SYSTEM DEVELOPMENT DOCUMENTATION

Revision 2
December 1, 1994

DSP Core

Digital Signal Processor Core User's Manual

adaptec®

NOTICE

This manual describes the proprietary 16-bit general-purpose Digital Signal Processor (DSP) Core, used in Adaptec's AIC-4411 DMC. DMC is an acronym for Drive Manager Chip containing a DSP core (PINE™) and proprietary circuitry on a single IC. The words PINE and DM (Drive Manager) may, at times, be used interchangeably in the text.

The information contained in this document is subject to change without notice.

Copyright© 1994 Adaptec, Inc. All rights reserved. This document contains proprietary information which is protected by U.S. and international copyright laws. It may not be used, copied, distributed, or disclosed without the express written permission of Adaptec, Inc.

Adaptec and the Adaptec logo are registered trademarks of Adaptec, Inc. All other trademarks used are held by their respective owners.

PINE, PINEASM, and COFFLINK are trademarks of DSP Semiconductors USA, Inc.

Microsoft and MS-DOS are registered trademarks of Microsoft Corporation.

```
;============================================================================
;00xxxxxxxxxxxxxx       PINE CONTROL OPCODES:         0000-3fff
;============================================================================
;0000000xxxxxxxxx,      3.13c,  nop                   0000-01ff
;00000010xxxxxxxx,      3.14,   trap                  0200-02ff
;00000011xxxxxxxx       unused [8]                    0300-03ff
;0000010xxxxxxxxx,      3.13a,  eint                  0400-05ff
;0000011xxxxxxxxx,      3.13b,  dint                  0600-07ff
;0000100xiiiiiiii,      3.6b,   lpg #imm              0800-09ff
;0000101xxxxxxxxx       unused [9]                    0a00-0bff
;0000110AxxxmmNNN,      3.10,   norm ax, (Rn)+        0c00-0dff       6/8 Rns
;0000111xxxxmmNNN,      3.11,   modr (Rn)+            0e00-0fff       6/8 Rns
;0001000xiiiiiiii,      3.6a,   rep #imm              1000-11ff
;0001001xxxxRRRRR       3.7,    rep reg               1200-13ff       31/32 regs
;0001010xiiiiiiii,      3.8,    bkrep #imm, ##long    1400-15ff
;0001011xxxxRRRRR,      3.9,    bkrep reg, ##long     1600-17ff       31/32 regs
;00011xxAffffcccc,      3.12,   moda funct, ax, cond  1800-1fff       13 func
;00100rrrrrrrcccc,      3.1a,   callr $+r, cond       2000-27ff
;00101rrrrrrrcccc,      3.1b,   brr $+r, cond         2800-2fff
;0011000xxxxxcccc,      3.2a,   call ##long, cond     3000-31ff
;0011001xxxxxcccc,      3.2b,   br ##long, cond       3200-33ff
;0011010xxxxxcccc,      3.3,    ret cond              3400-35ff
;0011011xxxxxxxxx,      3.4,    reti                  3600-37ff
;0011100Axxxxxxxx,      3.5,    calla axl             3800-39ff
;0011101xxxxxxxxx       unused [9]                    3a00-3bff
;0011110Aaaaaaaaa,      3.15,   divs diradd, ax       3c00-3dff
;0011111xxxxxxxxx       unused [9]                    3e00-3fff
;============================================================================
;01xxxxxxxxxxxxxx       PINE MOVE OPCODES:            4000-7fff
;============================================================================
;0100000AxxxRRRRR,      2.8,    movp (axl), reg       4000-41ff       31/32 regs
;0100001iiIImmNNN,      2.9,    movp (Rn)+, (Ri)+     4200-43ff       6/8 Rns
;010001xxxxxxxxxx       unused [10]                   4400-47ff
;010010RRRRRmmNNN,      2.3a,   mov reg, (Rn)+        4800-4bff       31/32 regs, 6/8 Rns
;010011RRRRRmmNNN,      2.3b,   mov (Rn)+, reg        4c00-4fff       6/8 Rns, 31/32 regs
;01010NNNssssssss,      2.7,    mov #+imm, Rn*        5000-57ff
;010110DDDDDSSSSS,      2.5,    mov sreg, dreg        5800-5bff       31/32 regs
;0101110xxxxRRRRR,      2.4,    mov ##long, reg       5c00-5dff       31/32 regs
;0101111xxxxxxxxx       unused [9]                    5e00-5fff
;0110000Aaaaaaaaa,      2.1b,   mov diradd, ax        6000-61ff
;0110001Aaaaaaaaa,      2.1d,   mov diradd, axh       6200-63ff
;0110010Aaaaaaaaa,      2.1c,   mov diradd, axl       6400-65ff
;110011Aaaaaaaaa,       2.1e,   mov diradd, axh, eu   6600-67ff
;1101NNNaaaaaaaa,       2.6a,   mov Rn*, diradd       6800-6fff
;1110NNNaaaaaaaa,       2.6b,   mov diradd, Rn*       7000-77ff
;11110HAaaaaaaaa,       2.1a,   mov axH, diradd       7800-7bff
;0111110Aiiiiiiii,      2.2a,   mov #imm, axl         7c00-7dff
;0111111Assssssss,      2.2b,   mov #+imm, axh        7e00-7fff
;============================================================================
;1xxxxxxxxxxxxxxx       PINE ALU OPCODES:             8000-ffff
;============================================================================
;100ooooA00xRRRRR,      1.3a,   oper reg, ax          8000-801f, 8100-811f     31/32 regs
;100ooooA00xRRRRR,      1.3b,   mpy  y, reg           8000-801f, 8100-811f     31/32 regs
;100ooooA00xRRRRR,      1.3c,   mac  y, reg, ax       8000-801f, 8100-811f     31/32 regs
;100ooooA00xRRRRR,      1.3d,   sqr  reg              8000-801f, 8100-811f     31/32 regs
;100ooooA00xRRRRR,      1.3e,   sqra reg, ax          8000-801f, 8100-811f     31/32 regs
;10000xxx01xxxxxx       unused [3,6]                  8040, etc
;100010xx01xxxxxx       unused [2,6]                  8840, 8940, 8a40, 8b40
;1000110x01xmmNNN,      1.6a,   mpy (Rn)+, ##long     8c40-8c5f, 8d40-8d5f     6/8 Rns
;1000111A01xmmNNN,      1.6b,   mac (Rn)+, ##long, ax 8e40-8e5f, 8f40-8f5f     6/8 Rns
;1001000A01xmmNNN,      1.6c,   msu (Rn)+, ##long, ax 9040-905f, 9140-913f     6/8 Rns
;1001001x01xxxxxx       unused [1,6]                  9240-927f, 9340-937f
;100101xx01xxxxxx       unused [2,6]                  9440, 9540, 9640, 9740
;10011xxx01xxxxxx       unused [3,6]                  9840, etc
;100ooooA10xmmNNN,      1.2a,   oper (Rn)+, ax        8080-809f, 8180-819f     6/8 Rns
;100ooooA10xmmNNN,      1.2b,   mpy  y, (Rn)+         8080-809f, 8180-819f     6/8 Rns
;100ooooA10xmmNNN,      1.2c,   mac  y, (Rn)+, ax     8080-809f, 8180-819f     6/8 Rns
;100ooooA10xmmNNN,      1.2d,   sqr  (Rn)+            8080-809f, 8180-819f     6/8 Rns
;100ooooA10xmmNNN,      1.2e,   sqra (Rn)+, ax        8080-809f, 8180-819f     6/8 Rns
;1000000A11xxxxxx,      1.5a,   add ##long, ax        80c0, 81c0
;1000001A11xxxxxx,      1.5b,   sub ##long, ax        82c0, 83c0
;1000010A11xxxxxx,      1.5c,   or ##long, ax         84c0, 85c0
;1000011A11xxxxxx,      1.5d,   xor ##long, ax        86c0, 87c0
;1000100A11xxxxxx,      1.5e,   and ##long, ax        88c0, 89c0
;1000101A11xxxxxx,      1.5f,   cmp ##long, ax        8ac0, 8bc0
;100011xx11xxxxxx       unused [2,6]                  8cc0, 8dc0, 8ec0, 8fc0
;1001xxxx11xxxxxx       unused [4,6]                  9ec0, etc
;101ooooAaaaaaaaa,      1.1a,   oper diradd, ax       a000-bfff
;101ooooAaaaaaaaa,      1.1b,   mpy  y, diradd        a000-bfff
```



```
;101ooooAaaaaaaaa,      1.1c,   mac  y, diradd, ax          a000-bfff
;101ooooAaaaaaaaa,      1.1d,   sqr  diradd                 a000-bfff
;101ooooAaaaaaaaa,      1.1e,   sqra diradd, ax             a000-bfff
;11000xxxxxxxxxxx       unused [11]                         c000-c7ff
;110010xxxxxxxxxx       unused [10]                         c800-cbff
;1100110x0jjiiJII,      1.7a,   mpy (Rj)+, (Ri)+            cc00-cc7f, cd00-cd7f
;1100110x1xxxxxxx       unused [1,7]                        cc80-ccff, cd80-cdff
;1100111A0jjiiJII,      1.7b,   mac (Rj)+, (Ri)+, ax        ce00-ce7f, cf00-cf7f
;1100111x1xxxxxxx       unused [1,7]                        ce80-ceff, cf80-cfff
;1101000A0jjiiJII,      1.7c,   msu (Rj)+, (Ri)+, ax        d000-d07f, d100-d17f
;1101000x1xxxxxxx       unused [1,7]                        d080-d0ff, d180-d1ff
;1101001xxxxxxxxx       unused [9]                          d200-d3ff
;110101xxxxxxxxxx       unused [10]                         d400-d7ff
;11011xxxxxxxxxxx       unused [11]                         d800-dfff
;1110oooAiiiiiiii,      1.4a,   oper #imm, ax               e000-ebff
;1110110Assssssss,      1.4b,   mpys #+imm, ax              ec00-edff
;1110111xxxxxxxxx       unused [9]                          ee00-efff
;1111xxxxxxxxxxxx       unused [12]                         f000-ffff
;----------------------------------------------------------------------------
```

16.3 DSP INSTRUCTION SET--execution bus cycle times (0-wait):

INSTRUCTION	DESCRIPTION	FLAGS	Reg	Ind	Dir	SIm	LIm
add op, ax	ax += +op	ZMNVCEL-	1	1	1	1	2
addh op, ax	ax += +op << 16> ax	ZMNVCEL-	1	1	1		
addl op, ax	ax += op	ZMNVCEL-	1	1	1		
and op, ax	ax &= op	ZMN--E--	1	1	1	1	2
bkrep, op, add	lc = op; lp flag = 1	--------	2			2	
br add [,cond]	pc = add	--------					2
brr $off [,cond]	pc += off	--------				2	
call add [,cond]	tos = pc; pc = add	--------					2
calla axl	tos = pc; pc = axl	--------	2				
callr $off [,cond]	tos = pc; pc += off	--------				2	
clr ax [,cond]	ax = 0	ZMN--E--	1				
clrr ax [,cond]	ax = 0x8000	ZMN--E--	1				
cmp op, ax	flags = ax - +op	ZMNVCEL-	1	1	1	1	2
copy ax [,cond]	a0 = a1 or a1 = a0	ZMN--E--	1				
dint	ie flag = 0	--------	1				
divs dir, ax	unsigned divide step	ZMN--E--			2		
eint	ie flag = 1	--------	1				
lpg #op	page register = op	--------				1	
mac op1, op2, ax	ax += p; p = op1 * op2	ZMNVCEL-	1	1	1		2
modr (rn)+	rn += 0, 1, -1, or step	-------R	1				
mov sop, dop [,eu]	dop = sop (data space)	--------	1	1	1	1	2
movp sop, dop	dop = sop (pgm space)	--------		3			
mpy op1, op2	p = (y=op1) * (x=op2)	--------	1	1	1		2
mpys y, #op	p = y * (x=op)	--------				1	
msu op1, op2, ax	ax -= p; p = op1 * op2	ZMNVCEL-	1	1	1		2
neg ax [,cond]	ax = -ax	ZMNVCEL-	1				
nop	no operation	--------	1				
norm ax, rn	if !N ax <<= 1; modr	ZMNVCELR	2				
not ax [,cond]	ax = ~ax	ZMN--E--	1				
or op, ax	ax |= op	ZMN--E--	1	1	1	1	2
pacr ax [,cond]	ax = p + 0x8000	ZMNVCEL-	1				
rep op	repeat next op times	--------	1			1	
ret [,cond]	pc = tos	--------	2				
reti	pc = tos; ie flag = 1	--------	2				
rnd ax [,cond]	ax += 0x8000	ZMNVCEL-	1				
rol ax [,cond]	rotate left through c	ZMN-CE--	1				
ror ax [,cond]	rotate right through c	ZMN-CE--	1				
shl ax [,cond]	ax <<= 1	ZMNVCE--	1				
shl4 ax [,cond]	ax <<= 4	ZMNVCE--	1				
shr ax [,cond]	ax >>= 1	ZMN0CE--	1				
shr4 ax [,cond]	ax >>= 4	ZMN0CE--	1				
sqr op	p = (y=op) * (x=op)	--------	1	1	1		
sqra op, ax	ax += p; p = op * op	ZMNVCEL-	1	1	1		
sqrs op, ax	ax -= p; p = op * op	ZMNVCEL-	1	1	1		
sub op, ax	ax -= +op	ZMNVCEL-	1	1	1	1	2
subh op, ax	ax -= +op << 16	ZMNVCEL-	1	1	1		
subl op, ax	ax -= op	ZMNVCEL-	1	1	1		
trap	tos = pc; pc = 10; ie=0	--------	2				
xor op, ax	ax ^= op	ZMN--E--	1	1	1	1	2

pacr

round product

Syntax: pacr aX

Operation: shifted p + 0x8000 -> aX

Affects flags:

Z	M	N	V	C	E	L	R
*	*	*	*	*	*	*	–

rep

repeat next instruction

Syntax: rep operand

Operand:
#short immediate
reg (except aX, p)

Operation: Begins a noninterruptible single word instruction loop, to be repeated operand + 1 (1..256) times.

Affects flags: No

ret

conditional return from subroutine

Syntax: ret [cond]

Operation:
```
If condition then
   tos -> pc
```

Affects flags: No

reti

return from interrupt

Syntax: reti

Operation:
```
tos -> pc
1 -> ie
```

Affects flags: No

mac

multiply and accumulate previous product

Syntax: mac operand1, operand2, aX

Operands:

```
y, direct address
y, (rn)
y, reg                          (except aX, p)
(rj), (ri)                      (XRAM & YRAM)
(rn), ##long immediate
```

Operation:

```
aX + shifted p -> aX
operand1 -> y
operand2 -> x
x * y -> p
```

Affects flags:

Z	M	N	V	C	E	L	R
*	*	*	*	*	*	*	-

moda

modify accumulator conditionally

Syntax: [moda] Function , aX [, cond]

Operation:

```
If condition then
   aX is modified by 'Function'
```

Function:

```
shr     aX = aX >> 1
shl     aX = aX << 1
shr4    aX = aX >> 4
shl4    aX = aX << 4
ror     Rotate aX right through carry
rol     Rotate aX left  through carry
not     aX = not(aX)
neg     aX = -aX
clr     aX = 0_
copy    aX = aX
rnd     aX = aX + 0x8000
pacr    aX = shifted p + 0x8000
clrr    aX = 0x8000
```

Affects flags: According to function, when condition is true.

modr

Modify r_n

Syntax: modr (r_n)

Operation: r_n is modified.

Affects flags:

Z	M	N	V	C	E	L	R
-	-	-	-	-	-	-	*

Note: R flag is set if r_n register is zero, otherwise cleared.

Table of Contents

SECTION 1 - Introduction **1**

1.1 General Description 1

1.2 Document Organization 1

1.3 Related Documentation 1

SECTION 2 - Architecture Features **3**

2.1 Technology Features 3

SECTION 3 - Programming Model **5**

3.1 General Description 5

3.2 Buses 7

3.2.1 Data Buses 7

3.2.2 Address Buses 7

3.3 Computation Unit 7

3.3.1 Multiplier 7

3.3.2 Data ALU/Shifter 7

3.4 Data Address Arithmetic Unit (DAAU) 8

3.4.1 Address Modification 8

3.5 Program Control Unit (PCU) 10

3.5.1 Program Address Arithmetic Unit (PAAU) 10

3.6 Memory Spaces and Organization 12

3.6.1 Data Memory 12

3.6.2 Program Memory 13

3.6.3 Memory Addressing Modes 14

3.7 Programming Model and Registers 15

3.7.1 Programming Model 15

3.7.2 Status Registers 17

3.7.3 Stack 22

3.7.4 User-Defined Registers 22

3.8 Input and Output 22

SECTION 4 - Instruction Set **23**
4.1 Introduction 23
4.2 Notation and Conventions 23
4.2.1 Notation 23
4.2.2 Conventions 26
4.3 Instruction Set Summary 27
4.4 Instruction Set Details 29
4.4.1 Arithmetic and Logical Instructions 29
4.4.2 Multiply Instructions 46
4.4.3 Move Instructions 51
4.4.4 Loop Instructions 55
4.4.5 Branch/Call Instructions 57
4.4.6 Control and Miscellaneous Instructions 61
4.5 Instruction Execution 64
4.5.1 Pipeline Method 64

SECTION 5 - Core Interface **65**
5.1 Introduction 65
5.2 Clock 65
5.3 Reset 65
5.4 Interrupts 66
5.4.1 BPI, INT0, INT1 66

SECTION 1
Introduction

1.1 General Description

Drive Manager's DSP (PINE) is a DSP engine for the Adaptec AIC-4411 Drive Manager Chip (DMC). It enables low-cost, low-power DSP processing. The core consists of the main blocks of a Central Processing Unit (CPU), including the ALU, multiplier, accumulators, RAM and ROM addressing units, and the program control logic.

The DSP consists of three main execution units operating in parallel: the Computation Unit (CU), the Data Addressing Arithmetic Unit (DAAU) and the Program Control Unit (PCU). It has two blocks of data RAM for parallel feeding of two inputs to the multiplier. The CU has a 16 x 16 multiplier, 36-bit ALU, and two 36-bit accumulators. The DSPDSP programming model and instruction set are aimed at straight forward generation of efficient and compact code.

1.2 Document Organization

The key features of the DSP core are described in Section 2. The core block diagram and detailed descriptions of each block are given in Section 3. The DSP instruction set is explained in Section 4. Section 5 describes the interface to the DSP core for ASIC design purposes, including details on clocking, exception handling.

1.3 Related Documents

DM_DBG User's Manual

DM_ASM and DM_COFFLINK User's Manual

AIC-4411 Drive Manager Chip Data Sheet

AIC-4411 Drive Manager Chip ROM Code User's Guide

This page intentionally left blank.

SECTION 2
Architecture Features

2.1 Technology Features

- 16x16 bit 2's complement parallel multiplier with 32-bit product.
- Single cycle multiply/accumulate instructions.
- 36-bit ALU.
- Two 36-bit accumulators.
- On-chip 1Kx16 bit program ROM operation. (See Adaptec's current "AIC-4411 Drive Manager Chip ROM Code User's Guide" for details.)
  - Diagnostic commands
  - Code download commands
- 32Kx16 bit maximum addressable off-chip program/data space and 24Kx16 bit maximum off-chip program-only memory space.
- Areas of the 32K off-chip program/data memory can be programmed in 8K blocks for access using either the RAM_CS or ROM_CS Chip Select signals.
- 2x256x16 bit on-core data RAM.
- On-chip 2Kx16 bit ROM and on-chip 2Kx16 bit RAM is accessible via program or data buses on the same address space.
- Data RAMs can be also viewed for programmer as a single continuous RAM.
- 16 level hardware stack.
- (4+2) x16 bit general purpose pointer-registers with 2 dedicated Address Arithmetic Units for data memory (RAM/ROM) indirect addressing, circular buffering, loop counters, and program memory indirect addressing.
- All general and most special-purpose registers are arranged as a global register set of 31 registers for most data move and ALU instructions.
- Shifting capability:
  - Accumulator shift +4, +1, -1, -4
  - Accumulator rotate +1, -1
  - Product register to accumulator +1, 0, -1.
- 2Kx16 bit common program/data RAM.
- 2Kx16 bit common program data ROM.
- Memory mapped I/O space (A000h-BFFFh) configurable to access any 8K-block of entire 64K off-chip program space, which can be used for Flash chip programming.

- Automatic saturation mode on overflow while reading content of accumulators.
- Zero overhead looping, REPEAT and BLOCKREPEAT instructions with one nesting level.
- Memory mapped I/O.
- Wait state support for off-chip memory or I/O.
- STOP mode of operation for stopping the core.
- Interrupts and exceptions:
  - 1 reset
  - 2 maskable interrupts
  - 1 TRAP (software interrupt)
- Divide step support.
- Normalize step support.

3.1 General Description

A high-level block diagram of the DSP architecture is shown in Figure 3-1. The major components of the DSP core are:

- Data Buses - XDB, YDB, PDB
- Address Buses - XAB, YAB, PAB
- Multiplier
  * Input registers - X, Y
  * Output register - P
  * Output shifter
- Data ALU/Shifter
  * Output accumulators - A0, A1
  * Saturation logic - SATU
- Data Address Arithmetic Unit - DAAU
  * DAAU registers - R0 ÷ R3, R4 ÷ R5
  * DAAU config. registers - CFGI, CFGJ
- Program Control Unit - PCU
  * Program Add. Arith. Unit - PAAU
  * Program decode controller
  * Interrupt controller
- Memories - XRAM, YRAM, (PROM)
- Stack
- Status Registers - ST0, ST1, ST2
- User-Defined Registers (off-core) - EXT0 ÷ EXT7
- Internal Bus Switches
- Input/Output

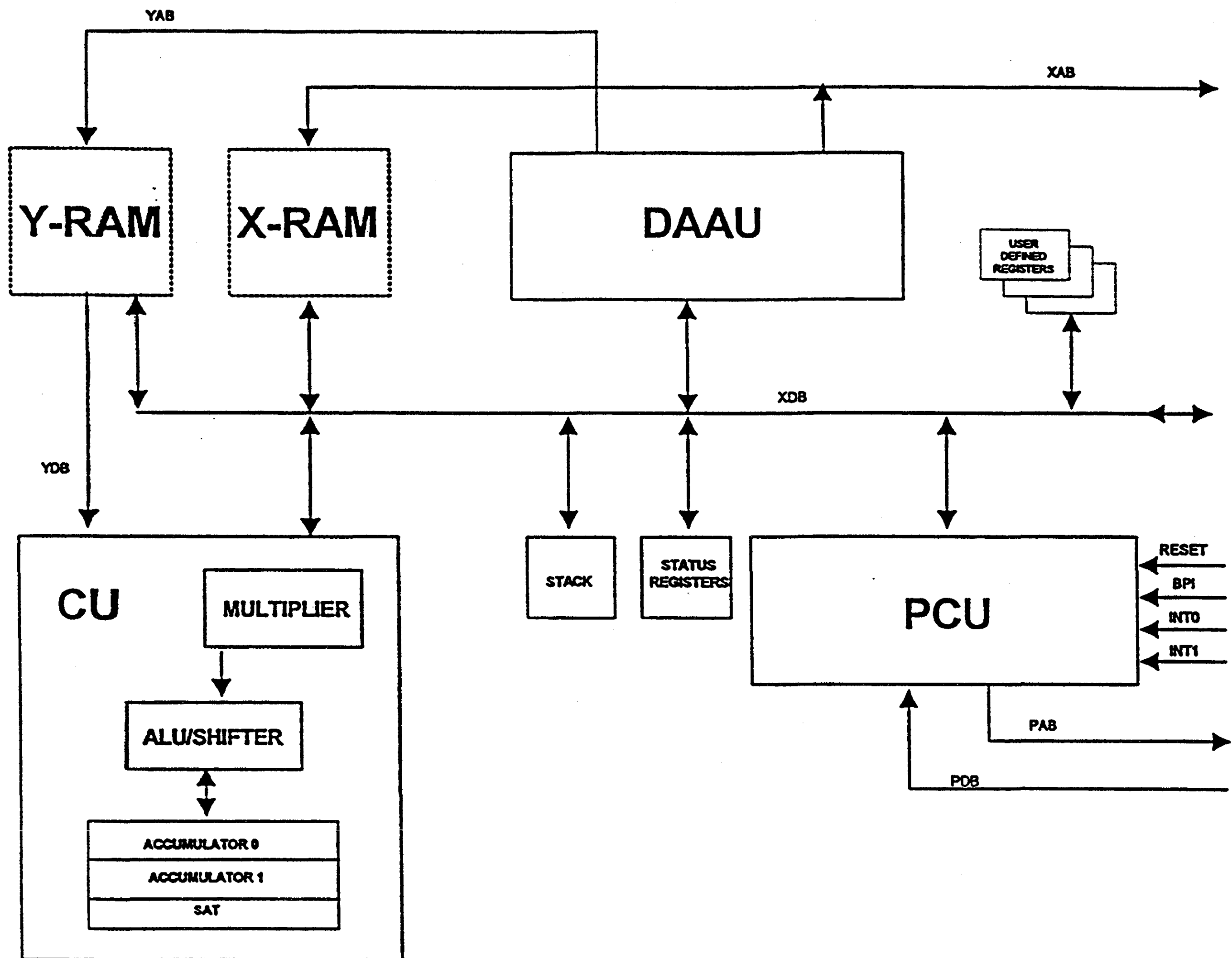


Figure 3-1 DSP Core Block Diagram

3.2 Buses

3.2.1 Data Buses

Data is transferred on the following 16-bit buses: a bidirectional X Data Bus (XDB); and two unidirectional buses - the Y Data Bus (YDB) and the Program Data bus (PDB). Data transfer between the Y Data Memory (YRAM) and the Multiplier (Y register) occurs over the YDB. Instruction word fetches take place in parallel over PDB. The bus structure supports register to register, register to memory, memory to register and program memory to data memory/register data movement. It can transfer up to two 16-bit words in the same instruction cycle.

3.2.2 Address Buses

Addresses are specified for the on-chip XRAM and YRAM on two unidirectional buses: the 16 bit X Address Bus (XAB), and the 10-bit Y Address Bus (YAB). Program memory addresses are specified on the 16-bit unidirectional Program Address Bus (PAB).

3.3 Computation Unit

3.3.1 Multiplier

The multiplier unit consists of a 16x16 to 32 bit parallel, single- cycle, non-pipelined multiplier, two 16-bit input registers (X and Y), a 32-bit output register (P), and an output shifter. Together with the Data ALU, PINE can perform a single-cycle Multiply-Accumulate (MAC) instruction. The P register is updated only after a multiply instructions and not after a change in the X and/or Y registers.

The X and Y registers may be read or written by the XDB, and the Y register by the YDB, as 16-bit operands. The 16-bit Most Significant Portion (MSP) of the P register, PH, may be written by the XDB as an operand. This enables a single-cycle restore of PH during interrupt service routine. The complete 32-bit P register can be used only by the ALU and can be moved only to the two accumulators.

The X and Y registers can be also used as general purpose temporary data registers.

The (P) register is sign extended into 36 bits and then shifted.

3.3.2 Data ALU/Shifter

The Data ALU/Shifter performs all arithmetic, logical and shifting operations on data operands. The Data ALU/Shifter consists of a 36-bit, parallel, single-cycle, non-pipelined ALU/Shift unit (ALU/S), two 36-bit accumulator registers (A0 and A1), and a saturation unit (SATU).

The Data ALU/Shifter receives one operand from Ax (x=0,1), and another operand from either the output shifter of the multiplier, the XDB (through the bus alignment logic), or from Ax. The source operands may be 16 or 36 bits. Operations between the two accumulators are also possible. ALU/S results are stored in one of the accumulators. The source and destination accumulator of an instruction is always the same.

The ALU/S can perform positive or negative accumulate, add, subtract, compare, shift, logical, and several other operations, most of them in one instruction cycle. It uses a two's complement arithmetic.

Unless otherwise specified, in all operations between the 16-bit operand and Ax (36 bit), the 16-bit operand will be regarded as the LSP of a 36-bit operand with a sign extension for arithmetic operations and a zero extension for logic operation. The ADDH, SUBH, ADDL and SUBL instructions are used when this convention is not adequate in arithmetic operation (refer to these instructions in Section 4).

Registers AxH and AxL can also be used as general-purpose temporary data registers.

3.4 Data Address Arithmetic Unit (DAAU)

The DAAU performs all address storage and effective address calculations necessary to address data operands in data and program memories. In addition, it supports loop counter operations in conjunction with the MODR instruction (see Chapter 4 on Instruction Set) and the R flag (see to Paragraph 3.7.2 on Status Registers). This unit operates in parallel with other core resources to minimize address generation overhead. The DAAU can implement two types of arithmetic: linear and modulo. The DAAU contains six 16-bit address registers (R0-R3 and R4-R5) for indirect addressing, and two 16-bit configuration registers (CFGI and CFGJ) for modulo and increment/ decrement step control. The registers are divided into two groups for simultaneous addressing over XAB and YAB (or PAB): R0-R3 with CFGI; and R4-R5 with CFGJ. Registers from both groups can be used for both XAB and YAB (or PAB) for instructions which use only one address register. In addition, in these instructions the XRAM and YRAM can be viewed as a single continuous data memory space.

All DAAU registers may be read or written by the XDB as 16-bit operands, thus serving as general-purpose registers.

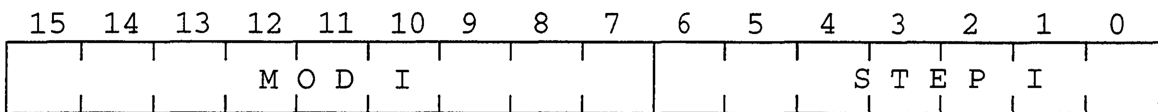
3.4.1 Address Modification

The DAAU can generate two 16-bit addresses every instruction cycle which can be post-modified by two modifiers: linear (step) and modulo modifier. The address modifiers allow the creation of data structures in memory for circular buffers, delay lines, FIFOs, software stacks, etc. They can also be used when the Rn registers are used as loop counters in conjunction with the MODR instruction (see Section 4 on Instruction Set) and the R flag of ST0 (see Section 3.7.2 on Status Registers). Address modification is performed using 16-bit (modulo 65,536) two's complement linear arithmetic. The range of values of the registers may be considered as signed (from -32,768 to +32,767) or unsigned (from 0 to +65,536).

3.4.1.1 Linear (Step) Modifier

During one instruction cycle, one or two (from different groups) of the address register, Rn, can be post increment/decrement by 1 or by a 2's complement 7-bit step (from -64 to +63). The selection of linear modifier type (one out of four) is included in the relevant instructions (see Section 4.2.2 on Conventions for Instruction set). Step values STEPI and STEPJ are stored as the 7 LSB of the configuration register CFGI and CFGJ respectively.

CFGI



CFGJ

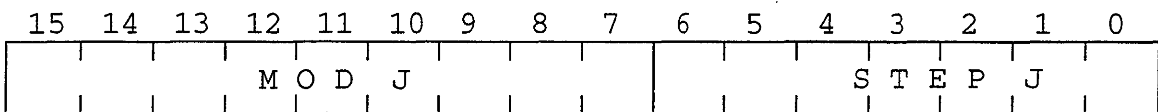


Figure 3-2 Configuration Registers

3.4.1.2 Modulo Modifier

The two modulo arithmetic units can update one or two address registers from different groups during one instruction cycle. They are capable of performing modulo calculations of up to 2**9. Each register can be set independently to be affected or unaffected by the modulo calculation using the six Mn status bits in the ST2 register. Modulo values MODI and MODJ are stored in 9 MSBs of configuration registers CFGI and CFGJ respectively.

For proper modulo calculation, the following constraints must be satisfied (M = modulo factor; q = STEPx, +1 or -1):

1. The lower boundary (base address) must have zeros in at least the k LSBs, where k is the minimal integer that satisfies 2**k > M-|q|
2. MODx (x denotes I or J) must be loaded with M-|q|.
3. M must be an integer multiple of q (this is always true for q=+/-1).

Note: |q| denotes the absolute value of q.

The modulo modifier operation, which is a post-modification of the Rn register, is defined as follows:

Rn <- 0 in k LSB; if Rn is equal to MODx in k LSB and q > 0,

Rn <- MODx in k LSB; if Rn is equal to 0 in k LSB and q < 0,

Rn <- Rn + q ; Otherwise

Note: R0-R3 can only work with STEPI and MODI, while R4-R5 can work only with STEPJ and MODJ.

Examples:

1. M=7 with STEPx=1 (or +1 selected in instruction), MODx=7-1=6, Rn=10H (hex). The sequence of Rn values will be: 10,11,12,13,14,15,16,10,11,...
2. M=8 with STEPx=2, MODx=8-2=6, Rn=10H. The sequence of Rn values will be: 10,12,14,16,10,12,...
3. M=9 with STEPx=-3, MODx=9-|-3|=6, Rn=16H. The sequence of Rn values will be: 16,13,10,16,13,...

3.5 Program Control Unit (PCU)

The Program Control Unit (PCU) performs instruction fetch, instruction decoding, exception handling, and wait state support. In addition, it controls the internal PROM protection (see Section 3.6.2).

3.5.1 Program Address Arithmetic Unit (PAAU)

The Program Address Arithmetic Unit (PAAU) generates the next address to the program memory and controls hardware loops. It contains the Repeat/Block-Repeat unit, and two 16-bit directly-accessible registers: the Program Counter (PC) and the Loop Counter (LC) of the block-repeat unit.

The PAAU selects and/or calculates the next address from several possible sources: the incremented PC in sequential program flow; PROM address in branch or call operations; short PC-relative address of 7-bit in relative branch operations; start address of hardware loop; interrupts vector in interrupt handling; or the Top of Stack (TOS) upon returning from subroutines and interrupts. It also writes the PC to the TOS in subroutines and interrupts.

The PC always contains the address of the next instruction.

3.5.1.1 Repeat and Block Repeat Unit

The Repeat/Block-Repeat unit performs the hardware-loop calculations and control, with no overhead other than the one-time execution of set-up instructions REP and BKREP. In block-repeat operation, it stores the first and the last addresses of a loop and counts the number of loop repetitions. It consists of two 16-bit dedicated registers for start and end addresses of the block repeat, and two 8-bit repeat and block-repeat counters (1 to 256 repetitions). The start and end address registers as well as the 8-bit repeat counter cannot be accessed as registers by the programmer. The 8-bit block-repeat counter is the 8 LSB of the LC register, which is one of the global registers. The LC register can be used as an index (e.g. address to an array) inside the block-repeat loop or for determining the value of the block-repeat counter when a jump out of the block-repeat loop occurs.

Single-level nesting of a single-instruction repeat in the block-repeat instruction is supported. Interrupts are disabled only during a single-instruction repeat, and when the instruction is being repeated. For details of specific limitations, refer to REP and BKREP instructions in Section 4 on the Instruction Set.

The number of repetitions can be a fixed value in the instruction code or a value contained in one of the processor's 16-bit registers. This supports calculating the number of repetitions in run-time.

For a clean jump (break) from a block-repeat, a special status bit, In-Loop (LP), is available in ST2. It is set when a block-repeat is executed and reset on normal completion of the block-repeat loop. The user must reset it when a jump out from the block-repeat loop occurs. See also Section 3.7.2 on Status Registers.

If the LP bit is cleared in the block repeat loop and not by one of the two last instructions of the loop, the processor is no longer in the loop state. Therefore, there will be no jumps to the first address of the loop and the counter will not be decremented.

If the LP bit is cleared by one of the last two instructions of the loop, its effect will take place only in the next loop. (Unless the only instruction in the block repeat loop clears the LP and the LC is 2 or more, then the loop will be performed 3 times.)

The LC register may also serve as a general-purpose register for temporary storage.

3.6 Memory Spaces and Organization

Two independent memory spaces are available: the data space (XRAM and YRAM) and the program space (PROM). Each is 64K words. The addresses from 1000h to 7FFFh access the same memory from either the program or data buses.

3.6.1 Data Memory

The data space is divided into an X data space for the XRAM (from zero to 64511 <63K-1>), and a Y data space for the YRAM (from -1 to -1024). The range of the data space can also be considered as unsigned, making the XRAM and YROM spaces continuous. The XRAM space has an internal space (on-core data RAM/ROM) of 1K (from 0 to 1023), and an external space of 62K. The YRAM space is internal only. See also Figure 3-3. The above data space partition allows modular expansion of the on-core XRAM and YRAM, and at the same time enables looking at the two RAMs as single continuous data RAM.

The on-core XRAM and YRAM sizes are at least 2x144x16 bits and can be expanded in 2x64x16 bit blocks from 2x128x16 bits (i.e. the next size is 2x192x16 bits, the one after that is 2x256x16 bits, etc.) up to 2x1Kx16 bits. The XRAM and YRAM can also be expanded by data ROM in 64x16 bit blocks up to a total (RAM + ROM) of 2x1Kx16 bits. The X data memory can be expanded off-core (with no additional wait state cycles) up to the YRAM boundary.

On-chip DMC peripherals are memory mapped I/O into the data space at address Exxxh. Wait state generation can be supported for off-chip memory. (See the current AIC-44XX Drive Manager Chip Data Sheet for programming details.)

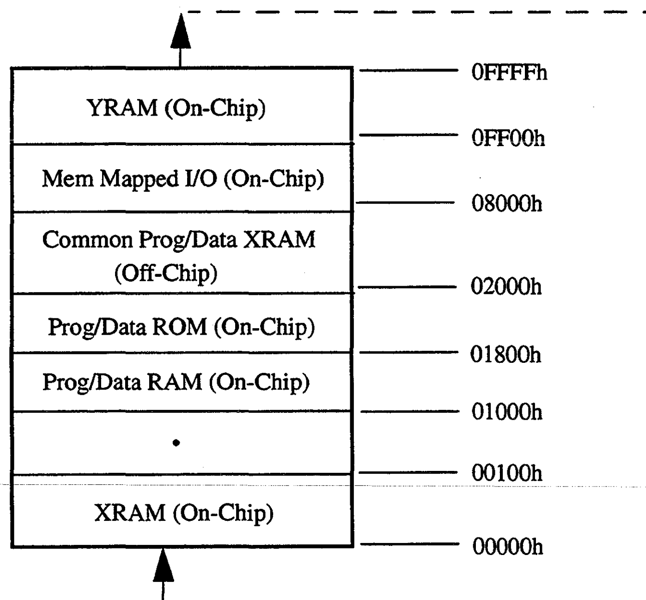


Figure 3-3 Data Memory Map

3.6.2 Program Memory

The program space starts at address 00010H. Addresses 0000H-0001H are used for Reset; and addresses 0008H-000FH are used for TRAP, BPI (Breakpoint interrupt), and two maskable interrupts respectively. Each interrupt address has been separated by two locations so that branch instructions can be accommodated in those locations if desired. Addresses 0002H-0007H are reserved (see Figure 3-4.).

The internal PROM is 1K words and can be extended in 1K-word blocks up to 32K words. The program space may be expanded off-chip up to 64K words.

The program memory addresses are generated by the PCU

.

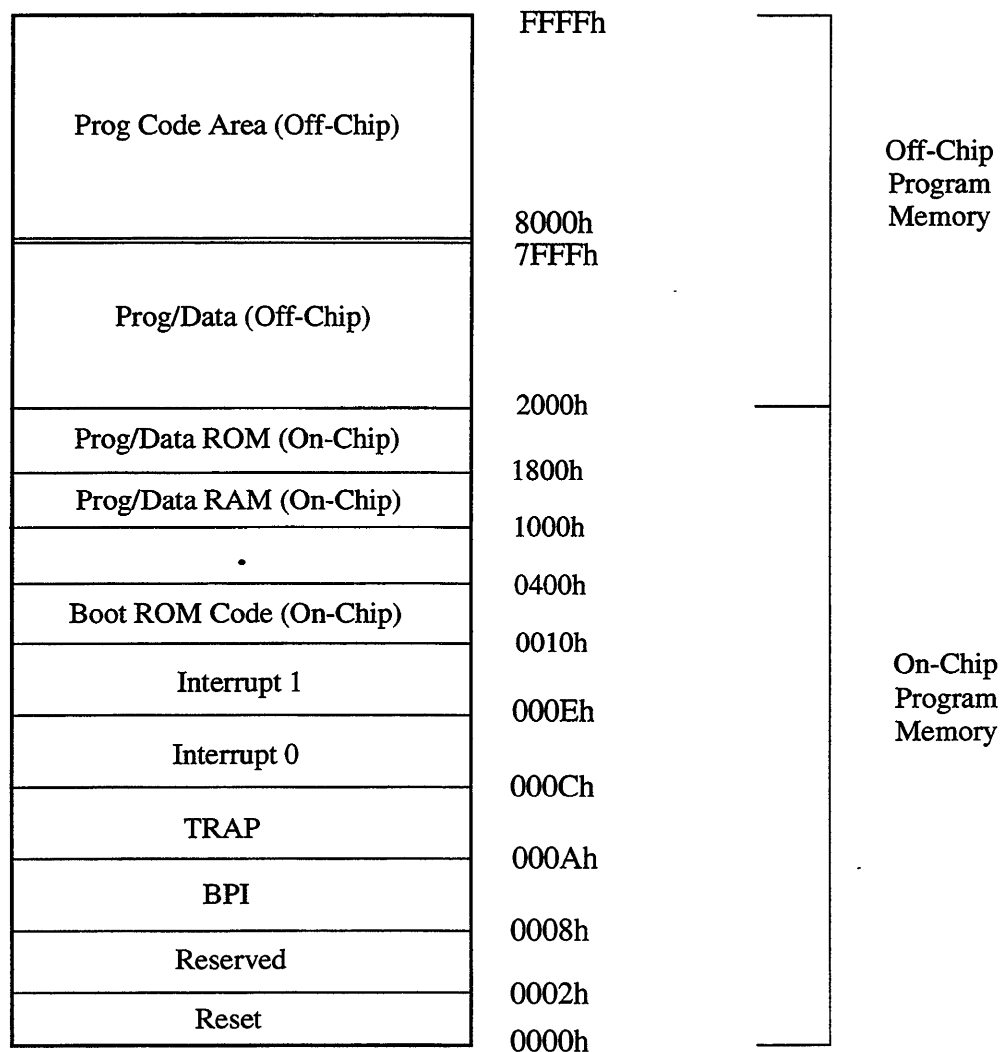


Figure 3-4 Program Memory Map

3.6.3 Memory Addressing Modes

There are two data addressing modes:

1. Direct Addressing Mode: Eight bits from instruction as LSB plus eight bits from status register ST1 (see Section 3.7.2 on Status Registers) as MSB compose the direct address to the Data memory. The pages are thus of 256 words each. For example, page 0 is from 0 to 255 in XRAM, page 1 is from 256 to 511 in XRAM, and page 255 is from -256 to -1 in YRAM.
2. Indirect Addressing Mode: The Rn registers of the DAAU are used for indirect addressing to the XRAM and YRAM.

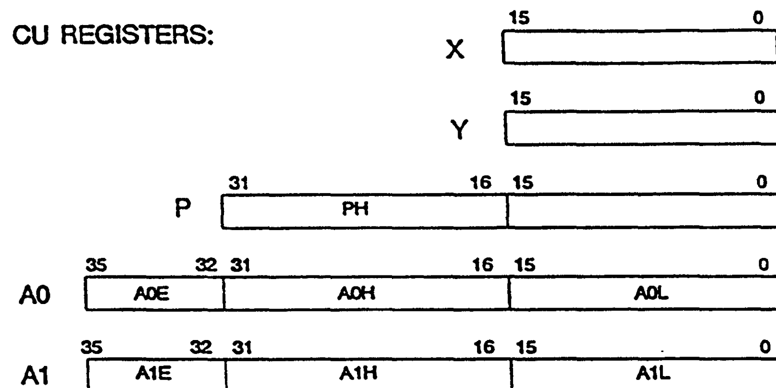
Addressing program memory is accomplished by:

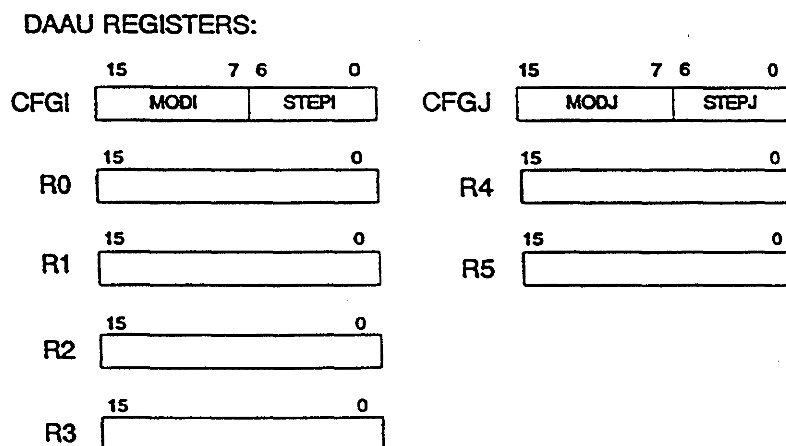
1. Indirect Addressing Mode: The Rn registers of the DAAU and the accumulator can be used for addressing the PROM in specific instructions.
2. Special Relative Addressing Mode: Special Branch-Relative (BRR) and Call-Relative (CALLR) instructions support jumping relative to the PC (from PC-63 to PC+64).

3.7 Programming Model and Registers

All of DSP core's visible registers are arranged as a global register set of 31 registers, which can be accessed by most move and ALU instructions. The registers are listed below, organized according to units' partition. Additional details on each register can be found in the description of each unit and in the following paragraphs.

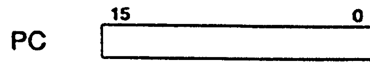
3.7.1 Programming Model





Programming Model (continued)

PCU REGISTERS:

PC


LC

GENERAL REGISTERS:

ST0

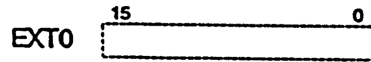
ST1

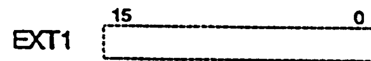
ST2

TOS*

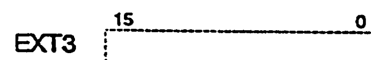
* Up to 16X16-bit stack levels

USER-DEFINED REGISTERS (Optional Off-Core):

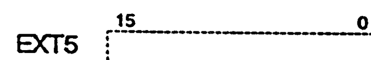
EXT0


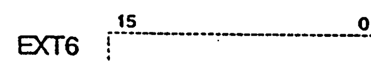
EXT1


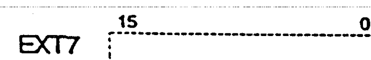
EXT2

EXT3


EXT4

EXT5


EXT6


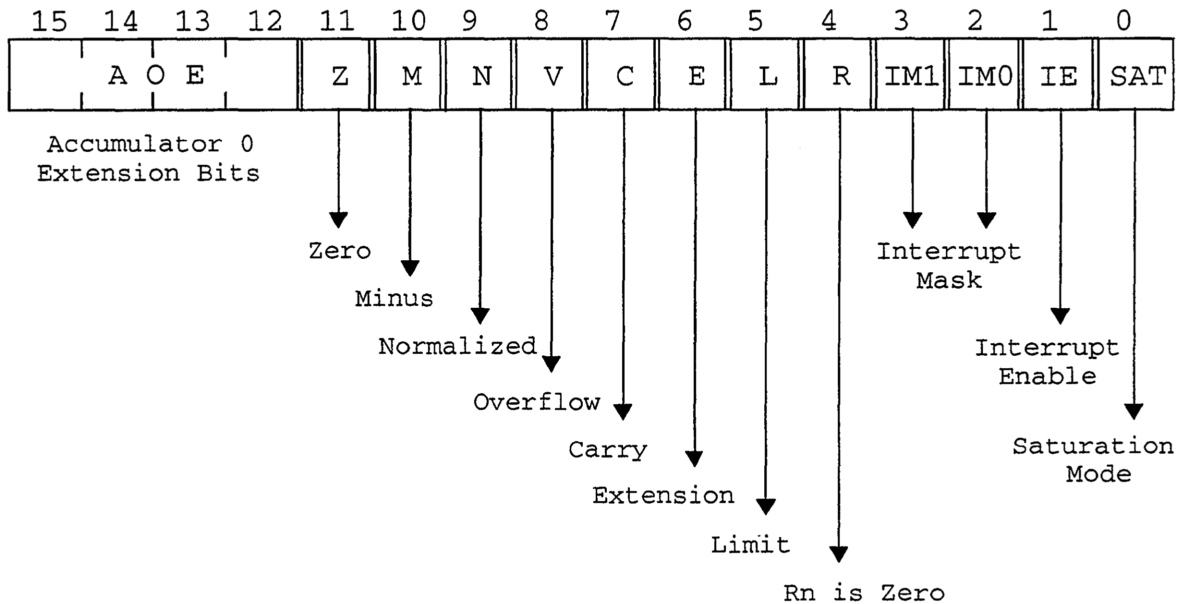
EXT7


3.7.2 Status Registers

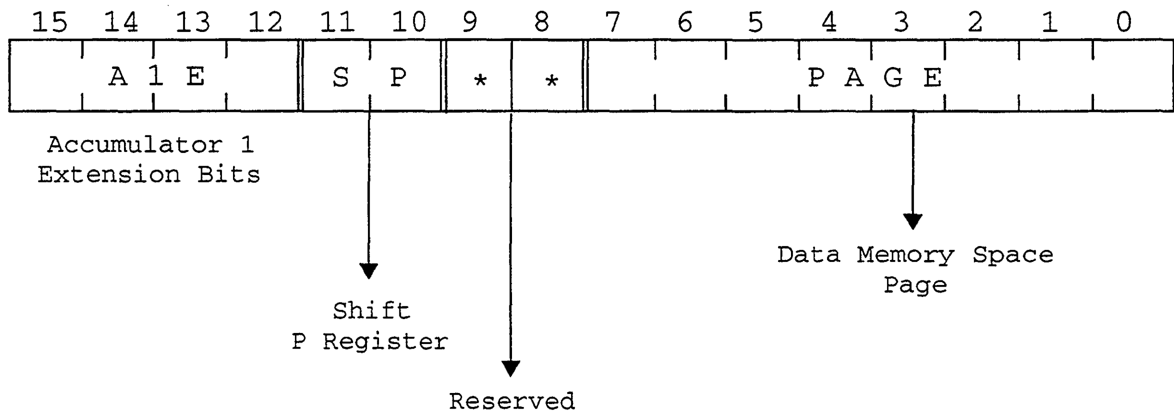
Three status registers are available for flags, status bits, control bits, user I/O bits, and paging bits for direct addressing. The contents of each register and their field definitions are described below.

3.7.2.1 Status Registers Format

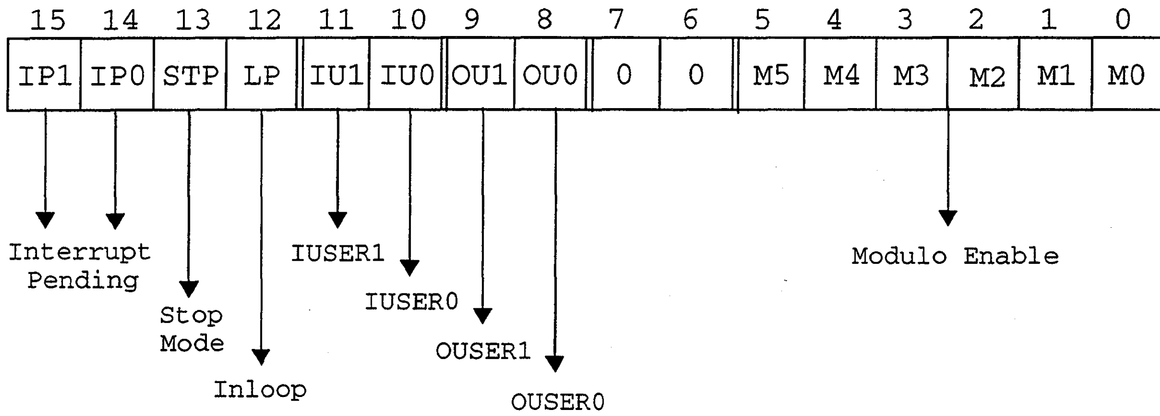
ST0



ST1



ST2



Notations:

*	Written as zero, read as don't care
0	Written as don't care, read as 0
~	Not
∪	Or
∩	And
⊕	Exclusive-Or

3.7.2.2 Status Register Field Definition

Most of the fields can be modified by writing to one of the status registers.

The flags (Z, M, N, V, C, E, L) indicate the result of the last ALU operation. At most of the cases the ALU output is latched at the destination accumulator, i.e. at most of the cases the flags indicates the destination accumulator status.

If an instruction has a different effect than those listed under the status register fields, refer to the specific instruction in Section 4.

ST0 Register

Zero (Z) - Bit 11
Set if the ALU output equals zero; cleared otherwise.

The zero flag is cleared during processor reset.
The zero flag can be modified by writing to ST0.

Minus (M) - Bit 10
Set if ALU output is a negative number; cleared otherwise. The minus flag is the same as the MSB of the ALU output (bit 35).

The minus flag is cleared during processor reset.
The minus flag can be modified by writing to ST0.

Normalized (N) - Bit 9
Set if the 32-bit of the accumulator which was the destination at the last instruction is normalized; cleared otherwise, i.e. set if $Z \cup ((\text{bit } 31 \oplus \text{bit } 30) \cap \sim E)$.

The normalized flag is cleared during processor reset.
The normalized flag can be modified by writing to ST0.

Overflow (V) - Bit 8
Set if an arithmetic overflow (36-bit overflow) occurs after an arithmetic operation; cleared otherwise. It indicates that the result of an operation cannot be represented in a 36 bits.

The overflow flag is cleared during processor reset.
The overflow flag can be modified by writing to ST0.

Carry (C) - Bit 7
Set if the result of an add generates a carry, or if the result of a subtract generates a borrow; cleared otherwise. It is also affected by the shift and rotate operations in the MODA instruction.

The carry flag is cleared during processor reset.
The carry flag can be modified by writing to ST0.

Extension (E) - Bit 6
Set if bits 35-31 of the ALU output, are not identical; cleared otherwise. It indicates that the result of an operation cannot be represented in a 32-bit accumulator.

The extension flag is cleared during processor reset.
The extension flag can be modified by writing to ST0.

Limit (L) - Bit 5
Set if the overflow flag was set (overflow latch) or a limitation occurred when performing a move instruction (MOV) from one of the accumulators (AxH and/or AxL) through the data bus. Otherwise it is not affected.

The limit flag is cleared during processor reset.
The limit flag can be modified by writing to ST0.

Rn register is zero (R) - Bit 4
This flag is affected by the MODR and NORM instructions. The R flag is set if the result of the Rn modification operation (Rn ; Rn+1; Rn-1; Rn+S) is zero; cleared otherwise.

Note: If the Modulo is enabled for the specific Rn register, Rn reaches the buffer boundary, and the Modulo mechanism sets a 0000H value in the register - the R flag is cleared.

The R flag status is latched until one of the above instructions is used.

The R flag is cleared during processor reset.
The R flag can be modified by writing to ST0.

Interrupt Mask (IM0, IM1) - Bits 2, 3
IM0 - Interrupt mask for INT0
IM1 - Interrupt mask for INT1
Clear - disable the specific interrupt
Set - enable the specific interrupt

The interrupt mask bits are cleared during processor reset.
The interrupt mask bits can be modified by writing to ST0.

Interrupt Enable (IE) - Bit 1
Clear - disable all maskable interrupts
Set - enable all maskable interrupts

The interrupt enable bit is cleared during processor reset.
The interrupt enable bit can be modified by instructions EINT (enable interrupts), DINT (disable interrupts), or by writing to ST0.

Saturation Mode (SAT) - Bit 0
Clear - enable the saturation when transferring the contents of the accumulator onto the data bus
Set - disable the saturation mode

The saturation enable bit is cleared during processor reset.
The saturation enable bit can be modified by writing to ST0.

ST1 Register

Shift P register (SP) - Bits 10, 11
The shift P register bits control the scaling shifter at the P register output.
SP bits 11, 10:
SP = 00 no shift
SP = 01 shift right
SP = 10 shift left
SP = 11 Reserved

The SP bits are cleared during processor reset.
The SP bits can be modified by writing to ST1.

RAM Page (PAGE) - Bits 0, 1, 2, 3, 4, 5, 6, 7
Used for direct address. Refer to Section 3.6.3 on Memory Addressing Modes.

The PAGE bits can be modified by the LPG instruction or by writing to ST1.

ST2 Register

INT PENDING (IP0, IP1) - Bits 14, 15
IP0 - Interrupt pending for INT0
IP1 - Interrupt pending for INT1
The interrupt pending bit is set when the corresponding interrupt is active. The bit reflects the interrupt level regardless of the mask bits.

The IPx bits are read only.

STOP (STP) - Bit 13
Set - stop processing mode (Low-power standby)
Clear - normal operation

The STOP bit is cleared during processor reset.
The STOP bit can be modified by writing to ST2.

INLOOP (LP) - Bit 12
Set if a block repeat is executed; cleared by the user or at the end of the block-repeat.

When transferring data into ST2, the LP bit will be influenced as follows:

"0" The LP bit is unaffected.

"1" The LP bit is cleared.

This bit can be used to break from a block repeat.

The inloop bit is cleared during processor reset.
The inloop bit can be cleared by writing to ST2.

In addition, refer to Section 3.5.1.1 on Repeat and Block-Repeat Unit.

IUSER0, IUSER1 (IU0, IU1) - Bits 10, 11
The IUSERx bits are read only and reflect the logic state of the corresponding user input pins.

OUSER0, OUSER1 (OU0, OU1) - Bits 8, 9
The OUSERx bits can be modified by writing to ST2 and define the logic state of the corresponding user output pins.

The OUSERx bits are cleared during processor reset.
The OUSERx bits can be modified by writing to ST2.

Modulo set (M0, M1, M2, M3, M4, M5) - Bits 0, 1, 2, 3, 4, 5
Cleared Mn bit - when using the corresponding Rn register, the Rn register will be modified as specified by the instruction regardless of the modulo option.
Set Mn bit - when using the corresponding Rn register, the Rn register will be modified as specified by the instruction using the suitable modulo.

The Mn bits are cleared during processor reset.
The Mn bits can be modified by writing to ST2.

3.7.3 Stack

A hardware stack is connected to the XDB and its top (TOS), and can be accessed as a Last-In-First-Out (LIFO) data register file. The stack is automatically loaded with the PC whenever a subroutine call or an interrupt occurs and popped back on return from subroutine or interrupt, respectively.

3.7.4 User-Defined Registers

The core supports 8 user-definable registers, which can be located on the DMC. This feature enables future expansion of the core. These registers appear in the data register fields of all relevant instructions. With these registers, external computation units can be loaded with data and read at the end of the computation directly into internal registers in single cycle.

3.8 Input and Output

Memory mapped I/O is used.

Two special input bits and two output bits are available as status bits in status register ST2.Conditional instructions can be executed according to the two input bits.

4.1 Introduction

This chapter provides an overview and detailed description of the DSP instruction set. The chapter also gives sufficient information to understand the nature of DSP programming and the capability of the instruction set itself.

4.2 Notation and Conventions

4.2.1 Notation

The following notations are used in this chapter:

Registers:

rN = Address registers: r0, r1, r2, r3, r4, r5
rI = Address registers: r0, r1, r2, r3
rJ = Address registers: r4, r5

aX = a0 or a1
aXl = Accumulator-low (LSP), x = 0, 1
aXh = Accumulator-high (MSP), x = 0, 1
aXe = Accumulator extension, x = 0, 1
ac = a0, a1, aoh, a1h, a0l, a1l

cfgX = Configuration registers of DAAU (MODI or MODJ, STEPI or STEPJ), x = I, J

tos = Top of stack
pc = Program counter
lc = Loop counter
extX = External registers, X = 0, 1,...7

reg = a0, a1, a0h, a1h, a0l, a1l, rN, x, y, p or ph, pc, lc, tos, st0, st1, st2, cfgI, cfgJ, extX

Address Operands:

On-chip data RAM/ROM
Off-chip data RAM/ROM:
Direct address = Paged direct address of 8 bits (see Chapter 3, Section 3.6.3 on addressing modes)

Program ROM:

Address	= Unsigned 16 bits	(0 to 65535)
$Offset address	= 2's complement 7 bits	(-64 to 63
	offset range:	-63 to 64)

Immediate Operands:

#Short immediate	= Unsigned 8 bits	(0 to 255)
#Signed Short immediate	= 2's complement 8 bits	(-128 to 127)
##Long immediate	= 2's complement 16 bits	(-32,768 to +32,767)

Assembler syntax:	___	decimal
	0b___ , 0B___	binary
	0x___ , 0X___	hexadecimal

When using hexadecimal representation, if the number starts with a letter (A-F), a leading zero must be inserted.

cond - condition field:

true	Always
eq	Equal to zero
neq	Not equal to zero
gt	Greater than zero
ge	Greater or equal to zero
lt	Less than zero
le	Less or equal to zero
nn	Normalize flag is cleared
v	Overflow flag is set
c	Carry flag is set
e	Extension flag is set
l	Limit flag is set
nr	R flag is cleared
niu0	IUSER0 input user pin 0 is cleared
iu0	IUSER0 input user pin 0 is set
iu1	IUSER1 input user pin 1 is set

Other:

(x)= The contents of x

{ } = Optional field at the instruction

[x] = Specific notes

-> = Is assigned to

>> = Shift right

<< = Shift left

_ = Not

_ = Or

_ = And

Flags Notation:

The effect of each instruction on the flags is described by the following notation:

* The flag is affected by the execution of the instruction.
\- The flag is not affected by the instruction.
1 or 0 The flag is unconditionally set or cleared by the instruction.

st0 bits	11	10	9	8	7	6	5	4
Flags	Z	M	N	V	C	E	L	R

For flag definitions, refer to Section 3.7.2.2 on Status Register Field Definitions.

4.2.2 Conventions

1. The arithmetic operations are performed in 2's complement.

2 When **rN** register is used by an instruction, the contents of rN register are post-modified as follows:

Options controlled by instruction:
rN, rN+1, rN-1, rN+step
Options controlled by configuration registers CFGx:
Step size: STEPI, STEPJ - 2's complement 7 bits (-64 to 63)
Modulo size: MODI, MODJ - unsigned 9 bits (1 to 512)
Options controlled by st2:
For each rN register it should be defined if MODULO is used or not.
For using MODI or MODJ the relative Mn bit must be set.

For more details on the modulo arithmetic unit refer to Section 3.4.1.2 on Modulo Modifier.

Assembler syntax: (rN) , (rN)+, (rN)- , (rN)+S

3 **ph** (the MSP of the p register) can be write only. The 32-bit p **reg** is updated after a multiply operation and can be read only by transferring it to the ALU, that is, it can be moved into aX or be an operand for arithmetic and logic operations. When transferring it into the ALU, it is sign-extended to 36 bits. This enables the user to store and restore the p register.

4 The **p register** is used as a **source** operand for different instructions as follows: as one of the reg registers; at moda instruction - PACR function; at multiply instructions where the p register is added or subtracted from one of the accumulators. When using the p register as a source operand, it always means using the **'shifted p register'**. Shifted p register means that the p register is sign-extended into 36 bit and then shifted as defined at the SP field, status register st1. In shift right the sign is extended, whereas in shift left a zero is appended into the LSB. The contents of the p register remain unchanged.

5 All move instructions using the accumulator (**aX**) as a destination are sign extended. All instructions which use the accumulator-low (**aXl**) as a destination, will clear the accumulator-high and the accumulator-extension. Therefore, they are sign extension suppressed.

All instructions using the accumulator-high (**aXh**) as a destination, will clear the accumulator-low and are sign extended. An exception is mov direct address,aXh,{eu}, when moving data into accumulator-high can be controlled with sign extension or with sign extension suppressed (the accumulator-extension aXe is unaffected).

6 In all arithmetic operations between 16-bit registers and aX (36 bits), the 16-bit register will be regarded as the 16 low-order bits of a 36-bit operand with a sign extension in the Most-Order-Bits.

7 It is recommended that the **flags** be used immediately after the ALU operation or moved into AC operations (see restrictions at 11.b). Otherwise, very careful programming is required (some flags may be changed in the meantime).

8 The **condition** field is an optional field; when the condition is missing then cond = true.

9 When transferring data into the hardware **stack,** the data is transferred to the **tos,** and the stack is pushed down one level. When transferring data out of the hardware stack, the data is copied to the destination, and the stack is popped one level.

10 **ALU instruction** is one of the following instructions: add, sub, or, and, xor, cmp, addl, subl, addh, subh, moda, norm, mac, msu, sqra, sqrs.

11 General Restrictions:

A Arithmetic and logical operations must not be performed with the same accumulator as the source (soperand) and the destination (doperand). Example: add a0,a0.

B **st0** cannot be used as a source operand after one of the following instructions:
1. An ALU instruction
2. An instruction where st0/a0/a0l/a0h is the destination operand

12 Following the mov operand, pc and movp soperand, pc, a nop instruction must be placed. After a move instruction to the **pc**, the nop instruction is executed. Only then is the pc updated according to the move instruction, e.g. performing a jump instruction. When an interrupt is accepted after a move to pc instruction (jump instruction), the tos contains the destination of the jump address.

4.3 Instruction Set Summary

The following is a summary of the Pine instruction set organized by instruction group. Page numbers for details of each instruction are shown at the right.

ARITHMETIC & LOGICAL INSTRUCTIONS

add	Add	(p. 29)
sub	Subtract	(p. 33)
or	OR	(p. 34)
and	AND	(p. 35)
xor	Exclusive - OR	(p. 37)
cmp	Compare	(p. 38)
addl	Add to Low Accumulator	(p. 38)
subl	Subtract from Low Accumulator	(p. 39)
addh	Add to High Accumulator	(p. 39)
subh	Subtract from High Accumulator	(p. 40)
moda	Modify Accumulator Conditionally	(p. 41)
norm	Normalize	(p. 44)
divs	Division Step	(p. 45)

MULTIPLY INSTRUCTIONS

mpy	Multiply	(p. 46)
mac	Multiply and Accumulate Previous Product	(p. 47)
msu	Multiply and Subtract Previous Product	(p. 48)
mpys	Multiply Signed Short Immediate	(p. 48)
sqr	Square	(p. 49)
sqra	Square and Accumulate Previous Product	(p. 49)
sqrs	Square and Subtract Previous Product	(p. 50)

MOVE INSTRUCTIONS

mov	Move Data	(p. 51)
movp	Move Program Memory	(p. 54)

LOOP INSTRUCTIONS

rep	Repeat Next Instruction	(p. 55)
bkrep	Block Repeat	(p. 56)

BRANCH / CALL INSTRUCTIONS

br	Conditional Branch	(p. 57)
brr	Relative Conditional Branch	(p. 58)
call	Conditional Call Subroutine	(p. 58)
callr	Relative Conditional Call Subroutine	(p. 59)
calla	Call Subroutine at Location Specified by the Accumulator	(p. 59)
ret	Return Conditionally	(p. 60)
reti	Return from Interrupt	(p. 60)

CONTROL & MISCELLANEOUS INSTRUCTIONS

nop	No Operation	(p. 61)
modr	Modify rN	(p. 61)
eint	Enable Interrupt	(p. 62)
dint	Disable Interrupt	(p. 62)
trap	Software Interrupt	(p. 63)
lpg	Load the Page Bits	(p. 63)

4.4 Instruction Set Details

This section provides detailed descriptions of each instruction. It includes instruction syntax, description of operation, operand details, effect on flags, number of execution cycles, and other relevant notes and exceptions.

The instructions are organized according to the groups listed in the summary in Section 4.3.

4.4.1 Arithmetic and Logical Instructions

add Add

```
add  operand , aX

    Operation:      aX    +   operand    ->   aX
                  Source      Source    Destination
                 operand 1   operand 2    operand
```

The instruction has **two source operands** which are added at the ALU, the ALU output is latched at the **destination operand.**

The flags are affected according to the ALU output, in this instruction it reflects the status of the destination accumulator.

```
    Affects flags:  Z   M   N   V   C   E   L   R
                    *   *   *   *   *   *   *   -

    Cycles:         1
                    2  when the operand is ##long immediate

    Words:          1
                    2  when the operand is ##long immediate
```

'aX' means one of the accumulators a0 or a1. This accumulator is both source operand and destination operand.

The **'Operand'** field is the other source operand, added at this instruction and can be one of the following options -

```
          operand:  reg
                    #short immediate
                    ##long  immediate
                    (rN)
                    direct address
```

reg - is one of the 31 Pine registers : a0, a1, a0h, a1h, a0l, a1l, rN, x, y, p, pc, lc, tos, st0, st1, st2, cfgI, cfgJ, extX. The contents of the source register is added to the accumulator. The operation result, the ALU output, is placed at the accumulator.

Example:

```
add  r1,a0
```

	Before execution	After execution
a0	1001H	1008H
r1	7H	7H

Short Immediate - the 8-bit (positive number) is one of the source operands. The 8-bit value is added, right-justified, to the accumulator. The operation result, the ALU output, is placed at the accumulator.

Example:

```
add  #255,a0
```

	Before execution	After execution
a0	1001	1256

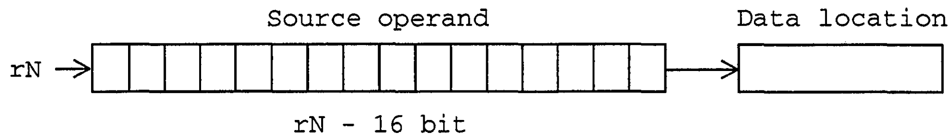
Long Immediate - the 16-bit value is one of the source operands. The 16-bit value is added, right-justified and sign-extended, to the accumulator. The operation result, the ALU output, is placed at the accumulator.

E.g.

```
add  ##0FFFFH,a1
```

	Before execution	After execution
a1	20H	1FH

(rN) - is indirect addressing.



One of the DAAU registers (r0, r1, r2, r3, r4, r5) points to one of the 64k data words. The data location contents, pointed by the register, is the source operand - added to the accumulator. The operation result, the ALU output, is placed at the accumulator.

The rN register is modified after the instruction is executed as follows:

(rN)	-	no update
(rN)+	-	rN is autoincrement
(rN)-	-	rN is autodecrement
(rN)+S	-	rN is autoincrement/autodecrement by the offset S

Each of these modifications can use the MODULO option.

For further details regarding the postmodification, see section 4.2.2 (2).

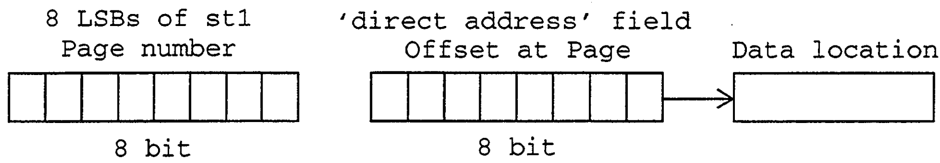
Example:

```
add  (r1)+S,a0
```

	Before execution	After execution
a0	1001H	1101H
r1	7H	9H
Data location 7H	100H	100H
CFJI meaning S=2	2H	2H
st2 meaning no modulo option for rl	XXX0H	XXX0H

Direct Address -

The data location, one of the 64k data words, is one of the source operands. The 16-bit data location is composed of the page number at st1 register and the 'direct address' field - the offset at the page. The data location contents is added to the accumulator. The operation result, the ALU output, is placed at the accumulator.



Example:

```
add  2,a1
```

	Before execution	After execution
a1	0FFFFFFFFFH	0H
st1 meaning page 1	0F301H	301H
Data location 102H	1H	1H

sub Subtract

```
sub  operand , aX

 Operation:     aX - operand -> aX

        operand:     reg
                     (rN)
                     direct address
                     #short immediate
                     ##long  immediate

Affects flags:  Z   M   N   V   C   E   L   R
                *   *   *   *   *   *   *   -

   Cycles:      1
                2  when the operand is ##long immediate
   Words:       1
                2  when the operand is ##long immediate
```

or OR

```
or  operand , aX

 Operation:    If operand is aX or p
                 aX[bits 35-0] OR operand -> aX[bits 35-0]

                 If operand is reg, (rN),
                               short immediate, long immediate
                   aX[bits 15-0] OR operand -> aX[bits 15-0]
                   aX[bits 35-16] -> aX[bits 35-16]

                 Clarification: If the operand is one of the accumula-
                 tors or  the p register, it is ORed with the destina-
                 tion accumulator.

                 If the operand is a 16-bit register or an immediate
                 value, the operand is zero-extended to form a 36 bits
                 operand, then ORed with the accumulator.  Therefore,
                 the upper bits of the accumulator are unaffected by
                 this instruction.

          operand:  reg
                    (rN)
                    direct address
                    #short immediate
                    ##long  immediate

Affects flags:  Z   M   N   V   C   E   L   R
                *   *   *   -   -   *   -   -

Cycles:         1
                2  when the operand is ##long immediate
Words:          1
                2  when the operand is ##long immediate
```

and AND

```
and  operand , aX

Operation:  If operand is aX or P
               aX[bits 35-0] AND operand -> aX[bits 35-0]

            If operand is short immediate
                  aX[bits 7-0 ] AND operand -> aX[bits 7-0]
                  aX[bits 15-8] -> aX[bits 15-8]  [1]
                  0  -> aX[bits 35-16]

            If operand is reg, (rN), long immediate
                     aX[bits 15-0] AND operand -> aX[bits 15-0]
                     0 -> aX[bits 35-16]

            Clarification: If the operand is one of the accumulators or
            the p register it is ANDed with the destination accumulator.

            If the operand is short immediate, the operand is zero-
            extended to form a 36-bit operand, then ANDed with the desti-
            nation accumulator.  Bits 15-8 are unaffected; other bits of
            the accumulator are cleared. [1]

            If the operand is a 16-bit register or a long  immediate
            value, the operand is zero-extended to form a 36-bit oper-
            and, then ANDed with the accumulator. Therefore, the upper
            bits of the  accumulator are cleared by this instruction.

     operand:  reg
               (rN)
               direct address
               #short immediate
               ##long  immediate

Affects flags:  Z   M   N   V   C   E   L   R
                *   *   *   -   -   *   -   -

            Z flag is set if all the bits at the ALU output are zeroed,
            otherwise cleared.  Note: when the operand is short immedi-
            ate, ALU output is  bit[35:8] = 0, bits[7:0] = aX[7:0] AND
            operand.

Cycles:     1
            2  when the operand is ##long immediate

Words:      1
            2  when the operand is ##long immediate
```

and AND (continued)

[1] The instruction and #short immediate, aX can be used for clearing some of the low-order bits at a 16-bit destination.

For example:

```
mov ram, aX
and #short immediate, aX
mov aX, ram
```

Using the and instruction, bits 15-8 are unaffected, therefore the high-order bits at the destination do not change.

In addition, this instruction can be used for BIT TEST, test one of the low-order bits of a destination.

For example:

```
mov ram, aX
and #short immediate, aX
br address,eq or br address,neq (check
  the zero flag)
```

xor Exclusive - OR

```
xor  operand , aX

Operation:  If operand is aX or p
               aX[bits 35-0] XOR operand -> aX[bits 35-0]

            If operand is reg, (rN),
                           short immediate, long immediate
                    aX[bits 15-0] XOR operand -> aX[bits 15-0]
                    aX[bits 35-16] -> aX[bits 35-16]

    Clarification:  If the operand is one of the accumulators or the p
                    register it is Exclusive-ORed with the destination
                    accumulator.

                    If the operand is a 16-bit register or an immedi-
                    ate value, the operand is zero-extended to form a
                    36 bits operand, then Exclusive-ORed with the
                    accumulator. Therefore, the upper bits of the
                    accumulator are unaffected by this instruction.

      operand:  reg
                (rN)
                direct address
                #short immediate
                ##long  immediate

Affects flags:  Z   M   N   V   C   E   L   R
                *   *   *   -   -   *   -   -

Cycles:      1
             2  when the operand is ##long immediate

Words:       1
             2  when the operand is ##long immediate
```

cmp Compare

```
cmp   operand , aX

Operation:  aX - operand

        operand: reg
                 (rN)
                 direct address
                 #short immediate
                 ##long  immediate

Affects flags:  Z   M   N   V   C   E   L   R
                *   *   *   *   *   *   *   -

Cycles:      1
             2  when the operand is ##long immediate

Words:       1
             2  when the operand is ##long immediate
```

addl Add to Low Accumulator

```
addl  operand , aX

Operation:  aX + operand -> aX
            The operand is sign-extension suppressed.

      operand: (rN)
               direct address
               reg             [1]

Affects flags:  Z   M   N   V   C   E   L   R
                *   *   *   *   *   *   *   -

Cycles:      1

Words:       1

[1] The reg cannot be: aX, p.
```

subl Subtract from Low Accumulator

```
subl   operand , aX

Operation:   aX - operand -> aX
             The operand is sign-extension suppressed.

      operand: (rN)
               direct address
               reg               [1]

Affects flags:  Z   M   N   V   C   E   L   R
                *   *   *   *   *   *   *   -

Cycles:      1

Words:       1

[1] The reg cannot be: aX, p.
```

addh Add to High Accumulator

```
addh  operand , aX

Operation:   aX + operand*2^16 -> aX
             The aXl is unaffected.

      operand: (rN)
               direct address
               reg               [1]

Affects flags:  Z   M   N   V   C   E   L   R
                *   *   *   *   *   *   *   -

Cycles:      1

Words:       1

[1] The reg cannot be: aX, p.
```

subh Subtract from High Accumulator

```
subh  operand , aX

Operation:  aX - operand*2^16 -> aX
            The aXl is unaffected.

      operand: (rN)
               direct address
               reg               [1]

Affects flags:  Z   M   N   V   C   E   L   R
                *   *   *   *   *   *   *   -

Cycles:     1

Words:      1

[1] The reg cannot be: aX, p.
```

moda Modify Accumulator Conditionally

moda func , aX , { cond }

Operation: If condition then aX is modified by 'func'

The accumulator and the flags are modified according to the function field only when the condition is met.

```
func:   SHR    aX = aX >> 1
        SHL    aX = aX << 1
        SHR4   aX = aX >> 4
        SHL4   aX = aX << 4
        ROR    Rotate aX right through carry
        ROL    Rotate aX left through carry
        NOT    aX = not(aX)
        NEG    aX = -aX
        CLR    aX = 0
        COPY   aX = aX
        RND    Round upper 20 bits of the aX
               aX = aX+8000H
        PACR   aX=shifted p + 8000H  [1]
        CLRR   aX = 8000H
```

Affects flags: See below.

Cycles: 1

Words: 1

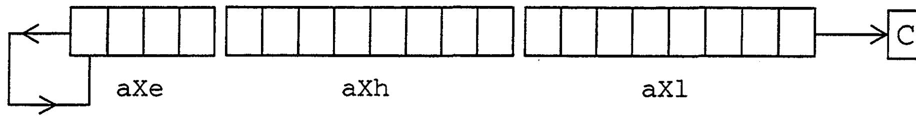
[1] Shifted p register means that the p register is sign-extended to 36 bits and then shifted as defined at the SP field, status register st1.

[2] At assembler syntax the moda can be omitted.

moda Modify Accumulator Conditionally (continued)

SHR, SHR4

Shift right step

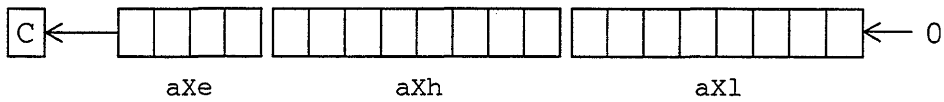


Affects flags:	Z	M	N	V	C	E	L	R
	*	*	*	0	*	*	-	-

C - Set according to the LSB (SHR bit 0, SHR4 bit 3) shifted out of the operand.

SHL, SHL4

Shift left step

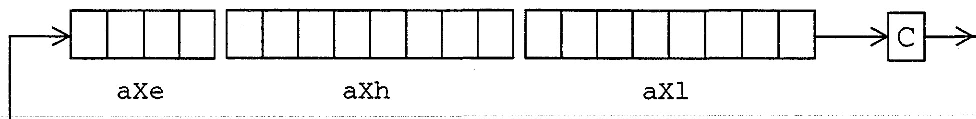


Affects flags:	Z	M	N	V	C	E	L	R
	*	*	*	*	*	*	-	-

C - Set according to the MSB (SHL bit 35, SHL4 bit 32) shifted out of the operand.

V - Cleared if the operand being shifted could be represented in 35 bits for SHL / in 31 bits for SHL4, set otherwise.

ROR

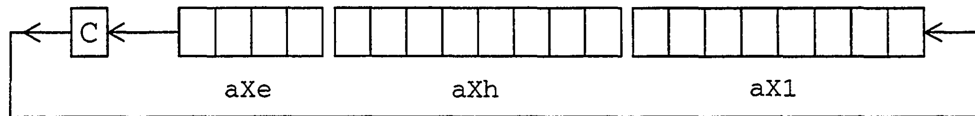


Affects flags:	Z	M	N	V	C	E	L	R
	*	*	*	-	*	*	-	-

C - Set according to the LSB (bit 0) shifted out of the operand.

moda Modify Accumulator Conditionally (continued)

ROL




```
Affects flags:  Z   M   N   V   C   E   L   R
                *   *   *   _   *   *   _   _

            C - Set according to the MSB (bit 35)
                shifted out of the operand.
```

NOT, COPY, CLR, CLRR

```
Affects flags:  Z   M   N   V   C   E   L   R
                *   *   *   _   _   *   _   _
```

NEG, RND, PACR

```
Affects flags:  Z   M   N   V   C   E   L   R
                *   *   *   *   *   *   *   _
```

norm Normalize

```
norm  aX , rN

Operation:  If N = 0 (aX is not normalized)
            then  aX = aX * 2
                  N is modified
               else nop
                    nop

            This instruction is used to normalize the signed number at
            the accumulator. Affects the rN register.

Affects flags:  Z   M   N   V   C   E   L   R
                *   *   *   *   *   *   *   *

            R flag is updated in norm instruction ONLY when rN pointer is
            modified.

            C is set or cleared as at SHL (moda).

Cycles:     2

Words:      1

[1]  The norm instruction uses the N flag to decide if shift or nop.
     Therefore when using norm at the first iteration, the flag must be
     updated according to aX.

[2]  To normalize a number using the norm instruction, the norm instruc-
     tion can be used together with a rep instruction.

     Example:  rep    #n
               norm   a0,(r0)+

Another method is to use the N flag for conditional branch.

     Example:  NRM    norm   a0,(r0)+
               brr    NRM,NN
```

divs Division Step

```
divs  direct address ,aX

Operation:   aX - (direct address*2^15) -> ALU output
             If ALU output < 0
                  then  aX = aX * 2
                  else  aX = ALU output * 2 + 1

Affects flags:  Z   M   N   V   C   E   L   R
                *   *   *           *

Cycles:      2

Words:       1
```

[1] The 16-bit dividend is placed at accumulator-low; the accumulator-high and the accumulator-extension are cleared.

The divisor is placed at the direct address.

For a 16-bit division, divs should be executed 16 times. After 16 times the quotient is in the accumulator-low and the remainder is in the accumulator-high.

The dividend and the divisor should both be positive.

4.4.2 Multiply Instructions

mpy Multiply

```
mpy  operand1 , operand2

Operation:  operand1 -> y    [1]
            operand2 -> x
            x * y -> P

     operand1 , operand2:  y    , direct address
                           y    , (rN)
                           y    , reg          [2]
                           (rJ) , (rI)         [3]
                           (rN) , ##long  immediate

Affects flags:  No

Cycles:     1
            2 when the operand is ##long immediate

Words:      1
            2 when the operand is ##long immediate
```

[1] y -> y means that y retains its value.

[2] The reg cannot be aX, p.

[3] The multiplication at mpy (rJ),(rI) is between XRAM and YRAM only. Where rJ points to YRAM, rI points to XRAM.

mac Multiply and Accumulate Previous Product

```
mac  operand1 , operand2 , aX

Operation:  aX + shifted p -> aX   [1]
            operand1 -> y          [2]
            operand2 -> x
            x * y -> p

     operand1 , operand2:  y    , direct address
                           y    , (rN)
                           y    , reg       [3]
                           (rJ) , (rI)      [4]
                           (rN) , ##long  immediate

Affects flags:  Z  M  N  V  C  E  L  R
                *  *  *  *  *  *  *  -

Cycles:     1
            2 when the operand is ##long immediate

Words:      1
            2 when the operand is ##long immediate
```

[1] Shifted p register means that the previous product is sign-extended into 36 bits, then shifted by defined at the SP field, status register st1.

[2] y -> y means that y retains its value.

[3] The reg cannot be aX, p.

[4] The multiplication at mac (rJ),(rI) is between XRAM and YRAM only. Where rJ points to YRAM , rI points to XRAM.

msu Multiply and Subtract Previous Product

```
msu  operand1 , operand2 , aX

Operation:  aX - shifted p -> aX   [1]
            operand1 -> y          [2]
            operand2 -> x
            x * y -> p

     operand1 , operand2:  y     , direct address
                           y     , (rN)
                           y     , reg[3]
                           (rJ)  , (rI)[4]
                           (rN)  , ##long  immediate

Affects flags:  Z   M   N   V   C   E   L   R
                *   *   *   *   *   *   *   -

Cycles:     1
            2 when the operand is ##long immediate

Words:      1
            2 when the operand is ##long immediate
```

[1] Shifted p register means that the previous product is sign-extended into 36 bits, then shifted as defined by the SP field, status register st1.

[2] y -> y means that y retains its value.

[3] The reg cannot be aX, p.

[4] The multiplication at msu (rJ),(rI) is between XRAM and YRAM only. Where rJ points to YRAM , rI points to XRAM.

mpys Multiply Signed Short Immediate

```
mpys  y, #signed short immediate

Operation:  #signed short immediate -> x
            x * y -> p

Affects flags:  No

Cycles:     1

Words:      1
```

sqr Square

```
sqr  operand

Operation:  operand -> y
            operand -> x
            y * x -> p

     operand:  (rN)
               reg                [1]
               direct address

Affects flags:  No

Cycles:     1

Words:      1

[1]  The reg cannot be aX, p.
```

sqra Square and Accumulate Previous Product

```
sqra  operand ,aX

Operation:  aX + shifted p -> aX  [1]
            operand -> y
            operand -> x
            y * x -> p

     operand:  (rN)
               reg             [2]
               direct address

Affects flags:  Z   M   N   V   C   E   L   R
                *   *   *   *   *   *   *   -

Cycles:     1

Words:      1

[1]  Shifted p register means that the previous product is sign-extended
     into 36 bits, then shifted as defined by the SP field, status regis-
     ter st1.

[2]  The reg cannot be aX, p.
```

sqrs Square and Subtract Previous Product

```
sqrs  operand , aX

Operation:  aX - shifted p -> aX     [1]
            operand -> y
            operand -> x
            y * x -> p

     operand :  (rN)
                reg                  [2]
                direct address

Affects flags:  Z   M   N   V   C   E   L   R
                *   *   *   *   *   *   *   -

Cycles:     1

Words:      1
```

[1] Shifted p register means that the previous product is sign-extended into 36 bits, then shifted as defined by the SP field, status register st1.

[2] The reg cannot be aX, p.

4.4.3 Move Instructions

mov Move Data

```
mov  soperand , doperand

Operation:    soperand -> doperand

     soperand , doperand    reg    ,  reg        [1],[2],[3],[4]
                            reg    ,  (rN)       [1],[5],[6]
                            (rN)   ,  reg        [4],[5],[6]

                            rN     ,  direct address
                            aXl    ,  direct address
                            aXh    ,  direct address
                            Y      ,  direct address
                            X      ,  direct address

                            direct address , rN
                            direct address , y
                            direct address , x
                            direct address , aX
                            direct address , aXl
                            direct address , aXh , {eu}    [7]
                            ##long immediate  , reg        [4]
                            #short immediate , aXl
                            #signed short immediate , aXh
                            #signed short immediate , rN   [8]
                            #signed short immediate , y    [8]
                            #signed short immediate , x    [8]

Affects flags:   No effect when doperand is not ac, st0 or when soperand
                 is not aXl, aXh

                 When soperand is aXl or aXh:

                 Z   M   N   V   C   E   L   R
                 -   -   -   -   -   -   *   -

                 When doperand is ac:

                 Z   M   N   V   C   E   L   R
                 *   *   *   -   -   *   -   -         [7]

                 If doperand is st0, the instruction affects all the
                 flags.

Cycles:      1
             2 when the operand is ##long immediate

Words:       1
             2 when the operand is ##long immediate
```

mov Move Data (continued)

[1] The 32-bit p register can be transferred only to aX (mov p,aX)

ph is a write-only register, therefore soperand cannot be ph.

[2] The 36-bit aX can be a soperand only with the following instructions: mov a0,a1 ; mov a1,a0.

[3] With mov reg, the soperand cannot be the same as the doperand.

[4] When the operand reg is the pc register, a nop instruction must be placed after the mov soperand,pc instruction.

[5] No mov's are permitted between off-core memory and external registers and vice versa. This means that mov extX,(rN) , mov (rN),extX rN can only point with internal RAM.

[6] It is not permitted to move data from RAM address pointed by one of the rN registers to the same rN register (and vice versa) with post modified.

[7] The eu field is an optional field.

eu = accumulator **e**xtension is **u**naffected (sign extension suppressed)

Instruction Fields		**Accumulator Content After The Instruction**		
ac	eu	Extension bits aXe	16 MSB aXh	16 LSB aXl
aX	-	sign-extended	sign-extended	DATA
aXl	-	clear	clear	DATA
aXh	-	sign-extended	DATA	clear
aXh	eu	unaffected	DATA	clear

The flags after executing mov direct address, aXh, eu are the same as after executing mov direct address,aXh.

[8] Loading the doperand by short immediate number with sign- extension.

mov Move Data (continued)

[9] Conventions:

```
The instruction at PROM address 0100H  mov  pc,ram
After execution     (ram)=0101H

mov  (r0),r0
```

	Before execution	After execution
r0	20H	1000
RAM address 20H	1000	1000

movp Move Program Memory

movp soperand , doperand

Operation: soperand points to PROM -> doperand

Move a word from Program ROM pointed by soperand to RAM or to reg pointed by doperand. When using aX as a soperand, the address is defined by accumulator-low.

soperand , doperand: (aX) , reg [1]
(rN) , (rI)

Affects flags: No effect when doperand is not ac, st0.

When doperand is ac:

Z	M	N	V	C	E	L	R
*	*	*	-	-	*	-	-

If the doperand is st0, the instruction affects all the flags.

Cycles: 3

Words: 1

[1] When the operand reg is the pc register, a nop instruction must be placed after the movp (aX),pc instruction.

4.4.4 Loop Instructions

rep Repeat Next Instruction

rep operand

Operation: Begins a single word instruction loop that is to be repeated operand+1 times.

Repetition times is between 1 to 256. The rep instruction and the instruction being repeated are not interruptable.

operand: #short immediate
reg [1]

Affects flags: No

Cycles: 1

Words: 1

[1] The reg cannot be aX, p.

[2] When using reg as an operand, the number of times the instruction is to be repeated is defined by the low-order 8 bits of the reg.

[3] Any instruction that breaks the program address continuity cannot be repeated. (brr, callr, movp, trap, ret, reti, mov operand,pc ,rep, calla)

[4] rep can be performed inside block-repeat (bkrep).

bkrep Block Repeat

bkrep operand , add

Operation: operand -> lc [1]
1 -> LP status bit

Begins a block repeat that is to be repeated operand+1 times.

The number of repetitions ranges from 1 to 256.

The first block address is the address after the bkrep instruction, and the last block, address is the address specified by the 'add' field. [2]

The operand is inserted into the loop counter lc register. The inloop status bit LP is set - enable block repeat.

The repeated block is interruptable.

operand: #short immediate
reg [3],[4]

Affects flags: No

Cycles: 2

Words: 2

[1] When using #short immediate as an operand, it is copied to the low-order 8 bits of the lc. The high-order 8 bits are undefined.

[2] In case the last instruction at the block repeat is:

a. One word instruction - 'add' is the address of this instruction.

b. Two words instruction - 'add' is the address of the second word of the instruction.

[3] When using reg as an operand the 16 bit register is transferred into the lc. The number of times the block is to be repeated is defined only by the low-order 8 bits.

[4] The reg cannot be aX, p, pc.

[5] When the block repeat is completed, the low-order 8 bits of the lc register contains 0FFH; the high-order 8 bits are unaffected.

[6] The block repeat length can be one instruction.

[7] If a mov st2, doperand is performed one instruction prior last instruction of the loop, the value of the LP bit undefined.

bkrep Block Repeat (continued)

[8] Restrictions:

1. The last two instructions of the bkrep loop cannot be br, brr, call, callr, calla, trap, mov soperand, pc, movp soperand, pc, ret, reti, rep, bkrep.

2. During a block-repeat loop there can be no jumps to the last address of the loop. Forbidden jumps are:

   brr, br, call, callr, calla, ret, reti, mov soperand, pc, movp soperand, pc.

3. Restrictions relating the lc register usage at the block-repeat loop are:

   3.1. The lc register must not be written during the block-repeat loop.

   3.2. The lc register must not be used one instruction prior to the last instruction of the block-repeat loop.

   3.3. If the block-repeat loop is one instruction long, the lc register must not be used in this instruction.

4. Notice that illegal instruction sequences are also restricted as the last and first instructions of a block-repeat loop.

4.4.5 Branch/Call Instructions

br Conditional Branch

br address , {cond}

Operation: If condition
then address -> pc

If the condition is met, branch to the program memory location specified by 'address'.

Affects flags: No

Cycles: 2

Words: 2

[1] If the condition is met, 'address' is the address/label of the new program memory location. The 'address' is the second word of the instruction.

brr Relative Conditional Branch

brr $offset address , {cond}

Operation: If condition then
'the brr inst.' + $offset address + 1 -> pc

If the condition is met, a branch is executed to the following program memory location: 'the brr instruction' + 'offset address' + 1

The offset range is -63 to 64. (Offset range is 'offset address'+1)

Affects flags: No

Cycles: 2

Words: 1

[1] Assembler syntax:
brr $offset address, {cond}
or
brr label , {cond}
Where 'label' is the new program memory location. The instruction word includes the 'offset address' calculated by the assembler as follows:
(label address) - (brr address) - 1.

call Conditional Call Subroutine

call address , {cond}

Operation: If condition
then pc -> tos
address -> pc

If the condition is met, the program counter is pushed into the stack and a branch is performed to the program memory location specified by 'address'.

Affects flags: No

Cycles: 2

Words: 2

[1] If the condition is met, 'address' is the address/label of the new program memory location. The 'address' is the second word of the instruction.

callr Relative Conditional Call Subroutine

callr $offset address , {cond}

Operation:

```
If condition then
  pc  -> tos
  'the brr inst.' + $offset address + 1 -> pc
```

If the condition is met, the program counter is pushed into the stack and a branch is executed to the following program memory location:

'the callr instruction' + 'offset address' + 1

The offset range is -63 to 64. (Offset range is 'offset address'+1).

Affects flags: No

Cycles: 2

Words: 1

[1] Assembler syntax:

```
callr $offset address , {cond}
or
callr label , {cond}
```

Where 'label' is the new program memory location. The instruction word includes the 'offset address' calculated by the assembler as follows:

(label address) - (callr address) - 1.

calla Call Subroutine at Location Specified by the Accumulator

calla aXl

Operation:

```
pc -> tos
(aX) -> pc
```

Call subroutine indirect (address from aX).

The program counter is pushed into the stack and a branch is executed to the address pointed by accumulator-low.

This instruction can be used to perform computed subroutine calls.

Affects flags: No

Cycles: 2

Words: 1

ret Return Conditionally

ret {cond}

```
Operation:      If condition
                  then  tos -> pc
```

If the condition is met, the program counter is pulled from the stack. The previous program counter is lost. This instruction is used to return from subroutines or interrupts.

Affects flags: No

Cycles: 2

Words: 1

[1] This instruction can also be used as return from interrupt (INT0 or INT1), to enable more interrupts, the IE bit at st0 must be set

.reti Return from Interrupt

reti

```
Operation:      tos -> pc
                1 -> IE    [1]
```

The program counter is pulled from the system stack. The previous program counter is lost. The IE bit is set - enable interrupts. [1] This instruction is used for return from interrupt.

Affects flags: No

Cycles: 2

Words: 1

[1] This instruction is used for returning from interrupts. The trap and BPI interrupt service routines must be ended with reti instruction. In these cases, the IE status will be the same as it was before entering the routine.

4.4.6 Control and Miscellaneous Instructions

nop No Operation

nop

Operation: No operation

Affects flags: No

Cycles: 1

Words: 1

modr Modify rN

modr (rN)

Operation: rN is modified.

Affects flags:

Z	M	N	V	C	E	L	R
-	-	-	-	-	-	-	*

R flag is set if the 16-bit rN register is zero; otherwise cleared.

Cycles: 1

Words: 1

[1] This instruction can be used also for loop control.

Example:
```
modr (r0)-
brr add , NR
```

eint Enable Interrupt

eint

```
Operation:      1 -> IE
                IE bit is set - Enable interrupts.

Affects flags: No

Cycles:     1

Words:      1
```

dint Disable Interrupt

dint

```
Operation:      0 -> IE
                IE bit is cleared - Disable interrupts.

Affects flags: No

Cycles:     1

Words:      1
```

trap Software Interrupt

trap

Operation: pc -> tos
000AH -> pc
Disable interrupts (INT0 , INT1).

Software interrupt.

The program counter which points to the next instruction is pushed into the stack. A branch to address location 000AH is executed.

The interrupts (INT0, INT1) are disabled regardless of the interrupt mask bits: IE, IM0, IM1 at st0.

Affects flags: No

Cycles: 2

Words: 1

[1] trap instruction cannot be used at: trap service routine, BPI service routine.

[2] For returning from trap service routine use reti instruction.

lpg Load the Page Bits

lpg #short immediate

Operation: #short immediate -> page bits

The page bits, the low-order 8 bits of st1, are loaded with an 8-bit constant (0 to 255).

Affects flags: No

Cycles: 1

Words: 1

4.5 Instruction Execution

4.5.1 Pipeline Method

The program controller implements a three-level pipeline architecture. In the operation of the pipeline, concurrent fetch, operand fetch and execution occur. This allows instruction execution to overlap. Thus, the effective execution time for most instructions is one cycle. Each pipeline stage is completed before its result is needed by the next instruction. The pipeline is an "interlocking" pipeline, transparent to the user, which simplifies programming.

The following chart shows the pipeline operation:

```
            |  cycle1  | cycle2  | cycle3  | cycle4  | cycle5  |
            |          |         |         |         |         |
fetch       |<---n---->|<--n+1-->|<--n+2-->|   ...   |         |
            |          |         |         |         |         |
op. fetch   |   ...    |<---n--->|<--n+1-->|<--n+2-->|   ...   |
            |          |         |         |         |         |
execution   |          |   ...   |<---n--->|<--n+1-->|<--n+2-->|
            |          |         |         |         |         |
```

Three instructions are executed for each cycle. For example in cycle 3, the fetch of instruction n+2, the operand fetch of instruction n+1 and the execution of instruction n are active.

SECTION 5
Core Interface

5.1 Introduction

This section describes the basic core clock interface. It includes descriptions of the core clock, the reset mode, the handling of interrupts and their priorities, the core signals and cycles.

5.2 Clock

The DSP core is driven by an off-core clock generator.

DSP status bit STP (bit 13 of ST2) is routed to the core edge and can be used by the system clock generator to stop the clock and enter the STOP mode.

5.3 Reset

Reset is a non-maskable interrupt that can be used at any time to put the DSP core into a known state. Reset is typically applied after power up when the machine is in a non-deterministic state. It is also used to exit STOP mode.

When a RESET is applied to the core, the processor enters the reset processing state, the processor terminates execution and forces the program counter to zero. Therefore, when the RESET signal is deactivated, execution starts from location 0000H. Reset affects various registers and status bits. However when RESET is applied during STOP mode, the contents of the RAM and other registers are unaffected.

The following register bits are cleared during reset:

> ST0 bits 0÷11, ST1 bits 10÷11, ST2 bits 0÷9, ST2 bits 13÷12
> PC register (0:15)

For more details about reset effects on status registers, see Section 3.7.2.2.

The RESET signal (reset request to the core) must be active for at least 6 cycles. The fetch from address 0000H is executed 1 cycle after RESET is deactivated.

5.4 Interrupts

The DSP core has three interrupts: two maskable (INT0 and INT1) interrupts and one break-point interrupt (BPI). It also has one software interrupt (TRAP). The hardware interrupts are high level-sensitive.

Table 5-1 DSP Core Interrupts

Memory * location	Interrupt name & function	Priority
0000H	**RESET**	1 highest
0008H	**BPI** External breakpoint interrupt	2
000AH	**TRAP** Software interrupt	3
000CH	**INT0** External user interrupt0	4
000EH	**INT1** External user interrupt1	5 lowest

* Start address for the interrupt/reset routine.

Jumping to the interrupt service routine takes two cycles. The only exception to this is after a move instruction to the PC, which takes only one cycle.

5.4.1 BPI, INT0, INT1

Interrupts are accepted and serviced at the end of the current instruction execution. Interrupt servicing will be delayed when one of the following cases occurs:

1. Until the completion of all the cycles of a multicycle instruction including expansion of read/write cycles due to wait states.
2. Until the end of the REP instruction and the instruction being repeated.
3. When the processor is in STOP mode.
4. In the cycle after fetch of the following commands: DINT; mov, soperand, ST0; movp (aX), ST0.

If more than one interrupt is pending and unmasked, the interrupt with the highest priority is accepted and serviced.

5.4.1.1 BPI - Breakpoint Interrupt

This interrupt is an active high, non-maskable interrupt. When executing a BPI service routine, another BPI will not be accepted.

When the BPI is accepted, DSP core performs the following:

PC -> TOS
0008H -> PC (interrupt starting address)
IACK pulse is generated

The TRAP instruction cannot be used inside a BPI service routine.
The BPI service routine must end with a RETI instruction.

5.4.1.2 5.4.1.2 TRAP - Software Interrupt

When executing a TRAP service routine, another TRAP cannot be used. Due to interrupt priorities (see above table) a BPI interrupt can be accepted inside a TRAP routine; INT0 and INT1 are not accepted.

When the TRAP is accepted, DSP core performs the following:

PC -> TOS
000AH -> PC (interrupt starting address)

A TRAP service routine must end with a RETI instruction.

5.4.1.3 INT0,INT1 - Maskable Interrupts

Interrupts INT0,INT1 are active high, maskable interrupts.

When an interrupt is accepted while the IE status bit is set and the individual interrupt is not masked (the corresponding IMx is set), DSP core performs the following:

IE status bit cleared
PC -> TOS
000CH or 000EH -> PC (interrupt starting address)
IACK pulse is generated
IMx is unaffected

Return from an interrupt service routine by RETI or by RET. When using the RET instruction, the IE flag must be set in order to enable interrupts again.

Notes:

1. A typical interrupt is activated by applying a high level to the INT0/INT1 input, thus setting the corresponding IPx bit. When the interrupt is acknowledged, the IE at the status register is disabled. If the interrupt signal at the input pin continues to be active and the IE is still disabled, no interrupts will be generated. If the interrupt signal level is removed before the IE is enabled, no further interrupts are generated. If the interrupt request continues to be active and the IE bit is subsequently set, then another interrupt will be generated after setting the IE.

2. The PRIORITY between INT0/INT1 is significant only if more than one interrupt is received at the same time or when the IE is disabled for some time and both INT0 and INT1 were received. In these cases, the interrupt will be acknowledged according to the interrupts priorities.

   In case the processor is handling the INT1 service routine and INT0 was received, the IP0 bit will be set and the processor will enter the INT0 service routine according to the status bits: IE, IM0. Similar handling will occur if the processor handles INT0 service routine and INT1 was received.

Adaptec, Inc.
691 South Milpitas Boulevard
Milpitas, CA 95035
Tel: (408) 945-8600
Fax: (408) 262-2533

P/N: 700175-011 Rev 2
Printed in U.S.A. DM/gc 12/1/94
Information is subject to change without notification.