << shift value;
```

Smart Encoding:

If #16bit is an 8-bit number and the shift is 0, then the assembler will encode this instruction as ADDB ACC, #8bit to improve efficiency. To override this encoding, use the ADDW ACC, #16bit instruction alias.

Flags and Modes

Flags and Modes	Description
Z	After the addition, the Z flag is set if the ACC value is zero, else the flag is cleared.
N	After the addition, the N flag is set if bit 31 of the ACC is 1, else the flag is cleared.
C	If the addition generates a carry, C is set; otherwise C is cleared.
V	If an overflow occurs, V is set; otherwise V is not affected.
OVC	If (OVM = 0, disabled) then if the operation generates a positive overflow, then the counter is incremented and if the operation generates a negative overflow, then the counter is decremented. If (OVM = 1, enabled) then the counter is not affected by the operation.
SXM	If sign extension mode bit is set; then the 16-bit immediate constant will be sign-extended before the addition. Else, the value will be zero extended.
OVM	If overflow mode bit is set; then the ACC value will saturate maximum positive (0x7FFFFFFF) or maximum negative (0x80000000) if the operation overflowed.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Calculate signed value: ACC = (VarB << 10) + (23 << 6);
SETC SXM             ; Turn sign extension mode on
MOV ACC,@VarB << #10 ; Load ACC with VarB left shifted by 10
ADD ACC,#23 << #6    ; Add 23 left shifted by 6 to ACC
```

ADD ACC,loc16 <<T *Add Value to Accumulator*

Syntax Options ADD ACC,loc16 <<T

Opcode

```
0101 0110 0010 0011
0000 0000 LLLL LLLL
```

Objmode 1

RPT –

CYC N+1

Operands

ACC – Accumulator register

loc16 – Addressing mode (see Chapter 5)

T – Upper 16 bits of the multiplicand register, XT(31:16)

Description

Add to the ACC register the left-shifted contents of the 16-bit location pointed to by the “loc16” addressing mode. The shift value is specified by the four least significant bits of the T register, T(3:0) = shift value = 0..15. Higher order bits of T are ignored. The shifted value is sign extended if sign extension mode is turned on (SXM = 1) else the shifted value is zero extended (SXM = 0). The lower bits of the shifted value are zero filled:

```
if(SXM = 1) // sign extension mode enabled
  ACC = ACC + S:[loc16] << T(3:0);
else       // sign extension mode disabled
  ACC = ACC + 0:[loc16] << T(3:0);
```

Flags and Modes

Flags and Modes	Description
Z	After the addition, the Z flag is set if the ACC value is zero, else Z is cleared.
N	After the addition, the N flag is set if bit 31 of the ACC is 1, else N is cleared.
C	If the addition generates a carry, C is set; otherwise C is cleared.
V	If an overflow occurs, V is set; otherwise V is not affected.
OVC	If OVM = 0, disabled and the operation generates a positive overflow, then the counter is incremented; if the operation generates a negative overflow, then the counter is decremented. If OVM = 1, enabled, then the counter is not affected by the operation.
SXM	If sign extension mode bit is set; then the 16-bit operand, addressed by the ”loc16” field, will be sign extended before the addition. Else, the value will be zero extended.
OVM	If overflow mode bit is set; then the ACC value will saturate maximum positive (0x7FFFFFFF) or maximum negative (0x80000000) if the operation overflowed.

Repeat

If this operation is repeated, then the instruction will be executed N+1 times. The state of the Z, N, C flags will reflect the final result. The V flag will be set if an intermediate overflow occurs. The OVC flag will count intermediate overflows, if overflow mode is disabled.

Example

```
; Calculate signed value: ACC = (VarA << SB) + (VarB << SB)
SETC SXM           ; Turn sign extension mode on
MOV T,@SA          ; Load T with shift value in SA
MOV ACC,@VarA << T ; Load in ACC shifted contents of VarA
MOV T,@SB          ; Load T with shift value in SB
ADD ACC,@VarB << T ; Add to ACC shifted contents of VarB
```

ADD ACC,loc16 << #0..16 *Add Value to Accumulator*

Syntax Options

Syntax Options	Opcode	Objmode	RPT	CYC
ADD ACC,loc16<<#0	`1000 0001 LLLL LLLL`	1	Y	N+1

ADD ACC,loc16 << #1..15	`0101 0110 0000 0100`
	`0000 SHFT LLLL LLLL`
		1	Y	N+1

ADD ACC,loc16 << #16	`0000 0101 LLLL LLLL`	X	Y	N+1
ADD ACC,loc16<<0...15	`1010 SHFT LLLL LLLL`	0	–	N+1

Operands

ACC – Accumulator register

loc16 – Addressing mode (see Chapter 5)

#0..16 – Shift value (default is "<<#0" if no value specified)

Description

Add the left shifted 16-bit location pointed to by the ”loc16” addressing mode to the ACC register. The shifted value is sign extended if sign extension mode is turned on (SXM = 1) else the shifted value is zero extended (SXM = 0). The lower bits of the shifted value are zero-filled:

```
if(SXM = 1) // sign extension mode enabled
  ACC = ACC + S:[loc16] << shift value;
else        // sign extension mode disabled
  ACC = ACC + 0:[loc16] << shift value;
```

Flags and Modes

Flags and Modes	Description
Z	After the addition, the Z flag is set if ACC is zero, else Z is cleared.
N	After the addition, the N flag is set if bit 31 of the ACC is 1, else N is cleared.
C	If the addition generates a carry, C is set; otherwise C is cleared. **Exception**: If a shift of 16 is used, the ADD instruction can set C but not clear C.
V	If an overflow occurs, V is set; otherwise V is not affected.
OVC	If (OVM = 0, disabled) then if the operation generates a positive overflow, then the counter is incremented and if the operation generates a negative overflow, then the counter is decremented. If (OVM = 1, enabled) then the counter is not affected by the operation.
SXM	If sign extension mode bit is set; then the 16-bit operand, addressed by the ”loc16” field, will be sign extended before the addition. Else, the value will be zero extended.
OVM	If overflow mode bit is set; then the ACC value will saturate maximum positive (0x7FFFFFFF) or maximum negative (0x80000000) if the operation overflowed.

Repeat

If the operation is repeatable, then the instruction will be executed N+1 times. The state of the Z, N, C flags will reflect the final result. The V flag will be set if an intermediate overflow occurs. The OVC flag will count intermediate overflows, if overflow mode is disabled. If the operation is not repeatable, the instruction will execute only once.

Example

```
; Calculate signed value: ACC = VarA << 10 + VarB << 6;
SETC SXM             ; Turn sign extension mode on
MOV ACC,@VarA << #10 ; Load ACC with VarA left shifted by 10
ADD ACC,@VarB << #6  ; Add VarB left shifted by 6 to ACC
```

ADD AX, loc16 *Add Value to AX*

Syntax Options ADD AX, loc16

Opcode `1001 010A LLLL LLLL`

Objmode X

RPT –

CYC 1

Operands **AX** – Accumulator high (AH) or accumulator low (AL) register

loc16 – Addressing mode (see Chapter 5)

Description Add the contents of the location pointed to by the “loc16” addressing mode to the specified AX register (AH or AL) and store the result in the AX register.

Flags and Modes

Flags and Modes	Description
N	After the addition, AX is tested for a negative condition. If bit 15 of AX is 1, then the negative flag bit is set, otherwise it is cleared.
Z	After the addition, AX is tested for a zero condition. The zero flag bit is set if the operation results in AX = 0; otherwise it is cleared.
C	If the addition generates a carry, C is set; otherwise, C is cleared.
V	If an overflow occurs, V is set; otherwise V is not affected. Signed positive overflow occurs if the result crosses the max positive value (0x7FFF) in the positive direction. Signed negative overflow occurs if the result crosses the max negative value (0x8000) in the negative direction.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Add the contents of VarA with VarB and store in VarC
MOV    AL,@VarA    ;    Load AL with contents of VarA
ADD    AL,@VarB    ;    Add to AL contents of VarB
MOV    @VarC,AL    ;    Store result in VarC
```

ADD loc16, AX *Add AX to Specified Location*

Syntax Options ADD loc16, AX

Opcode `0111 001A LLLL LLLL`

Objmode X

RPT –

CYC 1

Operands **loc16** – Addressing mode (see Chapter 5)

AX – Accumulator high (AH) or accumulator low (AL) register

Description Add the contents of the specified AX register (AH or AL) to the location pointed to by the “loc16” addressing mode and store the results in location pointed to by “loc16”:

```
[loc16] = [loc16] + AX;
```

This is a read-modify-write operation.

Flags and Modes

Flags and Modes	Description
N	After the addition, [loc16] is tested for a negative condition. If bit 15 of [loc16] is 1, then the negative flag bit is set, otherwise it is cleared.
Z	After the addition, [loc16] is tested for a zero condition. The zero flag bit is set if the operation generates [loc16] = 0; otherwise it is cleared
C	If the addition generates a carry, C is set; otherwise C is cleared.
V	If an overflow occurs, V is set; otherwise V is not affected. Signed positive overflow occurs if the result crosses the max positive value (0x7FFF) in the positive direction. Signed negative overflow occurs if the result crosses the max negative value (0x8000) in the negative direction.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Add the contents of VarA to index register AR0:
MOV AL,@VarA  ; Load AL with contents of VarA
ADD @AR0,AL   ; AR0 = AR0 + A
; Add the contents of VarB to VarC:
MOV AH,@VarB  ; Load AH with contents of VarB
ADD @VarC,AH  ; VarC = VarC + AH
```

ADD loc16,#16bitSigned *Add Constant to Specified Location*

Syntax Options ADD loc16,#16bitSigned

Opcode

```
0000 1000 LLLL LLLL
CCCC CCCC CCCC CCCC
```

Objmode X

RPT –

CYC 1

Operands

loc16 – Addressing mode (see Chapter 5)

#16bit-signed – 16-bit immediate signed constant value

Description

Add the specified signed 16-bit immediate constant to the signed 16-bit content of the location pointed to by the “loc16” addressing mode and store the 16-bit result in the location pointed to by “loc16”:

```
[loc16] = [loc16] + 16bitSigned;
```

Smart Encoding:

If loc16 = AL or AH and #16bitSigned is an 8-bit number then the assembler will encode this instruction as ADDB AX, #16bitSigned to improve efficiency. To override this encoding, use the ADDW loc16, #16bitSigned instruction alias.

Flags and Modes

Flags and Modes	Description
N	After the addition, if bit 15 of [loc16] is 1, then the N bit is set; else N cleared.
Z	After the addition, if [loc16] is zero, the Z is set, else Z is cleared.
C	If the addition generates a carry, C is set; otherwise, C is cleared.
V	If an overflow occurs, V is set; otherwise, V is cleared.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Calculate:
; VarA = VarA + 10
; VarB = VarB − 3
ADD    @VarA,#10    ; VarA = VarA + 10
ADD    @VarB,#−3    ; VarB = VarB − 3
```

ADDB ACC,#8bit *Add 8-bit Constant to Accumulator*

Syntax Options ADDB ACC,#8bit

Opcode `0000 1001 CCCC CCCC`

Objmode X

RPT –

CYC 1

Operands **ACC** – Accumulator register

#8bit – 8-bit immediate unsigned constant value

Description Add an 8-bit, zero-extended constant to the ACC register:

```
ACC = ACC + 0:8bit;
```

Flags and Modes

Flags and Modes	Description
Z	After the addition, the Z flag is set if ACC is zero, else Z is cleared.
N	After the addition, the N flag is set if bit 31 of the ACC is 1, else N is cleared.
C	If the addition generates a carry, C is set; otherwise C is cleared.
V	If an overflow occurs, V is set; otherwise V is not affected.
OVC	If (OVM = 0, disabled) then if the operation generates a positive overflow, then the counter is incremented and if the operation generates a negative overflow, then the counter is decremented. If (OVM = 1, enabled) then the counter is not affected by the operation.
OVM	If overflow mode bit is set; then the ACC value will saturate maximum positive (0x7FFFFFFF) or maximum negative (0x80000000) if the operation overflowed.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Increment contents of 32-bit location VarA:
MOVL ACC,@VarA   ; Load ACC with contents of VarA
ADDB ACC,#1      ; Add 1 to ACC
MOVL @VarA,ACC   ; Store result back into VarA
```

ADDB AX, #8bitSigned *Add 8-bit Constant to AX*

Syntax Options ADDB AX, #8bitSigned

Opcode `1001 110A CCCC CCCC`

Objmode X

RPT –

CYC 1

Operands

AX – Accumulator high (AH) or accumulator low (AL) register

#8bit-signed – 8-bit immediate signed 2s complement constant value (-128 to 127)

Description

Add the sign extended 8-bit constant to the specified AX register (AH or AL) and store the result in the AX register:

```
AX = AX + S:8bit;
```

Flags and Modes

Flags and Modes	Description
N	After the addition, AX is tested for a negative condition. If bit 15 of AX is 1, then the negative flag bit is set; otherwise it is cleared.
Z	After the addition, AX is tested for a zero condition. The zero flag bit is set if the operation results in AX = 0, otherwise it is cleared.
C	If the addition generates a carry, C is set; otherwise C is cleared.
V	If an overflow occurs, V is set; otherwise V is not affected. Signed positive overflow occurs if the result crosses the max positive value (0x7FFF) in the positive direction. Signed negative overflow occurs if the result crosses the max negative value (0x8000) in the negative direction.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Add 2 to VarA and subtract 3 from VarB:

MOV AL,@VarA ; Load AL with contents of VarA
ADDB AL,#2   ; Add to AL the value 0x0002 (2)
MOV @VarA,AL ; Store result in VarA
MOV AL,@VarB ; Load AL with contents of VarB
ADDB AL,#−3  ; Add to AL the value 0xFFFD (−3)
MOV @VarB,AL ; Store result in VarB
```

ADDB SP, #7bit *Add 7-bit Constant to Stack Pointer*

Syntax Options ADDB SP, #7bit

Opcode `1111 1110 0CCC CCCC`

Objmode X

RPT –

CYC 1

Operands **SP** – Stack pointer

#7bit – 7-bit immediate unsigned constant value

Description Add a 7-bit unsigned constant to SP and store the result in SP:

```
SP = SP + 0:7bit;
```

Flags and Modes None

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
FuncA:                  ; Function with local variables on stack.
    ADDB SP, #N         ; Reserve N 16-bit words of space for
                        ; local variables on stack:

.
.
.
SUBB SP, #N             ; Deallocate reserved stack space.
LRETR                   ; Return from function.
```

ADDB XARn, #7bit *Add 7-bit Constant to Auxiliary Register*

Syntax Options ADDB XARn, #7bit

Opcode

```
1101 1nnn 0CCC CCCC
```

Objmode X

RPT –

CYC 1

Operands **XARn** – XAR0−XAR7, 32-bit auxiliary registers

Description Add a 7-bit unsigned constant to XARn and store the result in XARn:

```
XARn = XARn + 0:7bit;
```

Flags and Modes None

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
MOVL XAR1,#VarA  ; Initialize XAR1 pointer with address
                 ; of VarA

MOVL XAR2,*XAR1  ; Load XAR2 with contents of VarA
ADDB XAR2,#10h   ; XAR2 = VarA + 0x10
```

ADDCL ACC,loc32 *Add 32-bit Value Plus Carry to Accumulator*

Syntax Options ADDCL ACC,loc32

Opcode

```
0101 0110 0100 0000
xxxx xxxx LLLL LLLL
```

Objmode 1

RPT –

CYC 1

Operands **ACC** – Accumulator register

loc32 – Addressing mode (see Chapter 5)

Description Add to the ACC register the 32-bit content of the location pointed to by the “loc32” addressing mode:

```
ACC = ACC + [loc32] + C;
```

Flags and Modes

Flags and Modes	Description
Z	After the addition, the Z flag is set if the ACC is zero, else Z is cleared.
N	After the addition, the N flag is set if bit 31 of the ACC is 1, else N is cleared.
C	The state of the carry bit before execution is included in the addition. If the addition generates a carry, C is set; otherwise C is cleared.
V	If an overflow occurs, V is set; otherwise V is not affected.
OVC	If (OVM = 0, disabled) then if the operation generates a positive overflow, then the counter is incremented and if the operation generates a negative overflow, then the counter is decremented. If (OVM = 1, enabled) then the counter is not affected by the operation.
OVM	If overflow mode bit is set; then the ACC value will saturate maximum positive (0x7FFFFFFF) or maximum negative (0x80000000) if the operation overflows.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Add two 64-bit values (VarA and VarB) and store result in VarC:
MOVL ACC,@VarA+0    ; Load ACC with contents of the low
                    ; 32 bits of VarA
ADDUL ACC,@VarB+0   ; Add to ACC the contents of the low
                    ; 32 bits of VarB
MOVL @VarC+0,ACC    ; Store low 32-bit result into VarC
MOVL ACC,@VarA+2    ; Load ACC with contents of the high
                    ; 32 bits of VarA
ADDCL ACC,@VarB+2   ; Add to ACC the contents of the high
                    ; 32 bits of VarB with carry
MOVL @VarC+2,ACC    ; Store high 32-bit result into VarC
```

ADDCU ACC,loc16 *Add Unsigned Value Plus Carry to Accumulator*

Syntax Options ADDCU ACC,loc16

Opcode `0000 1100 LLLL LLLL`

Objmode X

RPT –

CYC 1

Operands

ACC – Accumulator register

loc16 – Addressing mode (see Chapter 5)

Description

Add the 16-bit contents of the location pointed to by the “loc16” addressing mode, zero extended, plus the content of the carry flag bit to the ACC register:

```
ACC = ACC + 0:[loc16] + C;
```

Flags and Modes

Flags and Modes	Description
Z	After the addition, the Z flag is set if the ACC value is zero, else Z is cleared.
N	After the addition, the N flag is set if bit 31 of the ACC is 1, else N is cleared.
C	The state of the carry bit before execution is included in the addition. If the addition generates a carry, C is set; otherwise C is cleared.
V	If an overflow occurs, V is set; otherwise V is not affected.
OVC	If (OVM = 0, disabled) then if the operation generates a positive overflow, then the counter is incremented and if the operation generates a negative overflow, then the counter is decremented. If (OVM = 1, enabled) then the counter is not affected by the operation.
OVM	If overflow mode bit is set; then the ACC value will saturate maximum positive (0x7FFFFFFF) or maximum negative (0x80000000) if the operation overflowed.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Add three 32-bit unsigned variables by 16-bit parts:
MOVU ACC,@VarAlow         ; AH = 0, AL = VarAlow
ADD ACC,@VarAhigh << 16   ; AH = VarAhigh, AL = VarAlow
ADDU ACC,@VarBlow         ; ACC = ACC + 0:VarBlow
ADD ACC,@VarBhigh << 16   ; ACC = ACC + VarBhigh  << 16
ADDCU ACC,@VarClow        ; ACC = ACC + VarClow + Carry
ADD ACC,@VarChigh << 16   ; ACC = ACC + VarChigh << 16
```

ADDL ACC,loc32 *Add 32-bit Value to Accumulator*

Syntax Options ADDL ACC,loc32

Opcode `0000 0111 LLLL LLLL`

Objmode X

RPT Y

CYC N+1

Operands **ACC** – Accumulator register

loc32 – Addressing mode (see Chapter 5)

Description Add to the ACC register the 32-bit content of the location pointed to by the “loc32” addressing mode:

```
ACC = ACC + [loc32];
```

Flags and Modes

Flags and Modes	Description
N	After the addition, the N flag is set if bit 31 of the ACC is 1, else N is cleared.
Z	After the addition, the Z flag is set if the ACC is zero, else Z is cleared.
C	If the addition generates a carry, C is set; otherwise C is cleared.
V	If an overflow occurs, V is set; otherwise V is not affected.
OVC	If (OVM = 0, disabled) then if the operation generates a positive overflow, then the counter is incremented and if the operation generates a negative overflow, then the counter is decremented. If (OVM = 1, enabled) then the counter is not affected by the operation.
OVM	If overflow mode bit is set; then the ACC value will saturate maximum positive (0x7FFFFFFF) or maximum negative (0x80000000) if the operation overflows.

Repeat If this operation is repeated, then the instruction will be executed N+1 times. The state of the Z, N, C flags will reflect the final result. The V flag will be set if an intermediate overflow occurs. The OVC flag will count intermediate overflows, if overflow mode is disabled.

Example

```
; Calculate the 32-bit value: VarC = VarA + VarB
MOVL ACC,@VarA    ; Load ACC with contents of VarA
ADDL ACC,@VarB    ; Add to ACC the contents of VarB
MOVL @VarC,ACC    ; Store result into VarC
```

ADDL ACC,P << PM *Add Shifted P to Accumulator*

Syntax Options ADDL ACC,P << PM

Opcode `0001 0000 1010 1100`

Objmode X

RPT Y

CYC N+1

Note: This instruction is an alias for the ”MOVA T,loc16” operation with “loc16 = @T” addressing mode.

Operands

ACC – Accumulator register

P – Product register

<< PM – Product shift mode

Description Add to the ACC register the contents of the P register, shifted as specified by the product shift mode (PM):

```
ACC = ACC + P << PM
```

Flags and Modes

Flags and Modes	Description
Z	After the addition, the N flag is set if bit 31 of the ACC is 1, else N is cleared.
N	After the addition, the N flag is set if bit 31 of the ACC is 1, else N is cleared.
C	If the addition generates a carry, C is set; otherwise C is cleared.
V	If an overflow occurs, V is set; otherwise V is not affected.
OCV	If (OVM = 0, disabled) then if the operation generates a positive overflow, then the counter is incremented and if the operation generates a negative overflow, then the counter is decremented. If (OVM = 1, enabled) then the counter is not affected by the operation.
OVM	If overflow mode bit is set; then the ACC value will saturate maximum positive (0x7FFFFFFF) or maximum negative (0x80000000) if the operation overflowed.
PM	The value in the PM bits sets the shift mode for the output operation from the product register. If the product shift value is positive (logical left shift operation), then the low bits are zero filled. If the product shift value is negative (arithmetic right shift operation), the upper bits are sign extended.

Repeat If this operation is repeated, then the instruction will be executed N+1 times. The state of the Z, N, C flags will reflect the final result. The V flag will be set if an intermediate overflow occurs. The OVC flag will count intermediate overflows if overflow mode is disabled.

Example

```
; Calculate: Y = ((M*X >> 4) + (B << 11)) >> 10
; Y, M, X, B are Q15 values
SPM −4           ; Set product shift to >> 4
SETC SXM         ; Enable sign extension mode
MOV T,@M         ; T = M
MPY P,T,@X       ; P = M * X
MOV ACC,@B << 11 ; ACC = S:B << 11
ADDL ACC,P << PM ; ACC = (M*X >>4) + (S:B << 11)
MOVH @Y,ACC << 5 ; Store Q15 result into Y
```

ADDL loc32,ACC *Add Accumulator to Specified Location*

Syntax Options ADDL loc32,ACC

Opcode

```
0101 0110 0000 0001
0000 0000 LLLL LLLL
```

Objmode 1

RPT –

CYC 1

Operands

loc32 – Addressing mode (see Chapter 5)

ACC – Accumulator register

Description Add to the ACC register the 32-bit content of the location pointed to by the “loc32” addressing mode:

```
[loc32] = [loc32] + ACC;
```

This is a read-modify-write operation.

Flags and Modes

Flags and Modes	Description
N	After the addition, the N flag is set if bit 31 of the ACC is 1, else N is cleared.
Z	After the addition, the Z flag is set if the ACC is zero, else Z is cleared.
C	If the addition generates a carry, C is set; otherwise C is cleared.
V	If an overflow occurs, V is set; otherwise V is not affected.
OCV	If (OVM = 0, disabled) then if the operation generates a positive overflow, then the counter is incremented and if the operation generates a negative overflow, then the counter is decremented. If (OVM = 1, enabled) then the counter is not affected by the operation.
OVM	If overflow mode bit is set, the ACC value will saturate maximum positive (0x7FFFFFFF) or maximum negative (0x80000000) if the operation overflows.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Increment the 32-bit value VarA:
MOVB ACC,#1     ; Load ACC with 0x00000001
ADDL @VarA,ACC  ; VarA = VarA + ACC
```

ADDU ACC,loc16 *Add Unsigned Value to Accumulator*

Syntax Options ADDU ACC,loc16

Opcode `0000 1101 LLLL LLLL`

Objmode X

RPT Y

CYC N+1

Operands **ACC** – Accumulator register

loc16 – Addressing mode (see Chapter 5)

Description Add the 16-bit contents of the location pointed to by the “loc16” addressing mode to the ACC register. The addressed location is zero extended before the add:

```
ACC = ACC + 0:[loc16];
```

Flags and Modes

Flags and Modes	Description
Z	After the addition, the Z flag is set if ACC is zero, else Z is cleared.
N	After the addition, the N flag is set if bit 31 of the ACC is 1, else N is cleared.
C	If the addition generates a carry, C is set; otherwise C is cleared.
V	If an overflow occurs, V is set; otherwise V is not affected.
OVC	If (OVM = 0, disabled) then if the operation generates a positive overflow, then the counter is incremented and if the operation generates a negative overflow, then the counter is decremented. If (OVM = 1, enabled) then the counter is not affected by the operation.
OVM	If overflow mode bit is set; then the ACC value will saturate maximum positive (0x7FFFFFFF) or maximum negative (0x80000000) if the operation overflowed.

Repeat If this operation is repeated, then the instruction will be executed N+1 times. The state of the Z, N, C flags will reflect the final result. The V flag will be set if an intermediate overflow occurs. The OVC flag will count intermediate overflows, if overflow mode is disabled.

Example

```
; Add three 32-bit unsigned variables by 16-bit parts:
MOVU ACC,@VarAlow        ; AH = 0, AL = VarAlow
ADD ACC,@VarAhigh << 16  ; AH = VarAhigh, AL = VarAlow
ADDU ACC,@VarBlow        ; ACC = ACC + 0:VarBlow
ADD ACC,@VarBhigh << 16  ; ACC = ACC + VarBhigh << 16
ADDCU ACC,@VarClow       ; ACC = ACC + VarClow + Carry
ADD ACC,@VarChigh << 16  ; ACC = ACC + VarChigh << 16
```

ADDUL P,loc32 *Add 32-bit Unsigned Value to P*

Syntax Options ADDUL P,loc32

Opcode

```
0101 0110 0101 0111
0000 0000 LLLL LLLL
```

Objmode 1

RPT –

CYC 1

Operands

P – Product register

loc32 – Addressing mode (see Chapter 5)

Description

Add to the P register the 32-bit content of the location pointed to by the “loc32” addressing mode. The addition is treated as an unsigned ADD operation:

```
P = P + [loc32];    // unsigned add
```

Note: The difference between a signed and unsigned 32-bit add is in the treatment of the overflow counter (OVC). For a signed ADD, the OVC counter monitors positive/negative overflow. For an unsigned ADD, the OVC unsigned (OVCU) counter monitors the carry.

Flags and Modes

Flags and Modes	Description
N	After the addition, if bit 31 of the P register is 1, then set the N flag; otherwise clear N.
Z	After the addition, if the value of the P register is 0, then set the Z flag; otherwise clear Z.
C	If the addition generates a carry, set C; otherwise C is cleared.
V	If an overflow occurs, V is set; otherwise V is not affected.
OVCU	The overflow counter is incremented when the addition operation generates an unsigned carry. The OVM mode does not affect the OVCU counter.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Add 64-bit VarA + VarB and store result in VarC:
MOVL P,@VarA+0      ; Load P with low 32 bits of VarA MOVL
ACC,@VarA+2         ; Load ACC with high 32 bits of VarA
ADDUL P,@VarB+0     ; Add to P unsigned low 32 bits of VarB
ADDCL ACC,@VarB+2   ; Add to ACC with carry high 32 bits of VarB
MOVL @VarC+0,P      ; Store low 32-bit result into VarC
MOVL @VarC+2,ACC    ; Store high 32-bit result into VarC
```

ADDUL ACC, loc32 *Add 32-bit Unsigned Value to Accumulator*

Syntax Options ADDUL ACC, loc32

Opcode

```
0101 0110 0101 0011
xxxx xxxx LLLL LLLL
```

Objmode 1

RPT Y

CYC N+1

Operands **ACC** – Accumulator register

loc32 – Addressing mode (see Chapter 5)

Description Add to the ACC register the unsigned 32-bit content of the location pointed to by the "loc32" addressing mode:

```
ACC = ACC + [loc32];    // unsigned add
```

Note: The difference between a signed and unsigned 32-bit add is in the treatment of the overflow counter (OVC). For a signed ADD, the OVC counter monitors positive/negative overflow. For an unsigned ADD, the OVC unsigned (OVCU) counter monitors the carry.

Flags and Modes

Flags and Modes	Description
Z	After the addition, the Z flag is set if the ACC value is zero, else Z is cleared.
N	After the addition, the N flag is set if bit 31 of the ACC is 1, else N is cleared.
C	If the addition generates a carry, C is set; otherwise C is cleared.
V	If an overflow occurs, V is set; otherwise V is not affected.
OVCU	The overflow counter is incremented when the addition operation generates an unsigned carry. The OVM mode does not affect the OVCU counter.

Repeat If this operation is repeated, then the instruction will be executed N+1 times. The state of the Z, N, C flags will reflect the final result. The V flag will be set if an intermediate overflow occurs. The OVCU will count intermediate carries.

Example

```
; Add two 64-bit values (VarA and VarB) and store result in VarC:
MOVL ACC,@VarA+0   ; Load ACC with contents of the low
                   ; 32 bits of VarA
ADDUL ACC,@VarB+0  ; Add to ACC the contents of the low
                   ; 32 bits of VarB
MOVL @VarC+0,ACC   ; Store low 32-bit result into VarC
MOVL ACC,@VarA+2   ; Load ACC with contents of the high
                   ; 32 bits of VarA
ADDCL ACC,@VarB+2  ; Add to ACC the contents of the high
                   ; 32 bits of VarB with carry
MOVL @VarC+2,ACC   ; Store high 32-bit result into VarC
```

ADRK #8bit *Add to Current Auxiliary Register*

Syntax Options ADRK #8bit

Opcode

```
1111 1100 IIII IIII
```

Objmode X

RPT –

CYC 1

Operands **#8bit** – 8-bit immediate constant value

Description Add the 8-bit unsigned constant to the XARn register pointed to by ARP:

```
XAR(ARP) = XAR(ARP) + 0:8bit;
```

Flags and Modes

Flags and Modes	Description
ARP	The 3-bit ARP points to the current valid Auxiliary Register, XAR0 to XAR7. This pointer determines which Auxiliary register is modified by the operation.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once

Example

```
TableA: .word 0x1111
    .word 0x2222
    .word 0x3333
    .word 0x4444

FuncA:
    MOVL XAR1,#TableA   ; Initialize XAR1 pointer
    MOVZ AR2,*XAR1      ; Load AR2 with the 16-bit value
                        ; pointed to by XAR1 (0x1111)
                        ; Set ARP = 1

ADRK #2                 ; Increment XAR1 by 2
MOVZ AR3,*XAR1          ; Load AR3 with the 16-bit value
                        ; pointed to by XAR1 (0x3333)
```

AND ACC,#16bit << #0..16 *Description*

Syntax Options

Syntax Options	Opcode	Objmode	RPT	CYC

AND ACC, #16bit << #0..15	`0011 1110 0000 SHFT`
	`CCCC CCCC CCCC CCCC`
		1	-	1

AND ACC, #16bit << #16	`0101 0110 0000 1000`
	`CCCC CCCC CCCC CCCC`
		1	-	1

Operands

ACC – Accumulator register

#16bit – 16-bit immediate constant value

#0..16 – Shift value (default is "<< #0" if no value specified)

Description

Perform a bitwise AND operation on the ACC register with the given 16-bit unsigned constant value left shifted as specified. The value is zero extended and lower order bits are zero filled before the AND operation. The result is stored in the ACC register:

```
ACC = ACC AND (0:16bit << shift value);
```

Flags and Modes

Flags and Modes	Description
N	The load to ACC is tested for a negative condition. If bit 31 of ACC is 1, then the negative flag bit is set; otherwise it is cleared.
Z	The load to ACC is tested for a zero condition. The zero flag bit is set if the operation generates ACC = 0; otherwise it is cleared.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Calculate the 32-bit value: VarA = VarA AND 0x0FFFF000
MOVL ACC,@VarA         ; Load ACC with contents of VarA
AND ACC,#0xFFFF << 12  ; AND ACC with 0x0FFFF000
MOVL @VarA,ACC         ; Store result in VarA
```

AND ACC, loc16 *Bitwise AND*

Syntax Options AND ACC, loc16

Opcode `1000 1001 LLLL LLLL`

Objmode 1

RPT Y

CYC N+1

Operands **ACC** – Accumulator register

loc16 – Addressing mode (see Chapter 5)

Description Perform a bitwise AND operation on the ACC register with the zero-extended content of the location pointed to by the “loc16” address mode. The result is stored in the ACC register:

```
ACC = ACC AND 0:[loc16];
```

Flags and Modes

Flags and Modes	Description
N	Clear flag.
Z	The load to ACC is tested for a zero condition. The zero flag bit is set if the operation generates ACC = 0; otherwise it is cleared.

Repeat This operation is repeatable. If the operation follows a RPT instruction, then the AND instruction will be executed N+1 times. The state of the Z and N flags will reflect the final result.

Example

```
; Calculate the 32-bit value: VarA = VarA AND 0:VarB
MOVL ACC,@VarA  ; Load ACC with contents of VarA
AND ACC,@VarB   ; AND ACC with contents of 0:VarB
MOVL @VarA,ACC  ; Store result in VarA
```

AND AX, loc16, #16bit *Bitwise AND*

Syntax Options AND AX, loc16, #16bit

Opcode

```
1100 110A LLLL LLLL
CCCC CCCC CCCC CCCC
```

Objmode X

RPT –

CYC 1

Operands

AX – Accumulator high (AH) or accumulator low (AL) register

loc16 – Addressing mode (see Chapter 5)

#16bit – 16-bit immediate constant value

Description Perform a bitwise AND operation on the 16-bit contents of the location pointed to by the “loc16” addressing mode with the specified 16-bit immediate constant. The result is stored in the specified AX register:

```
AX = [loc16] AND 16bit;
```

Flags and Modes

Flags and Modes	Description
N	The load to AX is tested for a negative condition. If bit 15 of AX is 1, then the negative flag bit is set; otherwise it is cleared.
Z	The load to AX is tested for a zero condition. The zero flag bit is set if the operation generates AX = 0; otherwise it is cleared.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Branch if either of Bits 2 and 7 of VarA are non-zero:
AND AL,@VarA,#0x0084  ; AL = VarA AND 0x0084
SB Dest,NEQ           ; Branch if result is non-zero
; Merge Bits 0,1,2 of VarA with Bits 8,9,10 of VarB and store in
; VarC in bit locations 0,1,2,3,4,5:
AND AL,@VarA,#0x0007  ; Keep bits 0,1,2 of VarA
AND AH,@VarB,#0x0700  ; Keep bits 8,9,10 of VarB
LSR AH,#5             ; Scale back bits 8,9,10 to bits 3,4,5
OR AL,@AH             ; Merge bits
MOV @VarC,AL          ; Store result in VarC
```

AND IER,#16bit *Bitwise AND to Disable Specified CPU Interrupts*

Syntax Options

AND IER,#16bit

Opcode

```
0111 0110 0010 0110
CCCC CCCC CCCC CCCC
```

Objmode

X

RPT

–

CYC

2

Operands

IER – Interrupt enable register

#16bit – 16-bit immediate constant value (0x0000 to 0xFFFF)

Description

Disable specific interrupts by performing a bitwise AND operation with the IER register and the 16-bit immediate value. The result is stored in the IER register. Any changes take effect before the next instruction is processed.

```
IER = IER AND #16bit;
```

Flags and Modes

None

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Disable INT1 and INT6 only. Do not modify state of other
; interrupts enable:
AND IER,#0xFFDE  ; Disable INT1 and INT6
```

AND IFR,#16bit *Bitwise AND to Clear Pending CPU Interrupts*

Syntax Options AND IFR,#16bit

Opcode

```
0111 0110 0010 1111
CCCC CCCC CCCC CCCC
```

Objmode X

RPT –

CYC 2

Operands

IFR – Interrupt flag register

#16bit – 16-bit immediate constant value (0x0000 to 0xFFFF)

Description

Clear specific pending interrupts by performing a bitwise AND operation with the IFR register and the 16-bit immediate value. The result of the AND operation is stored in the IFR register:

```
IFR = IFR AND #16bit;
```

Note: Interrupt hardware has priority over CPU instruction operation in cases where the interrupt flag is being simultaneously modified by the hardware and the instruction.

Flags and Modes None

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Clear the contents of the IFR register. Disables all
; pending interrupts:
AND IFR,#0x0000  ; Clear IFR register
```

AND loc16, AX *Bitwise AND*

Syntax Options AND loc16, AX

Opcode `1100 000A LLLL LLLL`

Objmode X

RPT –

CYC 1

Operands **loc16** – Addressing mode (see Chapter 5)

AX – Accumulator high (AH) or accumulator low (AL) register

Description Perform a bitwise AND operation on the contents of the location pointed to by the “loc16” addressing mode with the specified AX register. The result is stored in location pointed to by ”loc16”:

```
[loc16] = [loc16] AND AX;
```

This is a read-modify-write operation.

Flags and Modes

Flags and Modes	Description
N	The load to [loc16] is tested for a negative condition. If bit 15 of [loc16] is 1, then the negative flag bit is set; otherwise it is cleared.
Z	The load to [loc16] is tested for a zero condition. The zero flag bit is set if the operation generates ([loc16] = 0); otherwise it is cleared.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; AND the contents of VarA with VarB and store in VarB:
MOV AL,@VarA  ; Load AL with contents of VarA
AND @VarB,AL  ; VarB = VarB AND AL
```

AND AX, loc16 *Bitwise AND*

Syntax Options AND AX, loc16

Opcode

```
1100 111A LLLL LLLL
```

Objmode X

RPT –

CYC 1

Operands

AX – Accumulator high (AH) or accumulator low (AL) register

loc16 – Addressing mode (see Chapter 5)

Description

Perform a bitwise AND operation on the contents of the specified AX register with the 16-bit contents of the location pointed to by the “loc16” addressing mode. The result is stored in the AX register:

```
AX = AX AND 16bit;
```

Flags and Modes

Flags and Modes	Description
N	The load to AX is tested for a negative condition. If bit 15 of AX is 1, then the negative flag bit is set; otherwise it is cleared.
Z	The load to AX is tested for a zero condition. The zero flag bit is set if the operation generates AX = 0; otherwise it is cleared.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; AND the contents of VarA and VarB and branch if non-zero:
MOV AL,@VarA  ; Load AL with contents of VarA
AND AL,@VarB  ; AND AL with contents of VarB
SB Dest,NEQ   ; Branch if result is non-zero
```

AND loc16,#16bitSigned *Bitwise AND*

Syntax Options

AND loc16,#16bitSigned

Opcode

```
0001 1000 LLLL LLLL
CCCC CCCC CCCC CCCC
```

Objmode

X

RPT

–

CYC

1

Operands

loc16 – Addressing mode (see Chapter 5)

#16bitsigned – 16-bit signed immediate constant value

Description

Perform a bitwise AND operation on the 16-bit content of the location pointed to by the “loc16” addressing mode and the specified 16-bit immediate constant. The result is stored in the location pointed to by “loc16”:

```
[loc16] = [loc16] AND 16bit;
```

Smart Encoding:

If loc16 = AH or AL and #16bitSigned is an 8-bit number, then the assembler will encode this instruction as ANDB AX, #8-bit to improve efficiency. To override this, use the ANDW AX, #16bitSigned instruction alias.

Flags and Modes

Flags and Modes	Description
N	After the operation if bit 15 of [loc16] 1, set N; otherwise, clear N.
Z	After the operation if [loc16] is zero, set Z; otherwise, clear Z.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Clear Bits 3 and 11 of VarA:
; VarA = VarA AND #~(1 << 3 | 1 << 11)
AND @VarA,#~(1 << 3 | 1  ; Clear bits 3 and 11 of VarA << 11)
```

ANDB AX, #8bit *Bitwise AND 8-bit Value*

Syntax Options ANDB AX, #8bit

Opcode `1001 000A CCCC CCCC`

Objmode X

RPT –

CYC 1

Operands **AX** – Accumulator high (AH) or accumulator low (AL) register

#8bit – 8-bit immediate constant value

Description Perform a bitwise AND operation with the content of the specified AX register (AH or AL) with the given 8-bit unsigned immediate constant zero extended. The result is stored in AX:

```
AX = AX AND 0:8bit;
```

Flags and Modes

Flags and Modes	Description
N	The load to AX is tested for a negative condition. If bit 15 of AX is 1, then the negative flag bit is set; otherwise it is cleared.
Z	The load to AX is tested for a zero condition. The zero flag bit is set if the operation generates AX = 0; otherwise it is cleared.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Add VarA to VarB, keep LSByte and store result in VarC:
MOV AL,@VarA    ; Load AL with contents of VarA
ADD AL,@VarB    ; Add to AL contents of VarB
ANDB AL,#0xFF   ; AND contents of AL with 0x00FF
MOV @VarC,AL    ; Store result in VarC
```

ASP *Align Stack Pointer*

Syntax Options ASP

Opcode `0111 0110 0001 1011`

Objmode X

RPT –

CYC 1

Operands None

Description Ensure that the stack pointer (SP) is aligned to an even address. If the least significant bit of SP is 1, SP points to an odd address and must be moved by incrementing SP by 1. The SPA bit is set as a record of this alignment. If instead the ASP instruction finds that the SP already points to an even address, SP is left unchanged and the SPA bit is cleared to indicate that no alignment has taken place. In either case, the change to the SPA bit is made in the decode 2 phase of the pipeline.

```
if(SP = odd)
  SP = SP + 1;
  SPA = 1;else
  SPA = 0;
```

To undo a previous alignment by the ASP instruction, use the NASP instruction.

Flags and Modes

Flags and Modes	Description
SPA	If SP holds an odd address before the operation, SPA is set; otherwise, SPA is cleared.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Alignment of stack pointer in interrupt service routine:
; Vector table:
INTx: .long INTxService  ; INTx interrupt vector
.
.

INTxService:
  ASP                    ; Align stack pointer
.
.
.
  NASP                   ; Re-align stack pointer
  IRET                   ; Return from interrupt.
```

ASR AX,#1...16 *Arithmetic Shift Right*

Syntax Options ASR AX,#1...16

Opcode `1111 1111 101A SHFT`

Objmode X

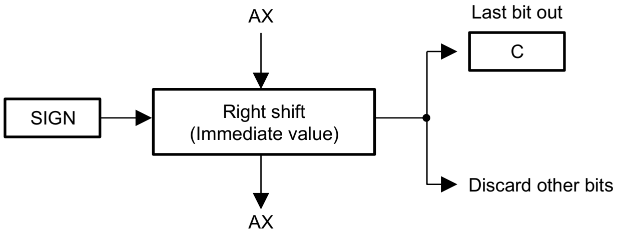
RPT –

CYC 1

Operands **AX** – Accumulator high (AH) or accumulator low (AL) register

#1-16 – Shift value

Description Perform an arithmetic right shift on the content of the specified AX register (AH or AL) by the amount given in the “shift value” field. During the shift, the value is sign extended and the last bit to be shifted out of the AX register is stored in the carry status flag bit:



Flags and Modes

Flags and Modes	Description
N	After the shift, if bit 15 of AX is 1 then the negative flag bit is set; otherwise it is cleared.
Z	After the shift, if AX is 0, then the Z bit is set; otherwise it is cleared.
C	The last bit to be shifted out of AH or AL is stored in C.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Calculate signed value: VarC = (VarA + VarB) >> 2
MOV AL,@VarA  ; Load AL with contents of VarA
ADD AL,@VarB  ; Add to AL contents of VarB
ASR AL,#2     ; Scale result by 2
MOV @VarC,AL  ; Store result in VarC
```

ASR AX,T *Arithmetic Shift Right*

Syntax Options ASR AX,T

Opcode `1111 1111 0110 010A`

Objmode X

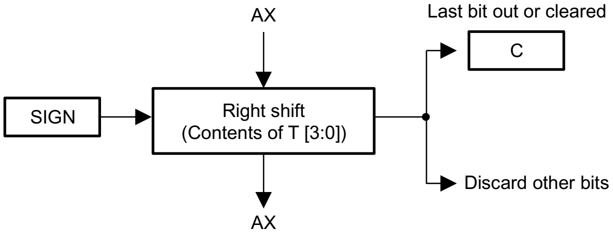
RPT –

CYC 1

Operands **AX** – Accumulator high (AH) or accumulator low (AL) register

T – Upper 16 bits of the multiplicand (XT) register

Description Perform an arithmetic shift right on the content of the specified AX register as specified by the four least significant bits of the T register, T(3:0) = shift value = 0…15. The contents of higher order bits are ignored. During the shift, the value is sign extended. If the T(3:0) register bits specify a shift of 0, then C is cleared; otherwise, C is filled with the last bit to be shifted out of AX:



Flags and Modes

Flags and Modes	Description
N	After the shift, if bit 15 of AX is 1 then the negative flag bit is set; otherwise it is cleared. Even if the T(3:0) register bits specify a shift of 0, the value of AH or AL is still tested for the negative condition and N is affected.
Z	After the shift, if AX is 0, then the Z bit is set, otherwise it is cleared. Even if the T(3:0) register bits specify a shift of 0, the value of AH or AL is still tested for the zero condition and Z is affected.
C	If T(3:0) specifies a shift of 0, then C is cleared; otherwise, C is filled with the last bit to be shifted out of AH or AL.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Calculate signed value: VarC = VarA >> VarB;
MOV T,@VarB   ; Load T with contents of VarB
MOV AL,@VarA  ; Load AL with contents of VarA
ASR AL,T      ; Scale AL by value in T bits 0 to 3
MOV @VarC,AL  ; Store result in VarC
```

ASR64 ACC:P,#1..16 *Arithmetic Shift Right of 64-bit Value*

Syntax Options ASR64 ACC:P,#1..16

Opcode `0101 0110 1000 SHFT`

Objmode 1

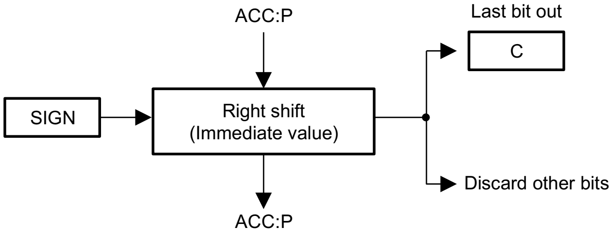
RPT –

CYC 1

Operands **ACC:P** – Accumulator register (ACC) and product register (P)

#1..16 – Shift value

Description Arithmetic shift right the 64-bit combined value of the ACC:P registers by the amount specified in the shift value field. As the value is shifted, the most significant bits are sign extended and the last bit shifted out is stored in the carry bit flag:



Flags and Modes

Flags and Modes	Description
N	After the shift, if bit 31 of the ACC register is 1 then ACC:P is negative and the N bit is set; otherwise N is cleared.
Z	After the shift, the Z flag is set if the combined 64-bit value of the ACC:P is zero; otherwise, Z is cleared.
C	The last bit shifted out of the combined 64-bit value is loaded into the C bit.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Arithmetic shift right the 64-bit Var64 by 10:
MOVL ACC,@Var64+2  ; Load ACC with high 32 bits of Var64
MOVL P,@Var64+0    ; Load P with low 32 bits of Var64
ASR64 ACC:P,#10    ; Arithmetic shift right ACC:P by 10
MOVL @Var64+2,ACC  ; Store high 32-bit result into Var64
MOVL @Var64+0,P    ; Store low 32-bit result into Var64
```

ASR64 ACC:P,T *Arithmetic Shift Right of 64-bit Value*

Syntax Options ASR64 ACC:P,T

Opcode `0101 0110 0010 1100`

Objmode 1

RPT –

CYC 1

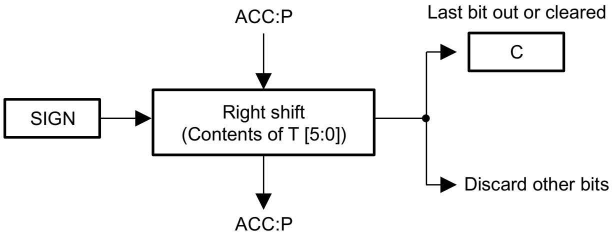
Operands

ACC:P – Accumulator register (ACC) and product register (P)

T – Upper 16 bits of the multiplicand register (XT)

Description

Arithmetic shift right the 64-bit combined value of the ACC:P registers by the amount specified in six least significant bits of the T register, T(5:0) = 0…63. Higher order bits are ignored. As the value is shifted, the most significant bits are sign extended. If T specifies a shift of 0, then C is cleared; otherwise, C is filled with the last bit to be shifted out of the ACC:P registers:



Flags and Modes

Flags and Modes	Description
N	After the shift, if bit 31 of the ACC register is 1 then ACC:P is negative and the N bit is set; otherwise N is cleared.
Z	After the shift, the Z flag is set if the combined 64-bit value of the ACC:P is zero; otherwise, Z is cleared.
C	If (T[5:0] = 0) clear C; otherwise, the last bit shifted out of the combined 64-bit value is loaded into the C bit.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Arithmetic shift right the 64-bit Var64 by contents of Var16:
MOVL ACC,@Var64+2  ; Load ACC with high 32 bits of Var64
MOVL P,@Var64+0    ; Load P with low 32 bits of Var64
MOV T,@Var16       ; Load T with shift value from Var16
ASR64 ACC:P,T      ; Arithmetic shift right ACC:P by T(5:0)
MOVL @Var64+2,ACC  ; Store high 32-bit result into Var64
MOVL @Var64+0,P    ; Store low 32-bit result into Var64
```

ASRL ACC,T *Arithmetic Shift Right of Accumulator*

Syntax Options ASRL ACC,T

Opcode `0101 0110 0001 0000`

Objmode 1

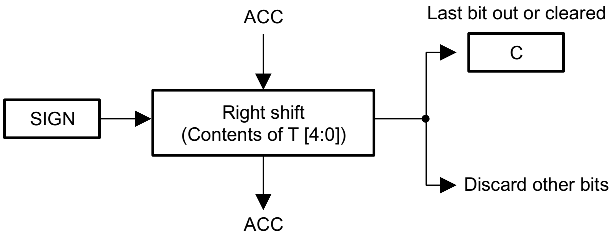
RPT –

CYC 1

Operands **ACC** – Accumulator register

T – Upper 16 bits of the multiplicand (XT) register

Description Perform an arithmetic shift right on the content of the ACC register as specified by the five least significant bits of the T register, T(4:0) = 0…31. Higher order bits are ignored. During the shift, the value is sign extended. If T specifies a shift of 0, then C is cleared; otherwise, C is filled with the last bit to be shifted out of the ACC register:



Flags and Modes

Flags and Modes	Description
Z	After the shift, the Z flag is set if the ACC value is zero, else Z is cleared. Even if the T register specifies a shift of 0, the content of the ACC register is still tested for the zero condition and Z is affected.
N	After the shift, the N flag is set if bit 31 of the ACC is 1, else N is cleared. Even if the T register specifies a shift of 0, the content of the ACC register is still tested for the negative condition and N is affected.
C	If (T(4:0) = 0) then C is cleared; otherwise, the last bit shifted out is loaded into the C flag bit.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Arithmetic shift right contents of VarA by VarB:
MOVL ACC,@VarA  ; ACC = VarA
MOV T,@VarB     ; T = VarB (shift value)
ASRL ACC,T      ; Arithmetic shift right ACC by T(4:0)
MOVL @VarA,ACC  ; Store result into VarA
```

B 16bitOffset,COND *Branch*

Syntax Options B 16bitOffset,COND

Opcode

```
1111 1111 1110 COND
CCCC CCCC CCCC CCCC
```

Objmode X

RPT –

CYC 7/4

Operands **16bit-offset** – 16-bit signed immediate constant offset value (−32768 to +32767 range)

COND – Conditional codes:

COND	Syntax	Description	Flags Tested
0000	NEQ	Not Equal To	Z = 0
0001	EQ	Equal To	Z = 1
0010	GT	Greater Than	Z = 0 AND N = 0
0011	GEQ	Greater Than or Equal To	N = 0
0100	LT	Less Than	N = 1
0101	LEQ	Less Than or Equal To	Z = 1 OR N = 1
0110	HI	Higher	C = 1 AND Z = 0
0111	HIS, C	Higher or Same, Carry Set	C = 1
1000	LO, NC	Lower, Carry Clear	C = 0
1001	LOS	Lower or Same	C = 0 OR Z = 1
1010	NOV	No Overflow	V = 0
1011	OV	Overflow	V = 1
1100	NTC	Test Bit Not Set	TC = 0
1101	TC	Test Bit Set	TC = 1
1110	NBIO	BIO Input Equal To Zero	BIO = 0
1111	UNC	Unconditional	–

Description Conditional branch. If the specified condition is true, then branch by adding the signed 16-bit constant value to the current PC value; otherwise continue execution without branching:

```
If (COND = true) PC = PC + signed 16-bit offset;
If (COND = false) PC = PC + 2;
```

Note: If (COND = true) then the instruction takes 7 cycles. If (COND = false) then the instruction takes 4 cycles.

Flags and Modes

Flags and Modes	Description
V	If the V flag is tested by the condition, then V is cleared.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

BANZ 16bitOffset,ARn− − *Branch if Auxiliary Register Not Equal to Zero*

Syntax Options BANZ 16bitOffset,ARn− −

Opcode

```
0000 0000 0000 1nnn
CCCC CCCC CCCC CCCC
```

Objmode X

RPT –

CYC 4/2

Operands

16bit-offset – 16-bit signed immediate constant value

ARn – Lower 16 bits of auxiliary registers XAR0 to XAR7

Description

If the 16-bit content of the specified auxiliary register is not equal to 0, then the 16-bit sign offset is added to the PC value. This forces program control to the new address (PC + 16bitOffset). The 16-bit offset is sign extended to 22 bits before the addition. Then, the content of the auxiliary register is decremented by 1. The upper 16 bits of the auxiliary register (ARnH) is not used in the comparison and is not affected by the post decrement:

```
if( ARn != 0 )
  PC = PC + signed 16-bit offset;
ARn = ARn - 1;
ARnH = unchanged;
```

Note: If branch is taken, then the instruction takes 4 cycles If branch is not taken, then the instruction takes 2 cycles

Flags and Modes None

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Copy the contents of Array1 to Array2:
; int32 Array1[N];
; int32 Array2[N];
; for(i=0; i < N; i++)
; Array2[i] = Array1[i];
  MOVL XAR2,#Array1  ; XAR2 = pointer to Array1
  MOVL XAR3,#Array2  ; XAR3 = pointer to Array2
  MOV @AR0,#(N−1)    ; Repeat loop N times
Loop:
  MOVL ACC,*XAR2++   ; ACC = Array1[i]
  MOVL *XAR3++,ACC   ; Array2[i] = ACC
  BANZ Loop,AR0−−    ; Loop if AR0 != 0, AR0−−
```

BAR 16bitOffset,ARn,ARm,EQ/NEQ *Branch on Auxiliary Register Comparison*

Syntax Options

Syntax Options	Opcode	Objmode	RPT	CYC

BAR 16bitOffset,ARn,ARm,EQ	`1000 1111 10nn nmmm`
	`CCCC CCCC CCCC CCCC`
		1	–	4/2

BAR 16bitOffset,ARn,ARm,NEQ	`1000 1111 11nn nmmm`
	`CCCC CCCC CCCC CCCC`
		1	–	4/2

Operands

16bit-offset – 16-bit signed immediate constant offset value (−32768 to +32767 range)

ARn – Lower 16 bits of auxiliary registers XAR0 to XAR7

ARm – Lower 16 bits of auxiliary registers XAR0 to XAR7

Syntax	Description	Condition Tested
NEQ	Not Equal To	ARn != ARm
EQ	Equal To	ARn = ARm

Description

Compare the 16-bit contents of the two auxiliary registers ARn and ARm registers and branch if the specified condition is true; otherwise continue execution without branching:

```
If (tested condition = true) PC = PC + signed 16-bit offset;
If (tested condition = false) PC = PC + 2;
```

Note: If (tested condition = true) then the instruction takes 4 cycles. If (tested condition = false) then the instruction takes 2 cycles.

Flags and Modes

None

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; String compare:
  MOVL XAR2,#StringA          ; XAR2 points to StringA
  MOVL XAR3,#StringB          ; XAR3 points to StringB
  MOV @AR4,#0                 ; AR4 = 0
Loop:
  MOVZ AR0,*XAR2++            ; AR0 = StringA[i]
  MOVZ AR1,*XAR3++            ; AR1 = StringB[i], i++
  BAR Exit,AR0,AR4,EQ         ; Exit if StringA[i] = 0
  BAR Loop,AR0,AR1,EQ         ; Loop if StringA[i] = StringB[i]
NotEqual:                     ; StringA and B not the same
.
Exit:                         ; StringA and B the same
```

BF 16bitOffset,COND *Branch Fast*

Syntax Options BF 16bitOffset,COND

Opcode

```
0101 0110 1100 COND
CCCC CCCC CCCC CCCC
```

Objmode 1

RPT –

CYC 4/4

Operands **16bit-offset** – 16-bit signed immediate constant offset value (−32768 to +32767 range)

COND – Conditional codes:

COND	Syntax	Description	Flags Tested
0000	NEQ	Not Equal To	Z = 0
0001	EQ	Equal To	Z = 1
0010	GT	Greater Than	Z = 0 AND N = 0
0011	GEQ	Greater Than or Equal To	N = 0
0100	LT	Less Than	N = 1
0101	LEQ	Less Than or Equal To	Z = 1 OR N = 1
0110	HI	Higher	C = 1 AND Z = 0
0111	HIS, C	Higher or Same, Carry Set	C = 1
1000	LO, NC	Lower, Carry Clear	C = 0
1001	LOS	Lower or Same	C = 0 OR Z = 1
1010	NOV	No Overflow	V = 0
1011	OV	Overflow	V = 1
1100	NTC	Test Bit Not Set	TC = 0
1101	TC	Test Bit Set	TC = 1
1110	NBIO	BIO Input Equal To Zero	BIO = 0
1111	UNC	Unconditional	–

Description Fast conditional branch. If the specified condition is true, then branch by adding the signed 16-bit constant value to the current PC value; otherwise continue execution without branching:

```
If (COND = true) PC = PC + signed 16-bit offset;
If (COND = false) PC = PC + 2;
```

Note: The branch fast (BF) instruction takes advantage of dual prefetch queue on the C28x core that reduces the cycles for a taken branch from 7 to 4:

```
If (COND = true) then the instruction takes 4 cycles.
If (COND = false) then the instruction takes 4 cycles.
```

Flags and Modes

Flags and Modes	Description
V	If the V flag is tested by the condition, then V is cleared.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

C27MAP *Set the M0M1MAP Bit*

Syntax Options C27MAP

Opcode `0101 0110 0011 1111`

Objmode X

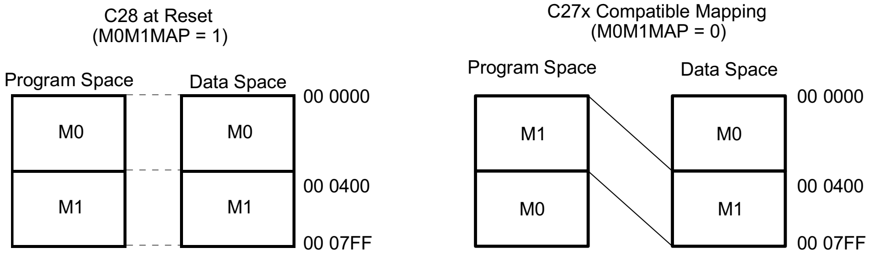
RPT –

CYC 5

Note: This instruction is an alias for the “CLRC M0M1MAP” operation.

Operands None

Description Clear the M0M1MAP status bit, configuring the mapping of the M0 and M1 memory blocks for C27x object-compatible operation. The memory blocks are mapped as follows:



Note: The pipeline is flushed when this instruction is executed.

Note: The pipeline is flushed when this instruction is executed.

Flags and Modes

Flags and Modes	Description
M0M1MAP	The M0M1MAP bit is cleared.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Set the device mode from reset to C27x object-compatible mode: Reset:
C27OBJ      ; Enable C27x Object Mode
C28ADDR     ; Enable C27x/C28x Address Mode
.c28_amode  ; Tell assembler we are using C27x/C28x addressing
C27MAP      ; Enable C27x Mapping Of M0 and M1 blocks
```

C27OBJ *Clear the Objmode Bit*

Syntax Options C27OBJ

Opcode `0101 0110 0011 0110`

Objmode X

RPT –

CYC 5

Note: This instruction is an alias for the “CLRC Objmode” operation.

Operands None

Description Clear the Objmode status bit in Status Register ST1, configuring the device to execute C27x object code. This is the default mode of the processor after reset.

Note: The pipeline is flushed when this instruction is executed.

Flags and Modes Clear the Objmode bit.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
 ; Set the device mode from reset to C27x:
Reset:
C27OBJ      ; Enable C27x Object Mode
C28ADDR     ; Enable C27x/C28x Address Mode
.c28_amode  ; Tell assembler we are in C27x/C28x addr mode
C27MAP      ; Enable C27x Mapping Of M0 and M1 blocks
```

C28ADDR *Clear the AMODE Status Bit*

Syntax Options C28ADDR

Opcode `0101 0110 0001 0110`

Objmode X

RPT –

CYC 1

Note: This instruction is an alias for the “CLRC AMODE” operation.

Operands None

Description Clear the AMODE status bit in Status Register ST1, putting the device in C27x/C28x addressing mode (see Chapter 5).

Note: This instruction does not flush the pipeline.

Flags and Modes

Flags and Modes	Description
AMODE	The AMODE bit is cleared.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Execute the operation VarC = VarA + VarB written in
; C2xLP syntax:
LPADDR      ; Full C2xLP address compatible mode
.lp_amode   ; Tell assembler we are in C2xLP mode
LDP #VarA   ; Initialize DP (low 64K only)
LACL VarA   ; ACC = VarA (ACC high = 0)
ADDS VarB   ; ACC = ACC + VarB (unsigned)
SACL VarC   ; Store result into VarC
C28ADDR     ; Return to C28x address mode
.c28_amode  ; Tell assembler we are in C28x mode
```

C28MAP *Set the M0M1MAP Bit*

Syntax Options C28MAP

Opcode `0101 0110 0001 1010`

Objmode X

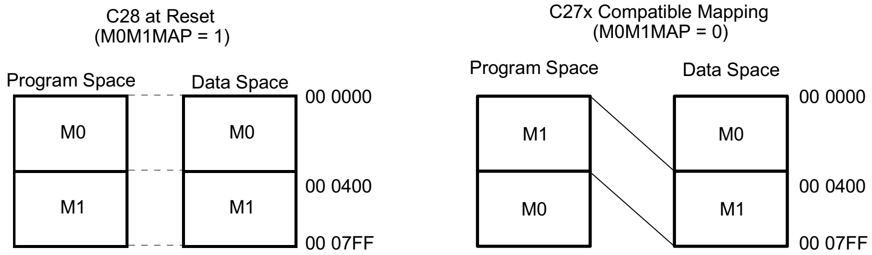
RPT –

CYC 5

Note: This instruction is an alias for the “SETC M0M1MAP” instruction.

Operands None

Description Set the M0M1MAP status bit in Status register ST1, configuring the mapping of the M0 and M1 memory blocks for C28x operation. The memory blocks are mapped as follows:



Note: The pipeline is flushed when this instruction is executed.

Note: The pipeline is flushed when this instruction is executed.

Flags and Modes

Flags and Modes	Description
M0M1MAP	The M0M1MAP bit is set.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Set the device mode from reset to C28x mode: Reset:
C28OBJ      ; Enable C28x Object Mode
C28ADDR     ; Enable C28x Address Mode
.c28_amode  ; Tell assembler we are in C28x address mode
C28MAP      ; Enable C28x Mapping Of M0 and M1 blocks
```

C28OBJ *Set the Objmode Bit*

Syntax Options C28OBJ

Opcode `0101 0110 0001 1111`

Objmode X

RPT –

CYC 5

Note: This instruction is an alias for the “SETC Objmode” instruction.

Operands None

Description Set the Objmode status bit, putting the device in C28x object mode (supports C2xLP source).

Flags and Modes

Flags and Modes	Description
Objmode	Set the Objmode bit.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Set the device mode from reset to C28x: Reset:
C28OBJ      ; Enable C28x Object Mode
C28ADDR     ; Enable C27x/C28x Address Mode
.c28_amode  ; Tell assembler we are in C27x/C28x address mode
C28MAP      ; Enable C28x Mapping Of M0 and M1 blocks
```

CLRC AMODE *Clear the AMODE Bit*

Syntax Options CLRC AMODE

Opcode `0101 0110 0001 0110`

Objmode X

RPT –

CYC 1

Operands **AMODE** – Status bit

Description Clear the AMODE status bit in Status Register ST1, enabling C27x/C28x addressing.

Note: This instruction does not flush the pipeline.

Flags and Modes

Flags and Modes	Description
AMODE	The AMODE bit is cleared.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Execute the operation VarC = VarA + VarB written in C2xLP
; syntax:
SETC AMODE  ; Full C2xLP address-compatible mode
.lp_amode   ; Tell assembler we are in C2xLP mode
LDP #VarA   ; Initialize DP (low 64K only)
LACL VarA   ; ACC = VarA (ACC high = 0)
ADDS VarB   ; ACC = ACC + VarB (unsigned)
SACL VarC   ; Store result into VarC
CLRC AMODE  ; Return to C28x address mode
.c28_amode  ; Tell assembler we are in C28x mode
```

CLRC M0M1MAP *Clear the M0M1MAP Bit*

Syntax Options CLRC M0M1MAP

Opcode `0101 0110 0011 1111`

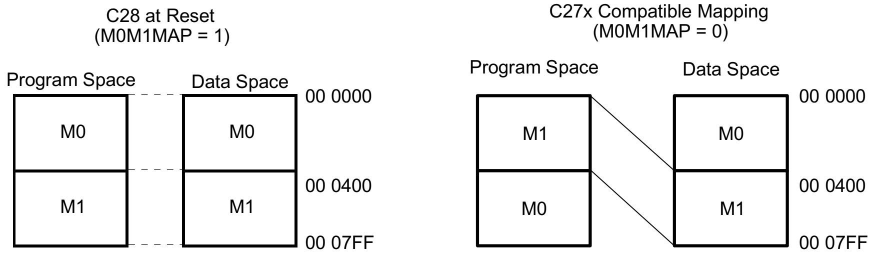
Objmode X

RPT –

CYC 5

Operands **M0M1MAP** – Status bit

Description Clear the M0M1MAP status bit in Status Register ST1, configuring the mapping of the M0 and M1 memory blocks for C27x operation. The memory blocks are mapped as follows:



Note: The pipeline is flushed when this instruction is executed.

Note: The pipeline is flushed when this instruction is executed. This bit is provided for compatibility for users migrating from C27x. The M0M1MAP bit should always remain set to 1 for users operating in C28x mode and C2xLP source-compatible mode.

Flags and Modes

Flags and Modes	Description
M0M1MAP	The M0M1MAP bit is cleared.

Example

```
; Set the device mode from reset to C27x object-compatible mode:
Reset:
  CLRC Objmode  ; Enable C27x Object Mode
  CLRC AMODE    ; Enable C27x/C28x Address Mode
  .c28_amode    ; Tell assembler we are in C27x/C28x addr mode
  CLRC M0M1MAP  ; Enable C27x Mapping Of M0 and M1 blocks
```

CLRC Objmode *Clear the Objmode Bit*

Syntax Options CLRC Objmode

Opcode `0101 0110 0011 0110`

Objmode X

RPT –

CYC 5

Operands **Objmode** – Status bit

Description Clear the Objmode status bit, enabling the device to execute C27x object code.

Note: The pipeline is flushed when this instruction is executed.

Flags and Modes

Table 6-4. Flags and Modes

Flags and Modes	Description
Objmode	The Objmode bit is cleared.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Set the device mode from reset to C27x object-compatible mode:
Reset:
  CLRC Objmode  ; Enable C27x Object Mode
  CLRC AMODE    ; Enable C27x/C28x Address Mode
  .c28_amode    ; Tell assembler we are in C27x/C28x addr mode
  CLRC M0M1MAP  ; Enable C27x Mapping Of M0 and M1 blocks
```

CLRC OVC *Clear Overflow Counter*

Syntax Options CLRC OVC

Opcode `0101 0110 0101 1100`

Objmode 1

RPT –

CYC 1

Note: This instruction is an alias for the “ZAP OVC” operation.

Operands **OVC** – Overflow counter bits in Status Register 0 (ST0)

Description Clear the overflow counter (OVC) bits in ST0.

Flags and Modes

Flags and Modes	Description
OVC	The 6-bit overflow counter bits (OVC) are cleared.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Calculate: VarD = sat(VarA + VarB + VarC)
CLRC OVC        ; Zero overflow counter
MOVL ACC,@VarA  ; ACC = VarA
ADDL ACC,@VarB  ; ACC = ACC + VarB
ADDL ACC,@VarC  ; ACC = ACC + VarC
SAT ACC         ; Saturate if OVC != 0
MOVL @VarD,ACC  ; Store saturated result into VarD
```

CLRC XF *Clear XF Status Bit*

Syntax Options CLRC XF

Opcode `0101 0110 0001 1011`

Objmode X

RPT –

CYC 1

Operands **XF** – XF status bit and output signal

Description Clear the XF status bit and pull the corresponding output signal low.

Flags and Modes

Flags and Modes	Description
XF	The XF status bit is cleared.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Pulse XF signal high if branch not taken:
MOV AL,@VarA  ; Load AL with contents of VarA
SB Dest,NEQ   ; ACC = VarA
SETC XF       ; Set XF bit and signal high
CLRC XF       ; Clear XF bit and signal low
.
. Dest:
```

CLRC Mode *Clear Status Bits*

Syntax Options

Syntax Options	Opcode	Objmode	RPT	CYC
CLRC mode	`0010 1001 CCCC CCCC`	X	–	1, 2
CLRC SXM	`0010 1001 0000 0001`	X	–	1
CLRC OVM	`0010 1001 0000 0010`	X	–	1
CLRC TC	`0010 1001 0000 0100`	X	–	1
CLRC C	`0010 1001 0000 1000`	X	–	1
CLRC INTM	`0010 1001 0001 0000`	X	–	2
CLRC DBGM	`0010 1001 0010 0000`	X	–	2
CLRC PAGE0	`0010 1001 0100 0000`	X	–	1
CLRC VMAP	`0010 1001 1000 0000`	X	–	1

Description Clear the specified status bits. Any change affects the next instruction in the pipeline. The mode operand is a mask value that relates to the status bits in this way:

Mode bit	Status Register	Flag	Cycles
0	ST0	SXM	1
1	ST0	OVM	1
2	ST0	TC	1
3	ST0	C	1
4	ST1	INTM	2
5	ST1	DBGM	2
6	ST1	PAGE0	1
7	ST1	VMAP	1

Note: The assembler accepts any number of flag names in any order.

Flags and Modes

Flags and Modes	Description
SXM OVM TC C INTM DBGM PAGE0 VMAP	Any of the specified bits can be cleared by the instruction.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Modify flag settings:
SETC INTM,DBGM     ; Set INTM and DBGM bits to 1
CLRC TC,C,SXM,OVM  ; Clear TC, C, SXM, OVM bits to 0
CLRC #0xFF         ; Clear all bits to 0
SETC #0xFF         ; Set all bits to 1
SETC C,SXM,TC,OVM  ; Set TC, C, SXM, OVM bits to 1
CLRC DBGM,INTM     ; Clear INTM and DBGM bits to 0
```

CMP AX, loc16 *Compare*

Syntax Options CMP AX, loc16

Opcode `0101 010A LLLL LLLL`

Objmode X

RPT –

CYC 1

Operands **AX** – Accumulator high (AH) or accumulator low (AL) register

loc16 – Addressing mode (see Chapter 5)

Description The content of the specified AX register (AH or AL) is compared with the 16-bit content of the location pointed to by the “loc16” addressing mode. The result of (AX−- [loc16]) is evaluated and the status flag bits set accordingly. The AX register and content of the location pointed to by “loc16” are left unchanged:

```
Set Flags On (AX – [loc16]);
```

Flags and Modes

Flags and Modes	Description
N	If the result of the operation is negative, then N is set; otherwise it is cleared. The CMP instruction assumes infinite precision when it determines the sign of the result. For example, consider the subtraction 0x8000 − 0x0001. If the precision were limited to 16 bits, the result would cause an overflow to the positive number 0x7FFF and N would be cleared. However, because the CMP instruction assumes infinite precision, it would set N to indicate that 0x8000 − 0x0001 actually results in a negative number.
Z	The comparison is tested for a zero condition. The zero flag bit is set if the operation (AX − [loc16]) = 0, otherwise it is cleared.
C	If the subtraction generates a borrow, then C is cleared; otherwise C is set.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
 ; Branch if VarA is higher then VarB:
MOV AL,@VarA   ; Load AL with contents of VarA
CMPB AL,@VarB  ; Set Flags On (AL – VarB)
SB Dest,HI     ; Branch if VarA higher then VarB
```

CMP loc16,#16bitSigned *Compare*

Syntax Options CMP loc16,#16bitSigned

Opcode

```
0001 1011 LLLL LLLL
CCCC CCCC CCCC CCCC
```

Objmode X

RPT –

CYC 1

Operands

loc16 – Addressing mode (see Chapter 5)

#16bitsigned – 16-bit immediate signed constant value

Description

Compare the 16-bit contents of the location pointed to by the “loc16” addressing mode to the signed 16-bit immediate constant value. To perform the comparison, the result of ([loc16] − #16bitSigned) is evaluated and the status flag bits are set accordingly. The content of “loc16” is left unchanged:

```
Modify flags on ([loc16] – 16bitSigned);
```

Smart Encoding:

If loc16 = AL or AH and #16bitSigned is an 8-bit number, then the assembler will encode this instruction as CMPB AX, #8bit, to override this encoding, use the CMPW AX, #16bitSigned instruction alias.

Flags and Modes

Flags and Modes	Description
N	If the result of the operation is negative, then N is set; otherwise it is cleared. The CMP instruction assumes infinite precision when it determines the sign of the result. For example, consider the subtraction 0x8000 − 0x0001. If the precision were limited to 16 bits, the result would cause an overflow to the positive number 0x7FFF and N would be cleared. However, because the CMP instruction assumes infinite precision, it would set N to indicate that 0x8000 − 0x0001 actually results in a negative number.
Z	The comparison is tested for a zero condition. The zero flag bit is set if the operation ([loc16] − 16bitSigned) = 0, otherwise it is cleared.
C	If the subtraction generates a borrow, then C is cleared; otherwise C is set.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

The examples in this chapter assume that the device is already operating in C28x Mode (Objmode = 1, AMODE = 0). To put the device into C28x mode following a reset, you must first set the Objmode bit in ST1 by executing the “C28OBJ” (or “SETC Objmode”) instruction.

```
; Calculate:
; if( VarA > 20 )
; VarA = 0;
CMP @VarA,#20    ; Set flags on (VarA – 20)
MOVB @VarA,#0,GT ; Zero VarA if greater then
```

CMP64 ACC:P *Compare 64-bit Value*

Syntax Options CMP64 ACC:P

Opcode `0101 0110 0101 1110`

Objmode 1

RPT –

CYC 1

Operands **ACC:P** – Accumulator register (ACC) and product register (P)

Description The 64-bit content of the combined ACC:P registers is compared against zero and the flags are set appropriately:

```
if((V = 1) & (ACC(bit 31) = 1))
    N = 0;
else
    N = 1;
if((V = 1) & (ACC(bit 31) = 0))
    N = 1;
else
    N = 0;
if(ACC:P = 0x8000 0000 0000 0000)
    Z = 1;
else
    Z = 0;
V = 0;
```

Note: This operation should be used as follows:

```
CMP64 ACC:P  ; Clear V flag perform 64-bit operation
CMP64 ACC:P  ; Set Z,N flags, V=0 conditionally branch
```

Flags and Modes

Flags and Modes	Description
N	The content of the ACC register is tested to determine if the 64-bit ACC:P value is negative. The CMP64 instruction takes into account the state of the overflow flag (V) to increase precision when determining if ACC is negative. For example, consider the subtraction on ACC of 0x8000 0000 − 0x0000 0001. This results in an overflow to a positive number (0x7FFF FFFF) and V would be set. Because the CMP64 instruction takes into account the overflow, it would interpret the result as a negative number and not a positive number. If the value is ACC is found to be negative, then N is set; otherwise N is cleared.
Z	The zero flag bit is set if the combined 64 bits of ACC:P is zero, otherwise it is cleared.
V	The state of the V flag is used along with bit 31 of the ACC register to determine if the value in the ACC:P register is negative. V is cleared by the operation.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; If 64-bit VarA > 64-bit VarB, branch:
MOVL P,@VarA+0     ; Load P with low 32 bits of VarA
MOVL ACC,@VarA+2   ; Load ACC with high 32 bits of VarA
SUBUL P,@VarB+0    ; Sub from P unsigned low 32 bits of VarB
CMP64 ACC:P        ; Clear V flag
SUBBL ACC,@VarB+2  ; Sub from ACC with borrow high 32 bits of VarB
CMP64 ACC:P        ; Set Z,N flags appropriately for ACC:P
SB Dest,GT         ; branch if VarA > VarB
```

CMPB AX, #8bit *Compare 8-bit Value*

Syntax Options CMPB AX, #8bit

Opcode `0101 001A CCCC CCCC`

Objmode X

RPT –

CYC 1

Operands

AX – Accumulator high (AH) or accumulator low (AL) register

#8bit – 8-bit immediate constant value

Description

Compare the content of the specified AX register (AH or AL) with the zero-extended 8-bit unsigned immediate constant. The result of (AX − 0:8bit) is evaluated and the status flag bits are set accordingly. The content of the AX register is left unchanged:

```
Set Flags On (AX – 0:8bit);
```

Flags and Modes

Flags and Modes	Description
N	If the result of the operation is negative, then N is set; otherwise it is cleared. The CMPB instruction assumes infinite precision when it determines the sign of the result. For example, consider the subtraction 0x8000 − 0x0001. If the precision were limited to 16 bits, the result would cause an overflow to the positive number 0x7FFF and N would be cleared. However, because the CMPB instruction assumes infinite precision, it would set N to indicate that 0x8000 − 0x0001 actually results in a negative number.
Z	The comparison is tested for a zero condition. The zero flag bit is set if the operation (AX − [0:8bit]) = 0, otherwise it is cleared.
C	If the subtraction generates a borrow, then C is cleared; otherwise C is set.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Check if VarA is within range 0x80 <= VarA <= 0xF0:
MOV AL,@VarA      ; Load AL with contents of VarA
CMPB AL,#0xF0     ; Set Flags On (AL – 0x00F0)
SB OutOfRange,GT  ; Branch if VarA greater then 0x00FF
CMPB AL,#0x80     ; Set Flags On (AL – 0x0080)
SB OutOfRange,LT  ; Branch if VarA less then 0x0080
```

CMPL ACC,loc32 *Compare 32-bit Value*

Syntax Options CMPL ACC,loc32

Opcode `0000 1111 LLLL LLLL`

Objmode X

RPT –

CYC 1

Operands **ACC** – Accumulator register

loc32 – Addressing mode (see Chapter 5)

Description The content of the ACC register is compared with the 32-bit location pointed to by the “loc32” addressing mode. The status flag bits are set according to the result of (ACC − [loc32]). The ACC register and the contents of the location pointed to by “loc32” are left unchanged:

```
Modify flags on (ACC – [loc32]);
```

Flags and Modes

Flags and Modes	Description
N	If the result of the operation is negative, then N is set; otherwise it is cleared. The CMPL instruction assumes infinite precision when it determines the sign of the result. For example, consider the subtraction 0x8000 0000 − 0x0000 0001. If the precision were limited to 32 bits, the result would cause an overflow to the positive number 0x7FFF FFFF and N would be cleared. However, because the CMPL instruction assumes infinite precision, it would set N to indicate that 0x8000 0000 − 0x0000 0001 actually results in a negative number.
Z	The comparison is tested for a zero condition. The zero flag bit is set if the operation (AX − [loc32]) = 0, otherwise it is cleared.
C	If the subtraction generates a borrow, C is cleared; otherwise C is set.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once

Example

```
; Swap the contents of 32-bit VarA and VarB if VarB is higher:
MOVL ACC,@VarB     ; ACC = VarB
MOVL P,@VarA       ; P = VarA
CMPL ACC,@P        ; Set flags on (VarB - VarA)
MOVL @VarA,ACC,HI  ; VarA = ACC if higher
MOVL @P,ACC,HI     ; P = ACC if higher
MOVL @VarA,P       ; VarA = P
```

CMPL ACC,P << PM *Compare 32-bit Value*

Syntax Options CMPL ACC,P << PM

Opcode `1111 1111 0101 1001`

Objmode X

RPT –

CYC 1

Operands

ACC – Accumulator register

P – Product register

<<PM – Product shift mode

Description The content of the ACC register is compared with the content of the P register, shifted by the amount specified by the product shift mode (PM). The status flag bits are set according to the result of (ACC −[P << PM]). The content of the ACC register and the P register are left unchanged:

```
Modify flags on (ACC − [P << PM]);
```

Flags and Modes

Flags and Modes	Description
N	If the result of the operation is negative, then N is set; otherwise it is cleared. The CMPL instruction assumes infinite precision when it determines the sign of the result. For example, consider the subtraction 0x8000 0000 − 0x0000 0001. If the precision were limited to 32 bits, the result would cause an overflow to the positive number 0x7FFF FFFF and N would be cleared. However, because the CMPL instruction assumes infinite precision, it would set N to indicate that 0x8000 0000 − 0x0000 0001 actually results in a negative number.
Z	The comparison is tested for a zero condition. The zero flag bit is set if the operation (AX − [P<<PM]) = 0, otherwise, it is cleared.
C	If the subtraction generates a borrow, C is cleared; otherwise C is set.
PM	The value in the PM bits sets the shift mode for the output operation from the product register. If the product shift value is positive (logical left shift operation), then the low bits are zero filled. If the product shift value is negative (arithmetic right shift operation), the upper bits are sign extended.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Compare the following (VarA − VarB << 4):
MOVL ACC,@VarA    ; ACC = VarA
SPM −4            ; Set the product shift mode to  "<< 4"
MOVL P,@VarB      ; P = VarB
CMPL ACC,P << PM  ; Compare (VarA − VarB << 4)
```

CMPR 0/1/2/3 *Compare Auxiliary Registers*

Syntax Options

Syntax Options	Opcode	Objmode	RPT	CYC
CMPR 0	`0101 0110 0001 1101`	1	–	1
CMPR 1	`0101 0110 0001 1001`	1	–	1
CMPR 2	`0101 0110 0001 1000`	1	–	1
CMPR 3	`0101 0110 0001 1100`	1	–	1

Operands None

Description Compare AR0 to the 16-bit auxiliary register pointed to by ARP. The comparison type is determined by the instruction.

```
CMPR 0: if(AR0 = AR[ARP]) TC = 1, else TC = 0
CMPR 1: if(AR0 > AR[ARP]) TC = 1, else TC = 0
CMPR 2: if(AR0 > AR[ARP]) TC = 1, else TC = 0
CMPR 3: if(AR0 != AR[ARP]) TC = 1, else TC = 0
```

Flags and Modes

Flags and Modes	Description
ARP	The 3-bit ARP points to the current valid Auxiliary Register, XAR0 to XAR7. This pointer determines which Auxiliary register is compared to AR0.
TC	If the test is true, TC is set, else TC is cleared.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
TableA: .word 0x1111
    .word 0x2222
FuncA:
  MOVL XAR1,#VarA   ; Initialize XAR1 Pointer
  MOVZ AR0,*XAR1++  ; Load AR0 with 0x1111, clear AR0H,
                    ; ARP = 1

  MOVZ AR1,*XAR1    ; Load AR1 with 0x2222, clear AR1H
  CMPR 0            ; AR0 = AR1? No, clear TC
  B Equal,TC        ; Don’t branch
  CMPR 2            ; AR1 > AR2? Yes, set TC
  B Less,TC         ; Branch to "Less"
```

CSB ACC *Count Sign Bits*

Syntax Options CSB ACC

Opcode `0101 0110 0011 0101`

Objmode 1

RPT –

CYC 1

Operands **ACC** – Accumulator register

Description Count the sign bits in the ACC register by determining the number of leading 0s or 1s in the ACC register and storing the result, minus one, in the T register:

```
T = 0, 1 sign bit
T = 1, 2 sign bits
.
.
T = 31, 32 sign bits
```

Note: The count sign bit operation is often used in normalization operations and is particularly useful for algorithms such as; calculating Square Root of a number, calculating the inverse of a number, searching for the first ”1” bit in a word.

Flags and Modes

Flags and Modes	Description
N	N is set if bit 31 of ACC is 1, else N is cleared.
Z	Z is set if ACC is 0, else Z is cleared.
TC	The TC bit will reflect the state of the sign bit after the operation (TC=1 for negative).

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once

Example

```
; Normalize the contents of VarA:
MOVL ACC,@VarA  ; Load ACC with contents of VarA
CSB ACC         ; Count sign bits
LSLL ACC,T      ; Logical shift left ACC by T(4:0)
MOVL @VarA,ACC  ; Store result into VarA
```

DEC loc16 *Decrement by 1*

Syntax Options DEC loc16

Opcode `0000 1011 LLLL LLLL`

Objmode X

RPT –

CYC 1

Operands **loc16** – Addressing mode (see Chapter 5)

Description Subtract 1 from the signed content of the location pointed to by the “loc16” addressing mode.

Flags and Modes

Flags and Modes	Description
N	After the operation if bit 15 of [loc16] is 1, set N; otherwise, clear N.
Z	After the operation if [loc16] is zero, set Z; otherwise, clear Z.
C	If the subtraction generates a borrow, C is cleared; otherwise C is set.
V	If an overflow occurs, V is set; otherwise V is not affected.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; VarA = VarA - 1;
DEC @VarA  ; Decrement contents of VarA
```

DINT *Disable Maskable Interrupts (Set INTM Bit)*

Syntax Options DINT

Opcode `0011 1011 0001 0000`

Objmode X

RPT –

CYC 2

Note: This instruction is an alias for the “SETC mode” operation with the ”mode” field = INTM.

Operands None

Description Disable all maskable CPU interrupts by setting the INTM status bit. Any change affects the next instruction in the pipeline. DINT has no effect on the unmaskable reset or NMI interrupts.

Flags and Modes

Flags and Modes	Description
INTM	The instruction sets this bit to disable interrupts.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Make the operation "VarC = VarA + VarB" atomic:
DINT             ; Disable interrupts (INTM = 1)
MOVL ACC,@VarA   ; ACC = VarA
ADDL ACC,@VarB   ; ACC = ACC + VarB
MOVL @VarC,ACC   ; Store result into VarC
EINT             ; Enable interrupts (INTM = 0)
```

DMAC ACC:P,loc32,*XAR7/++ *16-Bit Dual Multiply and Accumulate*

Syntax Options

Syntax Options	Opcode	Objmode	RPT	CYC

DMAC ACC:P,loc32,*XAR7	`0101 0110 0100 1011`
	`1100 0111 LLLL LLLL`
		1	Y	N+2

DMAC ACC:P,loc32,*XAR7++	`0101 0110 0100 1011`
	`1000 0111 LLLL LLLL`
		1	Y	N+2

Operands

ACC:P – Accumulator register (ACC) and product register (P)

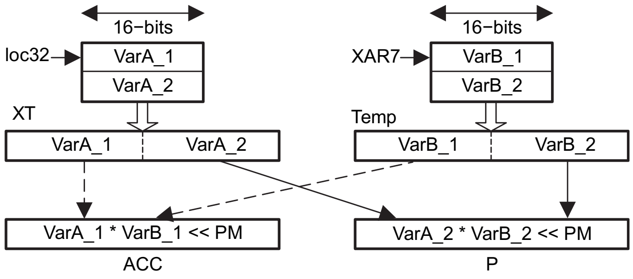
loc32 – Addressing mode (see Chapter 5)

Note: The @ACC and @P register addressing modes cannot be used. No illegal instruction trap will be generated if used (assembler will flag an error).

***XAR7 /++** – Indirect program-memory addressing using auxiliary register XAR7, can access full 4M x 16 program space range (0x000000 to 0x3FFFFF)

Description

Dual 16-bit x 16-bit signed multiply and accumulate. The first multiplication takes place between the upper words of the 32-bit locations pointed to by the “loc32” and *XAR7/++ addressing modes and second multiplication takes place with the lower words.



After the operation the ACC contains the result of multiplying and adding the upper word of the addressed 32-bit operands. The P register contains the result of multiplying and adding the lower word of the addressed 32-bit operands.

```
XT    = [loc32];
Temp  = Prog[*XAR7 or *XAR7++];
ACC   = ACC + (XT.MSW * Temp.MSW) << PM;
P     = P  + (XT.LSW * Temp.LSW) << PM;
```

Z, N, V, C flags and OVC counter are affected by the operation on ACC only. The PM shift affects both the ACC and P operations.

On the C28x devices, memory blocks are mapped to both program and data space (unified memory), hence the ”*XAR7/++” addressing mode can be used to access data space variables that fall within the program space address range.

With some addressing mode combinations, you can get conflicting references. In such cases, the C28x will give the “loc16/loc32” field priority on changes to XAR7. For example:

```
DMAC ACC:P,*−−XAR7,*XAR7++   ; −−XAR7  given priority
DMAC ACC:P,*XAR7++,*XAR7     ; *XAR7++ given priority
DMAC ACC:P,*XAR7,*XAR7++     ; *XAR7++ given priority
```

Flags and Modes

Flags and Modes	Description
Z	After the addition, the Z flag is set if the ACC value is zero, else Z is cleared.
N	After the addition, the N flag is set if bit 31 of the ACC is 1, else N is cleared.
C	If the addition generates a carry of the ACC register, C is set; otherwise C is cleared.
V	If an overflow of the ACC register occurs, V is set; otherwise V is not affected.
OVC	If overflow mode is disabled; and if the operation generates a positive overflow of the ACC register, then the counter is incremented. If overflow mode is disabled; and if the operation generates a negative overflow of the ACC register, then the counter is decremented.
OVM	If overflow mode bit is set; then the ACC value will saturate maximum positive (0x7FFFFFFF) or maximum negative (0x80000000) if the operation overflowed. Note that OVM only affects the ACC operation.
PM	The value in the PM bits sets the shift mode for the output operation from the product register. The PM mode affects both the ACC and P register accumulates. If the product shift value is positive (logical left shift operation), then the low bits are zero filled. If the product shift value is negative (arithmetic right shift operation), the upper bits are sign extended.

Repeat

This instruction is repeatable. If the operation follows a RPT instruction, then it will be executed N+1 times. The state of the Z, N, C and OVC flags will reflect the final result in the ACC. The V flag will be set if an intermediate overflow occurs in the ACC.

Example

```
;    Calculate sum of product using dual 16-bit multiply:
;    int16 X[N]    ; Data information
;    int16 C[N]    ; Coefficient information (located in low 4M)
;               ; Data and Coeff must be aligned to even address
;               ; N must be an even number
; sum = 0;
; for(i=0; i < N; i++)
; sum = sum + (X[i] * C[i]) >> 5;
  MOVL XAR2,#X            ; XAR2 = pointer to    X
  MOVL XAR7,#C            ; XAR7 = pointer to    C
  SPM −5                  ; Set product shift to ">> 5"
  ZAPA                    ; Zero ACC, P, OVC
  RPT #(N/2)−1            ; Repeat next instruction N/2 times
||DMAC P,*XAR2++,*XAR7++  ; ACC = ACC + (X[i+1] * C[i+1]) >> 5
                          ; P = P + (X[i] * C[i]) >> 5 i++
  ADDL ACC,@P             ; Perform final accumulate
  MOVL @sum,ACC           ; Store final result into sum
```

DMOV loc16 *Data Move Contents of 16-bit Location*

Syntax Options DMOV loc16

Opcode `1010 0101 LLLL LLLL`

Objmode 1

RPT Y

CYC N+1

Operands **loc16** – Addressing mode (see Chapter 5)

Note: For this operation, register−addressing modes cannot be used. The modes are: @ARn, @AH, @AL, @PH, @PL, @SP, @T. An illegal instruction trap will be generated.

Description Copy the contents pointed to by ”loc16” into the next highest address:

```
loc16 + 1] = [loc16];
```

Flags and Modes None

Repeat This instruction is repeatable. If the operation is follows a RPT instruction, then it will be executed N+1 times.

Example

```
; Calculate using 16-bit multiply:
; int16 X[3];
; int16 C[3];
; Y = (X[0]*C[0] >> 2) + (X[1]*C[1] >> 2) + (X[2]*C[2] >>2);
; X[2] = X[1];
; X[1] = X[0];
SPM −2              ; Set product shift  to >> 2
MOVP T,@X+2         ; T = X[2]
MPYS P,T,@C+2       ; P = T*C[2], ACC = 0
MOVA T,@X+1         ; T = X[1], ACC = X[2]*C[2] >> 2
MPY P,T,@C+1        ; P = T*C[1]
MOVA T,@X+0         ; T = X[0], ACC = ACC + X[1]*C[1] >> 2
MPY P,T,@C+0        ; P = T*C[0]
ADDL ACC,P << PM    ; ACC = ACC + X[0]*C[0] >> 2
DMOV @X+1           ; X[2] = X[1]
DMOV @X+0           ; X[1] = X[0]
MOVL @Y,ACC         ; Store result into Y
```

EALLOW *Enable Write Access to Protected Space*

Syntax Options EALLOW

Opcode `0111 0110 0010 0010`

Objmode X

RPT –

CYC 4

Operands None

Description Enable access to emulation space and other protected registers.

This instruction sets the EALLOW bit in status register ST1. When this bit is set, the C28x CPU allows write access to the memory-mapped registers as well as other protected registers. See the data sheet for your particular device to determine which registers the EALLOW bit protects.

To again protect against writes to the registers, use the EDIS instruction.

EALLOW only controls write access; reads are allowed even if EALLOW has not been executed.

On an interrupt or trap, the current state of the EALLOW bit is saved off onto the stack within ST1 and the EALLOW bit is autocratically cleared. Therefore, at the start of an interrupt service routine access to the protected registers is disabled. The IRET instruction will restore the current state of the EALLOW bit saved on the stack.

The EALLOW bit is overridden via the JTAG port, allowing full control of register accesses during debug from Code Composer Studio.

Flags and Modes

Flags and Modes	Description
EALLOW	The EALLOW flag is set.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Enable access to RegA and RegB which are EALLOW protected:
EALLOW              ; Enable access to selected registers
AND @RegA,#0x4000   ; RegA = RegA AND 0x0400
MOV @RegB,#0        ; RegB = 0
EDIS                ; Disable access to selected registers
```

EDIS *Disable Write Access to Protected Registers*

Syntax Options EDIS

Opcode `0111 0110 0001 1010`

Objmode X

RPT –

CYC 4

Operands None

Description Disable access to emulation space and other protected registers.

This instruction clears the EALLOW bit in status register ST1. When this bit is clear, the C28x CPU does not allow write access to the memory−mapped emulation registers and other protected registers. See the data sheet for your particular device to determine which registers the EALLOW bit protects.

To allow write access to the registers, use the EALLOW instruction.

Flags and Modes

Flags and Modes	Description
EALLOW	The EALLOW flag is cleared.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Enable access to RegA and RegB which are EALLOW protected:
  EALLOW              ; Enable access to selected registers
  NOP                 ; Wait 2 cycles for enable to take
                      ; effect. The number of cycles is device
                      ; and/or register dependant.
  NOP
  AND @RegA,#0x4000   ; RegA = RegA AND 0x0400
  MOV @RegB,#0        ; RegB = 0
  EDIS                ; Disable access to selected registers
```

EINT *Enable Maskable Interrupts (Clear INTM Bit)*

Syntax Options EINT

Opcode `0010 1001 0001 0000`

Objmode X

RPT –

CYC 2

Note: This instruction is an alias for the “CLRC mode” operation with the ”mode” field = INTM.

Operands None

Description Enable interrupts by clearing the INTM status bit. Any change affects the next instruction in the pipeline.

Flags and Modes

Flags and Modes	Description
INTM	This bit is cleared by the instruction to enable interrupts.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Make the operation "VarC = VarA + VarB" atomic:
  DINT            ; Disable interrupts (INTM = 1)
  MOVL ACC,@VarA  ; ACC = VarA
  ADDL ACC,@VarB  ; ACC = ACC + VarB
  MOVL @VarC,ACC  ; Store result into VarC
  EINT            ; Enable interrupts (INTM = 0)
```

ESTOP0 *Emulation Stop 0*

Syntax Options ESTOP0

Opcode `0111 0110 0010 0101`

Objmode X

RPT –

CYC 4

Operands None

Description Emulation Stop 0

This instruction is available for emulation purposes. It is used to create a software breakpoint.

When an emulator is connected to the C28x and emulation is enabled, this instruction causes the C28x to halt, regardless of the state of the DBGM bit in status register ST1. In addition, ESTOP0 does not increment the PC.

When an emulator is not connected or when a debug program has disabled emulation, the ESTOP0 instruction is treated the same way as a NOP instruction. It simply advances the PC to the next instruction.

Flags and Modes None

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

ESTOP1 *Emulation Stop 1*

Syntax Options ESTOP1

Opcode `0111 0110 0010 0100`

Objmode X

RPT –

CYC 1

Operands None

Description Emulation Stop 1

This instruction is available for emulation purposes. It is used to create an embedded software breakpoint.

When an emulator is connected to the C28x and emulation is enabled, this instruction causes the C28x to halt, regardless of the state of the DBGM bit in status register ST1. Before halting the processor, ESTOP1 increments the PC so that it points to the instruction following the ESTOP1.

When an emulator is not connected or when a debug program has disabled emulation, the ESTOP0 instruction is treated the same way as a NOP instruction. It simply advances the PC to the next instruction.

Flags and Modes None

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

FFC XAR7,22bit *Fast Function Call*

Syntax Options FFC XAR7,22bit

Opcode

```
0000 0000 11CC CCCC
CCCC CCCC CCCC CCCC
```

Objmode X

RPT –

CYC 4

Operands

XAR7 – Auxiliary register XAR7

22bit – 22-bit program-address (0x00 0000 to 0x3F FFFF range)

Description

Fast function call. The return PC value is stored into the XAR7 register and the 22-bit immediate destination address is loaded into the PC:

```
XAR7(21:0) = PC + 2;
XAR7(31:22) = 0;
PC = 22 bit;
```

Flags and Modes None

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Fast function call of FuncA:
FFC XAR7,FuncA  ; Call FuncA, return address in XAR7
.
.

FuncA:          ; Function A:
.
.
LB *XAR7        ; Return: branch to address in XAR7
```

FLIP AX *Flip Order of Bits in AX Register*

Syntax Options FLIP AX

Opcode `0101 0110 0111 000A`

Objmode 1

RPT –

CYC 1

Operands **AX** – Accumulator high (AH) or accumulator low (AL) register

Description Bit reverse the contents of the specified AX register (AH or AL):

```
temp = AX;
AX(bit 0) = temp(bit 15);
AX(bit 1) = temp(bit 14);
.
.
AX(bit 14) = temp(bit 1);
AX(bit 15) = temp(bit 0);
```

Flags and Modes

Flags and Modes	Description
N	After the operation, if bit 15 of AX is 1 then the negative flag bit is set; otherwise it is cleared.
Z	After the operation, if AX is 0, then the Z bit is set, otherwise it is cleared.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Flip the contents of 32-bit variable VarA:
MOV AH,@VarA+0  ; Load AH with low 16 bits of VarA
MOV AL,@VarA+1  ; Load AL with high 16 bits of VarA
FLIP AL         ; Flip contents of AL
FLIP AH         ; Flip contents of AH
MOVL @VarA,ACC  ; Store 32-bit result in VarA
```

IACK #16bit *Interrupt Acknowledge*

Syntax Options

IACK #16bit

Opcode

```
0111 0110 0011 1111
CCCC CCCC CCCC CCCC
```

Objmode

X

RPT

–

CYC

1

Operands

#16bit – 16-bit constant immediate value (0x0000 to 0xFFFF range)

Description

Acknowledge an interrupt by outputting the specified 16-bit constant on the low 16 bits of the data bus. Certain peripherals will provide the capability to capture this value to provide low-cost trace. See the data sheet for details for your device.

```
data_bus(15:0) = 16bit;
```

Flags and Modes

None

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

IDLE *Put Processor in Idle Mode*

Syntax Options IDLE

Opcode `0111 0110 0010 0001`

Objmode X

RPT –

CYC 5

Operands None

Description Put the processor into idle mode and wait for enabled or nonmaskable interrupt. Devices using the 28x CPU may use the IDLE instruction in combination with external logic to achieve different low-power modes. See the device-specific datasheets for more detail. The idle instruction causes the following sequence of events:

1. The pipeline is flushed.
2. All outstanding memory cycles are completed.
3. The IDLESTAT bit of status register ST1 is set.
4. Clocks to the CPU are stopped after the entire instruction buffer is full, placing the device in the idle state. In the idle state, CLKOUT (the clock output from the CPU) and all clocks to blocks outside the CPU (including the emulation block) continue to operate as long as CLKIN (the clock input to the CPU) is driven. The PC continues to hold the address of the IDLE instruction; the PC is not incremented before the CPU enters the idle state.
5. The IDLE output CPU signal is activated (driven high).
6. The device waits for an enabled or nonmaskable hardware interrupt. If such an interrupt occurs, the IDLESTAT bit is cleared, the PC is incremented by 1, and the device exits the idle state.

If the interrupt is maskable, it must be enabled in the interrupt enable register (IER). However, the device exits the idle state regardless of the value of the interrupt global mask bit (INTM) of status register ST1.

After the device exits the idle mode, the CPU must respond to the interrupt request. If the interrupt can be disabled by the INTM bit in status register ST1, the next event depends on INTM:

- If (INTM = 0), then the interrupt is enabled, and the CPU executes the corresponding interrupt service routine. On return from the interrupt, execution begins at the instruction following the IDLE instruction.
- If (INTM = 1), then the interrupt is blocked and program execution continues at the instruction immediately following the IDLE.

If the interrupt cannot be disabled by INTM, the CPU executes the corresponding interrupt service routine. On return from the interrupt, execution begins at the instruction following the IDLE.

Flags and Modes

Flags and Modes	Description
IDLESTAT	Before entering the idle mode, IDLESTAT is set; after exiting the idle mode IDLESTAT is cleared.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

IMACL P,loc32,*XAR7/++ *Signed 32 X 32-Bit Multiply and Accumulate (Lower Half)*

Syntax Options

Syntax Options	Opcode	Objmode	RPT	CYC

IMACL P,loc32,*XAR7	`0101 0110 0100 1101`
	`1100 0111 LLLL LLLL`
		1	Y	N+2

IMACL P,loc32,*XAR7++	`0101 0110 0100 1101`
	`1000 0111 LLLL LLLL`
		1	Y	N+2

Operands

P – Product register

loc32 – Addressing mode (see Chapter 5)

Note: The @ACC addressing mode cannot be used when the instruction is repeated. No illegal instruction trap will be generated if used (assembler will flag an error).

***XAR7/++** – Indirect program-memory addressing using auxiliary register XAR7; can access full 4Mx16 program space range (0x000000 to 0x3FFFFF)

Description

32-bit x 32-bit signed multiply and accumulate. First, add the unsigned previous product (stored in the P register), ignoring the product shift mode (PM), to the ACC register. Then, multiply the signed 32-bit content of the location pointed to by the “loc32” addressing mode by the signed 32-bit content of the program-memory location pointed to by the XAR7 register. The product shift mode (PM) then determines which part of the lower 38 bits of the 64-bit result are stored in the P register. If specified, post-increment the XAR7 register by 1:

```
ACC = ACC + unsigned P;
temp(37:0) = lower_38 bits(signed [loc32]
             * signed Prog[*XAR7 or XAR7++]);
if(PM = +4 shift)
    P(31:4) = temp(27:0), P(3:0) = 0;
if(PM = +1 shift)
    P(31:1) = temp(30:0), P(0) = 0;
if(PM = 0 shift)
    P(31:0) = temp(31:0);
if(PM = −1 shift)
    P(31:0) = temp(32:1);
if(PM = −2 shift)
    P(31:0) = temp(33:2);
if(PM = −3 shift)
    P(31:0) = temp(34:3);
if(PM = −4 shift)
    P(31:0) = temp(35:4);
if(PM = −5 shift)
    P(31:0) = temp(36:5);
if(PM = −6 shift)
    P(31:0) = temp(37:6);
```

On the C28x devices, memory blocks are mapped to both program and data space (unified memory), hence the ”*XAR7/++” addressing mode can be used to access data space variables that fall within the program space address range. With some addressing mode combinations, you can get conflicting references. In such cases, the C28x will give the “loc16/loc32” field priority on changes to XAR7. For example:

```
IMACL P,*−−XAR7,*XAR7++  ; −−XAR7 given priority
IMACL P,*XAR7++,*XAR7    ; *XAR7++ given priority
IMACL P,*XAR7,*XAR7++    ; *XAR7++ given priority
```

Flags and Modes

Flags and Modes	Description
Z	After the addition, the Z flag is set if the ACC value is zero, else Z is cleared.
N	After the addition, the N flag is set if bit 31 of the ACC is 1, else N is cleared.
C	If the addition generates a carry, C is set; otherwise C is cleared.
V	If an overflow occurs, V is set; otherwise V is not affected.
OVCU	The overflow counter is incremented when the addition operation generates an unsigned carry. The OVM mode does not affect the OVCU counter.
PM	The value in the PM bits sets the shift mode that determines which portion of the lower 38 bits of the 64-bit results are stored in the P register.

Repeat

This instruction is repeatable. If the operation follows a RPT instruction, then it will be executed N+1 times. The state of the Z, N, C and OVC flags will reflect the final result in the ACC. The V flag will be set if an intermediate overflow occurs in the ACC.

Example

```
; Calculate sum of product using 32-bit multiply and retain
; 64-bit result:
; int32 X[N];  // Data information
; int32 C[N];  // Coefficient information (located in
               // low 4M)
; int64 sum = 0;
; for(i=0; i < N; i++)
;   sum = sum + (X[i] * C[i]) >> 5;
; Calculate low 32 bits:
  MOVL XAR2,#X              ; XAR2 = pointer to X
  MOVL XAR7,#C              ; XAR7 = pointer to C
  SPM −5                    ; Set product shift to ">> 5"
  ZAPA                      ; Zero ACC, P, OVCU
  RPT #(N−1)                ; Repeat next instruction N times
||IMACL P,*XAR2++,*XAR7++  ;OVCU:ACC = OVCU:ACC + P,
                            ;P = (X[i] * C[i]) << 5
                            ;i++
  ADDUL ACC,@P              ; OVCU:ACC = OVCU:ACC + P
  MOVL @sum+0,ACC           ; Store low 32 bits result into sum
; Calculate high 32 bits:
  MOVU @AL,OVC              ; ACC = OVCU (carry count)
  MOVB AH,#0
  MPYB P,T,#0               ; P = 0
  MOVL XAR2,#X              ; XAR2 = pointer to X
  MOVL XAR7,#C              ; XAR7 = pointer to C
  RPT #(N−1)                ; Repeat next instruction N times
||QMACL P,*XAR2++,*XAR7++  ; ACC = ACC + P >> 5,
                            ; P = (X[i] * C[i]) >> 32,
                            ; i++
ADDL ACC,P << PM            ; ACC = ACC + P >> 5
MOVL @sum+2,ACC             ; Store high 32 bits result into sum
```

IMPYAL P,XT,loc32 *Signed 32-Bit Multiply (Lower Half) and Add Previous P*

Syntax Options

IMPYAL P,XT,loc32

Opcode

```
0101 0110 0100 1100
0000 0000 LLLL LLLL
```

Objmode

1

RPT

–

CYC

1

Operands

P – Product register

XT – Multiplicand register.

loc32 – Addressing mode (see Chapter 5)

Description

Add the unsigned content of the P register, ignoring the product shift mode (PM), to the ACC register. Multiply the signed 32-bit content of the XT register by the signed 32-bit content of the location pointed to by the “loc32” addressing mode. The product shift mode (PM) then determines which part of the lower 38 bits of the 64-bit result are stored in the P register:

```
ACC = ACC + unsigned P;
temp(37:0) = lower_38 bits(signed XT * signed [loc32]);
if(PM = +4 shift)
  P(31:4) = temp(27:0), P(3:0) = 0;
if(PM = +1 shift)
  P(31:1) = temp(30:0), P(0) = 0;
if(PM = 0 shift)
  P(31:0) = temp(31:0);
if(PM = −1 shift)
  P(31:0) = temp(32:1);
if(PM = −2 shift)
  P(31:0) = temp(33:2);
if(PM = −3 shift)
  P(31:0) = temp(34:3);
if(PM = −4 shift)
  P(31:0) = temp(35:4);
if(PM = −5 shift)
  P(31:0) = temp(36:5);
if(PM = −6 shift)
  P(31:0) = temp(37:6);
```

Flags and Modes

Flags and Modes	Description
Z	After the addition, the Z flag is set if the ACC value is zero, else Z is cleared.
N	After the addition, the N flag is set if bit 31 of the ACC is 1, else N is cleared.
C	If the addition generates a carry, C is set; otherwise C is cleared.
V	If an overflow occurs, V is set; otherwise V is not affected.
OVCU	The overflow counter is incremented when the addition operation generates an unsigned carry. The OVM mode does not affect the OVCU counter.
PM	The value in the PM bits sets the shift mode that determines which portion of the lower 38 bits of the 64-bit results are stored in the P register.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Calculate signed result:
; Y64 = (X0*C0 + X1*C1 + X2*C2) >> 2
SPM −2            ; Set product shift mode to ">> 2"
ZAPA              ; Zero ACC, P, OVCU
MOVL XT,@X0       ; XT = X0
IMPYL P,XT,@C0    ; P = low 32 bits of (X0*C0 << 2)
MOVL XT,@X1       ; XT = X1
IMPYAL P,XT,@C1   ; OVCU:ACC = OVCU:ACC + P,
                  ; P = low 32 bits of (X1*C1 << 2)
MOVL XT,@X2       ; XT = X2
IMPYAL P,XT,@C2   ; OVCU:ACC = OVCU:ACC + P,
                  ; P = low 32 bits of (X2*C2 << 2)
ADDUL ACC,@P      ; OVCU:ACC = OVCU:ACC + P
MOVL @Y64+0,ACC   ; Store low 32-bit result into Y64
MOVU @AL,OVC      ; ACC = OVCU (carry count)
MOVB AH,#0
QMPYL P,XT,@C2    ; P = high 32 bits of (X2*C2)
MOVL XT,@X1       ; XT = X1
QMPYAL P,XT,@C1   ; ACC = ACC + P >> 2,
                  ; P = high 32 bits of (X1*C1)
MOVL XT,@X0       ; XT = X0
QMPYAL P,XT,@C0   ; ACC = ACC + P >> 2,
                  ; P = high 32 bits of (X0*C0)
ADDL ACC,P << PM  ; ACC = ACC + P >> 2
MOVL @Y64+2,ACC   ; Store high 32-bit result into Y64
```

IMPYL ACC,XT,loc32 *Signed 32 X 32-Bit Multiply (Lower Half)*

Syntax Options IMPYL ACC,XT,loc32

Opcode

```
0101 0110 0100 0100
0000 0000 LLLL LLLL
```

Objmode 1

RPT –

CYC 2

Operands

ACC – Accumulator register

XT – Multiplicand register

loc32 – Addressing mode (see Chapter 5)

Description Multiply the signed 32-bit content of the XT register by the signed 32-bit content of the location pointed to by the “loc32” addressing mode and store the lower 32 bits of the 64-bit result in the ACC register:

```
ACC = signed XT * signed [loc32];
```

Flags and Modes

Flags and Modes	Description
Z	After the operation, the Z flag is set if the ACC value is zero, else Z is cleared.
N	After the operation, the N flag is set if bit 31 of the ACC is 1, else N is cleared.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Calculate result: Y32 = M32*X32 + B32
MOVL XT,@M32       ; XT = M32
IMPYL ACC,XT,@X32  ; ACC = low    32 bits of (M32*X32)
ADDL ACC,@B32      ; ACC = ACC + B32
MOVL @Y32,ACC      ; Store result into Y32
```

IMPYL P,XT,loc32 *Signed 32 X 32-Bit Multiply (Lower Half)*

Syntax Options IMPYL P,XT,loc32

Opcode

```
0101 0110 0000 0101
0000 0000 LLLL LLLL
```

Objmode 1

RPT –

CYC 1

Operands

P – Product register

XT – Multiplicand register.

loc32 – Addressing mode (see Chapter 5)

Description

Multiply the signed 32-bit content of the XT register by the signed 32-bit content of the location pointed to by the “loc32” addressing mode. The product shift mode (PM) then determines which part of the lower 38 bits of the 64-bit result gets stored in the P register as shown in the diagram below:

```
temp(37:0) = lower_38 bits(signed XT * signed [loc32]);
if(PM = +4 shift)
  P(31:4) = temp(27:0), P(3:0) = 0;
if(PM = +1 shift)
  P(31:1) = temp(30:0), P(0) = 0;
if(PM = 0 shift)
  P(31:0) = temp(31:0);
if(PM = −1 shift)
  P(31:0) = temp(32:1);
if(PM = −2 shift)
  P(31:0) = temp(33:2);
if(PM = −3 shift)
  P(31:0) = temp(34:3);
if(PM = −4 shift)
  P(31:0) = temp(35:4);
if(PM = −5 shift)
  P(31:0) = temp(36:5);
if(PM = −6 shift)
  P(31:0) = temp(37:6);
```

Flags and Modes

Flags and Modes	Description
PM	The value in the PM bits sets the shift mode that determines which portion of the lower 38 bits of the 64-bit results are stored in the P register.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Calculate signed result: Y64 = M32*X32
MOVL XT,@M32       ; XT = M32
IMPYL P,XT,@X32    ; P = low 32 bits of (M32*X32)
QMPYL ACC,XT,@X32  ; ACC = high 32 bits of (M32*X32)
MOVL @Y64+0,P      ; Store result into Y64
MOVL @Y64+2,ACC
```

IMPYSL P,XT,loc32 *Signed 32-Bit Multiply (Low Half) and Subtract P*

Syntax Options IMPYSL P,XT,loc32

Opcode

```
0101 0110 0100 0011
0000 0000 LLLL LLLL
```

Objmode 1

RPT –

CYC 1

Operands

P – Product register

XT – Multiplicand register.

loc32 – Addressing mode (see Chapter 5)

Description

Subtract the unsigned content of the P register, ignoring the product shift mode (PM), from the ACC register. Multiply the signed 32-bit content of the XT register by the signed 32-bit content of the location pointed to by the “loc32” addressing mode. The product shift mode (PM) then determines which part of the lower 38 bits of the 64-bit result are stored in the P register:

```
ACC = ACC - unsigned P;
temp(37:0) = lower_38 bits(signed XT * signed [loc32]);
if(PM = +4 shift)
  P(31:4) = temp(27:0), P(3:0) = 0;
if(PM = +1 shift)
  P(31:1) = temp(30:0), P(0) = 0;
if(PM = 0 shift)
  P(31:0) = temp(31:0);
if(PM = −1 shift)
  P(31:0) = temp(32:1);
if(PM = −2 shift)
  P(31:0) = temp(33:2);
if(PM = −3 shift)
  P(31:0) = temp(34:3);
if(PM = −4 shift)
  P(31:0) = temp(35:4);
if(PM = −5 shift)
  P(31:0) = temp(36:5);
if(PM = −6 shift)
  P(31:0) = temp(37:6);
```

Flags and Modes

Flags and Modes	Description
Z	After the subtraction, the Z flag is set if the ACC value is zero, else Z is cleared.
N	After the subtraction, the N flag is set if bit 31 of the ACC is 1, else N is cleared.
C	If the subtraction generates a borrow, C is cleared; otherwise C is set.
V	If an overflow occurs, V is set; otherwise V is not affected.
OVCU	The overflow counter is decremented when the subtraction operation generates an unsigned borrow. The OVM mode does not affect the OVCU counter.
PM	The value in the PM bits sets the shift mode that determines which portion of the lower 38 bits of the 64-bit results are stored in the P register.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Calculate signed result:
                  ; Y64 = (–X0*C0 – X1*C1 – X2*C2) >> 2
SPM –2            ; Set product shift mode  to ">> 2"
ZAPA              ; Zero ACC, P, OVCU
MOVL XT,@X0       ; XT = X0
IMPYL P,XT,@C0    ; P    = low 32 bits of (X0*C0 << 2)
MOVL XT,@X1       ; XT = X1
IMPYSL P,XT,@C1   ; OVCU:ACC = OVCU:ACC – P,
                  ; P    = low 32 bits of (X1*C1 << 2)
MOVL XT,@X2       ; XT = X2
IMPYSL P,XT,@C2   ; OVCU:ACC = OVCU:ACC – P,
                  ; P = low 32 bits of (X2*C2 << 2)
SUBUL ACC,@P      ; OVCU:ACC = OVCU:ACC – P
MOVL @Y64+0,ACC   ; Store low 32-bit result into Y64
MOVU @AL,OVC      ; ACC = OVCU (borrow count)
MOVB AH,#0
NEG ACC           ; Negate borrow
QMPYL P,XT,@C2    ; P = high 32 bits of (X2*C2)
MOVL XT,@X1       ; XT = X1
QMPYSL P,XT,@C1   ; ACC = ACC – P  >>  2,|
                  ; P = high 32 bits of (X1*C1)
MOVL XT,@X0       ; XT = X0
QMPYSL P,XT,@C0   ; ACC = ACC – P  >>  2,
                  ; P = high 32 bits of (X0*C0)
SUBL ACC,P << PM  ; ACC = ACC – P  >> 2
MOVL @Y64+2,ACC   ; Store high 32-bit result into Y64
```

IMPYXUL P,XT,loc32 *Signed 32 X Unsigned 32-Bit Multiply (Lower Half)*

Syntax Options IMPYXUL P,XT,loc32

Opcode

```
0101 0110 0110 0101
0000 0000 LLLL LLLL
```

Objmode 1

RPT –

CYC 1

Operands

P – Product register

XT – Multiplicand register.

loc32 – Addressing mode (see Chapter 5)

Description

Multiply the signed 32-bit content of the XT register by the unsigned 32-bit content of the location pointed to by the “loc32” addressing mode. The product shift mode (PM) then determines which part of the lower 38 bits of the 64-bit result are stored in the P register:

```
temp(37:0) = lower_38 bits(signed XT * unsigned [loc32]);
if(PM = +4 shift)
  P(31:4) = temp(27:0), P(3:0) = 0;
if(PM = +1 shift)
  P(31:1) = temp(30:0), P(0) = 0;
if(PM = 0 shift)
  P(31:0) = temp(31:0);
if(PM = −1 shift)
  P(31:0) = temp(32:1);
if(PM = −2 shift)
  P(31:0) = temp(33:2);
if(PM = −3 shift)
  P(31:0) = temp(34:3);
if(PM = −4 shift)
  P(31:0) = temp(35:4);
if(PM = −5 shift)
  P(31:0) = temp(36:5);
if(PM = −6 shift)
  P(31:0) = temp(37:6);
```

Flags and Modes

Flags and Modes	Description
PM	The value in the PM bits sets the shift mode that determines which portion of the lower 38 bits of the 64-bit results are stored in the P register.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Calculate result: Y64 = M64*X64 + B64
                  ; Y64 = Y1:Y0, M64 = M1:M0, X64 = X1:X0, B64 = B1:B0
MOVL XT,@X0       ; XT = X0
IMPYL P,XT,@M0    ; P    = low 32 bits of (uns M0 * uns X0)
MOVL ACC,@B0      ; ACC = B0
ADDUL ACC,@P      ; ACC = ACC + P
MOVL @Y0,ACC      ; Store result into Y0
QMPYUL P,XT,@M0   ; P = high 32 bits of (uns M0 * uns X0)
MOVL XT,@X1       ; XT = X1
MOVL ACC,@P       ; ACC = P
IMPYXUL P,XT,@M0  ; P = low 32 bits of (uns M0 * sign X1)
MOVL XT,@M1       ; XT = M1
ADDCL ACC,@P      ; ACC = ACC + P + carry
IMPYXUL P,XT,@X0  ; P = low 32 bits of (sign M1 * uns X0)
ADDUL ACC,@P      ; ACC = ACC + P
ADDUL ACC,@B1     ; ACC = ACC + B1
MOVL @Y1,P        ; Store result into Y1
```

IN loc16,*(PA) *Input Data From Port*

Syntax Options IN loc16,*(PA)

Opcode

```
1011 0100 LLLL LLLL
CCCC CCCC CCCC CCCC
```

Objmode 1

RPT Y

CYC N+2

Operands **loc16** – Addressing mode (see Chapter 5)

***(PA)** – Immediate I/O space memory address

Description Load the location pointed to by the “loc16” addressing mode with the content of the specified I/O location pointed to by ”*(PA)”:

```
[loc16] = IOspace[PA];
```

I/O Space is limited to 64K range (0x0000 to 0xFFFF). On the external interface (XINTF), the I/O strobe signal (XIS), if available on your particular device, is toggled during the operation. The I/O address appears on the lower 16 XINTF address lines (XA[15:0]) and the upper address lines are zeroed. The data is read on the lower 16 data lines (XD[15:0]).

Note: I/O space may not be implemented on all C28x devices. See the data sheet for your particular device for details.

Flags and Modes

Flags and Modes	Description
N	If (loc16 = @AX), then after the move AX is tested for a negative condition. The negative flag bit is set if bit 15 of AX is 1, otherwise it is cleared.
Z	If (loc16 = @AX), then after the move, AX is tested for a zero condition. The zero flag bit is set if AX = 0, otherwise it is cleared.

Repeat This instruction is repeatable. If the operation follows a RPT instruction, then it will be executed N+1 times. When repeated, the “(PA)” I/O space address is post-incremented by 1 during each repetition.

Example

```
; IORegA address = 0x0300;
; IOREgB address = 0x0301;
; IOREgC address = 0x0302;
; IORegA = 0x0000;
; IORegB = 0x0400;
; IORegC = VarA;
; if( IORegC = 0x2000 )
                          ; IORegC = 0x0000;
IORegA .set  0x0300       ; Define IORegA address
IORegB .set  0x0301       ; Define IORegB address
IORegC .set  0x0302       ; Define IORegC address
  MOV @AL,#0              ; AL = 0
  UOUT *(IORegA),@AL      ; IOspace[IORegA] = AL
  MOV @AL,#0x0400         ; AL = 0x0400
  UOUT *(IORegB),@AL      ; IOspace[IORegB] = AL
  OUT *(IORegC),@VarA     ; IOspace[IORegC] = VarA
  IN @AL,*(IORegC)        ; AL = IOspace[IORegC]
  CMP @AL,#0x2000         ; Set flags on (AL − 0x2000)
  SB $10,NEQ              ; Branch if not equal
  MOV @AL,#0              ; AL = 0
  UOUT *(IORegC),@AL      ; IOspace[IORegC] = AL
$10:
```

INC loc16 *Increment by 1*

Syntax Options INC loc16

Opcode

```
0000 1010 LLLL LLLL
```

Objmode X

RPT –

CYC 1

Operands **loc16** – Addressing mode (see Chapter 5)

Description Add 1 to the signed content of the location pointed to by the “loc16” addressing mode:

```
[loc16] = [loc16] + 1;
```

Flags and Modes

Flags and Modes	Description
N	After the operation if bit 15 of [loc16] 1, set N; otherwise, clear N.
Z	After the operation if [loc16] is zero, set Z; otherwise, clear Z.
C	If the addition generates a carry, C is set; otherwise C is cleared.
V	If an overflow occurs, V is set; otherwise V is not affected.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; VarA = VarA + 1;
INC @VarA  ; Increment contents of VarA
```

INTR *Emulate Hardware Interrupt*

Syntax Options

Syntax Options	Opcode	Objmode	RPT	CYC
INTR INTx	`0000 0000 0001 CCCC`	X	–	8
INTR DLOGINT	`0000 0000 0001 CCCC`	X	–	8
INTR RTOSINT	`0000 0000 0001 CCCC`	X	–	8
INTR NMI	`0111 0110 0001 0110`	X	–	8
INTR EMUINT	`0111 0110 0001 1100`	X	–	8

Operands

INTx – Maskable CPU interrupt vector name, x = 1 to 14

DLOGINT – Maskable CPU datalogging interrupt

RTOSINT – Maskable CPU real-time operating system interrupt

NMI – Nonmaskable interrupt

EMUINT – Maskable emulation interrupt

Description

Emulate an interrupt. The INTR instruction transfers program control to the interrupt service routine that corresponds to the vector specified by the instruction. The INTR instruction is not affected by the INTM bit in status register ST1. It is also not affected by enable bits in the interrupt enable register (IER) or the debug interrupt enable register (DBGIER). Once the INTR instruction reaches the decode 2 phase of the pipeline, hardware interrupts cannot be serviced until the INTR instruction is finished executing (until the interrupt service routine begins).

INTx, where x =	Interrupt Vector	INTx, where x =	Interrupt Vector
0	RESET	8	INT8
1	INT1	9	INT9
2	INT2	10	INT10
3	INT3	11	INT11
4	INT4	12	INT12
5	INT5	13	INT13
6	INT6	14	INT14
7	INT7		

Part of the operation involves saving pairs of 16-bit CPU registers onto the stack pointed to by the SP register. Each pair of registers is saved in a single 32-bit operation. The register forming the low word of the pair is saved first (to an even address); the register forming the high word of the pair is saved next (to the following odd address). For example, the first value saved is the concatenation of the T register and the status register ST0 (T:ST0). ST0 is saved first, then T.

This instruction should not be used with vectors 1−12 when the peripheral interrupt expansion (PIE) block is enabled.

```
if(not the NMI vector)
Clear the corresponding IFR bit;
Flush the pipeline;
temp = PC + 1;
Fetch specified vector;
SP = SP + 1;
[SP] = T:ST0;
SP = SP + 2;
[SP] = AH:AL;
SP = SP + 2;
[SP] = PH:PL;
SP = SP + 2;
[SP] = AR1:AR0;
SP = SP + 2;
[SP] = DP:ST1;
SP = SP + 2;
[SP] = DBGSTAT:IER;
SP = SP + 2;
[SP] = temp;
Clear corresponding IER bit;
INTM = 0;      // disable INT1−INT14, DLOGINT, RTOSINT
DBGM = 1;      // disable debug events
EALLOW = 0;    // disable access to emulation registers
LOOP = 0;      // clear loop flag
IDLESTAT = 0;  //clear idle flag
PC = fetched vector;
```

Flags and Modes

Flags and Modes	Description
DBGM	Debug events are disabled by setting the DBGM bit.
INTM	Setting the INTM bit disables maskable interrupts.
EALLOW	EALLOW is cleared to disable access to protected registers.
LOOP	The loop flag is cleared.
IDLESTAT	The idle flag is cleared.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

IRET *Interrupt Return*

Syntax Options IRET

Opcode `0111 0110 0000 0010`

Objmode X

RPT –

CYC 8

Operands None

Description Return from an interrupt. The IRET instruction restores the PC value and other register values that were automatically saved by an interrupt operation. The order in which the values are restored is opposite to the order in which they were saved. All values are popped from the stack using 32-bit operations. The stack pointer is not forced to align to an even address during the register restore operations:

```
SP = SP - 2;
PC = [SP];
SP = SP - 2;

DBGSTAT:IER = [SP];
SP = SP − 2;
DP:ST1 = [SP];

SP = SP - 2;
AR1:AR0 = [SP];

SP = SP - 2;
PH:PL = [SP];
SP = SP - 2;
AH:AL = [SP];
SP = SP - 2;
T:ST0 = [SP];
SP = SP - 1;
```

Note: Interrupts cannot be serviced until the IRET instruction completes execution.

Flags and Modes

Flags and Modes	Description
SXM	The operation restores the state of all flags and modes of the ST0 register.
OVM TC C Z N V PM OVC INTM DBGM PAGEO VMAP SPA EAL- LOW AMODE OBJ- MODE XF ARP	The operation restores the state of the specified flags and modes of the ST1 register. The following bits are not affected: LOOP, IDLESTAT, M0M1MAP

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Full interrupt context Save and Restore:
; Vector table:
INTx: .long INTxService  ; INTx interrupt vector
.
.
.
; Interrupt context save:
INTxService:             ; ACC, P, T, ST0, ST1, DP, AR0,
                         ; AR1, IER, DPGSTAT registers saved
                         ; on stack.
                         ; Return PC saved on stack.
                         ; IER bit corresponding to INTx
                         ; is disabled.
                         ; ST1(EALLOW bit = 0).
                         ; ST1(LOOP bit = 0).
                         ; ST1(DBGM bit = 1).
                         ; ST1(INTM bit = 1).
PUSH AR1H:AR0H           ; Save remaining registers. PUSH    XAR2
PUSH XAR3
PUSH XAR4
PUSH XAR5
PUSH XAR6
PUSH XAR7
PUSH XT
; Interrupt user code:
    .
    .
    .
; Interrupt context restore:
POP XT                   ; Restore registers.
POP XAR7
POP XAR6
POP XAR5
POP XAR4
POP XAR3
POP XAR2
POP AR1H:AR0H
IRET                     ; Return from interrupt.
```

LB *XAR7 *Long Indirect Branch*

Syntax Options LB *XAR7

Opcode `0111 0110 0010 0000`

Objmode X

RPT –

CYC 4

Operands ***XAR7** – Indirect program-memory addressing using auxiliary register XAR7, can access full 4Mx16 program space range (0x000000 to 0x3FFFFF)

Description Long branch indirect. Load the PC with the lower 22 bits of the XAR7 register:

```
PC = XAR7(21:0);
```

Flags and Modes None

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Branch to subroutines in SwitchTable selected by Switch value: SwitchTable:
; Switch address table:
  .long Switch0  ; Switch0 address
  .long Switch1  ; Switch1 address
  .
  .

  MOVL XAR2,#SwitchTable  ; XAR2 = pointer to SwitchTable
  MOVZ AR0,@Switch        ; AR0 = Switch index
  MOVL XAR7,*+XAR2[AR0]   ; XAR7 = SwitchTable[Switch]
  LB *XAR7                ; Indirect branch using XAR7
SwitchReturn:
  .
  .

Switch0:                  ; Function A:
  .
  .
  LB SwitchReturn         ; Return: long branch

Switch1:                  ; Function B:
  .
  .
  LB SwitchReturn         ; Return: long branch
```

LB 22bit *Long Branch*

Syntax Options LB 22bit

Opcode

```
0000 0000 01CC CCCC
CCCC CCCC CCCC CCCC
```

Objmode X

RPT –

CYC 4

Operands **22bit** – 22-bit program-address (0x000000 to 0x3FFFFF range)

Description Long branch. Load the PC with the selected 22-bit program address:

```
PC = 22bit;
```

Flags and Modes None

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Branch to subroutines in SwitchTable selected by Switch
; value:
SwitchTable:                ; Switch address table:
  .long Switch0             ; Switch0 address
  .long Switch1             ; Switch1 address
  .
  .

  MOVL XAR2,#SwitchTable    ; XAR2 = pointer to SwitchTable
  MOVZ AR0,@Switch          ; AR0 = Switch index
  MOVL XAR7,*+XAR2[AR0]     ; XAR7 = SwitchTable[Switch]
  LB *XAR7                  ; Indirect branch using XAR7
SwitchReturn:
  .
  .

Switch0:                    ; Function A:
  .
  .
  LB SwitchReturn           ; Return: long branch

Switch1:                    ; Function B:
  .
  .
  LB SwitchReturn           ; Return: long branch
```

LC *XAR7 *Long Indirect Call*

Syntax Options

LC *XAR7

Opcode

`0111 0110 0000 0100`

Objmode

X

RPT

–

CYC

4

Operands

***XAR7** – Indirect program-memory addressing using auxiliary register XAR7, can access full 4Mx16 program space range (0x000000 to 0x3FFFFF)

Description

Indirect long call. The return PC value is pushed onto the software stack, pointed to by SP register, in two 16-bit operations. Next, the destination address stored in the XAR7 register is loaded into the PC:

```
temp(21:0) = PC + 1;
[SP] = temp(15:0);
SP = SP + 1;
[SP] = temp(21:16);
SP = SP + 1;
PC = XAR7(21:0);
```

Note: For more efficient function calls when operating with Objmode = 1, use the LCR and LRETR instructions instead of the LC and LRET instructions.

Flags and Modes

None

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Call to subroutines in SwitchTable selected by Switch value:
SwitchTable:                ; Switch address table:
  .long Switch0             ; Switch0 address
  .long Switch1             ; Switch1 address
  .
  .
  MOVL XAR2,#SwitchTable    ; XAR2 = pointer to SwitchTable
  MOVZ AR0,@Switch          ; AR0 = Switch index
  MOVL XAR7,*+XAR2[AR0]     ; XAR7 = SwitchTable[Switch]
  LC *XAR7                  ; Indirect call using XAR7
  .
  .
Switch0:                    ; Subroutine 0:
  .
  .
  LRET                      ; Return

Switch1:                    ; Subroutine 1:
  .
  .
  LRET                      ; Return
```

LC 22bit — *Long Call*

Syntax Options

LC 22bit

Opcode

```
0000 0000 10CC CCCC
CCCC CCCC CCCC CCCC
```

Objmode

X

RPT

–

CYC

4

Operands

22bit – 22-bit program-address (0x00 0000 to 0x3F FFFF range)

Description

Long function call. The return PC value is pushed onto the software stack, pointed to by SP register, in two 16-bit operations. Next, the immediate 22-bit destination address is loaded onto the PC:

```
temp(21:0) = PC + 2;
[SP] = temp(15:0);
SP = SP + 1;
[SP] = temp(21:16)
SP = SP + 1;
PC = 22bit;
```

Note: For more efficient function calls when operating with Objmode = 1, use the LCR and LRETR instructions instead of the LC and LRET instructions.

Flags and Modes

None

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Standard function call of FuncA:
  LC FuncA     ; Call FuncA, return address on stack
  .
  .

FuncA:         ; Function A:
  .
  .
  LRET         ; Return from address on stack
```

LCR #22bit *Long Call Using RPC*

Syntax Options LCR #22bit

Opcode

```
0111 0110 01CC CCCC
CCCC CCCC CCCC CCCC
```

Objmode 1

RPT –

CYC 4

Operands **22bit** – 22-bit program-address (0x00 0000 to 0x3F FFFF range)

Description Long call using return PC pointer (RPC). The current RPC value is pushed onto the software stack, pointed to by SP register, in two 16-bit operations. Next, the RPC register is loaded with the return address. Next, the 22-bit immediate destination address is loaded into the PC:

```
[SP] = RPC(15:0);
SP = SP + 1;
[SP] = RPC(21:16);
SP = SP + 1;
RPC = PC + 2;
PC = 22bit;
```

Note: The LCR and LRETR operations, enable 4 cycle call and 4 cycle return. The standard LC and LRET operations only enable a 4 cycle call and 8 cycle return. The LCR and LRETR operations can be nested and can freely replace the LC and LRET operations. This is the case on interrupts also. Only on a task switch operation, does the RPC need to be manually saved and restored.

Flags and Modes None

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; RPC call of FuncA:
  LCR FuncA      ; Call FuncA, return address in RPC
  .
  .

FuncA:           ; Function A:
  .
  .
  LRETR          ; RPC return
```

LCR *XARn — *Long Indirect Call Using RPC*

Syntax Options LCR *XARn

Opcode `0011 1110 0110 0RRR`

Objmode 1

RPT –

CYC 4

Operands ***XARn** – Indirect program-memory addressing using auxiliary register XAR0 to XAR7, can access full 4Mx16 program space range (0x000000 to 0x3FFFFF)

Description Long indirect call using return PC pointer (RPC). The current RPC value is pushed onto the software stack, pointed to by SP register, in two 16-bit operations. Next, the RPC register is loaded with the return address. Next, the destination address stored in the XARn register is loaded into the PC:

```
[SP] = RPC(15:0);
SP = SP + 1;
[SP] = RPC(21:16);
SP = SP + 1;
RPC = PC + 1;
PC = XARn(21:0);
```

Note: The LCR and LRETR operations, enable 4 cycle call and 4 cycle return. The standard LC and LRET operations only enable a 4 cycle call and 8 cycle return. The LCR and LRETR operations can be nested and can freely replace the LC and LRET operations. This is the case on interrupts also. Only on a task switch operation, does the RPC need to be manually saved and restored.

Flags and Modes None

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Call to subroutines in SwitchTable selected by Switch value:
SwitchTable:              ; Switch address table:
  .long Switch0           ; Switch0 address
  .long Switch1           ; Switch1 address
  .
  MOVL XAR2,#SwitchTable  ; XAR2 = pointer to SwitchTable
  MOVZ AR0,@Switch        ; AR0 = Switch index
  MOVL XAR6,*+XAR2[AR0]   ; XAR6 = SwitchTable[Switch]
  LCR *XAR6               ; Indirect RPC call using XAR6
  .
Switch0:                  ; Subroutine 0:
  .
  .
  LRETR                   ; RPC Return
Switch1:                  ; Subroutine 1:
  .
  LRETR                   ; RPC Return
```

LOOPNZ loc16,#16bit *Loop While Not Zero*

Syntax Options

LOOPNZ loc16,#16bit

Opcode

```
0010 1110 LLLL LLLL
CCCC CCCC CCCC CCCC
```

Objmode

X

RPT

–

CYC

5N+5

Operands

loc16 – Addressing mode (see Chapter 5)

#16bit – 16-bit immediate value (0x0000 to 0xFFFF range)

Description

Loop while not zero.

```
while([loc16] & 16bit != 0);
```

The LOOPNZ instruction uses a bitwise AND operation to compare the value referenced by the “loc16” addressing mode and the 16-bit mask value. The instruction performs this comparison repeatedly for as long as the result of the operation is not 0. The process can be described as follows:

1. Set the LOOP bit in status register ST1.
2. Generate the address for the value referenced by the “loc16” addressing mode.
3. If “loc16” is an indirect-addressing operand, perform any specialized modification to the SP or the specified auxiliary register and/or the ARPn pointer.
4. Compare the addressed value with the mask value by using a bitwise AND operation.
5. If the result is 0, clear the LOOP bit and increment the PC by 2. If the result is not 0, then return to step 1.

The loop created by steps 1 through 5 can be interrupted by hardware interrupts. When an interrupt occurs, if the LOOPNZ instruction is still active, the return address saved on the stack points to the LOOPNZ instruction. Therefore, upon return from the interrupt the LOOPNZ instruction is fetched again.

While the result of the AND operation is not 0, the LOOPNZ instruction begins again every five cycles in the decode 2 phase of the pipeline. Thus the memory location or register is read once every five cycles. If you use an indirect addressing mode for the “loc16” operand, you can specify an increment or decrement for the pointer (SP or auxiliary register). If you do, the pointer is modified each time in the decode 2 phase of the pipeline. This means that the mask value is compared with a new data-memory value each time.

The LOOPNZ instruction does not flush prefetched instructions from the pipeline. However, when an interrupt occurs, prefetched instructions are flushed.

When any interrupt occurs, the current state of the LOOP bit is saved as ST1 is saved on the stack. The LOOP bit in ST1 is then cleared by the interrupt. The LOOP bit is a passive status bit. The LOOPNZ instruction changes LOOP, but LOOP does not affect the instruction.

You can abort the LOOPNZ instruction within an interrupt service routine. Test the LOOP bit saved on the stack. If it is set, then increment (by 2) the return address on the stack. Upon return from the interrupt, this incremented address is loaded into the PC and the instruction following the LOOPNZ is executed.

Flags and Modes

Flags and Modes	Description
N	If bit 15 of the result of the AND operation is 1, set N; otherwise, clear N.
Z	If the result of the AND operation is 0, set Z; otherwise, clear Z.
LOOP	LOOP is repeatedly set while the result of the AND operation is not 0. LOOP is cleared when the result is 0. If an interrupt occurs before the LOOPNZ instruction enters the decode 2 phase of the pipeline, the instruction is flushed from the pipeline and, thus, does not affect the LOOP bit.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Wait until bit 3 in RegA is cleared before writing to RegB:
LOOPNZ @RegA,#0x0004    ; Loop while (RegA AND 0x0004 != 0)
MOV    @RegB,#0x8000    ; RegB = 0x8000
```

LOOPZ loc16,#16bit *Loop While Zero*

Syntax Options LOOPZ loc16,#16bit

Opcode

```
0010 1100 LLLL LLLL
CCCC CCCC CCCC CCCC
```

Objmode X

RPT –

CYC 5N+5

Operands

loc16 – Addressing mode (see Chapter 5)

#16bit – 16-bit immediate value (0x0000 to 0xFFFF range)

Description

Loop while zero.

```
while([loc16] & 16bit = 0);
```

The LOOPZ instruction uses a bitwise AND operation to compare the value referenced by the “loc16” addressing mode and the 16-bit mask value. The instruction performs this comparison repeatedly for as long as the result of the operation is 0. The process can be described as follows:

1. Set the LOOP bit in status register ST1.
2. Generate the address for the value referenced by the “loc16” addressing mode.
3. If “loc16” is an indirect-addressing operand, perform any specialized modification to the SP or the specified auxiliary register and/or the ARPn pointer.
4. Compare the addressed value with the mask value by using a bitwise AND operation.
5. If the result is not 0, clear the LOOP bit and increment the PC by 2. If the result is 0, then return to step 1.

The loop created by steps 1 through 5 can be interrupted by hardware interrupts. When an interrupt occurs, if the LOOPZ instruction is still active, the return address saved on the stack points to the LOOPZ instruction. Therefore, upon return from the interrupt the LOOPZ instruction is fetched again.

While the result of the AND operation is 0, the LOOPZ instruction begins again every five cycles in the decode 2 phase of the pipeline. Thus the memory location or register is read once every five cycles. If you use an indirect addressing mode for the “loc16” operand, you can specify an increment or decrement for the pointer (SP or auxiliary register). If you do, the pointer is modified each time in the decode 2 phase of the pipeline. This means that the mask value is compared with a new data-memory value each time.

The LOOPZ instruction does not flush prefetched instructions fr4om the pipeline. However, when an interrupt occurs, prefetched instructions are flushed.

When any interrupt occurs, the current state of the LOOP bit is saved as ST1 is saved on the stack. The LOOP bit in ST1 is then cleared by the interrupt. The LOOP bit is a passive status bit. The LOOPZ instruction changes LOOP, but LOOP does not affect the instruction.

You can abort the LOOPZ instruction within an interrupt service routine. Test the LOOP bit saved on the stack. If it is set, then increment (by 2) the return address on the stack. Upon return from the interrupt, this incremented address is loaded into the PC and the instruction following the LOOPZ is executed.

Flags and Modes

Flags and Modes	Description
N	If bit 15 of the result of the AND operation is 1, set N; otherwise, clear N.
Z	If the result of the AND operation is 0, set Z; otherwise, clear Z.
LOOP	LOOP is repeatedly set while the result of the AND operation is 0. LOOP is cleared when the result is not 0. If an interrupt occurs before the LOOPZ instruction enters the decode 2 phase of the pipeline, the instruction is flushed from the pipeline and, thus, does not affect the LOOP bit.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Wait until bit 3 in RegA is set before writing to RegB:
LOOPZ @RegA,#0x0004   ; Loop while (RegA AND 0x0004 = 0)
MOV @RegB,#0x8000     ; RegB = 0x8000
```

LPADDR *Set the AMODE Bit*

Syntax Options LPADDR

Opcode `0101 0110 0001 1110`

Objmode X

RPT –

CYC 1

Operands None

Description Set the AMODE status bit, putting the device in C2xLP compatible addressing mode (see Chapter 5).

Note: This instruction does not flush the pipeline.

Flags and Modes

Flags and Modes	Description
AMODE	The AMODE bit is set.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Execute the operation "VarC = VarA + VarB" written in C2xLP syntax:
  LPADDR      ; Full C2xLP address compatible mode
  .lp_amode   ; Tell assembler we are in C2XLP mode
  LDP #VarA   ; Initialize DP (low 64K only)
  LACL VarA   ; ACC = VarA (ACC high = 0)
  ADDS VarB   ; ACC = ACC + VarB (unsigned)
  SACL VarC   ; Store result into VarC
  C28ADDR     ; Return to C28x address mode
  .c28_amode  ; Tell assembler we are in C28x mode
```

LRET *Long Return*

Syntax Options LRET

Opcode `0111 0110 0001 0100`

Objmode X

RPT –

CYC 8

Operands None

Description Long return. The return address is popped, from the software stack into the PC, in two 16-bit operations:

```
SP = SP - 1;
temp(31:16) = [SP];
SP = SP – 1;
temp(15:0) = [SP];
PC = temp(21:0);
```

Flags and Modes None

Note: For more efficient function calls when operating with Objmode = 1, use the LCR and LRETR instructions in place of the LC and LRET instructions.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Standard function call of FuncA:
  LC FuncA  ; Call FuncA, return address on stack
  .
  .

FuncA:      ; Function A:
  .
  .
  LRET      ; Return from address on stack
```

LRETE *Long Return and Enable Interrupts*

Syntax Options LRETE

Opcode `0111 0110 0001 0000`

Objmode X

RPT –

CYC 8

Operands None

Description Long return and enable interrupts. The return address is popped, from the software stack into the PC, in two 16-bit operations. Next, the global interrupt flag (INTM) is cleared. This enables global maskable interrupts:

```
SP = SP – 1;
temp(31:16) = [SP];
SP = SP – 1;
temp(15:0) = [SP];
PC = temp(21:0);
INTM = 0;
```

Flags and Modes

Flags and Modes	Description
INTM	This instruction enables interrupts by clearing the INTM bit.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Standard function call of FuncA. Disable interrupts on entry and
; enable interrupts on exit:
  LC FuncA   ; Call FuncA, return address on stack
  .
  .

FuncA:       ; Function A:
  SETC INTM  ; Disable interrupts
  .
  .
  LRETE      ; Return from address on stack,
             ; Enable interrupts
```

LRETR *Long Return Using RPC*

Syntax Options LRETR

Opcode `0000 0000 0000 0110`

Objmode 1

RPT –

CYC 4

Operands None

Description Long return using return PC pointer (RPC). The return address stored in the RPC register is loaded onto the PC. Next, the RPC register is loaded from the software stack in two 16-bit operations:

```
PC = RPC;
SP = SP - 1;
temp(31:16) = [SP];
SP = SP – 1;
temp(15:0) = [SP];
RPC = temp(21:0);
```

Note: The LCR and LRETR operations, enable 4 cycle call and 4 cycle return. The standard LC and LRET operations only enable a 4 cycle call and 8 cycle return. The LCR and LRETR operations can be nested and can freely replace the LC and LRET operations. This is the case on interrupts also. Only on a task switch operation, does the RPC need to be manually saved and restored.

Flags and Modes None

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; RPC call of FuncA:
  LCR FuncA  ; Call FuncA, return address in RPC
  .
  .

FuncA:       ; Function A:
  .
  .
  LRETR      ; RPC return
```

LSL ACC,#1..16 *Logical Shift Left*

Syntax Options LSL ACC,#1..16

Opcode `1111 1111 0011 SHFT`

Objmode X

RPT Y

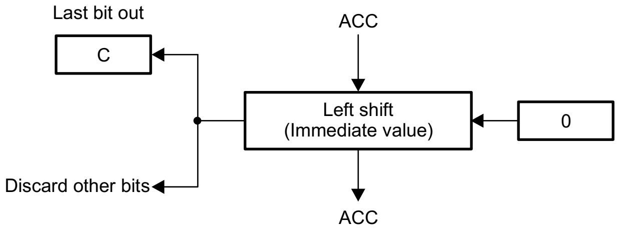
CYC N+1

Operands

ACC – Accumulator register

#1..16 – Shift value

Description Perform a logical shift left on the content of the ACC register by the amount specified by the shift value. During the shift, the low order bits of the ACC register are zero filled and the last bit shifted out is stored in the carry flag bit:



Flags and Modes

Flags and Modes	Description
N	After the shift, if bit 31 of ACC is 1 then the negative flag bit is set; otherwise it is cleared.
Z	After the shift, if ACC is 0, then the Z bit is set, otherwise it is cleared.
C	The last bit to be shifted out of ACC is stored in C.

Repeat This instruction is repeatable. If the operation follows a RPT instruction, then the LSL instruction will be executed N+1 times. The state of the Z, N, and C flags will reflect the final result.

Example

```
; Logical shift left contents of VarA by 4:
  MOVL ACC,@VarA  ; ACC = VarA
  LSL ACC,#4      ; Logical shift left ACC by 4
  MOVL @VarA,ACC  ; Store result into VarA
```

LSL ACC,T *Logical Shift Left by T(3:0)*

Syntax Options LSL ACC,T

Opcode `1111 1111 0101 0000`

Objmode X

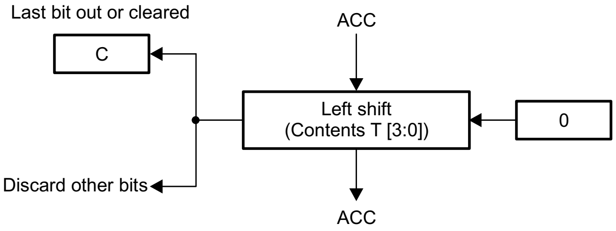
RPT –

CYC 1

Operands **ACC** – Accumulator register

T – Upper 16 bits of the multiplicand (XT) register

Description Perform a logical shift left on the content of the ACC register by the amount specified by the four least significant bits of the T register, T(3:0) = 0…15. Higher order bits are ignored. During the shift, the low order bits of the ACC register are zero filled. If T specifies a shift of 0, then C is cleared; otherwise, C is filled with the last bit to be shifted out of the ACC register:



Flags and Modes

Flags and Modes	Description
Z	After the shift, the Z flag is set if the ACC value is zero, else Z is cleared. Even if the T register specifies a shift of 0, the content of the ACC register is still tested for the zero condition and Z is affected.
N	After the shift, the N flag is set if bit 31 of the ACC is 1, else N is cleared. Even if the T register specifies a shift of 0, the content of the ACC register is still tested for the negative condition and N is affected.
C	If (T(3:0) = 0) then C is cleared; otherwise, the last bit shifted out is loaded into the C flag bit.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Logical shift left contents of VarA by VarB:
  MOVL ACC,@VarA  ; ACC = VarA
  MOV T,@VarB     ; T = VarB (shift value)
  LSL ACC,T       ; Logical shift left ACC by T(3:0)
  MOVL @VarA,ACC  ; Store result into VarA
```

LSL AX,#1...16 *Logical Shift Left*

Syntax Options LSL AX,#1...16

Opcode `1111 1111 100A SHFT`

Objmode X

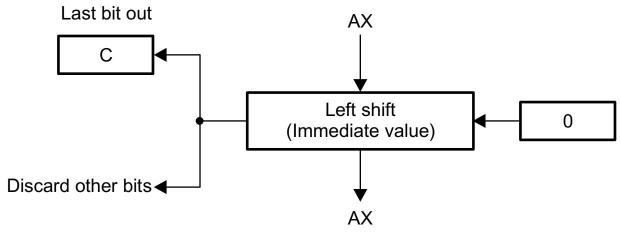
RPT –

CYC 1

Operands **AX** – Accumulator high (AH) or accumulator low (AL) register

#1..16 – Shift value

Description Perform a logical shift left on the content of the specified AX register (AH or AL) by the amount given “shift value” field. During the shift, the low order bits of the AX register are zero filled and the last bit to be shifted out is stored in the carry bit flag:



Flags and Modes

Flags and Modes	Description
N	After the shift, if bit 15 of AX is 1 then the negative flag bit is set; otherwise it is cleared.
Z	After the shift, if AX is 0, then the Z bit is set, otherwise it is cleared.
C	The last bit to be shifted out of AH or AL is stored in C.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Multiply index register AR0 by 2:
MOV AL,@AR0  ; Load AL with contents of AR0
LSL AL,#1    ; Scale result by 1 (*2)
MOV @AR0,AL  ; Store result back in AR0
```

LSL AX,T *Logical Shift Left by T(3:0)*

Syntax Options LSL AX,T

Opcode `1111 1111 0110 011A`

Objmode X

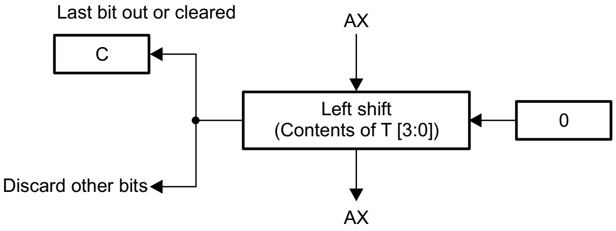
RPT –

CYC 1

Operands **AX** – Accumulator high (AH) or accumulator low (AL) register

T – Upper 16 bits of the multiplicand (XT) register

Description Perform a logical shift left on the content of the specified AX register by the amount specified by the four least significant bits of the T register, T(3:0). The contents of higher order bits are ignored. During the shift, the low order bits of the AX register are zero filled. If the T(3:0) register bits specify a shift of 0, then C is cleared; otherwise, C is filled with the last bit to be shifted out of AX:



Flags and Modes

Flags and Modes	Description
N	After the shift, if bit 15 of AX is 1 then the negative flag bit is set; otherwise it is cleared. Even if the T(3:0) register bits specify a shift of 0, the value of AH or AL is still tested for the negative condition and N is affected.
Z	After the shift, if AX is 0, then the Z bit is set, otherwise it is cleared. Even if the T(3:0) register bits specify a shift of 0, the value of AH or AL is still tested for the zero condition and Z is affected.
C	If T(3:0) specifies a shift of 0, then C is cleared; otherwise, C is filled with the last bit to be shifted out of AH or AL.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Calculate value: VarC = VarA <<  VarB;
MOV T,@VarB   ; Load T with contents of VarB
MOV AL,@VarA  ; Load AL with contents of VarA
LSL AL, T     ; Scale AL by value in T bits 0 to 3
MOV @VarC,AL  ; Store result in VarC
```

LSL64 ACC:P,#1..16 *Logical Shift Left*

Syntax Options LSL64 ACC:P,#1..16

Opcode `0101 0110 1010 SHFT`

Objmode 1

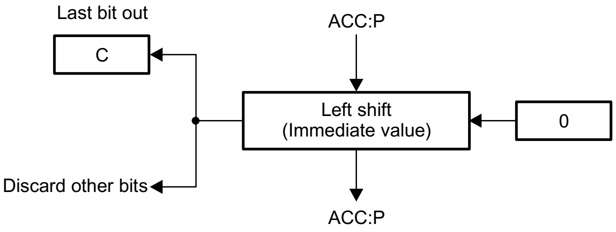
RPT –

CYC 1

Operands **ACC:P** – Accumulator register (ACC) and product register (P)

#1..16 – Shift value

Description Logical shift left the 64-bit combined value of the ACC:P registers by the amount specified in the shift value field. During the shift, the low order bits are zero-filled and the last bit shifted out is stored in the carry bit flag:



Flags and Modes

Flags and Modes	Description
n	After the shift, if bit 31 of the ACC register is 1 then ACC:P is negative and the N bit is set; otherwise N is cleared.
Z	After the shift, the Z flag is set if the combined 64-bit value of the ACC:P is zero; otherwise, Z is cleared.
C	The last bit shifted out of the combined 64-bit value is loaded into the C bit.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Logical shift left the 64-bit Var64 by 10:
MOVL ACC,@Var64+2  ; Load ACC with high 32 bits of Var64
MOVL P,@Var64+0    ; Load P with low 32 bits of Var64
LSL64 ACC:P,#10    ; Logical shift left ACC:P by 10
MOVL @Var64+2,ACC  ; Store high 32-bit result into Var64
MOVL @Var64+0,P    ; Store low 32-bit result into Var64
```

LSL64 ACC:P,T *64-Bit Logical Shift Left by T(5:0)*

Syntax Options LSL64 ACC:P,T

Opcode `0101 0110 0101 0010`

Objmode 1

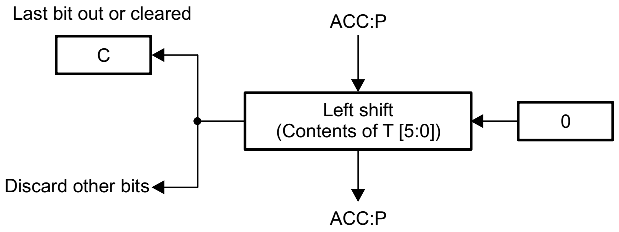
RPT –

CYC 1

Operands **ACC:P** – Accumulator register (ACC) and product register (P)

T – Upper 16 bits of the multiplicand register (XT)

Description Logical shift left the 64-bit combined value of the ACC:P registers by the amount specified in the six least significant bits of the T register, T(5:0) = 0…63. Higher order bits are ignored. During the shift, the low order bits are zero-filled. If T specifies a shift of 0, then C is cleared; otherwise, C is filled with the last bit to be shifted out of the ACC:P registers:



Flags and Modes

Flags and Modes	Description
N	After the shift, if bit 31 of the ACC register is 1 then ACC:P is negative and the N bit is set; otherwise N is cleared.
Z	After the shift, the Z flag is set if the combined 64-bit value of the ACC:P is zero; otherwise, Z is cleared.
C	If (T(5:0) = 0) clear C; otherwise, the last bit shifted out of the combined 64-bit value is loaded into the C bit.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Logical shift left the 64-bit Var64 by contents of Var16:
  MOVL ACC,@Var64+2  ; Load ACC with high 32 bits of Var64
  MOVL P,@Var64+0    ; Load P with low 32 bits of Var64
  MOV T,@Var16       ; Load T with shift value from Var16
  LSL64 ACC:P,T      ; Logical shift left ACC:P by T(5:0)
  MOVL @Var64+2,ACC  ; Store high 32-bit result into Var64
  MOVL @Var64+0,P    ; Store low 32-bit result into Var64
```

LSLL ACC,T *Logical Shift Left by T (4:0)*

Syntax Options LSLL ACC,T

Opcode `0101 0110 0011 1011`

Objmode 1

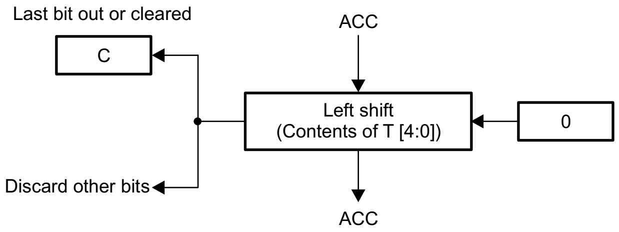
RPT –

CYC 1

Operands **ACC** – Accumulator register

T – Upper 16 bits of the multiplicand (XT) register

Description Perform a logical shift left on the content of the ACC register by the amount specified by the five least significant bits of the T register, T(4:0) = 0…31. Higher order bits are ignored. During the shift, the low order bits of the ACC register are zero filled. If T specifies a shift of 0, then C is cleared; otherwise, C is filled with the last bit to be shifted out of the ACC register:



Flags and Modes

Flags and Modes	Description
Z	After the shift, the Z flag is set if the ACC value is zero, else Z is cleared. Even if the T register specifies a shift of 0, the content of the ACC register is still tested for the zero condition and Z is affected.
N	After the shift, the N flag is set if bit 31 of the ACC is 1, else N is cleared. Even if the T register specifies a shift of 0, the content of the ACC register is still tested for the negative condition and N is affected.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Logical shift left contents of VarA by VarB:

  MOVL ACC,@VarA  ; ACC = VarA
  MOV T,@VarB     ; T = VarB (shift value)
  LSLL ACC,T      ; Logical shift left ACC by T(4:0)
  MOVL @VarA,ACC  ; Store result into VarA
```

LSR AX,#1...16 *Logical Shift Right*

Syntax Options LSR AX,#1...16

Opcode `1111 1111 110A SHFT`

Objmode X

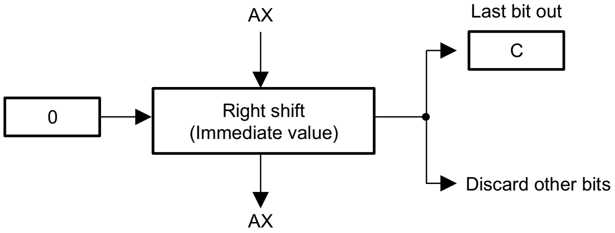
RPT –

CYC 1

Operands **AX** – Accumulator high (AH) or accumulator low (AL) register

#1..16 – Shift value

Description Perform a logical right shift on the content of the specified AX register by the amount given by the “shift value” field. During the shift, the high order bits of the AX register are zero filled and the last bit to be shifted out is stored in the carry flag bit:



Flags and Modes

Flags and Modes	Description
N	After the shift, if bit 15 of AX is 1 then the negative flag bit is set; otherwise it is cleared.
Z	After the shift, if AX is 0, then the Z bit is set, otherwise it is cleared.
C	The last bit to be shifted out of AH or AL is stored in C.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Divide index register AR0 by 2:
MOV AL,@AR0  ; Load AL with contents of AR0
LSR AL,#1    ; Scale result by 1 (/2)
MOV @AR0,AL  ; Store result back in AR0
```

LSR AX,T *Logical Shift Right by T(3:0)*

Syntax Options LSR AX,T

Opcode `1111 1111 0110 001A`

Objmode X

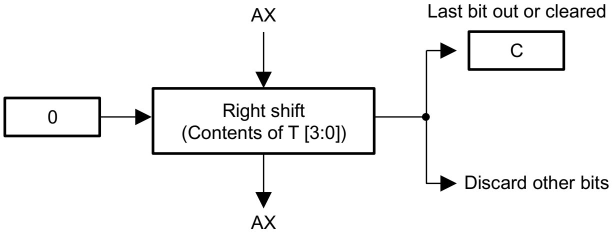
RPT –

CYC 1

Operands **AX** – Accumulator high (AH) or accumulator low (AL) register

T – Upper 16 bits of the multiplicand (XT) register

Description Perform a logical shift right on the content of the specified AX register (AH or AL) as specified by the four least significant bits of the T register, T(3:0). The contents of higher order bits are ignored. During the shift, the high order bits of the AX register are zero filled If the T(3:0) register bits specify a shift of 0, then C is cleared; otherwise, C is filled with the last bit to be shifted out of AX:



Flags and Modes

Flags and Modes	Description
N	After the shift, if bit 15 of AX is 1 then the negative flag bit is set; otherwise it is cleared. Even if the T(3:0) register bits specify a shift of 0, the value of AH or AL is still tested for the negative condition and N is affected.
Z	After the shift, if AX is 0, then the Z bit is set, otherwise it is cleared. Even if the T(3:0) register bits specify a shift of 0, the value of AH or AL is still tested for the zero condition and Z is affected.
C	If T(3:0) specifies a shift of 0, then C is cleared; otherwise, C is filled with the last bit to be shifted out of AH or AL.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Calculate un-signed value: VarC = VarA >> VarB;
MOV T,@VarB   ; Load T with contents of VarB
MOV AL,@VarA  ; Load AL with contents of VarA
LSR AL, T     ; Scale AL by value in T bits 0 to 3
MOV @VarC,AL  ; Store result in VarC
```

LSR64 ACC:P,#1..16 *64-Bit Logical Shift Right*

Syntax Options LSR64 ACC:P,#1..16

Opcode `0101 0110 1001 SHFT`

Objmode 1

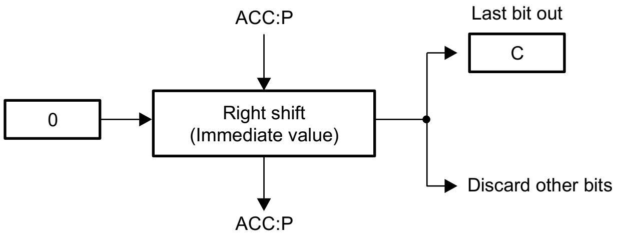
RPT –

CYC 1

Operands **ACC:P** – Accumulator register (ACC) and product register (P)

#1..16 – Shift value

Description Logical shift right the 64-bit combined value of the ACC:P registers by the amount specified in the shift value field. As the value is shifted, the most significant bits are zero filled and the last bit shifted out is stored in the carry bit flag:



Flags and Modes

Flags and Modes	Description
N	After the shift, if bit 31 of the ACC register is 1 then ACC:P is negative and the N bit is set; otherwise N is cleared.
Z	After the shift, the Z flag is set if the combined 64-bit value of the ACC:P is zero; otherwise, Z is cleared.
C	The last bit shifted out of the combined 64-bit value is loaded into the C bit.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Logical shift right the 64-bit Var64 by 10:
MOVL ACC,@Var64+2  ; Load ACC with high 32 bits of Var64

MOVL P,@Var64+0    ; Load P with low 32 bits of Var64
LSR64 ACC:P,#10    ; Logical shift right ACC:P by 10
MOVL @Var64+2,ACC  ; Store high 32-bit result into Var64
MOVL @Var64+0,P    ; Store low 32-bit result into Var64
```

LSR64 ACC:P,T *64-Bit Logical Shift Right by T(5:0)*

Syntax Options LSR64 ACC:P,T

Opcode `0101 0110 0101 1011`

Objmode 1

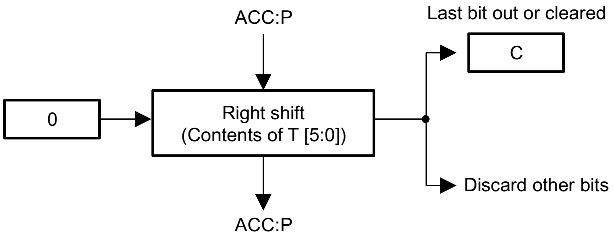
RPT –

CYC 1

Operands **ACC:P** – Accumulator register (ACC) and product register (P)

T – Upper 16 bits of the multiplicand register (XT)

Description Logical shift right the 64-bit combined value of the ACC:P registers by the amount specified by the six least significant bits of the T register, T(5:0) = 0…63. Higher order bits are ignored. As the value is shifted, the most significant bits are zero filled. If T specifies a shift of 0, then C is cleared; otherwise, C is filled with the last bit to be shifted out of the ACC:P registers:



Flags and Modes

Flags and Modes	Description
N	After the shift, if bit 31 of the ACC register is 1 then ACC:P is negative and the N bit is set; otherwise N is cleared.
Z	After the shift, the Z flag is set if the combined 64-bit value of the ACC:P is zero; otherwise, Z is cleared.
C	If (T(5:0) = 0) clear C; otherwise, the last bit shifted out of the combined 64-bit value is loaded into the C bit.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Arithmetic shift right the 64-bit Var64 by contents of Var16:
MOVL ACC,@Var64+2  ; Load ACC with high 32 bits of Var64
MOVL P,@Var64+0    ; Load P with low 32 bits of Var64
MOV T,@Var16       ; Load T with shift value from Var16
LSR64 ACC:P,T      ; Logical shift right ACC:P by T(5:0)
MOVL @Var64+2,ACC  ; Store high 32-bit result into Var64
MOVL @Var64+0,P    ; Store low 32-bit result into Var64
```

LSRL ACC,T *Logical Shift Right by T (4:0)*

Syntax Options LSRL ACC,T

Opcode `0101 0110 0010 0010`

Objmode 1

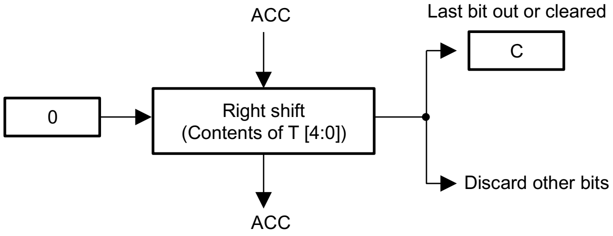
RPT –

CYC 1

Operands **ACC** – Accumulator register

T – Upper 16 bits of the multiplicand (XT) register

Description Perform a logical shift right on the content of the ACC register as specified by the five least significant bits of the T register, T(4:0) = 0…31. Higher order bits are ignored. During the shift, the high order bits of ACC are zero-filled. If T specifies a shift of 0, then C is cleared; otherwise, C is filled with the last bit to be shifted out of the ACC register:



Flags and Modes

Flags and Modes	Description
Z	After the shift, the Z flag is set if the ACC value is zero, else Z is cleared. Even if the T register specifies a shift of 0, the content of the ACC register is still tested for the zero condition and Z is affected.
N	After the shift, the N flag is set if bit 31 of the ACC is 1, else N is cleared. Even if the T register specifies a shift of 0, the content of the ACC register is still tested for the negative condition and N is affected.
C	If (T(4:0) = 0) then C is cleared; otherwise, the last bit shifted out is loaded into the C flag bit.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Logical shift right contents of VarA by VarB:
  MOVL ACC,@VarA  ; ACC = VarA
  MOV T,@VarB     ; T = VarB (shift value)
  LSRL ACC,T      ; Logical shift right ACC by T(4:0)
  MOVL @VarA,ACC  ; Store result into VarA
```

MAC P,loc16,0:pma *Multiply and Accumulate*

Syntax Options MAC P,loc16,0:pma

Opcode

```
0001 0100 LLLL LLLL
CCCC CCCC CCCC CCCC
```

Objmode X

RPT Y

CYC N+2

Operands

P – Product register

loc16 – Addressing mode (see Chapter 5)

0:pma – Immediate program memory address, access low 64K range of program space only (0x000000 to 0x00FFFF)

Description

1. Add the previous product (stored in the P register), shifted as specified by the product shift mode (PM), to the ACC register.
2. Load the T register with the content of the location pointed to by the “loc16” addressing mode.
3. Multiply the signed 16-bit content of the T register by the signed 16-bit content of the addressed program memory location and store the 32-bit result in the P register:

```
ACC = ACC + P << PM;
T = [loc16];
P = signed T * signed Prog[0x00:pma];
```

The C28x forces the upper 6 bits of the program memory address, specified by the “0:pma” addressing mode, to 0x00 when using this form of the MAC instruction. This limits the program memory address to the low 64K of program address space (0x000000 to 0x00FFFF). On the C28x devices, memory blocks are mapped to both program and data space (unified memory), hence the “0:pma” addressing mode can be used to access data space variables that fall within its address range.

Flags and Modes

Flags and Modes	Description
Z	After the addition, the Z flag is set if the ACC value is zero, else Z is cleared.
N	After the addition, the N flag is set if bit 31 of the ACC is 1, else N is cleared.
C	If the addition generates a carry, C is set; otherwise C is cleared.
V	If an overflow occurs, V is set; otherwise V is not affected.
OVC	If overflow mode is disabled; and if the operation generates a positive overflow, then the counter is incremented. If overflow mode is disabled; and if the operation generates a negative overflow, then the counter is decremented.
OVM	If overflow mode bit is set; then the ACC value will saturate maximum positive (0x7FFFFFFF) or maximum negative (0x80000000) if the operation overflowed.
PM	The value in the PM bits sets the shift mode for the output operation from the product register. If the product shift value is positive (logical left shift operation), then the low bits are zero filled. If the product shift value is negative (arithmetic right shift operation), the upper bits are sign extended.

Repeat This instruction is repeatable. If the operation follows a RPT instruction, then it will be executed N+1 times. The state of the Z, N, C and OVC flags will reflect the final result. The V flag will be set if an intermediate overflow occurs. When repeated, the program-memory address is incremented by 1 during each repetition.

Example

```
; Calculate sum of product using 16-bit multiply:
; int16 X[N]         ; Data information
; int16 C[N]         ; Coefficient information, located in low 64K
; sum = 0;
; for(i=0; i < N; i++)
;    sum = sum + (X[i] * C[i]) >> 5;
  MOVL XAR2,#X       ; XAR2 = pointer to X
  SPM −5             ; Set product shift to ">> 5"
  ZAPA               ; Zero ACC, P, OVC
  RPT #N−1           ; Repeat next instruction N times
||MAC P,*XAR2++,0:C  ; ACC = ACC + P >> 5,
                     ; P = *XAR2++ * *C++
  ADDL ACC,P << PM   ; Perform final accumulate
  MOVL @sum,ACC      ; Store final result into sum
```

MAC P ,loc16,*XAR7/++ *Multiply and Accumulate*

Syntax Options

Syntax Options	Opcode	Objmode	RPT	CYC

MAC P, loc16, *XAR7	`0101 0110 0000 0111`
	`1100 0111 LLLL LLLL`
		1	Y	N+2

MAC P, loc16, *XAR7++	`0101 0110 0000 0111`
	`1000 0111 LLLL LLLL`
		1	Y	N+2

Operands

P – Product register

loc16 – Addressing mode (see Chapter 5)

***XAR7/++** – Indirect program-memory addressing using auxiliary register XAR7, can access full 4M x 16 program space range (0x000000 to 0x3FFFFF)

Description

Use the following steps for this instruction:

1. Add the previous product (stored in the P register), shifted as specified by the product shift mode (PM), to the ACC register.
2. Load the T register with the content of the location pointed to by the “loc16” addressing mode.
3. Multiply the signed 16-bit content of the T register by the signed 16-bit content of the program memory location pointed to by the XAR7 register and store the 32-bit result in the P register. If specified, post-increment the XAR7 register by 1:

```
ACC = ACC + P << PM;
T = [loc16];
P = signed T * signed Prog[*XAR7 or *XAR7++];
```

On the C28x devices, memory blocks are mapped to both program and data space (unified memory), hence the “XAR7/++” addressing mode can be used to access data space variables that fall within the program space address range.

With some addressing mode combinations, you can get conflicting references. In such cases, the C28x will give the “loc16/loc32” field priority on changes to XAR7. For example:

```
MAC P,*−−XAR7,*XAR7++  ; −−XAR7 given priority
MAC P,*XAR7++,*XAR7          ; *XAR7++ given priority
MAC P,*XAR7,*XAR7++          ; *XAR7++ given priority
```

Flags and Modes

Flags and Modes	Description
Z	After the addition, the Z flag is set if the ACC value is zero, else Z is cleared.
N	After the addition, the N flag is set if bit 31 of the ACC is 1, else N is cleared.
C	If the addition generates a carry, C is set; otherwise C is cleared.
V	If an overflow occurs, V is set; otherwise V is not affected.
OVC	If overflow mode is disabled; and if the operation generates a positive overflow, then the counter is incremented. If overflow mode is disabled; and if the operation generates a negative overflow, then the counter is decremented.
OVM	If overflow mode bit is set; then the ACC value will saturate maximum positive (0x7FFFFFFF) or maximum negative (0x80000000) if the operation overflowed.
PM	The value in the PM bits sets the shift mode for the output operation from the product register. If the product shift value is positive (logical left shift operation), then the low bits are zero filled. If the product shift value is negative (arithmetic right shift operation), the upper bits are sign extended.

Repeat

This instruction is repeatable. If the operation follows a RPT instruction, then it will be executed N+1 times. The state of the Z, N, C and OVC flags will reflect the final result. The V flag will be set if an intermediate overflow occurs.

Example

```
; Calculate sum of product using 16-bit multiply:
; int16 X[N]             ; Data information
; int16 C[N]             ; Coefficient information (located in low 4M)
; sum = 0;
; for(i=0; i < N; i++)
;    sum = sum + (X[i] * C[i]) >> 5;
  MOVL XAR2,#X           ; XAR2 = pointer to X
  MOVL XAR7,#C           ; XAR7 = pointer to C
  SPM −5                 ; Set product shift  to ">> 5"
  ZAPA                   ; Zero ACC, P, OVC
  RPT #N−1               ; Repeat next instruction N times
||MAC P,*XAR2++,*XAR7++  ; ACC = ACC + P >> 5,
                         ; P = *XAR2++ * *XAR7++
  ADDL ACC,P << PM       ; Perform final accumulate
  MOVL @sum,ACC          ; Store final result into sum
```

MAX AX, loc16 *Find the Maximum*

Syntax Options MAX AX, loc16

Opcode

```
0101 0110 0111
001A 0000 0000 LLLL LLLL
```

Objmode 1

RPT Y

CYC N+1

Operands

AX – Accumulator high (AH) or accumulator low (AL) register

loc16 – Addressing modes (see Chapter 5)

Description

Compare the signed contents of the specified AX register (AH or AL) with the signed content of the location pointed to by the “loc16” addressing mode and load the AX register with the larger of these two values:

```
if(AX < [loc16]), AX = [loc16];
if(AX >= [loc16]), AX = unchanged;
```

Flags and Modes

Flags and Modes	Description
N	If AX is less than the contents of the addressed location (AX <[loc16]) then the negative flag bit will be set; otherwise, it will be cleared.
Z	If AX and the contents of the addressed location are equal (AX = [loc16]) then the zero flag bit will be set; otherwise, it will be cleared.
V	If AX is less than the contents of the addressed location (AX <[loc16]) then the overflow flag bit will be set. This instruction cannot clear the V flag.

Repeat

If the operation is follows a RPT instruction, the instruction will be executed N+1 times. The state of the N, Z, and V flags will reflect the final result.

Example

```
; Saturate VarA as follows:
; if(VarA > 2000) VarA = 2000;
; if(VarA < −2000) VarA = −2000;
MOV AL,@VarA   ; Load AL with contents of VarA
MOV @AH,#2000  ; Load AH with the value 2000
MIN AL,@AH     ; if(AL > AH) AL = AH
NEG AH         ; AH = −2000
MAX AL,@AH     ; if(AL < AH) AL = AH
MOV @VarA,AL   ; Store result into VarA
```

MAXCUL P,loc32 *Conditionally Find the Unsigned Maximum*

Syntax Options MAXCUL P,loc32

Opcode

```
0101 0110 0101 0001
0000 0000 LLLL LLLL
```

Objmode 1

RPT –

CYC 1

Operands **P** – Product register

loc32 – Addressing mode (see Chapter 5)

Description Based on the state of the N and Z flags, conditionally compare the unsigned contents of the P register with the 32-bit, unsigned content of the location pointed to by the “loc32” addressing mode and load the P register with the larger of the two numbers:

```
if( (N=1) & (Z=0))
  P = [loc32];
if((N=0) & (Z=1) & (P < [loc32]))
  V=1, P = [loc32];
if((N=0) & (Z=0))
  P = unchanged;
```

Note: The “P < [loc32]” operation is treated like a 32-bit unsigned compare.

This instruction is typically combined with the MAXL instruction to form a 64-bit maximum function. It is assumed that the N and Z flags will first be set by using a MAXL instruction to compare the upper 32 bits of a 64-bit value. The MAXCUL instruction is then used to conditionally compare the lower 32 bits based on the results of the upper 32-bit comparison.

Flags and Modes

Flags and Modes	Description
N	If (N = 1 and z = 0) then load P with [loc32].
Z	If (N = 0 and Z = 1) compare the unsigned content of the P with the unsigned [loc32] and load P with the larger of the two. If (N = 0 and Z = 0) do nothing.
V	If (N = 0 AND Z = 1 AND P < [loc32]) then V is set; otherwise, V is unchanged.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Saturate 64-bit Var64 as follows:
; if(Var64 > MaxPos64) Var64 = MaxPos64
; if(Var64 < MaxNeg64) Var64 = MaxNeg64

MOVL ACC,@Var64+2     ; Load ACC:P with Var64
MOVL P,@Var64+0

MINL ACC,@MaxPos64+2  ; if(ACC:P > MaxPos64) ACC:P = MaxPos64
MINCUL P,@MaxPos64+0
SB saturate,OV
MAXL ACC,@MaxNeg64+2  ; if(ACC:P < MaxNeg64) ACC:P = MaxNeg64
MAXCUL P,@MaxNeg64+0
Saturate:
MOVL @Var64+2,ACC     ; Store result into Var64
MOVL @Var64,P
```

MAXL ACC,loc32 *Find the 32-bit Maximum*

Syntax Options MAXL ACC,loc32

Opcode

```
0101 0110 0110 0001
0000 0000 LLLL LLLL
```

Objmode 1

RPT Y

CYC N+1

Operands **ACC** – Accumulator register

loc32 – Addressing mode (see Chapter 5)

Description Compare the content of the ACC register with the location pointed to by the “loc32” addressing mode and load the ACC register with the larger of these two values:

```
if(ACC < [loc32]), ACC = [loc32];
if(ACC >= [loc32]), ACC = unchanged;
```

Flags and Modes

Flags and Modes	Description
Z	If ACC is equal to the contents of the addressed location (ACC = [loc32]), set Z; otherwise, clear Z.
N	If ACC is less than the contents of the addressed location, (ACC <[loc32]), set N; otherwise clear N. The MAXL instruction assumes infinite precision when it determines the sign of the result. For example, consider the subtraction 0x8000 0000 − 0x0000 0001. If the precision were limited to 32 bits, the result would cause an overflow to the positive number 0x7FFF FFFF and N would be cleared. However, because the MAXL instruction assumes infinite precision, it would set N to indicate that 0x8000 0000 − 0x0000 0001 actually results in a negative number.
C	If (ACC − [loc32]) generates a borrow, clear the C bit; otherwise set C.
V	If ACC is less than the contents of the addressed location (ACC <[loc32]), set V. This instruction cannot clear the V flag.

Repeat This instruction is repeatable. If the operation follows a RPT instruction, then the MAXL instruction will be executed N+1 times. The state of the Z, N, and C flags will reflect the final result. The V flag will be set if an intermediate overflow occurs.

Example

```
; Saturate VarA as follows:
; if(VarA > MaxPos) VarA = MaxPos
; if(VarA < MaxNeg) VarA = MaxNeg
MOVL ACC,@VarA    ; ACC = VarA
MINL ACC,@MaxPos  ; if(ACC > MaxPos) ACC = MaxPos
MAXL ACC,@MaxNeg  ; if(ACC < MaxNeg) ACC = MaxNeg
MOVL @VarA,ACC    ; Store result into VarA
```

MIN AX, loc16 *Find the Minimum*

Syntax Options MIN AX, loc16

Opcode

```
0101 0110 0111 010A
0000 0000 LLLL LLLL
```

Objmode 1

RPT Y

CYC N+1

Operands **AX** – Accumulator high (AH) or accumulator low (AL) register

loc16 – Addressing modes (see Chapter 5)

Description Compare the signed content of the specified AX register (AH or AL) with the content of the signed location pointed to by the “loc16” addressing mode and load the AX register with the smaller of these two values:

```
if(AX > [loc16]), AX = [loc16];
if(AX <=[loc16]), AX = unchanged;
```

Flags and Modes

Flags and Modes	Description
N	If AX is less than the contents of the addressed location (AX <[loc16]) then the negative flag bit will be set; otherwise, it will be cleared.
Z	If AX and the contents of the addressed location are equal (AX = [loc16]) then the zero flag bit will be set; otherwise, it will be cleared.
V	If AX is greater then the contents of the addressed location (AX >[loc16]) then the overflow flag bit will be set. This instruction cannot clear the V flag.

Repeat If the operation is follows a RPT instruction, the instruction will be executed N+1 times. The state of the N, Z and V flags will reflect the final result.

Example

```
; Saturate VarA as follows:
; if(VarA > 2000) VarA = 2000;
; if(VarA < −2000) VarA = −2000;
MOV AL,@VarA   ; Load AL with contents of VarA
MOV @AH,#2000  ; Load AH with the value 2000
MIN AL,@AH     ; if(AL > AH) AL = AH
NEG AH         ; AH = −2000
MAX AL,@AH     ; if(AL < AH) AL = AH
MOV @VarA,AL   ; Store result into VarA
```

MINCUL P,loc32 *Conditionally Find the Unsigned Minimum*

Syntax Options MINCUL P,loc32

Opcode

```
0101 0110 0101 1001
xxxx xxxx LLLL LLLL
```

Objmode 1

RPT –

CYC 1

Operands

P – Product register

loc32 – Addressing mode (see <CrossReference href="#SPRU4307810"/>)

Description

Based on the state of the N and Z flags, conditionally compare the unsigned contents of the P register with the 32-bit, unsigned content of the location pointed to by the “loc32” addressing mode and load the P register with the smaller of the two numbers:

```
if( (N = 0) & (Z = 0))
  P = [loc32];
if((N = 0) & (Z = 1) & (P > [loc32]))
  V=1, P = [loc32];
if((N = 1) & (Z = 0))
  P = unchanged;
```

Note: The “p < [loc32]” operation is treated like a 32-bit unsigned compare.

This instruction is typically combined with the MINL instruction to form a 64-bit minimum function. It is assumed that the N and Z flags will first be set by using a MINL instruction to compare the upper 32 bits of a 64-bit value. The MINCUL instruction is then used to conditionally compare the lower 32 bits based on the results of the upper 32-bit comparison.

Flags and Modes

Flags and Modes	Description
N	If (N = 1 AND Z = 0), then load the P register with [loc32].
Z	If (N = 0 AND Z =1), compare unsigned and load P with the smaller P register to [loc32]. If (N = 0 AND Z = 0), do nothing.
V	If (N = 0 AND Z = 1 AND P <loc32]) then V is set; otherwise, V is unchanged.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Saturate 64-bit Var64 as follows:
; if(Var64 > MaxPos64 ) Var64 = MaxPos64
; if(Var64 < MaxNeg64 ) Var64 = MaxNeg64
MOVL ACC,@Var64+2     ; Load ACC:P with Var64
MOVL P,@Var64+0
MINL ACC,@MaxPos64+2  ; if(ACC:P > MaxPos64) ACC:P = MaxPos64
MINCUL P,@MaxPos64+0
MAXL ACC,@MaxNeg64+2  ; if(ACC:P < MaxNeg64) ACC:P = MaxNeg64
MAXCUL P,@MaxNeg64+0
MOVL @Var64+2,ACC     ; Store result into Var64
MOVL @Var64+0,P
```

MINL ACC,loc32 *Find the 32-bit Minimum*

Syntax Options MINL ACC,loc32

Opcode

```
0101 0110 0101 0000
0000 0000 LLLL LLLL
```

Objmode 1

RPT Y

CYC N+1

Operands

ACC – Accumulator register

loc32 – Addressing mode (see Chapter 5)

Description

Compare the content of the ACC register with the location pointed to by the “loc32” addressing mode and load the ACC register with the smaller of these two values:

```
if(ACC <= [loc32]), ACC = unchanged;
if(ACC > [loc32]), ACC = [loc32];
```

Flags and Modes

Flags and Modes	Description
Z	If ACC is equal to the contents of the addressed location (ACC = [loc32]), set Z; otherwise clear Z.
N	If ACC is less than the contents of the addressed location, (ACC <[loc32]), set N; otherwise clear N. The MINL instruction assumes infinite precision when it determines the sign of the result. For example, consider the subtraction 0x8000 0000 − 0x0000 0001. If the precision were limited to 32 bits, the result would cause an overflow to the positive number 0x7FFF FFFF and N would be cleared. However, because the MINL instruction assumes infinite precision, it would set N to indicate that 0x8000 0000 − 0x0000 0001 actually results in a negative number.
C	If (ACC − [loc32]) generates a borrow, clear the C bit; otherwise set C.
V	If ACC is greater than the contents of the addressed location (ACC <[loc32]), set V. This instruction cannot clear the V flag.

Repeat

This instruction is repeatable. If the operation follows a RPT instruction, then the MINL instruction will be executed N+1 times. The state of the Z, N, and C flags will reflect the final result. The V flag will be set if an intermediate overflow occurs.

Example

```
; Saturate VarA as follows:
; if(VarA > MaxPos) VarA = MaxPos
; if(VarA < MaxNeg) VarA = MaxNeg
MOVL ACC,@VarA    ; ACC = VarA
MINL ACC,@MaxPos  ; if (ACC > MaxPos) ACC = MaxPos
MAXL ACC,@MaxNeg  ; if (ACC < MaxNeg) ACC = MaxNeg
MOVL @VarA,ACC    ; Store result into VarA
```

MOV *(0:16bit), loc16 *Move Value*

Syntax Options

MOV *(0:16bit), loc16

Opcode

```
1111 0100 LLLL LLLL
CCCC CCCC CCCC CCCC
```

Objmode

X

RPT

Y

CYC

N+2

Operands

***(0:16bit)** – Immediate direct memory address, access low 64K range of data space only (0x00000000 to 0x0000FFFF)

loc16 – Addressing mode (see Chapter 5)

Description

Move the content of the location pointed to by the “loc16” addressing mode to the memory location specified by the ”0:16bit” constant address:

```
[0x0000:16bit] = [loc16];
```

Flags and Modes

None

Repeat

This instruction is repeatable. If the operation follows a RPT instruction, then it will be executed N+1 times. When repeated, the “(0:16bit)” data-memory address is post-incremented by 1 during each repetition. Only the lower 16 bits of the address is affected.

```
; Copy the contents of Array1 to Array2:
; int16 Array1[N];
; int16 Array2[N];    // Located in low 64K of data space
; for(i=0; i < N; i++)
; Array2[i] = Array1[i];
```

Example

```
MOVL XAR2,#Array1           ; XAR2 = pointer to Array1
RPT #(N−1)                  ; Repeat next instruction N times
||MOV *(0:Array2),*XAR2++   ; Array2[i] = Array1[i],
                            ; i++
```

MOV ACC,#16bit<<#0..15 *Load Accumulator With Shift*

Syntax Options

MOV ACC,#16bit<<#0..15

Opcode

```
1111 1111 0010 SHFT
CCCC CCCC CCCC CCCC
```

Objmode

X

RPT

–

CYC

1

Operands

ACC – Accumulator register

#16bit – 16-bit immediate constant value

#0..15 – Shift value (default is "<< #0” if no value specified)

Description

Load the ACC register with the left shifted contents of the 16-bit immediate value. The shifted value is sign extended if sign extension mode is turned on (SXM = 1) else the shifted value is zero extended (SXM = 0). The lower bits of the shifted value are zero filled:

```
if(SXM = 1)  // sign extension mode enabled
  ACC = S:16bit << shift value;
else         // sign extension mode disabled
  ACC = 0:16bit << shift value;
```

Flags and Modes

Flags and Modes	Description
N	After the load, the N flag is set if bit 31 of the ACC is 1, else N is cleared.
Z	After the load, the Z flag is set if the ACC value is zero, else Z is cleared.
SXM	If sign extension mode bit is set; then the 16-bit constant operand will be sign extended before the load; else, the value will be zero extended.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Calculate signed value: ACC = −2010 << 10 + VarB << 6;
SETC SXM               ; Turn sign extension mode on
MOV ACC,#−2010 << #10  ; Load ACC with −2010 left shifted by 10
ADD ACC,@VarB << #6    ; Add VarB left shifted by 6 to ACC
```

MOV ACC,loc16<<T *Load Accumulator With Shift*

Syntax Options

MOV ACC,loc16<<T

Opcode

```
0101 0110 0000 0110
0000 0000 LLLL LLLL
```

Objmode

1

RPT

–

CYC

1

Operands

ACC – Accumulator register

loc16 – Addressing mode (see Chapter 5)

T – Upper 16 bits of the multiplicand register, XT(31:16)

Description

Load the ACC register with the left-shifted contents of the 16-bit location pointed to by the "loc16" addressing mode. The shift value is specified by the four least significant bits of the T register, T(3:0) = shift value = 0..15. Higher order bits are ignored. The shifted value is sign extended if sign extension mode is turned on (SXM = 1) else the shifted value is zero extended (SXM = 0). The lower bits of the shifted value are zero filled:

```
if(SXM = 1)  // sign extension mode enabled
  ACC = S:[loc16] << T(3:0);
else         // sign extension mode disabled
  ACC = 0:[loc16] << T(3:0);
```

Flags and Modes

Flags and Modes	Description
N	After the load, the N flag is set if bit 31 of the ACC is 1, else N is cleared.
Z	After the load, the Z flag is set if the ACC value is zero, else Z is cleared.
SXM	If sign extension mode bit is set; then the 16-bit operand, addressed by the "loc16" field, will be sign extended before the load; else the value will be zero extended.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Calculate signed value: ACC = (VarA << SB) + (VarB << SB)
SETC SXM            ; Turn sign extension mode on
MOV T,@SA           ; Load T with shift value in SA
MOV ACC,@VarA << T  ; Load in ACC shifted contents of VarA MOV    T,@SB
                    ; Load T with shift value in SB
ADD ACC,@VarB << T  ; Add to ACC shifted contents of VarB
```

MOV ACC, loc16<<#0..16 *Load Accumulator With Shift*

Syntax Options

Syntax Options	Opcode	Obj- Mode	RPT	CYC

MOV ACC,loc16<<#0	`1000 0101 LLLL LLLL`
	`1110 0000 LLLL LLLL`
		1
		0
			–
			–
				1
				1

MOV ACC, loc16<<#1..15	`0101 0110 0000 0011`
	`0000 SHFT LLLL LLLL`
		1	–	1

	`1110 SHFT LLLL LLLL`	0	–	1
MOV ACC, loc16<<#16	`0010 0101 LLLL LLLL`	X	–	1

Operands

ACC – Accumulator register

loc16 – Addressing mode (see Chapter 5)

#0..16 – Shift value (default is "<< #0” if no value specified)

Description

Load the ACC register with the left shifted contents of the addressed location pointed to by the “loc16” addressing mode. The shifted value is sign extended if sign extension mode is turned on (SXM = 1) else the shifted value is zero extended (SXM = 0). The lower bits of the shifted value are zero filled:

```
if(SXM = 1)  // sign extension mode enabled
  ACC = S:[loc16] << shift value;
else         // sign extension mode disabled
  ACC = 0:[loc16] << shift value;
```

Flags and Modes

Flags and Modes	Description
N	After the load, the N flag is set if bit 31 of the ACC is 1, else N is cleared.
Z	After the load, the Z flag is set if the ACC is zero, else Z is cleared.
SXM	If sign extension mode bit is set; then the 16-bit operand, addressed by the ”loc16” field, will be sign extended before the load; else the value will be zero extended.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Calculate signed value: ACC = VarA << 10 + VarB << 6;
SETC SXM              ; Turn sign extension mode on
MOV ACC,@VarA << #10  ; Load ACC with VarA left shifted by 10
ADD ACC,@VarB << #6   ; Add VarB left shifted by 6 to ACC
```

MOV AR6/7, loc16 *Load Auxiliary Register*

Syntax Options

Syntax Options	Opcode	Objmode	RPT	CYC
MOV AR6, loc16	`0101 1110 LLLL LLLL`	X	–	1
MOV AR7, loc16	`0101 1111 LLLL LLLL`	X	–	1

Operands

AR6/7 – AR6 or AR7, auxiliary registers

loc16 – Addressing mode (see Chapter 5)

Description

Load AR6 or AR7 with the contents of the 16-bit location and leave the upper 16 bits of XAR6 and XAR7 unchanged:

```
AR6/7 = [loc16]; AR6/7H = unchanged;
```

Flags and Modes

None

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

MOV AX, loc16 *Load AX*

Syntax Options MOV AX, loc16

Opcode `1001 001A LLLL LLLL`

Objmode X

RPT –

CYC 1

Operands

AX – Accumulator high (AH) or accumulator low (AL) register

loc16 – Addressing mode (see Chapter 5)

Description

Load accumulator high register (AH) or accumulator low register (AL) register with the 16-bit contents of the location pointed to by the “loc16” addressing mode, leaving the other half of the accumulator register unchanged:

```
AX = [loc16];
```

Flags and Modes

Flags and Modes	Description
N	The load to AX is tested for a negative condition. If bit 15 of AX is 1, then this flag is set; otherwise it is cleared.
Z	The load to AX is tested for a zero condition. The bit is set if the operation results in AX = 0, otherwise it is cleared.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
MOV AH, *+XAR0[0]  ; Load AH with the 16-bit contents
                   ; of location pointed to by XAR0.
                   ; AL is unchanged.
SB NotZero,NEQ     ; Branch if contents of AH were non
                   ; zero.
```

MOV DP, #10bit *Load Data-Page Pointer*

Syntax Options MOV DP, #10bit

Opcode

```
1111 10CC CCCC CCCC
```

Objmode X

RPT –

CYC 1

Operands **DP** – Data page register

#10bit – 10-bit immediate constant value

Description Load the data page register with a 10-bit constant leaving the upper 6 bits unchanged:

```
DP(9:0) = 10bit;
DP(15:10) = unchanged;
```

Flags and Modes None

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
MOV DP, #VarA  ; Load DP with the data page that
               ; contains VarA. Assumes VarA is in
               ; the lower 0x0000 FFC0 of memory.
               ; DP(15:10) is left unchanged.
```

MOV IER,loc16 *Load the Interrupt-Enable Register*

Syntax Options

MOV IER,loc16

Opcode

```
0010 0011 LLLL LLLL
```

Objmode

X

RPT

–

CYC

5

Operands

IER – Interrupt-enable register

loc16 – Addressing mode (see Chapter 5)

Description

Enable and disable selected interrupts by loading the content of the location pointed to by the “loc16” addressing mode into the IER register. Any changes take effect before the next instruction is processed.

```
IER = [loc16];
```

Flags and Modes

None

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Push the contents of IER on the stack and load IER with the
; contents of VarA:
MOV *SP++,IER  ; Save IER on stack
MOV IER,@VarA  ; Load IER with contents of VarA
```

MOV loc16, #16bit *Save 16-bit Constant*

Syntax Options MOV loc16, #16bit

Opcode

```
0010 1000 LLLL LLLL
CCCC CCCC CCCC CCCC
```

Objmode X

RPT Y

CYC N+1

Operands **loc16** – Addressing mode (see Chapter 5)

#16bit – 16-bit constant immediate value

Description Load the location pointed to by the “loc16” addressing mode with the 16-bit constant immediate value:

```
[loc16] = 16bit;
```

Note: For #16bit = #0, see the MOV loc16, #0 instruction on page 6-166.

Smart Encoding:

If loc16 = AL or AH and #16bit is an 8-bit number, then the assembler will encode this instruction as MOVB AX, #8bit to improve efficiency. To override this, use the MOVW AX, #16bit alias instruction.

Flags and Modes

Flags and Modes	Description
N	If (loc16 = @AX), then the load to AX is tested for a negative condition. The negative flag bit is set if bit 15 of AX is 1, otherwise it is cleared.
Z	If (loc16 = @AX), then the load to AX is tested for a zero condition. The bit is set if the result of the operation on the AX register generates a 0 value, otherwise it is cleared.

Repeat If this operation follows a RPT instruction, then it will be executed N+1 times. The state of the N and Z flags will reflect the final result.

Example

```
; Initialize the contents of Array1 with 0xFFFF:
; int16 Array1[N];
; for(i=0; i < N; i++)
;    Array1[i] = 0xFFFF;
  MOVL XAR2,#Array1    ; XAR2 = pointer to Array1
  RPT #(N−1)           ;  Repeat next instruction N times
||MOV *XAR2++,#0xFFFF  ; Array1[i] = 0xFFFF,
                       ; i++
```

MOV loc16, *(0:16bit) *Move Value*

Syntax Options MOV loc16, *(0:16bit)

Opcode

```
1111 0101 LLLL LLLL
CCCC CCCC CCCC CCCC
```

Objmode X

RPT Y

CYC N+2

Operands **loc16** – Addressing mode (see Chapter 5)

***(0:16bit)** – Immediate direct memory address, access low 64K range of data space only (0x00000000 to 0x0000FFFF)

Description Move the content of the location specified by the constant direct memory address “0:16bit” into the location pointed to by the “loc16” addressing mode:

```
[loc16] = [0x0000:16bit];
```

Flags and Modes

Flags and Modes	Description
N	If (loc16 = @AX), then the load to AX is tested for a negative condition. The negative flag bit is set if bit 15 of AX is 1, otherwise it is cleared.
Z	If (loc16 = @AX), then the load to AX is tested for a zero condition. The bit is set if the result of the operation on the AX register generates a 0 value, otherwise it is cleared.

Repeat This instruction is repeatable. If the operation follows a RPT instruction, then it will be executed N+1 times. When repeated, the “(0:16bit)” data-memory address is post-incremented by 1 during each repetition. Only the lower 16 bits of the address are affected.

Example

```
; Copy the contents of Array1 to Array2:
; int16 Array1[N];    // Located in low 64K of data space
; int16 Array2 N];
; for(i=0; i < N; i++)
;    Array2[i] = Array1[i];

MOVL XAR2,#Array2          ; XAR2 = pointer to Array2
RPT #(N−1)                 ; Repeat next instruction N times
||MOV *XAR2++,*(0:Array1)  ; Array2[i] = Array1[i],
                           ; i++
```

MOV loc16, #0 *Clear 16-bit Location*

Syntax Options MOV loc16, #0

Opcode `0010 1011 LLLL LLLL`

Objmode X

RPT Y

CYC N+1

Operands **loc16** – Addressing mode (see Chapter 5)

#0 – Immediate constant value of zero

Description Load the location pointed to by the “loc16” addressing mode with the value 0x0000:

```
[loc16] = 0x0000;
```

Flags and Modes

Flags and Modes	Description
N	If (loc16 = @AX), then the load to AX is tested for a negative condition. The negative flag bit is set if bit 15 of AX is 1, otherwise it is cleared.
Z	If (loc16 = @AX), then the load to AX is tested for a zero condition. The bit is set if the result of the operation on the AX register generates a 0 value, otherwise it is cleared.

Repeat This instruction is repeatable. If the operation is follows a RPT instruction, then it will be executed N+1 times.

Example

```
; Initialize the contents of Array1 with zero:
; int16 Array1[N];
; for(i=0; i < N; i++)
;    Array1[i] = 0;

  MOVL XAR2,#Array1  ; XAR2 = pointer to Array1
  RPT #(N-1)         ; Repeat next instruction N times
||MOV *XAR2++,#0     ; Array1[i] = 0,
                     ; i++
```

MOV loc16,ACC << 1..8 *Save Low Word of Shifted Accumulator*

Syntax Options

Syntax Options	Opcode	Objmode	RPT	CYC
MOV loc16, ACC << 1	`1011 0001 LLLL LLLL`	1	Y	N+1

MOV loc16, ACC << 2..8	`0101 0110 0010 1101`
	`0000 0SHF LLLL LLLL`
		1	Y	N+1

	`1011 1SHF LLLL LLLL`	0	–	1

Operands

loc16 – Addressing mode (see Chapter 5)

ACC – Accumulator register

#1..8 – Shift value

Description

Load the content of the location pointed to by the “loc16” addressing mode with the low word of the ACC register after left−shifting by the specified value. The ACC register is not modified:

```
[loc16] = ACC >> (16 – shift value); [loc16] = low (ACC <<1...8)
```

Flags and Modes

Flags and Modes	Description
N	If (loc16 = @AX), then after the load AX is checked for a negative condition. The N flag is set if bit 15 of the AX is 1; else N is cleared.
Z	If (loc16 = @AX) then after the load AX is checked for a zero condition. The Z flag is set if AX is zero; else Z is cleared.

Repeat

If the operation is repeatable, then the instruction will be executed N+1 times. The state of the Z and N flags will reflect the final result. If the operation is not repeatable, the instruction will execute only once.

Example

```
; Multiply two Q15 numbers (VarA and VarB) and store result in
; VarC as a Q15 number:
MOV T,@VarA             ; T = VarA (Q15)
MPY ACC,T,@VarB         ; ACC = VarA * VarB (Q30)
MOVH @VarC,ACC << 1     ; VarC = ACC >> 16–1) (Q15)
                        ; VarC as a Q31 number:
MOV T,@VarA             ; T = VarA (T = Q14)
MPY ACC,T,@VarB         ; ACC = VarA * VarB    (ACC = Q28)
MOV @VarC+0,ACC << 3    ; VarC low = ACC << 3
MOVH @VarC+1,ACC << 3   ; VarC high = ACC >> (16–1) (VarC = Q31)
```

MOV loc16, ARn *Store 16-bit Auxiliary Register*

Syntax Options MOV loc16, ARn

Opcode `0111 1nnn LLLL LLLL`

Objmode X

RPT –

CYC 1

Operands **loc16** – Addressing mode (see Chapter 5)

ARn – AR0 to AR7, lower 16 bits of auxiliary registers

Description Load the contents of the 16-bit location with ARn:

```
[loc16] = ARn;
```

If(loc16 = @ARn), then only the lower 16 bits of the selected auxiliary register is modified. The upper 16 bits is unchanged.

Flags and Modes

Flags and Modes	Description
N	If (loc16 = @AX), then the load to AX is tested for a negative condition. Bit-15 of the AX register is the sign bit, 0 for positive, 1 for negative. The negative flag bit is set if the operation on the AX register generates a negative value, otherwise it is cleared.
Z	If (loc16 = @AX), then the load to AX is tested for a zero condition. The bit is set if the result of the operation on the AX register generates a 0 value, otherwise it is cleared.

Repeat This instruction is not repeatable. If the operation is follows a RPT instruction, it resets the repeat counter (RPTC) and only executes once.

Example

```
MOV @AL, AR3     ; Load AL with the 16-bit contents of
                 ; AR3. If bit 15 of AL is 1, set the
                 ; N flag, else clear it.
                 ; If AL is 0, set the Z flag.
MOV @AR4,AR3     ; Load AR4 with the value in AR3.
                 ; Upper 16 bits of XAR4 are
                 ; unchanged.
MOV *SP++,AR3    ; Push the contents of AR3 onto the
                 ; stack. Post increment SP.
MOV *XAR4++,AR4  ; Store contents of AR4 into location
                 ; specified by XAR4. Post-increment
                 ; the contents of XAR4.
MOV *−−XAR5,AR5  ; Pre-decrement the contents of XAR5.
                 ; Store the contents of AR5 into the
                 ; location specified by XAR5.
```

MOV loc16, AX *Store AX*

Syntax Options MOV loc16, AX

Opcode `1001 011A LLLL LLLL`

Objmode X

RPT Y

CYC N+1

Operands **loc16** – Addressing mode (see Chapter 5)

AX – Accumulator high (AH) or accumulator low (AL) register

Description Load the addressed location pointed to by the “loc16” addressing mode with the 16-bit content of the specified AX register (AH or AL):

```
[loc16] = AX;
```

Flags and Modes

Flags and Modes	Description
N	If (loc16 = @AX), then the load to AX is tested for a negative condition. The negative flag bit is set if bit 15 of AX is 1, otherwise it is cleared.
Z	If (loc16 = @AX), then the load to AX is tested for a zero condition. The bit is set if the result of the operation on the AX register generates a 0 value, otherwise it is cleared.

Repeat If this operation follows a RPT instruction, then it will be executed N+1 times. The state of the N and Z flags will reflect the final result.

Example

```
; Initialize all Array1 elements with the value 0xFFFF:
  MOV AH,#0xFFFF     ; Load AH with the value 0xFFFF
  MOVL XAR2,#Array1  ; Load XAR2 with address of Array1
  RPT #9             ; Repeat next instruction 10 times.

|| MOV *XAR2++, AH   ; Store contents of AH into location
                     ; pointed by XAR2 and post-increment
                     ; XAR2.
```

MOV loc16, AX, COND *Store AX Register Conditionally*

Syntax Options MOV loc16, AX, COND

Opcode

```
0101 0110 0010 101A
0000 COND LLLL LLLL
```

Objmode 1

RPT –

CYC 1

Operands

loc16 – Addressing mode (see Chapter 5)

AX – Accumulator high (AH) or accumulator low (AL) register

COND – Conditional codes:

COND	Syntax	Description	Flags Tested
0000	NEQ	Not Equal To	Z = 0
0001	EQ	Equal To	Z = 1
0010	GT	Greater Than	Z = 0 AND N = 0
0011	GEQ	Greater Than or Equal To	N = 0
0100	LT	Less Than	N = 1
0101	LEQ	Less Than or Equal To	Z = 1 OR N = 1
0110	HI	Higher	C = 1 AND Z = 0
0111	HIS, C	Higher or Same, Carry Set	C = 1
1000	LO, NC	Lower, Carry Clear	C = 0
1001	LOS	Lower or Same	C = 0 OR Z = 1
1010	NOV	No Overflow	V = 0
1011	OV	Overflow	V = 1
1100	NTC	Test Bit Not Set	TC = 0
1101	TC	Test Bit Set	TC = 1
1110	NBIO	BIO Input Equal To Zero	BIO = 0
1111	UNC	Unconditional	–

Description

If the specified condition being tested is true, then the location pointed to by the “loc16” addressing mode will be loaded with the contents of the specified AX register (AH or AL):

```
if(COND = true) [loc16] = AX;
```

Note: Addressing modes are not conditionally executed. Hence, if an addressing mode performs a pre or post modification, the modification will occur, regardless of whether the condition is true or not.

Flags and Modes

Flags and Modes	Description
N	If (COND = true AND loc16 = @AX), AX is tested for a negative condition after the move and if bit 15 of AX is 1, the negative flag bit is set.
Z	If (COND = true AND loc16 = @AX), after the move, AX is tested for a zero condition and the zero flag bit is set if AX = 0, otherwise, it is cleared.
V	If the V flag is tested by the condition, then V is cleared.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Swap the contents of VarA and VarB if VarB is higher then VarA:
MOV AL,@VarA     ; AL = VarA, XAR2 points to VarB
MOV AH,@VarB     ; AH = VarB, XAR2 points to VarA
CMP AH,@AL       ; Compare AH and AL
MOV @VarA,AH,HI  ; Store AH in VarA if higher
MOV @VarB,AL,HI  ; Store AL in VarB if higher
```

MOV loc16,IER *Store Interrupt-Enable Register*

Syntax Options MOV loc16,IER

Opcode `0010 0000 LLLL LLLL`

Objmode X

RPT –

CYC 1

Operands **loc16** – Addressing mode (see Chapter 5)

IER – Interrupt enable register

Description Save the content of the IER register in the location pointed to by the “loc16” addressing mode:

```
[loc16] = IER;
```

Flags and Modes

Flags and Modes	Description
N	If (loc16 = @AX) and bit 15 of AX is 1, then N is set; otherwise N is cleared.
Z	If (loc16 = @AX) and the value of AX is zero, then Z is set; otherwise Z is cleared.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Push the contents of IER on the stack and load IER with the
; contents of VarA:
  MOV *SP++,IER  ; Save IER on stack
  MOV IER,@VarA  ; Load IER with contents of VarA
```

MOV loc16,OVC *Store the Overflow Counter*

Syntax Options MOV loc16,OVC

Opcode

```
0101 0110 0010 1001
0000 0000 LLLL LLLL
```

Objmode 1

RPT –

CYC 1

Operands

loc16 – Addressing mode (see Chapter 5)

OVC – Overflow counter

Description

Store the 6 bits of the overflow counter (OVC) into the upper 6 bits of the location pointed to by the “loc16” addressing mode and zero the lower 10 bits of the addressed location:

```
[loc16(15:10)] = OVC; [loc16(9:0)]    = 0;
```

Flags and Modes

Flags and Modes	Description
N	If (loc16 = @AX) and bit 15 of AX is 1, then set N; otherwise clear N.
Z	If (loc16 = @AX) and AX is zero, then set Z; otherwise clear Z.

Repeat

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Save and restore contents of ACC and OVC bits:
MOV *SP++,OVC  ; Save OVC on stack
MOV *SP++,AL   ; Save AL on stack
MOV *SP++,AH   ; Save AH on stack
.
.
.
.
MOV AH,*--SP   ; Restore AH from stack
MOV AL,*--SP   ; Restore AL from stack
MOV OVC,*--SP  ; Restore OVC from  stack
```

MOV loc16,P *Store Lower Half of Shifted P Register*

Syntax Options MOV loc16,P

Opcode `0011 1111 LLLL LLLL`

Objmode X

RPT Y

CYC N+1

Operands **loc16** – Addressing mode (see Chapter 5)

P – Product register

Description The contents of the P register are shifted by the amount specified in the product shift mode (PM), and the lower half of the shifted value is stored into the 16-bit location pointed to by the “loc16” addressing mode. The P register is not modified by the operation:

```
[loc16] = P << PM;
```

Flags and Modes

Flags and Modes	Description
N	If (loc16 = @AX) and bit 15 of the AX register is 1, then the N bit is set; otherwise, N is cleared.
Z	If (loc16 = @AX) and the value of AX after the load is zero, then the Z bit is set; otherwise Z is cleared.
PM	The value in the PM bits sets the shift mode for the output operation from the product register. If the product shift value is positive (logical left shift operation), then the low bits are zero filled. If the product shift value is negative (arithmetic right shift operation), the upper bits are sign extended.

Repeat This instruction is repeatable. If the operation follows a RPT instruction, then it will be executed N+1 times. The state of the Z, and N flags will reflect the final result.

Example

```
; Calculate Y32 = M16*X16 >> 6
MOV T,@M16    ; T = M
MPY P,T,@X16  ; P = T * X
SPM −6        ; Set product shift to >> 6
MOV @Y32+0,P  ; Y32 = P >> 6
MOVH @Y32+1,P
```

MOV loc16, T *Store the T Register*

Syntax Options MOV loc16, T

Opcode `0010 0001 LLLL LLLL`

Objmode X

RPT –

CYC 1

Operands **loc16** – Addressing mode (see Chapter 5)

T – Upper 16 bits of the multiplicand register (XT)

Description Store the 16-bit T register contents into the location pointed to by the “loc16” addressing mode:

```
[loc16] = T;
```

Flags and Modes

Flags and Modes	Description
N	If (loc16 = @AX) and bit 15 of the AX register is 1, then the N bit is set; otherwise, N is cleared.
Z	If (loc16 = @AX) and the value of AX after the load is zero, then the Z bit is set; otherwise Z is cleared.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Calculate using 16-bit multiply:
; Y = (X0*C0) >> 2) + (X1*C1 >> 2) + (X2*C2 >> 2)
; X2 = X1
; X1 = X0
SPM –2              ; Set  product shift to  >> 2
MOV T,@X2           ; T = X2
MPY P,T,@C2         ; P = T*C2
MOVP T,@X1          ; T = X1, ACC = X2*C2 >> 2
MPY P,T,@C1         ; P = T*C1
MOV @X2,T           ; X2 = X1
MOVA T,@X0          ; T = X0, ACC = X1*C1 >> 2 + X2*C2 >> 2
MPY P,T,@C0         ; P = T*C0
MOV @X1,T           ; X1 = X0
ADDL ACC, P << PM   ; ACC = X0*C0 >> 2 + X1*C1 >> 2 + X2*C2 >> 2
MOVL @Y,ACC         ; Store result into Y
```

MOV OVC, loc16 *Load the Overflow Counter*

Syntax Options MOV OVC, loc16

Opcode

```
0101 0110 0000 0010
0000 0000 LLLL LLLL
```

Objmode 1

RPT –

CYC 1

Operands **OVC** – 6-bit overflow counter

Description Load the overflow counter (OVC) with the upper 6 bits of the location pointed to by the “loc16” addressing mode:

```
OVC = [loc16(15:10)];
```

Flags and Modes

Flags and Modes	Description
OVC	The 6-bit overflow counter is modified.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Save and restore contents of ACC and OVC bits:
MOV *SP++,OVC  ; Save OVC on stack
MOV *SP++,AL   ; Save AL on stack
MOV *SP++,AH   ; Save AH on stack
.
.
.
.

MOV AH,*−−SP   ; Restore AH from stack
MOV AL,*−−SP   ; Restore AL from stack
MOV OVC,*−−SP  ; Restore OVC from  stack
```

MOV PH, loc16 *Load the High Half of the P Register*

Syntax Options MOV PH, loc16

Opcode

```
0010 1111 LLLL LLLL
```

Objmode X

RPT –

CYC 1

Operands

PH – Upper 16 bits of the product register (P)

loc16 – Addressing mode (see Chapter 5)

Description

Load the high 16 bits of the P register (PH) with the 16-bit location pointed to by the "loc16" addressing mode; leave the lower 16 bits (PL) unchanged:

```
PH = [loc16];
PL = unchanged;
```

Flags and Modes None

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Swap the contents of AH and AL:
MOV PH,@AL  ; Load PH with AL
MOV PL,@AH  ; Load PL with AH
MOV ACC,@P  ; Load ACC with P (AH and AL swapped)
```

MOV PL, loc16 *Load the Low Half of the P Register*

Syntax Options MOV PL, loc16

Opcode `0010 0111 LLLL LLLL`

Objmode X

RPT –

CYC 1

Operands

PL – Lower 16 bits of the product register (P)

loc16 – Addressing mode (see Chapter 5)

Description

Load the high 16 bits of the P register (PL) with the 16-bit location pointed to by the “loc16” addressing mode; leave the lower 16 bits (PH) unchanged:

```
PL = [loc16];
PH = unchanged;
```

Flags and Modes None

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Swap the contents of AH and AL:
MOV PH,@AL  ; Load PH with AL
MOV PL,@AH  ; Load PL with AH
MOV ACC,@P  ; Load ACC with P (AH and AL swapped)
```

MOV PM, AX *Load Product Shift Mode*

Syntax Options MOV PM, AX

Opcode `0101 0110 0011 100A`

Objmode 1

RPT –

CYC 1

Operands **AX** – Accumulator high (AH) or accumulator low (AL) registers.

Description Load the product shift mode (PM) bits with the 3 least significant bits of register AX.

```
PM = AX(2:0);
```

Flags and Modes

Flags and Modes	Description
PM	The product shift mode bits are loaded with the 3 least significant bits of AX.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Calculate: Y32 = (M16*X16 >> Shift) + B32, Shift = 0 to 6
CLRC AMODE         ; Make sure AMODE = 0
MOV AL,@Shift      ; Load AL with contents of "Shift" ADDB AL,#1
                   ; Convert "Shift" to PM encoding
MOV PM,AX          ; Load PM bits with encoded "Shift" value
MOV T,@X16         ; T = X16
MPY P,XT,@M16      ; P = X16*M16
MOVL ACC,@B32      ; ACC = B32
ADDL ACC,P << PM   ; ACC = ACC + (P >> Shift)
MOVL @Y32,ACC      ; Store result into Y32
```

MOV T, loc16 *Load the Upper Half of the XT Register*

Syntax Options MOV T, loc16

Opcode `0010 1101 LLLL LLLL`

Objmode X

RPT –

CYC 1

Operands **T** – Upper 16 bits of the multiplicand register (XT)

loc16 – Addressing mode (see Chapter 5)

Description Load the T register with the 16-bit contents of the location pointed to by the “loc16” addressing mode:

```
T = [loc16];
```

Flags and Modes None

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Calculate using 16-bit multiply:
; Y = (X0*C0) >> 2) + (X1*C1 >> 2) + (X2*C2 >> 2)
; X2 = X1
; X1 = X0
SPM −2              ; Set product shift to  >> 2
MOV T,@X2           ; T = X2
MPY P,T,@C2         ; P = T*C2
MOVP T,@X1          ; T = X1, ACC = X2*C2 >> 2
MPY P,T,@C1         ; P = T*C1
MOV @X2,T           ; X2 = X1
MOVA T,@X0          ; T = X0, ACC = X1*C1 >> 2 + X2*C2 >> 2
MPY P,T,@C0         ; P = T*C0
MOV @X1,T           ; X1 = X0
ADDL ACC, P << PM   ; ACC = X0*C0 >> 2 + X1*C1 >> 2 + X2*C2 >> 2
MOVL @Y,ACC         ; Store result into Y
```

MOV TL, #0 *Clear the Lower Half of the XT Register*

Syntax Options MOV TL, #0

Opcode `0101 0110 0101 0110`

Objmode 1

RPT –

CYC 1

Operands

T – Upper 16 bits of the multiplicand register (XT)

#0 – Immediate constant value of zero

Description Load the lower half of the multiplicand register (TL) with zero, leaving the upper half (T) unchanged:

```
TL = 0x0000;
T = unchanged;
```

Flags and Modes None

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Calculate and keep low 32-bit result: Y32 = M32*X16 >> 32
MOV TL,#0        ; TL = 0
MOV T,@X16       ; T = X16
IMPYL P,XT,@M32  ; P = XT * M32 (high 32-bit of result)
MOVL @Y32,P      ; Store result into Y32
```

MOV XARn, PC *Save the Current Program Counter*

Syntax Options MOV XARn, PC

Opcode `0011 1110 0101 1nnn`

Objmode 1

RPT –

CYC 1

Operands

XARn – XAR0 to XAR7, 32-bit auxiliary registers

loc32 – Addressing mode (see Chapter 5)

PC – 22-bit program counter

Description Load XARn with the contents of the PC:

```
XARn = 0:PC;
```

Flags and Modes None

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
TableA:  ; Location of TableA is relative to
.long CONST1              ; the current program
  .long CONST2
  .long CONST3
. FuncA:
  MOV XAR5,PC
  SUBB XAR5,#($-TableA)   ; XAR5 = current PC location
  MOVL ACC,*+XAR5[2]      ; XAR5 = TableA start location
  MOVL @VarA,ACC          ; Load ACC with CONST2
                          ; Store CONST2 in VarA
```

MOVA T,loc16 *Load T Register and Add Previous Product*

Syntax Options MOVA T,loc16

Opcode `0001 0000 LLLL LLLL`

Objmode X

RPT Y

CYC N+1

Operands **T** – Upper 16 bits of the multiplicand register (XT)

loc16 – Addressing mode (see Chapter 5)

Description Load the T register with the 16-bit content of the location pointed to by the “loc16” addressing mode. Also, the content of the P register, shifted by the amount specified by the product shift mode (PM) bits, is added to the content of the ACC register:

```
T = [loc16];
ACC = ACC + P << PM;
```

Flags and Modes

Flags and Modes	Description
N	After the operation, if bit 31 of the ACC register is 1, the N bit is set; otherwise, N is cleared.
Z	After the operation, if the value of ACC is zero, the Z bit is set; otherwise Z is cleared.
C	If the addition generates a carry, then C is set; otherwise, C is cleared.
V	If an overflow occurs, V is set; otherwise V is not affected.
OVC	If overflow mode is disabled; and if the operation generates a positive overflow, the counter is incremented. If overflow mode is disabled; and if the operation generates a negative overflow, the counter is decremented.
OVM	If overflow mode bit is set; the ACC value will saturate maximum positive (0x7FFFFFFF) or maximum negative (0x80000000) if the operation overflows.
PM	The value in the PM bits sets the shift mode for the output operation from the product register. If the product shift value is positive (logical left shift operation), then the low bits are zero filled. If the product shift value is negative (arithmetic right shift operation), the upper bits are sign extended.

Repeat This instruction is repeatable. If the operation follows a RPT instruction, it will be executed N+1 times. The state of the Z, N, C and OVC flags reflect the final result. The V flag will be set if an intermediate overflow occurs.

Example

```
; Calculate using 16-bit multiply:
; Y = (X0*C0) >> 2) + (X1*C1 >> 2) + (X2*C2 >> 2)
; X2 = X1
; X1 = X0
SPM −2            ; Set product shift to >>  2
MOV T,@X2         ; T = X2
MPY P,T,@C2       ; P = T*C2
MOVP T,@X1        ; T = X1, ACC = X2*C2 >> 2
MPY P,T,@C1       ; P = T*C1
MOV @X2,T         ; X2 = X1
MOVA T,@X0        ; T = X0, ACC = X1*C1 >> 2 + X2*C2 >> 2
MPY P,T,@C0       ; P = T*C0
MOV @X1,T         ; X1 = X0
ADDL ACC,P << PM  ; ACC = X0*C0 >> 2 + X1*C1 >> 2 + X2*C2 >> 2
MOVL @Y,ACC       ; Store result into Y
```

MOVAD T, loc16 *Load T Register*

Syntax Options MOVAD T, loc16

Opcode `1010 0111 LLLL LLLL`

Objmode 1

RPT N

CYC 1

Operands

T – Upper 16 bits of the multiplicand register (XT)

loc16 – Addressing mode (see Chapter 5)

Note: For this operation, register-addressing modes cannot be used. The modes are: @ARn, @AH, @AL, @PH, @PL, @SP, @T. An illegal instruction trap will be generated.

Description

Load the T register with the 16-bit content of the location pointed to by the “loc16” addressing mode and then load the next highest 16-bit location pointed to by “loc16” with the content of T. In addition, add the content of the P register, shifted by the amount specified by the product shift mode (PM) bits, to the content of the ACC register:

```
T = [loc16];
[loc16 + 1] = T;
ACC = ACC + P << PM;
```

Flags and Modes

Flags and Modes	Description
N	After the operation, if bit 31 of the ACC register is 1, then the N bit is set; otherwise, N is cleared.
Z	After the operation, if the value of ACC is zero, then the Z bit is set; otherwise Z is cleared.
C	If the addition generates a carry, the C bit is set; otherwise, C is cleared.
V	If an overflow occurs, V is set; otherwise V is not affected.
OVC	If overflow mode is disabled; and if the operation generates a positive overflow, then the counter is incremented. If overflow mode is disabled; and if the operation generates a negative overflow, then the counter is decremented.
OVM	If overflow mode bit is set; then the ACC value will saturate maximum positive (0x7FFFFFFF) or maximum negative (0x80000000) if the operation overflows.
PM	The value in the PM bits sets the shift mode for the output operation from the product register. If the product shift value is positive (logical left shift operation), then the low bits are zero filled. If the product shift value is negative (arithmetic right shift operation), the upper bits are sign extended.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Calculate using 16-bit multiply:
; Y = (X0*C0) >> 2) + (X1*C1 >> 2) + (X2*C2 >> 2)
; X2 = X1
; X1 = X0
SPM −2              ; Set product shift to  >> 2
MOVP T,@X2          ; T = X2
MPYS P,T,@C2        ; P = T*C2, ACC = 0
MOVAD T,@X1         ; T = X1, ACC = X2*C2>>2, X2 = X1
MPY P,T,@C1         ; P = T*C1
MOVAD T,@X0         ; T = X0, ACC = X1*C1>>2 + X2*C2>>2, X1 = X0
MPY P,T,@C0         ; P = T*C0
ADDL ACC, P << PM   ; ACC = X0*C0>>2 + X1*C1>>2 + X2*C2>>2
MOVL @Y,ACC         ; Store result into Y
```

MOVB ACC,#8bit *Load Accumulator With 8-bit Value*

Syntax Options MOVB ACC,#8bit

Opcode `0000 0010 CCCC CCCC`

Objmode 1

RPT –

CYC 1

Operands **ACC** – Accumulator register

#8bit – 8-bit immediate unsigned constant value

Description Load the ACC register with the specified 8-bit, zero-extended immediate constant:

```
ACC = 0:8bit;
```

Flags and Modes

Flags and Modes	Description
N	After the load, the N flag is set if bit 31 of the ACC is 1, else N is cleared.
Z	After the load, the Z flag is set if the ACC value is zero, else Z is cleared.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Increment contents of 32-bit location VarA:
MOVB ACC,#1     ; Load ACC with the value 0x0000 0001
ADDL ACC,@VarA  ; Add to ACC the contents of VarA
MOVL @VarA,ACC  ; Store result back into VarA
```

MOVB AR6/7, #8bit *Load Auxiliary Register With an 8-bit Constant*

Syntax Options

Syntax Options	Opcode	Objmode	RPT	CYC
MOVB AR6, #8bit	`1101 0110 CCCC CCCC`	X	–	1
MOVB AR7, #8bit	`1101 0111 CCCC CCCC`	X	–	1

Operands

XARn – XAR6 OR XAR7, 32-bit auxiliary registers

#8bit – 8-bit immediate constant value

Description

Load AR6 or AR7 with an 8-bit unsigned constant and upper 16 bits of XAR6 and XAR7 are unchanged:

```
AR6/7 = 0:8bit; AR6/7H = unchanged;
```

Flags and Modes

None

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

MOVB AX, #8bit *Load AX With 8-bit Constant*

Syntax Options MOVB AX, #8bit

Opcode

```
1001 101A CCCC CCCC
```

Objmode X

RPT –

CYC 1

Operands

AX – Accumulator high (AH) or accumulator low (AL) register

#8bit – 8-bit immediate constant value

Description

Load accumulator high register (AH) or accumulator low register (AL) with an unsigned 8-bit constant zero extended, leaving the other half of the accumulator register unchanged:

```
AX = 0:8bit;
```

Flags and Modes

Flags and Modes	Description
N	Flag always set to zero.
Z	The load to AX is tested for a zero condition. The bit is set if the operation results in AX = 0, otherwise it is cleared.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
MOVB AL, #0xF0    ; Load AL with the value 0x00F0.
CMP AL,*+XAR0[0]  ; Compare contents pointed to by XAR0
                  ; with AL.
SB Dest,EQ        ; Branch if values are equal.
```

MOVB AX.LSB, loc16 *Load Byte Value*

Syntax Options MOVB AX.LSB, loc16

Opcode `1100 011A LLLL LLLL`

Objmode X

RPT –

CYC 1

Operands **AX.LSB** – Least significant byte of accumulator high (AH.LSB) or accumulator low (AL.LSB) register

loc16 – Addressing mode (see Chapter 5)

Description Load the least significant byte of the specified AX register (AH.LSB or AL.LSB) with 8 bits from the location pointed to by the “loc16” addressing mode. The most significant byte of AX is cleared. The form of the “loc16” operand determines which of its 8 bits are used to load AX.LSB:

```
if(loc16 = *+XARn[offset])
    {
    if(offset is an even number)
      AX.LSB = [loc16.LSB];
    if(offset is an odd value)
      AX.LSB = [loc16.MSB];
    }
else
    AX.LSB = [loc16.LSB];
AX.MSB = 0x00;
```

Note: offset = 3-bit immediate or AR0 or AR1 indexed addressing modes only.

For the following address modes, the returned result is undefined:

*AR6%++	(AMODE = 0)
*0++	(AMODE = x)
*0− −	(AMODE = x)
*BR0++	(AMODE = x)
*BR0−−	(AMODE = x)
*0++, ARPn	(AMODE = 1)
*0− −, ARPn	(AMODE = 1)
*BR0++, ARPn	(AMODE = 1)
*BR0−−, ARPn	(AMODE = 1)

Flags and Modes

Flags and Modes	Description
Z	After the move, AX is tested for a zero condition. The zero flag bit is set if AX = 0; otherwise it is cleared
N	After the move, AX is tested for a negative condition. The bit is set if bit 15 of AX is 1; otherwise it is cleared.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Swap the byte order in the 32-bit "Var32" location.
; Before operation: Var32 = B3 | B2 | B1 | B0
; After operation: Var32 = B0 | B1 | B2 | B3
MOVL XAR2,#Var32  ; Load XAR2 with address of "Var32"

MOVB              ; ACC(B0) = Var32(B3), ACC(B1) = 0
AL.LSB,*+XAR2[3]
MOVB              ; ACC(B2) = Var32(B1), ACC(B3) = 0
AH.LSB,*+XAR2[1]
MOVB              ; ACC(B1) = Var32(B2), ACC(B1) = unch
AL.MSB,*+XAR2[2]
MOVB              ; ACC(B3) = Var32(B0), ACC(B1) = unch
AH.MSB,*+XAR2[0]
MOVL @Var32,ACC   ; Store swapped result in "Var32"
```

MOVB AX.MSB, loc16 *Load Byte Value*

Syntax Options MOVB AX.MSB, loc16

Opcode `0011 100A LLLL LLLL`

Objmode X

RPT –

CYC 1

Operands

AX.MSB – Most significant byte of accumulator high (AH.MSB) or accumulator low (AL.MSB) register

loc16 – Addressing mode (see Chapter 5)

Description

Load the most significant byte of the specified AX register (AH.MSB or AH.LSB) with 8 bits from the location pointed to by the “loc16” addressing mode. The least significant byte of AX is left unchanged. The form of the “loc16” operand determines which of its 8 bits are used to load AX.MSB.

```
if(loc16 = *+XARn[offset])
  {
  if(offset is an even value)
    AX.MSB = [loc16.LSB];
  if(offset is an odd value)
    AX.MSB = [loc16.MSB];
  }
else
  AX.MSB = [loc16.LSB];
AX.LSB = unchanged;
```

Note: Offset = 3-bit immediate or AR0 or AR1 indexed addressing modes only.

For the following address modes, the returned result is undefined:

*AR6%++	(AMODE = 0)
*0++	(AMODE = x)
*0− −	(AMODE = x)
*BR0++	(AMODE = x)
*BR0−−	(AMODE = x)
*0++, ARPn	(AMODE = 1)
*0− −, ARPn	(AMODE = 1)
*BR0++, ARPn	(AMODE = 1)
*BR0−−, ARPn	(AMODE = 1)

Flags and Modes

Flags and Modes	Description
N	After the move AX is tested for a negative condition. The negative flag bit is set if bit 15 of AX is 1; otherwise it is cleared.
Z	After the move, AX is tested for a zero condition. The zero flag bit is set if AX = 0; otherwise it is cleared.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Swap the byte order in the 32-bit "Var32" location.
; Before operation: Var32 = B3 | B2 | B1 | B0
; After operation: Var32 = B0 | B1 | B2 | B3
MOVL XAR2,#Var32  ; Load XAR2 with address of "Var32"

MOVB ; ACC(B0) = Var32(B3), ACC(B1) = 0
AL.LSB,*+XAR2[3]
MOVB ; ACC(B2) = Var32(B1), ACC(B3) = 0
AH.LSB,*+XAR2[1]
MOVB ; ACC(B1) = Var32(B2), ACC(B1) = unch
AL.MSB,*+XAR2[2]
MOVB ; ACC(B3) = Var32(B0), ACC(B1) = unch
AH.MSB,*+XAR2[0]
MOVL @Var32,ACC  ; Store swapped result in "Var32"
```

MOVB loc16,#8bit,COND *Conditionally Save 8-bit Constant*

Syntax Options MOVB loc16,#8bit,COND

Opcode

```
0101 0110 1011 COND
CCCC CCCC LLLL LLLL
```

Objmode 1

RPT –

CYC 1

Operands

loc16 – Addressing mode (see Chapter 5)

#8bit – 8-bit immediate constant value

COND – Conditional codes:

COND	Syntax	Description	Flags Tested
0000	NEQ	Not Equal To	Z = 0
0001	EQ	Equal To	Z = 1
0010	GT	Greater Than	Z = 0 AND N = 0
0011	GEQ	Greater Than or Equal To	N = 0
0100	LT	Less Than	N = 1
0101	LEQ	Less Than or Equal To	Z = 1 OR N = 1
0110	HI	Higher	C = 1 AND Z = 0
0111	HIS, C	Higher or Same, Carry Set	C = 1
1000	LO, NC	Lower, Carry Clear	C = 0
1001	LOS	Lower or Same	C = 0 OR Z = 1
1010	NOV	No Overflow	V = 0
1011	OV	Overflow	V = 1
1100	NTC	Test Bit Not Set	TC = 0
1101	TC	Test Bit Set	TC = 1
1110	NBIO	BIO Input Equal To Zero	BIO = 0
1111	UNC	Unconditional	–

Description If the specified condition being tested is true, then the 8-bit zero extended constant is stored in the location pointed to by the “loc16” addressing mode:

```
if(COND = true) [loc16] = 0:8bit;
```

Note: Addressing modes are not conditionally executed; therefore, if an addressing mode performs a pre or post-modification, it will execute regardless of whether the condition is true or not.

Flags and Modes

Flags and Modes	Description
N	If (COND = true AND loc16 = @AX), then after the move AX is tested for a negative condition. The negative flag bit is set if bit 15 of AX is 1, otherwise it is cleared.
Z	If (COND = true AND loc16 = @AX), then after the move, AX is tested for a zero condition. The zero flag bit is set if AX = 0, otherwise it is cleared.
V	If the V flag is tested by the condition, then V is cleared.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Calculate:
; if(VarA > 20)
;    VarA = 0;
  CMP @VarA,#20     ; Set flags on (VarA - 20)
  MOVB @VarA,#0,GT  ; Zero VarA if greater then
```

MOVB loc16, AX.LSB *Store LSB of AX Register*

Syntax Options MOVB loc16, AX.LSB

Opcode `0011 110A LLLL LLLL`

Objmode X

RPT –

CYC 1

Operands

loc16 – Addressing mode (see Chapter 5)

AX.LSB – Least significant byte of accumulator high (AH.LSB) or accumulator low (AL.LSB) register

Description

Load 8 bits of the location pointed to by the “loc16” addressing mode with the least significant byte of the specified AX register (AH.LSB or AL.LSB). The form of the “loc16” operand determines which of its 8 bits are loaded and which of its 8 bits are left unchanged:

```
if(loc16 = *+XARn[offset])
  {
  if(offset is an even value)
    [loc16.LSB] = AX.LSB;
    [loc16.MSB] = unchanged;
  if(offset is an odd value)
    [loc16.LSB] = unchanged;
    [loc16.MSB] = AX.LSB;
  }
else
    [loc16.LSB] = AX.LSB;
    [loc16.MSB] = unchanged;
```

Note: offset = 3-bit immediate or AR0 or AR1 indexed addressing modes only.

This is a read-modify-write operation.

For the following address modes, the returned result is undefined:

*AR6%++	(AMODE = 0)
*0++	(AMODE = x)
*0– –	(AMODE = x)
*BR0++	(AMODE = x)
*BR0––	(AMODE = x)
*0++, ARPn	(AMODE = 1)
*0– –, ARPn	(AMODE = 1)
*BR0++, ARPn	(AMODE = 1)
*BR0––, ARPn	(AMODE = 1)

Flags and Modes

Flags and Modes	Description
N	If (loc16 = @AX), then after the move AX is tested for a negative condition. The negative flag bit is set if bit 15 of AX is 1, otherwise it is cleared.
Z	If (loc16 = @AX), then after the move, AX is tested for a zero condition. The zero flag bit is set if AX = 0, otherwise it is cleared.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Store the 32-bit contents of the ACC into the
; 32-bit contents of "Var32" location in reverse byte order:
; Before operation: ACC = B3 | B2 | B1 | B0
; After operation: Var32 = B0 | B1 | B2 | B3
MOVL XAR2,#Var32  ; Load XAR2 with address of "Var32"

MOVB              ; Var32(B0) = ACC(B3)
*+XAR2[0],AH.MSB
MOVB              ; Var32(B1) = ACC(B2)
*+XAR2[1],AH.LSB
MOVB              ; Var32(B2) = ACC(B1)
*+XAR2[2],AL.MSB
MOVB              ; Var32(B3) = ACC(B0)
*+XAR2[3],AL.LSB
```

MOVB loc16, AX.MSB *Store MSB of AX Register*

Syntax Options MOVB loc16, AX.MSB

Opcode `1100 100A LLLL LLLL`

Objmode X

RPT –

CYC 1

Operands

loc16 – Addressing mode (see Chapter 5)

AX.MSB – Most significant byte of accumulator high (AH.MSB) or accumulator low (AL.MSB) register

Description

Load 8 bits of the location pointed to by the “loc16” addressing mode with the most significant byte of the specified AX register (AH.MSB or AL.MSB). The form of the “loc16” operand determines which of its 8 bits are loaded and which of its 8 bits are left unchanged:

```
if(loc16 = *+XARn[offset])
  {
  if( offset is an even number)
    [loc16.LSB] = AX.MSB;
    [loc16.MSB] = unchanged;
  if( offset is an odd number)
    [loc16.LSB] = unchanged;
    [loc16.MSB] = AX.MSB;

  }
else
    [loc16.LSB] = AX.MSB;
    [loc16.MSB] = unchanged;
```

Note: offset = 3-bit immediate or AR0 or AR1 indexed addressing modes only.

This is a read-modify-write operation.

For the following address modes, the returned result is undefined:

*AR6%++	(AMODE = 0)
*0++	(AMODE = x)
*0− −	(AMODE = x)
*BR0++	(AMODE = x)
*BR0−−	(AMODE = x)
*0++, ARPn	(AMODE = 1)
*0− −, ARPn	(AMODE = 1)
*BR0++, ARPn	(AMODE = 1)
*BR0−−, ARPn	(AMODE = 1)

Flags and Modes

Flags and Modes	Description
N	If (loc16 = @AX), then after the move AX is tested for a negative condition. The negative flag bit is set if bit 15 of AX is 1, otherwise it is cleared.
Z	If (loc16 = @AX), then after the move, AX is tested for a zero condition. The zero flag bit is set if AX = 0, otherwise it is cleared.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Store the 32-bit contents of the ACC into the
; 32-bit contents of "Var32" location in reverse byte order:
; Before operation: ACC = B3 | B2 | B1 | B0
; After operation: Var32 = B0 | B1 | B2 | B3
MOVL XAR2,#Var32  ; Load XAR2 with address of "Var32"

MOVB              ; Var32(B0) = ACC(B3)
*+XAR2[0],AH.MSB
MOVB              ; Var32(B1) = ACC(B2)
*+XAR2[1],AH.LSB
MOVB              ; Var32(B2) = ACC(B1)
*+XAR2[2],AL.MSB
MOVB              ; Var32(B3) = ACC(B0)
*+XAR2[3],AL.LSB
```

MOVB XARn, #8bit *Load Auxiliary Register With 8-bit Value*

Syntax Options

Syntax Options	Opcode	Objmode	RPT	CYC
MOVB XAR...5, #8bit	`1101 0nnn CCCC CCCC`	X	–	1
MOVB XAR6, #8bit	`1011 1110 CCCC CCCC`	1	–	1
MOVB XAR7, #8bit	`1011 0110 CCCC CCCC`	1	–	1

Operands

XARn – XAR0 to XAR7, 32-bit auxiliary registers

#8bit – 8-bit immediate constant value

Description

Load XARn with the 8-bit unsigned immediate value:

```
XARn = 0:8bit;
```

Flags and Modes

None

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
MOVB XAR0, #F2h  ; Load XAR0 with 0x0000 00F2
```

MOVDL XT,loc32 *Store XT and Load New XT*

Syntax Options MOVDL XT,loc32

Opcode `1010 0110 LLLL LLLL`

Objmode 1

RPT Y

CYC N+1

Operands **XT** – Multiplicand register

loc32 – Addressing mode (see Chapter 5)

Note: For this operation, register-addressing modes cannot be used. The modes are: @XARn, @ACC, @P, @XT. An illegal instruction trap will be generated.

Description Load the XT register with the 32-bit content of the location pointed to by the “loc32” addressing mode and then load the next highest 32-bit location pointed to by “loc32” with the content of XT:

```
XT = [loc32];
[loc32 + 2] = XT;
```

Flags and Modes None

Repeat This instruction is repeatable. If this instruction follows the RPT instruction, then it will be executed N+1 times.

Example

```
; Calculate using 32-bit multiply, retaining high result:
; Y = (X0*C0) >> 2) + (X1*C1 >> 2) + (X2*C2 >> 2)
; X2 = X1
; X1 = X0
SPM –2              ; Set product shift to  >> 2
ZAPA                ; Zero ACC, P, OVC
MOVL XT,@X2         ; XT = X2
QMPYL P,XT,@C2      ; P = XT*C2
MOVDL XT,@X1        ; XT = X1, X2 = X1
QMPYAL P,XT,@C1     ; P = XT*C1, ACC = X2*C2>>2
MOVDL XT,@X0        ; XT = X0, X1 = X0
QMPYAL P,XT,@C0     ; P = XT*C0, ACC = X1*C1>>2 + X2*C2>>2
ADDL ACC,P << PM    ; ACC = X0*C0>>2 + X1*C1>>2 + X2*C2>>2
MOVL @Y,ACC         ; Store result into Y
```

MOVH loc16,ACC << 1..8 *Description*

Syntax Options

Syntax Options	Opcode	Objmode	RPT	CYC
MOVH loc16, ACC << 1	`1011 0011 LLLL LLLL`	1	Y	N+1

MOVH loc16, ACC << 2..8	`0101 0110 0010 1111`
	`0000 0SHF LLLL LLLL`
		1	Y	N+1

	`1011 0SHF LLLL LLLL`	0	–	1

Operands

loc16 – Addressing mode (see Chapter 5)

ACC – Accumulator register

#1..8 – Shift value

Description

Load the content of the location pointed to by the “loc16” addressing mode with the high word of the ACC register after left−shifting by the specified value. The ACC register is not modified:

```
[loc16] = ACC >> (16 – shift value);
```

Flags and Modes

Flags and Modes	Description
N	If (loc16 = @AX), then after the load AX is checked for a negative condition. The N flag is set if bit 15 of the AX is 1; else N is cleared.
Z	If (loc16 = @AX) then after the load AX is checked for a zero condition. The Z flag is set if AX is zero; else Z is cleared.

Repeat

If the operation is repeatable, then the instruction will be executed N+1 times. The state of the Z and N flags will reflect the final result. If the operation is not repeatable, the instruction will execute only once.

Example

```
; Multiply two Q15 numbers (VarA and VarB) and store result in
; VarC as a Q15 number:
MOV T,@VarA          ; T = VarA (Q15)
MPY ACC,T,@VarB      ; ACC = VarA * VarB (Q30)
MOVH @VarC,ACC << 1  ; VarC = ACC >> 16–1) (Q15)
                     ; VarC as a Q31 number:
MOV T,@VarA          ; T = VarA (T = Q14)
MPY ACC,T,@VarB      ; ACC = VarA * VarB (ACC = Q28)
MOV @VarC+0,ACC <<   ; VarC low = ACC >> 3
MOVH @VarC+1,ACC <<  ; VarC high = ACC >> (16–1) (VarC = Q31)
```

MOVH loc16, P *Save High Word of the P Register*

Syntax Options MOVH loc16, P

Opcode `0101 0111 LLLL LLLL`

Objmode X

RPT Y

CYC N+1

Operands **loc16** – Addressing mode (see Chapter 5)

P – Product register

Description The contents of the P register are shifted by the amount specified in the product shift mode (PM), and the upper half of the shifted value is stored into the 16-bit location pointed to by the “loc16” addressing mode. The P register is not modified by the operation:

```
[loc16] = (P << PM) >> 16;
```

Flags and Modes

Flags and Modes	Description
N	If (loc16 = @AX) and bit 15 of the AX register is 1, then the N bit is set; otherwise, N is cleared.
Z	If (loc16 = @AX) and the value of AX after the load is zero, then the Z bit is set; otherwise Z is cleared.
PM	The value in the PM bits sets the shift mode for the output operation from the product register. If the product shift value is positive (logical left shift operation), then the low bits are zero filled. If the product shift value is negative (arithmetic right shift operation), the upper bits are sign extended.

Repeat This instruction is repeatable. If the operation follows a RPT instruction, then it will be executed N+1 times. The state of the Z, and N flags will reflect the final result.

Example

```
; Calculate Y32 = M16*X16 >> 6
MOV T,@M16    ; T = M
MPY P,T,@X16  ; P = T * X
SPM −6        ; Set product shift to  >> 6
MOV @Y32+0,P  ; Y32 = P >> 6
MOVH @Y32+1,P
```

MOVL ACC,loc32 *Load Accumulator With 32 Bits*

Syntax Options MOVL ACC,loc32

Opcode `0000 0110 LLLL LLLL`

Objmode X

RPT –

CYC 1

Operands **ACC** – Accumulator register

loc32 – Addressing mode (see Chapter 5)

Description Load the ACC register with the content of the location pointed to by the “loc32” addressing mode.

```
ACC = [loc32];
```

Flags and Modes

Flags and Modes	Description
N	After the load, the N flag is set if bit 31 of the ACC is 1, else N is cleared.
Z	After the load, the Z flag is set if the ACC is zero, else Z is cleared.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
Calculate the 32-bit value: VarC = VarA + VarB;
MOVL ACC,@VarA  ; Load ACC with contents of VarA
ADDL ACC,@VarB  ; Add to ACC the contents of VarB
MOVL @VarC,ACC  ; Store result into VarC
```

MOVL ACC,P << PM *Load the Accumulator With Shifted P*

Syntax Options MOVL ACC,P << PM

Opcode `0001 0110 1010 1100`

Objmode X

RPT –

CYC 1

Note: This instruction is an alias for the ”MOVP T,loc16” operation with “loc16 = @T” addressing mode.

Operands

ACC – Accumulator register

P – Product register

<< PM – Product shift mode

Description Load the ACC register with the content of the P register shifted as specified by the product shift mode (PM):

```
ACC = P << PM;
```

Flags and Modes

Flags and Modes	Description
N	After the load, the N flag is set if bit 31 of the ACC is 1, else N is cleared.
Z	After the load, the Z flag is set if the ACC is zero, else Z is cleared.
PM	The value in the PM bits sets the shift mode for the output operation from the product register. If the product shift value is positive (logical left shift operation), then the low bits are zero filled. If the product shift value is negative (arithmetic right shift operation), the upper bits are sign extended.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Calculate: Y = Y + (M*X >> 4)
; Y is a 32-bit value, M and X are 16-bit values
SPM −4            ; Set product shift to  >> 4
MOV T,@M          ; T = M
MPY P,T,@X        ; P = M * X
MOVL ACC,P << PM  ; ACC = M*X >> 4
ADDL @Y,ACC       ; Y = Y + ACC
```

MOVL loc32, ACC *Store 32-bit Accumulator*

Syntax Options MOVL loc32, ACC

Opcode `0001 1110 LLLL LLLL`

Objmode X

RPT –

CYC 1

Operands

ACC – Accumulator register

loc32 – Addressing mode (see Chapter 5)

Description

Store the contents of the ACC register into the location pointed to by the “loc32” addressing mode:

```
[loc32] = ACC;
```

Flags and Modes

Flags and Modes	Description
N	If (loc32 = @ACC) then after the load, the N flag is set if bit 31 of the ACC is 1, else N is cleared.
Z	If (loc32 = @ACC) then after the load, the Z flag is set if ACC is zero, else Z is cleared.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
Calculate the 32-bit value: VarC = VarA + VarB;
MOVL ACC,@VarA  ; Load ACC with contents of VarA
ADDL ACC,@VarB  ; Add to ACC the contents of VarB
MOVL @VarC,ACC  ; Store result into VarC
```

MOVL loc32,ACC,COND *Conditionally Store the Accumulator*

Syntax Options MOVL loc32,ACC,COND

Opcode

```
0101 0110 0100 1000
0000 COND LLLL LLLL
```

Objmode X

RPT –

CYC 1

Operands

loc32 – Addressing mode (see Chapter 5)

ACC – Accumulator register

COND – Conditional codes:

COND	Syntax	Description	Flags Tested
0000	NEQ	Not Equal To	Z = 0
0001	EQ	Equal To	Z = 1
0010	GT	Greater Than	Z = 0 AND N = 0
0011	GEQ	Greater Than or Equal To	N = 0
0100	LT	Less Than	N = 1
0101	LEQ	Less Than or Equal To	Z = 1 OR N = 1
0110	HI	Higher	C = 1 AND Z = 0
0111	HIS, C	Higher or Same, Carry Set	C = 1
1000	LO, NC	Lower, Carry Clear	C = 0
1001	LOS	Lower or Same	C = 0 OR Z = 1
1010	NOV	No Overflow	V = 0
1011	OV	Overflow	V = 1
1100	NTC	Test Bit Not Set	TC = 0
1101	TC	Test Bit Set	TC = 1
1110	NBIO	BIO Input Equal To Zero	BIO = 0
1111	UNC	Unconditional	–

Description

If the specified condition being tested is true, then the location pointed to by the “loc32” addressing mode will be loaded with the contents of the ACC register:

```
if(COND = true) [loc32] = ACC;
```

Note: Addressing modes are not conditionally executed. Hence, if an addressing mode performs a pre or post modification, the modification will occur regardless of whether the condition is true or not.

Flags and Modes

Flags and Modes	Description
N	If (COND = true AND loc32 = @ACC), then after the move if bit 31 of ACC is 1, N is set; otherwise N cleared.
Z	If (COND = true AND loc32 = @ACC), then after the move if (ACC = 0), then the Z bit is set; otherwise it is cleared.
V	If the V flag is tested by the condition, then V is cleared.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Swap the contents of 32-bit VarA and VarB if VarB is higher:
MOVL ACC,@VarB     ; ACC = VarB
MOVL P,@VarA       ; P = VarA
CMPL ACC,@P        ; Set flags on (VarB - VarA)
MOVL @VarA,ACC,HI  ; VarA = ACC if higher
MOVL @P,ACC,HI     ; P = ACC if higher
MOVL @VarA,P       ; VarA = P
```

MOVL loc32,P *Store the P Register*

Syntax Options MOVL loc32,P

Opcode `1010 1001 LLLL LLLL`

Objmode 1

RPT –

CYC 1

Operands **loc32** – Addressing mode (see Chapter 5)

P – Product register

Description Store the P register contents into the location pointed to by the “loc32” addressing mode:

```
[loc32] = P;
```

Flags and Modes

Flags and Modes	Description
N	If (loc32 = @ACC) and bit 31 of the ACC register is 1, then the N bit is set; otherwise, N is cleared.
Z	If (loc32 = @ACC) and the value of ACC after the load is zero, then the Z bit is set; otherwise Z is cleared.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Add 64-bit VarA, VarB and VarC, and store result in VarD:
MOVL P,@VarA+0     ; Load P with low 32 bits of VarA
MOVL ACC,@VarA+2   ; Load ACC with high 32 bits of VarA
ADDUL P,@VarB+0    ; Add to P unsigned low 32 bits of VarB
ADDCL ACC,@VarB+2  ; Add to ACC with carry high 32 bits of VarB
ADDUL P,@VarC+0    ; Add to P unsigned low 32 bits of VarC
ADDCL ACC,@VarC+2  ; Add to ACC with carry high 32 bits of VarC
MOVL @VarD+0,P     ; Store low 32-bit result into VarD
MOVL @VarD+2,ACC   ; Store high 32-bit result into VarD
```

MOVL loc32, XARn *Store 32-bit Auxiliary Register*

Syntax Options

Syntax Options	Opcode	Objmode	RPT	CYC
MOVL loc32, XAR0	`0011 1010 LLLL LLLL`	1	–	1
MOVL loc32, XAR1	`1011 0010 LLLL LLLL`	1	–	1
MOVL loc32, XAR2	`1010 1010 LLLL LLLL`	1	–	1
MOVL loc32, XAR3	`1010 0010 LLLL LLLL`	1	–	1
MOVL loc32, XAR4	`1010 1000 LLLL LLLL`	1	–	1
MOVL loc32, XAR5	`1010 0000 LLLL LLLL`	1	–	1
MOVL loc32, XAR6	`1100 0010 LLLL LLLL`	X	–	1
MOVL loc32, XAR7	`1100 0011 LLLL LLLL`	X	–	1

Operands

loc32 – Addressing mode (see Chapter 5)

XARn – XAR0 to XAR7, 32-bit auxiliary registers

Description

Load the contents of the 32-bit addressed location with the contents of XARn:

```
[loc32] = XARn;
```

Flags and Modes

Flags and Modes	Description
N	If (loc32 = @ACC), then the load to ACC is tested for a negative condition. Bit-31 of the ACC register is the sign bit, 0 for positive, 1 for negative. The negative flag bit is set if the operation on the ACC register generates a negative value, otherwise it is cleared.
Z	If (loc32 = @ACC), then the load to ACC is tested for a zero condition. The bit is set if the result of the operation on the ACC register generates a 0 value, otherwise it is cleared.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
MOVL @ACC, XAR0    ; Move the 32-bit contents of XAR0 into ACC.
                   ; If bit 31 of the ACC is 1 set N. If
                   ; ACC = 0, set Z.
MOVL *XAR1, XAR7   ; Move the 32-bit contents of XAR7 into the
                   ; location pointed to by XAR1.
MOVL *XAR6++,XAR6  ; Move the 32-bit contents of XAR6 into the
                   ; location pointed to by XAR6. Post-increment
                   ; the contents of XAR6.
MOVL *−−XAR5,XAR5  ; Predecrement the contents of XAR5. Move the
                   ; 32-bit contents of XAR5 into the location
                   ; pointed to by XAR5.
```

MOVL loc32,XT *Store the XT Register*

Syntax Options MOVL loc32,XT

Opcode `1010 1011 LLLL LLLL`

Objmode 1

RPT –

CYC 1

Operands **loc32** – Addressing mode (see Chapter 5)

XT – Multiplicand register

Description Store the XT register into 32-bit location pointed to by the “loc32” addressing mode:

```
[loc32] = XT;
```

Flags and Modes

Flags and Modes	Description
N	If (loc32 = @ACC) and bit 31 of the ACC register is 1, then the N bit is set; otherwise, N is cleared.
Z	If (loc32 = @ACC) and the value of ACC after the load is zero, then the Z bit is set; otherwise Z is cleared.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Calculate using 32-bit multiply, retaining high result:
; Y = (X0*C0) >> 2) + (X1*C1 >> 2) + (X2*C2  >>2)
; X2 = X1
; X1 = X0
SPM −2              ; Set product shift to >> 22
ZAPA                ; Zero ACC, P, OVC
MOVL XT,@X2         ; XT = X2
QMPYL P,XT,@C2      ; P = XT*C2
MOVL XT,@X1         ; XT = X1, ACC = X2*C2 >> 2
QMPYAL P,XT,@C1     ; P = XT*C1
MOVL @X2,XT         ; X2 = X1
MOVL XT,@X0         ; XT = X0, ACC = X1*C1 >> 2 + X2*C2 >> 2
QMPYAL P,XT,@C0     ; P = XT*C0
MOVL @X1,XT         ; X1 = X0
ADDL ACC, P << PM   ; ACC = X0*C0 >> 2 + X1*C1 >> 2 + X2*C2 >> 2
MOVL @Y,ACC         ; Store result into Y
```

MOVL P,ACC *Load P From the Accumulator*

Syntax Options

MOVL P,ACC

Opcode

`1111 1111 0101 1010`

Objmode

X

RPT

–

CYC

1

Operands

P – Product register

ACC – Accumulator register

Description

Load the P register with the content of the ACC register:

```
P = ACC;
```

Flags and Modes

None

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
Calculate the 32-bit value: VarC = abs(VarA) + abs(VarB)
MOVL ACC,@VarA  ; Load ACC with contents of VarA
ABS ACC         ; Take absolute value of VarA
MOVL P,ACC      ; Temp save ACC in P register
MOVL ACC,@VarB  ; Load ACC with contents of VarB
ABS ACC         ; Take absolute value of VarB
ADDL ACC,@P     ; Add contents of P to ACC
MOVL @VarC,ACC  ; Store result into VarC
```

MOVL P,loc32 *Load the P Register*

Syntax Options MOVL P,loc32

Opcode 1010 0011 LLLL LLLL

Objmode 1

RPT –

CYC 1

Operands **P** – Product register

loc32 – Addressing mode (see Chapter 5)

Description Load the P register with the 32-bit location pointed to by the “loc32” addressing mode:

```
P = [loc32];
```

Flags and Modes None

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Add 64-bit VarA, VarB and VarC, and store result in VarD:
MOVL P,@VarA+0     ; Load P with low 32 bits of VarA M
OVL ACC,@VarA+2    ; Load ACC with high 32 bits of VarA
ADDUL P,@VarB+0    ; Add to P unsigned low 32 bits of VarB
ADDCL ACC,@VarB+2  ; Add to ACC with carry high 32 bits of VarB
ADDUL P,@VarC+0    ; Add to P unsigned low 32 bits of VarC
ADDCL ACC,@VarC+2  ; Add to ACC with carry high 32 bits of VarC
MOVL @VarD+0,P     ; Store low 32-bit result into VarD
MOVL @VarD+2,ACC   ; Store high 32-bit result into VarD
```

MOVL XARn, loc32 *Load 32-bit Auxiliary Register*

Syntax Options

Syntax Options	Opcode	Objmode	RPT	CYC
MOVL XAR0, loc32	`1000 1110 LLLL LLLL`	1	–	1
MOVL XAR1, loc32	`1000 1011 LLLL LLLL`	1	–	1
MOVL XAR2, loc32	`1000 0110 LLLL LLLL`	1	–	1
MOVL XAR3, loc32	`1000 0010 LLLL LLLL`	1	–	1
MOVL XAR4, loc32	`1000 1010 LLLL LLLL`	1	–	1
MOVL XAR5, loc32	`1000 0011 LLLL LLLL`	1	–	1
MOVL XAR6, loc32	`1100 0100 LLLL LLLL`	X	–	1
MOVL XAR7, loc32	`1100 0101 LLLL LLLL`	X	–	1

Operands

XARn – XAR0 to XAR7, 32-bit auxiliary registers

loc32 – Addressing mode

Description

Load XARn with the contents of the 32-bit addressed location:

```
XARn = [loc32];
```

Flags and Modes

None

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
MOVL XAR0,@ACC     ; Move the 32-bit contents of ACC into
                   ; XAR0
MOVL XAR2,*XAR0++  ; Move the 32-bit value pointed to by
                   ; XAR0 into XAR2. Post increment XAR0
                   ; by 2
MOVL XAR3,*XAR3++  ; Move the 32-bit value pointed to by
                   ; XAR3 into XAR3. Address modification
                   ; of XAR3 is ignored.
MOVL XAR4,*−−XAR4  ; Predecrement the contents of XAR4.
                   ; Move the 32-bit value pointed to by
                   ; XAR4 into XAR4.
```

MOVL XARn, #22bit *Load 32-bit Auxiliary Register With Constant Value*

Syntax Options

Syntax Options	Opcode	OBJ- MODE	RPT	CYC

MOVL XAR0, #22bit	`1000 1101 00CC CCCC`
	`CCCC CCCC CCCC CCCC`
		1	–	1

MOVL XAR1, #22bit	`1000 1101 01CC CCCC`
	`CCCC CCCC CCCC CCCC`
		1	–	1

MOVL XAR2, #22bit	`1000 1101 10CC CCCC`
	`CCCC CCCC CCCC CCCC`
		1	–	1

MOVL XAR3, #22bit	`1000 1101 11CC CCCC`
	`CCCC CCCC CCCC CCCC`
		1	–	1

MOVL XAR4, #22bit	`1000 1111 00CC CCCC`
	`CCCC CCCC CCCC CCCC`
		1	–	1

MOVL XAR5, #22bit	`1000 1111 01CC CCCC`
	`CCCC CCCC CCCC CCCC`
		1	–	1

MOVL XAR6, #22bit	`0111 0110 10CC CCCC`
	`CCCC CCCC CCCC CCCC`
		X	–	1

MOVL XAR7, #22bit	`0111 0110 11CC CCCC`
	`CCCC CCCC CCCC CCCC`
		X	–	1

Operands

XARn – XAR0 to XAR7, 32-bit auxiliary registers

#22bit – 22-bit immediate constant value

Description

Load XARn with a 22-bit unsigned constant:

```
XARn = 0:22bit;
```

Flags and Modes

None

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
MOVL XAR4,#VarA  ; Initialize XAR4 pointer with the
                 ; 22-bit address of VarA
```

MOVL XT,loc32 *Load the XT Register*

Syntax Options MOVL XT,loc32

Opcode `1000 0111 LLLL LLLL`

Objmode 1

RPT –

CYC 1

Operands **T** – Upper 16 bits of the multiplicand register (XT)

loc32 – Addressing mode (see Chapter 5)

Description Load the XT register with the 32-bit content of the location pointed to by the “loc32” addressing mode:

```
XT = [loc32];
```

Flags and Modes None

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Calculate using 32-bit multiply, retaining high result:
; Y = (X0*C0) >> 2) + (X1*C1 >> 2) + (X2*C2 >> 2)
; X2 = X1
; X1 = X0
SPM −2            ; Set product shift to >>  2
ZAPA              ; Zero ACC, P, OVC
MOVL XT,@X2       ; XT = X2
QMPYL P,XT,@C2    ; P = XT*C2
MOVL XT,@X1       ; XT = X1, ACC = X2*C2 >> 2
QMPYAL P,XT,@C1   ; P = XT*C1
MOVL @X2,XT       ; X2 = X1
MOVL XT,@X0       ; XT = X0, ACC = X1*C1 >> 2 + X2*C2 >> 2
QMPYAL P,XT,@C0   ; P = XT*C0
MOVL @X1,XT       ; X1 = X0
ADDL ACC,P << PM  ; ACC = X0*C0 >> 2 + X1*C1 >> 2 + X2*C2 >> 2
MOVL @Y,ACC       ; Store result into Y
```

MOVP T,loc16 *Load the T Register and Store P in the Accumulator*

Syntax Options MOVP T,loc16

Opcode `0001 0110 LLLL LLLL`

Objmode X

RPT –

CYC 1

Operands **T** – Upper 16 bits of the multiplicand register (XT)

loc16 – Addressing mode (see Chapter 5)

Description Load the T register with the 16-bit content of the location pointed to by the “loc16” addressing mode. Also, the content of the P register, shifted by the amount specified by the product shift mode (PM) bits, is loaded into the ACC register:

```
T = [loc16];
ACC = P << PM;
```

Flags and Modes

Flags and Modes	Description
N	After the operation if bit 31 of the ACC register is 1, then the N bit is set; otherwise, N is cleared.
Z	After the operation, if the value of ACC is zero, then the Z bit is set; otherwise Z is cleared.
PM	The value in the PM bits sets the shift mode for the output operation from the product register. If the product shift value is positive (logical left shift operation), then the low bits are zero filled. If the product shift value is negative (arithmetic right shift operation), the upper bits are sign extended.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Calculate using 16-bit multiply:
; Y = (X0*C0) >> 2) + (X1*C1 >> 2) + (X2*C2 >> 2)
; X2 = X1
; X1 = X0
SPM −2              ; Set product shift to >> 2
MOV T,@X2           ; T = X2
MPY P,T,@C2         ; P = T*C2
MOVP T,@X1          ; T = X1, ACC = X2*C2 >> 2
MPY P,T,@C1         ; P = T*C1
MOV @X2,T           ; X2 = X1
MOVA T,@X0          ; T = X0, ACC = X1*C1 >> 2 + X2*C2 >> 2
MPY P,T,@C0         ; P = T*C0
MOV @X1,T           ; X1 = X0
ADDL ACC, P << PM   ; ACC = X0*C0 >> 2 + X1*C1 >> 2 + X2*C2 >> 2
MOVL @Y,ACC         ; Store result into Y
```

MOVS T,loc16 — *Load T and Subtract P From the Accumulator*

Syntax Options MOVS T,loc16

Opcode `0001 0001 LLLL LLLL`

Objmode X

RPT Y

CYC N+1

Operands

T – Upper 16 bits of the multiplicand register (XT)

loc16 – Addressing mode (see Chapter 5)

Description

Load the T register with the 16-bit content of the location pointed to by the “loc16” addressing mode. Also, the content of the P register, shifted by the amount specified by the product shift mode (PM) bits, is subtracted from the content of the ACC register:

```
T = [loc16];
ACC = ACC − P << PM;
```

Flags and Modes

Flags and Modes	Description
N	After the operation, if bit 31 of the ACC register is 1, then the N bit is set; otherwise, N is cleared.
Z	After the operation, if the value of ACC is zero, then the Z bit is set; otherwise Z is cleared.
C	If the subtraction generates a borrow, the C bit is cleared; otherwise, C is set.
V	If an overflow occurs, V is set; otherwise V is not affected.
OVC	If overflow mode is disabled; and if the operation generates a positive overflow, then the counter is incremented. If overflow mode is disabled; and if the operation generates a negative overflow, then the counter is decremented.
OVM	If overflow mode bit is set; then the ACC value will saturate maximum positive (0x7FFFFFFF) or maximum negative (0x80000000) if the operation overflows.
PM	The value in the PM bits sets the shift mode for the output operation from the product register. If the product shift value is positive (logical left shift operation), then the low bits are zero filled. If the product shift value is negative (arithmetic right shift operation), the upper bits are sign extended.

Repeat

This instruction is repeatable. If the operation follows a RPT instruction, then it will be executed N+1 times. The state of the Z, N, C and OVC flags will reflect the final result. The V flag will be set if an intermediate overflow occurs.

Example

```
; Calculate using 16-bit multiply:
; Y = (X0*C0) >> 2) + (X1*C1 >> 2) + (X2*C2 >> 2)
; X2 = X1
; X1 = X0
SPM −2            ; Set product shift to >> 2
MOVP T,@X2        ; T = X2
MPYS P,T,@C2      ; P = T*C2, ACC = 0
MOVS T,@X1        ; T = X1, ACC = −X2*C2 >> 2
MPY P,T,@C1       ; P = T*C1
MOV @X2,T         ; X2 = X1
MOVA T,@X0        ; T = X0, ACC = −X1*C1 >> 2 − X2*C2 >> 2
MPY P,T,@C0       ; P = T*C0
MOV @X1,T         ; X1 = X0
SUBL ACC,P << PM  ; ACC = −X0*C0 >> 2 − X1*C1 >> 2 − X2*C2 >> 2
MOVL @Y,ACC       ; Store result into Y
```

MOVU ACC,loc16 *Load Accumulator With Unsigned Word*

Syntax Options MOVU ACC,loc16

Opcode `0000 1110 LLLL LLLL`

Objmode X

RPT –

CYC 1

Operands **ACC** – Accumulator register

loc16 – Addressing mode (see Chapter 5)

Description Load the low half of the accumulator (AL) with the 16-bit contents of the addressed location pointed to by the “loc16” addressing mode and fill the high half of the accumulator (AH) with 0s:

```
AL = [loc16];
AH = 0x0000;
```

Flags and Modes

Flags and Modes	Description
N	Clear flag.
Z	After the load, the Z flag is set if the ACC value is zero, else Z is cleared.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Add three 32-bit unsigned variables by 16-bit parts:
MOVU ACC,@VarAlow   ; AH = 0, AL = VarAlow
ADD ACC,@VarAhigh   ; AH = VarAhigh, AL = VarAlow
<< 16
ADDU ACC,@VarBlow   ; ACC = ACC + 0:VarBlow
ADD ACC,@VarBhigh   ; ACC = ACC + VarBhigh << 16
<< 16
ADDCU ACC,@VarClow  ; ACC = ACC + VarClow + Carry
ADD ACC,@VarChigh   ; ACC = ACC + VarChigh << 16
<< 16
```

MOVU loc16,OVC *Store the Unsigned Overflow Counter*

Syntax Options

MOVU loc16,OVC

Opcode

```
0101 0110 0010 1000
0000 0000 LLLL LLLL
```

Objmode

1

RPT

–

CYC

1

Operands

loc16 – Addressing mode (see Chapter 5)

OVC – Overflow counter

Description

Store the 6 bits of the overflow counter (OVC) into the lower 6 bits of the location pointed to by the “loc16” addressing mode and zero the upper 10 bits of the addressed location:

```
[loc16(15:6)]    = 0;
[loc16(5:0)]    = OVC;
```

Flags and Modes

Flags and Modes	Description
N	If (loc16 = @AX) and bit 15 of AX is 1, then set N; otherwise clear N.
Z	If (loc16 = @AX) and AX is zero, then set Z; otherwise clear Z.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Save and restore contents of ACC and OVC bits:
MOVU *SP++,OVC  ; Save OVC on stack
MOV *SP++,AL    ; Save AL on stack
MOV *SP++,AH    ; Save AH on stack
.
.
.
.
MOV AH,*--SP     ; Restore AH from stack
MOV AL,*--SP     ; Restore AL from stack
MOVU OVC,*--SP   ; Restore OVC from stack
```

MOVU OVC,loc16 *Load Overflow Counter With Unsigned Value*

Syntax Options MOVU OVC,loc16

Opcode

```
0101 0110 0110 0010
0000 0000 LLLL LLLL
```

Objmode 1

RPT –

CYC 1

Operands **OVC** – 6-bit overflow counter

Description Load the overflow counter (OVC) with the lower 6 bits of the location pointed to by the “loc16” addressing mode:

```
OVC = [loc16(5:0)]
```

Flags and Modes

Flags and Modes	Description
OVC	The 6-bit overflow counter is modified.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Save and restore contents of ACC and OVC bits:
MOVU *SP++,OVC  ; Save OVC on stack
MOV *SP++,AL    ; Save AL on stack
MOV *SP++,AH    ; Save AH on stack
.
.
.
.
MOV AH,*−−SP    ; Restore AH from stack
MOV AL,*−−SP    ; Restore AL from stack M
OVU OVC,*−−SP   ; Restore OVC from stack
```

MOVW DP, #16bit *Load the Entire Data Page*

Syntax Options

MOVW DP, #16bit

Opcode

```
0111 0110 0001 1111
CCCC CCCC CCCC CCCC
```

Objmode

X

RPT

–

CYC

1

Operands

DP – Data page register

#16bit – 16-bit immediate constant value

Description

Load the data page register with a 16-bit constant:

```
DP(15:0) = 16bit;
```

Flags and Modes

None

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
MOVW DP, #VarA    ; Load DP with the data page that
                  ; contains VarA.    Assumes VarA is in the
                  ; lower 0x003F FFC0 of memory
MOVW DP, #0F012h  ; Load DP with data page number 0xF012
```

MOVX TL,loc16 *Load Lower Half of XT With Sign Extension*

Syntax Options MOVX TL,loc16

Opcode

```
0101 0110 0010 0001
xxxx xxxx LLLL LLLL
```

Objmode 1

RPT –

CYC 1

Operands

TL – Lower 16 bits of the multiplicand register (XT)

loc16 – Addressing mode (see Chapter 5)

Description

Load the lower 16 bits of the multiplicand register (TL) with the 16-bit contents of the location pointed to by the “loc16” addressing mode and then sign extend that value into the upper upper 16 bits of XT:

```
TL = [loc16];
T = sign extension of TL;
```

Flags and Modes None

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Calculate and keep low 32-bit result: Y32 = M32*X16
MOVX TL,@X16     ; XT = S:X16
IMPYL P,XT,@M32  ; P = XT * M32 (low 32 bits of result)
MOVL @Y32,P      ; Store result into Y32
```

MOVZ ARn, loc16 *Load Lower Half of XARn and Clear Upper Half*

Syntax Options

Syntax Options	Opcode	OBJ- MODE	RPT	CYC
MOVZ AR0...5, loc16	`0101 1nnn LLLL LLLL`	X	–	1
MOVZ AR6, loc16	`1000 1000 LLLL LLLL`	1	–	1
MOVZ AR7, loc16	`1000 0000 LLLL LLLL`	1	–	1

Operands

ARn – AR0 to AR7, lower 16 bits of auxiliary registers

loc16 – Addressing modes (See chapter 5)

Description

Load ARn with the contents of the 16-bit location and clear ARnH:

```
ARn = [loc16];
ARnH = 0;
```

Flags and Modes

None

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
MOVL XAR7, #ArrayA   ; Initialize XAR2 pointer
MOVZ AR0, *+XAR2[0]  ; Load 16-bit value pointed to by XAR2
                     ; into AR0. XAR0(31:16) = 0.
MOVZ AR7, *−SP[1]    ; Load the first 16-bit value off of the
                     ; stack into AR7. XAR7(31:16) = 0.
```

MOVZ DP, #10bit *Load Data Page and Clear High Bits*

Syntax Options MOVZ DP, #10bit

Opcode

```
1011 10CC CCCC CCCC
```

Objmode 1

RPT –

CYC 1

Operands

DP – Data page register

#10bit – 10-bit immediate constant value

Description

Load the data page register with a 10-bit constant and clear the upper 6 bits:

```
DP(9:0) = 10bit;
DP(15:10) = 0;
```

Flags and Modes None

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
MOVZ DP, #VarA  ; Load DP with the data page that contains
                ; VarA. Assumes VarA is in the lower
                ; 0x0000 FFC0 of memory
MOVZ DP, #3FFh  ; Load DP with page number 0x03FF.
```

MPY ACC,loc16, #16bit *16 X 16-bit Multiply*

Syntax Options MPY ACC,loc16, #16bit

Opcode

```
0011 0100 LLLL LLLL
CCCC CCCC CCCC CCCC
```

Objmode X

RPT –

CYC 1

Operands

ACC – Accumulator register

loc16 – Addressing mode (see Chapter 5)

16-bit immediate constant value

Description

Load the T register with the 16-bit content of the location pointed to by the “loc16” addressing mode; then, multiply the signed 16-bit content of the T register by the specified signed 16-bit constant value:

```
T = [loc16];
ACC = signed T * signed 16bit;
```

Flags and Modes

Flags and Modes	Description
Z	After the operation, the Z flag is set if the ACC is zero, else Z is cleared.
N	After the operation, the N flag is set if bit 31 of the ACC is 1, else N is cleared.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Calculate signed using 16-bit multiply:
; Y32 = Y32 + X16 * 2000
MPY ACC,@X16,#2000  ; T = X16, ACC = X16 * 2000
ADDL @Y32,ACC       ; Y32 = Y32 + ACC
```

MPY ACC, T, loc16 *16 X 16-bit Multiply*

Syntax Options MPY ACC, T, loc16

Opcode

```
0001 0010 LLLL LLLL
```

Objmode X

RPT –

CYC 1

Operands

ACC – Accumulator register

T – Multiplicand register

loc16 – Addressing mode (see Chapter 5)

Description

Multiply the signed 16-bit content of the T register by the signed 16-bit contents of the location pointed to by the “loc16” addressing mode and store the result in the ACC register:

```
ACC = signed T * signed [loc16];
```

Flags and Modes

Flags and Modes	Description
Z	After the operation, the Z flag is set if the ACC is zero, else Z is cleared.
N	After the operation, the N flag is set if bit 31 of the ACC is 1, else N is cleared.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Calculate signed using 16-bit multiply:
; Y32 = Y32 + X16*M16
MOV T,@X16      ; T = X16
MPY ACC,T,@M16  ; ACC = T * M16
ADDL @Y32,ACC   ; Y32 = Y32 + ACC
```

MPY P,loc16,#16bit *16 X 16-Bit Multiply*

Syntax Options

MPY P,loc16,#16bit

Opcode

```
1000 1100 LLLL LLLL
CCCC CCCC CCCC CCCC
```

Objmode

1

RPT

–

CYC

1

Operands

P – Product register

loc16 – Addressing mode (see Chapter 5)

#16bit – 16-bit immediate constant value

Description

Multiply the signed 16-bit contents of the location pointed to by the “loc16” addressing mode by the 16-bit immediate value and store the 32-bit result in the P register:

```
P = signed [loc16] * signed 16bit;
```

Flags and Modes

None

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; ; Calculate using 16-bit multiply:
; Y = (X0*C0) >> 2) + (X1*C1 >> 2) + (X2*C2 >> 2),
; C0, C1 and C2 are constants
SPM −2              ; Set product shift to >> 2
MOVB ACC,#0         ; Zero ACC
MPY P,@X2,#C2       ; P = X2*C2
MPYA P,@X1,#C1      ; ACC = X2*C2>>2, P = X1*C1
MPYA P,@X0,#C0      ; ACC = X1*C1>>2 + X2*C2>>2, P = X0*C0
ADDL ACC,P << PM    ; ACC = X0*C0>>2 + X1*C1>>2 + X2*C2>>2
MOVL @Y,ACC         ; Store result into Y
```

MPY P,T,loc16 *16 X 16 Multiply*

Syntax Options MPY P,T,loc16

Opcode `0011 0011 LLLL LLLL`

Objmode X

RPT –

CYC 1

Operands

P – Product register

T – Multiplicand register

loc16 – Addressing mode (see Chapter 5)

Description Multiply the signed 16-bit content of the T register by the signed 16-bit contents of the location pointed to by the “loc16” addressing mode and store the 32-bit result in the P register:

```
P = signed T * signed [loc16];
```

Flags and Modes None

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Calculate using 16-bit multiply:
; Y = (X0*C0) >> 2) + (X1*C1 >> 2) + (X2*C2 >> 2)
; X2 = X1
; X1 = X0
SPM −2             ; Set product shift to >> 2
MOVP T,@X2         ; T = X2
MPYS P,T,@C2       ; P = T*C2, ACC = 0
MOVAD T,@X1        ; T = X1, ACC = X2*C2>>2, X2 = X1
MPY P,T,@C1        ; P = T*C1
MOVAD T,@X0        ; T = X0, ACC = X1*C1>>2 + X2*C2>>2, X1 = X0
MPY P,T,@C0        ; P = T*C0
ADDL ACC, P << PM  ; ACC = X0*C0>>2 + X1*C1>>2 + X2*C2>>2
MOVL @Y,ACC        ; Store result into Y
```

MPYA P,loc16,#16bit *16 X 16-Bit Multiply and Add Previous Product*

Syntax Options MPYA P,loc16,#16bit

Opcode

```
0001 0101 LLLL LLLL
CCCC CCCC CCCC CCCC
```

Objmode X

RPT –

CYC 1

Operands

P – Product register

loc16 – Addressing mode (see Chapter 5)

#16bit – 16-bit immediate constant value

Description

Add the previous product (stored in the P register), shifted as specified by the product shift mode (PM) bits, to the ACC register. Load the T register with the content of the location pointed to by the “loc16” addressing mode. Multiply the signed 16-bit content of the T register by the signed 16-bit constant value and store the 32-bit result in the P register:

```
ACC = ACC + P << PM;
T = [loc16];
P = signed T * signed 16bit;
```

Flags and Modes

Flags and Modes	Description
Z	After the operation, the Z flag is set if the ACC is zero, else Z is cleared.
N	After the addition, the N flag is set if bit 31 of the ACC is 1, else N is cleared.
C	If the addition generates a carry, C is set; otherwise C is cleared.
V	If an overflow occurs, V is set; otherwise V is not affected.
OVC	If overflow mode is disabled; and if the operation generates a positive overflow, then the counter is incremented. If overflow mode is disabled; and if the operation generates a negative overflow, then the counter is decremented.
OVM	If overflow mode bit is set; then the ACC value will saturate maximum positive (0x7FFFFFFF) or maximum negative (0x80000000) if the operation overflowed.
PM	The value in the PM bits sets the shift mode for the output operation from the product register. If the product shift value is positive (logical left shift operation), then the low bits are zero filled. If the product shift value is negative (arithmetic right shift operation), the upper bits are sign extended.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Calculate using 16-bit multiply:
; Y = (X0*C0) >> 2) + (X1*C1 >> 2) + (X2*C2 >> 2),
; C0, C1 and C2 are constants
SPM −2    ;    Set product shift to >> 2
MOVB ACC,#0        ; Zero ACC
MPY P,@X2,#C2      ; P = X2*C2
MPYA P,@X1,#C1     ; ACC = X2*C2>>2, P = X1*C1
MPYA P,@X0,#C0     ; ACC = X1*C1>>2 + X2*C2>>2, P = X0*C0
ADDL ACC, P << PM  ; ACC = X0*C0>>2 + X1*C1>>2 + X2*C2>>2
MOVL @Y,ACC        ; Store result into Y
```

MPYA P,T,loc16 *16 X 16-bit Multiply and Add Previous Product*

Syntax Options MPYA P,T,loc16

Opcode `0001 0111 LLLL LLLL`

Objmode X

RPT Y

CYC N+1

Operands

P – Product register

T – Multiplicand register

loc16 – Addressing mode (see Chapter 5)

Description Add the previous product (stored in the P register), shifted as specified by the product shift mode (PM), to the ACC register. Multiply the signed 16-bit content of T by the signed 16-bit content of the location pointed to by the “loc16” addressing mode and store the 32-bit result in the P register:

```
ACC = ACC +    P << PM;
P    = signed T * signed [loc16];
```

Flags and Modes

Flags and Modes	Description
Z	After the operation, the Z flag is set if the ACC is zero, else Z is cleared.
N	After the addition, the N flag is set if bit 31 of the ACC is 1, else N is cleared.
C	If the addition generates a carry, C is set; otherwise C is cleared.
V	If an overflow occurs, V is set; otherwise V is not affected.
OVC	If overflow mode is disabled; and if the operation generates a positive overflow, then the counter is incremented. If overflow mode is disabled; and if the operation generates a negative overflow, then the counter is decremented.
OVM	If overflow mode bit is set; then the ACC value will saturate maximum positive (0x7FFFFFFF) or maximum negative (0x80000000) if the operation overflowed.
PM	The value in the PM bits sets the shift mode for the output operation from the product register. If the product shift value is positive (logical left shift operation), then the low bits are zero filled. If the product shift value is negative (arithmetic right shift operation), the upper bits are sign extended.

Repeat This instruction is repeatable. If the operation follows a RPT instruction, then it will be executed N+1 times. The state of the Z, N, C and OVC flags will reflect the final result. The V flag will be set if an intermediate overflow occurs.

Example

```
; Calculate using 16-bit multiply:
; Y = (X0*C0) >> 2) + (X1*C1 >> 2) + (X2*C2 >> 2)
SPM –2             ; Set product shift to >> 2
MOVP T,@X2         ; ACC = P, T = X2
MPYS P,T,@C2       ; ACC = ACC – P = 0, P = T*C2
MOV T,@X1          ; T = X1
MPYA P,T,@C1       ; ACC = X2*C2>>2, P = T*C1
MOV T,@X0          ; T = X0
MPYA P,T,@C0       ; ACC = X1*C1>>2 + X2*C2>>2, P = T*C0
ADDL ACC,P << PM   ; ACC = X0*C0>>2 + X1*C1>>2 + X2*C2>>2
MOVL @Y,ACC        ; Store result into Y
```

MPYB ACC,T,#8bit *Multiply by 8-bit Constant*

Syntax Options MPYB ACC,T,#8bit

Opcode `0011 0101 CCCC CCCC`

Objmode X

RPT –

CYC 1

Operands

ACC – Accumulator register

T – Multiplicand register

#8bit – 8-bit immediate constant value

Description Multiply the signed 16-bit content of the T register by the unsigned 8-bit constant value zero extended and store the result in the ACC register:

```
ACC = signed T * 0:8bit
```

Flags and Modes

Flags and Modes	Description
Z	After the operation, the Z flag is set if the ACC is zero, else Z is cleared.
N	After the operation, the N flag is set if bit 31 of the ACC is 1, else N is cleared.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Calculate signed using 16-bit multiply:
; Y32 = Y32 + (X16 * 5)
MOV T,@X16     ; T = X16
MPYB ACC,T,#5  ; ACC = T * 5
ADDL @Y32,ACC  ; Y32 = Y32 + ACC
```

MPYB P,T,#8bit *Multiply Signed Value by Unsigned 8-bit Constant*

Syntax Options MPYB P,T,#8bit

Opcode `0011 0001 CCCC CCCC`

Objmode X

RPT –

CYC 1

Operands

P – Product register

T – Multiplicand register

#8bit – 8-bit immediate constant value

Description Multiply the signed 16-bit content of the T register by the unsigned 8-bit immediate constant value zero extended and store the 32-bit result in the P register:

```
P = signed T * 0:8bit;
```

Flags and Modes None

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Calculate: Y32 = X16 * 5;
MOV T,@X16    ; T = X16
MPYB P,T,#5   ; P = T * #5
MOVL @Y,P     ; Store result into Y32
```

MPYS P,T,loc16 *16 X 16-bit Multiply and Subtract*

Syntax Options MPYS P,T,loc16

Opcode `0001 0011 LLLL LLLL`

Objmode X

RPT Y

CYC N+1

Operands

P – Product register

T – Multiplicand register

loc16 – Addressing mode (see Chapter 5)

Description Subtract the previous product (stored in the P register), shifted as specified by the product shift mode (PM), from the ACC register. In addition, multiply the signed 16-bit content of the T register by the signed 16-bit constant value and store the result in the P register:

```
ACC = ACC - P << PM;
P = signed T * signed [loc16];
```

Flags and Modes

Flags and Modes	Description
Z	After the operation, the Z flag is set if the ACC is zero, else Z is cleared.
N	After the subtraction, the N flag is set if bit 31 of the ACC is 1, else N is cleared.
C	If the subtraction generates a carry, C is set; otherwise C is cleared.
V	If an overflow occurs, V is set; otherwise V is not affected.
OVC	If overflow mode is disabled; and if the operation generates a positive overflow, then the counter is incremented. If overflow mode is disabled; and if the operation generates a negative overflow, then the counter is decremented.
OVM	If overflow mode bit is set; then the ACC value will saturate maximum positive (0x7FFFFFFF) or maximum negative (0x80000000) if the operation overflowed.
PM	The value in the PM bits sets the shift mode for the output operation from the product register. If the product shift value is positive (logical left shift operation), then the low bits are zero filled. If the product shift value is negative (arithmetic right shift operation), the upper bits are sign extended.

Repeat This instruction is repeatable. If the operation follows a RPT instruction, then it will be executed N+1 times. The state of the Z, N, C and OVC flags will reflect the final result. The V flag will be set if an intermediate overflow occurs.

Example

```
; Calculate using 16-bit multiply:
; Y = (X0*C0) >> 2) + (X1*C1 >> 2) + (X2*C2 >> 2)
SPM -2              ; Set product shift to >> 2
MOVP T,@X2          ; ACC = P, T = X2
MPYS P,T,@C2        ; ACC = ACC - P = 0, P = T*C2
MOV T,@X1           ; T = X1
MPYA P,T,@C1        ; ACC = X2*C2>>2, P = T*C1
MOV T,@X0           ; T = X0
MPYA P,T,@C0        ; ACC = X1*C1>>2 + X2*C2>>2, P = T*C0
ADDL ACC, P << PM   ; ACC = X0*C0>>2 + X1*C1>>2 + X2*C2>>2
MOVL @Y,ACC         ; Store result into Y
```

MPYU P,T,loc16 *Unsigned 16 X 16 Multiply*

Syntax Options

MPYU P,T,loc16

Opcode

`0011 0111 LLLL LLLL`

Objmode

X

RPT

–

CYC

1

Operands

P – Product register

T – Multiplicand register

loc16 – Addressing mode (see Chapter 5)

Description

Multiply the signed 16-bit content of the T register by the signed 16-bit contents of the location pointed to by the “loc16” addressing mode and store the 32-bit result in the P register:

```
P = unsigned T * unsigned [loc16];
```

Flags and Modes

None

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Calculate unsigned value: Y32 = X16 * M16;
MOV T,@X16     ; T = X16
MPYU P,T,@M16  ; P = T * M16
MOVL @Y,P      ; Store result into Y32
```

MPYU ACC,T,loc16 *16 X 16-bit Unsigned Multiply*

Syntax Options MPYU ACC,T,loc16

Opcode `0011 0110 LLLL LLLL`

Objmode X

RPT –

CYC 1

Operands **ACC** – Accumulator register

T – Multiplicand register

loc16 – Addressing mode (see Chapter 5)

Description Multiply the unsigned 16-bit content of the T register by the unsigned 16-bit content of the location pointed to by the “loc16” addressing mode and store the 32-bit results in the ACC register:

```
ACC = unsigned T * unsigned [loc16];
```

Flags and Modes

Flags and Modes	Description
Z	After the operation, the Z flag is set if the ACC is zero, else Z is cleared.
N	After the operation, the N flag is set if bit 31 of the ACC is 1, else N is cleared.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Calculate unsigned using 16-bit multiply:
; Y32 = Y32 + X16*M16
MOV T,@X16       ; T = X16
MPYU ACC,T,@M16  ; ACC = T * M16
ADDL @Y32,ACC    ; Y32 = Y32 + ACC
```

MPYXU ACC, T, loc16 *Multiply Signed Value by Unsigned Value*

Syntax Options MPYXU ACC, T, loc16

Opcode `0011 0000 LLLL LLLL`

Objmode X

RPT –

CYC 1

Operands

ACC – Accumulator register

T – Multiplicand register

loc16 – Addressing mode (see Chapter 5)

Description Multiply the signed 16-bit content of the T register by the unsigned 16-bit content of the location pointed to by the “loc16” addressing mode and store the result in the ACC register:

```
ACC = signed T * unsigned [loc16];
```

Flags and Modes

Flags and Modes	Description
Z	After the operation, the Z flag is set if the ACC is zero, else Z is cleared.
N	After the operation, the N flag is set if bit 31 of the ACC is 1, else N is cleared.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Calculate signed using 16-bit multiply:
; Y32 = Y32 + (signed) X16 * (unsigned) M16
MOV T,@X16        ; T = X16
MPYXU ACC,T,@M16  ; ACC = T * M16
ADDL @Y32,ACC     ; Y32 = Y32 + ACC
```

MPYXU P,T,loc16 *Multiply Signed Value by Unsigned Value*

Syntax Options MPYXU P,T,loc16

Opcode `0011 0010 LLLL LLLL`

Objmode X

RPT –

CYC 1

Operands

P – Product register

T – Multiplicand register

loc16 – Addressing mode (see Chapter 5)

Description Multiply the signed 16-bit content of the T register by the signed 16-bit contents of the location pointed to by the “loc16” addressing mode and store the 32-bit result in the P register:

```
P = signed T * unsigned [loc16];
```

Flags and Modes None

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Calculate "Y32 = X32 * M32" by parts using 16-bit multiply:
MOV T,@X32+0       ; T = unsigned low X32
MPYU ACC,T,@M32+0  ; ACC = T * unsigned low M32
MOV @Y32+0,AL      ; Store low result into Y32
MOVU ACC,@AH       ; Logical shift right ACC by 16
MOV T,@X32+1       ; T = signed high X32
MPYXU P,T,@M32+0   ; ACC = T * low unsigned M32
MOVA T,@M32+1      ; T = signed high M32, ACC += P
MPYXU P,T,@X32+0   ; ACC = T * low unsigned X32
ADDL ACC,P @P      ; Add P to ACC
MOV @Y32+1,AL      ; Store high result into Y32
```

NASP *Unalign Stack Pointer*

Syntax Options NASP

Opcode `0111 0110 0001 0111`

Objmode X

RPT –

CYC 1

Operands None

Description If the SPA bit is 1, the NASP instruction decrements the stack pointer (SP) by 1 and then clears the SPA status bit. This undoes a stack pointer alignment performed earlier by the ASP instruction. If the SPA bit is 0, then the NASP instruction performs no operation.

```
if(SPA = 1)
  {
  SP = SP – 1;
  SPA = 0;
  }
```

Flags and Modes

Flags and Modes	Description
SPA	If (SPA = 1), then SPA is cleared.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Alignment of stack pointer in interrupt service routine:
; Vector table:

INTx: .long INTx-   ; INTx interrupt vector
Service
  .
  .
  INTxService:
    ASP             ; Align stack pointer
    .
    .
    .
    NASP            ; Re–align stack pointer
    IRET            ; Return from interrupt.
```

NEG ACC *Negate Accumulator*

Syntax Options NEG ACC

Opcode `1111 1111 0101 0100`

Objmode X

RPT –

CYC 1

Operands **ACC** – Accumulator register

Description Negate the contents of the ACC register:

```
if(ACC = 0x8000 0000)
  {
   V = 1;
   if(OVM = 1)
     ACC = 0x7FFF FFFF;
   else
     ACC = 0x8000 0000;
  }
else
  ACC = −ACC;
if(ACC = 0x0000 0000)
  C = 1;
else
  C = 0;
```

Flags and Modes

Flags and Modes	Description
N	After the operation, the N flag is set if bit 31 of the ACC is 1, else N is cleared.
Z	After the operation, the Z flag is set if the ACC is zero, else Z is cleared.
C	If (ACC = 0), set C; otherwise, clear C.
V	If (ACC = 0x8000 0000) at the start of the operation, this is considered an overflow value and V is set. Otherwise, V is not affected.
OVM	If (ACC = 0x8000 0000) at the start of the operation, this is considered an overflow value, and the ACC value after the operation depends on the state of OVM: If OVM is cleared, ACC will be filled with 0x8000 0000. If OVM is set ACC will be saturated to 0x7FFF FFFF.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Negate contents of VarA, make sure value is saturated:
MOVL ACC,@VarA  ; Load ACC with contents of VarA
SETC OVM        ; Turn overflow mode on
NEG ACC         ; Negate ACC and saturate
MOVL @VarA,ACC  ; Store result into VarA
```

NEG AX *Negate AX Register*

Syntax Options NEG AX

Opcode `1111 1111 0101 110A`

Objmode X

RPT –

CYC 1

Operands **AX** – Accumulator high (AH) or accumulator low (AL) register

Description Replace the contents of the specified AX register with the negative of AX:

```
if(AX = 0x8000)
  {
  AX = 0x8000;
  V flag = 1;
  }
else
  AX = −AX;
if(AX = 0x0000)
  C flag = 1;
else
  C flag = 0;
```

Flags and Modes

Flags and Modes	Description
N	After the operation, if bit 15 of AX is 1, then the negative flag bit is set; otherwise, it is cleared.
Z	After the operation, if AX is 0, then the Z bit is set, otherwise it is cleared.
C	If AX is 0, C is set; otherwise, it is cleared.
V	If AX is 0x8000 at the start of the operation, then this is considered an overflow and V is set. Otherwise V is not affected.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Take the absolute value of VarA:
  MOV AL,@VarA          ; Load AL with contents of VarA
  NEG AL                ; If Al = 8000h, then V = 1

  SB NoOver- flow,NOV   ; Branch and save −AL if no overflow
  MOV                   ; Save 7FFF if overflow
@VarA,0x7FFFh
NoOverflow:
  MOV @VarA,AL          ; Save NEG AL if no overflow
```

NEG64 ACC:P *Negate Accumulator Register and Product Register*

Syntax Options NEG64 ACC:P

Opcode `0101 0110 0101 1000`

Objmode 1

RPT –

CYC 1

Operands **ACC:P** – Accumulator register (ACC) and product register (P)

Description Negate the 64-bit content of the combined ACC:P registers:

```
if(ACC:P = 0x8000 0000 0000 0000)
  {
  V = 1;
  if(OVM = 1)
    ACC:P = 0x7FFF FFFF FFFF FFFF;
  else
    ACC:P = 0x8000 0000 0000 0000;
  }
else
  ACC:P = −ACC:P;
if(ACC:P = 0x0000 0000 0000 0000)
  C = 1;
else
  C = 0;
```

Flags and Modes

Flags and Modes	Description
N	After the shift, if bit 31 of the ACC register is 1 then ACC:P is negative and the N bit is set; otherwise N is cleared.
Z	After the operation, the Z flag is set if the combined 64-bit value of the ACC:P is zero; otherwise, Z is cleared.
C	If (ACC:P= = 0) then the C bit is set; otherwise C is cleared.
V	if(ACC:P = 0x8000 0000 0000 0000) then the V flag is set; otherwise, V is not modified.
OVM	If at the start of the operation, ACC:P = 0x8000 0000 0000 0000, then this is considered an overflow value and the ACC:P value after the operation depends on OVM. If (OVM = 1) ACC:P is filled with its greatest positive number (0x7FFF FFFF FFFF FFFF). If (OVM = 0) then ACC:P is not modified.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Negate the contents of the 64-bit Var64 and saturate:
MOVL ACC,@Var64+2  ; Load ACC with high 32-bits of Var64
MOVL P,@Var64+0    ; Load P with low 32-bits of Var64
SETC OVM           ; Enable overflow mode (saturate)
NEG64 ACC:P        ; Negate ACC:P with saturation
MOVL @Var64+2,ACC  ; Store high 32-bit result into Var64
MOVL @Var64+0,P    ; Store low 32-bit result into Var64
```

NEGTC ACC *If TC is Equivalent to 1, Negate ACC*

Syntax Options NEGTC ACC

Opcode `0101 0110 0011 0010`

Objmode 1

RPT –

CYC 1

Operands **ACC** – Accumulator register

Description Based on the state of the test control (TC) bit, conditionally replace the content of the ACC register with its negative:

```
if( TC = 1 )
  {
  if(ACC = 0x8000 0000)
    {
     V = 1;
     if(OVM = 1)
       ACC = 0x7FFF FFFF;
     else
       ACC = 0x8000 0000
     }
  else
    ACC = -ACC;
  if(ACC = 0x0000 0000)
    C = 1;
  else
    C = 0;
  }
```

Flags and Modes

Flags and Modes	Description
N	After the operation, the N flag is set if bit 31 of the ACC is 1, else N is cleared.
Z	After the operation, the Z flag is set if the ACC is zero, else Z is cleared.
C	If (TC = 1 AND ACC = 0) set C; if (TC = 1 AND ACC != 0) clear C; otherwise C is not modified.
V	If (TC = 1 AND ACC = 0x8000 0000) at the start of the operation, this is considered an overflow value and V is set. Otherwise, V is not affected.
TC	The state of the TC bit is used as a test condition for the operation.
OVM	If at the start of the operation, ACC = 0x8000 0000, then this is considered an overflow value and the ACC value after the operation depends on OVM. If OVM is cleared and TC = 1, ACC will be filled with 0x8000 0000. If OVM is set and TC = 1, ACC will be saturated to 0x7FFF FFFF.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Calculate signed: Quot16 = Num16/Den16, Rem16 = Num16%Den16
CLRC TC                ; Clear TC flag, used as sign flag
MOV ACC,@Den16 << 16   ; AH = Den16, AL = 0
ABSTC ACC              ; Take abs value, TC = sign ^ TC
MOV T,@AH              ; Temp save Den16 in T register
MOV ACC,@Num16 << 16   ; AH = Num16, AL = 0
ABSTC ACC              ; Take abs value, TC = sign ^ TC
MOVU ACC,@AH           ; AH = 0, AL = Num16
RPT #15                ; Repeat operation 16 times
||SUBCU @T             ; Conditional subtract with Den16
MOV @Rem16,AH          ; Store remainder in Rem16
MOV ACC,@AL << 16      ; AH = Quot16, AL = 0
NEGTC ACC              ; Negate if TC = 1
MOV @Quot16,AH         ; Store quotient in Quot16
```

NOP {*ind}{ARPn} *No Operation With Optional Indirect Address Modification*

Syntax Options NOP {*ind}{ARPn}

Opcode `0111 0111 LLLL LLLL`

Objmode X

RPT Y

CYC N+1

Operands **{*ind}** – Indirect address mode (see chapter 5)

ARPn – Auxiliary register pointer (ARP0 to ARP7)

Description Modify the indirect address operand as specified and change the auxiliary register pointer (ARP) to the given auxiliary register. If no operands are given, then do nothing.

Flags and Modes None

Repeat This instruction is repeatable. If this instruction follows the RPT instruction, it will execute N+1 times.

Example

```
; Copy the contents of Array1 to Array2:
    ; int32 Array1[N];
    ; int32 Array2[N];
    ; for(i=0; i < N; i++)
    ; Array2[i] = Array1[i];
    ; This example only works for code located in upper 64K
    ; of program space:
  MOVL XAR2,#Array1    ; XAR2 = pointer to Array1
  MOVL XAR3,#Array2    ; XAR3 = pointer to Array2
  MOV @AR0,#(N-1)      ; Repeat loop N times
  NOP *,ARP2           ; Point to XAR2 (ARP = 2)
  SETC AMODE           ; Full C2xLP address mode compatible
Loop:
  MOVL ACC,*           ; ACC = Array1[i]
  NOP *++,ARP3         ; Increment XAR2 and point to XAR3
  RPT #19              ; Do nothing for 20 cycles
||NOP
  MOVL *++,ACC,ARP0    ; Array2[i] = ACC, point to XAR0
  XBANZ Loop,*--,ARP2  ; Loop if AR[ARP] != 0, AR[ARP]--,
                       ; point to XAR2
```

NORM ACC, *ind *Normalize ACC and Modify Selected Auxiliary Register*

Syntax Options

Syntax Options	Opcode	Objmode	RPT	CYC
NORM ACC, *	`0101 0110 0010 0100`	1	Y	N+4
NORM ACC, *++	`0101 0110 0101 1010`	1	Y	N+4
NORM ACC, *- -	`0101 0110 0010 0000`	1	Y	N+4
NORM ACC, *0++	`0101 0110 0111 0111`	1	Y	N+4
NORM ACC, *0- -	`0101 0110 0011 0000`	1	Y	N+4

Operands

ACC – Accumulator register

***ind** – *, *++, *− −, *0++, *0− − indirect addressing modes (see Chapter 5)

Description

Normalize the signed content of the ACC register and modify, as specified by the indirect addressing mode, the auxiliary register (XAR0 to XAR7) pointed to by the auxiliary register pointer (ARP).

Note: The NORM instruction normalizes a signed number in the ACC register by finding the magnitude of the number. An XOR operation is performed on ACC bits 31 and 30. If the bits are the same, then the content of the ACC register is logically shifted left by 1 to eliminate the extra sign bit and the selected pointer is modified. If the bits are different, the ACC is not shifted and the selected pointer is not modified. The selected pointer does not access any memory location.

Flags and Modes

Flags and Modes	Description
Z	After the operation, the Z flag is set if the ACC value is zero, else Z is cleared.
N	After the operation, the N flag is set if bit 31 of the ACC is 1, else N is cleared.
TC	If the operation set TC, no normalization was needed (ACC did not need to be modified). If the operation cleared TC, bits 31 and 30 were the same and, as a result, the ACC register was logically shifted left by 1.
ARP	Auxiliary register pointer selects which pointer to modify as part of the operation (XAR0 to XAR7).

Repeat

This instruction is repeatable. If the operation follows a RPT instruction, then the NORM instruction will be executed N+1 times. The state of the Z, N, and TC flags will reflect the final result. Note: If you only want the NORM instruction to execute until normalization is done, you can create a loop that checks the value of the TC bit. When TC = 1, normalization is complete.

Example

```
; Normalize the contents of VarA,
; XAR2 will contain shift value at the end of the operation:
  MOVL ACC,@VarA  ; ACC = VarA
  MOVB XAR2,#0    ; Initialize XAR2 to zero
  NOP *,ARP2      ; Set ARP pointer to point to XAR2
  SBF Skip,EQ     ; Skip if ACC value is zero
  RPT #31         ; Repeat next operation 32 times
||NORM ACC,*++    ; Normalize contents of ACC
Skip:
```

NORM ACC,XARn++/− − *Normalize ACC and Modify Selected Auxiliary Register*

Syntax Options

Syntax Options	Opcode	Objmode	RPT	CYC
NORM ACC,XARn++	`1111 1111 0111 1nnn`	X	Y	N+4
NORM ACC,XARn- -	`1111 1111 0111 0nnn`	X	Y	N+4

Operands

ACC – Accumulator register

XARn – XAR0 to XAR7, auxiliary registers post incremented or decremented

Description

Normalize the signed content of the ACC register and modify the specified auxiliary register (XAR0 to XAR7):

```
if(ACC != 0x0000 0000)
  {
  if((ACC(31) XOR ACC(30)) = 0)
    {
    ACC = ACC << , TC = 0;
    if(XARn++ addressing mode) XARn += 1;
    if(XARn−− addressing mode) XARn −= 1;
    }
  else
    TC = 1;
  }
else
  TC = 1;
```

Note: The NORM instruction normalizes a signed number in the ACC register by finding the magnitude of the number. An XOR operation is performed on ACC bits 31 and 30. If the bits are the same, then the content of the ACC register is logically shifted left by 1 to eliminate the extra sign bit and the selected pointer is modified. If the bits are different, the ACC is not shifted and the selected pointer is not modified. The selected pointer does not access any memory location.

Flags and Modes

Flags and Modes	Description
Z	After the operation, the Z flag is set if the ACC value is zero, else Z is cleared.
N	After the operation, the N flag is set if bit 31 of the ACC is 1, else N is cleared.
TC	If the operation set TC, no normalization was needed (ACC did not need to be modified). If the operation cleared TC, bits 31 and 30 were the same and, as a result, the ACC register was logically shifted left by 1.

Repeat

This instruction is repeatable. If the operation follows a RPT instruction, then the NORM instruction will be executed N+1 times. The state of the Z, N, and TC flags will reflect the final result. Note: If you only want the NORM instruction to execute until normalization is done, you can create a loop that checks the value of the TC bit. When TC = 1, normalization is complete.

Example

```
; Normalize the contents of VarA,
; XAR2 will contain shift value at the end of the operation:
  MOVL ACC,@VarA   ; ACC = VarA
  MOVB XAR2,#0     ; Initialize XAR2 to zero
  SBF Skip,EQ      ; Skip if ACC value is zero
  RPT #31          ; Repeat next operation 32 times
||NORM ACC,XAR2++  ; Normalize contents of ACC
Skip:
```

NOT ACC *Complement Accumulator*

Syntax Options NOT ACC

Opcode `1111 1111 0101 0101`

Objmode X

RPT –

CYC 1

Operands **ACC** – Accumulator register

Description The content of the ACC register is replaced with its complement:

```
ACC = ACC XOR 0xFFFFFFFF;
```

Flags and Modes

Flags and Modes	Description
N	After the operation, the N flag is set if bit 31 of the ACC is 1, else N is cleared.
Z	After the operation, the Z flag is set if the ACC is zero, else Z is cleared.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Complement the contents of VarA:
MOVL ACC,@VarA  ; ACC = VarA
NOT ACC         ; Complement ACC contents
MOVL @VarA,ACC  ; Store result into VarA
```

NOT AX *Complement AX Register*

Syntax Options NOT AX

Opcode `1111 1111 0101 111A`

Objmode X

RPT –

CYC 1

Operands **AX** – Accumulator high (AH) or accumulator low (AL) register

Description Replace the contents of the specified AX register (AH or AL) with its complement:

```
AX = AX XOR 0xFFFF;
```

Flags and Modes

Flags and Modes	Description
N	After the operation, if bit 15 of AX is 1 then the negative flag bit is set; otherwise it is cleared.
Z	After the operation, if AX is 0, then the Z bit is set, otherwise it is cleared.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Complement the contents of VarA:
MOV AL,@VarA  ; Load AL with contents of VarA
NOT AL        ; Complement contents of AL
MOV @VarA,AL  ; Store result in VarA
```

OR ACC, loc16 *Bitwise OR*

Syntax Options OR ACC, loc16

Opcode `1010 1111 LLLL LLLL`

Objmode 1

RPT Y

CYC N+1

Operands **ACC** – Accumulator register

loc16 – Addressing mode (see Chapter 5)

Description Perform a bitwise OR operation on the ACC register with the zero-extended content of the location pointed to by the “loc16” address mode. The result is stored in the ACC register:

```
ACC = ACC OR 0:[loc16];
```

Flags and Modes

Flags and Modes	Description
N	The load to ACC is tested for a negative condition. If bit 31 of ACC is 1, then the negative flag bit is set; otherwise it is cleared.
Z	The load to ACC is tested for a zero condition. The zero flag bit is set if the operation generates ACC = 0; otherwise it is cleared.

Repeat This operation is repeatable. If the operation follows a RPT instruction, then the OR instruction will be executed N+1 times. The state of the Z and N flags will reflect the final result.

Example

```
; Calculate the 32-bit value: VarA = VarA OR 0:VarB
MOVL ACC,@VarA  ; Load ACC with contents of VarA
OR ACC,@VarB    ; OR ACC with contents of 0:VarB
MOVL @VarA,ACC  ; Store result in VarA
```

OR ACC,#16bit << #0..16 *Bitwise OR*

Syntax Options

Syntax Options	Opcode	Objmode	RPT	CYC

OR ACC,#16bit << #0..15	`0011 1110 0001 SHFT`
	`CCCC CCCC CCCC CCCC`
		1	–	1

OR ACC,#16bit << #16	`0101 0110 0100 1010`
	`CCCC CCCC CCCC CCCC`
		1	–	1

Operands

ACC – Accumulator register

#16bit – 16-bit immediate constant value

#0..16 – Shift value (default is "<<#0" if no value specified)

Description

Perform a bitwise OR operation on the ACC register with the given 16-bit unsigned constant value left shifted as specified. The value is zero extended and lower order bits are zero filled before the OR operation. The result is stored in the ACC register:

```
ACC = ACC OR (0:16bit << shift value);
```

Flags and Modes

Flags and Modes	Description
N	The load to ACC is tested for a negative condition. If bit 31 of ACC is 1, then the negative flag bit is set; otherwise it is cleared.
Z	The load to ACC is tested for a zero condition. The zero flag bit is set if the operation generates ACC = 0; otherwise it is cleared.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Calculate the 32-bit value: VarA = VarA OR 0x08000000
MOVL ACC,@VarA        ; Load ACC with contents of VarA
OR ACC,#0x8000 << 12  ; OR ACC with 0x08000000
MOVL @VarA,ACC        ; Store result in VarA
```

OR AX, loc16 *Bitwise OR*

Syntax Options OR AX, loc16

Opcode `1100 101A LLLL LLLL`

Objmode X

RPT –

CYC 1

Operands **AX** – Accumulator high (AH) or accumulator low (AL) register

loc16 – Addressing mode (see Chapter 5)

Description Perform a bitwise OR operation on the specified AX register with the contents of the location pointed to by the “loc16” addressing mode. The result is stored in AX:

```
AX = AX OR [loc16];
```

Flags and Modes

Flags and Modes	Description
N	The load to AX is tested for a negative condition. If bit 15 of AX is 1, then the negative flag bit is set; otherwise it is cleared.
Z	The load to AX is tested for a zero condition. The zero flag bit is set if the operation generates AX = 0, otherwise it is cleared.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; OR the contents of VarA and VarB and store in VarC:
MOV AL,@VarA  ; Load AL with contents of VarA
OR AL,@VarB   ; OR AL with contents of VarB M
OV @VarC,AL   ; Store result in VarC
```

OR IER,#16bit *Bitwise OR*

Syntax Options OR IER,#16bit

Opcode

```
0111 0110 0010 0011
CCCC CCCC CCCC CCCC
```

Objmode X

RPT –

CYC 2

Operands **IER** – Interrupt enable register

#16bit-Mask – 16-bit immediate constant value

Description Enable specific interrupts by performing a bitwise OR operation with the IER register and the 16-bit immediate value. The result is stored in the IER register. Any changes take effect before the next instruction is processed.

```
IER = IER OR #16bit;
```

Flags and Modes None

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Enable INT1 and INT6 only. Do not modify state of other
; interrupt’s enable:
OR IER,#0x0061  ; Enable INT1 and INT6
```

OR IFR,#16bit *Bitwise OR*

Syntax Options OR IFR,#16bit

Opcode

```
0111 0110 0010 0111
CCCC CCCC CCCC CCCC
```

Objmode X

RPT –

CYC 2

Operands

IFR – Interrupt flag register

#16bit – 16-bit immediate constant value

Description

Enable specific interrupts by performing a bitwise OR operation with the IFR register and the 16-bit immediate value. The result of the OR operation is stored in the IFR register.

```
IFR = IFR OR #16bit;
```

Note: Interrupt hardware has priority over CPU instruction operation in cases where the interrupt flag is being simultaneously modified by the hardware and the instruction.

This instruction should not be used with interrupts 1−12 when the peripheral interrupt expansion (PIE) block is enabled.

Flags and Modes None

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Trigger INT1 and INT6 only. Do not modify state of other
; interrupt’s flags:
OR IFR,#0x0061  ; Trigger INT1 and INT6
```

OR loc16,#16bit *Bitwise OR*

Syntax Options OR loc16,#16bit

Opcode

```
0001 1010 LLLL LLLL
CCCC CCCC CCCC CCCC
```

Objmode X

RPT –

CYC 1

Operands

loc16 – Addressing mode (see Chapter 5)

#16bit – 16-bit immediate constant value

Description

Perform a bitwise OR operation on the content of the location pointed to by the “loc16” addressing mode and the 16-bit immediate constant value. The result is stored in the location pointed to by “loc16”:

```
[loc16] = [loc16] OR 16bit;
```

Smart Encoding:

If loc16 = AH or AL and #16bit is an 8-bit number, then the assembler will encode this instruction as ORB AX, #8bit to improve efficiency. To override this encoding, use the ORW AX, #16bit instruction alias.

Flags and Modes

Flags and Modes	Description
N	After the operation if bit 15 of [loc16] 1, set N; otherwise, clear N.
Z	After the operation if [loc16] is zero, set Z; otherwise, clear Z.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Set Bits 4 and 7 of VarA:
; VarA = VarA OR #(1 << 4 | 1 << 7)
OR @VarA,#(1 << 4 | 1 <<7)  ; Set bits 4 and 7 of VarA
```

OR loc16, AX *Bitwise OR*

Syntax Options OR loc16, AX

Opcode

```
1001 100A LLLL LLLL
```

Objmode X

RPT –

CYC 1

Operands

loc16 – Addressing mode (see Chapter 5)

AX – Accumulator high (AH) or accumulator low (AL) register

Description

Perform a bitwise OR operation on the contents of location pointed to by the “loc16” addressing mode with the specified AX register. The result is stored in the addressed location specified by “loc16”:

```
[loc16] = [loc16] OR AX;
```

This instruction performs a read-modify-write operation.

Flags and Modes

Flags and Modes	Description
N	The load to [loc16] is tested for a negative condition. If bit 15 of [loc16] is 1, then the negative flag bit is set; otherwise it is cleared.
Z	The load to [loc16] is tested for a zero condition. The zero flag bit is set if the operation generates [loc16] = 0, otherwise it is cleared.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; OR the contents of VarA with VarB and store in VarB:
MOV AL,@VarA  ; Load AL with contents of VarA
OR @VarB,AL   ; VarB = VarB OR AL
```

ORB AX,#8bit *Bitwise OR 8-bit Value*

Syntax Options ORB AX,#8bit

Opcode `0101 000A CCCC CCCC`

Objmode X

RPT –

CYC 1

Operands **AX** – Accumulator high (AH) or accumulator low (AL) register

#8bit – 8-bit immediate constant value

Description Perform a bitwise OR operation on the specified AX register with the 8-bit unsigned immediate constant zero extended. The result is stored in AX:

```
AX = AX OR 0x00:8bit;
```

Flags and Modes

Flags and Modes	Description
N	The load to AX is tested for a negative condition. If bit 15 of AX is 1, then the negative flag bit is set; otherwise it is cleared.
Z	The load to AX is tested for a zero condition. The zero flag bit is set if the operation generates AX = 0, otherwise it is cleared.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Set bit 7 of VarA and store result in VarB:
MOV    AL,@VarA    ; Load AL with contents of VarA
ORB    AL,#0x80    ; OR contents of AL with 0x0080
MOV    @VarB,AL    ; Store result in VarB
```

OUT *(PA),loc16 *Output Data to Port*

Syntax Options OUT *(PA),loc16

Opcode

```
1011 1100 LLLL LLLL
CCCC CCCC CCCC CCCC
```

Objmode 1

RPT –

CYC 4

Operands

***(PA)** – Immediate I/O space memory address

loc16 – Addressing mode (see Chapter 5)

Description

Store the 16-bit value from the location pointed to by the “loc16” addressing mode into the I/O space location pointed to by the *(PA) operand):

```
IOspace[0x0000PA] = [loc16];
```

I/O Space is limited to 64K range (0x0000 to 0xFFFF). On the external interface (XINTF), if available on a particular device, the I/O strobe signal (XISn) is toggled during the operation. The I/O address appears on the lower 16 XINTF address lines (XA(15:0)) and the upper address lines are zeroed. The data appears on the lower 16 data lines (XD(15:0).

Note: The UOUT operation is not pipeline protected. Hence, if an IN instruction immediately follows a UOUT instruction, the IN will occur before the UOUT. To be certain of the sequence of operation, use the OUT instruction, which is pipeline protected.

Note: The UOUT operation is not pipeline protected. Therefore, if an IN instruction immediately follows a UOUT instruction, the IN will occur before the UOUT. To be certain of the sequence of operation, use the OUT instruction, which is pipeline protected. I/O space may not be implemented on all C28x devices. See the data sheet for your particular device for details.

Flags and Modes None

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; IORegA address = 0x0300;
; IOREgB address = 0x0301;
; IOREgC address = 0x0302;
; IORegA = 0x0000;
; IORegB = 0x0400;
; IORegC = VarA;
; if(IORegC = 0x2000)
;    IORegC = 0x0000;
IORegA .set 0x0300     ; Define IORegA address
IORegB .set 0x0301     ; Define IORegB address
IORegC .set 0x0302     ; Define IORegC address
  MOV @AL,#0           ; AL = 0
  UOUT *(IORegA),@AL   ; IOspace[IORegA] = AL
  MOV @AL,#0x0400      ; AL = 0x0400
  UOUT *(IORegB),@AL   ; IOspace[IORegB] = AL
  OUT *(IORegC),@VarA  ; IOspace[IORegC] = VarA
  IN @AL,*(IORegC)     ; AL = IOspace[IORegC]
  CMP @AL,#0x2000      ; Set flags on (AL − 0x2000)
  SB $10,NEQ           ; Branch if not equal
  MOV @AL,#0           ; AL = 0
  UOUT *(IORegC),@AL   ; IOspace[IORegC] = AL
$10:
```

POP ACC *Pop Top of Stack to Accumulator*

Syntax Options POP ACC

Opcode `0000 0110 1011 1110`

Objmode X

RPT –

CYC 1

Operands **ACC** – Accumulator

Description Predecrement SP by 2. Load ACC with the 32-bit value pointed to by SP:

```
SP -= 2;
ACC = [SP];
```

Flags and Modes

Flags and Modes	Description
N	The load to ACC is tested for a negative condition. Bit-31 of the ACC register is the sign bit, 0 for positive, 1 for negative. The negative flag bit is set if the operation on the ACC register generates a negative value, otherwise it is cleared.
Z	The load to ACC is tested for a zero condition. The bit is set if the result of the operation on the ACC register generates a 0 value, otherwise it is cleared .

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

POP ARn:ARm *Pop Top of Stack to 16-bit Auxiliary Registers*

Syntax Options

Syntax Options	Opcode	Objmode	RPT	CYC
POP AR1:AR0	`0111 0110 0000 0111`	X	–	1
POP AR3:AR2	`0111 0110 0000 0101`	X	–	1
POP AR5:AR4	`0111 0110 0000 0110`	X	–	1

Operands **ARn:ARm** – AR1:AR0 or AR3:AR2 or AR5:AR4 auxiliary registers

Description AR1:AR0 or AR3:AR2 or AR5:AR4 Predecrement SP by 2. Load the contents of two 16-bit auxiliary registers (ARn and ARm)with the value pointed to by SP and SP+1.

```
POP AR1:AR0
  SP −= 2;
  AR0 = [SP];
  AR1 = [SP+1];
  AR1H:AR0H = unchanged;

POP AR3:AR2
  SP −= 2;
  AR2 = [SP];
  AR3 = [SP+1];
  AR3H:AR2H = unchanged;

POP AR5:AR4
  SP −= 2;
  AR4 = [SP];
  AR5 = [SP+1];
  AR5H:AR4H = unchanged;
```

Flags and Modes None

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

POP AR1H:AR0H *Pop Top of Stack to Upper Half of Auxiliary Registers*

Syntax Options POP AR1H:AR0H

Opcode `0000 0000 0000 0011`

Objmode X

RPT –

CYC 1

Operands **AR1H:AR0H** – Upper 16-bits of XAR1 and XAR0 auxiliary registers

Description Predecrement SP by 2. Load the contents of AR0H with the value pointed to by SP and AR1H with the value pointed to by SP+1. The lower 16 bits of the auxiliary registers (AR0 and AR1) are left unchanged.

```
SP    −= 2; AR0H
= [SP]; AR1H =
[SP+1];
AR1:AR0 = unchanged;
```

Flags and Modes None

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
. ; Full context restore for an
. ; interrupt or trap function
.
POP XT         ; 32-bit XT restore
POP XAR7       ; 32-bit XAR7 restore
POP XAR6       ; 32-bit XAR6 restore
POP XAR5       ; 32-bit XAR5 restore
POP XAR4       ; 32-bit XAR4 restore
POP XAR3       ; 32-bit XAR5 restore
POP XAR2       ; 32-bit XAR2 restore
POP AR1H:AR0H  ; 16-bit AR1H and 16-bit AR0H restore
IRET
```

POP DBGIER *Pop Top of Stack to DBGIER*

Syntax Options POP DBGIER

Opcode `0111 0110 0001 0010`

Objmode X

RPT –

CYC 5

Operands **DBGIER** – Debug interrupt-enable register

Description Predecrement SP by 1. Load the contents of DBGIER with the value pointed to by SP:

```
SP −= 1;
DBGIER = [SP];
```

Flags and Modes None

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

POP DP *Pop Top of Stack to the Data Page*

Syntax Options POP DP

Opcode `0111 0110 0000 0011`

Objmode X

RPT –

CYC 1

Operands **DP** – Data-page register

Description Predecrement SP by 1. Load the contents of DP with the value pointed to by SP:

```
SP −= 1;
DP = [SP];
```

Flags and Modes None

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

POP DP:ST1 *Pop Top of Stack to DP and ST1*

Syntax Options POP DP:ST1

Opcode `0111 0110 0000 0001`

Objmode X

RPT –

CYC 5

Operands **DP:ST1** – Data page register and status register 1

Description Predecrement SP by 2. Load ST1 with the value pointed to by SP and load DP with the value pointed to by SP+1:

```
SP −= 2;
ST1 = [SP];
DP = [SP+1];
```

Flags and Modes None

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

POP IFR *Pop Top of Stack to IFR*

Syntax Options	POP IFR
Opcode	`0000 0000 0000 0010`
Objmode	X
RPT	–
CYC	5
Operands	**IFR** – Interrupt flag register

Description

Predecrement SP by 1. Load the contents of IFR with the value pointed to by SP:

```
SP –= 1;
IFR = [SP];
```

Flags and Modes

None

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

POP loc16 *Pop Top of Stack*

Syntax Options POP loc16

Opcode `0010 1010 LLLL LLLL`

Objmode X

RPT –

CYC 2

Operands **loc16** – Addressing mode (See Chapter 5)

Description Predecrement SP by 1. Load the contents of loc16 with the 16-bit value pointed to by SP.

```
SP -= 1;
[loc16] = [SP];
```

Flags and Modes

Flags and Modes	Description
N	If (loc16 = @AX), then the load to AX is tested for a negative condition. Bit-15 of the AX register is the sign bit, 0 for positive, 1 for negative. The negative flag bit is set if the operation on the AX register generates a negative value, otherwise it is cleared.
Z	If (loc16 = @AX), then the load to AX is tested for a zero condition. The bit is set if the result of the operation on the AX register generates a 0 value, otherwise it is cleared.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
POP @T       ; Predecrement SP by 1. Load
             ; XT(31:15) with the
             ; contents of the location pointed to
             ; by SP. TL is unchanged.
POP @AL      ; Predecrement SP by 1. Load AL with
             ; the contents of the location pointed
             ; to by SP. AH is unchanged.
POP @AR4     ; Predecrement SP by 1. Load AR4 with
             ; the contents of the location pointed
             ; to by SP. AR4H is unchanged.
POP *XAR4++  ; Predecrement SP by 1. Load the
             ; 16-bit location pointed to by XAR4
             ; with the contents of the location
             ; pointed to by SP. Post-increment
             ; XAR4 by 1
```

POP P *Pop top of Stack to P*

Syntax Options POP P

Opcode `0111 0110 0001 0001`

Objmode X

RPT –

CYC 1

Operands **P** – Product register

Description Predecrement SP by 2. Load P with the 32-bit value pointed to by SP:

```
SP −= 2;
P = [SP];
```

Flags and Modes None

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

POP RPC *Pop RPC Register From Stack*

Syntax Options POP RPC

Opcode `0000 0000 0000 0111`

Objmode X

RPT –

CYC 3

Operands **RFC** – Return program counter register

Description Predecrement SP by 2. Load the contents of RPC with the value pointed to by SP:

```
SP −= 2;
RPC = [SP];
```

Flags and Modes None

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

POP ST0 *Pop Top of Stack to ST0*

Syntax Options POP ST0

Opcode

```
0111 0110 0001 0011
```

Objmode X

RPT –

CYC 1

Operands **ST0** – Status register 0

Description Predecrement SP by 1. Load the contents of ST0 with the value pointed to by SP:

```
SP -= 1;
ST0 = [SP];
```

Flags and Modes

Flags and Modes	Description
C N V Z TC SXM OVC PM	The bit value of each flag and mode listed is replaced by the value popped off of the stack

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

POP ST1 *Pop Top of Stack to ST1*

Syntax Options POP ST1

Opcode `0111 0110 0000 0000`

Objmode X

RPT –

CYC 5

Operands **ST1** – Status register 1

Description Predecrement SP by 1. Load the contents of ST0 with the value pointed to by SP:

```
SP -= 1;
ST1 = [SP];
```

Flags and Modes

Flags and Modes	Description
DBGM INTM VMAP SPA PAGE0 AMODE ARP EALLOW Objmode XF	The bit values for each flag and mode listed is replaced by the value popped off of the stack

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

POP T:ST0 *Pop Top of Stack to T and ST0*

Syntax Options POP T:ST0

Opcode

```
0111 0110 0001 0101
```

Objmode X

RPT –

CYC 1

Operands **T:ST0** – The upper 16-bits of the multiplicand register and status register 0

Description Predecrement SP by 2. Load ST0 with the value pointed to by SP and load T with the value pointed to by SP+1. The low 16 bits of the XT Register (TL) are left unchanged:

```
SP -= 2;
T = [SP];
ST0 = [SP+1];
TL = unchanged;
```

Flags and Modes None

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

POP XARn *Pop Top of Stack to 32-bit Auxiliary Register*

Syntax Options

Syntax Options	Opcode	Objmode	RPT	CYC
XAR0	`1000 1110 1011 1110`	1	–	1
XAR1	`1000 1011 1011 1110`	1	–	1
XAR2	`1000 0110 1011 1110`	1	–	1
XAR3	`1000 0010 1011 1110`	1	–	1
XAR4	`1000 1010 1011 1110`	1	–	1
XAR5	`1000 0011 1011 1110`	1	–	1
XAR6	`1100 0100 1011 1110`	X	–	1
XAR7	`1100 0101 1011 1110`	X	–	1

Operands

XARn – XAR0 to XAR7, 32-bit auxiliary registers

Description

Predecrement SP by 2. Load XARn with the 32-bit value pointed to by SP:

```
SP −= 2;
XARn = [SP];
```

Flags and Modes

None

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
. ; Full context restore for an
. ; interrupt or trap function
.
POP XT         ; 32-bit XT restore
POP XAR7       ; 32-bit XAR7 restore
POP XAR6       ; 32-bit XAR6 restore
POP XAR5       ; 32-bit XAR5 restore
POP XAR4       ; 32-bit XAR4 restore
POP XAR3       ; 32-bit XAR3 restore
POP XAR2       ; 32-bit XAR2 restore
POP AR1H:AR0H  ; 16-bit AR1H and 16-bit AR0H restore
IRET
```

POP XT *Pop Top of Stack to XT*

Syntax Options POP XT

Opcode `1000 0111 1011 1110`

Objmode X

RPT –

CYC 1

Operands **XT** – Multiplicand register

Description Predecrement SP by 2. Load XT with the 32-bit value pointed to by SP:

```
SP –= 2;
XT = [SP];
```

Flags and Modes None

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

PREAD loc16,*XAR7 *Read From Program Memory*

Syntax Options PREAD loc16,*XAR7

Opcode

```
0010 0100 LLLL LLLL
```

Objmode X

RPT Y

CYC N+2

Operands

loc16 – Addressing mode (see Chapter 5)

***XAR7** – Indirect program−memory addressing using auxiliary register XAR7, can access full 4Mx16 program space range (0x000000 to 0x3FFFFF)

Description

Load the data memory−location pointed to by the “loc16” addressing mode with the 16-bit content of the program−memory location pointed to by ”*XAR7”:

```
[loc16] = Prog[*XAR7];
```

On the C28x devices, memory blocks are mapped to both program and data space (unified memory), hence the ”*XAR7” addressing mode can be used to access data space variables that fall within the program space address range.

With some addressing mode combinations, you can get conflicting references. In such cases, the C28x will give the “loc16/loc32” field priority on changes to XAR7. For example:

```
PREAD *−−XAR7,*XAR7   ; *−−XAR7 given priority
PREAD *XAR7++,*XAR7  ; *XAR7++ given priority
```

Flags and Modes

Flags and Modes	Description
N	If (loc16 = @AX) and bit 15 of AX is 1, then N is set; otherwise N is cleared.
Z	If (loc16 = @AX) and the value of AX is zero, then Z is set; otherwise Z is cleared.

Repeat

This instruction is repeatable. If the operation follows a RPT instruction, then it will be executed N+1 times. When repeated, the ”*XAR7” program−memory address is copied to an internal shadow register and the address is post−incremented by 1 during each repetition.

Example

```
; Copy the contents of Array1 to Array2:
; int16 Array1[N]
;    // Located in program space
; int16 Array [N]
;    // Located in data space
; for(i=0; i N; i++)
;    Array2[i] = Array1[i];
  MOVL XAR7,#Array1    ; XAR7 = pointer to Array1
  MOVL XAR2,#Array2    ; XAR2 = pointer to Array2
  RPT #(N−1)           ; Repeat next instruction N times
||PREAD *XAR2++,*XAR7  ; Array2[i] = Array1[i],
                       ; i++
```

PUSH ACC *Push Accumulator Onto Stack*

Syntax Options PUSH ACC

Opcode `0001 1110 1011 1101`

Objmode X

RPT –

CYC 2

Note: This instruction is an alieas for the MOV*SP++, ACC instruction.

Operands **ACC** – Accumulator register

Description Push the 32-bit contents of ACC onto the stack pointed to by SP. Post-increment SP by 2:

```
[SP] = ACC;
SP += 2;
```

Flags and Modes None

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
MOVL XAR4, #VarA     ; Initialize XAR4 pointer with the
                     ; 22-bit address of VarA
MOVL ACC, *+XAR4[0]  ; Load the 32-bit contents of VarA
                     ; into ACC
PUSH ACC             ; Push the 32-bit ACC into the
                     ; location pointed to by SP.
                     ; Post-increment SP by 2
```

PUSH ARn:ARm *Push 16-bit Auxiliary REgisters Onto Stack*

Syntax Options

Syntax Options	Opcode	Objmode	RPT	CYC
PUSH AR1:AR0	`0111 0110 0000 1101`	X	–	1
PUSH AR3:AR2	`0111 0110 0000 1111`	X	–	1
PUSH AR5:AR4	`0111 0110 0000 1100`	X	–	1

Operands

ARn:ARm – AR1:AR0 or AR3:AR2 or AR5:AR4 auxiliary registers

Description

Push the contents of two 16-bit auxiliary registers (ARn and ARm) onto the stack pointed to by SP. Post-increment SP by 2:

```
PUSH AR1:AR0
  [SP] = AR0;
  [SP+1] = AR1;
  SP += 2;

PUSH AR3:AR2
  [SP] = AR2;
  [SP+1] = AR3;
  SP += 2;

PUSH AR5:AR4
  [SP] = AR4;
  [SP+1] = AR5;
  SP += 2;
```

Flags and Modes

None

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

PUSH AR1H:AR0H *Push AR1H and Ar0H Registers on Stack*

Syntax Options PUSH AR1H:AR0H

Opcode `0000 0000 0000 0101`

Objmode X

RPT –

CYC 1

Operands **AR1H:AR0H** – Upper 16-bits of XAR1 and XAR0 auxiliary registers

Description Push the contents of AR0H followed by the contents of AR1H onto the stack pointed to by SP. Post-increment SP by 2:

```
[SP]  = AR0H;
[SP+1] = AR1H;
SP += 2;
```

Flags and Modes None

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
IntX:            ; Full context save code for an
                 ; interrupt or trap function

PUSH AR1H:AR0H   ; 16-bit AR1H and 16-bit AR0H store
PUSH XAR2        ; 32-bit store of XAR2
PUSH XAR3        ; 32-bit store of XAR3
PUSH XAR4        ; 32-bit store of XAR4
PUSH XAR5        ; 32-bit store of XAR5
PUSH XAR6        ; 32-bit store of XAR6
PUSH XAR7        ; 32-bit store of XAR7
PUSH XT          ; 32-bit store of XT
```

PUSH DBGIER *Push DBGIER Register Onto Stack*

Syntax Options PUSH DBGIER

Opcode `0111 0110 0000 1110`

Objmode X

RPT –

CYC 1

Operands **DBGIER** – Debug interrupt enable register

Description Push the 16-bit contents of DBGIER onto the stack pointed to by SP. Post-increment SP by 1:

```
[SP] = DBGIER;
SP += 1;
```

Flags and Modes None

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

PUSH DP *Push DP Register Onto Stack*

Syntax Options PUSH DP

Opcode `0111 0110 0000 1011`

Objmode X

RPT –

CYC 1

Operands **DP** – Data-page register

Description Push the 16-bit contents of DP onto the stack pointed to by SP. Post-increment SP by 1:

```
[SP] = DP;
SP += 1;
```

Flags and Modes None

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

PUSH DP:ST1 *Push DP and ST1 Onto Stack*

Syntax Options PUSH DP:ST1

Opcode `0111 0110 0000 1001`

Objmode X

RPT –

CYC 1

Operands **DP:ST1** – Data-page register and status register 1

Description Push the 16bit contents of ST1 followed by the 16-bit contents of DP onto the stack pointed to by SP. Post-increment SP by 2:

```
[SP] = ST1;
[SP+1] = DP;
SP += 2;
```

Flags and Modes None

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

PUSH IFR *Push IFR Onto Stack*

Syntax Options PUSH IFR

Opcode `0111 0110 0000 1010`

Objmode X

RPT –

CYC 1

Operands **IFR** – Interrupt flag register

Description Push the 16-bit contents of IFR onto the stack pointed to by SP. Post-increment SP by 1:

```
[SP] = IFR;
SP += 1;
```

Flags and Modes None

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

PUSH loc16 *Push 16-bit Value on Stack*

Syntax Options PUSH loc16

Opcode `0010 0010 LLLL LLLL`

Objmode X

RPT –

CYC 2

Operands **loc16** – Addressing mode (see Chapter 5)

Description Push a 16-bit value pointed to by the “loc16” operand on the stack pointed to by SP. Post-increment SP by 1:

```
[SP] = [loc16];
SP += 1;
```

Flags and Modes None

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
PUSH @T       ; Push the contents of XT(31:15) into
              ; the location pointed to by
              ; SP. Post-increment SP by 1
PUSH @AL      ; Push the contents of AL onto into
              ; the location pointed to by
              ; SP. Post-increment SP by 1
PUSH @AR4     ; Push the lower 16-bits of XAR4 into
              ; the location pointed to by
              ; SP. Post-increment SP by 1
PUSH *XAR4++  ; Push the value pointed to by XAR4
              ; into the location pointed to
              ; by SP. Post-increment SP and XAR4
              ; by 1
```

PUSH P *Push P Onto Stack*

Syntax Options PUSH P

Opcode `0111 0110 0001 1101`

Objmode X

RPT –

CYC 1

Operands **P** – Product register

Description Push the 32-bit contents of P onto the stack pointed to by SP Post-increment SP by 2:

```
[SP] = P;
SP += 2;
```

Flags and Modes None

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
MOVL XAR5, #VarA   ; Initialize XAR5 pointer with the
                   ; 22-bit address of VarA
MOVL P, *+XAR5[0]  ; Load the 32-bit contents of VarA
                   ; into P
PUSH P             ; Push the 32-bit P into the
                   ; location pointed to by SP.
                   ; Post-increment SP by 2
```

PUSH RPC *Push RPC Onto Stack*

Syntax Options PUSH RPC

Opcode `0000 0000 0000 0100`

Objmode X

RPT –

CYC 1

Operands **RPC** – Return program counter register

Description Push the contents of the RPC register onto the stack pointed to by SP. Post-increment SP by 2:

```
[SP] = RPC;
SP += 2;
```

Flags and Modes None

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

PUSH ST0 *Push ST0 Onto Stack*

Syntax Options PUSH ST0

Opcode `0111 0110 0001 1000`

Objmode X

RPT –

CYC 1

Operands **ST0** – Status register 0

Description Push the 16-bit contents of ST0 onto the stack pointed to by SP. Post-increment SP by 1:

```
[SP] = ST0;
SP += 1;
```

Flags and Modes None

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

PUSH ST1 *Push ST1 Onto Stack*

Syntax Options PUSH ST1

Opcode `0111 0110 0000 1000`

Objmode X

RPT –

CYC 1

Operands **ST1** – Status register 1

Description Push the 16-bit contents of ST1 onto the stack pointed to by SP. Post-increment SP by 1:

```
[SP] = ST1;
SP += 1;
```

Flags and Modes None

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

PUSH T:ST0 *Push T and ST0 Onto Stack*

Syntax Options PUSH T:ST0

Opcode `0111 0110 0001 1001`

Objmode X

RPT –

CYC 1

Operands **T:ST0** – The upper 16-bits of the multiplicand register and status register 0

Description Push the 16- bit contents of ST0 followed by the 16-bit contents of T onto the stack pointed to by SP. Post-increment SP by 2:

```
[SP] = ST0;
[SP+1] = T;
SP += 2;
```

Flags and Modes None

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

PUSH XARn *Push 32-bit Auxiliary Register Onto Stack*

Syntax Options

Syntax Options	Opcode	Objmode	RPT	CYC
PUSH XAR0	`0011 1010 1011 1101`	1	–	1
PUSH XAR1	`1011 0010 1011 1101`	1	–	1
PUSH XAR2	`1010 1010 1011 1101`	1	–	1
PUSH XAR3	`1010 0010 1011 1101`	1	–	1
PUSH XAR4	`1010 1000 1011 1101`	1	–	1
PUSH XAR5	`1010 0000 1011 1101`	1	–	1
PUSH XAR6	`1100 0010 1011 1101`	X	–	1
PUSH XAR7	`1100 0011 1011 1101`	X	–	1

Operands **XARn** – XAR0 to XAR7, 32-bit auxiliary register

Description Push the 32-bit contents of XARn onto the stack pointed to by SP. Post-increment SP by 2:

```
[SP] = XARn;
 SP += 2;
```

Flags and Modes None

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
IntX:             ; Full context save code for an
                  ; interrupt or trap function

  PUSH AR1H:AR0H  ; 16-bit AR1H and 16-bit AR0H store
  PUSH XAR2       ; 32-bit store of XAR2
  PUSH XAR3       ; 32-bit store of XAR3
  PUSH XAR4       ; 32-bit store of XAR4
  PUSH XAR5       ; 32-bit store of XAR5
  PUSH XAR6       ; 32-bit store of XAR6
  PUSH XAR7       ; 32-bit store of XAR7
  PUSH XT         ; 32-bit store of XT
```

PUSH XT *Push XT Onto Stack*

Syntax Options PUSH XT

Opcode

```
1010 1011 1011 1101
```

Objmode X

RPT –

CYC 1

Operands **XT** – Multiplicand register

Description Push the 32-bit contents of XT onto the stack pointed to by SP. Post-increment SP by 2:

```
[SP] = XT;
SP += 2;
```

Flags and Modes None

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
MOVL XAR1, #VarA    ; Initialize XAR1 pointer with the
                    ; 22-bit address of VarA
MOVL XT, *+XAR5[0]  ; Load the 32-bit contents of VarA
                    ; into XT
PUSH XT             ; Push the 32-bit XT into the
                    ; location pointed to by SP.
                    ; Post-increment SP by 2
```

PWRITE *XAR7,loc16 *Write to Program Memory*

Syntax Options

PWRITE *XAR7,loc16

Opcode

```
0010 0110 LLLL LLLL
```

Objmode

X

RPT

Y

CYC

N+5

Operands

***XAR7** – Indirect program−memory addressing using auxiliary register XAR7, can access full 4Mx16 program space range (0x000000 to 0x3FFFFF)

loc16 – Addressing mode (see Chapter 5)

Description

Load the program−memory location pointed to by the ”*XAR7” with the content of the location pointed to by the “loc16” addressing mode:

```
Prog[*XAR7] = [loc16];
```

On the C28x devices, memory blocks are mapped to both program and data space (unified memory), hence the ”*XAR7” addressing mode can be used to access data space variables that fall within the program space address range.

With some addressing mode combinations, you can get conflicting references. In such cases, the C28x will give the “loc16/loc32” field priority on changes to XAR7. For example:

```
PWRITE *XAR7,*−−XAR7   ; *−−XAR7 given priority
PWRITE *XAR7,*XAR7++   ; *XAR7++ given priority
```

Flags and Modes

None

Repeat

This instruction is repeatable. If the operation follows a RPT instruction, then it will be executed N+1 times. When repeated, the ”*XAR7” program−memory address is copied to an internal shadow register and the address is post−incremented by 1 during each repetition.

Example

```
; Copy the contents of Array1 to Array2:
; int16 Array1[N];    // Located in data space
; int16 Array2[N];    // Located in program space
; for(i=0; i < N; i++)
;    Array2[i] = Array1[i];
  MOVL XAR2,#Array1     ; XAR2 = pointer to Array1
  MOVL XAR7,#Array2     ; XAR7 = pointer to Array2
  RPT #(N−1)            ; Repeat next instruction N times
||PWRITE *XAR7,*XAR2++  ; Array2[i] = Array1[i],
                        ; i++
```

QMACL P,loc32,*XAR7/++ *Signed 32 X 32-bit Multiply and Accumulate (Upper Half)*

Syntax Options

Syntax Options	Opcode	Objmode	RPT	CYC

QMACL P,loc32,*XAR7	`0101 0110 0100 1111`
	`1100 0111 LLLL LLLL`
		1	Y	N+2

QMACL P,loc32,*XAR7++	`0101 0110 0100 1111`
	`1000 0111 LLLL LLLL`
		1	Y	N+2

Operands

P – Product register

loc32 – Addressing mode (see Chapter 5)

Note: The @ACC addressing mode cannot be used when the instruction is repeated. No illegal instruction trap will be generated if used (assembler will flag an error).

***XAR7/++** – Indirect program−memory addressing using auxiliary register XAR7, can access full 4Mx16 program space range (0x000000 to 0x3FFFFF)

Description

32-bit x 32-bit signed multiply and accumulate. First, add the previous product (stored in the P register), shifted as specified by the product shift mode (PM), to the ACC register. Then, multiply the signed 32-bit content of the location pointed to by the “loc32” addressing mode by the signed 32-bit content of the program−memory location pointed to by the XAR7 register and store the upper 32−bits of the 64-bit result in the P register. If specified, post-increment the XAR7 register by 2:

```
ACC = ACC + P << PM;
P = (signed T * signed Prog[*XAR7 or *XAR7++]) >> 32;
```

On the C28x devices, memory blocks are mapped to both program and data space (unified memory), hence the ”*XAR7/++” addressing mode can be used to access data space variables that fall within the program space address range.

With some addressing mode combinations, you can get conflicting references. In such cases, the C28x will give the “loc16/loc32” field priority on changes to XAR7. For example:

```
QMACL P,*−−XAR7,*XAR7++  ; −−XAR7  given priority
QMACL P,*XAR7++,*XAR7    ; *XAR7++ given priority
QMACL P,*XAR7,*XAR7++    ; *XAR7++ given priority
```

Flags and Modes

Flags and Modes	Description
Z	After the addition, the Z flag is set if the ACC value is zero, else Z is cleared.
N	After the addition, the N flag is set if bit 31 of the ACC is 1, else N is cleared.
C	If the addition generates a carry, C is set; otherwise C is cleared.
V	If an overflow occurs, V is set; otherwise V is not affected.
OVC	If overflow mode is disabled; and if the operation generates a positive overflow, then the counter is incremented. If overflow mode is disabled; and if the operation generates a negative overflow, then the counter is decremented.
OVM	If overflow mode bit is set; then the ACC value will saturate maximum positive (0x7FFFFFFF) or maximum negative (0x80000000) if the operation overflowed.
PM	The value in the PM bits sets the shift mode for the output operation from the product register. If the product shift value is positive (logical left shift operation), then the low bits are zero filled. If the product shift value is negative (arithmetic right shift operation), the upper bits are sign extended.

Repeat

This instruction is repeatable. If the operation follows a RPT instruction, then it will be executed N+1 times. The state of the Z, N, C and OVC flags will reflect the final result in the ACC. The V flag will be set if an intermediate overflow occurs in the ACC.

Example

```
; Calculate sum of product using 32-bit multiply and retain
; high result:
; int32 X[N];    // Data information
; int32 C[N];    // Coefficient information (located in low 4M)
; int32 sum = 0;
; for(i=0; i < N; i++)
;    sum = sum + ((X[i] * C[i]) >> 32) >> 5;
  MOVL XAR2,#X             ; XAR2 = pointer to X
  MOVL XAR7,#C             ; XAR7 = pointer to C
  SPM −5                   ; Set product shift to  ">> 5"
  ZAPA                     ; Zero ACC, P, OVC
  RPT #(N−1)               ; Repeat next instruction N times
||QMACL P,*XAR2++,*XAR7++  ; ACC = ACC + P >> 5,
                           ; P = (X[i] * C[i]) >> 32
                           ; i++
  ADDL ACC, P << PM        ; Perform final accumulate
  MOVL @sum,ACC            ; Store final result into sum
```

QMPYAL P,XT,loc32 *Signed 32-bit Multiply (Upper Half) and Add Previous P*

Syntax Options QMPYAL P,XT,loc32

Opcode

```
0101 0110 0100 0110
0000 0000 LLLL LLLL
```

Objmode 1

RPT –

CYC 1

Operands

P – Product register

XT – Multiplicand register

loc32 – Addressing mode (see Chapter 5)

Description

Signed 32-bit x 32-bit multiply and accumulate the previous product. Add the previous signed product (stored in the P register), shifted as specified by the product shift mode (PM), to the ACC register. In addition, multiply the signed 32-bit content of the XT register by the signed 32-bit content of the location pointed to by the “loc32” addressing mode and store the upper 32−bits of the 64-bit result in the P register:

```
ACC = ACC + P << PM;
P = (signed T * signed [loc32]) >> 32;
```

Flags and Modes

Flags and Modes	Description
Z	After the addition, the Z flag is set if the ACC value is zero, else Z is cleared.
N	After the addition, the N flag is set if bit 31 of the ACC is 1, else N is cleared.
C	If the addition generates a carry, C is set; otherwise C is cleared.
V	If an overflow occurs, V is set; otherwise V is not affected.
OVC	If overflow mode is disabled; and if the operation generates a positive overflow, then the counter is incremented. If overflow mode is disabled; and if the operation generates a negative overflow, then the counter is decremented.
OVM	If overflow mode bit is set; then the ACC value will saturate maximum positive (0x7FFFFFFF) or maximum negative (0x80000000) if the operation overflowed.
PM	The value in the PM bits sets the shift mode for the output operation from the product register. If the product shift value is positive (logical left shift operation), then the low bits are zero filled. If the product shift value is negative (arithmetic right shift operation), the upper bits are sign extended.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Calculate signed result:
; Y32 = (X0*C0 + X1*C1 + X2*C2) >> (32 + 2)
SPM −2            ; Set product shift mode to ">> 2"
ZAPA              ; Zero ACC, P, OVC
MOVL XT,@X0       ; XT = X0
QMPYL P,XT,@C0    ; P = high 32−bits of (X0*C0)
MOVL XT,@X1       ; XT = X0
QMPYAL P,XT,@C1   ; ACC = ACC + P >> 2,
                  ; P = high 32−bits of (X1*C1)
MOVL XT,@X2       ; XT = X0
QMPYAL P,XT,@C2   ; ACC = ACC + P >> 2,
                  ; P = high 32−bits of (X2*C2)
ADDL ACC,P << PM  ; ACC = ACC + P >> 2
MOVL @Y32,ACC     ; Store result into Y32
```

QMPYL P,XT,loc32 *Signed 32 X 32-bit Multiply (Upper Half)*

Syntax Options

QMPYL P,XT,loc32

Opcode

```
0101 0110 0110 0111
0000 0000 LLLL LLLL
```

Objmode

1

RPT

–

CYC

1

Operands

P – Product register

XT – Multiplicand register

loc32 – Addressing mode (see Chapter 5)

Description

Multiply the signed 32-bit content of the XT register by the signed 32-bit content of the location pointed to by the “loc32” addressing mode and store the upper 32−bits of the 64-bit result (a Q30 number) in the P register:

```
P = (signed XT * signed [loc32]) >> 32;
```

Flags and Modes

None

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Calculate signed result: Y64 = M32*X32 + B64
MOVL XT,@M32     ; XT = M32
IMPYL P,XT,@X32  ; P = low 32−bits of (M32*X32)
MOVL ACC,@B64+2  ; ACC = high 32−bits of B64
ADDUL P,@B64+0   ; P = P + low 32−bits of B64
MOVL @Y64+0,P    ; Store low 32-bit result into Y64
QMPYL P,XT,@X32  ; P = high 32−bits of (M32*X32)
ADDCL ACC,@P     ; ACC = ACC + P + carry
MOVL @Y64+2,ACC  ; Store high 32-bit result into Y64
```

QMPYL ACC,XT,loc32 *Signed 32 X 32-bit Multiply (Upper Half)*

Syntax Options QMPYL ACC,XT,loc32

Opcode

```
0101 0110 0110 0011
0000 0000 LLLL LLLL
```

Objmode 1

RPT –

CYC 2

Operands

P – Product register

XT – Multiplicand register

ACC – Accumulator register

Description Multiply the signed 32-bit content of the XT register by the signed 32-bit content of the location pointed to by the "loc32" addressing mode and store the upper 32-bits of the 64-bit result (a Q30 number) in the ACC register:

```
ACC = (signed XT * signed [loc32]) >> 32;
```

Flags and Modes

Flags and Modes	Description
Z	After the operation, the Z flag is set if the ACC value is zero, else Z is cleared.
N	After the operation, the N flag is set if bit 31 of the ACC is 1, else N is cleared.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Calculate signed result: Y64 = M32*X32
MOVL XT,@M32       ; XT = M32
IMPYL P,XT,@X32    ; P = low 32−bits of (M32*X32)
QMPYL ACC,XT,@X32  ; ACC = high 32−bits of (M32*X32)
MOVL @Y64+0,P      ; Store    result into Y64
MOVL @Y64+2,ACC
```

QMPYSL P,XT,loc32 *Signed 32-bit Multiply (Upper Half) and Subtract Previous P*

Syntax Options

QMPYSL P,XT,loc32

Opcode

```
0101 0110 0100 0101
0000 0000 LLLL LLLL
```

Objmode

1

RPT

–

CYC

1

Operands

P – Product register

XT – Multiplicand register

loc32 – Addressing mode (see Chapter 5)

Description

Signed 32-bit x 32-bit multiply and subtract the previous product. Subtract the previous signed product (stored in the P register), shifted as specified by the product shift mode (PM), from the ACC register. In addition, multiply the signed 32-bit content of the XT register by the signed 32-bit constant value and store the upper 32−bits of the 64-bit result in the P register:

```
ACC = ACC − P << PM;
P = (signed T * signed [loc32]) >> 32;
```

Flags and Modes

Flags and Modes	Description
Z	After the subtraction, the Z flag is set if the ACC value is zero, else Z is cleared.
N	After the subtraction, the N flag is set if bit 31 of the ACC is 1, else N is cleared.
C	If the subtraction generates a borrow, C is cleared; otherwise C is set.
V	If an overflow occurs, V is set; otherwise V is not affected.
OVC	If overflow mode is disabled; and if the operation generates a positive overflow, then the counter is incremented. If overflow mode is disabled; and if the operation generates a negative overflow, then the counter is decremented.
OVM	If overflow mode bit is set; then the ACC value will saturate maximum positive (0x7FFFFFFF) or maximum negative (0x80000000) if the operation overflowed.
PM	The value in the PM bits sets the shift mode for the output operation from the product register. If the product shift value is positive (logical left shift operation), then the low bits are zero filled. If the product shift value is negative (arithmetic right shift operation), the upper bits are sign extended.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Calculate signed result:
; Y32 = −(X0*C0 + X1*C1 + X2*C2) >> (32 + 2)
SPM −2              ; Set product shift mode to ">> 2"
ZAPA                ; Zero ACC, P, OVC
MOVL XT,@X0         ; XT = X0
QMPYL P,XT,@C0      ; P = high 32−bits of (X0*C0)
MOVL XT,@X1         ; XT = X0
QMPYSL P,XT,@C1     ; ACC = ACC − P >> 2,
                    ; P = high 32−bits of (X1*C1)
MOVL XT,@X2         ; XT = X0
QMPYSL P,XT,@C2     ; ACC = ACC − P >> 2,
                    ; P = high 32−bits of (X2*C2)
SUBL ACC,P << PM    ; ACC = ACC − P >> 2
MOVL @Y32,ACC       ; Store result into Y32
```

QMPYUL P,XT,loc32 *Unsigned 32 X 32-bit Multiply (Upper Half)*

Syntax Options QMPYUL P,XT,loc32

Opcode

```
0101 0110 0100 0111
0000 0000 LLLL LLLL
```

Objmode 1

RPT –

CYC 1

Operands

P – Product register

XT – Multiplicand register

loc32 – Addressing mode (see Chapter 5)

Description

Multiply the unsigned 32-bit content of the XT register by the unsigned 32-bit content of the location pointed to by the “loc32” addressing mode and store the upper 32−bits of the 64-bit result in the P register:

```
P = (unsigned XT * unsigned [loc32]) >> 32;
```

Flags and Modes None

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Calculate unsigned result: Y64 = M32*X32 + B64
MOVL XT,@M32      ; XT = M32
IMPYL P,XT,@X32   ; P = low 32–bits of (M32*X32)
MOVL ACC,@B64+2   ; ACC = high 32–bits of B64
ADDUL P,@B64+0    ; P = P + low 32–bits of B64
MOVL @Y64+0,P     ; Store low 32-bit result into Y64
QMPYUL P,XT,@X32  ; P = high 32–bits of (M32*X32)
ADDCL ACC,@P      ; ACC = ACC + P + carry
MOVL @Y64+2,ACC   ; Store high 32-bit result into Y64
```

QMPYXUL P,XT,loc32 *Signed X Unsigned 32-bit Multiply (Upper Half)*

Syntax Options QMPYXUL P,XT,loc32

Opcode

```
0101 0110 0100 0010
0000 0000 LLLL LLLL
```

Objmode 1

RPT –

CYC 1

Operands

P – Product register

XT – Multiplicand register

loc32 – Addressing mode (see Chapter 5)

Description

Multiply the signed 32-bit content of the XT register by the unsigned 32-bit content of the location pointed to by the “loc32” addressing mode and store the upper 32−bits of the 64-bit result in the P register:

```
P = (signed XT * unsigned [loc32]) >> 32;
```

Flags and Modes None

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Calculate signed result: Y64 = (M64*X64) >> 64 + B64
 ; Y64 = Y1:Y0, M64 = M1:M0, X64 = X1:X0, B64 = B1:B0
MOVL XT,@X1       ; XT = X1
QMPYXUL P,XT,@M0  ; P    = high 32−bits of (uns M0 * sign X1)
MOV @T,#32        ; T = 32
LSL64 ACC:P,T     ; ACC:P = ACC:P << T
ASR64 ACC:P,T     ; ACC:P = ACC:P >> T
MOVL @XAR4,P      ; XAR5:XAR4 = ACC:P
MOVL @XAR5,ACC
MOVL XT,@M1       ; XT = M1
QMPYXUL P,XT,@X0  ; P = high 32−bits of (sign M1 * uns X0)
MOV @T,#32        ; T = 32
LSL64 ACC:P,T     ; ACC:P = ACC:P << T
ASR64 ACC:P,T     ; ACC:P = ACC:P >> T
MOVL @XAR6,P      ; XAR7:XAR6 = ACC:P MOVL    @XAR7,ACC
IMPYL P,XT,@X1    ; P = low 32−bits of (sign M1 * sign X1)
QMPYL ACC,XT,@X1  ; ACC = high 32−bits of (sign M1 * sign X1)
ADDUL P,@XAR4     ; ACC:P = ACC:P + XAR5:XAR4
ADDCL ACC,@XAR5
ADDUL P,@XAR6     ; ACC:P = ACC:P + XAR7:XAR6
ADDCL ACC,@XAR7
ADDUL P,@B0       ; ACC:P = ACC:P + B64
ADDCL ACC,@B1
MOVL @Y0,P        ; Store result into Y64
MOVL @Y1,ACC
```

ROL ACC *Rotate Accumulator Left*

Syntax Options ROL ACC

Opcode `1111 1111 0101 0011`

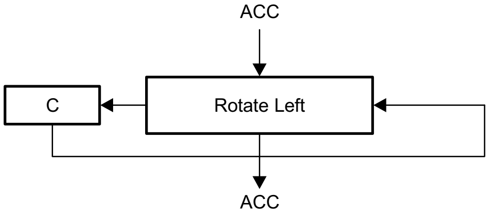
Objmode X

RPT Y

CYC N+1

Operands **ACC** – Accumulator register

Description Rotate the content of the ACC register left by one bit, filling bit 0 with the content of the carry flag and loading the carry flag with the bit shifted out:



Flags and Modes

Flags and Modes	Description
N	After the operation, the N flag is set if bit 31 of the ACC is 1, else N is cleared.
Z	After the operation, the Z flag is set if the ACC is zero, else Z is cleared.
C	The value in bit 31 of the ACC register is transferred to C. The value in C before the rotation is transferred to bit 0 of the ACC.

Repeat This instruction is repeatable. If the operation follows a RPT instruction, then the ROL instruction will be executed N+1 times. The state of the Z, N, and C flags will reflect the final result.

Example

```
; Rotate contents of VarA left by 5:
  MOVL ACC,@VarA  ; ACC = VarA
  RPT #4          ; Repeat next instruction 5 times
||ROL ACC         ; Rotate ACC left
  MOVL @VarA,ACC  ; Store result into VarA
```

ROR ACC *Rotate Accumulator Right*

Syntax Options ROR ACC

Opcode `1111 1111 0101 0010`

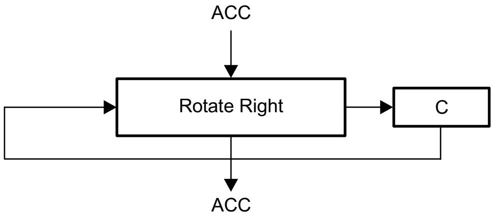
Objmode X

RPT Y

CYC N+1

Operands **ACC** – Accumulator register

Description Rotate the content of the ACC register right by one bit, filling bit 31 with the content of the carry flag and loading the carry flag with the bit shifted out:



Flags and Modes

Flags and Modes	Description
N	After the operation, the N flag is set if bit 31 of the ACC is 1, else N is cleared.
Z	After the operation, the Z flag is set if the ACC is zero, else Z is cleared.
C	The value in bit 0 of the ACC register is transferred to C. The value in C before the rotation is transferred to bit 31 of the ACC.

Repeat This instruction is repeatable. If the operation follows a RPT instruction, then the ROR instruction will be executed N+1 times. The state of the Z, N, and C flags will reflect the final result.

Example

```
; Rotate contents of VarA right by 5:
  MOVL ACC,@VarA  ; ACC = VarA
  RPT #4          ; Repeat next instruction 5 times
||ROR ACC         ; Rotate ACC right
  MOVL @VarA,ACC  ; Store result into VarA
```

RPT #8bit/loc16 *Repeat Next Instruction*

Syntax Options

Syntax Options	Opcode	Objmode	RPT	CYC
RPT #8bit	`1111 0110 CCCC CCCC`	X	–	1
RPT loc16	`1111 0111 LLLL LLLL`	X	–	4

Operands

#8bit – 8-bit constant immediate value (0 to 255 range)

loc16 – Addressing mode (see Chapter 5)

Description

Repeat the next instruction. An internal repeat counter (RPTC) is loaded with a value N that is either the specified #8bit constant value or the content of the location pointed to by the “loc16” addressing mode. After the instruction that follows the RPT is executed once, it is repeated N times; that is, the instruction following the RPT executes N + 1 times. Because the RPTC cannot be saved during a context switch, repeat loops are regarded as multicycle instructions and are not interruptible.

Note on syntax: Parallel bars (||) before the repeated instruction are used as a reminder that the instruction is repeated and is not interruptable. When writing inline assembly, use the syntax

```
asm(|| RPT #8bt/ loc16 || instruction");
```

Not all instructions are repeatable. If an instruction that is not repeatable follows the RPT instruction, the RPTC counter is reset to 0 and the instruction only executes once. The 28x Assembly Language tools check for this condition and issue warnings.

Flags and Modes

None

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Copy the number of elements specified in VarA from Array1
; to Array2:
; int16 Array1[N]; // Located in high 64K of program space
; int16 Array2[N]; // Located in data space
; for(i=0; i < VarA; i++)
;    Array2[i] = Array1[i];
  MOVL XAR2,#Array2  ; XAR2 = pointer to Array2
  RPT @VarA          ; Repeat next instruction
                     ; [VarA] + 1 times
|| XPREAD            ; Array2[i] = Array1[i],
*XAR2++,*(Array1)    ; i++
```

SAT ACC *Saturate Accumulator*

Syntax Options SAT ACC

Opcode `1111 1111 0101 0111`

Objmode X

RPT –

CYC 1

Operands **ACC** – Accumulator register

Description Saturate the ACC register to reflect the net overflow represented in the 6-bit overflow counter (OVC):

```
if(OVC > 0)
  ACC = 0x7FFF FFFF;
  V = 1;
if(OVC < 0)
  ACC = 0x8000 0000;
  V = 1;
if(OVC = 0)
  ACC = unchanged;
  OVC = 0;
  V = 0;
```

Flags and Modes

Flags and Modes	Description
N	After the operation, the N flag is set if bit 31 of the ACC is 1, else N is cleared.
Z	After the operation, the Z flag is set if the ACC is zero, else Z is cleared.
C	C is cleared.
V	If (OVC != 0) at the start of the operation, V is set; otherwise, V is cleared.
OVC	If (OVC > 0) then ACC is saturated to its maximum positive value. If (OVC < 0) then ACC is saturated to its maximum negative value. If (OVC = 0) then ACC is not modified. After the operation, OVC is cleared.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Add VarA, VarB and VarC and saturate result and store in VarD:
ZAP OVC         ; Clear overflow counter
MOVL ACC,@VarA  ; Load ACC with contents of VarA
ADDL ACC,@VarB  ; Add to ACC contents of VarB
ADDL ACC,@VarC  ; Add to ACC contents of VarC
SAT ACC         ; Saturate ACC based on OVC value
MOVL @VarD,ACC  ; Store result into VarD
```

SAT64 ACC:P *Saturate 64-bit Value ACC:P*

Syntax Options SAT64 ACC:P

Opcode `0101 0110 0011 1110`

Objmode 1

RPT –

CYC 1

Operands **ACC:P** – Accumulator register (ACC) and product register (P)

Description Saturate the 64-bit content of the combined ACC:P registers to reflect the net overflow represented in the overflow counter (OVC):

```
if(OVC > 0)
  ACC:P = 0x7FFF FFFF FFFF FFFF;
  V=1;
if(OVC < 0)
  ACC:P = 0x8000 0000 0000 0000;
  V=1;
if(OVC = 0)
  ACC:P = unchaged;
  OVC = 0;
```

Flags and Modes

Flags and Modes	Description
N	After the shift, if bit 31 of the ACC register is 1 then ACC:P is negative and the N bit is set; otherwise N is cleared.
Z	After the operation, the Z flag is set if the combined 64-bit value of the ACC:P is zero; otherwise, Z is cleared.
C	The C bit is cleared.
V	At the start of the operation, if (OVC = 0) then V is cleared; otherwise, V is set.
OVC	If (OVC = 0), then no saturation takes place: ACC:P is unchanged. If(OVC > 0), then saturate ACC:P the maximum positive value: ACC:P = 0x7FFF FFFF FFFF FFFF If(OVC < 0), then saturate ACC:P to the maximum negative value: ACC = 0x8000 0000 or ACC:P = 0x8000 0000 0000 0000 At the end of the operation, OVC is cleared.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Add 64-bit VarA, VarB and VarC, sat and store result in VarD:
ZAP OVC    ; Clear overflow counter
MOVL P,@VarA+0    ; Load P with low 32-bits of VarA
ADDUL P,@VarB+0   ; Add to P unsigned low 32-bits of VarB
ADDUL P,@VarC+0   ; Add to P unsigned low 32-bits of VarC
MOVU @AL,OVC      ; Store overlow (repeated carry) in the ACC
                  ; and then add higher portion of the 64 bit
                  ; variables
MOVB AH,#0        ; Store overlow (repeated carry) in the ACC
                  ; and then add higher portion of the 64 bit
                  ; variables
ZAP OVC           ; Clear overflow counter
ADDL ACC,@VarA+2  ; Add to ACC with carry high 32-bits of VarA
ADDL ACC,@VarB+2  ; Add to ACC with carry high 32-bits of VarB
ADDL ACC,@VarC+2  ; Add to ACC with carry high 32-bits of VarC
SAT64 ACC:P       ; Saturate ACC:P based on OVC value
MOVL @VarD+0,P    ; Store low 32-bit result into VarD
MOVL @VarD+2,ACC  ; Store high 32-bit result into VarD
```

SB 8bitOffset,COND *Need description here*

Syntax Options SB 8bitOffset,COND

Opcode `0110 COND CCCC CCCC`

Objmode X

RPT –

CYC 7/4

Operands **8bitOffset** – 8-bit signed immediate constant offset value (−128 to +127 range)

COND – Conditional codes:

COND	Syntax	Description	Flags Tested
0000	NEQ	Not Equal To	Z = 0
0001	EQ	Equal To	Z = 1
0010	GT	Greater Than	Z = 0 AND N = 0
0011	GEQ	Greater Than or Equal To	N = 0
0100	LT	Less Than	N = 1
0101	LEQ	Less Than or Equal To	Z = 1 OR N = 1
0110	HI	Higher	C = 1 AND Z = 0
0111	HIS, C	Higher or Same, Carry Set	C = 1
1000	LO, NC	Lower, Carry Clear	C = 0
1001	LOS	Lower or Same	C = 0 OR Z = 1
1010	NOV	No Overflow	V = 0
1011	OV	Overflow	V = 1
1100	NTC	Test Bit Not Set	TC = 0
1101	TC	Test Bit Set	TC = 1
1110	NBIO	BIO Input Equal To Zero	BIO = 0
1111	UNC	Unconditional	–

Description Short conditional branch. If the specified condition is true, then branch by adding the signed 8-bit constant value to the current PC value; otherwise continue execution without branching:

```
If (COND = true) PC = PC + signed 8-bit offset;
If (COND = false) PC = PC + 1;
```

Note: If (COND = true) then the instruction takes 7 cycles.

If (COND = false) then the instruction takes 4 cycles.

If (COND = UNC) then the instruction takes 4 cycles.

Flags and Modes

Flags and Modes	Description
V	If the V flag is tested by the condition, then V is cleared.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

SBBU ACC,loc16 *Subtract Unsigned Value Plus Inverse Borrow*

Syntax Options SBBU ACC,loc16

Opcode `0001 1101 LLLL LLLL`

Objmode X

RPT –

CYC 1

Operands **ACC** – Accumulator register

loc16 – Addressing mode (see Chapter 5)

Description Subtract the 16-bit contents of the location pointed to by the “loc16” addressing mode, zero extended, and subtract the compliment of the carry flag bit from the ACC register:

```
ACC = ACC - 0:[loc16] - ~C;
```

Flags and Modes

Flags and Modes	Description
Z	After the subtraction, the Z flag is set if ACC is zero, else Z is cleared.
N	After the subtraction, the N flag is set if bit 31 of the ACC is 1, else N is cleared.
C	The state of the carry bit before execution is included in the subtraction. If the subtraction generates a borrow, C is cleared; otherwise C is set.
V	If an overflow occurs, V is set; otherwise V is not affected.
OVC	If(OVM = 0, disabled) then if the operation generates a positive overflow, then the counter is incremented and if the operation generates a negative overflow, then the counter is decremented. If(OVM = 1, enabled) then the counter is not affected by the operation.
OVM	If overflow mode bit is set; then the ACC value will saturate maximum positive (0x7FFFFFFF) or maximum negative (0x80000000) if the operation overflowed.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Subtract three 32-bit unsigned variables by 16-bit parts:
MOVU ACC,@VarAlow        ; AH = 0, AL = VarAlow
ADD ACC,@VarAhigh << 16  ; AH = VarAhigh, AL = VarAlow
SUBU ACC,@VarBlow        ; ACC = ACC - 0:VarBlow
SUB ACC,@VarBhigh << 16  ; ACC = ACC - VarBhigh << 16
SBBU ACC,@VarClow        ; ACC = ACC - VarClow - ~Carry
SUB ACC,@VarChigh << 16  ; ACC = ACC - VarChigh << 16
```

SBF 8bitOffset,EQ/NEQ/TC/NTC *Short Branch Fast*

Syntax Options

Syntax Options	Opcode	Objmode	RPT	CYC
SBF 8bitOffset,EQ	`1110 1100 CCCC CCCC`	1	–	4/4
SBF 8bitOffset,NEQ	`1110 1101 CCCC CCCC`	1	–	4/4
SBF 8bitOffset,TC	`1110 1110 CCCC CCCC`	1	–	4/4
SBF 8bitOffset,NTC	`1110 1111 CCCC CCCC`	1	–	4/4

Operands **8bitOffset** – 8-bit signed immediate constant offset value (−128 to +127 range)

Syntax	Description	Flags Tested
NEQ	Not Equal To	Z = 0
EQ	Equal To	Z = 1
NTC	Test Bit Not Set	TC = 0
TC	Test Bit Set	TC = 1

Description Short fast conditional branch. If the specified condition is true, then branch by adding the signed 8-bit constant value to the current PC value; otherwise continue execution without branching:

```
If (tested condition = true) PC = PC + signed 8-bit off- set;
If (tested condition = false) PC = PC + 1;
```

Note: The short branch fast (SBF) instruction takes advantage of dual pre−fetch queue on the C28x core that reduces the cycles for a taken branch from 7 to 4:

```
If (tested condition = true) then the instruction takes 4 cycles.
If (tested condition = false) then the instruction takes 4 cycles.
```

Flags and Modes None

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

SBRK #8bit *Subtract From Current Auxiliary Register*

Syntax Options SBRK #8bit

Opcode `1111 1101 CCCC CCCC`

Objmode X

RPT –

CYC 1

Operands **#8bit** – 8-bit constant immediate value

Description Subtract the 8-bit unsigned constant from the XARn register pointed to by ARP:

```
XAR(ARP) = XAR(ARP) - 0:8bit;
```

Flags and Modes

Flags and Modes	Description
ARP	The 3-bit ARP points to the current valid auxiliary register, XAR0 to XAR7. This pointer determines which auxiliary register is modified by the operation.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
  .word 0xEEEE
  .word 0x0000

TableA:
  .word 0x1111
  .word 0x2222
  .word 0x3333
  .word 0x4444

FuncA:
  MOVL XAR1,#TableA  ; Initialize XAR1 pointer
  MOVZ AR2,*XAR1     ; Load AR2 with the 16-bit value
                     ; pointed to by XAR1 (0x1111)
                     ; Set ARP = 1

SBRK #2              ; Decrement XAR1 by 2

MOVZ AR3,*XAR1       ; Load AR3 with the 16-bit value
                     ; pointed to by XAR1 (0xEEEE)
```

SETC Mode *Set Multiple Status Bits*

Syntax Options

Syntax Options	Opcode	Objmode	RPT	CYC
SETC Mode	`0011 1011 CCCC CCCC`	X	–	1,2
SETC SXM	`0011 1011 0000 0001`	X	–	1
SETC OVM	`0011 1011 0000 0010`	X	–	1
SETC TC	`0011 1011 0000 0100`	X	–	1
SETC C	`0011 1011 0000 1000`	X	–	1
SETC INTM	`0011 1011 0001 0000`	X	–	2
SETC DBGM	`0011 1011 0010 0000`	X	–	2
SETC PAGE0	`0011 1011 0100 0000`	X	–	1
SETC VMAP	`0011 1011 1000 0000`	X	–	1

Operands **Mode** – 8-bit immediate mask (0x00 to 0xFF)

Description Set the specified status bits. Any change affects the next instruction in the pipeline. The ”mode” operand is a mask value that relates to the status bits in this way:

"Mode" bit	Status Register	Flag	Cycles
0	ST0	SXM	1
1	ST0	OVM	1
2	ST0	TC	1
3	ST0	C	1
4	ST1	INTM	2
5	ST1	DBGM	2
6	ST1	PAGE0	1
7	ST1	VMAP	1

Note: The assembler will accept any number of flag names in any order. For example:

```
SETC INTM,TC        ; Set INTM and TC bits to 1
SETC TC,INTM,OVM,C  ; Set TC, INTM, OVM, C bits to 1
```

Flags and Modes

Flags and Modes	Description
SXM OVM TC C INTM DBGM PAGE0 VMAP	Any of the specified bits can be set by the instruction.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once

Example

```
; Modify flag settings:
SETC INTM,DBGM     ; Set INTM and DBGM bits to 1
CLRC TC,C,SXM,OVM  ; Clear TC, C, SXM, OVM bits to 0
CLRC #0xFF         ; Clear all bits to 0
SETC #0xFF         ; Set all bits to 1
SETC C,SXM,TC,OVM  ; Set TC, C, SXM, OVM bits to 1
CLRC DBGM,INTM     ; Clear INTM and DBGM bits to 0
```

SETC M0M1MAP *Set the M0M1MAP Status Bit*

Syntax Options SETC M0M1MAP

Opcode `0101 0110 0001 1010`

Objmode X

RPT –

CYC 5

Operands **M0M1MAP** – Status bit

Description Set the M0M1MAP status bit, configuring the mapping of the M0 and M1 memory blocks for C28x/C2XLP operation. The memory blocks are mapped as follows:

M0M1MAP bit	Data Space	Program Space
0	M0: 0x000 to 0x3FF	M0: 0x400 to 0x7FF
(C27x)	M1: 0x400 to 0x7FF	M1: 0x000 to 0x3FF
1	M0: 0x000 to 0x3FF	
(C28x/C2XLP)	M1: 0x400 to 0x7FF	

Note: The pipeline is flushed when this instruction is executed.

Flags and Modes

Flags and Modes	Description
M0M1MAP	The M0M1MAP bit is set.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Set the device mode from reset to C28x:
Reset:
  SETC Objmode  ; Enable C28x Object Mode
  CLRC AMODE    ; Enable C28x Address Mode
  .c28_amode    ; Tell assembler we are in C28x address mode
  SETC M0M1MAP  ; Enable C28x Mapping Of M0 and M1 blocks
    .
    .
```

SETC Objmode *Set the Objmode Status Bit*

Syntax Options SETC Objmode

Opcode `0101 0110 0001 1111`

Objmode X

RPT –

CYC 5

Operands **Objmode** – Status bit

Description Set the Objmode status bit, putting the device in C28x object mode (supports C2XLP source).

Flags and Modes

Flags and Modes	Description
Objmode	Set the Objmode bit.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Set the device mode from reset to C28x:
Reset:
  SETC Objmode  ; Enable C28x Object Mode
  CLRC AMODE    ; Enable C28x Address Mode
  .c28_amode    ; Tell assembler we are in C28x address mode
  SETC M0M1MAP  ; Enable C28x Mapping Of M0 and M1 blocks
    .
    .
```

SETC XF *Set XF Bit and Output Signal*

Syntax Options SETC XF

Opcode `0101 0110 0010 0110`

Objmode X

RPT –

CYC 1

Operands **XF** – Status bit and output signal

Description Set the XF status bit and pull the corresponding output signal high.

Flags and Modes

Flags and Modes	Description
XF	The XF status bit is set.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Pulse XF signal high if branch not taken:
  MOV AL,@VarA  ; Load AL with contents of VarA
  SB Dest,NEQ   ; ACC = VarA
  SETC XF       ; Set XF bit and signal high
  CLRC XF       ; Clear XF bit and signal low
    .
    .
Dest:
    .
```

SFR ACC,#1..16 *Shift Accumulator Right*

Syntax Options SFR ACC,#1..16

Opcode `1111 1111 0100 SHFT`

Objmode X

RPT Y

CYC N+1

Operands **ACC** – Accumulator register

#1..16 – Shift value

Description Right shift the content of the ACC register by the amount specified in the shift field. The type of shift (arithmetic or logical) is determined by the state of the sign extension mode (SXM) bit:

```
if(SXM = 1)                       // sign extension mode enabled
    ACC = S:ACC >> shift value;  // arithmetic shift right
else                              //sign extension mode disabled
    ACC = 0:ACC >> shift value;  // logical shift right
```

Flags and Modes

Flags and Modes	Description
Z	After the shift, the Z flag is set if the ACC value is zero, else Z is cleared.
N	After the shift, the N flag is set if bit 31 of the ACC is 1, else N is cleared.
C	The last bit shifted out is loaded into the C flag bit.
SXM	If (SXM = 1), then the operation behaves like an arithmetic right shift. If (SXM = 0), then the operation behaves like a logical right shift.

Repeat This instruction is repeatable. If the operation follows a RPT instruction, then the SFR instruction will be executed N+1 times. The state of the Z, N and C flags will reflect the final result.

Example

```
; Arithmetic shift right contents of VarA by 10:
MOVL ACC,@VarA  ; ACC = VarA
SETC SXM        ; Enable sign extension mode
SFR ACC,#10     ; Arithmetic shift right ACC by 10
MOVL @VarA,ACC  ; Store result into VarA
```

SFR ACC,T *Shift Accumulator Right*

Syntax Options SFR ACC,T

Opcode `1111 1111 0101 0001`

Objmode X

RPT –

CYC N+1

Operands **ACC** – Accumulator register

T – Upper 16-bits of the multiplicand (XT) register

Description Right shift the content of the ACC register by the amount specified in the four least significant bits of the T register, T(3:0) = 0..15. Higher order bits are ignored. The type of shift (arithmetic or logical) is determined by the state of the sign extension mode (SXM) bit:

```
if(SXM = 1)                  // sign extension mode enabled
    ACC = S:ACC >> T(3:0);  // arithmetic shift right
else                         // sign extension mode disabled
    ACC = 0:ACC >> T(3:0);  // logical shift right
```

Flags and Modes

Flags and Modes	Description
Z	After the shift, the Z flag is set if the ACC value is zero, else Z is cleared. Even if the T register specifies a shift of 0, the content of the ACC register is still tested for the zero condition and Z is affected.
N	After the shift, the N flag is set if bit 31 of the ACC is 1, else N is cleared. Even if the T register specifies a shift of 0, the content of the ACC register is still tested for the negative condition and N is affected.
C	If (T(3:0) = 0) then C is cleared; otherwise, the last bit shifted out is loaded into the C flag bit.
SXM	If (SXM = 1), then the operation behaves like an arithmetic right shift. If (SXM = 0), then the operation behaves like a logical right shift.

Repeat This instruction is repeatable. If the operation follows a RPT instruction, then the SFR instruction will be executed N+1 times. The state of the Z, N and C flags will reflect the final result.

Example

```
; Arithmetic shift right contents of VarA by VarB:
MOVL ACC,@VarA  ; ACC = VarA
MOV T,@VarB     ; T = VarB (shift value)
SETC SXM        ; Enable sign extension mode
SFR ACC,T       ; Arithmetic shift right ACC by T(3:0)
MOVL @VarA,ACC  ; Store result into VarA
```

SPM shift *Set Product Mode Shift Bits*

Syntax Options

Syntax Options	Opcode	Objmode	RPT	CYC
SPM +1	`1111 1111 0110 1000`	X	–	1
SPM 0	`1111 1111 0110 1001`	X	–	1
SPM –1	`1111 1111 0110 1010`	X	–	1
SPM –2	`1111 1111 0110 1011`	X	–	1
SPM –3	`1111 1111 0110 1100`	X	–	1
SPM –4 (Valid only when AMODE = 0) SPM +4 (Valid only when AMODE = = 1)	`1111 1111 0110 1101`	X	–	1
SPM –5	`1111 1111 0110 1110`	X	–	1
SPM –6	`1111 1111 0110 1111`	X	–	1

Operands b – Product shift mode (+4, +1, 0, −1, −2, −3, −4, −5, −6)

Description Specify a product shift mode. A negative value indicates an arithmetic right shift; positive numbers indicate a logical left shift. The following table shows the relationship between the “shift” operand and the 3-bit value that gets loaded into the product shift mode (PM) bits in ST0. The address mode bit (AMODE) selects between two types of shift decodes as shown in the table below:

PM Bits	AMODE = 1	AMODE = 0
000	SPM +1	SPM +1
001	SPM 0	SPM 0
010	SPM –1	SPM –1
011	SPM –2	SPM –2
100	SPM –3	SPM –3
101	SPM +4	SPM –4
110	SPM –5	SPM –5
111	SPM –6	SPM –6

Flags and Modes

Flags and Modes	Description
PM	PM is loaded with the 3-bit value specified by the selected ”shift” value.

Repeat This instruction is not repeatable. If the operation follows a RPT instruction, it resets the repeat counter (RPTC) and executes once.

Example

```
; Calculate: Y32 = M16*X16 >>4 + B32
CLRC AMODE        ; Make sure AMODE = 0
SPM -4            ; Set product shift mode to ">> 4"
MOV T,@X16        ; T = X16
MPY P,XT,@M16     ; P = X16*M16
MOVL ACC,@B32     ; ACC = B32
ADDL ACC,P << PM  ; ACC = ACC + (P >> 4)
MOVL @Y32,ACC     ; Store result into Y32
```

SQRA loc16 ***Square Value and Add P to ACC***

Syntax Options SQRA loc16

Opcode

```
0101 0110 0001 0101
0000 0000 LLLL LLLL
```

Objmode 1

RPT Y

CYC N+1

Operands **loc16** – Addressing mode (see Chapter 5)

Description Add the previous product (stored in the P register), shifted by the amount specified by the product shift mode (PM), to the ACC register. Then the content of the location pointed to by the “loc16” addressing mode is loaded into the T register, squared, and stored in the P register:

```
ACC = ACC + P << PM;
T = [loc16];
P = T * [loc16];
```

Flags and Modes

Flags and Modes	Description
Z	After the addition, the Z flag is set if the ACC value is zero, else Z is cleared.
N	After the addition, the N flag is set if bit 31 of the ACC is 1, else N is cleared.
C	If the addition generates a carry, C is set; otherwise C is cleared.
V	If an overflow occurs, V is set; otherwise V is not affected.
OVC	If overflow mode is disabled; and if the operation generates a positive overflow, then the counter is incremented. If overflow mode is disabled; and if the operation generates a negative overflow, then the counter is decremented.
OVM	If overflow mode bit is set; then the ACC value will saturate maximum positive (0x7FFFFFFF) or maximum negative (0x80000000) if the operation overflowed.
PM	The value in the PM bits sets the shift mode for the output operation from the product register. If the product shift value is positive (logical left shift operation), then the low bits are zero filled. If the product shift value is negative (arithmetic right shift operation), the upper bits are sign extended.

Repeat This instruction is repeatable. If the operation follows a RPT instruction, then it will be executed N+1 times. The state of the Z, N, C and OVC flags will reflect the final result. The V flag is set if an intermediate overflow occurs.

Example

```
; Calculate sum of squares using 16-bit multiply:
; int16 X[N]        ; Data information
; sum = 0;
; for(i=0; i < N; i++)
;    sum = sum + (X[i] * X[i]) >> 5;
  MOVL XAR2,#X      ; XAR2 = pointer to X
  SPM -5            ; Set product shift to  ">> 5"
  ZAPA              ; Zero ACC, P, OVC
  RPT #N–1          ; Repeat next instruction N times
||SQRA *XAR2++      ; ACC = ACC + P >> 5,
                    ; P = (*XAR2++)^2
  ADDL ACC,P << PM  ; Perform final accumulate
  MOVL @sum,ACC     ; Store final result into sum
```

SQRS loc16 *Square Value and Subtract P From ACC*

Syntax Options SQRS loc16

Opcode

```
0101 0110 0001 0001
xxxx xxxx LLLL LLLL
```

Objmode 1

RPT Y

CYC N+1

Operands **loc16** – Addressing mode (see Chapter 5)

Description Subtract the previous product (stored in the P register), shifted by the amount specified by the product shift mode (PM), from the ACC register. Then the content of the location pointed to by the “loc16” addressing mode is loaded into the T register, squared, and stored in the P register:

```
ACC = ACC − P << PM;
T = [loc16];
P = T * [loc16];
```

Flags and Modes

Flags and Modes	Description
Z	After the addition, the Z flag is set if the ACC value is zero, else Z is cleared.
N	After the subtraction, the N flag is set if bit 31 of the ACC is 1, else N is cleared.
C	If the subtraction generates a borrow, C is cleared; otherwise C is set.
V	If an overflow occurs, V is set; otherwise V is not affected.
OVC	If overflow mode is disabled; and if the operation generates a positive overflow, then the counter is incremented. If overflow mode is disabled; and if the operation generates a negative overflow, then the counter is decremented.
OVM	If overflow mode bit is set; then the ACC value will saturate maximum positive (0x7FFFFFFF) or maximum negative (0x80000000) if the operation overflowed.
PM	The value in the PM bits sets the shift mode for the output operation from the product register. If the product shift value is positive (logical left shift operation), then the low bits are zero filled. If the product shift value is negative (arithmetic right shift operation), the upper bits are sign extended.

Repeat This instruction is repeatable. If the operation follows a RPT instruction, then it will be executed N+1 times. The state of the Z, N, C and OVC flags will reflect the final result. The V flag will be set if an intermediate overflow occurs.

Example

```
; Calculate sum of negative squares using 16-bit multiply:
; int16 X[N]        ; Data information
; sum = 0;
; for(i=0; i < N; i++)
;    sum = sum − (X[i] * X[i])  >> 5;
  MOVL XAR2,#X      ; XAR2 = pointer to X
  SPM -5            ; Set product shift to  ">> 5"
  ZAPA              ; Zero ACC, P, OVC
  RPT #N−1          ; Repeat next instruction N times
||SQRS *XAR2++      ; ACC = ACC − P >> 5,
                    ; P = (*XAR2++)^2
  SUBL ACC,P << PM  ; Perform final subtraction
  MOVL @sum,ACC     ; Store final result into sum
```

SUB ACC,loc16 << #0..16 *Subtract Shifted Value From Accumulator*

Syntax Options

Syntax Options	Opcode	Objmode	RPT	CYC
SUB ACC,loc16 << #0	`1010 1110 LLLL LLLL`	1	Y	N+1
	`1000 0000 LLLL LLLL`	0	–	1

SUB ACC,loc16 << #1..15	`0101 0110 0000 0000`
	`0000 SHFT LLLL LLLL`
		1	Y	N+1

	`1000 SHFT LLLL LLLL`	0	–	1
SUB ACC,loc16 << #16	`0000 0100 LLLL LLLL`	X	Y	N+1

Operands

ACC – Accumulator register

loc16 – Addressing mode (see Chapter 5)

#0..16 – Shift value (default is "<< #0” if no value specified)

Description

Subtract the left-shifted 16-bit location pointed to by the ”loc16” addressing mode from the ACC register. The shifted value is sign extended if sign extension mode is turned on (SXM=1) else the shifted value is zero extended (SXM= 0). The lower bits of the shifted value are zero filled:

```
if(SXM = 1)    // sign extension mode enabled
    ACC = ACC – S:[loc16] << shift value;
else           // sign extension mode disabled
    ACC = ACC – 0:[loc16] << shift value;
```

Flags and Modes

Flags and Modes	Description
Z	After the subtraction, the Z flag is set if ACC is zero, else Z is cleared.
N	After the subtraction, the N flag is set if bit 31 of the ACC is 1, else Z is cleared.
C	If the subtraction generates a borrow, C is cleared; otherwise C is set. Exception: If a shift of 16 is used, the SUB instruction can clear C but not set it.
V	If an overflow occurs, V is set; otherwise V is not affected.
OVC	If(OVM = 0, disabled) then if the operation generates a positive overflow, then the counter is incremented and if the operation generates a negative overflow, then the counter is decremented. If(OVM = 1, enabled) then the counter is not affected by the operation.
SXM	If sign extension mode bit is set; then the 16-bit operand, addressed by the ”loc16” field, will be sign extended before the addition. Else, the value will be zero extended.
OVM	If overflow mode bit is set; then the ACC value will saturate maximum positive (0x7FFF FFFF) or maximum negative (0x8000 0000) if the operation overflowed.

Repeat

If the operation is repeatable, then the instruction will be executed N+1 times. The state of the Z, N, C flags will reflect the final result. The V flag will be set if an intermediate overflow occurs. The OVC flag will count intermediate overflows, if overflow mode is disabled. If the operation is not repeatable, the instruction will execute only once.

Example

```
; Calculate signed value: ACC = (VarA << 10) - (VarB << 6);
SETC SXM              ; Turn sign extension mode on
MOV ACC,@VarA << #10  ; Load ACC with VarA left shifted by 10
SUB ACC,@VarB << #6   ; Subtract VarB left shifted by 6 to ACC0
```

SUB ACC,loc16 << T *Subtract Shifted Value From Accumulator*

Syntax Options SUB ACC,loc16 << T

Opcode

```
0101 0110 0010 0111
0000 0000 LLLL LLLL
```

Objmode 1

RPT Y

CYC N+1

Operands

ACC – Accumulator register

loc16 – Addressing mode (see Chapter 5)

T – Upper 16−bits of the multiplicand register, XT(31:16)

Description

Subtract from the ACC register the left−shifted contents of the 16-bit location pointed to by the “loc16” addressing mode. The shift value is specified by the four least significant bits of the T register, T(3:0) = shift value = 0..15. Higher order bits are ignored. The shifted value is sign extended if sign extension mode is turned on (SXM=1) else the shifted value is zero extended (SXM=0). The lower bits of the shifted value are zero filled:

```
if(SXM = 1)    // sign extension mode enabled
    ACC = ACC − S:[loc16] << T(3:0);
else           // sign extension mode disabled
    ACC = ACC − 0:[loc16] << T(3:0);
```

Flags and Modes

Flags and Modes	Description
Z	After the subtraction, the Z flag is set if the ACC value is zero, else Z is cleared.
N	After the subtraction, the N flag is set if bit 31 of the ACC is 1, else N is cleared.
C	If the subtraction generates a borrow, C is cleared; otherwise C is set.
V	If an overflow occurs, V is set; otherwise V is not affected.
OVC	If(OVM = 0, disabled) then if the operation generates a positive overflow, then the counter is incremented and if the operation generates a negative overflow, then the counter is decremented. If(OVM = 1, enabled) then the counter is not affected by the operation.
SXM	If sign extension mode bit is set; then the 16-bit operand, addressed by the ”loc16” field, will be sign extended before the addition. Else, the value will be zero extended.
OVM	If overflow mode bit is set; then the ACC value will saturate maximum positive (0x7FFF FFFF) or maximum negative (0x8000 0000) if the operation overflowed.

Repeat

If this operation is repeated, then the instruction will be executed N+1 times. The state of the Z, N, C flags will reflect the final result. The V flag will be set if an intermediate overflow occurs. The OVC flag will count intermediate overflows, if overflow mode is disabled.

Example

```
; Calculate signed value: ACC = (VarA << SB) − (VarB << SB)
SETC SXM             ; Turn sign extension mode on
MOV T,@SA            ; Load T with shift value in SA
MOV ACC,@VarA << T   ; Load in ACC shifted contents of VarA
MOV T,@SB            ; Load T with shift value in SB
SUB ACC,@VarB << T   ; Subtract from ACC shifted contents
                     ; of VarB
```

SUB ACC,#16bit << #0..15 *Subtract Shifted Value From Accumulator*

Syntax Options SUB ACC,#16bit << #0..15

Opcode

```
1111 1111 0000 SHFT
CCCC CCCC CCCC CCCC
```

Objmode X

RPT –

CYC 1

Operands

ACC – Accumulator register

#16bit – 16-bit immediate constant value

#0..15 – Shift value (default is ”<<#0” if no value specified)

Description

Subtract the left shifted 16-bit immediate constant value from the ACC register. The shifted value is sign extended if sign extension mode is turned on (SXM=1) else the shifted value is zero extended (SXM=0). The lower bits of the shifted value are zero filled:

```
if(SXM = 1)    // sign extension mode enabled
  ACC = ACC – S:16bit << shift value;
else           // sign extension mode disabled
  ACC = ACC – 0:16bit << shift value;
```

Smart Encoding:

If #16bit is an 8-bit number and the shift is zero, then the assembler will encode this instruction as SUBB ACC, #8bit for improved efficiency. To override this encoding, use the SUBW ACC, #16bit instruction alias.

Flags and Modes

Flags and Modes	Description
Z	After the subtraction, the Z flag is set if ACC is zero, else Z is cleared.
N	After the subtraction, the N flag is set if bit 31 of the ACC is 1, else N is cleared.
C	If the subtraction generates a borrow, C is cleared; otherwise C is set.
V	If an overflow occurs, V is set; otherwise V is not affected.
OVC	If(OVM = 0, disabled) then if the operation generates a positive overflow, then the counter is incremented and if the operation generates a negative overflow, then the counter is decremented. If(OVM = 1, enabled) then the counter is not affected by the operation.
SXM	If sign extension mode bit is set; then the 16-bit operand, addressed by the ”loc16” field, will be sign extended before the addition. Else, the value will be zero extended.
OVM	If overflow mode bit is set; then the ACC value will saturate maximum positive (0x7FFFFFFF) or maximum negative (0x80000000) if the operation overflowed.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Calculate signed value: ACC = (VarB << 10) - (23 <<  6);
SETC SXM              ; Turn sign extension mode on
MOV ACC,@VarB << #10  ; Load ACC with VarB left shifted by 10
SUB ACC,#23 << #6     ; Subtract from ACC 23 left shifted by 6
```

SUB AX, loc16 *Subtract Specified Location From AX*

Syntax Options SUB AX, loc16

Opcode `1001 111A LLLL LLLL`

Objmode X

RPT –

CYC 1

Operands

AX – Accumulator high (AH) or accumulator low (AL) register

loc16 – Addressing mode (see Chapter 5)

Description

Subtract the 16−bit content of the location pointed to by the “loc16” addressing mode from the specified AX register (AH or AL) and store the results in AX:

```
AX = AX - [loc16];
```

Flags and Modes

Flags and Modes	Description
N	After the subtraction, AX is tested for a negative condition. If bit 15 of AX is 1, then the negative flag bit is set; otherwise it is cleared.
Z	After the subtraction, AX is tested for a zero condition. The zero flag bit is set if the operation generates AX = 0, otherwise it is cleared.
C	If the subtraction generates a borrow, C is cleared; otherwise C is set.
V	If an overflow occurs, V is set; otherwise V is not affected. Signed positive overflow occurs if the result crosses the max positive value (0x7FFF) in the positive direction. Signed negative overflow occurs if the result crosses the max negative value (0x8000) in the negative direction.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Subtract the contents of VarA with VarB and store in VarC
MOV AL,@VarA  ; Load AL with contents of VarA
SUB AL,@VarB  ; Subtract from AL contents of VarB
MOV @VarC,AL  ; Store result in VarC
```

SUB loc16, AX *Reverse-Subtract Specified Location From AX*

Syntax Options SUB loc16, AX

Opcode `0111 010A LLLL LLLL`

Objmode X

RPT –

CYC 1

Operands **loc16** – Addressing mode (see Chapter 5)

AX – Accumulator high (AH) or accumulator low (AL) register

Description Subtract the content of the specified AX register (AH or AL) from the 16-bit content of the location pointed to by the “loc16” addressing mode and store the result in location pointed to by ”loc16”:

```
[loc16] = [loc16] − AX;
```

Flags and Modes

Flags and Modes	Description
N	After the subtraction, [loc16] is tested for a negative condition. If bit 15 of [loc16] is 1, then the negative flag bit is set; otherwise it is cleared.
Z	After the subtraction, [loc16] is tested for a zero condition. The zero flag bit is set if the operation generates [loc16] = 0; otherwise it is cleared
C	If the subtraction generates a borrow, C is cleared; otherwise C is set.
V	If an overflow occurs, V is set; otherwise V is not affected. Signed positive overflow occurs if the result crosses the max positive value (0x7FFF) in the positive direction. Signed negative overflow occurs if the result crosses the max negative value (0x8000) in the negative direction.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Subtract the contents of VarA from index register AR0:
MOV AL,@VarA  ; Load AL with contents of VarA
SUB @AR0,AL   ; AR0 = AR0 − AL
              ; Subtract the contents of VarB from VarC:
MOV AH,@VarB  ; Load AH with contents of VarB
SUB @VarC,AH  ; VarC = VarC − AH
```

SUBB ACC,#8bit *Subtract 8-bit Value*

Syntax Options SUBB ACC,#8bit

Opcode `0001 1001 CCCC CCCC`

Objmode X

RPT –

CYC 1

Operands **ACC** – Accumulator register

#8bit – 8-bit immediate constant value

Description Subtract the zero−extended, 8-bit constant from the ACC register:

```
ACC = ACC – 0:8bit;
```

Flags and Modes

Flags and Modes	Description
Z	After the subtraction, the Z flag is set if ACC is zero, else Z is cleared.
N	After the subtraction, the N flag is set if bit 31 of the ACC is 1, N is cleared.
C	If the subtraction generates a borrow, C is cleared; otherwise C is set.
V	If an overflow occurs, V is set; otherwise, V is not affected.
OVC	If(OVM = 0, disabled) then if the operation generates a positive overflow, then the counter is incremented and if the operation generates a negative overflow, then the counter is decremented. If(OVM = 1, enabled) then the counter is not affected by the operation.
OVM	If overflow mode bit is set; then the ACC value will saturate maximum positive (0x7FFFFFFF) or maximum negative (0x80000000) if the operation overflowed.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
Example    ; Decrement contents of 32-bit location VarA:
MOVL ACC,@VarA  ; Load ACC with contents of VarA
SUBB ACC,#1     ; Subtract 1 from ACC
MOVL @VarA,ACC  ; Store result back into VarA
```

SUBB SP,#7bit *Subtract 7-bit Value*

Syntax Options SUBB SP,#7bit

Opcode `1111 1110 1CCC CCCC`

Objmode X

RPT –

CYC 1

Operands

SP – Stack pointer

#7bit – 7-bit immediate constant value

Description Subtract a 7-bit unsigned constant from SP and store the result in SP:

```
SP = SP - 0:7bit;
```

Flags and Modes None

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
FuncA:      ; Function with local variables on
            ; stack.
ADDB SP,#N  ; Reserve N 16-bit words of space for
            ; local variables on stack:
.
.
.
SUBB SP,#N  ; Deallocate reserved stack space.
LRETR       ; Return from function.
```

SUBB XARn,#7bit *Subtract 7-Bit From Auxiliary Register*

Syntax Options SUBB XARn,#7bit

Opcode `1101 1nnn 1CCC CCCC`

Objmode X

RPT –

CYC 1

Operands **XARn** – XAR0 to XAR7, 32-bit auxiliary registers

#7bit – 7−bit immediate constant value

Description Subtract the 7−bit unsigned constant from XARn and store the result in XARn:

```
XARn = XARn − 0:7bit;
```

Flags and Modes None

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
MOVL XAR1,#VarA  ; Initialize XAR1 pointer with address
                 ; of VarA
MOVL XAR2,*XAR1  ; Load XAR2 with contents of VarA
SUBB XAR2,#10h‘  ; XAR2 = VarA − 0x10
```

SUBBL ACC, loc32 *Subtract 32-bit Value Plus Inverse Borrow*

Syntax Options SUBBL ACC, loc32

Opcode

```
0101 0110 0101 0100
0000 0000 LLLL LLLL
```

Objmode 1

RPT –

CYC 1

Operands

loc32 – Addressing mode (see Chapter 5)

ACC – Accumulator register

Description

Subtract from the ACC the 32-bit location pointed to by the “loc32” addressing mode and the logical inversion of the value in the carry flag bit:

```
ACC = ACC - [loc32] - ~C;
```

Flags and Modes

Flags and Modes	Description
Z	After the subtraction, the Z flag is set if the ACC value is zero, else Z is cleared.
N	After the subtraction, the N flag is set if bit 31 of the ACC is 1, else N is cleared.
C	The state of the carry bit before execution is included in the subtraction. If the subtraction generates a borrow, C is cleared; otherwise C is set.
V	If an overflow occurs, V is set; otherwise V is not affected.
OVC	If(OVM = 0, disabled) then if the operation generates a positive overflow, then the counter is incremented and if the operation generates a negative overflow, then the counter is decremented. If(OVM = 1, enabled) then the counter is not affected by the operation.
OVM	If overflow mode bit is set; then the ACC value will saturate maximum positive (0x7FFFFFFF) or maximum negative (0x80000000) if the operation overflowed.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Subtract two 64-bit values (VarA and VarB) and store result
; in VarC:
MOVL ACC,@VarA+0   ; Load ACC with contents of the low
                   ; 32-bits of VarA
SUBUL ACC,@VarB+0  ; Subtract from ACC the contents of
                   ; the low 32-bits of VarB
MOVL @VarC+0,ACC   ; Store low 32-bit result into VarC
MOVL ACC,@VarA+2   ; Load ACC with contents of the high
                   ; 32-bits of VarA
SUBBL ACC,@VarB+2  ; Subtract from ACC the contents of
                   ; the high 32-bits of VarB with borrow
MOVL @VarC+2,ACC   ; Store high 32-bit result into VarC
```

SUBCU ACC,loc16 *Subtract Conditional 16 Bits*

Syntax Options SUBCU ACC,loc16

Opcode `0001 1111 LLLL LLLL`

Objmode X

RPT Y

CYC N+1

Operands

ACC – Accumulator register

loc32 – Addressing mode (see Chapter 5)

Description

Perform 16-bit conditional subtraction, which can be used for unsigned modulus division:

```
temp(32:0) = ACC << 1 - [loc16] << 16
if( temp(32:0) > 0 )
  ACC = temp(31:0) + 1
else
  ACC = ACC << 1
```

To perform 16-bit unsigned modulus division, the AH register is zeroed and the AL register is loaded with the ”Numerator” value prior to executing the SUBCU instruction. The value pointed to be the ”loc16” addressing mode contains the ”Denominator” value. After executing the SUBCU instruction 16 times, the AH register will contain the ”Remainder” and the AL register will contain the ”Quotient” results. To perform signed modulus division, the ”Numerator” and ”Denominator” values must be converted to unsigned quantities, before executing the SUBCU instruction. The final ”Quotient” result must be negated if the ”Numerator” and ”Denominator” values were of different sign else the quotient is left unchanged.

Flags and Modes

Flags and Modes	Description
Z	At the end of the operation, the Z flag is set if the ACC value is zero, else Z is cleared. The calculation of temp(32:0) has no effect on the Z bit.
N	At the end of the operation, the N flag is set if bit 31 of the ACC is 1, else N is cleared. The calculation of temp(32:0) has no effect on the N bit.
C	If the calculation of temp(32:0) generates a borrow, C is cleared; otherwise C is set. Note: The V and OVC flags are not affected by the operation.

Repeat

If this operation is repeated, then the instruction will be executed N+1 times. The state of the Z, N, C flags will reflect the final result. The V flag will be set if an intermediate overflow occurs. The OVC flag will count intermediate overflows, if overflow mode is disabled.

Example 1

```
; Calculate unsigned: Quot16 = Num16Den16, Rem16 = Num16%Den16
  MOVU ACC,@Num16       ; AL = Num16, AH = 0
  RPT #15               ; Repeat operation 16 times
||SUBCU ACC,@Den16      ; Conditional subtract with Den16
  MOV @Rem16,AH         ; Store remainder in Rem16
  MOV @Quot16,AL        ; Store quotient in Quot16
```

Example 2

```
; Calculate signed: Quot16 = Num16Den16, Rem16 = Num16%Den16
  CLRC TC                ; Clear TC flag, used as sign flag
  MOV ACC,@Den16 << 16   ; AH = Den16, AL = 0
  ABSTC ACC              ; Take abs value, TC = sign ^ TC
  MOV T,@AH              ; Temp save Den16 in T register
  MOV ACC,@Num16 << 16   ; AH = Num16, AL = 0
  ABSTC ACC              ; Take abs value, TC = sign ^ TC
  MOVU ACC,@AH           ; AH = 0, AL = Num16
  RPT #15                ; Repeat operation 16 times
||SUBCU ACC,@T           ; Conditional subtract with Den16
  MOV @Rem16,AH          ; Store remainder in Rem16
  MOV ACC,@AL << 16      ; AH = Quot16, AL = 0
  NEGTC ACC              ; Negate if TC = 1
  MOV @Quot16,AH         ; Store quotient in Quot16
```

Example 3

```
; Calculate unsigned: Quot32 = Num32/Den16, Rem16 = Num32%Den16
  MOVU ACC,@Num32+1      ; AH = 0, AL = high 16-bits of Num32
  RPT #15                ; Repeat operation 16 times
||SUBCU ACC,@Den16       ; Conditional subtract with Den16
  MOV @Quot32+1,AL       ; Store high 16-bit in Quot32
  MOV AL,@Num32+0        ; AL = low 16-bits of Num32
  RPT #15                ; Repeat operation 16 times
||SUBCU ACC,@Den16       ; Conditional subtract with Den16
  MOV @Rem16,AH          ; Store remainder in Rem16
  MOV @Quot32+0,AL       ; Store low 16-bit in Quot32
```

Example 4

```
; Calculate signed: Quot32 = Num32/Den16, Rem16 = Num32%Den16
  CLRC TC                ; Clear TC flag, used as sign flag
  MOV ACC,@Den16 << 16   ; AH = Den16, AL = 0
  ABSTC ACC              ; Take abs value, TC = sign ^ TC
  MOV T,@AH              ; Temp save Den16 in T register
  MOVL ACC,@Num32        ; ACC = Num32
  ABSTC ACC              ; Take abs value, TC = sign ^ TC
  MOV P,@ACC             ; P = Num32
  MOVU ACC,@PH           ; AH = 0, AL = high 16-bits of Num32
  RPT #15                ; Repeat operation 16 times
||SUBCU ACC,@T           ; Conditional subtract with Den16
  MOV @Quot32+1,AL       ; Store high 16-bit in Quot32
  MOV AL,@PL             ; AL = low 16-bits of Num32
  RPT #15                ; Repeat operation 16 times
||SUBCU ACC,@T           ; Conditional subtract with Den16
  MOV @Rem16,AH          ; Store remainder in Rem16
  MOV ACC,@AL << 16      ; AH = low 16-bits of Quot32, AL = 0
  NEGTC ACC              ; Negate if TC = 1
  MOV @Quot32+0,AH       ; Store low 16-bit in Quot32
```

SUBCUL ACC,loc32 *Subtract Conditional 32 Bits*

Syntax Options SUBCUL ACC,loc32

Opcode

```
0101 0110 0001 0111
0000 0000 LLLL LLLL
```

Objmode 1

RPT Y

CYC N+1

Operands

ACC – Accumulator register

loc32 – Addressing mode (see Chapter 5)

Description

Perform 32-bit conditional subtraction, which can be used for unsigned modulus division:

```
temp(32:0) = ACC << 1 + P(31) - [loc32];
if( temp(32:0) >= if( temp(32:0) >= 0 )
  ACC = temp(31:0);
  P = (P << 1) + 1;
else
  ACC:P = ACC:P << 1;
```

To perform 32-bit unsigned modulus division, the ACC register is zeroed and the P register is loaded with the ”Numerator” value prior to executing the SUBCUL instruction. The value pointed to be the ”loc32” addressing mode contains the ”Denominator” value. After executing the SUBCUL instruction 32 times, the ACC register will contain the ”Remainder” and the P register will contain the ”Quotient” results. To perform signed modulus division, the ”Numerator” and ”Denominator” values must be converted to unsigned quantities, before executing the SUBCUL instruction. The final ”Quotient” result must be negated if the ”Numerator” and ”Denominator” values were of different sign else the quotient is left unchanged.

Flags and Modes

Flags and Modes	Description
Z	At the end of the operation, the Z flag is set if the ACC value is zero, else Z is cleared. The calculation of temp(32:0) has no effect on the Z bit.
N	At the end of the operation, the N flag is set if bit 31 of the ACC is 1, else N is cleared. The calculation of temp(32:0) has no effect on the N bit.
C	If the calculation of temp(32:0) generates a borrow, C is cleared; otherwise C is set. Note: The V and OVC flags are not affected by the operation.

Repeat

If this operation is repeated, then the instruction will be executed N+1 times. The state of the Z, N, C flags will reflect the final result. The V flag will be set if an intermediate overflow occurs. The OVC flag will count intermediate overflows, if overflow mode is disabled.

Example 1

```
; Calculate unsigned: Quot32 = Num32/Den32, Rem32 = Num32%Den32
  MOVB ACC,#0         ; Zero ACC
  MOVL P,@Num32       ; Load P register with Num32
  RPT #31             ; Repeat operation 32 times
||SUBCUL ACC,@Den32   ; Conditional subtract with Den32
  MOVL @Rem32,ACC     ; Store remainder in Rem32
  MOVL @Quot32,P      ; Store quotient in Quot32
```

Example 2

```
; Calculate signed: Quot32 = Num32/Den32, Rem32 = Num32%Den32
  CLRC TC             ; Clear TC flag, used as sign flag
  MOVL ACC,@Den32     ; Load ACC with contents of Den32
  ABSTC ACC           ; Take absolute value, TC = sign ^ TC
  MOVL XT,@ACC        ; Temp save denominator in XT register
  MOVL ACC,@Num32     ; Load ACC register with Num32
  ABSTC ACC           ; Take abs value, TC = sign ^ TC
  MOVL P,@ACC         ; Load P register with numerator
  OVB ACC,#0          ; Zero ACC
  RPT #31             ; Repeat operation 32 times
||SUBCUL ACC,@XT      ; Conditional subtract with denominator
  MOVL @Rem32,ACC     ; Store remainder in Rem32
  MOVL ACC,@P         ; Load ACC with quotient
  NEGTC ACC           ; Negate ACC if TC=1 (negative result)
  MOVL @Quot32,ACC    ; Store quotient in Quot32
```

Example 3

```
; Calculate unsigned: Quot64 = Num64Den32, Rem32 = Num64%Den32
  MOVB ACC,#0         ; Zero ACC
  MOVL P,@Num64+2     ; Load P with high 32-bits of Num64
  RPT #31             ; Repeat operation 32 times
||SUBCUL ACC,@Den32   ; Conditional subtract with Den32
  MOVL @Quot64+2,P    ; Store high 32 bit quotient in Quot64
  MOVL P,@Num64+0     ; Load P with low 32-bits of Num64
  RPT #31             ; Repeat operation 32 times
||SUBCUL ACC,@Den32   ; Conditional subtract with Den32
  MOVL @Rem32,ACC     ; Store remainder in Rem32
  MOVL @Quot64+0,P    ; Store low 32 bit quotient in Quot64
```

Example 4

```
; Calculate signed: Quot64 = Num364Den32, Rem32 = Num64%Den32
  MOVL ACC,@Num64+2   ; Load ACC:P with 64-bit numerator
  MOVL P,@Num64+0
  TBIT @AH,#15        ; TC = sign of numerator
  SBF $10,NTC         ; Take absolute value of numerator
  NEG64 ACC:P
$10:
  MOVL @XAR3,P        ; Temp save numerator low in XAR3
  MOVL P,@ACC         ; Load P register with numerator high
  MOVL ACC,@Den32     ; Load ACC with contents of Den32
  ABSTC ACC           ; Take absolute value, TC = sign ^ TC
  MOVL XT,@ACC        ; Temp save denominator in XT register
  MOVB ACC,#0         ; Zero ACC
  RPT #31             ; Repeat operation 32 times
||SUBCUL ACC,@XT      ; Conditional subtract with denominator
  MOVL @XAR4,P        ; Store high quotient in XAR4
  MOVL P,@XAR3        ; Load P with low numerator
  RPT #31             ; Repeat operation 32 times
||SUBCUL ACC,@XT      ; Conditional subtract with denominator
  MOVL @Rem32,ACC     ; Store remainder in Rem32
  MOVL ACC,@XAR4      ; Load ACC with high quotient from XAR4
  SBF $20,NTC         ; Take absolute value of quotient
  NEG64 ACC:P
$20:
  MOVL @Quot64+0,P    ; Store low quotient into Quot64
  MOVL @Quot64+2,ACC  ; Store high quotient into Quot64
```

SUBL ACC, loc32 *Subtract 32-bit Value*

Syntax Options SUBL ACC, loc32

Opcode `0000 0011 LLLL LLLL`

Objmode 1

RPT –

CYC 1

Operands

ACC – Accumulator register

loc32 – Addressing mode (see Chapter 5)

Description Subtract the 32-bit location pointed to by the “loc32” addressing mode from the ACC register :

```
ACC = ACC - [loc32];
```

Flags and Modes

Flags and Modes	Description
Z	After the subtraction, the Z flag is set if the ACC value is zero, else Z is cleared.
N	After the subtraction, the N flag is set if bit 31 of the ACC is 1, else N is cleared.
C	If the subtraction generates a borrow, C is cleared; otherwise C is set.
V	If an overflow occurs, V is set; otherwise V is not affected.
OVC	If OVM = 0 (disabled), then if the operation generates a positive overflow, then the counter is incremented and if the operation generates a negative overflow, then the counter is decremented. If OVM = 1 (enabled), then the counter is not affected by the operation.
OVM	If overflow mode bit is set; then the ACC value will saturate maximum positive (0x7FFFFFFF) or maximum negative (0x80000000) if the operation overflowed.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Calculate the 32-bit value: VarC = VarA-VarB
MOVL ACC,@VarA  ; Load ACC with contents of VarA
SUBL ACC,@VarB  ; Subtract from ACC the contents of VarB
MOVL @VarC,ACC  ; Store result into VarC
```

SUBL ACC,P << PM *Subtract 32-bit Value*

Syntax Options SUBL ACC,P << PM

Opcode `0001 0001 1010 1100`

Objmode X

RPT Y

CYC N+1

Note: This instruction is an alias for the ”MOVS T,loc16” operation with “loc16 = @T” addressing mode.

Operands

ACC – Accumulator register

P – Product register

<<PM – Product shift mode

Description Subtract the content of the P register, shifted as specified by the product shift mode (PM), from the content of the ACC register:

```
ACC = ACC - P << PM;
```

Flags and Modes

Flags and Modes	Description
Z	After the subtraction, the Z flag is set if the ACC value is zero, else Z is cleared.
N	After the subtraction, the N flag is set if bit 31 of the ACC is 1, else N is cleared.
C	If the subtraction generates a borrow, C is cleared; otherwise C is set.
V	If an overflow occurs, V is set; otherwise V is not affected.
OVC	If OVM = 0 (disabled) and the operation generates a positive overflow, the counter is incremented; if the operation generates a negative overflow, the counter is decremented. If OVM = 1 (enabled), the counter is not affected by the operation.
OVM	If overflow mode bit is set; then the ACC value will saturate maximum positive (0x7FFFFFFF) or maximum negative (0x80000000) if the operation overflowed.
PM	The value in the PM bits sets the shift mode for the output operation from the product register. If the product shift value is positive (logical left shift operation), then the low bits are zero filled. If the product shift value is negative (arithmetic right shift operation), the upper bits are sign extended.

Repeat If this operation is repeated, then the instruction will be executed N+1 times. The state of the Z, N, C flags will reflect the final result. The V flag will be set if an intermediate overflow occurs. The OVC flag will count intermediate overflows, if overflow mode is disabled.

Example

```
; Calculate: Y = ((B << 11) - (M*X << 4 )) << 10
; Y, M, X, B are Q15 values
SPM -4            ; Set product shift to <<  4
SETC SXM          ; Enable sign extension mode
MOV T,@M          ; T = M
MPY P,T,@X        ; P = M * X
MOV ACC,@B << 11  ; ACC = S:B << 11
SUBL ACC,P << PM  ; ACC = (S:B << 11) - (M*X << 4)
MOVH @Y,ACC << 5  ; Store Q15 result into Y
```

SUBL loc32, ACC *Subtract 32-bit Value*

Syntax Options SUBL loc32, ACC

Opcode

```
0101 0110 0100 0001
0000 0000 LLLL LLLL
```

Objmode 1

RPT –

CYC 1

Operands **loc32** – Addressing mode (see Chapter 5)

ACC – Accumulator register

Description Subtract the content of the ACC register from the location pointed to by the “loc32” addressing mode:

```
[loc32] = [loc32] - ACC;
```

Flags and Modes

Flags and Modes	Description
Z	After the subtraction, the Z flag is set if the ACC value is zero, else Z is cleared.
N	After the subtraction, the N flag is set if bit 31 of the [loc32] is 1, else N is cleared.
C	If the subtraction generates a borrow, C is cleared; otherwise C is set.
V	If an overflow occurs, V is set; otherwise V is not affected.
OVC	If OVM = 0 (disabled) and the operation generates a positive overflow, the counter is incremented and if the operation generates a negative overflow, the counter is decremented. If OVM = 1 (enabled) the counter is not affected by the operation.
OVM	If overflow mode bit is set; then the ACC value will saturate maximum positive (0x7FFFFFFF) or maximum negative (0x80000000) if the operation overflowed.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Decrement the 32-bit value VarA:
MOVB ACC,#1     ; Load ACC with 0x00000001
SUBL @VarA,ACC  ; VarA = VarA - ACC
```

SUBR loc16,AX *Reverse-Subtract Specified Location From AX*

Syntax Options SUBR loc16,AX

Opcode `1110 101A LLLL LLLL`

Objmode 1

RPT –

CYC 1

Operands **loc16** – Addressing mode (see Chapter 5)

AX – Accumulator high (AH) or accumulator low (AL) register

Description Subtract the 16−bit content of the location pointed to by the “loc16” addressing mode from the specified AX register (AH or AL), and store the result in location pointed to by ”loc16”:

```
[loc16] = AX - [loc16]
```

This instruction performs a read-modify-write operation.

Flags and Modes

Flags and Modes	Description
N	After the subtraction, [loc16] is tested for a negative condition. If bit 15 of [loc16] is 1, then the negative flag bit is set; otherwise it is cleared.
Z	After the subtraction, [loc16] is tested for a zero condition. The zero flag bit is set if the operation generates [loc16] = 0, otherwise it is cleared.
C	If the subtraction generates a borrow, C is cleared; otherwise C is set.
V	If an overflow occurs, V is set; otherwise V is not affected. Signed positive overflow occurs if the result crosses the max positive value (0x7FFF) in the positive direction. Signed negative overflow occurs if the result crosses the max negative value (0x8000) in the negative direction.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Subtract index register AR0 from VarA and store in AR0:
MOV AL,@VarA   ; Enable ; Load AL with contents of VarA
               ; sign extension with a left shift of 3

SUBR @AR0,AL   ; AR0 = AL - AR0
               ; Subtract the contents of VarC from VarB and store in VarC:
MOV AH,@VarB   ; Load AH with contents of VarB
SUBR @VarC,AH  ; VarC = AH - VarC
```

SUBRL loc32, ACC *Reverse-Subtract Specified Location From ACC*

Syntax Options

SUBRL loc32, ACC

Opcode

```
0101 0110 0100 1001
0000 0000 LLLL LLLL
```

Objmode

1

RPT

–

CYC

1

Operands

loc32 – Addressing mode (see Chapter 5)

ACC – Accumulator register

Description

Subtract from the ACC register the 32-bit location pointed to by the “loc32” addressing mode and store the result in the location pointed to by “loc32”:

```
[loc32] = ACC - [loc32];
```

Flags and Modes

Flags and Modes	Description
Z	After the subtraction, the Z flag is set if the ACC value is zero, else Z is cleared.
N	After the subtraction, the N flag is set if bit 31 of the ACC is 1, else N is cleared.
C	If the subtraction generates a borrow, C is cleared; otherwise C is set.
V	If an overflow occurs, V is set; otherwise V is not affected.
OVC	If(OVM = 0, disabled) then if the operation generates a positive overflow, then the counter is incremented and if the operation generates a negative overflow, then the counter is decremented. If(OVM = 1, enabled) then the counter is not affected by the operation.
OVM	If overflow mode bit is set; then the ACC value will saturate maximum positive (0x7FFFFFFF) or maximum negative (0x80000000) if the operation overflowed.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Calculate the 32-bit value: VarA = VarB - VarA
MOVL ACC,@VarB    ; Load ACC with contents of VarB
SUBRL @VarA,ACC   ; VarA = ACC - VarA
```

SUBU ACC, loc16 *Subtract Unsigned 16-bit Value*

Syntax Options SUBU ACC, loc16

Opcode `0000 0001 LLLL LLLL`

Objmode X

RPT Y

CYC N+1

Operands **loc16** – Addressing mode (see Chapter 5)

ACC – Accumulator register

Description Subtract the 16-bit contents of the location pointed to by the “loc16” addressing mode from the ACC register. The addressed location is zero extended before the add:

```
ACC = ACC - 0:[loc16];
```

Flags and Modes

Flags and Modes	Description
Z	After the subtraction, the Z flag is set if ACC is zero, else Z is cleared.
N	After the subtraction, the N flag is set if bit 31 of the ACC is 1, else N is cleared.
C	If the subtraction generates a borrow, C is cleared; otherwise C is set.
V	If an overflow occurs, V is set; otherwise V is not affected.
OVC	If OVM = 0 (disabled) and the operation generates a positive overflow, the counter is incremented and if the operation generates a negative overflow, the counter is decremented. If OVM = 1 (enabled), the counter is not affected by the operation.
OVM	If overflow mode bit is set; then the ACC value will saturate maximum positive (0x7FFFFFFF) or maximum negative (0x80000000) if the operation overflowed.

Repeat If this operation is repeated, then the instruction will be executed N+1 times. The state of the Z, N, C flags will reflect the final result. The V flag will be set if an intermediate overflow occurs. The OVC flag will count intermediate overflows, if overflow mode is disabled.

Example

```
; Subtract three 32-bit unsigned variables by 16-bit parts:
MOVU ACC,@VarAlow        ; AH = 0, AL = VarAlow
ADD ACC,@VarAhigh << 16  ; AH = VarAhigh, AL = VarAlow
SUBU ACC,@VarBlo293w     ; ACC = ACC - 0:VarBlow
SUB ACC,@VarBhigh << 16  ; ACC = ACC - VarBhigh  << 16
SBBU ACC,@VarClow        ; ACC = ACC - VarClow - ~Carry
SUB ACC,@VarChigh << 16  ; ACC = ACC - VarChigh  << 16
```

SUBUL ACC, loc32 *Subtract Unsigned 32-bit Value*

Syntax Options SUBUL ACC, loc32

Opcode

```
0101 0110 0101 0101
0000 0000 LLLL LLLL
```

Objmode 1

RPT –

CYC 1

Operands

loc32 – Addressing mode (see Chapter 5)

ACC – Accumulator register

Description

Subtract from the ACC register the 32-bit the location pointed to by the “loc32” addressing mode. The subtraction is treated as an unsigned SUBL operation:

```
ACC = ACC - [loc32];    // unsigned subtraction
```

Note: The difference between a signed and unsigned 32-bit subtract is in the treatment of the overflow counter (OVC). For a signed SUBL, the OVC counter monitors positive/negative overflow. For an unsigned SUBL, the OVC unsigned (OVCU) counter monitors the borrow.

Flags and Modes

Flags and Modes	Description
Z	After the subtraction, the Z flag is set if the ACC value is zero, else Z is cleared.
N	After the subtraction, the N flag is set if bit 31 of the ACC is 1, else N is cleared.
C	If the subtraction generates a borrow, C is cleared; otherwise C is set.
V	If an overflow occurs, V is set; otherwise V is not affected.
OVCU	The overflow counter is decremented whenever a subtraction operation generates an unsigned borrow. The OVM mode does not affect the OVCU counter.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Subtract two 64-bit values (VarA and VarB) and store result
; in VarC:
MOVL ACC,@VarA+0   ; Load ACC with contents of the low
                   ; 32-bits of VarA
SUBUL ACC,@VarB+0  ; Subtract from ACC the contents of
                   ; the low 32-bits of VarB
MOVL @VarC+0,ACC   ; Store low 32-bit result into VarC
MOVL ACC,@VarA+2   ; Load ACC with contents of the high
                   ; 32-bits of VarA
SUBBL ACC,@VarB+2  ; Subtract from ACC the contents of
                   ; the high 32-bits of VarB with borrow
MOVL @VarC+2,ACC   ; Store high 32-bit result into VarC
```

SUBUL P,loc32 *Subtract Unsigned 32-bit Value*

Syntax Options SUBUL P,loc32

Opcode

```
0101 0110 0101 1101
0000 0000 LLLL LLLL
```

Objmode 1

RPT –

CYC 1

Operands

P – Product register

loc32 – Addressing mode (see Chapter 5)

Description

Subtract from the P register the 32-bit content of the location pointed to by the “loc32” addressing mode. The addition is treated as an unsigned SUB operation:

```
P = P - [loc32];    // unsigned subtract
```

Note: The difference between a signed and unsigned 32-bit subtract is in the treatment of the overflow counter (OVC). For a signed SUBL, the OVC counter monitors positive/negative overflow. For an unsigned SUBL, the OVC unsigned (OVCU) counter monitors the borrow.

Flags and Modes

Flags and Modes	Description
Z	After the subtraction, the Z flag is set if the P value is zero, else Z is cleared.
N	After the subtraction, the N flag is set if bit 31 of P is 1, else N is cleared.
C	If the subtraction generates a borrow, C is cleared; otherwise C is set.
V	If a signed overflow occurs, V is set; otherwise V is not affected.
OVCU	The overflow counter is decremented whenever a subtraction operation generates an unsigned borrow. The OVM mode does not affect the OVCU counter.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Subtract 64-bit VarA - VarB and store result in VarC:
MOVL P,@VarA+0     ; Load P with low 32-bits of VarA
MOVL ACC,@VarA+2   ; Load ACC with high 32-bits of VarA
SUBUL P,@VarB+0    ; Sub from P unsigned low 32-bits of VarB
SUBBL ACC,@VarB+2  ; Sub from ACC with borrow high 32-bits of VarB
MOVL @VarC+0,P     ; Store low 32-bit result into VarC
MOVL @VarC+2,ACC   ; Store high 32-bit result into VarC
```

TBIT loc16,#bit *Test Specified Bit*

Syntax Options TBIT loc16,#bit

Opcode `0100 BBBB LLLL LLLL`

Objmode X

RPT –

CYC 1

Operands **#bit** – Immediate constant bit index from 0 to 15

loc16 – Addressing mode (see Chapter 5)

Description Test the specified bit of the data value in the location pointed to by the “loc16” addressing mode:

```
TC = [loc16(bit)];
```

The value specified for the #bit immediate operand directly corresponds to the bit number. For example, if #bit = 0, you will access bit 0 (least significant bit) of the addressed location; if #bit = 15, you will access bit 15 (most significant bit).

Flags and Modes

Flags and Modes	Description
TC	If the bit tested is 1, TC is set; if the bit tested is 0, TC is cleared.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; if( VarA.Bit4 = 1 )
;    VarB.Bit6 = 1;
;
;    else
VarB.Bit6 = 0;
    TBIT @VarA,#4  ; Test bit 4 of VarA contents
    SB $10,NTC     ; Branch if TC = 0
    TSET @VarB,#6  ; Set bit 6 of VarB contents
    SB $20,UNC     ; Branch unconditionally
$10:
TCLR @VarB,#6   ; Clear bit 6 of VarB contents
$20:
```

TBIT loc16,T *Test Bit Specified by Register*

Syntax Options

TBIT loc16,T

Opcode

```
0101 0110 0010 0101
0000 0000 LLLL LLLL
```

Objmode

1

RPT

–

CYC

1

Operands

T – Upper 16 bits of the multiplicand register (XT)

loc16 – Addressing mode (see Chapter 5)

Description

Test the bit specified by the four least significant bits of the T register, T(3:0) = 0…15 of the data value in the location pointed to by the “loc16” addressing mode. Upper bits of the T register are ignored:

```
bit = 15 – T(3:0);
TC = [loc16(bit)];
```

A value of 15 in the T register corresponds to bit 0 (least significant bit). A value of 0 in the T register corresponds to bit 15 (most significant bit). The upper 12 bits of the T register are ignored.

Flags and Modes

Flags and Modes	Description
TC	If the bit tested is 1, TC is set; if the bit tested is 0, TC is cleared.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; if( VarA.VarB = 1 )
;    VarC.Bit6 = 1;
; else
;    VarC.Bit6 = 0;
  MOV T,@VarB    ; Load T with bit value in VarB
  XOR @T,#15     ; Reverse order of bit testing
  TBIT @VarA,T   ; Test bit of VarA selected by VarB
  SB $10,NTC     ; Branch if TC = 0
  TSET @VarB,#6  ; Set bit 6 of VarB contents
  SB $20,UNC     ; Branch unconditionally
$10:
  TCLR @VarB,#6  ; Clear bit 6 of VarB contents
$20:
```

TCLR loc16,#bit *Test and Clear Specified Bit*

Syntax Options

TCLR loc16,#bit

Opcode

```
0101 0110 0000 1001
0000 BBBB LLLL LLLL
```

Objmode

1

RPT

–

CYC

1

Operands

#bit – Immediate constant bit index from 0 to 15

loc16 – Addressing mode (see Chapter 5)

Description

Test the specified bit of the data value in the location pointed to by the “loc16” addressing mode and then clear that same bit to 0:

```
TC = [loc16(bit)];
[loc16(bit)] = 0;
```

The value specified for the #bit immediate operand directly corresponds to the bit number. For example, if #bit = 0, you will access bit 0 (least significant bit) of the addressed location; if #bit = 15, you will access bit 15 (most significant bit).

TCLR performs a read-modify-write operation.

Flags and Modes

Flags and Modes	Description
N	If (loc16 = @AX) and bit 15 (MSB) of @AX is 1, then N flag is set.
Z	If (loc16 = @AX) and @AX gets zeroed out, then Z flag is set.
TC	If the bit tested is 1, TC is set; if the bit tested is 0, TC is cleared.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; if( VarA.Bit4 = 1 )
;    VarB.Bit6 = 1;
;    else
  VarB.Bit6 = 0;
    TBIT @VarA,#4  ; Test bit 4 of VarA contents
    SB $10,NTC     ; Branch if TC = 0
    TSET @VarB,#6  ; Set bit 6 of VarB contents
    SB $20,UNC     ; Branch unconditionally
$10:
  TCLR @VarB,#6  ; Clear bit 6 of VarB contents
$20:
```

TEST ACC *Test for Accumulator Equal to Zero*

Syntax Options TEST ACC

Opcode `1111 1111 0101 1000`

Objmode X

RPT –

CYC 1

Operands **ACC** – Accumulator register

Description Compare the ACC register to zero and set the status flag bits accordingly:

```
Modify flags on (ACC – 0x00000000);
```

Flags and Modes

Flags and Modes	Description
N	If bit 31 of the ACC is 1, N is set; else N is cleared.
Z	If ACC is zero, Z is set; else Z is cleared.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Test contents of ACC and branch if zero:
TEST ACC    ; Modify flags on (ACC – 0x00000000)
SB Zero,EQ  ; Branch if zero
```

TRAP #VectorNumber *Software Trap*

Syntax Options TRAP #VectorNumber

Opcode `0000 0000 001C CCCC`

Objmode X

RPT –

CYC 8

Operands **Vector Number** – CPU interrupt vector 0 to 31

Description The TRAP instruction transfers program control to the interrupt service routine that corresponds to the vector specified in the instruction. It does not affect the interrupt flag register (IFR) or the interrupt enable register (IER), regardless of whether the chosen interrupt has corresponding bits in these registers. The TRAP instruction is not affected by the interrupt global mask bit (INTM) in status register ST1. It also is not affected by the enable bits in the IER or the debug interrupt enable register (DBGIER). Once the TRAP instruction reaches the decode phase of the pipeline, hardware interrupts cannot be serviced until the TRAP instruction is done executing (until the interrupt service routine begins).

The following table indicates which interrupt vector is associated with a chosen value for the VectorNumber operand:

Vector Number	Interrupt Vector	Vector Number	Interrupt Vector
0	RESET	16	RTOSINT
1	INT1	17	Reserved
2	INT2	18	NMI
3	INT3	19	ILLEGAL
4	INT4	20	USER1
5	INT5	21	USER2
6	INT6	22	USER3
7	INT7	23	USER4
8	INT8	24	USER5
9	INT9	25	USER6
10	INT10	26	USER7
11	INT11	27	USER8
12	INT12	28	USER9
13	INT13	29	USER10
14	INT14	30	USER11
15	DLOGINT	31	USER12

Part of the operation involves saving pairs of 16-bit core registers onto the stack pointed to by the SP register. Each pair of registers is saved in a single 32-bit operation. The register forming the low word of the pair is saved first (to an even address); the register forming the high word of the pair is saved next (to the following odd address). For example, the first value saved is the concatenation of the T register and the status register ST0 (T:ST0). ST0 is saved first, then T.

This instruction should not be used with vectors 1−12 when the peripheral interrupt expansion (PIE) is enabled.

Note: The TRAP #0 instruction does not initiate a full reset. It only forces execution of the interrupt service routine that corresponds to the RESET interrupt vector.

```
Flush the pipeline;
```

```
temp = PC + 1;
Fetch specified vector;
SP = SP + 1;
[SP] = T:ST0;
SP = SP + 2;
[SP] = AH:AL;
SP = SP + 2;
[SP] = PH:PL;
SP = SP + 2;
[SP] = AR1:AR0;
SP = SP + 2;
[SP] = DP:ST1;
SP = SP + 2;
[SP] = DBGSTAT:IER;
SP = SP + 2;
[SP] = temp;
SP = SP = 2;
INTM = 0;       // disable INT1−INT14, DLOGINT, RTOSINT
DBGM = 1;       // disable debug events
EALLOW = 0;     // disable access to emulation registers
LOOP = 0;       // clear loop flag
IDLESTAT = 0;   // clear idle flag
PC = fetched vector;
```

Flags and Modes

Flags and Modes	Description
DBGM	Debug events are disabled by setting the DBGM bit.
INTM	Setting the INTM bit disables maskable interrupts.
EALLOW	EALLOW is cleared to disable access to protected registers.
LOOP	The loop flag is cleared.
IDLESTAT	The idle flag is cleared.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

TSET loc16,#16bit *Test and Set Specified Bit*

Syntax Options TSET loc16,#16bit

Opcode

```
0101 0110 0000 1101
0000 BBBB LLLL LLLL
```

Objmode 1

RPT –

CYC 1

Operands **#bit** – Immediate constant bit index from 0 to 15

loc16 – Addressing mode (see Chapter 5)

Description Test the specified bit of the data value in the location pointed to by the “loc16” addressing mode and then set the same bit to 1:

```
TC = [loc16(bit)];
[loc16(bit)] = 1;
```

The value specified for the #bit immediate operand directly corresponds to the bit number. For example, if #bit = 0, you will access bit 0 (least significant bit) of the addressed location; if #bit = 15, you will access bit 15 (most significant bit).

TSET performs a read-modify-write operation.

Flags and Modes

Flags and Modes	Description
N	If (loc16 = = @AX) and bit 15 (MSB) of @AX is 1, then N flag is set.
Z	If (loc16 = = @AX) and @AX gets zeroed out, then Z flag is set.
TC	If the bit tested is 1, TC is set; if the bit tested is 0, TC is cleared.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; if(VarA.Bit4 = 1)
;    VarB.Bit6 = 1;
;    else
VarB.Bit6 = 0;
    TBIT @VarA,#4   ; Test bit 4 of VarA contents
    SB $10,NTC      ; Branch if TC = 0
    TSET @VarB,#6   ; Set bit 6 of VarB contents
    SB $20,UNC      ; Branch unconditionally
$10:
  TCLR @VarB,#6  ; Clear bit 6 of VarB contents
$20:
```

UOUT *(PA),loc16 *Unprotected Output Data to I/O Port*

Syntax Options UOUT *(PA),loc16

Opcode

```
1011 0000 LLLL LLLL
CCCC CCCC CCCC CCCC
```

Objmode 1

RPT Y

CYC N+2

Operands

***(PA)** – Immediate I/O space memory address

loc16 – Addressing mode (see Chapter 5)

Description

Store the 16-bit value from the location pointed to by the “loc16” addressing mode into the I/O space location pointed to by ”*(PA):

```
IOspace[0x000:PA] = loc16;
```

I/O Space is limited to 64K range (0x0000 to 0xFFFF). On the external interface (XINTF), if available on a particular device, the I/O strobe signal (XISn) is toggled during the operation. The I/O address appears on the lower 16 address lines (XA(15:0)) and the upper address lines are zeroed. The data appears on the lower 16 data lines (XD(15:0).

Note: The UOUT operation is not pipeline protected. Therefore, if an IN instruction immediately follows a UOUT instruction, the IN will occur before the UOUT. To be certain of the sequence of operation, use the OUT instruction, which is pipeline protected. I/O space may not be implemented on all C28x devices. See the data sheet for your particular device for details.

Flags and Modes None

Repeat

This instruction is repeatable. If the operation follows a RPT instruction, then it will be executed N+1 times. When repeated, the ”*(PA)” I/O space address is post-incremented by 1 during each repetition.

Example

```
; IORegA address = 0x0300;
; IOREgB address = 0x0301;
; IOREgC address = 0x0302;
; IORegA = 0x0000;
; IORegB = 0x0400;
; IORegC = VarA;
; if(IORegC = 0x2000)
; IORegC = 0x0000;
IORegA .set 0x0300     ; Define IORegA address
IORegB .set 0x0301     ; Define IORegB address
IORegC .set 0x0302     ; Define IORegC address
  MOV @AL,#0           ; AL = 0
  UOUT *(IORegA),@AL   ; IOspace[IORegA] = AL
  MOV @AL,#0x0400      ; AL = 0x0400
  UOUT *(IORegB),@AL   ; IOspace[IORegB] = AL
  OUT *(IORegC),@VarA  ; IOspace[IORegC] = VarA
  IN @AL,*(IORegC)     ; AL = IOspace[IORegC]
  CMP @AL,#0x2000      ; Set flags on (AL – 0x2000)
  SB $10,NEQ           ; Branch if not equal
  MOV @AL,#0           ; AL = 0
  UOUT *(IORegC),@AL   ; IOspace[IORegC] = AL
$10:
```

XB *AL — *C2 xLP Source-Compatible Indirect Branch*

Syntax Options

XB *AL

Opcode

`0101 0110 0001 0100`

Objmode

1

RPT

–

CYC

7

Operands

***AL** – Indirect program-memory addressing using register AL, can only access high 64K of program space range (0x3F0000 to 0x3FFFFF)

Description

Unconditional indirect branch by loading the low 16 bits of PC with the contents of register AL and forcing the upper 6 bits of the PC to 0x3F:

```
PC = 0x3F:AL;
```

Note: This branch instruction can only branch to a location located in the upper 64K range of program space (0x3F0000 to 0x3FFFFF).

Flags and Modes

None

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Branch to subroutines in SwitchTable selected by Switch value.
; This example only works for code located in upper 64K of
; program space:
SwitchTable:               ; Switch address table:
  .word Switch0            ; Switch0 address
  .word Switch1            ; Switch1 address
    .
    .
  MOVL XAR2,#SwitchTable   ; XAR2 = pointer to SwitchTable
  MOVZ AR0,@Switch         ; AR0 = Switch index
  MOV AL,*+XAR2[AR0]       ; AL = SwitchTable[Switch]
  XB *AL                   ; Indirect branch using AL
SwitchReturn:
  .
Switch0:                   ; Subroutine 0:
  .
  .
  XB SwitchReturn,UNC      ; Return: branch

Switch1:                   ; Subroutine 1:
  .
  .
  XB SwitchReturn,UNC      ; Return: branch
```

XB pma,*,ARPn *C2xLP Source-Compatible Branch with ARP Modification*

Syntax Options XB pma,*,ARPn

Opcode

```
0011 1110 0111 0nnn
CCCC CCCC CCCC CCCC
```

Objmode 1

RPT –

CYC 4

Operands

pma – 16-bit immediate program -memory address, can only access high 64K of program space range (0x3F0000 to 0x3FFFFF)

ARPn – 3-bit auxiliary register pointer (ARP0 to ARP7)

Description

Unconditional branch with ARP modification by loading the low 16 bits of PC with the 16-bit immediate value ”pma” and forcing the upper 6 bits of the PC to 0x3F. Also, change the auxiliary register pointer as specified by the “ARPn” operand:

```
PC = 0x3F:pma;
ARP = n;
```

Note: This branch instruction can only branch to a location located in the upper 64K range of program space (0x3F0000 to 0x3FFFFF).

Flags and Modes None

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Branch to SubA and set ARP. Load ACC with pointer pointed to
; by ARP and return to. This example only works for code
; located in upper 64K of program space:
  XB SubA,*,ARP1  ; Branch to SubA with ARP pointing
                  ; to XAR1
SubReturn:
  .
SubAA:            ; Subroutine A:
  MOVL ACC,*      ; Load ACC with contents
                  ; pointed to by XAR(ARP)
XB SubReturn,UNC  ; Return unconditionally
```

XB pma,COND *C2 xLP Source-Compatible Branch*

Syntax Options XB pma,COND

Opcode

```
0101 0110 1101 COND
CCCC CCCC CCCC CCCC
```

Objmode 1

RPT –

CYC 7/4

Operands **pma** – 16-bit immediate program -memory address, can only access high 64K of program space range (0x3F0000 to 0x3FFFFF)

COND – Conditional codes:

COND	Syntax	Description	Flags Tested
0000	NEQ	Not Equal To	Z = 0
0001	EQ	Equal To	Z = 1
0010	GT	Greater Than	Z = 0 AND N = 0
0011	GEQ	Greater Than or Equal To	N = 0
0100	LT	Less Than	N = 1
0101	LEQ	Less Than or Equal To	Z = 1 OR N = 1
0110	HI	Higher	C = 1 AND Z = 0
0111	HIS, C	Higher or Same, Carry Set	C = 1
1000	LO, NC	Lower, Carry Clear	C = 0
1001	LOS	Lower or Same	C = 0 OR Z = 1
1010	NOV	No Overflow	V = 0
1011	OV	Overflow	V = 1
1100	NTC	Test Bit Not Set	TC = 0
1101	TC	Test Bit Set	TC = 1
1110	NBIO	BIO Input Equal To Zero	BIO = 0
1111	UNC	Unconditional	–

Description Conditional branch. If the specified condition is true, then branch by loading the low 16 bits of PC with the 16-bit immediate value ”pma” and forcing the upper 6 bits of the PC to 0x3F.; otherwise continue execution without branching:

```
If (COND = true) PC(15:0) = pma;
If (COND = false) PC(15:0) = PC(15:0) + 2;
PC(21:16) = 0x3F;
```

Note: If (COND = true) then the instruction takes 7 cycles.

If (COND = false) then the instruction takes 4 cycles.

Flags and Modes

Flags and Modes	Description
V	If the V flag is tested by the condition, then V is cleared.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Branch to subroutines in SwitchTable selected by Switch value.
; This example only works for code located in upper 64K of
; program space:
SwitchTable:            ; Switch address table:
  .word Switch0         ; Switch0 address
  .word Switch1         ; Switch1 address
  .
  .
  MOVL XAR2,#Switch-    ; XAR2 = pointer to SwitchTable
Table
  MOVZ AR0,@Switch      ; AR0 = Switch index
  MOV AL,*+XAR2[AR0]    ; AL = SwitchTable[Switch]
  XB *AL                ; Indirect branch using AL
SwitchReturn:
.

  Switch0:              ; Subroutine 0:
  .
  .

  XB SwitchReturn,UNC   ; Return: branch

Switch1:                ; Subroutine 1:
  .
  .

  XB SwitchReturn,UNC   ; Return: branch
```

XBANZ pma,*ind{,ARPn} *C2 x LP Source-Compatible Branch If ARn Is Not Zero*

Syntax Options

Syntax Options	Opcode	Objmode	RPT	CYC

XBANZ pma,*	`0101 0110 0000 1100`
	`CCCC CCCC CCCC CCCC`
		1	–	4/2

XBANZ pma,*++	`0101 0110 0000 1010`
	`CCCC CCCC CCCC CCCC`
		1	–	4/2

XBANZ pma,*—	`0101 0110 0000 1011`
	`CCCC CCCC CCCC CCCC`
		1	–	4/2

XBANZ pma,*0++	`0101 0110 0000 1110`
	`CCCC CCCC CCCC CCCC`
		1	–	4/2

XBANZ pma,*0—	`0101 0110 0000 1111`
	`CCCC CCCC CCCC CCCC`
		1	–	4/2

XBANZ pma,*,ARPn	`0011 1110 0011 0nnn`
	`CCCC CCCC CCCC CCCC`
		1	–	4/2

XBANZ pma,*++,ARPn	`0011 1110 0011 1nnn`
	`CCCC CCCC CCCC CCCC`
		1	–	4/2

XBANZ pma,*—',ARPn	`0011 1110 0100 0nnn`
	`CCCC CCCC CCCC CCCC`
		1	–	4/2

XBANZ pma,*0++,ARPn	`0011 1110 0100 1nnn`
	`CCCC CCCC CCCC CCCC`
		1	–	4/2

XBANZ pma,*0—,ARPn	`0011 1110 0101 0nnn`
	`CCCC CCCC CCCC CCCC`
		1	–	4/2

Operands

pma – 16-bit immediate program -memory address, can only access high 64K of program space range (0x3F0000 to 0x3FFFFF)

ARPn – 3-bit auxiliary register pointer (ARP0 to ARP7)

Description

If the lower 16 bits of the auxiliary register pointed to by the current auxiliary register pointer (ARP) is not equal to 0, then a branch is taken by loading the lower 16 bits of the PC with the 16-bit immediate “pma” value and forcing the upper 6 bits of the PC to 0x3F. Then, the current auxiliary register, pointed to by the ARP, is modified as specified by the indirect mode. Then,, if indicated, the ARP pointer value is changed to point a new auxiliary register:

```
if(AR[ARP] != 0)
  PC = 0x3F:pma
if(*++ indirect mode) XAR[ARP] = XAR[ARP] + 1;
if(*–– indirect mode) XAR[ARP] = XAR[ARP] - 1;
if(*0++ indirect mode) XAR[ARP] = XAR[ARP] + AR0;
if(*0–– indirect mode) XAR[ARP] = XAR[ARP] – AR0;
if(ARPn specified) ARPn = n;
```

This instruction can only transfer program control to a location located in the upper 64K range of program space (0x3F0000 to 0x3FFFFF). The cycle times for this operation are:

If branch is taken, then the instruction takes 4 cycles

If branch is not taken, then the instruction takes 2 cycles

Flags and Modes

None

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Copy the contents of Array1 to Array2:
; int32 Array1[N];
; int32 Array2[N];
; for(i=0; i < N; i++)
;    Array2[i] = Array1[i];
; This example only works for code located in upper 64K of
; program space:
  MOVL XAR2,#Array1   ; XAR2 = pointer to Array1
  MOVL XAR3,#Array2   ; XAR3 = pointer to Array2
  MOV @AR0,#(N−1)     ; Repeat loop N times
  NOP *,ARP2          ; Point to XAR2
  SETC AMODE          ; Full C2XLP address mode compatible
Loop:
  MOVL ACC,*++,ARP3   ; ACC = Array1[i], point to XAR3
  MOVL *++,ACC,ARP0   ; Array2[i] = ACC, point to XAR0
  BANZ Loop,*−−,ARP2  ; Loop if AR[ARP] != 0, AR[ARP]−−,
                      ; point to XAR2
```

XCALL *AL *C2 x LP Source-Compatible Function Call*

Syntax Options XCALL *AL

Opcode `0101 0110 0011 0100`

Objmode 1

RPT –

CYC 7

Operands ***AL** – Indirect program-memory addressing using register AL, can only access high 64K of program space range (0x3F0000 to 0x3FFFFF)

Description Indirect call with destination address in AL. The lower 16 bits of the current PC address are saved onto the software stack. Then, the low 16 bits of PC is loaded with the contents of register AL and the upper 6 bits of the PC are loaded with 0x3F:

```
temp(21:0) = PC + 1;
[SP] = temp(15:0);
SP = SP + 1;
C = 0x3F:AL;
```

Note: This instruction can only transfer program control to a location located in the upper 64K range of program space (0x3F0000 to 0x3FFFFF). To return from a call made by XCALL, the XRETC instruction must be used.

Flags and Modes None

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Call function in FuncTable selected by FuncIndex value.
; This example only works for code located in upper 64K of
; program space:
FuncTable:                ; Function address table:
  .word FuncA             ; FuncA address
  .word FuncB             ; FuncB address
  .
  .
  MOVL XAR2,#FuncTable    ; XAR2 = pointer to FuncTable
  MOVZ AR0,@FuncIndex     ; AR0 = FuncTable index
  MOV AL,*+XAR2[AR0]      ; AL = Table[FuncIndex]
  XCALL *AL               ; Indirect call using AL
  .
  .
FuncA:                    ; Function A:
  .
  .
  XRETC UNC               ; Return unconditionally

FuncB:                    ; Function B:
  .
  .
  XRETC UNC               ; Return unconditionally
```

XCALL pma,*,ARPn *C2 x LP Source-Compatible Function Call*

Syntax Options

XCALL pma,*,ARPn

Opcode

```
0011 1110 0110 1nnn
CCCC CCCC CCCC CCCC
```

Objmode

1

RPT

–

CYC

4

Operands

pma – 16-bit immediate program -memory address, can only access high 64K of program space range (0x3F0000 to 0x3FFFFF)

ARPn – 3-bit auxiliary register pointer (ARP0 to ARP7)

Description

Unconditional call with ARP modification. The lower 16 bits of the return address are pushed onto the software stack. Then, the lower 16 bits of the PC are loaded with the 16-bit immediate ”pma” value and the upper 6 bits of the PC are forced to 0x3F. Then, the 3-bit ARP pointer will be set to the ”ARPn” field value:

```
temp(21:0) = PC + 1;
[SP] = temp(15:0);
SP = SP + 1;
PC = 0x3F:pma;
ARP = n;
```

Note: This instruction can only transfer program control to a location located in the upper 64K range of program space (0x3F0000 to 0x3FFFFF). To return from a call made by XCALL, the XRETC instruction must be used.

Flags and Modes

None

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Call FuncA and set ARP. Load ACC with pointer pointed to by ARP.
; This example only works for code located in upper 64K of program
; space:
  XCALL FuncA,*,ARP1  ; Call FuncA with ARP pointing to XAR1
. Fun;A:              ; Function A:
  MOVL ACC,*          ; Load ACC with contents pointed to
                      ; by XAR (ARP)
  XRETC UNC           ; Return unconditionally
```

XCALL pma,COND *C2xLP Source-Compatible Function Call*

Syntax Options XCALL pma,COND

Opcode

```
0101 0110 1110 COND
CCCC CCCC CCCC CCCC
```

Objmode 1

RPT –

CYC 7/4

Operands **pma** – 16-bit immediate program -memory address, can only access high 64K of program space range (0x3F0000 to 0x3FFFFF)

COND – Conditional codes:

COND	Syntax	Description	Flags Tested
0000	NEQ	Not Equal To	Z = 0
0001	EQ	Equal To	Z = 1
0010	GT	Greater Than	Z = 0 AND N = 0
0011	GEQ	Greater Than or Equal To	N = 0
0100	LT	Less Than	N = 1
0101	LEQ	Less Than or Equal To	Z = 1 OR N = 1
0110	HI	Higher	C = 1 AND Z = 0
0111	HIS, C	Higher or Same, Carry Set	C = 1
1000	LO, NC	Lower, Carry Clear	C = 0
1001	LOS	Lower or Same	C = 0 OR Z = 1
1010	NOV	No Overflow	V = 0
1011	OV	Overflow	V = 1
1100	NTC	Test Bit Not Set	TC = 0
1101	TC	Test Bit Set	TC = 1
1110	NBIO	BIO Input Equal To Zero	BIO = 0
1111	UNC	Unconditional	–

Description Conditional call. If the specified condition is true, then the low 16 bits of the return address is pushed onto the software stack and the low 16 bits of the PC are loaded with the 16-bit immediate ”pma” value and the upper 6 bits of the PC are forced to 0x3F; otherwise continue execution with instruction following the XCALL operation:

```
if(COND = true)
  {
  temp(21:0) = PC + 2;
  [SP] = temp(15:0);
  SP = SP + 1;
  PC    = 0x3F:pma;
  }
else
  PC = PC + 2;
```

Note: This instruction can only transfer program control to a location located in the upper 64K range of program space (0x3F0000 to 0x3FFFFF). To return from a call made by XCALL, the XRETC instruction must be used. The cycle times for this operation are:

If (COND = true) then the instruction takes 7 cycles.

If (COND = false) then the instruction takes 4 cycles.

Flags and Modes

Flags and Modes	Description
V	If the V flag is tested by the condition, then V is cleared.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Call FuncA if VarA does not equal zero. This example only
; works for code located in upper 64K of program space:
  MOV AL,@VarA     ; Load AL with VarA
  XCALL FuncA,NEQ  ; Call FuncA if not equal to zero
  .
  .
FuncA:             ; Function A:
  .
  .
  XRETC UNC        ; Return unconditionally
```

XMAC P,loc16,*(pma) *C2xLP Source-compatible Multiply and Accumulate*

Syntax Options XMAC P,loc16,*(pma)

Opcode

```
1000 0100 LLLL LLLL
CCCC CCCC CCCC CCCC
```

Objmode 1

RPT Y

CYC N+2

Operands

P – Product register

loc16 – Addressing mode (see Chapter 5)

***(pma)** – Immediate program memory address, access high 64K range of program space only (0x3F0000 to 0x3FFFFF)

Description

Add the previous product (stored in the P register), shifted as specified by the product shift mode (PM), to the ACC register. Next, load the T register with the content of the location pointed to by the “loc16” addressing mode. Last, multiply the signed 16-bit content of the T register by the signed 16-bit content of the addressed program memory location and store the 32-bit result in the P register:

```
ACC = ACC + P << PM;
T = [loc16];
P = signed T * signed Prog[0x3F:pma];
```

The C28x forces the upper 6 bits of the program memory address, specified by the ”*(pma)” addressing mode, to 0x3F when using this form of the MAC instruction. This limits the program memory address to the high 64K of program address space (0x3F0000 to 0x3FFFFF). On the C28x devices, memory blocks are mapped to both program and data space (unified memory), hence the ”*(pma)” addressing mode can be used to access data space variables that fall within its address range.

Flags and Modes

Flags and Modes	Description
Z	After the addition, the Z flag is set if the ACC value is zero, else Z is cleared.
N	After the addition, the N flag is set if bit 31 of the ACC is 1, else N is cleared.
C	If the addition generates a carry, C is set; otherwise C is cleared.
V	If an overflow occurs, V is set; otherwise V is not affected.
OVC	If overflow mode is disabled; and if the operation generates a positive overflow, then the counter is incremented. If overflow mode is disabled; and if the operation generates a negative overflow, then the counter is decremented.
OVM	If overflow mode bit is set; then the ACC value will saturate maximum positive (0x7FFFFFFF) or maximum negative (0x80000000) if the operation overflowed.
PM	The value in the PM bits sets the shift mode for the output operation from the product register. If the product shift value is positive (logical left shift operation), then the low bits are zero filled. If the product shift value is negative (arithmetic right shift operation), the upper bits are sign extended.

Repeat

This instruction is repeatable. If the operation follows a RPT instruction, then it will be executed N+1 times. The state of the Z, N, C and OVC flags will reflect the final result. The V flag will be set if an intermediate overflow occurs. When repeated, the program-memory address is incremented by 1 during each repetition.

Example

```
; Calculate sum of product using 16-bit multiply:
; int16 X[N]    ; Data information
; int16 C[N]    ; Coefficient information, located in high 64K
; sum = 0;
; for(i=0; i < N; i++)
;    sum = sum + (X[i] * C[i]) >> 5;
  MOVL XAR2,#X         ; XAR2 = pointer to X
  SPM -5               ; Set product shift to ">> 5"
  ZAPA                 ; Zero ACC, P, OVC
  RPT #N−1             ; Repeat next instruction N times
||XMAC P,*XAR2++,*(C)  ; ACC = ACC + P >> 5,
                       ; P = *XAR2++ * *C++
  ADDL ACC, P << PM    ; Perform final accumulate
  MOVL @sum,ACC        ; Store final result into sum
```

XMACD P,loc16,*(pma) *C2xLP Source-Compatible Multiply and Accumulate With Data Move*

Syntax Options

XMACD P,loc16,*(pma)

Opcode

```
1010 0100 LLLL LLLL
CCCC CCCC CCCC CCCC
```

Objmode

1

RPT

Y

CYC

N+2

Operands

P – Product register

loc16 – Addressing mode (see Chapter 5)

Note: For this operation, register-addressing modes cannot be used. The modes are: @ARn, @AH, @AL, @PH, @PL, @SP, @T. An illegal instruction trap will be generated.

***(pma)** – Immediate program memory address, access high 64K range of program space only (0x3F0000 to 0x3FFFFF)

Description

The XMACD instruction functions in the same manner as the XMAC, with the addition of a data move. Add the previous product (stored in the P register), shifted as specified by the product shift mode (PM), to the ACC register. Next, load the T register with the content of the location pointed to by the “loc16” addressing mode. Then, multiply the signed 16-bit content of the T register by the signed 16-bit content of the addressed program memory location and store the 32-bit result in the P register. Last, store the content in the T register onto the next highest memory address pointed to by ”loc16” addressing mode:

```
ACC = ACC + P << PM;
T = [loc16];
P = signed T * signed Prog[0x3F:pma];
[loc16 + 1] = T;
```

The C28x forces the upper 6 bits of the program memory address, specified by the ”*(pma)” addressing mode, to 0x3F when using this form of the MAC instruction. This limits the program memory address to the high 64K of program address space (0x3F0000 to 0x3FFFFF). On the C28x devices, memory blocks are mapped to both program and data space (unified memory), therefore, the “(pma)” addressing mode can be used to access data-space variables that fall within its address range.

Flags and Modes

Flags and Modes	Description
Z	After the addition, the Z flag is set if the ACC value is zero, else Z is cleared.
N	After the addition, the N flag is set if bit 31 of the ACC is 1, else N is cleared.
C	If the addition generates a carry, C is set; otherwise C is cleared.
V	If an overflow occurs, V is set; otherwise V is not affected.
OVC	If overflow mode is disabled and if the operation generates a positive overflow, the counter is incremented. If overflow mode is disabled and if the operation generates a negative overflow, the counter is decremented.
OVM	If overflow mode bit is set, the ACC value will saturate maximum positive (0x7FFFFFFF) or maximum negative (0x80000000) if the operation overflowed.
PM	The value in the PM bits sets the shift mode for the output operation from the product register. If the product shift value is positive (logical left shift operation), then the low bits are zero filled. If the product shift value is negative (arithmetic right shift operation), the upper bits are sign extended.

Repeat

This instruction is repeatable. If the operation follows a RPT instruction, then it will be executed N+1 times. The state of the Z, N, C and OVC flags will reflect the final result. The V flag will be set if an intermediate overflow occurs. When repeated, the program-memory address is incremented by 1 during each repetition.

Example

```
; Calculate FIR filter using 16-bit multiply:
; int16 X[N]    ; Data information
; int16 C[N]    ; Coefficient information, located in high 64K
; sum = X[N–1] * C[0];
; for(i=1; i < N; i++)
;    {
;    sum = sum + (X[N–1–i] * C[i])  >>  5;
;    X[N–i] = X[N–1–i];
;    }
; X[1] = X[0];
  MOVL XAR2,#X+N          ; XAR2 = point to end of X array
  SPM -5                  ; Set product shift to  ">> 5"
  ZAPA                    ; Zero ACC, P, OVC
  XMAC P,*––XAR2,*(C)     ; ACC = 0, P = X[N–1] * C[0]
  RPT #N–2                ; Repeat next instruction N–1  times
||XMACD P,*––XAR2,*(C+1)  ; ACC = ACC + P  >> 5,
                          ; P = X[N–1–i] * C[i],
                          ; i++
  MOV *+XAR2[2],T         ; X[1] = X[0]
  ADDL ACC, P << PM       ; Perform final accumulate
  MOVL @sum,ACC           ; Store final result into sum
```

XOR ACC,loc16 *Bitwise Exclusive OR*

Syntax Options XOR ACC,loc16

Opcode `1011 0111 LLLL LLLL`

Objmode 1

RPT Y

CYC N+1

Operands **ACC** – Accumulator register

loc16 – Addressing mode (see Chapter 5)

Description Perform a bitwise XOR operation on the ACC register with the zero-extended content of the location pointed to by the “loc16” address mode. The result is stored in the ACC register:

```
ACC = ACC XOR 0:[loc16];
```

Flags and Modes

Flags and Modes	Description
N	The load to ACC is tested for a negative condition. If bit 31 of ACC is 1, then the negative flag bit is set; otherwise it is cleared.
Z	The load to ACC is tested for a zero condition. The zero flag bit is set if the operation generates ACC = 0; otherwise it is cleared

Repeat This operation is repeatable. If the operation follows a RPT instruction, then the XOR instruction will be executed N+1 times. The state of the Z and N flags will reflect the final result.

Example

```
; Calculate the 32-bit value: VarA = VarA XOR 0:VarB
MOVL ACC,@VarA  ; Load ACC with contents of VarA
XOR ACC,@VarB   ; XOR ACC with contents of 0:VarB
MOVL @VarA,ACC  ; Store result in VarA
```

XOR ACC,#16bit << #0..16 *Bitwise Exclusive OR*

Syntax Options

Syntax Options	Opcode	Objmode	RPT	CYC

XOR ACC,#16bit << #0..15	`0011 1110 0010 SHFT`
	`CCCC CCCC CCCC CCCC`
		1	–	1

XOR ACC,#16bit << #16	`0101 0110 0100 1110`
	`CCCC CCCC CCCC CCCC`
		1	–	1

Operands

ACC – Accumulator register

#16bit – 16-bit immediate constant value

#0..16 – Shift value (default is "<<#0" if no value specified)

Description

Perform a bitwise XOR operation on the ACC register with the given 16-bit unsigned constant value left shifted as specified. The value is zero extended and lower order bits are zero filled before the XOR operation. The result is stored in the ACC register:

```
ACC = ACC XOR (0:16bit << shift value);
```

Flags and Modes

Flags and Modes	Description
N	The load to ACC is tested for a negative condition. If bit 31 of ACC is 1, then the negative flag bit is set; otherwise it is cleared.
B	The load to ACC is tested for a zero condition. The zero flag bit is set if the operation generates ACC = 0; otherwise it is cleared.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Calculate the 32-bit value: VarA = VarA XOR 0x08000000
MOVL ACC,@VarA         ; Load ACC with contents of VarA
XOR ACC,#0x8000 << 12  ; XOR ACC with 0x08000000
MOVL @VarA,ACC         ; Store result in VarA
```

XOR AX,loc16 *Bitwise Exclusive OR*

Syntax Options XOR AX,loc16

Opcode `0111 000A LLLL LLLL`

Objmode X

RPT –

CYC 1

Operands

AX – Accumulator high (AH) or accumulator low (AL) register

loc16 – Addressing mode (see Chapter 5)

Description

Perform a bitwise exclusive OR operation on the specified AX register (AH or AL) and the contents of the location pointed to by the “loc16” addressing mode. The result is stored in the specified AX register:

```
AX = AX XOR [loc16];
```

Flags and Modes

Flags and Modes	Description
N	The load to AX is tested for a negative condition. If bit 15 of AX is 1, then the negative flag bit is set; otherwise it is cleared.
Z	The load to AX is tested for a zero condition. The zero flag bit is set if the operation generates AX = 0, otherwise it is cleared.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; XOR the contents of VarA and VarB and store in VarC:
MOV AL,@VarA  ; Load AL with contents of VarA
XOR AL,@VarB  ; XOR AL with contents of VarB
MOV @VarC,AL  ; Store result in VarC
```

XOR loc16, AX *Bitwise Exclusive OR*

Syntax Options XOR loc16, AX

Opcode

```
1111 001A LLLL LLLL
```

Objmode X

RPT –

CYC 1

Operands

AX – Accumulator high (AH) or accumulator low (AL) register

loc16 – Addressing mode (see Chapter 5)

Description

Perform a bitwise exclusive OR operation on the 16-bit contents of location pointed to by the “loc16” addressing mode and the specified AX register (AH or AL). The result is stored in the location pointed to by “loc16”:

```
[loc16] = [loc16] XOR AX;
```

This instruction performs a read-modify-write operation.

Flags and Modes

Flags and Modes	Description
N	The load to [loc16] is tested for a negative condition. If bit 15 of [loc16] is 1, then the negative flag bit is set; otherwise it is cleared.
Z	The load to [loc16] is tested for a zero condition. The zero flag bit is set if the operation generates [loc16] = 0, otherwise it is cleared.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; XOR the contents of VarA with VarB and store in VarB:
MOV AL,@VarA  ; Load AL with contents of VarA
XOR @VarB,AL  ; VarB = VarB XOR AL
```

XOR loc16,#16bit *Bitwise Exclusive OR*

Syntax Options XOR loc16,#16bit

Opcode

```
0001 1100 LLLL LLLL
CCCC CCCC CCCC CCCC
```

Objmode X

RPT –

CYC 1

Operands **loc16** – Addressing mode (see Chapter 5)

#16bit – 16-bit immediate constant value

Description Perform a bitwise XOR operation on the content of the location pointed to by the “loc16” addressing mode and the 16-bit immediate constant value. The result is stored in the location pointed to by “loc16”:

```
[loc16] = [loc16] XOR 16bit;
```

Smart Encoding:

If loc16 = AH or AL and #16bit is an 8-bit number, then the assembler will encode this instruction as XO”RB AX,#8bt. To override this encoding, use the XORW AX,#16bit instruction alias.

Flags and Modes

Flags and Modes	Description
N	After the operation if bit 15 of [loc16] 1, set N; otherwise, clear N.
Z	After the operation if [loc16] is zero, set Z; otherwise, clear Z.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Toggle Bits 2 and 14 of VarA:
; VarA = VarA XOR #(1 << 2 | 1 << 14)
XOR @VarA,#(1 << 2 | 1 << 14)  ; Toggle bits 2 and 11 of VarA
```

XORB AX, #8bit *Bitwise Exclusive OR 8-bit Value*

Syntax Options XORB AX, #8bit

Opcode `1111 000A CCCC CCCC`

Objmode X

RPT –

CYC 1

Operands

AX – Accumulator high (AH) or accumulator low (AL) register

#8bit – 8-bit immediate constant value

Description

Perform a bitwise exclusive OR operation on the specified AX register and the 8-bit unsigned immediate constant zero extended. The result is stored in the AX register:

```
AX = AX XOR 0x00:8bit;
```

Flags and Modes

Flags and Modes	Description
N	The load to AX is tested for a negative condition. If bit 15 of AX is 1, then the negative flag bit is set; otherwise it is cleared.
Z	The load to AX is tested for a zero condition. The zero flag bit is set if the operation generates [loc16] = 0, otherwise it is cleared.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Toggle bit 7 of VarA and store result in VarB:
MOV AL,@VarA    ; Load AL with contents of VarA
XORB AL,#0x80   ; XOR contents of AL with 0x0080
MOV @VarB,AL    ; Store result in VarB
```

XPREAD loc16, *(pma) *C2xLP Source-Compatible Program Read*

Syntax Options XPREAD loc16, *(pma)

Opcode

```
1010 1100 MMMM MMMM
LLLL LLLL LLLL LLLL
```

Objmode 1

RPT Y

CYC N+2

Operands

loc16 – Addressing mode (see Chapter 5)

***(pma)** – Immediate program-memory address, can only access high 64K of program space range (0x3F0000 to 0x3FFFFF)

Description

Load the 16-bit data-memory location pointed to by the “loc16” addressing mode with the 16-bit content of the program-memory location pointed to by ”*(pma)” addressing mode:

```
[loc16] = Prog[0x3F:pma];
```

The C28x forces the upper 6 bits of the program memory address, specified by the ”*(pma)” addressing mode, to 0x3F when using this form of the XPREAD instruction. This limits the program memory address to the high 64K of program address space (0x3F0000 to 0x3FFFFF). On the C28x devices, memory blocks are mapped to both program and data space (unified memory), hence the ”*(pma)” addressing mode can be used to access data space variables that fall within its address range.

Flags and Modes

Flags and Modes	Description
N	If (loc16 = @AX) and bit 15 of AX is 1, then N is set; otherwise N is cleared.
Z	If (loc16 = @AX) and the value of AX is zero, then Z is set; otherwise Z is cleared.

Repeat

This instruction is repeatable. If the operation follows a RPT instruction, then it will be executed N+1 times. When repeated, the ”*(pma)” program-memory address is copied to an internal shadow register and the address is post-incremented by 1 during each repetition.

Example

```
; Copy the contents of Array1 to Array2:
; int16 Array1[N];    // Located in high 64K of program space
; int16 Array2[N];    // Located in data space
; for(i=0; i < N; i++)
;    Array2[i] = Array1[i];
  MOVL XAR2,#Array2         ; XAR2 = pointer to Array2
  RPT #(N-1)                ; Repeat next instruction N times
||XPREAD *XAR2++,*(Array1)  ; Array2[i] = Array1[i],
                            ; i++
```

XPREAD loc16, *AL *C2xLP Source-Compatible Program Read*

Syntax Options XPREAD loc16, *AL

Opcode

```
0101 0110 0011 1100
0000 0000 LLLL LLLL
```

Objmode 1

RPT Y

CYC N+4

Operands

loc16 – Addressing mode (see Chapter 5)

***AL** – Indirect program-memory addressing using register AL, can only access high 64K of program space range (0x3F0000 to 0x3FFFFF)

Description

Load the 16-bit data-memory location pointed to by the “loc16” addressing mode with the 16-bit content of the program-memory location pointed to by ”*AL” addressing mode:

```
[loc16] = Prog[0x3F:AL];
```

The C28x forces the upper 6 bits of the program memory address, specified by the ”*AL” addressing mode, to 0x3F when using this form of the XPREAD instruction. This limits the program memory address to the high 64K of program address space (0x3F0000 to 0x3FFFFF). On the C28x devices, memory blocks are mapped to both program and data space (unified memory), hence the ”*AL” addressing mode can be used to access data space variables that fall within its address range.

Flags and Modes

Flags and Modes	Description
N	If (loc16 = @AX) and bit 15 of AX is 1, then N is set; otherwise N is cleared.
Z	If (loc16 = @AX) and the value of AX is zero, then Z is set; otherwise Z is cleared.

Repeat

This instruction is repeatable. If the operation follows a RPT instruction, then it will be executed N+1 times. When repeated, the ”*AL” program-memory address is copied to an internal shadow register and the address is post-incremented by 1 during each repetition.

Example

```
; Copy the contents of Array1 to Array2:
; int16 Array1[N];    // Located in high 64K of program space
; int16 Array2[N];    // Located in data space
; for(i=0; i < N; i++)
;    Array2[i] = Array1[i];
  MOV @AL,#Array1     ; AL = pointer to Array1
  MOVL XAR2,#Array2   ; XAR2 = pointer to Array2
  RPT #(N−1)          ; Repeat next instruction N times
||XPREAD *XAR2++,*AL  ; Array2[i] = Array1[i],
                      ; i++
```

XPWRITE *A,loc16 *C2xLP Source-Compatible Program Write*

Syntax Options

XPWRITE *A,loc16

Opcode

```
0101 0110 0011 1101
0000 0000 LLLL LLLL
```

Objmode

1

RPT

Y

CYC

N+4

Operands

loc16 – Addressing mode (see Chapter 5)

***AL** – Indirect program-memory addressing using register AL, can only access high 64K of program space range (0x3F0000 to 0x3FFFFF)

Description

Load the 16-bit program-memory location pointed to by ”*AL” addressing mode with the 16-bit content of the location pointed to by the “loc16” addressing mode:

```
Prog[0x3F:AL] = [loc16];
```

The C28x forces the upper 6 bits of the program memory address, specified by the ”*AL” addressing mode, to 0x3F when using this form of the XPWRITE instruction. This limits the program memory address to the high 64K of program address space (0x3F0000 to 0x3FFFFF). On the C28x devices, memory blocks are mapped to both program and data space (unified memory), hence the ”*AL” addressing mode can be used to access data space variables that fall within its address range.

Flags and Modes

None

Repeat

This instruction is repeatable. If the operation follows a RPT instruction, then it will be executed N+1 times. When repeated, the ”*AL” program-memory address is copied to an internal shadow register and the address is post-incremented by 1 during each repetition.

Example

```
; Copy the contents of Array1 to Array2:
; int16 Array1[N];    // Located in data space
; int16 Array2[N];    // Located in high 64K of program space
; for(i=0; i < N; i++)
;    Array2[i] = Array1[i];
  MOVL XAR2,#Array1    ; XAR2 = pointer to Array1
  MOV @AL,#Array2      ; AL = pointer to Array2
  RPT #(N–1)           ; Repeat next instruction N times
||XPWRITE *AL,*XAR2++  ; Array2[i] = Array1[i],
                       ; i++
```

XRET *C2xLP Source-Compatible Return*

Syntax Options XRET

Opcode `0101 0110 1111 1111`

Objmode 1

RPT –

CYC 7

Note: XRET is an alias for RETC unconditional.

Operands None

Description Return conditionally. If the specified condition is true, a 16-bit value is popped from the stack and stored into the low 16 bits of the PC while the upper 6 bits of the PC are forced to 0x3F; Otherwise, execution continues with the instruction following the XRETC operation:

```
if(COND = true)
  SP = SP – 1;
  PC = 0x3F:[SP];
```

Note: This instruction can transfer program control only to a location located in the upper 64K range of program space (0x3F0000 to 0x3FFFFF). To return from a call made by XCALL, the XRET instruction must be used.

Flags and Modes

Flags and Modes	Description
V	If the V flag is tested by the condition, then V is cleared.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Return from FuncA if VarA does not equal zero, else set VarB
; to zero and return. This example only works for code located
; in upper 64K of program space:
  XCALL FuncA      ; Call FuncA
  .
FuncA:             ; Function A:
  .
  .
  .
  .
  MOV AL,@VarA     ; Load AL with contents of VarA
  XRET NEQ         ; Return if VarA does not equal 0
  MOV @VarA,#0     ; Store 0 into VarB
  XRETC UNC        ; Return unconditionally
```

XRETC COND *C2xLP Source-Compatible Conditional Return*

Syntax Options XRETC COND

Opcode `0101 0110 1111 COND`

Objmode 1

RPT –

CYC 4/7

Operands **COND** – Conditional codes:

COND	Syntax	Description	Flags Tested
0000	NEQ	Not Equal To	Z = 0
0001	EQ	Equal To	Z = 1
0010	GT	Greater Than	Z = 0 AND N = 0
0011	GEQ	Greater Than or Equal To	N = 0
0100	LT	Less Than	N = 1
0101	LEQ	Less Than or Equal To	Z = 1 OR N = 1
0110	HI	Higher	C = 1 AND Z = 0
0111	HIS, C	Higher or Same, Carry Set	C = 1
1000	LO, NC	Lower, Carry Clear	C = 0
1001	LOS	Lower or Same	C = 0 OR Z = 1
1010	NOV	No Overflow	V = 0
1011	OV	Overflow	V = 1
1100	NTC	Test Bit Not Set	TC = 0
1101	TC	Test Bit Set	TC = 1
1110	NBIO	BIO Input Equal To Zero	BIO = 0
1111	UNC	Unconditional	–

Description Return conditionally. If the specified condition is true, a 16-bit value is popped from the stack and stored into the low 16 bits of the PC while the upper 6 bits of the PC are forced to 0x3F; Otherwise, execution continues with the instruction following the XRETC operation:

```
if(COND = true)
  {
  SP = SP – 1;
  PC = 0x3F:[SP];
  }
else
  PC = PC + 1;
```

Note: This instruction can only transfer program control to a location located in the upper 64K range of program space (0x3F0000 to 0x3FFFFF). To return from a call made by XCALL, the XRETC instruction must be used. The cycle times for this operation are:

If (COND = true) then the instruction takes 7 cycles.

If (COND = false) then the instruction takes 4 cycles.

Flags and Modes

Flags and Modes	Description
V	If the V flag is tested by the condition, then V is cleared.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Return from FuncA if VarA does not equal zero, else set VarB
; to zero and return. This example only works for code located
; in upper 64K of program space:
  XCALL FuncA   ; Call FuncA
  .
FuncA:          ; Function A:
  .
  .
  .
  .
  MOV AL,@VarA  ; Load AL with contents of VarA
  XRETC NEQ     ; Return if VarA does not equal 0
  MOV @VarA,#0  ; Store 0 into VarB
  XRETC UNC     ; Return unconditionally
```

ZALR ACC,loc16 *Zero AL and Load AH With Rounding*

Syntax Options ZALR ACC,loc16

Opcode

```
0101 0110 0001 0011
0000 0000 LLLL LLLL
```

Objmode 1

RPT –

CYC 1

Operands

ACC – Accumulator register

loc16 – Addressing mode (see Chapter 5)

Description

Load low accumulator (AL) with the value 0x8000 and load high accumulator (AH) with the 16-bit contents pointed to by the ”loc16” addressing mode.

```
AH = [loc16];
AL = 0x8000;
```

Flags and Modes

Flags and Modes	Description
N	The load to ACC is tested for a negative condition. If bit 31 of ACC is 1, then the negative flag bit is set; otherwise it is cleared.
Z	The load to ACC is tested for a zero condition. The zero flag bit is set if the operation generates ACC = 0; otherwise it is cleared.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Calculate: Y = round(M*X << 1 + B << 16)
; Y, M, X, B are all Q15 numbers
SPM +1             ; Set product shift mode to << 1
MOV T,@M           ; T = M (Q15)
MPY P,T,@X         ; P = M * X (Q30)
ZALR ACC,@B        ; ACC = B << 16 + 0x8000 (Q31)
ADDL ACC,P << PM   ; Add P to ACC with shift (Q31)
MOV @Y,AH          ; Store AH into Y (Q15)
```

ZAP OVC *Clear Overflow Counter*

Syntax Options ZAP OVC

Opcode `0101 0110 0101 1100`

Objmode 1

RPT –

CYC 1

Operands **OVC** – Overflow counter bits in Status Register 0 (ST0)

Description Clear the overflow counter (OVC) bits in Status Register 0 (ST0).

Flags and Modes

Flags and Modes	Description
OVC	The 6-bit overflow counter bits (OVC) are cleared.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Calculate: VarD = sat(VarA + VarB + VarC)
ZAP OVC         ; Zero overflow counter
MOVL ACC,@VarA  ; ACC = VarA
ADDL ACC,@VarB  ; ACC = ACC + VarB
ADDL ACC,@VarC  ; ACC = ACC + VarC
SAT ACC         ; Saturate if OVC != 0
MOVL @VarD,ACC  ; Store saturated result into VarD
```

ZAPA *Zero Accumulator and P Register*

Syntax Options ZAPA

Opcode `0101 0110 0011 0011`

Objmode 1

RPT –

CYC 1

Operands None

Description Zero the ACC and P registers as well as the overflow counter (OVC):

```
ACC = 0;
P = 0;
OVC = 0;
```

Flags and Modes

Flags and Modes	Description
N	The N bit is set.
Z	The Z bit is cleared.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Example

```
; Calculate sum of product using 32-bit multiply and retain
; high result:
; int32 X[N]; // Data information
; int32 C[N]; // Coefficient information (located in low 4M)
; int32 sum = 0;
; for(i=0; i < N; i++)
;    sum = sum + ((X[i] * C[i]) >> 32) >> 5;
  MOVL XAR2,#X              ; XAR2 = pointer to X
  MOVL XAR7,#C              ; XAR7 = pointer to C
  SPM -5                    ; Set  product shift to ">> 5"
  ZAPA                      ; Zero ACC, P, OVC
  RPT #(N−1)                ; Repeat next instruction N times
||QMACL P,*XAR2++,*XAR7++   ; ACC = ACC + P >> 5,
                            ; P = (X[i] * C[i]) >> 32
                            ; i++
  ADDL ACC, P << PM         ; Perform final accumulate
  MOVL @sum,ACC             ; Store final result into sum
```

Emulation Features

The CPU in the C28x contains hardware extensions for advanced emulation features that can assist you in the development of your application system (software and hardware). This chapter describes the emulation features that are available on all C28x devices using only the JTAG port (with TI extensions).

For more information about instructions shown in examples in this chapter, see Chapter 6, *Assembly Language Instructions*.

Topic **Page**

7.1 Overview of Emulation Features

The CPU’s hardware extensions for advanced emulation features provide simple, inexpensive, and speed-independent access to the CPU for sophisticated debugging and economical system development, without requiring the costly cabling and access to processor pins required by traditional emulator systems. It provides this access without intruding on system resources.

The on-chip development interface provides:

- Minimally intrusive access to internal and external memory
- Minimally intrusive access to CPU and peripheral registers
- Control of the execution of background code while continuing to service time-critical interrupts
  - Break on a software breakpoint instruction (instruction replacement)
  - Break on a specified program or data access without requiring instruction replacement (accomplished using bus comparators)
  - Break on external attention request from debug host or additional hardware
  - Break after the execution of a single instruction (single-stepping)
  - Control over the execution of code from device power up
- Nonintrusive determination of device status
  - Detection of a system reset, emulation/test-logic reset, or power-down occurrence
  - Detection of the absence of a system clock or memory-ready signal
  - Determination of whether global interrupts are enabled
  - Determination of why debug accesses might be blocked
- Rapid transfer of memory contents between the device and a host (data logging)
- A cycle counter for performance benchmarking. With a 100-MHz cycle clock, the counter can benchmark actions up to 3 hours in duration.

7.2 Debug Interface

The target-level TI debug interface uses the five standard IEEE 1149.1 (JTAG) signals ($\overline{\text{TRST}}$, TCK, TMS, TDI, and TDO) and the two TI extensions (EMU0 and EMU1). Table D-3 shows the 14-pin JTAG header that is used to interface the target to a scan controller, and Table 7-1 defines the pins.

As shown in Table 7-1, the header requires more than the five JTAG signals and the TI extensions. It also requires a test clock return signal (TCK_RET), the target supply (V_{CC}) and ground (GND). TCK_RET is a test clock out of the scan controller and into the target system. The target system uses TCK_RET if it does not supply its own test clock (in which case TCK would simply not be used). In many target systems, TCK_RET is simply connected to TCK and used as the test clock.

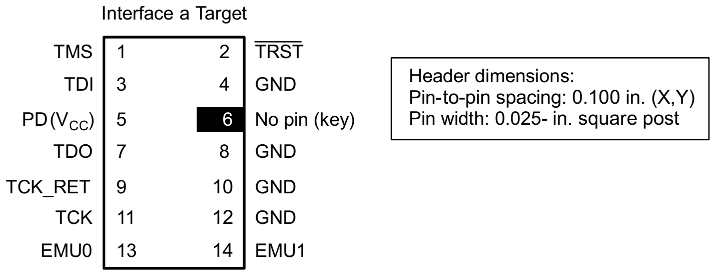


Figure 7-1. JTAG Header to Interface a Target to the Scan Controller

Table 7-1. 14-Pin Header Signal Descriptions

Signal	Description	Emulator State[1]	Target State[1]
EMU0	Emulation pin 0	I	I/O
EMU1	Emulation pin 1	I	I/O
GND	Ground		
PD (V_{CC})	Presence detect. Indicates that the emulation cable is connected and that the target is powered up. PD should be tied to V_{CC} in the target system.	I	O
TCK	Test clock. TCK is a clock source from the emulation cable pod. This signal can be used to drive the system test clock.	O	I
TCK_RET	Test clock return. Test clock input to the emulator. Can be a buffered or unbuffered version of TCK.	I	O
TDI	Test data input O I	O	I
TDO	Test data output	I	O
TMS	Test mode select	O	I
$\overline{\text{TRST}}$[2]	Test reset	O	I

[1] I = input; O = output

[2] Do not use pullup resistors on $\overline{\text{TRST}}$: it has an internal pulldown device. In a low-noise environment, $\overline{\text{TRST}}$ can be left floating. In a high-noise environment, an additional pulldown resistor may be needed. (The size of this resistor should be based on electrical current considerations.)

The state of the $\overline{\text{TRST}}$, EMU0, and EMU1 signals at device power up determine the operating mode of the device. The operating mode takes effect as soon as the device has sufficient power to operate. Should the $\overline{\text{TRST}}$ signal rise, the EMU0 and EMU1 signals are sampled on its rising edge and the\at operating mode is latched. Some of these modes are reserved for test purposes, but those that can be of use in a target system are detailed in Table 7-2. A target system is not required to support any mode other than normal mode.

Table 7-2. Selecting Device Operating Modes By Using $\overline{\text{TRST}}$, EMU0, and EMU1

$\overline{\text{TRST}}$	EMU1	EMU0	Device Operating Mode	JTAG Cable Active?
Low	Low	Low	*Slave mode*. Disables the CPU and memory portions of the C28x. Another processor treats the C28x as a peripheral.	No
Low	Low	High	Reserved for testing	No
Low	High	Low	*Wait-in-reset mode*. Prolongs the device's reset until released by external means. This allows a C28x to power up in reset, provided external hardware holds EMU0 low only while power-up reset is active.	Yes
Low	High	High	*Normal mode with emulation disabled*. This is the setting that should be used on target systems when a scan controller (such as the XDS510) is not attached. TRST will be pulled down and EMU1 and EMU0 pulled up within the C28x; this is the default mode.	No
High	Low or High	Low or High	*Normal mode with emulation enabled*. This is the setting to use on target systems when a scan controller is attached (the scan controller will control TRST). TRST should not be high during device power-up.	Yes

7.3 Debug Terminology

The following definitions will help you to understand the information in the rest of this chapter:

- **Background code.** The body of code that can be halted during debugging because it is not time-critical.
- **Foreground code.** The code of time-critical interrupt service routines, which are executed even when background code is halted.
- **Debug-halt state.** The state in which the device does not execute back-ground code.
- Time-critical interrupt. An interrupt that must be serviced even when background code is halted. For example, a time-critical interrupt might service a motor controller or a high-speed timer.
- **Debug event.** An action, such as the decoding of a software breakpoint instruction, the occurrence of an analysis breakpoint/watchpoint, or a request from a host processor that can result in special debug behavior, such as halting the device or pulsing one of the signals EMU0 or EMU1.
- **Break event.** A debug event that causes the device to enter the debug-halt state.

7.4 Execution Control Modes

The C28x supports two debug execution control modes:

- Stop mode
- Real-time mode

Stop mode provides complete control of program execution, allowing for the disabling of all interrupts. Real-time mode allows time-critical interrupt service routines to be performed while execution of other code is halted. Both execution modes can suspend program execution at break events, such as occurrences of software breakpoint instructions or specified program-space or data-space accesses.

7.4.1 Stop Mode

Stop mode causes break events, such as software breakpoints and analysis watchpoints, to suspend program execution at the next interrupt boundary (which is usually identical to the next instruction boundary). When execution is suspended, all interrupts (including $\overline{\text{NMI}}$ and $\overline{\text{RS}}$) are ignored until the CPU receives a directive to run code again. In stop mode, the CPU can operate in the following execution states:

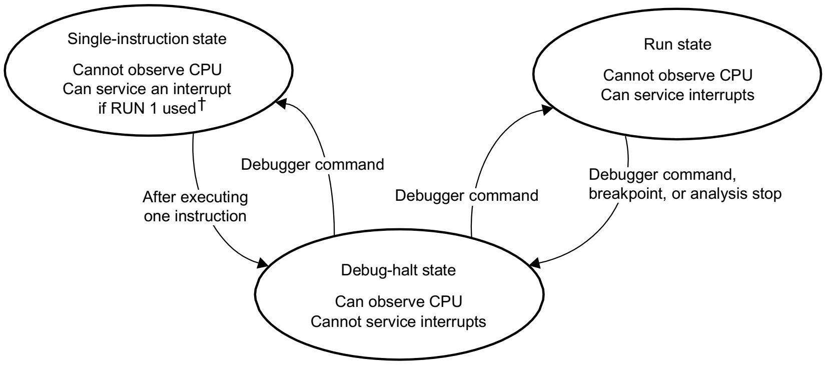
- **Debug-halt state.** This state is entered through a break event, such as the decoding of a software breakpoint instruction or the occurrence of an analysis breakpoint/watchpoint. This state can also be entered by a request from the host processor. In the stop mode debug-halt state, the CPU is halted. You can place the device into one of the other two states by giving the appropriate command to the debugger.

  The CPU cannot service any interrupts, including $\overline{\text{NMI}}$ and $\overline{\text{RS}}$ (reset). When multiple instances of the same interrupt occurs without the first instance being serviced, the later instances are lost.
- **Single-instruction state.** This state is entered when you tell the debugger to execute a single instruction by using a RUN 1 command or a STEP 1 command. The CPU executes the single instruction pointed to by the PC and then returns to the debug-halt state (it executes from one interrupt boundary to the next). The CPU is only in the single-instruction state until that single instruction is done.

  If an interrupt occurs in this state, the command used to enter this state determines whether that interrupt can be serviced. If a RUN 1 command was used, the CPU can service the interrupt. If a STEP 1 command was used, the CPU cannot, even if the interrupt is $\overline{\text{NMI}}$ or $\overline{\text{RS}}$.
- **Run state.** This state is entered when you use a run command from the debugger interface. The CPU executes instructions until a debugger command or a debug event returns the CPU to the debug-halt state.

  The CPU can service all interrupts in this state. When an interrupt occurs simultaneously with a debug event, the debug event has priority; however, if interrupt processing began before the debug event occurred, the debug event cannot be processed until the interrupt service routine begins.

Figure 7-2 illustrates the relationship among the three states. Notice that the C28x cannot pass directly between the single-instruction and run states. Notice also that the CPU can be observed only in the debug-halt state. In practical terms, this means the contents of CPU registers and memory are not updated in the debugger display in the single-instruction state or the run state. Maskable interrupts occurring in any state are latched in the interrupt flag register (IFR).



[†] If you use a RUN 1 command to execute a single instruction, an interrupt can be serviced in the single-instruction state. If you use a STEP 1 command for the same purpose, an interrupt cannot be serviced.

Figure 7-2. Stop Mode Execution States

7.4.2 Real-Time Mode

Real-time mode provides for the debugging of code that interacts with interrupts that must not be disabled. Real-time mode allows you to suspend back-ground code at break events while continuing to execute time-critical interrupt service routines (also referred to as foreground code). In real-time mode, the CPU can operate in the following execution states:

- **Debug-halt state.** This state is entered through a break event such as the decoding of a software breakpoint instruction or the occurrence of an analysis breakpoint/watchpoint. This state can also be enter by a request from the host processor. You can place the device into one of the other two states by giving the appropriate command to the debugger.

  In this state, only time-critical interrupts can be serviced. No other code can be executed. Maskable interrupts are considered time-critical if they are enabled in the debug interrupt enable register (DBGIER). If they are also enabled in the interrupt enable register (IER), they are serviced. The interrupt global mask bit (INTM) is ignored. $\overline{\text{NMI}}$ and $\overline{\text{RS}}$ are also considered time-critical, and are always serviced once requested. It is possible for multiple interrupts to occur and be serviced while the device is in the debug-halt state.

  Suspending execution adds only one cycle to interrupt latency. When the C28x returns from a time-critical ISR, it reenters the debug-halt state.

  If a CPU reset occurs (initiated by $\overline{\text{RS}}$), the device runs the corresponding interrupt service routine until that routine clears the debug enable mask bit (DBGM) in status register ST1. When a reset occurs, DBGM is set, disabling debug events. To reenable debug events, the interrupt service routine must clear DBGM. Only then will the outstanding emulation-suspend condition be recognized.

  NOTE: Should a time-critical interrupt occur in real-time mode at the precise moment that the debugger receives a RUN command, the time-critical interrupt will be taken and serviced in its entirety before the CPU changes states.

- **Single-instruction state.** This state is entered when you tell the de-bugger to execute a single instruction by using a RUN 1 command or a STEP 1 command. The CPU executes the single instruction pointed to by the PC and then returns to the debug-halt state (it executes from one interrupt boundary to the next).
- If an interrupt occurs in this state, the command used to enter this state deter-mines whether that interrupt can be serviced. If a RUN 1 command was used, the CPU can service the interrupt. If a STEP 1 command was used, the CPU cannot, even if the interrupt is $\overline{\text{NMI}}$ or $\overline{\text{RS}}$. In real-time mode, if the DBGM bit is 1 (debug events are disabled), a RUN 1 or STEP 1 command forces continuous execution

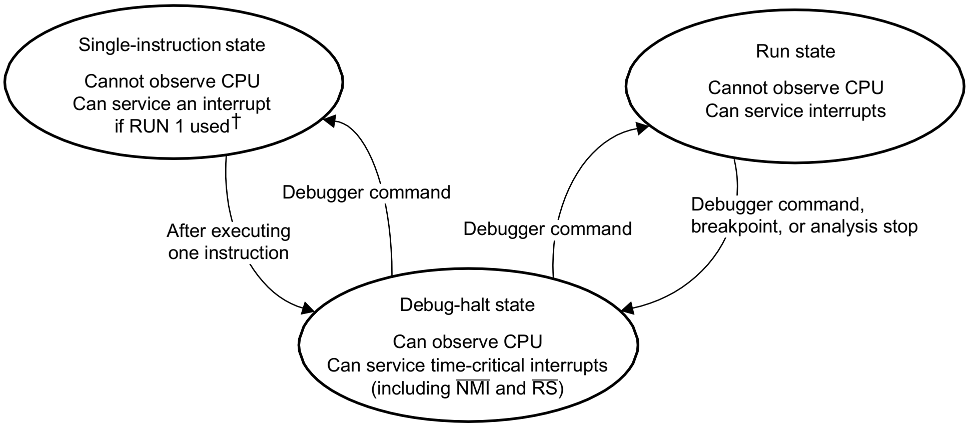
of instructions until DBGM is cleared.

NOTE: If you single-step an instruction in real-time emulation mode and that instruction sets DBGM, the CPU continues to execute instructions until DBGM is cleared. If you want to single-step through a non-time-critical interrupt service routine (ISR), you must initiate a CLRC DBGM instruction at the beginning of the ISR. Once you clear DBGM, you can single-step or place breakpoints.

- **Run state.** This state is entered when you use a run command from the debugger interface. The CPU executes instructions until a debugger command or a debug event returns the CPU to the debug-halt state.

  The CPU can service all interrupts in this state. When an interrupt occurs simultaneously with a debug event, the debug event has priority; however, if interrupt processing began before the debug event occurred, the debug event cannot be processed until the interrupt service routine begins.

Figure 7-3 illustrates the relationship among the three states. Notice that the C28x cannot pass directly between the single-instruction and run states. Notice also that the CPU can be observed in the debug-halt state and in the run state. In the single-instruction state, the contents of CPU registers and memory are not updated in the debugger display. In the debug-halt and run states, register and memory values are updated unless DBGM = 1. Maskable interrupts occurring in any state are latched in the interrupt flag register (IFR).



† If you use a RUN 1 command to execute a single instruction, an interrupt can be serviced in the single-instruction state. If you use a STEP 1 command for the same purpose, an interrupt cannot be serviced.

Figure 7-3. Real-time Mode Execution States

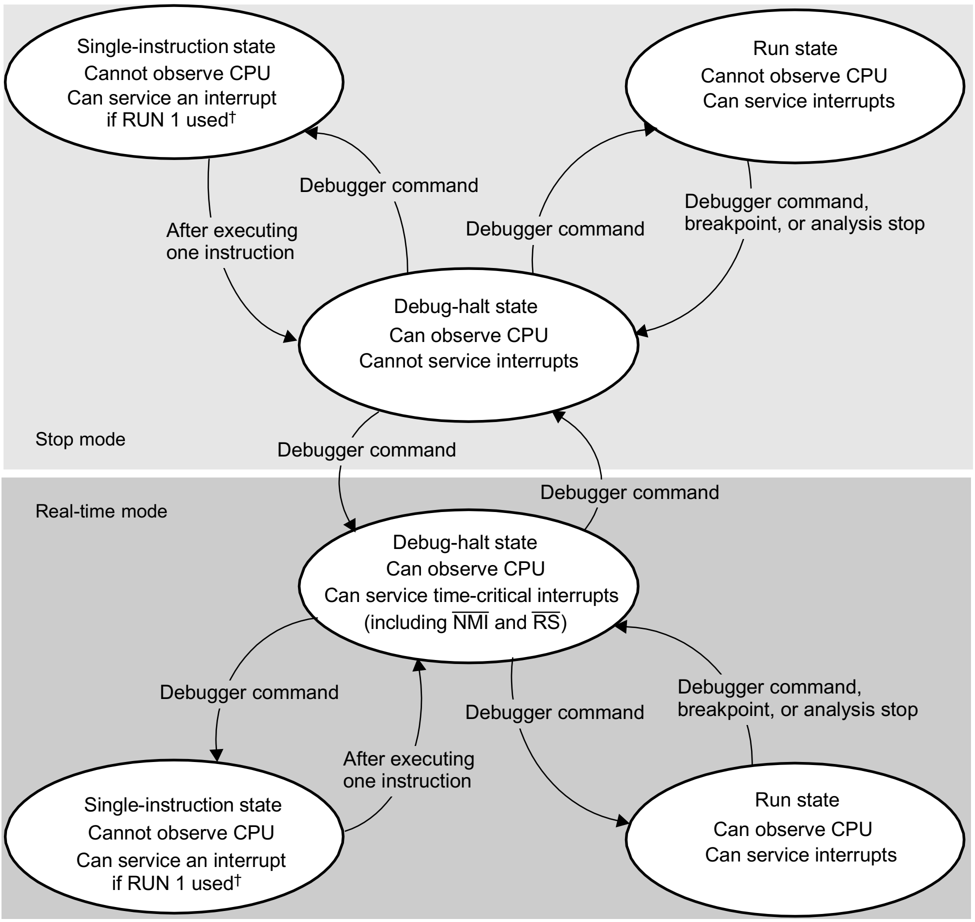
Caution about breakpoints within time-critical interrupt service routines

Do not use breakpoints within time-critical interrupt service routines. They will cause the device to enter the debug-halt state, just as if the breakpoint were located in normal code. Once in the debug-halt state, the CPU services requests for $\overline{\text{RS}}$, $\overline{\text{NMI}}$, and those interrupts enabled in the DBGIER and the IER.

After approving a maskable interrupt, the CPU disables the interrupt in the IER. This prevents subsequent occurrences of the interrupt from being serviced until the IER is restored by a return from interrupt (IRET) instruction or until the interrupt is deliberately re-enabled in the interrupt service routine (ISR). Do not reenable that interrupt's IER bit while using breakpoints within the ISR. If you do so and the interrupt is triggered again, the CPU performs a new context save and restarts the interrupt service routine.

7.4.3 Summary of Stop Mode and Real-Time Mode

Figure 7-4 is a graphical summary of the differences between the execution states of stop mode and real-time mode. Table 7-3 is a summary of how interrupts are handled in each of the states of stop mode and real-time mode.



† If you use a RUN 1 debugger command to execute a single instruction, an interrupt can be serviced in the single-instruction state. If you use a STEP 1 debugger command for the same purpose, an interrupt cannot be serviced.

Figure 7-4. Stop Mode Versus Real-Time Mode

Table 7-3. Interrupt Handling Information By Mode and State

Mode	State	If This Interrupt Occurs ...	The Interrupt Is ...
Stop	Debug-halt	RS	Not serviced
		NMI	Not serviced
		Maskable interrupt	Latched in IFR but not serviced
	Single-instruction	RS	If running: Serviced If stepping: Not serviced
		NMI	If running: Serviced If stepping: Not serviced
		Maskable interrupt	If running: Serviced If stepping: Latched in IFR but not serviced
	Run	RS	Serviced
		NMI	Serviced
		Maskable interrupt	Serviced

Table 7-3. Interrupt Handling Information By Mode and State (continued)

Mode	State	If This Interrupt Occurs ...	The Interrupt Is ...
Real-time	Debug-halt	RS	Serviced
		NMI	Serviced
		Maskable interrupt	If time-critical: Serviced. If not time-critical: Latched in IFR but not serviced
	Single-instruction	RS	If running: Serviced If stepping: Not serviced
		NMI	If running: Serviced If stepping: Not serviced
		Maskable interrupt	If running: Serviced If stepping: Latched in IFR but not serviced
	Run	RS	Serviced
		NMI	Serviced
		Maskable interrupt	Serviced

NOTE: Unless you are using a real-time operating system, do not enable the real-time operating system interrupt (RTOSINT). RTOSINT is completely disabled when bit 15 in the IER is 0 and bit 15 in the DBGIER is 0.

7.5 Aborting Interrupts With the ABORTI Instruction

Generally, a program uses the IRET instruction to return from an interrupt. The IRET instruction restores all the values that were saved to the stack during the automatic context save. In restoring status register ST1 and the debug status register (DBGSTAT), IRET restores the debug context that was present before the interrupt.

In some target applications, you might have interrupts that must not be returned from by the IRET instruction. Not using IRET can cause a problem for the emulation logic, because the emulation logic assumes the original debug context will be restored. The abort interrupt (ABORTI) instruction is provided as a means to indicate that the debug context will not be restored and the debug logic needs to be reset to its default state. As part of its operation, the ABORTI instruction:

- Sets the DBGM bit in ST1. This disables debug events.
- Modifies select bits in DBGSTAT. The effect is a resetting of the debug context. If the CPU was in the debug-halt state before the interrupt occurred, the CPU does not halt when the interrupt is aborted. The CPU automatically switches to the run state. If you want to abort an interrupt, but keep the CPU halted, insert a breakpoint after the ABORTI instruction.

The ABORTI instruction does not modify the DBGIER, the IER, the INTM bit, or any analysis registers (for example, registers used for breakpoints, watch-points, and data logging).

7.6 DT-DMA Mechanism

The debug-and-test direct memory access (DT-DMA) mechanism provides access to memory, CPU registers, and memory-mapped registers (such as emulation registers and peripheral registers) without direct CPU intervention. DT-DMAs intrude on CPU time; however, you can block them by setting the debug enable mask bit (DBGM) in ST1.

Because the DT-DMA mechanism uses the same memory-access mechanism as the CPU, any read or write access that the CPU can perform in a single operation can be done by a DT-DMA. The DT-DMA mechanism presents an address (and data, in the case of a write) to the CPU, which performs the operation during an unused bus cycle (referred to as a hole). Once the CPU has obtained the desired data, it is presented back to the DT-DMA mechanism. The DT-DMA mechanism can operate in the following modes:

- **Nonpreemptive mode.** The DT-DMA mechanism waits for a hole on the desired memory buses. During the hole, the DT-DMA mechanism uses them to perform its read or write operation. These holes occur naturally while the CPU is waiting for newly fetched instructions, such as during a branch.
- **Preemptive mode.** In preemptive mode, the DT-DMA mechanism forces the creation of a hole and performs the access.

DT-DMAs can be polite or rude.

- **Polite accesses.** Polite DT-DMAs require that DBGM = 0.
- **Rude accesses.** Rude DT-DMAs ignore DBGM.

Figure 7-5 summarizes the process for handling a request from the DT-DMA mechanism.

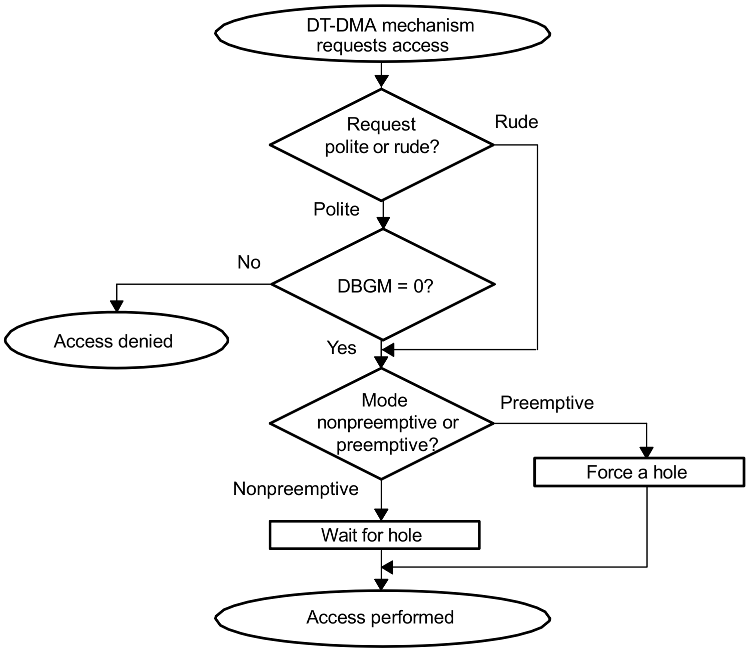


Figure 7-5. Process for Handling a DT-DMA Request

Some key concepts of the DT-DMA mechanism are:

- Even if DBGM = 0, when the mechanism is in nonpreemptive mode, it must wait for a hole. This minimizes the intrusiveness of the debug access on a system.
- Real-time-mode accesses are typically polite (although there may be reasons, such as error recovery, to perform rude accesses in real-time mode). If the DBGM bit is permanently set to 1 due to a coding bug but you need to regain debug control, use rude accesses, which ignore the state of DBGM.
- In stop mode, DBGM is ignored, and the DT-DMA mode is set to preemptive. This ensures that you can gain visibility to and control of your system if an otherwise unrecoverable error occurs (for example, if ST1 is changed to an undesired value due to stack corruption).
- The DT-DMA mechanism does not cause a program-flow discontinuity. No interrupt-like save/restore is performed. When a preemptive DT-DMA forces a hole, no program address counters increment during that cycle.
- A DT-DMA request awakens the device from the idle state (initiated by the IDLE instruction). However, unlike returning from an interrupt, the CPU returns to the idle state upon completion of the DT-DMA.

NOTE: The information shown on the debugger screen is gathered at different times from the target; therefore, it does not represent a snapshot of the target state, but rather a composite. It also takes the host time to process and display the data. The data does not correspond to the current target state, but rather, the target state as of a few milliseconds ago.

7.7 Analysis Breakpoints, Watchpoints, and Counter(s)

All C28x devices include two analysis units AU1 and AU2. Analysis Unit 1 (AU1) counts events or monitors address buses. Analysis Unit 2 (AU2) monitors address and data buses. You can configure these two analysis units as analysis breakpoints or watchpoints. In addition, AU1 can be configured as a benchmark counter or event counter.

This section describes thee types of analysis features: analysis breakpoints, watchpoints, and counters. Typical analysis unit configurations are presented in Section 7.7.4. Data logging is described in Section 7.8.

7.7.1 Analysis Breakpoints

An analysis breakpoint is sometimes called a hardware breakpoint, because it acts like a software breakpoint instruction (in this case, the ESTOP0 instruction) but does not require a modification to the application software. An analysis breakpoint triggers a debug event when an instruction at a breakpoint address would have entered the decode 2 phase of the pipeline; this halts the CPU before the instruction is executed. A bus comparator watches the program address bus, comparing its contents against a reference address and a bit mask value.

Consider the following example. If a hardware breakpoint is set at T0, the CPU stops after returning from the T1 subroutine, with the instruction counter (IC) pointing to T0.

```
    NOP
    CALL     T1
T0: MOVB     AL, #0x00
    SB       TIMINGS, UNC
T1: NOP
    RET
T2: NOP
```

Hardware breakpoints allow masking of address bits. For example, a hardware breakpoint could be placed on the address range $00\ 0200_{16}$–$00\ 02FF_{16}$ by specifying the following mask address, where the eight LSBs are don’t cares:

$00\ 0000\ 0000\ 0010\ XXXX\ XXXX_2$

7.7.2 Watchpoints

A hardware watchpoint triggers a debug event when either an address or an address and data match a compare value. The address portion is compared against a reference address and bit mask, and the data portion is compared against a reference data value and a bit mask.

When comparing two addresses, you can set two watchpoints. When comparing an address and a data value, you can set only one watchpoint. When performing a read watchpoint, the address is available a few cycles earlier than the data; the watchpoint logic accounts for this.

The point where execution stops depends on whether the watchpoint was a read or write watchpoint, and whether it was an address or an address/data read watchpoint. In the following example, a read address watchpoint occurs when the address X is accessed, and the CPU stops with the instruction counter (IC) pointing three instructions after that point:

```
MOV    AR4,#X
MOV    AL,*+AR4[0]    ; Data read nop
nop
nop                   ; The IC will point here
```

For a read watchpoint that requires both an address and data match, the CPU stops with the IC pointing six instructions after that point:

```
MOV    AR4,#X
MOV    AL,*+AR4[0]    ; Data read nop
nop
nop
nop
nop
nop                  ; The IC will point here
```

In the following example, a write address watchpoint occurs when the address Y is accessed, and the CPU stops with the IC pointing six instructions after that point:

```
MOV   AR4,#Y
MOV   *+AR4[0],AL    ; Data write
nop
nop
nop
nop
nop
nop                   ; The IC will point here
```

7.7.3 Benchmark Counter/Event Counter(s)

The 40-bit performance counter on the C28x can be used as a benchmark counter to increment every CPU clock cycle (it can be configured not to count when the CPU is in the debug-halt state). Wait states affect the counter. Wait states in the read 1 and write pipeline phases of an executing instruction affect the counter, regardless of whether an instruction is being single-stepped or run. However, wait states in the fetch 1 pipeline phase do not affect the counter during single-stepping, because the cycle counting does not begin until the decode 2 pipeline phase. The counter counts wait states caused by instructions that are fetched but not executed. In most cases, these effects cancel each other out. Benchmarking is best used for larger portions of code. Do not rely heavily on the precision of the benchmarking. (For more information about the pipeline, Chapter 4.)

Alternatively, you can configure the 40-bit performance counter as two 16-bit or one 32-bit event counter if you want to generate a debug event when the counter equals a match value. The comparison between the counter value and the match value is done before the count value is incremented. For example, suppose you initialize a counter to 0. A match value of 0 causes an immediate debug event (when the action to be counted occurs), and the counter holds 1 afterward.

You can also clear the counter when a hardware breakpoint or address watchpoint occurs. With this feature, you can implement a mechanism similar to a watchdog timer: if a certain address is not seen on the address bus within a certain number of CPU clock cycles, a debug event occurs.

7.7.4 Typical Analysis Unit Configurations

Each analysis unit can be configured to perform one analysis job at a time. Typical configurations for these two analysis units can be any one of the following:

- Two analysis breakpoints (i.e., hardware breakpoints)

  Detect when an instruction is executed from a specified address or range of addresses. Each hardware breakpoint only requires one analysis unit.
- Two hardware address watch points

  Detect when any value is either read from or written to a specified address or a range of addresses. In this case, the data written or read is not specified. Only the address of the location is specified and whether to watch for reads or writes to that address. Each watchpoint only requires one analysis unit.
- One address with data watchpoint

  Detect when a specified data value is either read from or written to a specified address. In this configuration you can either watch for a read or a write but not both reads and writes. This type of watchpoint requires both analysis units.
- A set of two chained breakpoints

  Detect when a given instruction is executed after another specified instruction.
- A benchmark counter/event counter

  The benchmark counter is only available with analysis unit 1. This counter can be used as a benchmark counter to count cycles or instructions. It can also be used to count AU2 events.

Configuration of the analysis resources is supported in Code Composer Studio. For more information on configuring these, use the Code Composer online help.

7.8 Data Logging

Data logging enables the C28x to send selected memory values to a host processor using the standard JTAG port and an XDS510 or other compatible scan controller. You control data logging activity with your application code.

To perform data logging, you must create a linear buffer of 32-bit words to hold a packet of information. Your application code controls the size, format, and location of this buffer and also determines when to send a buffer's contents to the host. You can control the size of a data logging buffer in two ways:

- Specify a count value in the upper eight bits of ADDRH (when the number of 32-bit words you want to log is between 1 and 256)
- Specify an end address

NOTE: When the debugger is not active, the data logging transfers are considered complete as soon as they are enabled to prevent the application software from getting stuck when there is nothing to receive the data.

7.8.1 Creating a Data Logging Transfer Buffer

To create a data logging transfer buffer, follow these steps in your application code:

1. Execute the EALLOW instruction to enable access to emulation registers.
2. Specify the start address of the buffer in ADDRL and the six LSBs of ADDRH (see Figure 7-6 and Figure 7-7). The address in ADDRL and ADDRH is called the transfer address.
3. Use either of the following methods to specify when data logging is to end:

   (a) If the number of words you want to log is between 1 and 256, specify a count value in the upper eight bits of ADDRH (see Figure 7-7). The form of the count value is 256−n, where n is the number of 32-bit words you want to log. As each word is transferred, both the transfer address and the count value are decremented.

   (b) If the number of words you want to log is greater than 256, specify a data logging end address in REFL and the six LSBs of REFH (see Figure 7-8 and Figure 7-9). Load the ten MSBs of REFH with 0s. When using this method, be sure to set the data logging end address control register (EVT_CNTRL) first, and then the DMA control register (DMA_CNTRL). EVT_CNTRL is described in Table 7-5, and DMA_CNTRL is described in Table 7-4.

NOTE: The application must not read from the end address of the buffer during the data logging operation. When the end address appears on the address bus, the C28x ends the transfer.

4. Execute the EDIS instruction to disable access to emulation registers.

   See Table 7-4 and Table 7-5 on the following pages for descriptions of the registers associated with data logging.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
16 LSBs of transfer address															

Figure 7-6. ADDRL (at Data-Space Address 00 0838_{16})

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Word counter								Reserved		6 MSBs of transfer address					

Figure 7-7. ADDRH (at Data-Space Address 00 0839_{16})

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
16 LSBs of end address															

Figure 7-8. REFL (at Data-Space Address 00 $084A_{16}$)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	6 MSBs of end address					

Figure 7-9. REFH (at Data-Space Address 00 084B$_{16}$)

Table 7-4. Start Address and DMA Registers

Address	Name	Access	Description	
00 0838$_{16}$	ADDRL	R/W	**Start address register (lower 16 bits)**	
			15:0	Lower 16 bits of start address
00 0839$_{16}$	ADDRH	R/W	**Word counter/start address register (upper 6 bits)**	
			15:8	Word counter. When using this to stop the data logging transfer, set the counter to 256 − *n*, where n is the number of 32-bit words to transfer. Otherwise set the counter to 0.
			7:6	Reserved. Set to 0.
			5:0	Upper 6 bits of start address
00 083E$_{16}$	DMA_CNTRL	R/W	**DMA control register**	
			15:14	Set to 0
			13	Set to 1
			12	Set to 1
			11	Give higher priority to: 0: CPU (nonpreemptive mode) 1: Data logging (preemptive mode)
			10	Allow data logging during time-critical ISR? 0: No 1: Yes
			9	Allow data logging while DBGM = 1? 0: No (polite accesses) 1: Yes (rude accesses)
			8:6	Set to 1
			5:4	0: EMU0/EMU1 using TCK 1: EMU0/EMU1 using FCK/2 2: JTAG signals 3: Reserved
			3:2	Method for ending data logging session: 0: Use the count register to stop data logging 1: Use an end address to stop data logging
			1:0	Data logging control/status: 0: Release resource from data logging operation 1: Claim resource for data logging operation 2: Enable resource for data logging operation 3: Data logging operation is complete. Bits 14:10 are corrupted when this occurs.
00 083F$_{16}$	DMA_ID	R	**DMA ID register**	
			15:14	Resource control: 0: Resource is free 1: Application owns resource 2: Debugger owns resource
			13:12	Set to 3.
			11:0	Set to 1.

Table 7-5. End-Address Registers

Address	Name	Access	Description	
00 0848$_{16}$	MASKL	R/W	Set to 0	
00 0849$_{16}$	MASKH	R/W	Set to 0	
00 084A$_{16}$	REFL	R/W	**Data logging end reference address (lower 16 bits)**	
			15:0	Lower 16 bits of start address
00 084B$_{16}$	REFH	R/W	**Data logging end reference address (upper 6 bits)**	
			15:6	Set to 0
			5:0	Upper 6 bits of start address
00 084E$_{16}$	EVT_CNTRL	R/W	**Data logging end address control register**	
			15:14	Set to 0
			13	Set to 1
			12	Set to 1
			11:5	Set to 0
			4:2	Set to 1
			1:0	End-address resource control/status: 0: Release end-address resource. 1: Claim end-address resource. 2: Enable end-address resource. 3: Data logging operation has ended. Bits 14:10 are corrupted when this occurs.
00 084F$_{16}$	EVT_ID	R	**Data logging end address ID register**	
			15:14	Resource control: 0: Resource is free 1: Application owns resource 2: Debugger owns resource
			13:12	Set to 1
			11:0	Set to 2

7.8.2 Accessing the Emulation Registers Properly

Make sure your application code follows the following protocol when accessing the emulation registers that have been provided for data logging. Each resource has a control register and an ID register.

1. Enable writes to memory-mapped registers by using the EALLOW instruction.
2. Write to the appropriate control register to claim the resource you want to use. The resource for data logging transfers uses DMA_CNTRL (see Table 7-4). The resource for detecting the data logging end address uses EVT_CNTRL (see Table 7-5).
3. Wait at least three cycles so that the write to the control register (done in the write phase of the pipeline) occurs before the read from the ID register in step 4. You can fill in the extra cycles with NOP (no operation) instructions or with other instructions that do not involve accessing the emulation registers
4. Read the appropriate ID register and verify that the application is the owner. The resource for data logging transfers uses DMA_ID (see Table 7-4). The resource for detecting the data logging end address uses EVT_ID (see Table 7-5). If the application is not the owner, then go back to step 2 until this succeeds (you may want a time-out function to prevent an endless loop). This step is optional. The application would fail to become the owner only if the debugger already owns the resource.
5. If the application is the owner, the remaining registers for that function can be programmed, and the control register written to again, to enable the function. However, if the application is not the owner, then all of its writes are ignored.
6. Disable writes to memory-mapped emulation registers by executing the EDIS instruction.

If an interrupt occurs between the EALLOW instruction in step 1 and the EDIS instruction in step 6, access to emulation registers are automatically disabled by the CPU before the interrupt service routine begins and automatically re-enabled when the CPU returns from the interrupt. This means that there is no need to disable interrupts between the EALLOW instruction and the EDIS instruction.

The debugger can, at your request, seize ownership of a register from the application; however, that is not the normal mode of operation.

7.8.3 Data Log Interrupt (DLOGINT)

The completion of a data logging transfer (determined either by the word counter or by the end address) triggers a DLOGINT request. DLOGINT is serviced only if it is properly enabled. If the CPU is halted in real-time mode, DLOGINT must be enabled in both the DBGIER and the IER. Otherwise, DLOGINT must be enabled in the IER and by the INTM bit in status register ST1.

This interrupt capability is most useful when there are multiple buffers of data to be transferred through data logging and the completion of one transfer should begin the next.

7.8.4 Examples of Data Logging

Section 7.8.4.1 shows how to log 20 32-bit words, starting at address 00 0100_{16} in data memory. The accesses are preemptive (they have higher priority than the CPU) and rude (they ignore the state of the DBGM bit). In addition, data logging can occur during time-critical interrupt service routines. The application can determine whether the data logging operation is complete by polling the LSB of the DMA control register (DMA_CNTRL) at 00 $083E_{16}$. When the operation is complete, that bit is set to 1.

7.8.4.1 Example 1: Initialization Code for Data Logging With Word Counter

```
; Base addresses
ADMA      .set    0838h

; Offsets
DMA_ADDRL .set    0
DMA_ADDRH .set    1
DMA_CNTRL .set    6
DMA_ID    .set    7

EALLOW
MOV    AR4, #ADMA                      ; AR4 pointing to register base addr
MOV    *+AR4[#DMA_CNTRL],#1            ; Attempt to claim resource
NOP
NOP
NOP
CMP    *+AR4[#DMA_ID],#7001h          ; Value expected in ID register
B      FAIL, NEQ                      ; If we don’t see the correct ID, then we
                                     ; failed (the resource is already in use)

MOV    *+AR4[#DMA_ADDRL],#0100h       ; Set starting address of buffer,
                                      ;    and then the count

MOV    *+AR4[#DMA_ADDRH], #((256 - 20) << 8)

MOV    *+AR4[#DMA_CNTRL],#3E62h
```

Section 7.8.4.2 shows how to log from address 00 0100_{16} to address 00 $02FF_{16}$ in data memory. The accesses are nonpreemptive (they have lower priority than the CPU), and are polite (they are not performed when the DBGM bit is 0). The data logging cannot occur when a time-critical interrupt is being serviced. An end address of 00 $02FF_{16}$ is used to end the transfer. The application must not read from 00 $02FF_{16}$ during the data logging; a read from that address stops the data logging. As in Section 7.8.4.1, the application can poll the LSB of DMA_CNTRL for a 1 to determine whether the data logging operation is complete.

7.8.4.2 Example 2: Initialization Code for Data Logging With Word Counter

```
; Base addresses
ADMA            .set    0838h
DEVT            .set    0848h

; Offsets
DMA_ADDRL      .set    0
DMA_ADDRH      .set    1
DMA_CNTRL      .set    6
DMA_ID         .set    7
MASKL          .set    0
MASKH          .set    1
REFL           .set    2
REFH           .set    3
EVT_CNTRL      .set    6
EVT_ID         .set    7


EALLOW
MOV    AR5, #DEVT                               ; AR5 pointing to End Address registers
MOV    AR4, #ADMA                               ; AR4 pointing to Start/Control base
MOV    *+AR5[#EVT_CNTRL],#1                     ; Attempt to claim End Address
MOV    *+AR4[#DMA_CNTRL],#1                     ; Attempt to claim Start/Control
NOP
NOP
NOP

CMP    *+AR5[#EVT_ID],#5002h                   ; Value expected in ID register

B Fail, NEQ                                    ; If we don't see the correct ID, FAIL

CMP *+AR4[#DMA_ID],#7001h                      ; Value expected in ID register
B      FAIL, NEQ                                ; If we    expected in ID register, FAIL

MOV    *+AR5[#MASKL],#0                        ;    Attempt to claim End    Address
MOV    *+AR5[#MASKH],#0                        ;    Attempt to claim End    Address
MOV    *+AR5[#REFL],#02FFh                     ;    Stop data logging at    address 0x02FF
MOV    *+AR5[#REFH],#0                         ;    Attempt to claim End    Addr

MOV   *+AR5 [#EVT_CNTRL], #  ( 2 | (1<<2)  | (1<<12)  |  (1<<13)  )

MOV    *+AR4[#DMA_ADDRL],#0100h             ; Set buffer start address and then the count
MOV *+AR4[DMA_ADDRH],#0


MOV     *+AR4[DMA_CNTRL],#3066h

EDIS
```

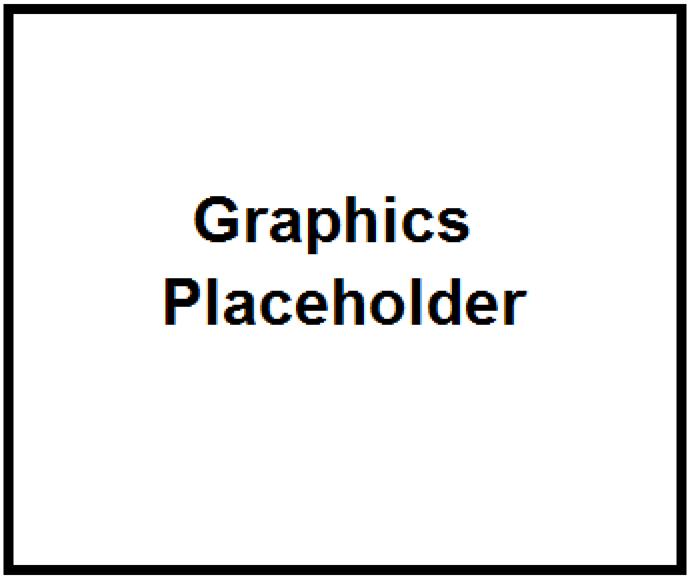
7.9 Sharing Analysis Resources

You can use analysis breakpoints, watchpoints, and a benchmark/event counter through the debugger, and you can use data logging through application code. Table 7-6 lists the analysis resources, and Figure 7-10 shows which resources are available to be used at the same time.

When the application owns analysis resources, they will be cleared (made unowned and set to the completed state) by a reset. When the debugger owns the resources, they are not cleared by reset but by the JTAG test-logic reset. This ensures that when you are using the debugger, the resources can be used even while the target system undergoes a reset.

Table 7-6. Analysis Resources

Resource	Purpose
BA0	Break on contents of program address or memory address bus
BA1	Break on contents of program address or memory address bus
DB	Break on contents of program data, memory read data, or memory write data in addition to an address bus
Data log	Perform data logging using counter
Benchmark	Count CPU cycles



[†] The data logging mode that uses the word counter allows this combination, but not the data logging mode that uses the end address (see Section 7.8).

Figure 7-10. Valid Combinations of Analysis Resources

7.10 Diagnostics and Recovery

Debug registers within the CPU keep track of the state of several key signals. This allows diagnosis of such problems as a floating READY signal, $\overline{\text{NMI}}$ signal, or $\overline{\text{RS}}$ (reset) signal. Should the debug software attempt an operation that does not complete after a certain time-out period (as determined by the debug software), it attempts to determine the probable cause and display the situation to you. You can then abort, correct the situation or allow it to correct itself, or chose to override it.

Such situations include:

- $\overline{\text{RS}}$ being asserted
- A ready signal not being asserted for a memory access
- $\overline{\text{NMI}}$ being asserted
- The absence of a functional clock
- The occurrence of a JTAG test-logic-reset

Appendix A
SPRU430F–August 2001–Revised April 2015

Register Quick Reference

For the status and control registers of the ’28x, this appendix summarizes:

- Their reset values
- The instructions available for accessing them
- The functions of their bits

Topic **Page**

A.1 Reset Values of and Instructions for Accessing the Registers

Table A-1 lists the CPU status and control registers, their reset values, and the instructions that are available for accessing the registers.

Table A-1. Reset Values of the Status and Control Registers

Register	Description	Reset Values	Instructions
ST0	Status register 0	$0000\ 0000\ 0000\ 0000_2$	PUSH, POP, SETC, CLRC
ST1	Status register 1	$0000\ M000\ 0000\ V011_2$	PUSH, POP, SETC, CLRC
IFR	Interrupt flag register	$0000\ 0000\ 0000\ 0000_2$	PUSH, POP, AND, OR
IER	Interrupt enable register	$0000\ 0000\ 0000\ 0000_2$	MOV, AND, OR
DBGIER	Debug interrupt enable register	$0000\ 0000\ 0000\ 0000_2$	PUSH, POP

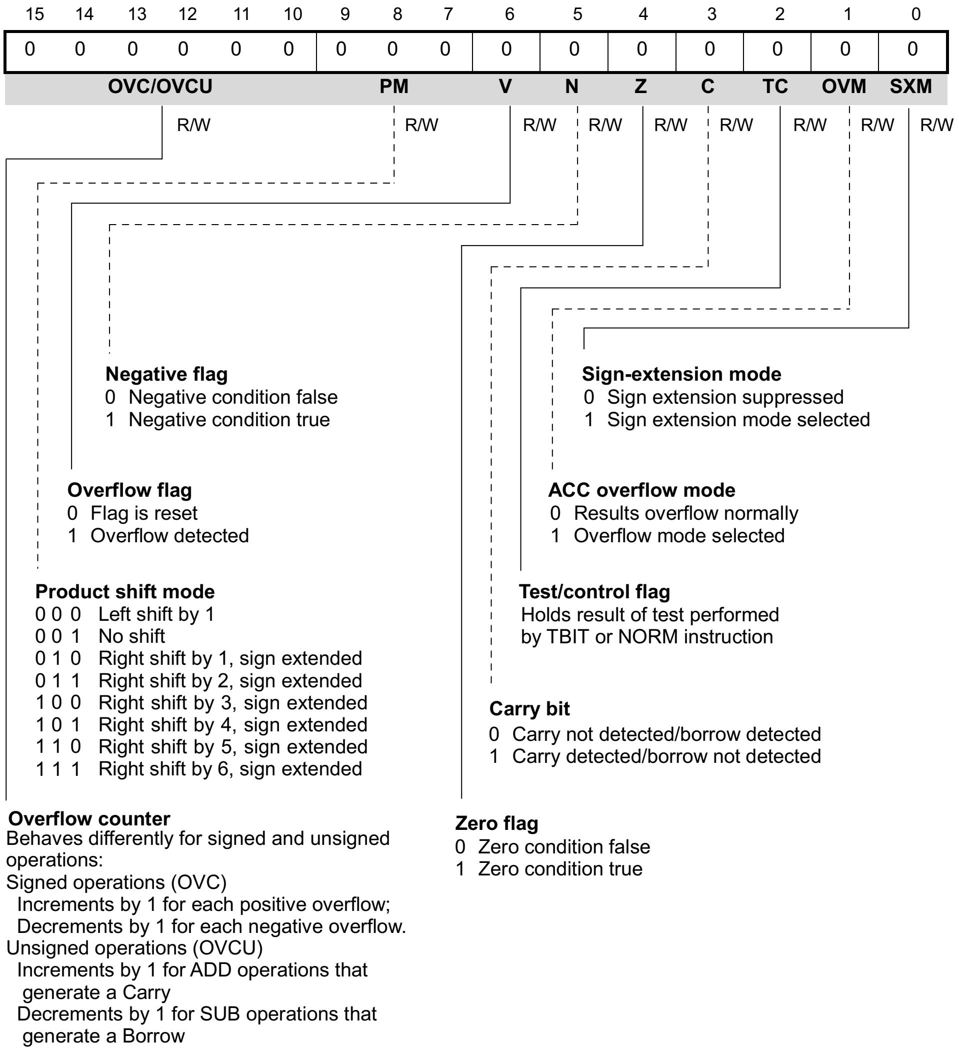
NOTE: V: Bit 3 of ST1 (the VMAP bit) depends on the level of the VMAP input signal at reset. If the VMAP signal is low, the VMAP bit is 0 after reset; if the VMAP signal is high, the VMAP bit is 1 after reset. For C28x devices that do not pin out VMAP, the signal is tied high internal to the device.

M: Bit 11 of ST1 (the M0M1MAP bit) depends on the level of the M0M1MAP input signal at reset. If the M0M1MAP signal is low, the bit is 0, high bit is 1. For C28x devices that do not pinout MOM1MAP, the signal is tied high internal to the device.

A.2 Register Figures

The following figures summarize the content of the ’28x status and control registers. Each figure in this provides information in this way:

- The value shown in the register is the value after reset.
- Each unreserved bit field or set of bits has a callout that very briefly de- scribes its effect on the processor.
- Each nonreserved bit field or set of bits is labeled with one of the following symbols:
  - R indicates that your software can read the bit field but cannot write to it.
  - R indicates that your software can read the bit field but cannot write to it.
- Where needed, footnotes provide additional information for a particular figure.



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
OVC/OVCU						PM			V	N	Z	C	TC	OVM	SXM
R/W						R/W			R/W	R/W	R/W	R/W	R/W	R/W	R/W

Negative flag
0 Negative condition false
1 Negative condition true

Overflow flag
0 Flag is reset
1 Overflow detected

Product shift mode

0 0 0	Left shift by 1
0 0 1	No shift
0 1 0	Right shift by 1, sign extended
0 1 1	Right shift by 2, sign extended
1 0 0	Right shift by 3, sign extended
1 0 1	Right shift by 4, sign extended
1 1 0	Right shift by 5, sign extended
1 1 1	Right shift by 6, sign extended

Overflow counter
Behaves differently for signed and unsigned operations:
Signed operations (OVC)
Increments by 1 for each positive overflow;
Decrements by 1 for each negative overflow.
Unsigned operations (OVCU)
Increments by 1 for ADD operations that generate a Carry
Decrements by 1 for SUB operations that generate a Borrow

Sign-extension mode
0 Sign extension suppressed
1 Sign extension mode selected

ACC overflow mode
0 Results overflow normally
1 Overflow mode selected

Test/control flag
Holds result of test performed by TBIT or NORM instruction

Carry bit
0 Carry not detected/borrow detected
1 Carry detected/borrow not detected

Zero flag
0 Zero condition false
1 Zero condition true

Figure A-1. Status Register ST0

NOTE: Note: For more details about ST0, see section 2.3 on page 2-16.

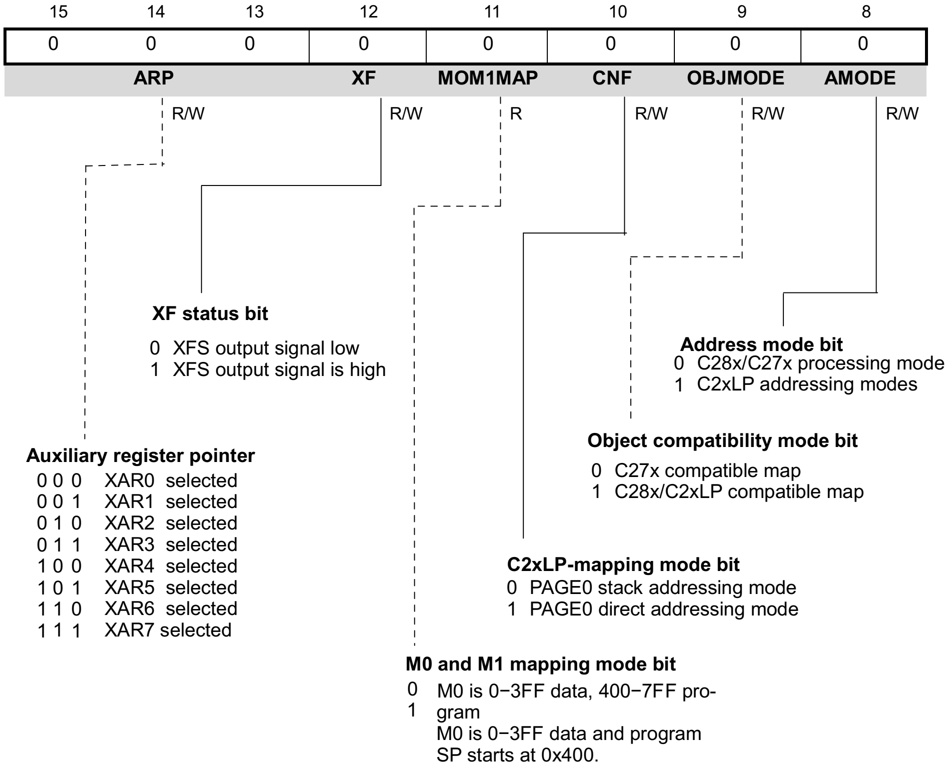
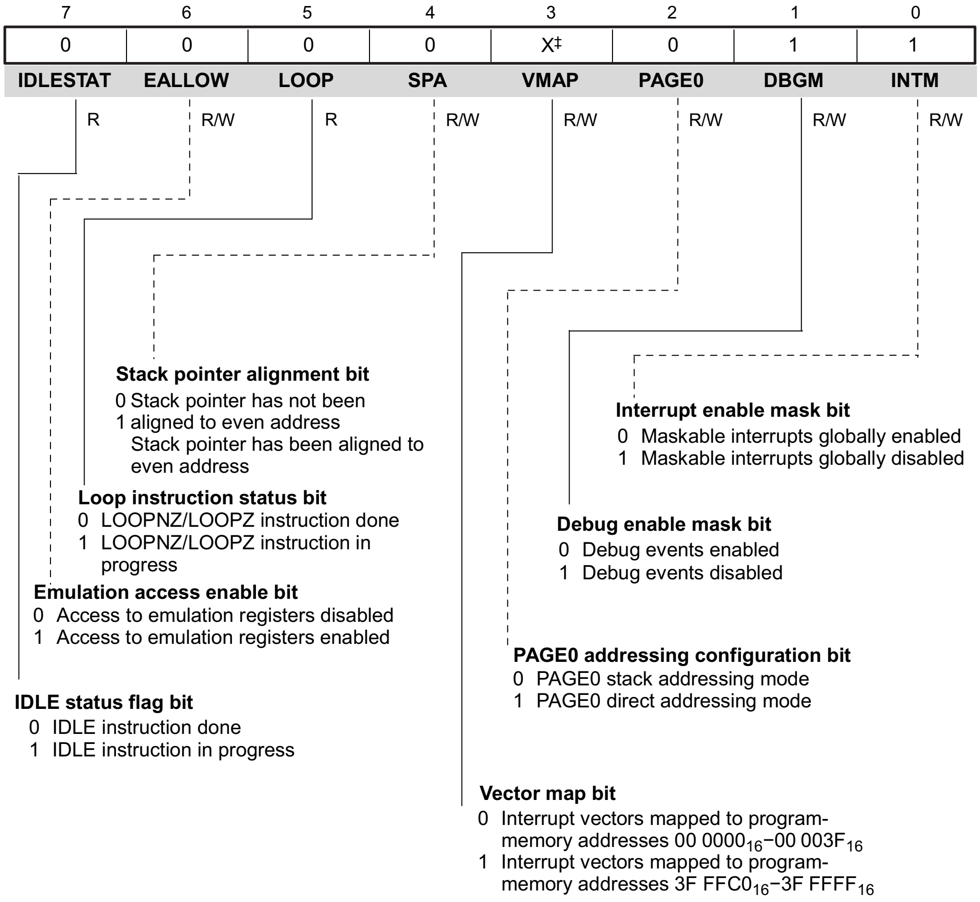


Figure A-2. Status Register ST1, Bits 15-8



7	6	5	4	3	2	1	0
0	0	0	0	X‡	0	1	1
IDLESTAT	**EALLOW**	**LOOP**	**SPA**	**VMAP**	**PAGE0**	**DBGM**	**INTM**
R	R/W	R	R/W	R/W	R/W	R/W	R/W

Stack pointer alignment bit
0 Stack pointer has not been
1 aligned to even address
Stack pointer has been aligned to even address

Loop instruction status bit
0 LOOPNZ/LOOPZ instruction done
1 LOOPNZ/LOOPZ instruction in progress

Emulation access enable bit
0 Access to emulation registers disabled
1 Access to emulation registers enabled

IDLE status flag bit
0 IDLE instruction done
1 IDLE instruction in progress

Interrupt enable mask bit
0 Maskable interrupts globally enabled
1 Maskable interrupts globally disabled

Debug enable mask bit
0 Debug events enabled
1 Debug events disabled

PAGE0 addressing configuration bit
0 PAGE0 stack addressing mode
1 PAGE0 direct addressing mode

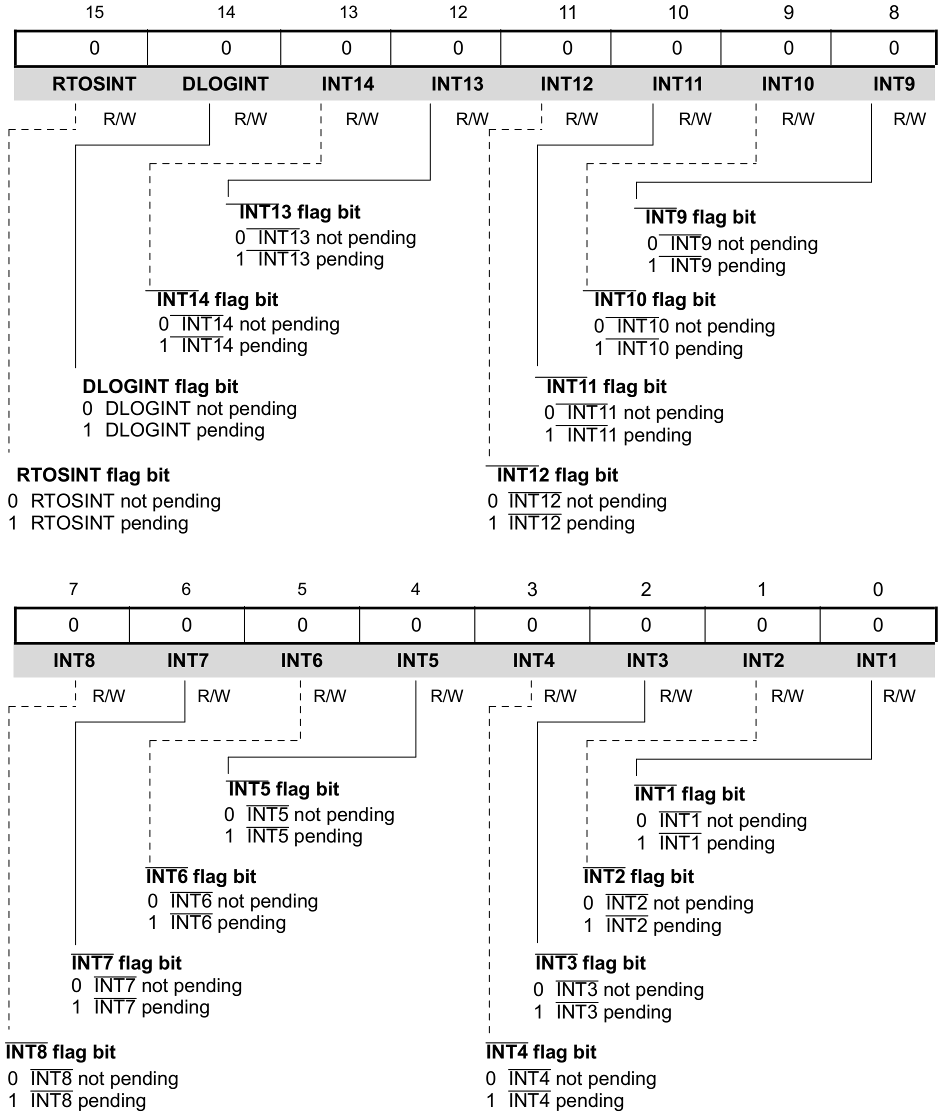
Vector map bit
0 Interrupt vectors mapped to program-memory addresses 00 0000_{16}–00 $003F_{16}$
1 Interrupt vectors mapped to program-memory addresses 3F $FFC0_{16}$–3F $FFFF_{16}$

(1) These reserved bits are always 0s and are not affected by writes.

(2) The VMAP bit depends on the level of the VMAP input signal at reset. If the VMAP signal is low, the VMAP bit is 0 after reset; if the VMAP signal is high, the VMAP bit is 1 after reset. For C28x devices that do not pin out the VMAP signal, the signal is tied high internal to the device.

Figure A-3. Status Regsiter ST1, Bits 7-0

NOTE: For more details about ST1, see section 2.4 on page 2-34.



15	14	13	12	11	10	9	8
0	0	0	0	0	0	0	0
RTOSINT	DLOGINT	$\overline{\text{INT14}}$	$\overline{\text{INT13}}$	$\overline{\text{INT12}}$	$\overline{\text{INT11}}$	$\overline{\text{INT10}}$	$\overline{\text{INT9}}$
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

$\overline{\text{INT13}}$ flag bit
0 $\overline{\text{INT13}}$ not pending
1 $\overline{\text{INT13}}$ pending

$\overline{\text{INT14}}$ flag bit
0 $\overline{\text{INT14}}$ not pending
1 $\overline{\text{INT14}}$ pending

DLOGINT flag bit
0 DLOGINT not pending
1 DLOGINT pending

RTOSINT flag bit
0 RTOSINT not pending
1 RTOSINT pending

$\overline{\text{INT9}}$ flag bit
0 $\overline{\text{INT9}}$ not pending
1 $\overline{\text{INT9}}$ pending

$\overline{\text{INT10}}$ flag bit
0 $\overline{\text{INT10}}$ not pending
1 $\overline{\text{INT10}}$ pending

$\overline{\text{INT11}}$ flag bit
0 $\overline{\text{INT11}}$ not pending
1 $\overline{\text{INT11}}$ pending

$\overline{\text{INT12}}$ flag bit
0 $\overline{\text{INT12}}$ not pending
1 $\overline{\text{INT12}}$ pending

7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0
$\overline{\text{INT8}}$	$\overline{\text{INT7}}$	$\overline{\text{INT6}}$	$\overline{\text{INT5}}$	$\overline{\text{INT4}}$	$\overline{\text{INT3}}$	$\overline{\text{INT2}}$	$\overline{\text{INT1}}$
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

$\overline{\text{INT5}}$ flag bit
0 $\overline{\text{INT5}}$ not pending
1 $\overline{\text{INT5}}$ pending

$\overline{\text{INT6}}$ flag bit
0 $\overline{\text{INT6}}$ not pending
1 $\overline{\text{INT6}}$ pending

$\overline{\text{INT7}}$ flag bit
0 $\overline{\text{INT7}}$ not pending
1 $\overline{\text{INT7}}$ pending

$\overline{\text{INT8}}$ flag bit
0 $\overline{\text{INT8}}$ not pending
1 $\overline{\text{INT8}}$ pending

$\overline{\text{INT1}}$ flag bit
0 $\overline{\text{INT1}}$ not pending
1 $\overline{\text{INT1}}$ pending

$\overline{\text{INT2}}$ flag bit
0 $\overline{\text{INT2}}$ not pending
1 $\overline{\text{INT2}}$ pending

$\overline{\text{INT3}}$ flag bit
0 $\overline{\text{INT3}}$ not pending
1 $\overline{\text{INT3}}$ pending

$\overline{\text{INT4}}$ flag bit
0 $\overline{\text{INT4}}$ not pending
1 $\overline{\text{INT4}}$ pending

Figure A-4. Interrupt Flag Register (IFR)

NOTE: For more details about the IFR, see section 3.3.1 on page 3-7.

15	14	13	12	11	10	9	8
0	0	0	0	0	0	0	0
RTOSINT	**DLOGINT**	**INT14**	**INT13**	**INT12**	**INT11**	**INT10**	**INT9**
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

$\overline{\text{INT13}}$ enable bit
0 $\overline{\text{INT13}}$ disabled
1 $\overline{\text{INT13}}$ enabled

$\overline{\text{INT14}}$ enable bit
0 $\overline{\text{INT14}}$ disabled
1 $\overline{\text{INT14}}$ enabled

DLOGINT enable bit
0 DLOGINT disabled
1 DLOGINT enabled

RTOSINT enable bit
0 RTOSINT disabled
1 RTOSINT enabled

$\overline{\text{INT9}}$ enable bit
0 $\overline{\text{INT9}}$ disabled
1 $\overline{\text{INT9}}$ enabled

$\overline{\text{INT10}}$ enable bit
0 $\overline{\text{INT10}}$ disabled
1 $\overline{\text{INT10}}$ enabled

$\overline{\text{INT11}}$ enable bit
0 $\overline{\text{INT11}}$ disabled
1 $\overline{\text{INT11}}$ enabled

$\overline{\text{INT12}}$ enable bit
0 $\overline{\text{INT12}}$ disabled
1 $\overline{\text{INT12}}$ enabled

7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0
INT8	**INT7**	**INT6**	**INT5**	**INT4**	**INT3**	**INT2**	**INT1**
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

$\overline{\text{INT5}}$ enable bit
0 $\overline{\text{INT5}}$ disabled
1 $\overline{\text{INT5}}$ enabled

$\overline{\text{INT6}}$ enable bit
0 $\overline{\text{INT6}}$ disabled
1 $\overline{\text{INT6}}$ enabled

$\overline{\text{INT7}}$ enable bit
0 $\overline{\text{INT7}}$ disabled
1 $\overline{\text{INT7}}$ enabled

$\overline{\text{INT8}}$ enable bit
0 $\overline{\text{INT8}}$ disabled
1 $\overline{\text{INT8}}$ enabled

$\overline{\text{INT1}}$ enable bit
0 $\overline{\text{INT1}}$ disabled
1 $\overline{\text{INT1}}$ enabled

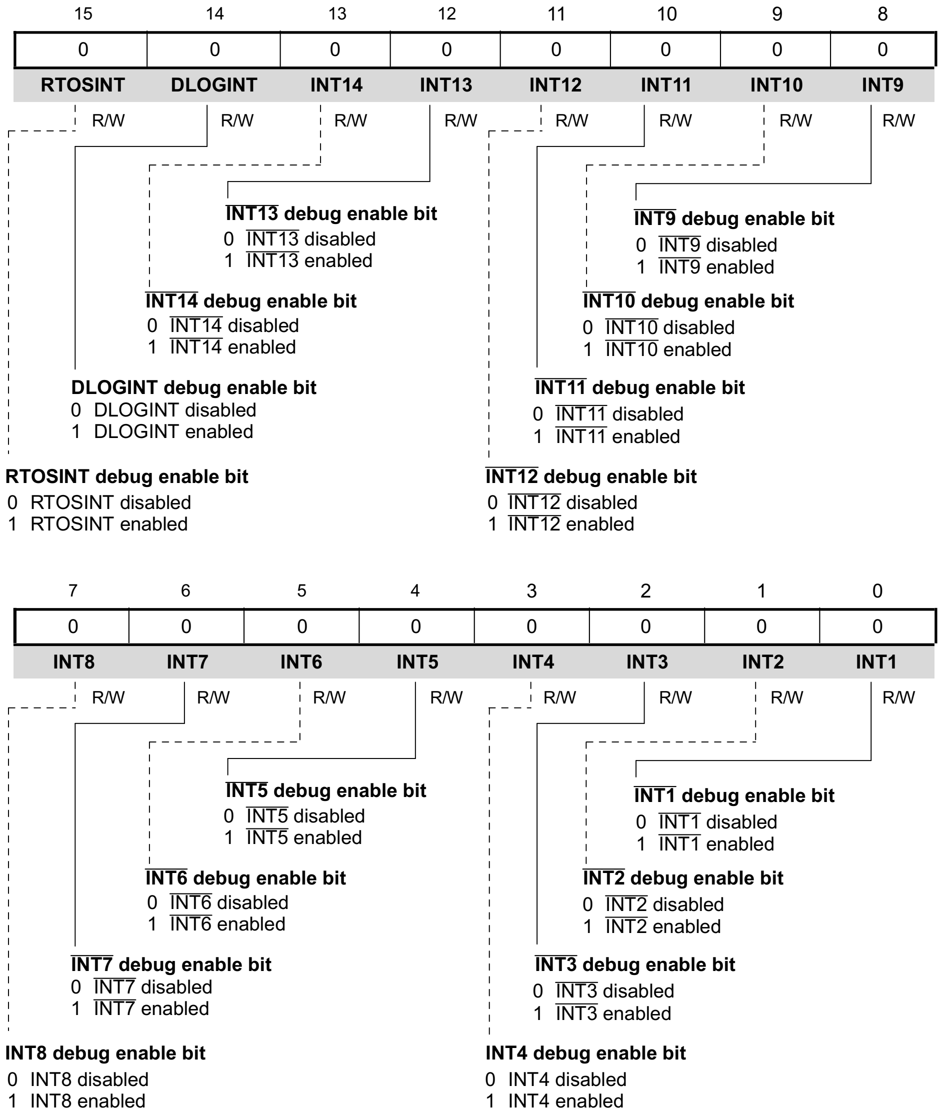
$\overline{\text{INT2}}$ enable bit
0 $\overline{\text{INT2}}$ disabled
1 $\overline{\text{INT2}}$ enabled

$\overline{\text{INT3}}$ enable bit
0 $\overline{\text{INT3}}$ disabled
1 $\overline{\text{INT3}}$ enabled

$\overline{\text{INT4}}$ enable bit
0 $\overline{\text{INT4}}$ disabled
1 $\overline{\text{INT4}}$ enabled

Figure A-5. Interrupt Enable Register (IER)

NOTE: For more details about the IER, see section 3.3.2 on page 3-8.



15	14	13	12	11	10	9	8
0	0	0	0	0	0	0	0
RTOSINT	**DLOGINT**	**INT14**	**INT13**	**INT12**	**INT11**	**INT10**	**INT9**
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

RTOSINT debug enable bit
0 RTOSINT disabled
1 RTOSINT enabled

DLOGINT debug enable bit
0 DLOGINT disabled
1 DLOGINT enabled

$\overline{\text{INT14}}$ debug enable bit
0 $\overline{\text{INT14}}$ disabled
1 $\overline{\text{INT14}}$ enabled

$\overline{\text{INT13}}$ debug enable bit
0 $\overline{\text{INT13}}$ disabled
1 $\overline{\text{INT13}}$ enabled

$\overline{\text{INT12}}$ debug enable bit
0 $\overline{\text{INT12}}$ disabled
1 $\overline{\text{INT12}}$ enabled

$\overline{\text{INT11}}$ debug enable bit
0 $\overline{\text{INT11}}$ disabled
1 $\overline{\text{INT11}}$ enabled

$\overline{\text{INT10}}$ debug enable bit
0 $\overline{\text{INT10}}$ disabled
1 $\overline{\text{INT10}}$ enabled

$\overline{\text{INT9}}$ debug enable bit
0 $\overline{\text{INT9}}$ disabled
1 $\overline{\text{INT9}}$ enabled

7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0
INT8	**INT7**	**INT6**	**INT5**	**INT4**	**INT3**	**INT2**	**INT1**
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

INT8 debug enable bit
0 INT8 disabled
1 INT8 enabled

$\overline{\text{INT7}}$ debug enable bit
0 $\overline{\text{INT7}}$ disabled
1 $\overline{\text{INT7}}$ enabled

$\overline{\text{INT6}}$ debug enable bit
0 $\overline{\text{INT6}}$ disabled
1 $\overline{\text{INT6}}$ enabled

$\overline{\text{INT5}}$ debug enable bit
0 $\overline{\text{INT5}}$ disabled
1 $\overline{\text{INT5}}$ enabled

INT4 debug enable bit
0 INT4 disabled
1 INT4 enabled

$\overline{\text{INT3}}$ debug enable bit
0 $\overline{\text{INT3}}$ disabled
1 $\overline{\text{INT3}}$ enabled

$\overline{\text{INT2}}$ debug enable bit
0 $\overline{\text{INT2}}$ disabled
1 $\overline{\text{INT2}}$ enabled

$\overline{\text{INT1}}$ debug enable bit
0 $\overline{\text{INT1}}$ disabled
1 $\overline{\text{INT1}}$ enabled

Figure A-6. Debug Interrupt Enable Register (DBGIER)

NOTE: For more details about the DBGIER, see section 3.3.2 on page 3-8

C2xLP and C28x Architectural Differences

This appendix highlights some of the architecture differences between the C2xLP and the C28x. Not all of the changes are listed here. An emphasis is placed on those changes of which you need to be aware while migrating from a C2xLP-based design to a C28x design. In particular changes in CPU registers and memory map are addressed.

Topic **Page**

B.1 Summary of Architecture Differences Between C2xLP and C28x

The C28x CPU features many improvements over the C2xLP CPU. A summary of the enhancements is given here.

Table B-1. General Features

Features	C2xLP	C28x
Program memory space	64K (16 address signals)	4M (22 address signals)
Data memory space	64K (16 address signals)	4G (32 address signals)
Number of internal buses	3 (prog, data-read, data-write)	3 (prog, data-read, data-write)
Addressable word size	16	16/32
Multiplier	16 bits	16/32 bits
Maskable CPU interrupts	6	14

B.1.1 Enhancements of the C28x over the C2xLP

- Much higher MHz operation
- 32 x 32 MAC
- 16 x16 Dual MAC
- 32-bit register file
- 32-bit single-cycle operations
- 4M linear program-address reach
- 4G linear data-address reach
- Dedicated software stack pointer
- Monitorless real-time emulation
- 40−50% better C code efficiency than C2xLP
- 20−30% better assembly code efficiency than C2xLP
- Atomic operation eliminates need to disable/re-enable interrupts
- Extended debugging features (Analysis block, data logging, etc.)
- Faster interrupt context save/restore
- More efficient addressing modes
- Unified memory map
- Byte packing and unpacking operations

When you first recompile your C2xLP code set for C28x, you will not be able to take advantage of every enhancement since you are limited by the original source code. Once you begin migrating your code, however, you will quickly begin to take advantage of the full capabilities the C28x offers. See Appendix D for help with migration to C28x.

B.2 Registers

The register modifications to the C2xLP are shown in Figure B-1. Registers that are shaded show the changes or enhancements on the C28x. The italicized names on the left are the original C2xLP names for the registers. The names on the right are the C28x names for the registers.

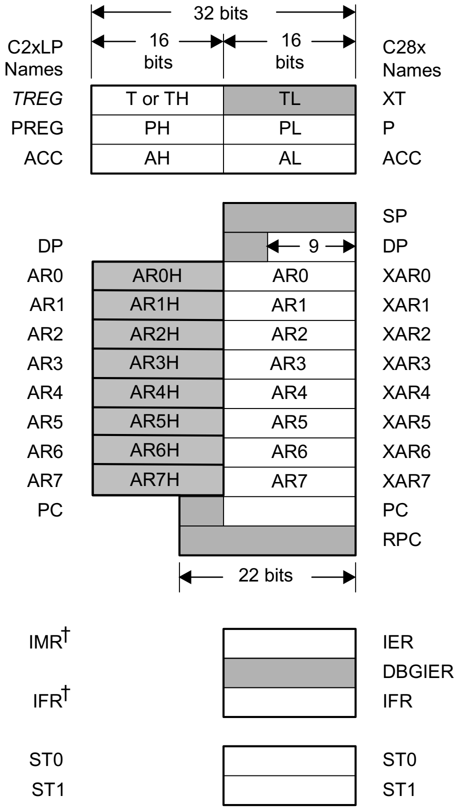


Figure B-1. Register Changes from C2xLP to C28x

B.2.1 CPU Register Changes

A brief description of the register modifications is given below. For a complete description of each register, see descriptions in the C2xLP and C28x Reference Guides.

XT — Multiplicand register

The 32-bit multiplicand register is called XT on the C28x. The C2xLP TREG is represented by the upper 16 bits (T). The lower 16 bit area is known as TL. The assembler will also accept TH in place of T for the upper 16 bits of the XT register.

P — Product register

This register is the same as the C2xLP PREG. You can separately access the high half (PH) or the low half (PL) on the C28x.

ACC — Accumulator

The size of ACC is the same on the C28x. Access to the register has been enhanced. On C28x, you can access it as two 16-bit registers (AL and AH).

SP — Stack Pointer

The SP is new on the C28x. It points directly to the C28x software stack

XAR0 - XAR7 — Auxiliary registers

All of the auxiliary registers (XARn) are increased to 32 bits on the C28x. This enables a full 32-bit address reach in data space. Some instructions separately access the low half of the registers (ARn).

PC — Program counter

The PC is 22 bits on C28x. On the C2xLP, the PC is 16 bits

RPC — Return program counter

The RPC register is new on the C28x. When a call operation is performed, the return address is saved in the RPC register and the old value in the RPC is saved on the stack. When a return operation is performed, the return address is read from the RPC register and the value on the stack is written into the RPC register. The net result is that return operations are faster (4 instead of 8 cycles). This register is only used when certain call and return instructions are used. Normal call and return instructions bypass this register.

IER — Interrupt enable register

The IER is analogous to the Interrupt Mask Register (IMR) on the C2xLP. It performs the same function, however, the name has changed to more appropriately describe the function of the register. Each bit in the register enables one of the maskable interrupts. On the C2xLP, there are six maskable CPU interrupts. On the C28x CPU, there are 16 CPU interrupts. On the C2xLP, the IMR was memory mapped.

DBGIER — Debug interrupt-enable register

The DBGIER is new on the C28x. It enables interrupts during debug events and allows the processor and debugger to perform real-time emulation.

IFR — Interrupt flag register

The IFR functions the same as on the C2xLP. There are more valid bits in this register to accommodate the additional interrupts on the C28x. On the C2xLP, the IFR was memory mapped.

STO/ST1 — Status Registers

The C28x status register bit positions are different compared to the C2xLP. Figure B-3 shows the differences.

DP — Data Page Pointer

On the C2xLP the DP is part of status register ST0. The DP on the C28x is a separate register and is increased from 9 to 16 bits.

B.2.2 Data Page (DP) Pointer Changes

B.2.2.1 C2xLP DP

The direct addressing mode on the C2xLP can access any data memory location in the 64K address range of the device using a 9-bit data page pointer and a 7-bit offset, supplied by the instruction, which is concatenated with the data page pointer value to form the 16-bit data address location. An example C2xLP operation is as follows:

```
LDP #VarA    ; Load DP with page location for VarA
LACL VarA    ; Load ACC low with contents of VarA
```

The first instruction initializes the DP register value with the ”page” location for the specified variable. Each page is 128 words in size. The assembler/linker automatically resolve the page value by dividing the absolute address of the specified location by 128. For example:

```
If "VarA" address = 0x3456, then the DP value is:
    DP(8:0) = 0x3456/128 = 0x69
```

The next instruction will then calculate the 7-bit offset of the specified variable within the 128-word page. This offset value is then embedded in the address field for that instruction. The assembler/linker automatically resolves the offset value by taking the first 7 bits of the absolute address of the specified location. For example:

```
If "VarA" address = 0x3456, then the 7bit offset value is:
    7-bit offset = 0x3456 & 0x007F = 0x56
```

B.2.2.2 C28x DP

The C28x also supports the direct addressing mode using the DP register; however, the following changes and enhancements have been made:

- Supports 22-bit address reach
- DP increased from 9 to 16 bits
- DP is a separate 16-bit register
- When AMODE == 0, page size is 64 words and DP(15:0) is used
- When AMODE == 1, page size is 128 words and DP(15:1) is used, bit 0 of DP is ignored

When AMODE == 1, the DP and the direct addressing mode behaves identically to the C2xLP but are enhanced to 22-bit address reach from 16. When AMODE == 0, the page size is reduced by half. This was done to accommodate other useful addressing modes.

The mapping of the direct addressing modes between the C2xLP and the C28x is as shown in Figure B-2.

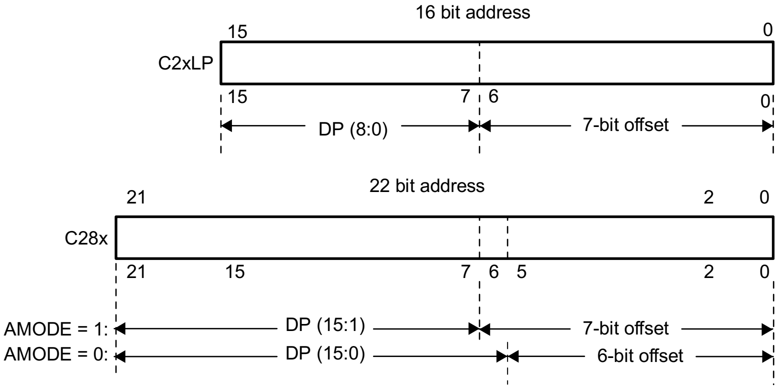


Figure B-2. Direct Addressing Mode Mapping

Using the previous example, the assembler/linker will initialize the DP and offset values as follows on the C28x:

C2xLP Original Source Mode ("−v28 −m20" mode, AMODE == 1)

```
LDP #VarA ; DP(15:0) = 0x3456/128 << 1 = 0x00D1
LACL VarA ; 7-bit offset = 0x3456 & 0x007F = 0x56
```

Equivalent C28x Mnemonics (after C2xLP source is reassembled with the C28x assembler)

```
MOVZ DP,#VarA ; DP(15:0) = 0x3456/128 << 1 = 0x00D1
MOVU ACC,@@VarA ; 7-bit offset = 0x3456 & 0x007F = 0x56
```

C28x Addressing Mode ("−v28" mode, AMODE == 0)

```
MOVZ DP,#VarA ; DP(15:0) = 0x3456/64 = 0x00D1
MOVU ACC,@VarA ; 6-bit offset = 0x3456 & 0x003F = 0x16
```

NOTE: When using C28x syntax, the 128 word data page is indicated by using the double "@@" symbol. The 64 word data page is indicated by the single "@" symbol. This helps the user and assembler to track which mode is being used.

B.2.3 Status Register Changes

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ARP			OV	OVM	1	INTM	DP								
R/W-X			R/W-0	R/W-X		R/W-1	R/W-X								

LEGEND: R/W = Read/Write; R = Read only; -*n* = value after reset

Figure B-3. C2xLP Status Register ST0

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OVC/OVCU						PM			V	N	Z	C	TC	OVM	SXM
R/W−000000						R/W−000			R/W−0	R/W−0	R/W−0	R/W−0	R/W−0	R/W−0	R/W−0

LEGEND: R/W = Read/Write; R = Read only; -*n* = value after reset

Figure B-4. C28X Status Register ST0

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ARB			CNF	TC	SXM	C	1	1	1	1	XF	1	1	PM	
R/W-X			R/W-0	R/W-X	R/W-1	R/W-1	R/W-1							R/W-00	

LEGEND: R/W = Read/Write; R = Read only; -*n* = value after reset

Figure B-5. C28XLP Status Register ST1

15	14	13	12	11	10	9	8
ARP			XF	M0M1MAP	Reserved	OBJMODE	AMODE
R/W−000			R/W−0	R−1	R−0	R/W−0	R/W−0
7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
IDLESTAT	EALLOW	LOOP	SPA	VMAP	PAGE0	DBGM	INTM
R−0	R/W−0	R−0	R/W−0	R/W−1	R/W−0	R/W−1	R/W−1

LEGEND: R/W = Read/Write; R = Read only; -*n* = value after reset

Figure B-6. C28x Status Register ST1

Z — Zero flag

Z is new on the C28x. It is involved in determining if the results of certain operations are 0. It is also used for conditional operations.

N — Negative flag

N is new on the C28x. It is involved in determining if the results of certain operations are negative. It is also used for conditional operations.

V — Overflow flag

V has changed names from OV on the C2xLP. It flags overflow conditions in the accumulator.

PM — Product shift mode

The PM has increased to a 3-bit register with additional capabilities. Below is a comparison of the PM register in the C2xLP and the C28x. Note that the register behaves differently depending on the operational mode of the C28x device. The XSPM instructions correspond to equivalent C2xLP instructions conversion. On the C2xLP, the PM bits corresponded to no shift at reset. On C28x, however, the PM corresponds to a left shift of 1 at reset.

Table B-2. C2xLP Product Mode Shifter

Bits	Shift Value	Instruction
00	no shift	SPM 0
01	shift left 1	SPM 1
10	shift left 4	SPM 2
11	shift right 6	SPM 3

Table B-3. C28x Product Mode Shifter

	C2xLP Source-Compatible Mode AMODE == 1 OBJMODE = 1 PAGE0 == 0		C28x Mode AMODE == 0 OBJMODE = 1 PAGE0 == 0	
Bits	**Shift Value**	**Instruction**	**Shift Value**	**Instruction**
000	shift left 1	SPM + 1 (or SPM)	shift left 1	SPM+1
001	no shift	SPM 0 (or SPM 0)	no shift	SPM 0
010	shift right 1	SPM–1	shift right 1	SPM–1
011	shift right 2	SPM–2	shift right 2	SPM–2
100	shift right 3	SPM–3	shift right 3	SPM–3
101	shift left 4	SPM+4 (or SPM 2)	shift left 4	SPM–4
110	shift right 5	SPM–5	shift right 5	SPM–5
111	shift right 6	SPM–6 (or SPM 3)	shift right 6	SPM–6

OVC — Overflow counter

OVC is new on the C28x. It can be viewed as an extension of the accumulator. For signed operations, the OVC counter is an extension of the overflow mode. For unsigned operations, the OVC counter (OVCU) is an extension of the carry mode.

DBGM — Debug enable mask bit

DBGM is new on the C28x. It is analogous to the INTM bit and works in cooperation with the DBGIER register to globally enable interrupts in real-time emulation.

PAGE0 — PAGE0 addressing mode configuration bit

The PAGE0 bit is new on the C28x. It is used for compatibility to the C27x and should be left as 0 for users moving from the C2xLP to C28x.

VMAP — Vector map bit

The VMAP bit is new on the C28x. It determines from where in memory interrupt vectors will be fetched.

SPA — Stack pointer alignment bit

The SPA bit is new on the C28x. It is a flag used to determine if aligning the stack pointer caused an adjustment in the stack pointer address.

LOOP — Loop instruction status bit

The LOOP bit is new on the C28x. It is used in conjunction with the LOOPZ/LOOPNZ instructions.

EALLOW — Emulation access enable bit

The EALLOW bit is new on the C28x. It allows access to the emulation register on the C28x.

IDLESTAT — IDLE status bit

The IDLESTAT bit is new on the C28x. It flags an IDLE condition on the C28x, and is mainly used when returning from an interrupt.

AMODE — Address mode bit

The AMODE bit is new on the C28x. This mode bit is used to select between C28x addressing mode (AMODE == 0) and C2xLP addressing mode (AMODE == 1).

OBJMODE — Object mode bit

The OBJMODE bit is new on the C28x. It is used to select between C27x object mode (OBJMODE == 0) and C28x object mode (OBJMODE == 1). For users moving from C2xLP to C28x, this bit should always be set to 1.

NOTE: Upon reset of the C28x, this bit is set to 0 and needs to be changed in firmware.

M0M1MAP — M0 M1 map bit

The M0M1MAP bit is new on the C28x. It is only used for C27x compatibility. For users transitioning from the C2xLP to C28x this bit should always be set to 1.

XF — XF pin status bit

The XF pin has the same function as on the C2xLP. Please note that the reset state has changed on the C28x.

ARP — Auxiliary register pointer

The ARP has the same functionality as on the C2xLP. It should, however, only be used when transitioning code to the C28x. The C28x has enhanced addressing modes which eliminate the need to keep track of the ARP.

The functionality of the remaining bits is the same on C28x as they are on C2xLP. It should be noted that although the functionality did not change, the bit position in the registers did. These bits are:

- Sign extension mode (SXM)
- Overflow mode (OVM)
- Test/control flag (TC)
- Carry bit (C)
- Interrupt global mask bit (INTM)

B.2.4 Register Reset Conditions

The reset conditions of internal registers have changed between the C2xLP and C28x as shown in Table B-4. Most C28x registers are cleared on a reset.

Differences in Table B-5 are highlighted in **bold**.

Table B-4. Reset Conditions of Internal Registers[1]

C2xLP Register	C2xLP Reset	C28x Register	C28x Reset
T	X	XT	0x00000000
P	X	P	0x00000000
ACC	X	ACC	0x00000000
AR0 − AR7	X	XAR0 − XAR7	0x00000000
PC	0x0000	PC	0x3FFFC0
ST0	See Table B-5	ST0	0x0000
ST1	See Table B-5	ST1	0x080B
DP	X	DP	0x0000
−	−	SP	0x0400
IMR	0x00	IER	0x0000
−	−	DBGIER	0x0000
IFR	0x0000	IFR	0x0000
GREG	0x0000	−	−
−	−	RPC	0x000000

[1] X = Unintiated

Table B-5. Status Register Bits

Reg	C2xLP Bit Name	C2xLP Reset Value	C28x Bit Name	C28x Reset Value
ST0	DP	XXXXXXXXX	**SXM**	0
	INTM	1	OVM	0
	OVM	X	TC	0
	OV	0	**C**	0
	ARP	XXX	Z	0
			N	0
			V	0
			PM	000 (left shift 1)
ST1	**PM**	**00 (no shift)**	INTM	1
	XF	1	DBGM	1
	C	**1**	PAGE0	0
	SXM	**1**	VMAP	1
	TC	X	SPA	0
	CNF	0	LOOP	0
	ARB	XXX	EALLOW	0
			IDLESTAT	0
			AMODE	0
			OBJMODE	0
			CNF not implemented	0
			M0M1MAP	1
			XF	0
			ARP	000

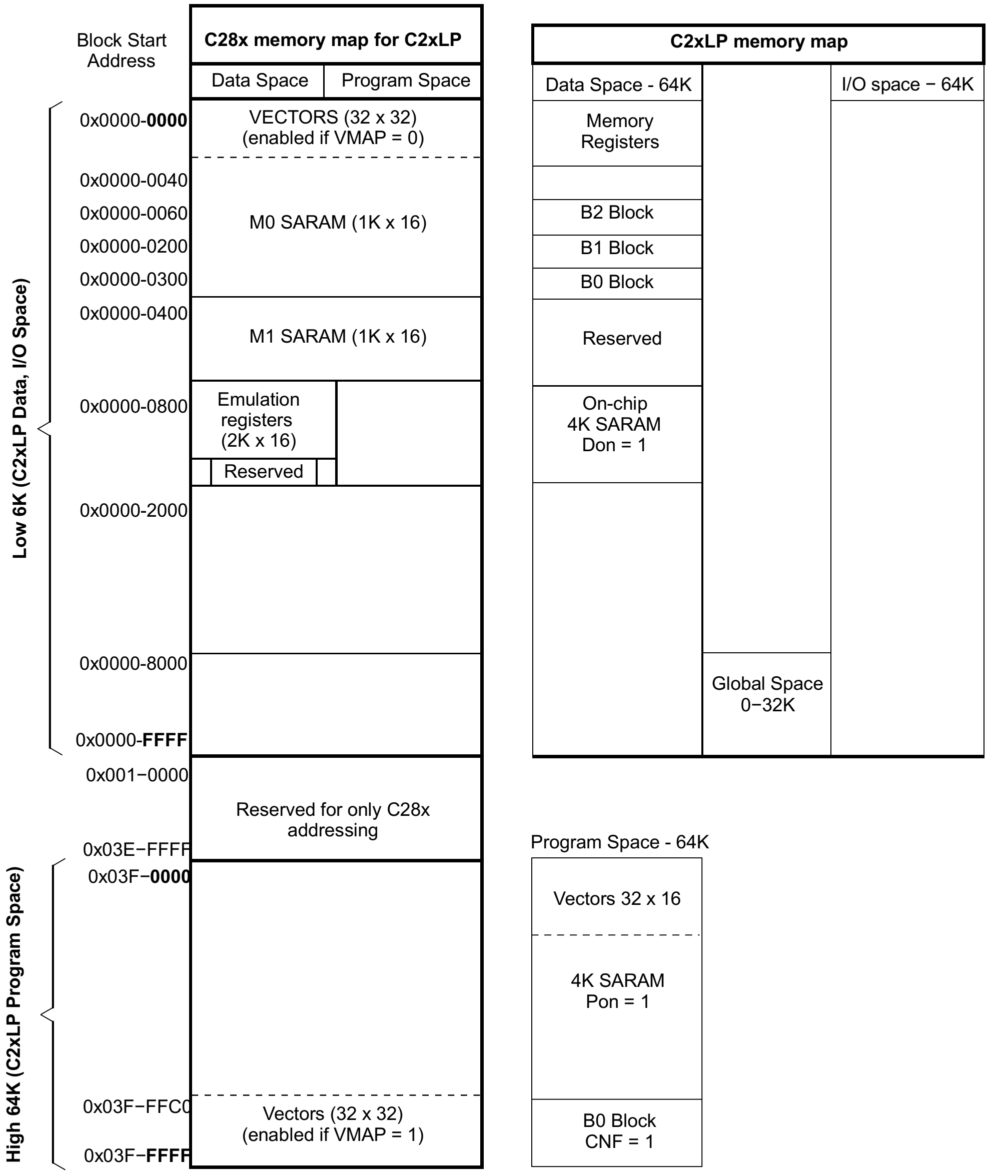
B.3 Memory Map

The major changes between the C2xLP and C28x memory maps are outlined in this section. There are several differences between the C2xLP and C28x memory maps. These improvements are due to the expanded architecture of the C28x. The C28x CPU memory map ranges from 4G to 4M in data and program memory, respectively. However, C28x CPU-based devices may not use the entire memory range. See the device data sheet for the specific memory range applicable to that device.

Vectors. On the C2xLP, only one vector table is present at address 0x0000. These vectors were generally branch instructions to different interrupt service routines. On the C28x, the vector table can be placed in two different locations depending on the state of the VMAP input pin. On devices that do not pin out the VMAP signal, it is tied internal to the device. Generally, vectors will be located in non-volatile memory at 0x3FFFC0−0x3FFFFF. To take advantage of relocatable vectors or fetching vectors from fast internal memory space, place the vectors at address 0x000000−0x00003F. Often the C28x CPU interrupt vectors are expanded using external hardware logic. In such cases, see the related documents for the expanded vector map.

Memory space. On the C2xLP, the memory space for program, data, and I/O space is each 64K words. On the C28x, the program memory space is 4M words (22 address signals). The data memory space is 4G words (32 address signals). The global space (32K) and I/O space (64K) is generally used for C2xLP compatibility.

Program space. On the C2xLP CPU, program space could be mapped anywhere from (0x0−0xFFFF). With the extended address reach of the C28x (22 bits), the compatible region in program space for the C2xLP is 0x3F0000−0x3FFFF. Thus, any program memory on the C2xLP must be remapped to this upper region on the C28x. When the processor accesses program memory, the upper bits (bits 16−22) will be forced to all 1’s when C2xLP- compatible instructions are used (See Appendix E).



A Memory map is not to scale.

Figure B-7. Memory Map Comparison (See Note A)

Data memory. The C2xLP has three internal memory regions (B0, B1, B2) totaling 544 words. The C28x has two internal memory regions (M0,M1) totaling 1K words each. Note that for strict C2xLP compatibility, the memory regions are placed at the same addresses as noted in Table B-6.

Table B-6. B0 Memory Map

C28x in C2xLP - Compatible Mode	C2xLP
CNF Not Available	CNF = 0
B0 range mapped in M0 block 200 − 2FFh. (No mirroring of the block)	B0 in Data space 100 − 1FFh (mirrored locations) 200 − 2FFh
CNF Not Available	CNF = 1
B0 range cannot be enabled in C2xLP-equivalent program memory	B0 in program space FE00 − FEFFh (mirrored locations) FF00 − FFFFh

I/O space. I/O space has remained on the C28x for compatibility reasons, and can only be accessed using IN and OUT/UOUT instructions. Not all C28x devices will support I/O space. See the data sheet of your particular device for details.

Global space. Global space is not supported on all C28x devices. See the data sheet specific to your device for details.

Reserved memory. Reserved memory regions have changed on the C28x. No user-defined memory or peripherals are allowed at addresses 0x800−0x9FF on the C28x. While using C2xLP-compatible mode, these addresses are reserved. It is recommended that C2xLP memory or peripherals be relocated to avoid memory conflicts.

Stack space. The C28x has a dedicated software stack pointer. This pointer is initialized to address 0x0400 (the beginning of block M1) at reset, and it grows upward in address. It is up to the user to move this stack pointer if needed in firmware.

C2xLP Migration Guidelines

The C28x DSP is source-code compatible with C2xLP DSP based devices. The C28x DSP assembler accepts all C2xLP mnemonics with the exception of a few instructions. This chapter provides guidelines for C2xLP code migration to a C28x device. C2xLP refers to the CPU used in all TMS320C24x, TMS320C24xx, and TMS320C20x DSP devices.

Topic **Page**

C.1 Introduction

This chapter provides guidelines that are intended for conversion from C2xLP assembly source to C28x object code. The conversion steps highlight the architectural changes between C2xLP and C28x operating modes. Future releases of documents will contain code conversion examples and software library modules facilitating the conversion from C2xLP mixed C and assembly source to C28x object code.

This chapter will be best understood if the reader has prior knowledge of Appendix C and Appendix E, as they explain the architectural and instructional enhancements between the C2xLP and C28x DSPs.

C.2 Recommended Migration Flow

Use the following steps (shown in Figure C-1) to migrate code:

1. Install the latest development tools for the C28x DSP (e.g. Code Composer StudioTM version 2.x or higher)
2. Build the project with following C28x assembler options:

```
−m20     ; enable C2xLP instructions
− g      ; enable source level debug to view the C2xLP
         ; instructions
−mw      ; enable additional assembly checks
```

   Code Composer Studio 2.x will assemble all C2xLP instructions and map all the compatible instructions to their equivalent C28x instructions and mnemonics. Code Composer Studio 2.x disassembly will display the instructions in the memory as C28x mnemonics only. If the source is built with –g option, the relevant C2xLP source file will be also displayed and will facilitate C2xLP instruction readability during debug.

3. **Memory map:**

   Define your C28x device memory map with C2xLP compatible memory sections. Build a linker command file (*.cmd). See Table C-8.

   Select a C2xLP assembly source code *.asm for migration to C28x architecture.

4. **Boot Code:**

   Add the C2xLP mode conversion code segment shown in Section C.4.1 as the first set of instructions after reset.

   After reset, the C28x powers up in C27x object−compatible mode. Adding these few lines of initialization code will place the device in the proper operating mode for executing reassembled C2xLP code.

NOTE: The C27x object-compatible mode is for use only for migration from the C27x CPU. It is a reserved operating mode for all C28x and C2xLP applications.

5. This step will facilitate faster code conversion. In the C2xLP source file modify the interrupt section with suggestions from the reference table in section D.5.
   In particular, modify the following types of code:
   (a) **IMR** and **IFR** − See the example code in Section C.4.2.
   (b) **Context Save/Restore** − See the example code in Section C.4.3
   (c) Comment all the known incompatible instructions or map with equivalent instructions. See Table D-2.

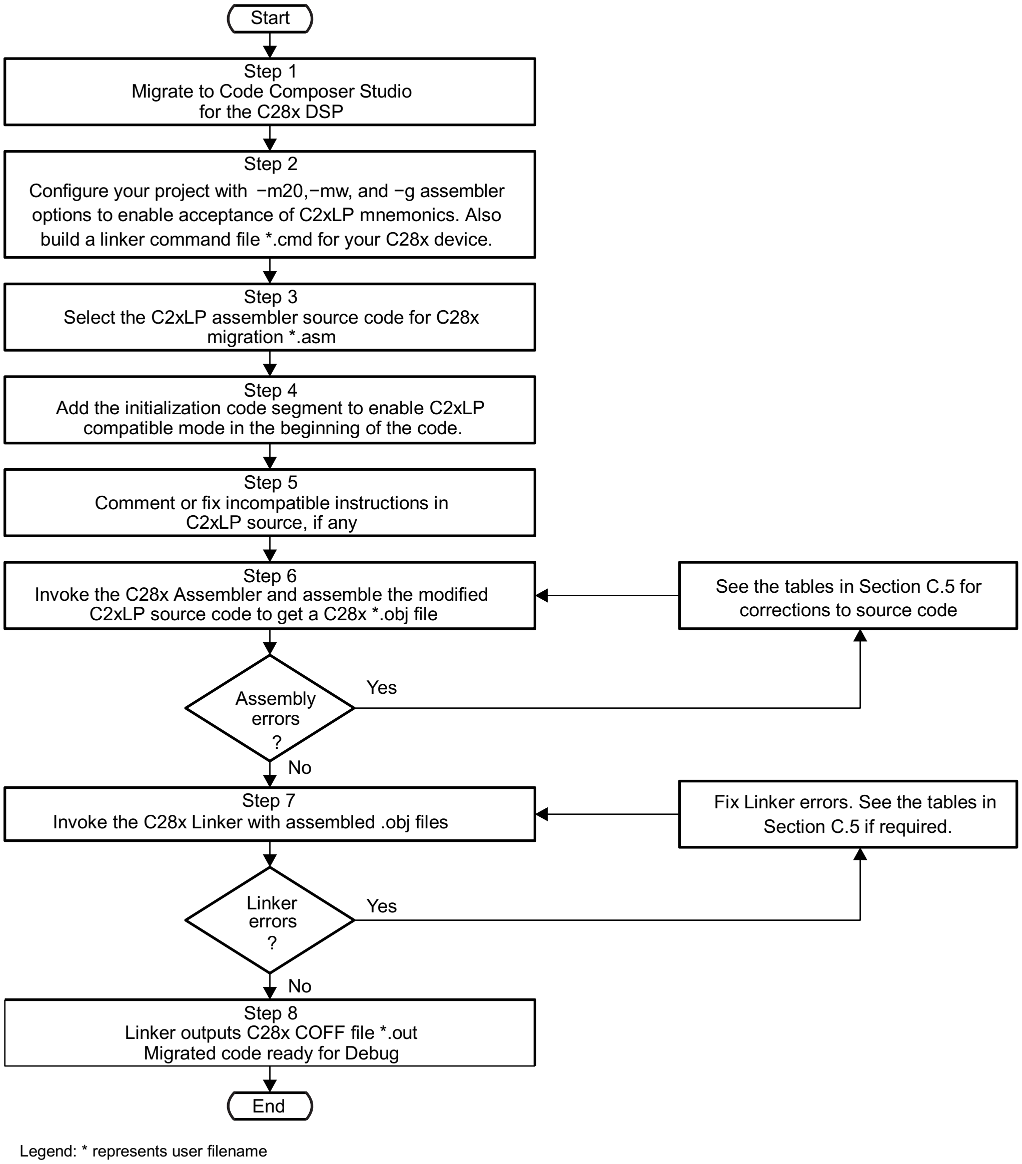

Start
Step 1
Migrate to Code Composer Studio
for the C28x DSP
Step 2
Configure your project with −m20,−mw, and −g assembler
options to enable acceptance of C2xLP mnemonics. Also
build a linker command file *.cmd for your C28x device.
Step 3
Select the C2xLP assembler source code for C28x
migration *.asm
Step 4
Add the initialization code segment to enable C2xLP
compatible mode in the beginning of the code.
Step 5
Comment or fix incompatible instructions in
C2xLP source, if any
Step 6
Invoke the C28x Assembler and assemble the modified
C2xLP source code to get a C28x *.obj file
See the tables in Section C.5 for
corrections to source code
Assembly
errors
?
Yes
No
Step 7
Invoke the C28x Linker with assembled .obj files
Fix Linker errors. See the tables in
Section C.5 if required.
Linker
errors
?
Yes
No
Step 8
Linker outputs C28x COFF file *.out
Migrated code ready for Debug
End
Legend: * represents user filename

Figure C-1. Flow Chart of Recommended Migration Steps

6. Link the assembled code with the linker command file generated in Step 2. Relink if necessary to avoid any linker related errors.
7. Assemble or reassemble using the C28x assembler until the assembly is successful with no errors. The tables in Section C.5 will help to resolve most of the errors during the assembly process. This will prepare a ***.obj** file, ready for C28x Linker processing.
8. The Linker output COFF file, ***.out**, will be the migrated code and should be ready for Debug and integration.

C.3 Mixing C2xLP and C28x Assembly

At this point your original C2xLP code will be running on the C28x device. To facilitate further migration to C28x code, there are special assembler directives that will facilitate mixing of C2xLP code and C28x code segments.

The .c28_amode and .lp_amode directives tell the assembler to override the assembler mode.

.c28_amode— The .c28_amode directive tells the assembler to operate in the C28x object mode (−v28).

.lp_amode —The .lp_amode directive tells the assembler to operate in C28x object − accept C2xLP syntax mode (−m20).

These directives can be repeated throughout a source file.

For example, if a file is assembled with the −m20 option, the assembler begins the assembly in the C28x object − accept C2xLP syntax mode. When it encounters the .c28_amode directive, it changes the mode to C28x object mode and remains in that mode until it encounters an .lp_amode directive or the end of file.

EXAMPLE

In this example, C28x code is inserted in the existing C2xLP code.

```
; C2xLP source code
.lp_amode
LDP #VarA
LACL VarA
LAR AR0 *+, AR2
SACL *+
.
.
CALL FuncA
.
.
; The C2xLP code in function FuncA is replaced with C28x Code
; using C28x addressing (AMODE = 0)

.c28_amode ; Override the assembler mode to C28x syntax
FuncA:
    C28ADDR ; Set AMODE to 0 C28x addressing
    MOV DP, #VarB
    MOV AL, @VarB
    MOVL XAR0, *XAR0++
    MOV *XAR2++, AL
    .lp_amode    ; Change back the assembler mode to C2xLP.
    LPADDR       ; Set AMODE to 1 to resume C2xLP addressing.
    LRET
```

C.4 Code Examples

C.4.1 Boot Code for C28x Operating Initialization

NOTE: The following code fragment must be placed in your code just after reset. This code will place the device in the proper operating mode to execute C2xLP converted code:

```
Code Explanation
SETC OBJMODE          ;C28OBJ = 1 enable 28x object mode
CLRC PAGE0            ;PAGE0 = 0 not relevant for 28x mode,
                      ;cleared to zero
SETC AMODE            ;AMODE = 1 enable C2xLP compatible
                      ;addressing mode
SETC SXM              ;SXM = 1 for C2xLP at reset, SXM = 0
                      ;for 28x at reset
SETC C                ;Carry bit =1 for C2xLP at reset,
                      ;Carry bit = 0 for 28x at reset
SPM 0                 ;Set product shift mode zero, that is PM bits = 001
                      ;compatible to C2xLP PM reset;mode
```

C.4.2 IER/IFR Code

Table C-1. Code to Save Contents Of IMR (IER) And Disabling Lower Priority Interrupts At Beginning Of ISR

C2xLP	C28x
INTx: . MAR *,AR1 LDP #0 LACL IMR SACL *+ AND #~INT_MASK SACL IMR . .	INTx: . AND IER,#~INT_MASK . Note: C28x saves IER as part of automatic context save operation and disables the current interrupt automatically to prevent recursive interrupts.

Table C-2. Code to Disable an Interrupt

C2xLP	C28x
SETC INTM LDP #0 LACL IMR AND #~INTx SACL IMR CLRC INTM	AND IER,#~INTx ;operation is atomic and ;will not be interrupted.

Table C-3. Code to Enable an Interrupt

C2xLP	C28x
SETC INTM LDP #0 LACL IMR OR #INTx SACL IMR CLRC INTM	;write 0 to clear AND IFR,#~INTx ;operation is atomic and ;will not be interrupted

Table C-4. Code to Clear the IFR Register

C2xLP	C28x
;write 1 to clear SETC INTM LDP #0 SPLK #0FFFFh,IFR CLRC INTM	;write 0 to clear AND IFR,#~INTx ;operation is atomic and ;will not be interrupted

C.4.3 Context Save/Restore

The C28x automatically saves a number of registers on each interrupt. To perform a full context save, some additional code must be added. Table C-5 shows a typical full context save and restore for both processors.

Table C-5. Full Context Save/Restore Comparison

C2xLP Full Context Save/Restore	C28x Full Context Save/Restore
INTx_ISR: ; context save MAR *, AR1 MAR *+ SST #1,*+ SST #0,*+ SACH *+ SACL *+ SPH *+ SPL *+ MPY #1 SPL *+ SAR AR0, *+ SAR AR2, *+ SAR AR3, *+ SAR AR4, *+ SAR AR5, *+ SAR AR6, *+ SAR AR7, *+ . ;interrupt code goes here . . ; context restore MAR *, AR1 MAR *– LAR AR7, *– LAR AR6, *– LAR AR5, *– LAR AR4, *– LAR AR3, *– LAR AR2, *– LAR AR0, *– SETC INTM MAR *– SPM 0 LT *+ MPY #1 LT *– MAR *– LPH *– LACL *– ADD *–, 16 LST #0, *– LST #1, *– CLRC INTM RET	;C28x automatically saves the ;following registers: ;T,ST0,AH,AL,PH,PL,AR1,AR0,DP,ST1, ;DBGSTAT,IER,PC INTx_ISR: ;interrupt context save PUSH AR1H:AR0H ; 32–bit PUSH XAR2 ; 32–bit PUSH XAR3 ; 32–bit PUSH XAR4 ; 32–bit PUSH XAR5 ; 32–bit PUSH XAR6 ; 32–bit PUSH XAR7 ; 32–bit PUSH XT ; 32–bit . . ;interrupt code goes here . . ;interrupt context restore POP XT POP XAR7 POP XAR6 POP XAR5 POP XAR4 POP XAR3 POP XAR2 POP AR1H:AR0H IRET

C.5 Reference Tables for C2xLP Code Migration Topics

Table C-6 through Table C-10 explain the major differences between the C2xLP and C28x architectures and in their respective code generation process. These tables are organized to highlight the differences in interrupts, CPU registers, memory maps, instructions, registers, and syntax. While migrating the C2xLP code, check the tables for these key differences to make the necessary changes to the source to avoid assembler or linker errors.

Table C-6. C2xLP and C28x Differences in Interrupts

Migration topic		C2xLP	C28x
1	Interrupt flag register	IFR − Memory mapped register Write 1 to clear bits set in IFR	IFR is a CPU register Write 0 to clear bits set in IFR
2	Interrupt enable register	IMR – Memory mapped register	Renamed as IER and is a CPU register
3	TRAP instruction	Only one TRAP vector TRAP Affects: INTM bit is not affected	multiple,32− TRAP vectors TRAP 0, .. TRAP31 Affects: INTM bit is set to 1
4	INTR instruction syntax	INTR0 ... INTR31 Affects: IFR not cleared IMR not affected INTM bit =1	INTR INT0 ... INTR INT31 Affects: IFR cleared IER affected INTM bit =1
5	NMI Instruction	NMI	TRAP NMI
6	CLRC INTM instruction	CLRC INTM instruction blocks all interrupts until the next instruction is executed. CLRC INTM next_instn ;interrupts ;blocked ;until this ;executed	Interrupts enabled after the instruction CLRC INTM
7	Interrupt enable and return from interrupt service	CLRC INTM RET	IRET
8	Interrupt enable and return from function call	CLRC INTM next_instn	next_instn CLRC INTM
9	Interrupts Vector	Uses Branch statements at the vector address. Ex: B Start ;assembly ;code ; opcode in memory 0x7980 ;branch ;instruction 0x0040 ;branch ;address	32−bit absolute addresses. ; code in vector location 0x0040 (low address) 0x003F (high address)
10	Context save	No automatic context save See Section D.3 for a full context save/restore example	Automatic context save of CPU registers T, ST0, AH, AL, PH, PL, AR1, AR0, DP, ST1, DBGSTAT, IER, PC See Table C-5 for a full context save/ restore example

Table C-7. C2xLP and C28x Differences in Status Registers

Migration topic		C2xLP	C28x

1	Saving ST0/ST1 registers	Save:
		`SST #0,mem ;store ST0`
		`SST #1,mem ;store ST1`
		`Restore:`
		`LST #0,mem ;load ST0`
		`LST #1,mem ;load ST1`
			Save:
			`PUSH ST ;store ST0 to stack`
			`PUSH ST ;store ST1 to stack`
			`Restore:`
			`POP ST1 ;load ST1`
			`        ;from stack`
			`POP ST0 ;load ST0`
			`        ;from stack`

2	ST0/ST1 bit differences	ST0/ST1 bits have CPU registers and status bits	ST0/ST1 bits are rearranged compared to C2xLP registers.
3	INTM bit in ST0	Cannot be saved if ST0 register is saved	Saved along with ST0 register

4	Data page pointer
	DP save
		DP save/restored along with ST0.
		`SST #0,mem ;store ST0`
		`LST #0,mem ;load ST0`
			DP is a register, hence explicit store/restore is required.
			`PUSH DP     ;store DP`
			`            ;to stack`
			`PUSH DP:ST1 ; 32-bit`
			`            ; save`
			`POP    DP   ;load DP from`
			`            ;stack`
			`POP DP:ST1  ; 32-bit`
			`            ; restore`

Table C-8. C2xLP and C28x Differences in Memory Maps

Migration topic		C2xLP	C28x
1	Program memory	16−bit address Size : 64kx16 Range :0x0000−0xFFFFh	22 – bit address Size : 64kx16 mapped to Range: 0x3F 0000h – 0x3F FFFFh
2	Data memory	Size : 64kx16 mapped to Range: 0x00 0000h – 0x00 FFFFh	Size : 64kx16 Range :0x0000−0xFFFFh
6	B2 Block	Size: 32 words Range: 0x0060−0x007F	Located in M0 Block 1Kx16 Size: 1K words Range: 0x00 0060 –0x00 07Fh
7	B1 Block	Size: 256 words Range: 0x0100−0x01FF (mirrored) : 0x0200−0x02FF	Located in M0 Block − 1Kx16 Not Mirrored Range: 0x00 0200 –0x00 02FFh
8	B0 Block	Mirrored locations Size: 256 words Range: 0x0300−0x03FF : 0x0400−0x04FF	Located in M0 Block − 1Kx16 Not Mirrored Range: 0x00 0300 –0x00 03FFh
9	CNF bit mapping of B0 Block	CNF bit maps B0 in data and program memory CNF =0 − B0 in data memory Range: 0x0300−0x03FF : 0x0400−0x04FF CNF =1 − B0 in program memory Range: 0xFE00−0xFEFF : 0xFF00−0xFFFF	Not applicable
10	Vector table range	Size: 32x16 words Range: 0x0000−0x003F	Size 32x32 words 0x3F FFC0 – 0x3F FFFF − at reset In C28x based DSP devices may use additional expanded vector table (e.g., PIE)
11	Internal SARAM mapping in data memory	Mapped as internal memory map	Reserved for emulation registers Range : 0x0800 −0x1000h
5	I/O space	Range: 0x0000 −0xFFFFh	Range: 0x0x00 000 −0x00 FFFFh I/O Space may or may not be implemented on a particular device. See the device datasheet for details.
6	Global space	Range: 0x8000 −0xFFFFh	Implemented via the XINTF Global Space may or may not be implemented on a specific C28x device. See the device datasheet for details.

Table C-9. C2xLP and C28x Differences in Instructions and Registers

Migration topic		C2xLP	C28x
1	Conditional Instructions Branches, Calls, Returns	Can take more than one condition in these instructions	The C28x assembler will automatically break the instructions into multiple instructions.
2	When are CPU Flags updated?	Conditional flags update on Accumulator operation only	Conditional flags update on Accumulator, register and memory operations
3	Repeat instructions	Many instructions are repeatable	Same instructions are repeatable. For additional repeatable instructions see Table D-3.
4	GREG register	Memory mapped register	Memory mapped register in XINTF Global Space may or may not be implemented on a particular device. See the device data sheet for details.

5	ARx registers	ARx registers are 16−bit only
		`LAR    AR1, #0FFFFh`
		`ADRK #1`
		Result:
		AR1 = 0x0000h
			XARn registers are 32 bits. Some instructions access only the lower 16 bits known as ARn
			`MOV XAR1, #0FFFFh`
			`ADD    XAR1,#1`
			Result:
			XAR1 = 0x10000h

6	2s complement subtraction to ARx	`LAR    AR1, #0FFFFh`
		`ADRK #0FE`
		Result:
		AR1 = 0xFFFDh
			`MOV XAR1, #0FFFFh`
			`ADD    XAR1,#0FE`
			Result:
			XAR1 = 0x1FFFDh

7	I/O instructions	Supports IN, OUT instructions	Supports IN, OUT,UOUT I/O Space may or may not be implemented on a particular device. See the device datasheet for details.
8	Stack	Uses 8−deep Hardware stack C2xLP Compiler uses AR1 as Stack Pointer	Uses software stack pointer register (SP) Compiler will use SP register, as stack pointer
9	Program counter	16 bits in size B 5000h ; Branch to 5000 ; address	22 bits in size The C28x assembler will use special C2xLP compatible instructions that force the upper program address lines to 0x3F thus creating a 16−bit C2xLP compatible PC. B 0x3F5000 ; or XB 5000h

Table C-10. Code Generation Tools and Syntax Differences

Migration topic		C2xLP	C28x
1	Mnemonic	Source or destination not always specified. LACL, source SACL, destination	Instructions are always of the form mnemonic destination, source MOV destination,source
2	Direct addressing syntax −@ symbol	LACL dma	MOV ACC, @@dma ; C2xLP mode MOV ACC, @dma ; 28x mode @@ − means 128 word data page @ − means 64 word data page
3	Indirect address pointer buffer, ARB	In indirect addressing, Auxiliary register will be pointed by ARP register in ST0. ARB is ARP pointer buffer in ST1. MAR *,AR2 ; ARP =AR2 LACL *	No ARB equivalent in 28x. Selected ARx is referenced in the instruction itself. MOV ACC,*AR2
4	New Address pointers syntax − *(0	BLDD #4545h,RegA	MOV @REGA, *(0:0x4545)

5	Repeat instructions
	syntax change − ||
		No additional syntax
		`RPT #5`
		`NOP`
			Uses || syntax with repeat instructions
			`RPT #5`
			`|| NOP`

6	Reserved register names Application code should not use these reserved words	ST0, ST1, IFR, IMR, GREG	ST0, ST1, AH, AL, PH, PL,T, TL, XAR0, XAR1, XAR2, XAR3, XAR4, XAR5, XAR6, XAR7, DP, ST1, DBGSTAT, IER, PC, RPC
7	Increment/Decrement syntax change	MAR *,AR2 LACL *+ …. LACL *−	MOV ACC, *AR2++ ….. MOV ACC, *AR2−−
8	Shift syntax change	LACL dma, 4	MOV ACC,dma <<4

9	Number radix usage	`x    .set 09    ;Assembler`
		`;accepts`
		`;this as`
		`;decimal 9`
			x .set 09
			Avoid leading zeros, else the assembler will be use this as octal number.

10	Order of precedence in expressions – Syntax change	Expressions in assembly statements do not require parenthesis. `x.set A<<B = C>>D`	Expressions in assembly statements do require parenthesis. `x.set (A<<B = C>>D)`
11	Tools Directives	.mmregs ; reserved register use .port .globl	not applicable not applicable .global
12	Macros	Useful in coding style	Useful in coding style All C2xLP Macros are not directly used Convert them individually to 28x mode.
13	Assembler options	−v2xx	−m20, −v28

C2xLP Instruction Set Compatibility

This appendix highlights the differences in syntax between the C2xLP and the C28x instructions, and details which C2xLP compatible instructions are repeatable on the C28x. The C28x assembler accepts both C28x and C2xLP assembly source syntax. This enables you to quickly port C2xLP code with minimal effort. Additionally, all compatible C2xLP instructions have an equivalent C28x style syntax. The C28x disassembler will show the C28x equivalent syntax.

Topic **Page**

D.1 Condition Tests on Flags

On the C28x, all EQ/NEQ/GT/LT/LEQ conditional tests are performed on the state of the Z and N flags. On the C2xLP, the same condition tests are performed on the contents of the ACC register.

Table D-1. C28x and C2xLP Flags

Designation	C28x Modes	C2xLP Equivalent
NEQ	!= 0	ACC != 0
EQ	== 0	ACC == 0
GT	> 0	ACC > 0
GEQ	>= 0	ACC >= 0
LT	< 0	ACC < 0
LEQ	<= 0	ACC <= 0
HI	higher	–
HIS, C	higher or same, carry set	C == 1
LO, NC	lower, carry clear	C == 0
LOS	lower or same	–
NOV	no overflow	OV == 0
OV	overflow	OV == 1
NTC	TC == 0	TC == 0
TC	TC == 1	TC == 1
NBIO	test BIO input == 0	BIO == 0
UNC	unconditional	UNC

On the C28x, the Z and N flags are set on all ACC operations. That includes ACC loads. Therefore, the Z and N flags reflect the current state of the ACC immediately after an operation on the ACC.

NOTE: The NBIO condition requires an external pin that is only present on TMS320x2801x devices.

D.2 C2xLP vs. C28x Mnemonics

Table D-2 lists the C2xLP instructions with the C28x equivalent syntax. The C28x assembler will accept either the C2xLP syntax or the equivalent C28x syntax. The disassembler will decode and display the C28x syntax.

The C2xLP cycle count numbers shown are for zero wait-state internal memory, where n equals the number of repetitions (i.e., if an instruction is repeated, using the RPT instruction for repeatable instructions, n times it is executed n+1 times).

Table D-2. C2xLP Instructions and C28x Equivalent Instructions

C2xLP				C28x			
Instruction	Mnemonic	Cycles	Size	Instruction	Mnemonic	Cycles	Size
ABS		n+1	16	ABS	ACC	1	16
ADD	loc16[,0]	n+1	16	ADD	ACC,loc16 {<<0}	n+1	16
ADD	loc16,1..15	n+1	16	ADD	ACC,loc16 << 1..15	n+1	32
ADD	loc16,16	n+1	16	ADD	ACC,loc16 << 16	n+1	16
ADD	#8bit	1	16	ADDB	ACC,#8bit	1	16
ADD	#16bit[,0..15]	2	32	ADD	ACC,#16bit {<<0..15}	1	32
ADDC	loc16	n+1	16	ADDCU	ACC,loc16	1	16
ADDS	loc16	n+1	16	ADDU	ACC,loc16	n+1	16
ADDT	loc16	n+1	16	ADD	ACC,loc16 << T	n+1	32
ADRK	#8bit	1	16	ADRK	#8bit	1	16

Table D-2. C2xLP Instructions and C28x Equivalent Instructions (continued)

C2xLP				C28x			
Instruction	Mnemonic	Cycles	Size	Instruction	Mnemonic	Cycles	Size
AND	loc16	n+1	16	AND	ACC,loc16	n+1	16
AND	#16bit,16	2	32	AND	ACC,#16bit<<16	1	32
AND	#16bit[,0..15]	2	32	AND	ACC,loc16 {<< 0..15}	1	32
APAC		n+1	16	ADDL	ACC,P<<PM	n+1	16
B	pma	4	32	XB	pma,UNC	7	32
B	pma,*,ARn	4	32	XB	pma,*,ARPn	4	32
B	pma,*ind	4	32	NOP XB	*ind pma, UNC	8	32
B	pma,*ind,ARn	4	32	NOP XB	*ind pma,*,ARPn	5	48
BACC		4	16	XB	*AL	7	16
BANZ	pma,*ind[,ARn]	4/2	32	XBANZ	pma,*ind[,ARAPn]	4/2	32
BANZ	pma,*BR0+/*BR0−[,ARn]	4/2	32	Not applicable			
BCND	pma[,COND]	4/2	32	XB or	pma,COND	7/4	32
				SB	#8bitOff,COND		16
BCND	pma,COND1,COND2,.., CONDn	4/2	32	SB SB . XB skip:	skip,opposite of COND1 skip,opposite of COND2 . pma,CONDn	7+	48+
BIT	loc16,15−bit	n+1	16	TBIT	loc16,#bit	1	16
BITT	loc16	n+1	16	TBIT	loc16,T	1	32
BLDD	#src_addr,loc16	n+3	32	MOV	loc16,*(0:src_addr)	n+2	32
BLDD	loc16,#dest_addr	n+3	32	MOV	*(0:dest_addr),loc16	n+2	32
BLPD	#pma,loc16	n+3	32	XPREAD	loc16,*(pma)	n+2	32
CALA		4	16	XCALL	*AL	7	16
CALL	pma	4	32	XCALL	pma,UNC	7	32
CALL	pma,*,ARn	4	32	XCALL	pma,*,ARPn	4	32
CALL	pma,*ind	4	32	NOP XCALL	*ind pma,UNC	8	48
CALL	pma,*ind,ARn	4	32	NOP XCALL	*ind pma,*,ARPn	5	48
CC	pma,COND	4/2	32	XCALL	pma,COND	7/4	32
CC	pma,COND1,..,CONDn	4/2	32	SB SB . XCALL skip:	skip,opposite of COND1 skip,opposite of COND2 pma,CONDn	7+	48+
CLRC	INTM	n+1	16				
CLRC	XF/OVM/SXM/TC/C	n+1	16	CLRC	XF/OVM/SXM/TC/C	2,1	16
CLRC	CNF	n+1	16	Not applicable			
CMPL		n+1	16	NOT	ACC	1	16
CMPR	0/1/2/3	n+1	16	CMPR	0/1/2/3	1	16
DMOV	loc16	n+1	16	DMOV	loc16	n+1	16
IDLE		1	16	IDLE		5	16
IN	loc16,PA	2(n+1)	32	IN	loc16,*(PA)	n+2	32
INTR	K	4	16	Not applicable			

Table D-2. C2xLP Instructions and C28x Equivalent Instructions (continued)

C2xLP				C28x			
Instruction	**Mnemonic**	**Cycles**	**Size**	**Instruction**	**Mnemonic**	**Cycles**	**Size**
LACC	loc16[,0]	n+1	16	MOV	ACC,loc16 [<< 0]	1	16
LACC	loc16,1..15	n+1	16	MOV	ACC,loc16 << 1..15	1	32
LACC	loc16,16	n+1	16	MOV	ACC,loc16 << 16	1	16
LACC	#16bit,0..15	2	32	MOV	ACC,#16bit << 0..15	1	32
LACL	loc16	n+1	16	MOVU	ACC,loc16	1	16
LACL	#8bit	1	16	MOVB	ACC,#8bit	1	16
LACT	loc16	n+1	16	MOV	ACC,loc16 << T	1	32
LAR	ARn,loc16	2(n+1)	16	MOVZ	ARn,loc16	1	16
LAR	ARn,#8bit	2	16	MOVB	XARn,#8bit	1	16
LAR	ARn,#16bit	2	32	MOVL	XARn,#22bit	1	32
LDP	loc16	2(n+1)	16	Not applicable			
LDP	#9bit	2	16	MOVZ	DP,#10bit >> 1	1	16
LPH	loc16	n+1	16	MOV	PH,loc16	1	16
LST	#0/1,loc16	2(n+1) 16					
LT	loc16	n+1	16	MOV	T,loc16	1	16
LTA	loc16	n+1	16	MOVA	T,loc16	n+1	16
LTD	loc16	n+1	16	MOVAD	T,loc16	1	16
LTP	loc16	n+1	16	MOVP	T,loc16	1	16
LTS	loc16	n+1	16	MOVS	T,loc16	n+1	16
MAC	pma,loc16	n+3	32	XMAC	P,loc16,*(pma)	n+2	32
MACD	pma,loc16	n+3	32	XMACD	P,loc16,*(pma)	n+2	32
MAR	*ind[,ARn]	n+1	16	NOP	*ind[,ARPn]	n+1	16
MPY	loc16	n+1	16	MPY	P,T,loc16	1	16
MPY	#13bit	1	16	MPY	P,@T,#16bit	1	32
MPYA	loc16	n+1	16	MPYA	P,T,loc16	n+1	16
MPYS	loc16	n+1	16	MPYS	P,T,loc16	n+1	16
MPYU	loc16	n+1	16	MPYU	P,T,loc16	1	16
NEG		n+1	16	NEG	ACC	1	16
NMI		4	16	Not applicable			
NOP		n+1	16	NOP		n+1	16
NORM	*/*+/*−/*0+/*0−	n+1	16	NORM	*BR0+/*BR0−	n+l	16
NORM	ACC,*/*++/*−−/*0++/*0−−	n+4	16	Not applicable			
OR	loc16	n+1	16	OR	ACC,loc16	n+1	16
OR	#16bit,16	2	32	OR	ACC,#16bit<<16	1	32
OR	#16bit[,0..15]	2	32	OR	ACC,#16bit {<< 0..15}	1	32
OUT	loc16,PA	3(n+1)	32	OUT	*(PA),loc16	4	32
PAC		n+1	16	MOV	ACC,P<<PM	1	16
POP		n+1	16	MOVU	ACC,*−−SP	1	16
POPD	loc16	n+1	16	POP	loc16	2	16
PSHD	loc16	n+1	16	PUSH	loc16	2	16
PUSH		n+1	16	MOV	*SP++,AL	n+1	16
RET		4	16	XRETC	UNC	7	16
RETC	COND	4/2[1]		XRETC	COND	7/4	16

[1] True/False

Table D-2. C2xLP Instructions and C28x Equivalent Instructions (continued)

C2xLP				C28x			
Instruction	Mnemonic	Cycles	Size	Instruction	Mnemonic	Cycles	Size
RETC	COND1,COND2,.., CONDn	4/2	16	SB SB . XRETC $10:	$10,opposite of COND1 $10,opposite of COND2 . CONDn	7+	48+
ROL		n+1	16	ROL	ACC	n+1	16
ROR		n+1	16	ROR	ACC	n+1	16
RPT	loc16	1	16	RPT	loc16	1	16
RPT	#8bit	1	16	RPT	#8bit	1	16
SACH	loc16[,0]	n+1	16	MOV	loc16,AH	n+1	16
SACH	loc16,1	n+1	16	MOVH	loc16,ACC << 1	n+1	16
SACH	loc16,2..7	n+1	16	MOVH	loc16,ACC << 2..7	n+1	32
SACL	loc16[,0]	n+1	16	MOV	loc16,AL	n+1	16
SACL	loc16,1	n+1	16	MOV	loc16,ACC << 1	n+1	16
SACL	loc16,2..7	n+1	16	MOV	loc16,ACC << 2..7	n+1	32
SAR	ARn,loc16	n+1	16	MOV	loc16,ARn	1	16
SBRK	#8bit	1	16	SBRK	#8bit	1	16
SETC	INTM	n+1	16	SETC	INTM	2	16
SETC	XF/OVM/SXM/TC/C	n+1	16	SETC	XF/OVM/SXM/TC/C	2,1	16
SETC	CNF	n+1	16	Not applicable			
SFL		n+1	16	LSL	ACC,1	n+1	16
SFR		n+1	16	SFR	ACC,1	n+1	16
SPAC		n+1	16	SUB	ACC,P<<PM	n+1	16
SPH	loc16	n+1	16	MOVH	loc16,P	n+1	16
SPL	loc16	n+1	16	MOV	loc16,P	n+1	16
SPLK	#0x0000,loc16	2	32	MOV	loc16,#0	n+1	16
SPLK	#16bit,loc16	2	32	MOV	loc16,#16bit	n+1	32
SPM	0	1	16	SPM	0	1	16
SPM	1	1	16	SPM	1 (or +1)	1	16
SPM	2	1	16	SPM	2 (or +4)	1	16
SPM	3	1	16	SPM	3 (or −6)	1	16
SQRA	loc16	n+1	16	SQRA	loc16	n+1	32
SQRS	loc16	n+1	16	SQRS	loc16	n+1	32
SST	#0/1,loc16	n+1	16	Not applicable			
SUB	loc16[,0]	n+1	16	SUB	ACC,loc16 {<< 0}	n+1	16
SUB	loc16,1..15	n+1	16	SUB	ACC,loc16 << 1..15	n+1	32
SUB	loc16,16	n+1	16	SUB	ACC,loc16 << 16	n+1	16
SUB	#8bit	1	16	SUBB	ACC,#8bit	1	16
SUB	#16bit[,0..15]	2	32	SUB	ACC,#16bit {<< 0..15}	1	32
SUBB	loc16	n+1	16	SUBU	ACC,loc16	1	16
SUBC	loc16	n+1	16	SUBCU	ACC,loc16	n+1	16
SUBS	loc16	n+1	16	SUBU	ACC,loc16	n+1	16
SUBT	loc16	n+1	16	SUB	ACC,loc16 << T	n+1	32
TBLR	loc16	n+3	16	XPREAD	loc16,*AL	n+4	32
TBLW	loc16	n+3	16	XPWRITE	*AL,loc16	n+4	32
TRAP	4	16		Not applicable			

Table D-2. C2xLP Instructions and C28x Equivalent Instructions (continued)

C2xLP				C28x			
Instruction	Mnemonic	Cycles	Size	Instruction	Mnemonic	Cycles	Size
XOR	loc16	n+1	16	XOR	ACC,loc16	n+1	16
XOR	#16bit,16	2	32	XOR	ACC,#16bit<<16	1	32
XOR	#16bit[,0..15]	2	32	XOR	ACC,#16bit [<< 0..15]	1	32
ZALR	loc16	n+1	16	ZALR	ACC,loc16	1	32

D.3 Repeatable Instructions

Not all of the repeatable instructions on the C2xLP are repeatable on the C28x. The ones that were not made repeatable do not make sense to repeat from a functionality standpoint. Also, some instructions that were not repeatable on the C2xLP are repeatable on the C28x.

Table D-3 shows which C2xLP operations are repeatable, and which ones are repeatable on the C28x.

Table D-3. Repeatable Instructions for the C2xLP and C28x

C2xLP Instruction	C2xLP Repeatable	C28x Repeatable
ABS	X	
ADD mem,shift1	X	X
ADDC mem	X	
ADDS mem	X	X
ADDT mem	X	X
AND mem	X	X
APAC	X	X
BIT mem,bit_code	X	
BITT mem	X	
BLDD #addr,mem	X	X
BLDD mem,#addr	X	X
BLPD #pma,mem	X	X
CLRC CNF/XF/INTM/OVM/SXM/TC/C	X	
CMPL	X	
CMPR constant	X	
DMOV mem	X	X
IN mem,PA	X	X
INTR K	X	
LACC mem[,shift1]	X	
LACL mem	X	
LACT mem	X	
LAR AR,mem	X	
LDP mem	X	
LPH mem	X	
LST #n,mem	X	
LT mem	X	
LTA mem	X	X
LTD mem	X	
LTP mem	X	
LTS mem	X	X
MAC pma,mem	X	X
MACD pma,mem	X	X

Table D-3. Repeatable Instructions for the C2xLP and C28x (continued)

C2xLP Instruction	C2xLP Repeatable	C28x Repeatable
MAR {ind}[,nextARP]	X	X
MPY mem	X	
MPY #k	X	
MPYA mem	X	X
MPYS mem	X	X
MPYU mem	X	
NEG X NOP	X	X
NORM {ind}	X	X
OR mem	X	X
OUT mem,PA	X	X
PAC	X	
POP	X	
POPD mem	X	
PSHD mem	X	
PUSH	X	
ROL	X	X
ROR	X	X
SACH mem[,shift]	X	X
SACL mem[,shift]	X	X
SAR AR,mem	X	
SETC CNF/XF/INTM/OVM/SXM/TC/C	X	
SFL	X	X
SFR	X	X
SPAC	X	X
SPH mem	X	X
SPL mem	X	X
SPLK #lk,mem	X	X
SQRA mem	X	X
SQRS mem	X	X
SST #n,mem	X	
SUB mem[,shift1]	X	X
SUBB mem	X	
SUBC mem	X	X
SUBS mem	X	X
SUBT mem	X	X
TBLR mem	X	X
TBLW mem	X	X
XOR mem	X	X
ZALR mem	X	

Migration from C27x to C28x

This appendix highlights the architecture differences between the C27x and the C28x and describes how to migrate your code from a C27x-based design to a C28x-based design.

Topic **Page**

E.1 Architecture Changes

Certain changes to the architecture that are important when migrating from the C27x to the C28x include:

- Changes to registers
- Full context save and restore
- B0/B1 memory map consideration

E.1.1 Changes to Registers

The register modifications from the C27x are shown in Figure E-1. Shaded registers highlight the changes or enhancements for the C28x.

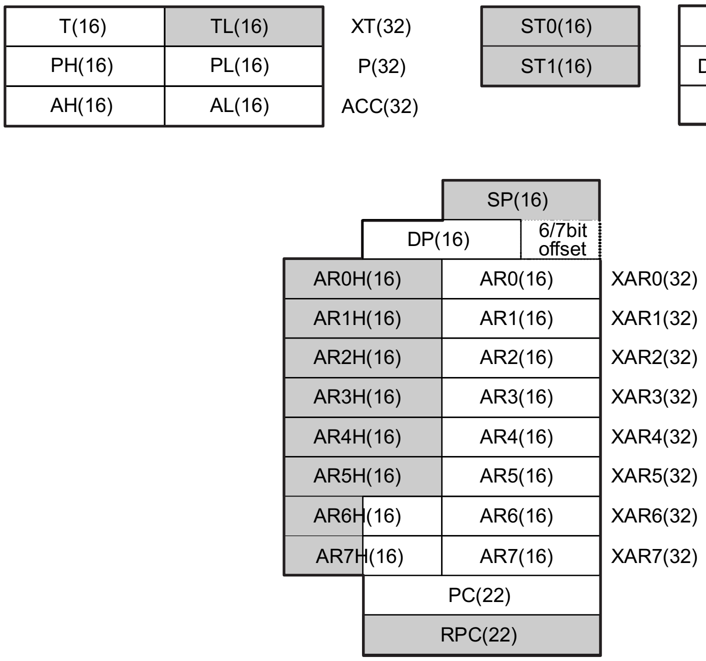


Figure E-1. C28x Registers

A brief description of the register modifications is given below:

XT(32), TL(16) —The T register is increased to 32-bits and called the XT register. The existing C27x T register portion represents the upper 16-bits of the new 32-bit register. The additional 16-bits, called the TL portion, represents the lower 16-bits.

XAR0,..,XAR7(32) —All of the AR registers are stretched to 32-bits. This enables a full 22-bit address. For addressing operations, only the lower 22-bits of the registers are used, the upper 10-bits are ignored. For operations between the ACC, all 32-bits are valid (register addressing mode @XARx). For 16-bit operations to the low 16-bit of the registers (register addressing mode @ARx), the upper 16-bits are ignored.

RPC(22) —This is the return PC register. When a call operation is performed, the return address is saved in the RPC register and the old value in the RPC is saved on the stack (in two 16-bit operations). When a return operation is performed, the return address is read from the RPC register and the value on the stack is written into the RPC register (in two 16-bit operations). The net result is that return operations are faster (4 instead of 8 cycles)

SP(16) —By default the C28x SP register is initialized to 0x400 after a reset.

ST0 (16) —Shaded items indicate a change or addition from the C27x.

Table E-1. ST0 Register Bits

Bit(s)	Mnemonic	Description	Reset Value	R/W
0	SXM	Sign Extension Mode Bit	0	R/W
1	OVM	Overflow Mode Bit	0	R/W
2	TC	Test Control Bit	0	R/W
3	C	Carry Bit	0	R/W
4	Z	Zero Condition Bit	0	R/W
5	N	Negative Condition Bit	0	R/W
6	V	Overflow Condition Bit	0	R/W
9:7	PM	Product Shift Mode	0 (+1 shift)	R/W
15:10	OVC/OVCU	ADC Overflow Counter	0	R/W

PM — Functionality of the Product Shift Mode changes if the AMODE bit in ST1 is set to 1. C27x users will not modify the AMODE bit and PM will function as they did on the C27x.

OVC/OVCU — The overflow counter is modified so that it behaves differently for signed or unsigned operations. For signed operations (OVC), it behaves as it does on the C27x (increment for positive overflow, decrement for negative underflow of a signed number). For unsigned operations (OVCU), the overflow counter increments for an ADD operation when there is a carry generated and decrements for a SUB operation when a borrow is generated. Basically, in unsigned mode, the OVCU behaves like a carry (C) counter and in signed mode the OVC behaves like an overflow (V) counter.

Table E-2. ST1 Register Bits

Bit(s)	Mnemonic	Description	Reset Value	R/W
0	INTM	Interrupt Enable Mask Bit	1 (disabled)	R/W
1	DBGM	Debug Enable Mask Bit	1 (disabled)	R/W
2	PAGE0	PAGE0 Direct/Stack Address Mode	0	R/W
3	VMAP	Vector Map Bit	VMAP input	R/W
4	SPA	Stack Pointer Align Bit	0	R/W
5	LOOP	Loop Instruction Status Bit	0	R
6	EALLOW	Emulation Access Enable Bit	0	R/W
7	IDLESTAT	IDLE Status Flag Bit	0	R
8	AMODE	Address Mode Bit	0	R/W
9	OBJMODE	Object Compatibility Mode Bit	0	R/W
10	RESERVED	Reserved for future use	0	R
11	M0M1MAP	M0 and M1 Mapping Mode Bit	0	R
12	XF	XF Status Bit	0	R/W
15:13	ARP	Auxiliary Register Pointer	0	R/W

AMODE — This mode selects the appropriate addressing mode decodes for compatibility with the C2xLP device. For all C27x/C28x based projects leave this bit as 0.

OBJMODE — This mode is used to select between C27x object mode (OBJMODE == 0) and C28x object mode (OBJMODE == 1) compatibility. This bit is set by the ”C28OBJ” (or ”SETC OBJMODE”) instructions. This bit is cleared by the ”C27OBJ” (or ”CLRC OBJMODE”) instructions. The pipeline is flushed when setting or clearing this bit using the given instructions. This bit can be saved and restored by interrupts and when restoring the ST1 register. This bit is set to 0 on reset.

M0M1MAP — This mode is used to remap block M0 and M1 in program memory space as discussed in detail in Section E.1.2. This bit is set by the ”C28MAP” (or ”SETC M0M1MAP”) instructions. This bit is cleared by the ”C27MAP” (or ”CLRC M0M1MAP”) instructions. The pipeline is flushed when setting or clearing this bit using the given instructions. This bit cannot be restored by interrupts and when restoring the ST1 register (read only).

XF — This bit reflects the current state of the XFS output signal. This signal is for C2xLP compatibility and is not used by C27x users.

E.1.2 Full Context Save and Restore

On both C27x and C28x, the registers in Figure E-2 are automatically saved on the stack on an interrupt or trap operation and automatically restored on an IRET instruction.

31 16	1 0
T	ST0
AH	AL
PH	PL
AR1	AR0
DP	ST1
DBGSTAT	IER
PCH	PCL

Figure E-2. Full Context Save/Restore

Due to the register changes described in Section E.1.1. C28x additional registers must be saved for a full-context store. Figure E-3 shows the difference between a C27x and C28x full-context save/restore for an interrupt or trap.

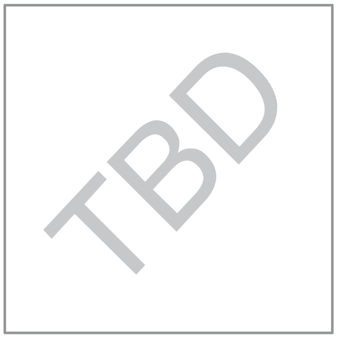


Figure E-3. Code for a Full Context Save/Restore for C28x vs C27x

If you perform a task-switch operation (stack changes), the RPC register must be manually saved. You are not to save the RPC register if the stack is not changed.

E.1.3 B0/B1 Memory Map Consideration

Another architecture change to consider is the C27x mapping of blocks B0 and B1. To avoid confusion, on the C28x these blocks are known as M1 and M0 respectively. On the C27x, block B1 was mapped to only data space and block B0 was mapped both in program and data space. In addition, block B0 was mapped to different address ranges in program and in data space. The C27x mapping of these blocks is shown in Figure E-4.

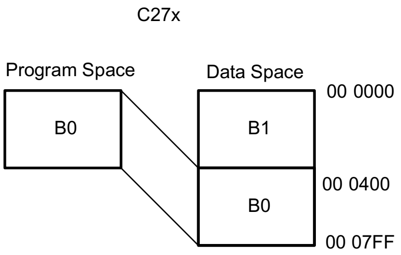


Figure E-4. Mapping of Memory Blocks B0 and B1 on C27x

On a C28x device at reset, these blocks are mapped uniformly in both program and data space as shown in Figure E-5. This can cause issues when running C27x object code that relies on the C27x mapping. If your code relies on this mapping, you can flip-block M0 and M1 in program space only by clearing the M0M1MAP bit in status register 1 (ST1) to a 0. Executing the "C27MAP" (or "CLRC M0M1MAP") instruction is the only way to clear this bit. With M0M1MAP == 0, the mapping is compatible with the C27x B0 and B1 blocks as shown in Figure D-4. Remember that after a reset M0 and M1 revert to the C28x mapping.

It is strongly recommended that you migrate your code to use the default C28x mapping of these blocks and not rely on the compatible mapping.

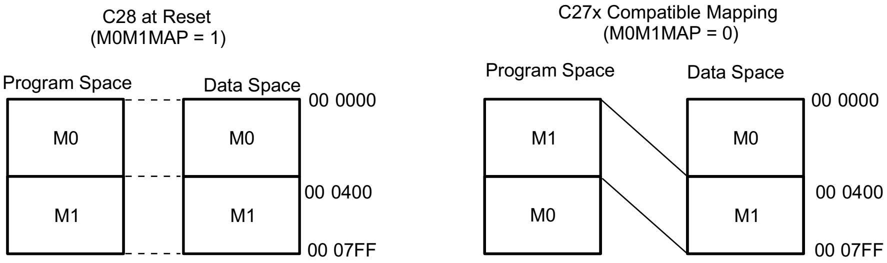


Figure E-5. C27x Compatible Mapping of Blocks M0 and M1

E.1.4 C27x Object Compatibility

At reset, the C28x operates in C27x object mode (OBJMODE == 0). In this mode, the C28x CPU is 100% object-code compatible and cycle-count compatible with the C27x. In this case, you will compile your code just as you would for a C27x design as shown in Figure E-6.

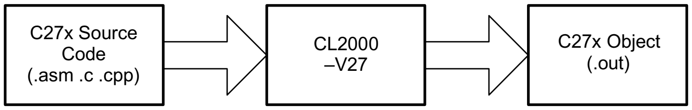


Figure E-6. Building a C27x Object File From C27x Source

-v27 Accepts C27x syntax only. Generates C27x object only (assumes OBJMODE = 0).

Once you have taken the mapping of blocks M0 and M1 into account as previously described, you can simply load the C27x object (.out) code into the C28x and run it. When using the C27x compatible mode, you are limited to the C27x instruction set. To take advantage of advanced C28x operations, you should migrate to C28x object code.

When the device is operating in C27x object mode (OBJMODE == 0), the upper bits of the stretched registers (XAR0(31:16) to XAR5(31:16), XAR6(31:22), XAR7(31:22)) are protected from writes. Hence, if the registers are set to zero by a reset then the XARn pointers behave like they do on the C27x and overflow problems are not of concern.

E.2 Moving to a C28x Object

The C28x instruction set is a superset of the C27x instruction set. The syntax of a number of instructions however has changed slightly due to the modifications in registers as previously described. (For a summary of syntax changes, see Section E.3.1). To quickly move to C28x object code, the codegen tools allow you to build a C28x object file with a switch allowing for C27x source syntax:

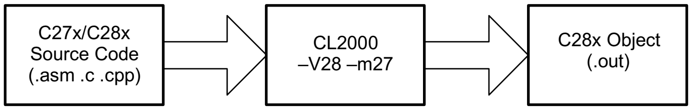


Figure E-7. Building a C28x Object File From Mixed C27x/C28x Source

-v28-m27 Accepts C28x & C27x syntax. Generates C28x object only (assumes OBJMODE == 1).

Prior to running C28x object you must set the mode of the device appropriately (OBJMODE == 1). To do this, you set the OBJMODE bit in ST1 to 1 after reset. This can be done with a ”C28OBJ” (or ”SETC OBJMODE”) instruction. Note that before the “C28OBJ” instruction is executed, the disassembly window in the debugger may display incorrect information. This is because the debugger will decode memory as C27x opcodes until after you execute the “C28OBJ” instruction.

When running in this mode, the disassembly window in your debugger will show the C28x instruction syntax for all instructions. For example, the C27x MOV AR0,@SP instruction will look like MOVZ AR0,@SP, which is the C28x-equivalent instruction.

Now that you are using a C28x object file, you can add C28x operations to your source code.

E.2.1 Caution When Changing OBJMODE

On reset, the XARn registers are forced to 0x0000 0000 and OBJMODE == 0. When operating in C27x compatible mode (OBJMODE == 0), the upper bits of the XARn registers are protected from writes. Some things to be aware of when changing OBJMODE:

- When operating in C28x object mode (OBJMODE == 1) overflow can occur to the extended portion of XARn registers and program execution is not specified. This would be an issue for assembly code that is reassembled in C28x mode when you relied on the fact that C27x registers were a certain size.
- If the user switches to C28x object mode (OBJMODE == 1), then the upper bits of XARn registers may be modified. If you then switch back to C27x mode (OBJMODE == 0), the upper bits of XARn registers may contain nonzero values. You MUST zero out the upper bits of the XARn registers when switching from OBJMODE == 1 to OBJMODE == 0.
- It is recommended that you not switch modes frequently in your code. Typically, you will select the appropriate operating mode at boot time and stick to one mode for the whole program.

E.3 Migrating to C28x Object Code

This section describes additional changes to C27x necessary for migrating your C27x code to pure C28x code.

E.3.1 Instruction Syntax Changes

Syntax changes were necessary for clarity and because of changes in the auxiliary registers stretched pointers. Table E-3 shows the C27x instructions that changed syntax on the C28x. For all other C27x instructions, the syntax remains the same. For new C28x instructions, the syntax is documented in Chapter 6.

Table E-3. Instruction Syntax Change

C27x Syntax	C28x Syntax

`ADDB ARn,#7bit`
`ADDB XAR6/7,#7bit`
	`ADDB XARn,#7bit`

`SUBB ARn,#7bit`
`SUBB XAR6/7,#7bit`
	`SUBB XARn,#7bit`

`MOV AR0/../5,loc16`	`MOVZ AR0/../5,loc16`
`MOVB AR0/../5,#8bit`	`MOVB XAR0/../5,#8bit`

`MOV XAR6/7,loc32`
`MOVL XAR6/7,loc32`
	`MOVL XAR6/7,loc32`

`MOV loc32,XAR6/7`
`MOVL loc32, XAR6/7`
	`MOVL loc32,XAR6/7`

`CALL 22bit`
`LC 22bit`
	`LC 22bit`

`CALL *XAR7`
`LC *XAR7`
	`LC *XAR7`

`RET` `LRET`	`LRET`

Table E-3. Instruction Syntax Change (continued)

C27x Syntax	C28x Syntax
RETE LRETE	LRETE
MOV ACC,P {MOVP T,@T decode}	MOVL ACC,P << PM {MOVP T,@T decode}
ADD ACC,P {MOVA T,@T decode}	ADD ACC,P << PM {MOVA T,@T decode}
SUB ACC,P {MOVS T,@T decode}	SUBL ACC,P << PM {MOVS T,@T decode}
CMP ACC,P	CMPL ACC,P << PM
MOV P,ACC	MOVL P,ACC
NORM ACC,ARn++ NORM ACC,XAR6/7++	NORM ACC,XARn++
NORM ACC,ARn-- NORM ACC,XAR6/7--	NORM ACC,XARn--
B 16bitOff {unconditional}	B 16bitOff,UNC [2]
SB 8bitOff {unconditional}	SB 8bitOff,UNC [2]

For conditional branches on the C28x, the UNC code must always be specified for unconditional tests. This will help to distinguish between unconditional C2xLP branches (which have the same mnemonic ”B”).

E.3.2 Repeatable Instructions

On the C28x, additional instructions have been made repeatable. The following two tables list those instructions that are repeatable on the C28x device. These instructions are repeatable in both C27x compatible mode (OBJMODE = 0) and C28x native mode (OBJMODE = 1). Any instruction that is not listed, which follows a repeat instruction, will execute only once.

C27x operations that were already repeatable include the following:

ROR	ACC
ROL	ACC
NORM	ACC,XARn++
NORM	ACC,XARn--
SUBCU	ACC,loc16
MAC	P,loc16,0:pma
MOV	*(0:addr),loc16
MOV	loc16,*(0:addr)
MOV	loc16,#16bit
MOV	loc16,#0
PREAD	loc16,*XAR7
PWRITE	*XAR7,loc16
NOP	loc16

C27x Operations That Are Made Repeatable On C28x include the following:

MOV	loc16,AX
ADD	ACC,loc16<<16
ADDU	ACC,loc16
SUB	ACC,loc16<<16
SUBU	ACC,loc16
ADDL	ACC,loc32
SFR	ACC,1..16
LSL	ACC,1..16

MOVH	loc16,P
MOV	loc16,P
MOVA	T,loc16
MOVS	T,loc16
MPYA	PT,loc16
MPYS	PT,loc16

E.3.3 Changes to the SUBCU Instruction

The SUBCU instruction changed slightly from the C27x to the C28x. Under the prescribed usage of the SUBCU operation, the change will yield the same result as the C27x.

The SUBCU instruction operates as follows on the C27x device:

```
temp(31:0) = ACC - [loc16] << 15
if(temp32 <= 0)
     ACC = temp(31:0) >> 1 + 1;
else
     ACC = ACC << 1;
```

To simplify the implementation, the SUBCU operation changed as follows on the C28x:

```
temp(32:0) = ACC << 1 - [loc16] << 16
if(temp(32:0) >= 0)
     ACC = temp(31:0) +1;
else
     ACC = ACC << 1;
```

- The ”temp(32:0)” value is the result of an unsigned 33-bit compare. The carry bit is used to select between ? or < condition.
- The C flag is affected by the unsigned 33-bit compare operation. The Z, N flags reflect the value in the ACC after the operation is complete. The operation of the C, N, Z flags should be identical to the C27x implementation.
- The V flag and overflow counter (OVC) are not affected by the operation. On the C27x the V and OVC flags are affected.

The V and OVC flags may be affected on the C27x and not on the C28x implementation. The values of these flags are not usable under prescribed usage of such an operation.

E.4 Compiling C28x Source Code

Once you move your code to C28x native instructions, you will no longer use the −m27 switch to allow for C27x source as shown in Figure E-8.

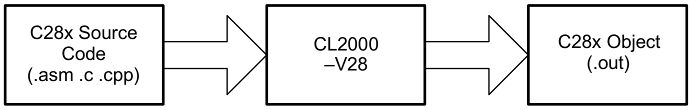


Figure E-8. Compiling C28x Source

−v28: Accepts C28x syntax only. Generates C28x object only (assumes OBJMODE = 1)

Glossary

F.1 Glossary

The following are defined terms used in this document.

16-bit operation — An operation that reads or writes 16 bits.

32-bit operation — An operation that reads or writes 32 bits.

absolute branch — A branch to an address that is permanently assigned to a memory location. See also *offset branch*.

ACC — See *accumulator (ACC)*.

access — A term used in this document to mean *read from* or *write to*. For example, to access a register is to read from or write to that register.

accumulator (ACC) — A 32-bit register involved in a majority of the arithmetic and logical calculations done by the C28x. Some instructions that affect ACC use all 32 bits of the register. Others use one of the following portions of ACC: AH (bits 31 through 16), AL (bits 15 through 0), AH.MSB (bits 31 through 24), AH.LSB (bits 23 through 16), AL.MSB (bits 15 through 8), and AL.LSB (bits 7 through 0).

address-generation logic — Hardware in the CPU that generates the addresses used to fetch instructions or data from memory.

address reach — The range of addresses beginning with 00 0000_{16} that can be used by a particular addressing mode.

address register arithmetic unit (ARAU) — Hardware in the CPU that generates addresses for values that must be fetched from data memory. The ARAU is also the hardware used to increment or decrement the stack pointer (SP) and the auxiliary registers (AR0, AR1, AR2, AR3, AR4, AR5, XAR6, and XAR7).

addressing mode — The method by which an instruction interprets its operands to acquire the data and/or addresses it needs.

AH — *High word of the accumulator.* The name given to bits 31 through 16 of the accumulator.

AH.LSB — *Least significant byte of AH.* The name given to bits 23 through 16 of the accumulator.

AH.MSB — *Most significant byte of AH.* The name given to bits 31 through 24 of the accumulator.

AL — *Low word of the accumulator.* The name given to bits 15 through 0 of the accumulator.

AL.LSB — *Least significant byte of AL.* The name given to bits 7 through 0 of the accumulator.

AL.MSB — *Most significant byte of AL.* The name given to bits 15 through 8 of the accumulator.

ALU — See *arithmetic logic unit (ALU).*

analysis logic — A portion of the emulation logic in the core. The analysis logic is responsible for managing the following debug activities: hardware breakpoints, hardware watchpoints, data logging, and benchmark/event counting.

approve an interrupt request— Allow an interrupt to be serviced. If the interrupt is maskable, the CPU approves the request only if it is properly enabled. If the interrupt is nonmaskable, the CPU approves the request immediately. See also *interrupt request* and *service an interrupt*.

ARAU — See *address register arithmetic unit (ARAU).*

arithmetic logic unit (ALU) — A 32-bit hardware unit in the CPU that performs 2s-complement arithmetic and Boolean logic operations. The ALU accepts inputs from data from registers, from data memory, or from the program control logic. The ALU sends results to a register or to data memory.

arithmetic shift — A shift that treats the shifted value as signed. See also *logical shift*.

ARP — See *auxiliary register pointer (ARP).*

ARP indirect addressing mode — The indirect addressing mode that uses the current auxiliary register to point to a location in data space. The current auxiliary register is the auxiliary register pointed to by the ARP. See also *auxiliary register pointer (ARP).*

automatic context save — A save of system context (modes and key register values) performed by the CPU just prior to executing an interrupt service routine. See also *context save.*

auxiliary register — One of eight registers used as a pointer to a memory location. The register is operated on by the auxiliary register arithmetic unit (ARAU) and is selected by the auxiliary register pointer (ARP). See also *AR0−AR5, AR6/AR7,* and *XAR6/XAR7.*

auxiliary-register indirect addressing mode — The indirect addressing mode that allows you to use the name of an auxiliary register in an operand that uses that register as a pointer. See also *ARP indirect addressing mode.*

auxillary register pointer (ARP) — A 3-bit field in status register ST1 that selects the current auxiliary register. When an instruction uses ARP indirect addressing mode, that instruction uses the current auxiliary register to point to data space. When an instruction specifies auxiliary register *n* by using auxiliary-register indirect addressing mode, the ARP is updated, so that it points to auxiliary register *n*. See also *current auxiliary register.*

background code — The body of code that can be halted during debugging because it is not time-critical.

barrel shifter — Hardware in the CPU that performs all left and right shifts of register or data-space values.

bit field — One or more register bits that are differentiated from other bits in the same register by a specific name and function.

bit manipulation — The testing or modifying of individual bits in a register or data-space location.

boundary scan — The use of scan registers on the border of a chip or section of logic to capture the pin states. By scanning these registers, all pin states can be transmitted through the JTAG port for analysis.

branch — 1) A forcing of program control to a new address. 2) An instruction that forces program control to a new address but neither saves a return address (like a call) nor restores a return address (like a return).

break event — A debug event that causes the CPU to enter the debug-halt state.

breakpoint — A place in a routine specified by a breakpoint instruction or hardware breakpoint, where the execution of the routine is to be halted and the debug-halt state entered.

C bit — See *carry (C) bit.*

call — 1) The operation of saving a return address and then forcing program control to a new address. 2) An instruction that performs such an operation. See also *return.*

carry (C) bit — A bit in status register ST0 that reflects whether an addition has generated a carry or a subtraction has generated a borrow.

circular addressing mode — The indirect addressing mode that can be used to implement a circular buffer.

circular buffer — A block of addresses referenced by a pointer using circular addressing mode, so that each time the pointer reaches the bottom of the block, the pointer is modified to point back to the top of the block.

clear — To clear a bit is to write a 0 to it. To clear a register or memory location is to load all its bits with 0s. See also *set*.

COFF — *Common object file format.* A binary object file format that promotes modular programming by supporting the concept of sections, where a section is a relocatable block of code or data that ultimately occupies a space adjacent to other blocks of code in the memory map.

conditional branch instruction — A branch instruction that may or may not cause a branch, depending on a specified or predefined condition (for example, the state of a bit).

context restore — A restoring of the previous state of a system (for example, modes and key register values) prior to returning from a subroutine. See also *context save.*

context save — A save of the current state of a system (for example, modes and key register values) prior to executing the main body of a subroutine that requires a different context. See also *context restore.*

core — The portion of the C28x that consists of a CPU, a block of emulation circuitry, and a set of signals for interfacing with memory and peripheral devices.

current auxiliary register — The register selected by the auxiliary register pointer (ARP) in status register. For example, if ARP = 3, the current auxiliary register is AR3. See also *auxiliary registers.*

current data page — The data page selected by the data page pointer. For example, if DP = 0, the current data page is 0. See also *data page.*

D1 phase — See *decode 1 (D1) phase.*

D2 phase — See *decode 2 (D2) phase.*

data logging — Transferring one or more packets of data from CPU registers or memory to an external host processor.

data log interrupt (DLOGINT) — A maskable interrupt triggered by the onchip emulation logic when a data logging transfer has been completed.

data page — A 64-word portion of the total 4M words of data space. Each data page has a specific start address and end address. See also *data page pointer (DP)* and *current data page.*

data page pointer (DP) — A 16-bit pointer that identifies which 64-word data page is accessed in DP direct addressing mode. For example, for as long as DP = 500, instructions that use DP direct addressing mode will access data page 500.

data-/program-write data bus (DWDB) — The bus that carries data during writes to data space or program space.

data-read address bus (DRAB) — The bus that carries addresses for reads from data space.

data-read data bus (DRDB) — The bus that carries data during reads from data space.

data-write address bus (DWAB) — The bus that carries addresses for writes to data space.

DBGIER — See *debug interrupt enable register (DBGIER).*

DBGM bit — See *debug enable mask (DBGM) bit.*

DBGSTAT — See *debug status register (DBGSTAT).*

debug-and-test direct memory access (DT−DMA) — An access of a register or memory location to provide visibility to this location during debugging. The access is performed with variable levels of intrusiveness by a hardware DT-DMA mechanism inside the core.

debug enable mask (DBGM) bit — A bit in status register ST1 used to enable (DBGM = 0) or disable (DBGM = 1) debug events such as analysis breakpoints or debug-and-test direct memory accesses (DT-DMAs).

debug event — A debug execution state that is entered through a break event. In this state the CPU is halted. See also *single-instruction state* and *run state.*

debug-halt state —

debug host — See *host processor.*

debug interrupt enable register (DBGIER) — The register that determines which of the maskable interrupts are time-critical when the CPU is halted in real-time mode. If a bit in the DBGIER is 1, the corresponding interrupt is time-critical/enabled; otherwise, it is disabled. Time-critical interrupts also must be enabled in the interrupt enable register (IER) to be serviced.

debug status register (DBGSTAT) — A register that holds special debug status information. This register, which need not be read from or written to, is saved and restored during interrupt servicing, to preserve the debug context during debugging.

decode an instruction — To identify an instruction and prepare the CPU to perform the operation the instruction requires.

decode 1 (D1) phase — The third of eight pipeline phases an instruction passes through. In this phase, the CPU identifies instruction boundaries in the instruction-fetch queue and determines whether the next instruction to be executed is an illegal instruction. See also *pipeline phases.*

decode 2 (D2) phase — The fourth of eight pipeline phases an instruction passes through. In this phase, the CPU accepts an instruction from the instruction-fetch queue and completes the decoding of that instruction, performing such activities as address generation and pointer modification. See also *pipeline phases.*

decrement — To subtract 1 or 2 from a register or memory value. The value subtracted depends on the circumstance. For example, if you use the operand *−−AR4, the auxiliary register AR4 is decremented by 1 for a 16-bit operation and by 2 for a 32-bit operation.

device reset — See *reset.*

direct addressing modes — The addressing modes that access data space as if it were 65 536 separate blocks of 64 words each. DP direct addressing mode uses the data page pointer (DP) to select a data page from 0 to 65 535. PAGE0 direct addressing mode uses data page 0, regardless of the value in the DP.

discontinuity — See *program-flow discontinuity.*

DLOGINT — See *data log interrupt (DLOGINT).*

DP — See *data page pointer (DP).*

DP direct addressing mode — A direct addressing mode that uses the data page pointer (DP) to select a data page from 0 to 65 535. See also *PAGE0 direct addressing mode.*

DRAB — See *data-read address bus (DRAB).*

DRDB — See *data-read data bus (DRDB).*

DT-DMA — See *debug-and-test direct memory access (DT-DMA).*

DWAB — See *data-write address bus (DWAB).*

DWDB — See *data-/program-write data bus (DWDB).*

E phase — See *execute (E) phase.*

EALLOW bit — See *emulation access enable (EALLOW) bit.*

EMU0 and EMU1 pins — Pins known as the TI extensions to the JTAG interface. These pins can be used as either inputs or outputs and are available to help monitor and control an emulation target system that is using a JTAG interface.

emulation access enable (EALLOW) bit — A bit in status register ST1 that enables (EALLOW = 1) or disables (EALLOW = 0) access to the emulation registers. The EALLOW instruction sets the EALLOW bit, and the EDIS instruction clears the EALLOW bit.

emulation logic — The block of hardware in the core that is responsible controlling emulation activities such as data logging and switching among debug execution states.

emulation registers — Memory-mapped registers that are available for controlling and monitoring emulation activities.

enable bit — See *interrupt enable bits.*

execute an instruction — Take an instruction from the decode 2 phase of the pipeline through the write phase of the pipeline.

execute (E) phase — The seventh of eight pipeline phases an instruction passes through. In this phase, the CPU performs all multiplier, shifter, and arithmetic-logic-unit (ALU) operations. See also *pipeline phases.*

extended auxiliary registers — See *XAR6/XAR7.*

F1 phase — See *fetch 1 (F1) phase.*

F2 phase — See *fetch 2 (F2) phase.*

FC — See *fetch counter (FC).*

fetch 1 (F1) phase — The first of eight pipeline phases an instruction passes through. In this phase, the CPU places on the program-read bus the address of the instruction(s) to be fetched. See also pipeline phases.

fetch 2 (F2) phase — The second of eight pipeline phases an instruction passes through. In this phase, the CPU fetches an instruction or instructions from program memory. See also *pipeline phases.*

fetch counter (FC) — The register that contains the address of the instruction that is being fetched from program memory.

field — See *bit field.*

hardware interrupt — An interrupt initiated by a physical signal (for example, from a pin or from the emulation logic). See also *software interrupt.*

hardware interrupt priority — A priority ranking used by the CPU to determine the order in which simultaneously occurring hardware interrupts are serviced.

hardware reset — See *reset.*

high addresses — Addresses closer to 3F FFFF$_{16}$ than to 00 0000$_{16}$. See also low addresses.

high bits — See *MSB.*

high word — The 16 MSBs of a 32-bit value. See also *low word.*

host processor — The processor running the user interface for a debugger.

IC —

IDLESTAT (IDLE status) bit —

idle state —

IEEE 1149.1 standard —

IER — See *interrupt enable register (IER).*

IFR — See *interrupt flag register (IFR).*

illegal instruction — An unacceptable value read from program memory during an instruction fetch. Unacceptable values are 0000_{16}, $FFFF_{16}$, or any value that does not match a defined opcode.

illegal-instruction trap — A trap that is serviced when an illegal instruction is decoded.

immediate address — An address that is specified directly in an instruction as a constant.

immediate addressing modes — Addressing modes that accept a constant as an operand.

immediate constant/data — A constant specified directly as an operand of an instruction.

immediate-constant addressing mode — An immediate addressing mode that accepts a constant as an operand and interprets that constant as data to be stored or processed.

immediate-pointer addressing mode — An immediate addressing mode that accepts a constant as an operand and interprets that constant as the 16 LSBs of a 22-bit address. The six MSBs of the address are filled with 0s.

increment — To add 1 or 2 to a register or memory value. The value added depends on the circumstance. For example, if you use the operand *AR4++, the auxiliary register AR4 is incremented by 1 for a 16-bit operation and by 2 for a 32-bit operation.

indirect addressing modes — Addressing modes that use pointers to access memory. The available pointers are auxiliary registers AR0−AR5, extended auxiliary registers XAR6 and XAR7, and the stack pointer (SP).

instruction boundary — The point where the CPU has finished one instruction and is considering what it will do next — move on to the next instruction.

instruction counter (IC) — The register that points to the instruction in the decode 1 phase (the instruction that is to enter the decode 2 phase next). Also, on an interrupt or call operation, the IC value represents the return address, which is saved to the stack or to auxiliary register XAR7.

instruction-fetch mechanism — The hardware for the fetch 1 and fetch 2 phases of the pipeline. This hardware is responsible for fetching instructions from program memory and filling an instruction-fetch queue.

instruction-fetch queue — A queue of four 32-bit registers that receives fetched instructions and holds them for decoding. When a program-flow discontinuity occurs, the instruction-fetch queue is emptied.

instruction-not-available condition — The condition that occurs when the decode 2 pipeline hardware requests an instruction but there are no instructions waiting in the instruction-fetch queue. This condition causes the decode 2 through write phases of the pipeline to freeze until one or more new instructions have been fetched.

instruction register — The register that contains the instruction that has reached the decode 2 pipeline phase.

instruction word — Either an entire 16-bit opcode or one of the halves of a 32-bit opcode.

INT1-$\overline{\text{INT14}}$ — Fourteen general-purpose interrupts that are triggered by signals at pins of the same names. These interrupts are maskable and have corresponding bits in the interrupt flag register (IFR), the interrupt enable register (IER), and the debug interrupt enable register (DBGIER).

interrupt boundary — An instruction boundary where the CPU can insert an interrupt between two instructions. See also *instruction boundary.*

interrupt enable bits — Bits responsible for enabling or disabling maskable interrupts. The enable bits are all the bits in the interrupt enable register (IER), all the bits in the debug interrupt enable register (DBGIER), and the interrupt global mask bit (INTM in status register ST1).

interrupt enable register (IER) — Each of the maskable interrupts has an interrupt enable bit in this register. If a bit in the IER is 1, the corresponding interrupt is enabled; otherwise, it is disabled. See also *debug interrupt enable register (DBGIER).*

interrupt flag bit — A bit in the interrupt flag register (IFR). If the interrupt flag bit is 1, the corresponding interrupt has been requested by hardware and is awaiting approval by the CPU.

interrupt flag register (IFR) — The register that contains the interrupt flag bits for the maskable interrupts. If a bit in the IFR is 1, the corresponding interrupt has been requested by hardware and is awaiting approval by the CPU.

interrupt global mask (INTM) bit — A bit in status register ST1 that globally enables or disables the maskable interrupts. If an interrupt is enabled in the interrupt enable register (IER) but not by the INTM bit, it is not serviced. The only time this bit is ignored is when the CPU is in real-time mode and is in the debug-halt state; in this situation, the interrupt must be enabled in the IER and in the DBGIER (debug interrupt enable register).

interrupt priority — See *hardware interrupt priority.*

interrupt request — A signal or instruction that requests the CPU to execute a particular interrupt service routine. See also *approve an interrupt request* and *service an interrupt.*

interrupt service routine (ISR) — A subroutine that is linked to a specific interrupt by way of an interrupt vector.

interrupt vector — The start address of an interrupt service routine. After approving an interrupt request, the CPU fetches the interrupt vector from your interrupt vector table and uses the vector to branch to the start of the corresponding interrupt service routine.

interrupt vector location — The preset location in program memory where an interrupt vector must reside.

interrupt vector table — The list of interrupt vectors you assign in program memory.

INTM bit — See *interrupt global mask (INTM) bit.*

ISR — See *interrupt service routine (ISR).*

JTAG — *Joint Test Action Group.* The Joint Test Action Group was formed in 1985 to develop economical test methodologies for systems designed around complex integrated circuits and assembled with surface-mount technologies. The group drafted a standard that was subsequently adopted by IEEE as IEEE Standard 1149.1-1990, “IEEE Standard Test Access Port and Boundary-Scan Architecture”. See also *boundary scan; test access port (TAP).*

JTAG port — See *test access port (TAP).*

latch — Hold a bit at the same value until a given event occurs. For example, when an overflow occurs in the accumulator, the V bit is set and latched at 1 until it is cleared by a conditional branch instruction or by a write to status register ST0. An interrupt is latched when its flag bit has been latched in the interrupt flag register (IFR).

least significant bit (LSB) — The bit in the lowest position of a binary number. For example, the LSB of a 16-bit register value is bit 0. See also *MSB, LSByte*, and *MSByte.*

least significant byte (LSByte) — The byte in the lowest position of a binary value. The LSByte of a value consists of the eight LSBs. See also *MSByte, LSB*, and *MSB.*

location — A space where data can reside. A location may be a CPU register or a space in memory.

logical shift — A shift that treats the shifted value as unsigned. See also *arithmetic shift.*

LOOP (loop instruction status) bit — A bit in status register ST1 that indicates when a LOOPNZ or LOOPZ instruction is being executed (LOOP = 1).

low addresses — Addresses closer to 00 0000_{16} than to 3F FFFF_{16}. See also *high addresses*.

low bits — See *LSB*.

low word — The 16 LSBs of a 32-bit value. See also *high word.*

LSB — When used in a syntax of the MOVB instruction, LSB means least significant byte. Otherwise, LSB means least significant bit. See *least significant bit (LSB)* and *least significant byte (LSByte).*

LSByte — See *least significant byte (LSByte).*

maskable interrupt — An interrupt that can be disabled by software so that the CPU does not service it until it is enabled by software. See also *non-maskable interrupt.*

memory interface — The buses and signals responsible for carrying communications between the core and on-chip memory/peripherals.

memory-mapped register — A register that can be accessed at addresses in data space.

memory wrapper — The hardware around a memory block that identifies access requests and controls accesses for that memory block.

mirror — A range of addresses that is the same size and is mapped to the same physical memory block as another range of addresses.

most significant (MSB) — The bit in the highest position of a binary number. For example, the MSB of a 16-bit register value is bit 15. See also *LSB, LSByte*, and *MSByte.*

most significant byte (MSByte) — The byte in the highest position of a binary value. The MSByte of a value consists of the eight MSBs. See also *LSByte, LSB*, and *MSB.*

MSB — When used in a syntax of the MOVB instruction, MSB means most significant byte. Otherwise MSB means most significant bit. See *most significant bit (MSB)* and *most significant byte (MSByte).*

MSByte — See *most significant byte (MSByte).*

multiplicand register (T) — The primary function of this register, also called the T register, is to hold one of the values to be multiplied during a multiplication. The following shift instructions use the four LSBs to hold the shift count: ASR (arithmetic shift right), LSL (logical shift left), LSR (logical shift right), and SFR (shift accumulator right). The T register can also be used as a general-purpose 16-bit register.

N (negative flag) bit — A bit in status register ST0 that indicates whether the result of a calculation is a negative number (N = 1). N is set to match the MSB of the result.

nested interrupt — An interrupt that occurs within an interrupt service routine.

$\overline{\text{NMI}}$ — A hardware interrupt that is nonmaskable, like reset (RS), but does not reset the CPU. $\overline{\text{NMI}}$ simply forces the CPU to execute its interrupt service routine.

nonmaskable interrupt — An interrupt that cannot be blocked by software and is approved by the CPU immediately. See also *maskable interrupt.*

offset branch — A branch that uses a specified or generated offset value to jump to an address relative to the current position of the program counter (PC). See also *absolute branch.*

opcode — This document uses opcode to mean the complete code for an instruction. Thus, an opcode includes the binary sequence for the instruction type and the binary sequence and/or constant in which the operands are encoded.

operand — : This document uses operand to mean one of the values entered after the instruction mnemonic and separated by commas (or for a shift operand, separated by the symbol <<). For example, in the CLRC INTM instruction, CLRC is the mnemonic and INTM is the operand.

operation — 1) A defined action; namely, the act of obtaining a result from one or more operands in accordance with a rule that completely specifies the result of any permitted combination of operands. 2) The set of such acts specified by a rule, or the rule itself. 3) The act specified by a single computer instruction. 4) A program step undertaken or executed by a computer; for example, addition, multiplication, extraction, comparison, shift, transfer, etc. 5) The specific action performed by a logic element.

OVC — See *overflow counter (OVC).*

OVM — See *overflow mode (OVM) bit.*

overflow counter (OVC) — A 6-bit counter in status register ST0 that can be used to track overflows in the accumulator (ACC). The OVC is enabled only when the overflow mode (OVM) bit in ST0 is 0. When OVM = 0, the OVC is incremented by 1 for every overflow in the positive direction (too large a positive number) and decremented by 1 for every overflow in the negative direction (too large a negative number). The saturate (SAT) instruction modifies ACC to reflect the net overflow represented in the OVC.

overflow flag (V) — A bit in status register ST0 that indicates when the result of an operation causes an overflow in the location holding the result (V = 1). If no overflow occurs, V is not modified.

overflow mode (OVM) bit — A bit in the status register ST0 that enables or disables overflow mode. When overflow mode is on (OVM = 1) and an overflow occurs, the CPU fills the accumulator (ACC) with a saturation value. When overflow mode is off (OVM = 0), the CPU lets ACC overflow normally but keeps track of each overflow by incrementing or decrementing by 1 the overflow counter (OVC) in ST0.

P register — See p*roduct register (P).*

PAB — See *program address bus (PAB).*

PAGE0 bit — *PAGE0 addressing mode configuration bit.* This bit, in status register ST1, selects between two addressing modes: PAGE0 stack addressing mode (PAGE = 0) and PAGE0 direct addressing mode (PAGE0 = 1).

PAGE0 direct addressing mode — The direct addressing mode that uses data page 0 regardless of the value in the data page pointer (DP). This mode is available only when the PAGE0 bit in status register ST1 is 1. See also *DP direct addressing mode and PAGE0 stack addressing mode.*

PAGE0 stack addressing mode — The indirect addressing mode that references a value on the stack by subtracting a 6-bit offset from the current position of the stack pointer (SP). This mode is available only when the PAGE0 bit in status register ST1 is 0. See also *stack-pointer indirect addressing mode.*

PC — See *program counter (PC).*

pending interrupt — An interrupt that has been requested but is waiting for approval from the CPU. See also *approve an interrupt request.*

peripheral-interface logic — Hardware that is responsible for handling communications between a processor and a peripheral.

PH — The high word (16 MSBs) of the P register.

phases — See *pipeline phases.*

pipeline — The hardware in the CPU that takes each instruction through eight independent phases for fetching, decoding, and executing. During any given CPU cycle, there can be up to eight instructions in the pipeline, each at a different phase of completion. The phases, listed in the order in which instructions pass through them, are fetch 1, fetch 2, decode 1, decode 2, read 1, read 2, execute, and write.

pipeline conflict — A situation in which two instructions in the pipeline try to access a register or memory location out of order, causing improper code operation. The C28x pipeline inserts as many inactive cycles as needed between conflicting instructions to prevent pipeline conflicts.

pipeline freeze — A halt in pipeline activity in one of the two decoupled portions of the pipeline. Freezes in the fetch 1 through decode 1 portion of the pipeline are caused by a not-ready signal from program memory. Freezes in the decode 2 through write portion are caused by lack of instructions in the instruction-fetch queue or by not-ready signals from memory.

pipeline phases — The eight stages an instruction must pass through to be fetched, decoded, and executed. The phases, listed in the order in which instructions pass through them, are fetch 1, fetch 2, decode 1, decode 2, read 1, read 2, execute, and write.

pipeline-protection mechanism — The mechanism responsible for identifying potential pipeline conflicts and preventing them by adding inactive cycles between the conflicting instructions.

PL — The low word (16 LSBs) of the P register.

PM bits — See *product shift mode (PM) bits.*

PRDB — See *program-read data bus (PRDB).*

priority — See *interrupt priority.*

product register (P) — This register, also called the P register, is given the results of most multiplications done by the CPU. The only other register that can be given the result of a multiplication is the accumulator (ACC). See also $_{PH}$ and $^{PL.}$

product shift mode (PM) bits — A 3-bit field in status register ST0 that enables you to select one of eight product shift modes. The product shift mode determines whether or how the P register value is shifted before being used by an instruction. You have the choices of a left shift by 1 bit, no shift, or a right shift by N, where N is a number from 1 to 6.

program address bus (PAB) — The bus that carries addresses for reads and writes from program space.

program address generation logic — This logic generates the addresses used to fetch instructions or data from program memory and places each address on the program address bus (PAB).

program control logic — This logic stores a queue of instructions that have been fetched from program memory by way of the program-read bus (PRDB). It also decodes these instructions and passes commands and constant data to other parts of the CPU.

program counter (PC) — When the pipeline is full, the 22-bit PC always points to the instruction that is currently being processed—the instruction that has just reached the decode 2 phase of the pipeline.

program-flow discontinuity — A branching to a nonsequential address caused by a branch, a call, an interrupt, a return, or the repetition of an instruction.

program-read data bus (PRDB) — The bus that carries instructions or data during reads from program space.

R1 phase — See *read 1 (R1) phase.*

R2 phase — See *read 2 (R2) phase.*

read 1 (R1) phase — The fifth of eight pipeline phases an instruction passes through. In this phase, if data is to be read from memory, the CPU drives the address(es) on the appropriate address bus(es). See also *pipeline phases.*

read 2 (R2) phase — The sixth of eight pipeline phases an instruction passes through. In this phase, data addressed in the read 1 phase is fetched from memory. See also *pipeline phases*.

ready signals — When the core requests a read from or write to a memory device or peripheral device, that device can take more time to finish the data transfer than the core allots by default. Each device must use one of the core's *ready signals* to insert wait states into the data transfer when it needs more time. Wait-state requests freeze a portion of the pipeline if they are received during the fetch 1, read 1, or write pipeline phase of an instruction.

real-time mode— An emulation mode that enables you execute certain interrupts (time-critical interrupts), even when the CPU is halted. See also *stop mode.*

real-time operating system interrupt (RTOSINT) — A maskable hardware interrupt generated by the emulation hardware in response to certain debug events. This interrupt should be disabled in the interrupt enable register (IER) and the debug interrupt enable register (DBGIER) unless there is a real-time operating system present in your debug system.

reduced instruction set computer (RISC) — A computer whose instruction set and related decode mechanism are much simpler than those of microprogrammed complex instruction set computers.

register addressing mode — An addressing mode that enables you to reference registers by name.

register conflict — A pipeline conflict that would occur if an instruction read a register value before that value were changed by a prior instruction. The C28x pipeline inserts as many inactive cycles as needed between conflicting instructions to prevent register conflicts.

register pair — One of the pairs of CPU register stored to the stack during an automatic context save.

repeat counter (RPTC) — The counter that is loaded by the RPT (repeat) instruction. The number in the counter is the number of times the instruction qualified by RPT is to be repeated after its initial execution.

reserved — A term used to describe memory locations or other items that you cannot use or modify.

reset — To return the DSP to a known state; an action initiated by the reset ($\overline{\text{RS}}$) signal.

return — 1) The operation of forcing program control to a return address. 2) An instruction that performs such an operation. See also *call.*

return address — The address at which the CPU resumes processing after executing a subroutine or interrupt service routine.

RISC — See *reduced instruction set computer (RISC).*

rotate operation — An operation performed by the ROL (rotate accumulator left) or ROR (rotate accumulator right) instruction. The operation, which involves a shift by 1 bit, can be seen as the rotation of a 33-bit value that is the concatenation of the carry bit (C) and the accumulator (ACC).

RPTC — See *repeat counter (RPTC).*

RTOSINT — See real-time operating system interrupt (RTOSINT).

RUN command — A debugger command used to execute all or a portion of a program. The RUN 1 command causes the debugger to execute a single instruction.

run state — A debug execution state. In this state, the CPU is executing code and servicing interrupts freely. See also *debug-halt state* and *single-instruction state.*

select signal — An output signal from the C28x that can be used to select specific memory or peripheral devices for particular types of read and write operations.

scan controller — A device that performs JTAG state sequences sent to it by a host processor. These sequences, in turn, control the operation of a target device.

service an interrupt — The CPU services an interrupt by preparing for and then executing the corresponding interrupt service routine. See also *interrupt request and approve an interrupt request.*

set — To set a bit is to write a 1 to it. If a bit is set, it contains 1. See also *clear.*

sign extend — To fill the unused most significant bits (MSBs) of a value with copies of the value's sign bit.

sign-extension mode (SXM) bit — A bit in status register ST0 that enables or suppresses sign extension. When sign-extension is enabled (SXM = 1), operands of certain instructions are treated as signed and are sign extended during shifting.

single-instruction state — A debug execution state. In this state, the CPU executes one instruction and then returns to the debug-halt state. See also *debug-halt state* and *run state.*

16-bit operation — An operation that reads or writes 16 bits.

software interrupt — An interrupt initiated by an instruction. See also *hardware interrupt.*

SP — See *stack pointer (SP).*

SPA bit — *See stack pointer alignment (SPA) bit.*

ST0 — See *status registers ST0 andST1.*

ST1 — See *status registers ST0 andST1.*

stack — The C28x stack is a software stack implemented by the use of a stack pointer (SP). The SP, a 16-bit CPU register, can be used to reference a value in the first 64K words of data memory (addresses 00 0000_{16}−00 $FFFF_{16}$).

stack pointer (SP) — A 16-bit CPU register that enables you to use any portion of the first 64K words of data memory as a software stack. The SP always points to the next empty location in the stack.

stack pointer alignment (SPA) bit — A bit in status register ST1 that indicates whether an ASP instruction has forced the SP to align to the next even address (SPA = 1).

stack-pointer indirect addressing mode — The indirect addressing mode that references a data-memory value at the current position of the stack pointer (SP). See also *PAGE0 stack addressing mode.*

status register ST0 and ST1 — These CPU registers contain control bits that affect the operation of the C28x and contain flag bits that reflect the results of operations.

STEP command — A debugger command that causes the debugger to single-step through a program. The STEP1 command causes the debugger to execute a single instruction.

stop mode — An emulation mode that provides complete control of program execution. When the CPU is halted in stop mode, all interrupts (including reset and nonmaskable interrupts) are ignored until the CPU receives a directive to run code again. See also *real-time mode.*

suppress sign extension — Prevent sign extension from occurring during a shift operation. See also *sign extend.*

SXM bit — See *sign-extension mode (SXM) bit.*

T register — The primary function of this register, also called the multiplicand register, is to hold one of the values to be multiplied during a multiplication. The following shift instructions use the four LSBs to hold the shift count: ASR (arithmetic shift right), LSL (logical shift left), LSR (logical shift right), and SFR (shift accumulator right). The T register can also be used as a general-purpose 16-bit register.

TAP — See *test access port (TAP).*

target device/system — The device/system on which the code you have developed is executed.

TC bit — See *test/control flag (TC).*

test access port (TAP) — A bit in status register ST0 that shows the result of a test performed by the TBIT (test bit) instruction or the NORM (normalize) instruction.

test/control flag (TC) — A bit in status register ST0 that shows the result of a test performed by the TBIT (test bit) instruction or the NORM (normalize) instruction.

test-logic-reset — A test and emulation logic condition that occurs when the TRST signal is pulled low or when the TMS signal is used to advance the JTAG state machine to the TLR state. This logic is a different type than that used by the CPU, which resets functional logic.

32-bit operation — An operation that reads or writes 32 bits.

TI extension pins — See *EMU0* and *EMU1 pins.*

time-critical interrupt — An interrupt that must be serviced even when background code is halted. For example, a time-critical interrupt might service a motor controller or a high-speed timer. See also *debug interrupt enable register (DBGIER).*

USER1-USER12 interrupts— The interrupt vector table contains twelve locations for user-defined software interrupts. These interrupts, called USER1−USER12 in this document, can be initiated only by way of the TRAP instruction.

V bit (overflow flag) — A bit in status register ST0 that indicates when the result of an operation causes an overflow in the location holding the result (V = 1). If no overflow occurs, V is not modified.

vector — See *interrupt vector.*

vector location — See *interrupt vector location.*

vector map (VMAP) bit — A bit in status register ST1 that determines the addresses to which the interrupt vectors are mapped. When VMAP = 0, the interrupt vectors are mapped to addresses 00 0000_{16}−00 $003F_{16}$ in program memory. When VMAP = 1, the vectors are mapped to addresses 3F $FFC0_{16}$−3F $FFFF_{16}$ in program memory.

vector table — See *interrupt vector table.*

W phase — See *write (W) phase.*

wait state — A cycle during which the CPU waits for a memory or peripheral device to be ready for a read or write operation.

watchpoint — A place in a routine where it is to be halted if an address or an address and data combination match specified compare values. When a watchpoint is reached, the routine is halted and the CPU enters the debug-halt state.

word — In this document, a word is 16 bits unless specifically stated to be otherwise.

write (W) phase — The last of eight pipeline phases an instruction passes through. In this phase, if a value or result is to be written to memory, the CPU sends to memory the destination address and the data to be written. See also *pipeline phases.*

zero fill — Fill the unused low- and/or high-order bits of a value with 0s.

zero flag (Z) — A bit in status register ST0 that indicates when the result of an operation is 0 (Z = 1).

Revision History

Changes from January 30, 2009 to May 30, 2015 **Page**

- Chapter 2: **Central Processing Unit** 20
- Section 2.4: Added a note to further defineBbit 12 41
- Chapter 3: **CPU Interrupts and Reset** 52
- Section 3.3.2: Added this second note preceding Figure 3-2 56
- Chapter 4: **Pipeline** 67
- Section 4.1: In Decode 2, revised the sentence beginning "Once an instruction reaches..." 68
- Section 4.4.3: Added this new section 76
- Chapter 6: **C28x Assembly Language Instructions** 107
- AND IER: Added to the Description, "Any changes take effect before the next instruction is processed" 149
- CLRC Mode: Added to the Description, "Any change affects the next instruction in the pipeline" 175
- DINT: Added to the Description, "Any change affects the next instruction in the pipeline" 185
- EINT: Added to Description, "Any change affects the next instruction in the pipeline" 191
- MOV IER: Added to the Description, "Any changes take effect before the next instruction is processed" 258
- mov LOC16: Modified the Repeat text for MOV loc16,#16bit 259
- OR IER: Added to the Description, "Any changes take effect before the next instruction is processed" 346
- POP XARn:The opcodes have been changed 365
- SETC Mode: Added to the Description, "Any change affects the next instruction in the pipeline" 402
- Appendix D: **C2xLP Instruction Set Compatibility** 522
- Section D.1: Added the note at the end of the section 523

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, enhancements, improvements and other changes to its semiconductor products and services per JESD46, latest issue, and to discontinue any product or service per JESD48, latest issue. Buyers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All semiconductor products (also referred to herein as “components”) are sold subject to TI’s terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its components to the specifications applicable at the time of sale, in accordance with the warranty in TI’s terms and conditions of sale of semiconductor products. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by applicable law, testing of all parameters of each component is not necessarily performed.

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

TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right relating to any combination, machine, or process in which TI components or services are used. Information published by TI regarding third-party products or services does not constitute a license 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 significant portions of TI 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. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

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

Buyer acknowledges and agrees that it is solely responsible for compliance with all legal, regulatory and safety-related requirements concerning its products, and any use of TI components in its applications, notwithstanding any applications-related information or support that may be provided by TI. Buyer represents and agrees that it has all the necessary expertise to create and implement safeguards which anticipate dangerous consequences of failures, monitor failures and their consequences, lessen the likelihood of failures that might cause harm and take appropriate remedial actions. Buyer will fully indemnify TI and its representatives against any damages arising out of the use of any TI components in safety-critical applications.

In some cases, TI components may be promoted specifically to facilitate safety-related applications. With such components, TI’s goal is to help enable customers to design and create their own end-product solutions that meet applicable functional safety standards and requirements. Nonetheless, such components are subject to these terms.

No TI components are authorized for use in FDA Class III (or similar life-critical medical equipment) unless authorized officers of the parties have executed a special agreement specifically governing such use.

Only those TI components which TI has specifically designated as military grade or “enhanced plastic” are designed and intended for use in military/aerospace applications or environments. Buyer acknowledges and agrees that any military or aerospace use of TI components which have ***not*** been so designated is solely at the Buyer's risk, and that Buyer is solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI has specifically designated certain components as meeting ISO/TS16949 requirements, mainly for automotive use. In any case of use of non-designated products, TI will not be responsible for any failure to meet ISO/TS16949.

Products		**Applications**	
Audio	www.ti.com/audio	Automotive and Transportation	www.ti.com/automotive
Amplifiers	amplifier.ti.com	Communications and Telecom	www.ti.com/communications
Data Converters	dataconverter.ti.com	Computers and Peripherals	www.ti.com/computers
DLP® Products	www.dlp.com	Consumer Electronics	www.ti.com/consumer-apps
DSP	dsp.ti.com	Energy and Lighting	www.ti.com/energy
Clocks and Timers	www.ti.com/clocks	Industrial	www.ti.com/industrial
Interface	interface.ti.com	Medical	www.ti.com/medical
Logic	logic.ti.com	Security	www.ti.com/security
Power Mgmt	power.ti.com	Space, Avionics and Defense	www.ti.com/space-avionics-defense
Microcontrollers	microcontroller.ti.com	Video and Imaging	www.ti.com/video
RFID	www.ti-rfid.com		
OMAP Applications Processors	www.ti.com/omap	**TI E2E Community**	e2e.ti.com
Wireless Connectivity	www.ti.com/wirelessconnectivity		

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