

7 DATA MOVE INSTRUCTIONS

The instruction set provides move instructions for transferring data between the DSP's data registers, memory, I/O registers, and system control registers. Transfer operations include reading, writing, loading, and storing data from one location to another. Data move operations include:

- “Register to Register Move” on page 7-22
- “Direct Memory Read/Write—Immediate Address” on page 7-24
- “Direct Register Load” on page 7-27
- “Indirect 16-bit Memory Read/Write—postmodify” on page 7-30
- “Indirect 16-bit Memory Read/Write—premodify” on page 7-34
- “Indirect 24-bit Memory Read/Write—postmodify” on page 7-37
- “Indirect 24-bit Memory Read/Write—premodify” on page 7-41
- “Indirect DAG Register Write (premodify or postmodify), with DAG Register Move” on page 7-45
- “Indirect Memory Read/Write—immediate postmodify” on page 7-49
- “Indirect Memory Read/Write—immediate premodify” on page 7-52
- “Indirect 16-bit Memory Write—immediate data” on page 7-55
- “Indirect 24-bit Memory Write—immediate data” on page 7-57

Data Move Instructions

- “External IO Port Read/Write” on page 7-59
- “System Control Register Read/Write” on page 7-61
- “Modify Address Register—indirect” on page 7-63
- “Modify Address Register—direct” on page 7-65

This chapter describes each of the move instructions and the following related topics:

- “Core Registers” on page 7-2
- “PX Register” on page 7-4
- “DAG Registers” on page 7-6
- “Register Load Latencies” on page 7-9
- “Direct Addressing” on page 7-12
- “Indirect Addressing” on page 7-12
- “Circular Data Buffer Addressing” on page 7-15
- “Bit-Reversed Addressing” on page 7-17

Core Registers

Table 7-1 lists the registers that reside in the DSP’s core. Most are 16-bit registers, but some `Reg3` registers are shorter—`ASTAT[9]`, `MSTAT[7]`,

SSTAT[8], LPCSTACKP[9], CCODE[4], PX[8], DMPG1[8], DMPG2[8], IOPG[8], IJPG[8], and STACKP[8].

Table 7-1. Core registers

Register Groups			
Reg0 (Dreg)	Reg1 (G1reg)	Reg2 (G2reg)	Reg3 (G3reg)
AX0	I0	I4	ASTAT
AX1	I1	I5	MSTAT
MX0	I2	I6	SSTAT
MX1	I3	I7	LPSTACKP
AY0	M0	M4	CCODE
AY1	M1	M5	SE
MY0	M2	M6	SB
MY1	M3	M7	PX
MR2	L0	L4	DMPG1
SR2	L1	L5	DMPG2
AR	L2	L6	IOPG
SI	L3	L7	IJPG
MR1	IMASK	Reserved	Reserved
SR1	IRPTL	Reserved	Reserved
MR0	ICNTL	CNTR	Reserved
SR0	STACKA	LPSTACKA	STACKP

Data Move Instructions

As shown, registers are grouped along functional lines:

- Reg0 (Dreg) Consists of data registers.
- Reg1 (G1reg) Consists of DAG1 addressing registers, interrupt control registers, and the lower part of the PC stack register.
- Reg2 (G2reg) Consists of DAG2 addressing registers, the loop counter register, and the lower part of the loop PC register.
- Reg3 (G3reg) Consists of status registers, page registers.

PX Register

The PX register, an 8-bit extension register, enables applications to transfer 24-bit data between 24-bit memory and 16-bit data registers. Only 24-bit accesses of 24-bit memory use the PX register. (So, a 16-bit read of 24-bit memory does not load the PX register, and a 16-bit write fills the lower eight bits in 24-bit memory with zeros (0).)

On reads, the PX register stores the lower eight bits of the 24-bit data transferring from memory to a destination register, and on writes, it supplies them for the data written to 24-bit data space.

Only two instructions use the PX register:

- ALU/MAC with dual indirect memory reads (see page [“Compute with Dual Memory Read”](#) on page 6-3)
- Indirect 24-bit memory read or write with pre- or postmodify addressing option (see [page 7-37](#) and [page 7-41](#))

To access 24-bit memory, you typically use the `PM(Ireg += Mreg)` syntax shown here:

```
AX1 = PM(I0 += M2);    /* Read 24 bits, load 16 MSbits in AX1 */
                       /* PX auto-loaded w/8 memory LSbits */
AY1 = PX;              /* Load lower 8 bits from PX in AY1 */

PX = MR2;              /* Load lower 8 bits into PX */
PM(I4 += M5) = MR1;   /* Write all 24 bits from MR1 and PX */
```

On data reads using the `PX` register, the DSP transfers the upper sixteen bits of the 24-bit data to the destination data register and the lower eight bits to the `PX` register. The data loaded from memory is right-justified in the destination registers.

On data writes using the `PX` register, the DSP transfers the upper sixteen bits of the 24-bit data from the bus and the lower eight bits from the `PX` register, except for indirect writes of 24-bit immediate data, in which the instruction supplies the eight LSBs. The data written is right-justified in memory.



`PX` transfers to and from memory occur automatically and transparently to the user, but the user must transfer data between the `PX` register and the data registers.

Because the DSP has a unified memory space, the address, not the syntax, determines whether the reference accesses 16-bit memory or 24-bit memory at run time.

- For 24-bit references that read 16-bit memory, the `PX` register receives whatever data the memory system outputs for the eight LSBs. For internal memory, this value is `0x00`.
- For 24-bit references that write 16-bit memory, the DSP discards the data in the `PX` register.

Data Move Instructions

DAG Registers

The DAGs generate memory addresses for data transfers to and from memory. To do so, each uses a set of address registers and a page register. For fast context switching during interrupt servicing, the DAGs provide a secondary set of address registers. This section describes these registers.

DAG Address Registers

Each DAG has a set of address registers that it uses to generate memory addresses for loading or storing data in memory. Each DAG can use its own set of address registers only. DAG1 uses registers 0 through 3, and DAG2 uses registers 4 through 7.

The DAG address registers are:

- **Index (Ireg)** Pointer to the current memory address. DAG1 (I0-I3); DAG2 (I4-I7).
- **Modify (Mreg)** Offset (from index) value for pre- or post-modify addressing. DAG1 (M0-M3); DAG2 (M4-M7).
- **Length (Lreg)** Number of memory locations in a buffer. DAG1 (L0-L3); DAG2 (L4-L7). For linear buffers, you must explicitly set $L_{reg} = 0$; for circular buffers, you must explicitly set L_{reg} to the length of the buffer.
- **Base (Breg)** Starting address of a circular buffer. DAG1 (B0-B3); DAG2 (B4-B7). Used with circular buffering only.

Each base (Breg) and length (Lreg) register is associated with its specific index (Ireg) register—I0/B0/L0, I1/B1/L1, ..., and I7/B7/L7. So, although you can mix and match any of the index (Ireg) and modify (Mreg) registers within the same DAG group, you must always use the base (Breg) and length (Lreg) register that is associated with the particular index (Ireg) register you use.

DAG Page Registers (DMPGx)

The DAGs and their associated page registers generate 24-bit addresses for accessing the data needed by instructions. For data accesses, the DSP's unified memory space is organized into 256 pages, with 64K locations per page. The page registers provide the eight MSBs of the 24-bit address, specifying the page on which the data is located. The DAGs provide the sixteen LSBs of the 24-bit address, specifying the exact location of the data on the page.

- The DMPG1 page register is associated with DAG1 (registers I0–I3) indirect memory accesses as well as immediate, direct memory accesses. It supplies the upper 8 MSBs for direct memory addressed instructions.
- The DMPG2 page register is associated with DAG2 (registers I4–I7) indirect memory accesses.

At power up, the DSP initializes both page registers to 0x0. Although initializing page registers is unnecessary unless the data is located on other than the current page. For good programming practice, we recommend that you set the corresponding page register whenever you initialize a DAG index register (Ireg) to set up a data buffer.

For example,

```
DMPG1 = 0x12;          /* set page register */
/* or DMPG1 = page(data_buffer); for relative addressing */

I2 = 0x3456;          /* init data buffer; 24b addr=0x123456 */
L2 = 0;               /* define linear buffer */
M2 = 1;               /* increment address by one */
                       /* two stall cycles inserted here */
DM(I2 += M2) = AX0;   /* write data to buffer and update I2 */
```



DAG register (DMPGx, Ireg, Mreg, Lreg, Breg) loads can incur up to two stall cycles when a memory access based on the initialized register immediately follows the initialization.

Data Move Instructions

To avoid these unproductive stall cycles, you could code the memory access sequence like this:

```
DMPG1 = 0x12;          /* set page register */
/* or DMPG1 = page(data_buffer); for relative addressing */

I2 = 0x3456;          /* init data buffer; 24b addr=0x123456 */
L2 = 0;               /* define linear buffer */
M2 = 1;               /* increment address by one */
AX0 = 0xAAAA;
AR = AX0 - 1;
DM(I2 += M2) = AR;    /* write data to buffer and update I2 */
```

Typically, you load both page registers with the same page value (0-255), but you can increase memory flexibility by loading each with a different page value. For example, loading the page registers with different page values, you could:

- Separate DMA space from the application's data space
- Perform high-speed data transfers between pages

This operation is not automatic and requires explicit programming.

Secondary DAG Registers

The secondary set of DAG address registers (Ireg, Mreg, Lreg, and Breg) enable single-cycle context-switching to support real-time control functions and to reduce overhead associated with interrupt servicing.

By default, system power-up and reset enable the primary set of DAG address registers. To enable or disable the secondary address registers, you must set or clear, respectively, the SEC_DAG bit (bit 6) in MSTAT (for details, see [“Mode Status \(MSTAT\) Register” on page 2-11](#)). The instruction set provides three methods for doing so. Each method incurs a latency, which is the delay between the time the instruction effecting the change executes and the time the change takes effect and is available to other instructions. [Table 7-2 on page 7-10](#) shows the latencies associated with each method.

When switching between primary and secondary DAG registers, applications need to account for the latency associated with the method they use. For example, after the `MSTAT = data12;` instruction, three cycles of latency occur before the mode change takes effect. So, you must issue at least three instructions after `MSTAT = 0x20;` before attempting to use the new set of DAG registers. Otherwise, you will overwrite the primary set and lose data.

The `ENA/DIS mode` instruction is more efficient for enabling and disabling DSP modes since it incurs no cycles of effect latency. For example:

```
CCODE = 0x9; NOP;
If SWCOND JUMP do_data; /* Jump to do_data */
do_data:
    ENA SEC_DAG;          /* Switch to 2nd DAGs */
    ENA SEC_REG;         /* Switch to 2nd Dregs */
    AX0 = DM(buffer);    /* if buffer empty, go */
    AR = PASS AX0;       /* right to fill and */
    IF NE JUMP fill;     /* get new data */
    RTI;
fill:                    /* fill routine */
    NOP;
buffer:                  /* buffer data */
    NOP;
```

Register Load Latencies

An effect latency occurs when some instructions write or load a value into a register, which changes the value of one or more bits in the register. Effect latency refers to the time it takes after the write or load instruction for the effect of the new value to become available for other instructions to use.

Effect latency values are given in terms of instruction cycles. A 0 latency means that the effect of the new value is available on the next instruction following the write or load instruction. For register changes that have an effect latency greater than 0, make sure you do not try to use the register right after you write or load a new value to avoid using the old value.

Data Move Instructions

Table 7-2 gives the effect latencies for writes or loads of various interrupt and status registers.

Table 7-2. Effect latencies for register changes

Register	Bits	REG = value	ENA/DIS mode	POP STS	SET/CLR INT
ASTAT	All	1 cycle	NA	0 cycles	NA
CCODE	All	1 cycle	NA	NA	NA
CNTR	All	1 cycle ¹	NA	NA	NA
ICNTL	All	1 cycle	NA	NA	0 cycles
IMASK	All	1 cycle	NA	0 cycles	NA
MSTAT	SEC_REG	1 cycle	0 cycles	1 cycle	NA
	BIT_REV	3 cycles	0 cycles	3 cycles	NA
	AV_LATCH	0 cycles	0 cycles	0 cycles	NA
	AR_SAT	1 cycle	0 cycles	1 cycle	NA
	M_MODE	1 cycle	0 cycles	1 cycle	NA
	TIMER	1 cycle	0 cycles	1 cycle	NA
	SEC_DAG	3 cycles	0 cycles	3 cycles	NA

¹ This latency applies only to IF COND instructions, not to the DO UNTIL instruction. Loading the CNTR register has 0 effect latency for the DO UNTIL instruction.



A PUSH or POP PC has one cycle of latency for all SSTAT register bits, but a PUSH or POP LOOP or STS has one cycle of latency only for the STKOVERFLOW bit in the SSTAT register.

When you load some Group 2 and 3 registers (see Table 7-1 on page 7-3), the effect of the new value is not immediately available to subsequent

instructions that might use it. For interlocked registers (DAG address and page registers, `IOPG`, `IJPG`), the DSP automatically inserts stall cycles as needed, but for noninterlocked registers, to accommodate the required latency, you must insert either the necessary number of `NOP` instructions or other instructions that are not dependent upon the effect of the new value.

The noninterlocked registers are:

- Status registers `ASTAT` and `MSTAT`
- Condition code register `CCODE`
- Interrupt control register `ICNTL`

The number of `NOP` instructions you must insert is specific to the register and the load instruction as shown in [Table 7-2](#). A zero (0) latency indicates that the new value is effective on the next cycle after the load instruction executes. An n latency indicates that the effect of the new value is available up to n cycles after the load instruction executes. When you use a modified register before the required latency, you may get the register's old value.

Since unscheduled or unexpected events (interrupts, DMA operations, etc.) often interrupt normal program flow, do not rely on these load latencies when you structure your program's flow. A delay in executing a subsequent instruction based on a newly loaded register could result in erroneous results—whether the subsequent instruction is based on the effect of the register's new or old value.



Load latency applies only to the time it takes the loaded value to effect the change in operation, not to the number of cycles required to load the new value. A loaded value is always available to a read access on the next instruction cycle.

Data Move Instructions

Data Addressing Methods

The instruction set supports two addressing methods for accessing memory data:

- Direct addressing The user supplies an explicit address in the instruction.
- Indirect addressing The DAG address registers generate addresses.

Direct Addressing

Direct addressing is the simplest method to use. An explicit address or a label included in the instruction specifies the address of a memory access. A label is a symbolic name that you assign to an address.

You specify an explicit address or label in a data move instruction like this:

```
DM(I1 += M0) = 0x1234; /* write data 0x1234 and post-modify */
AX0 = DM(0x3333);      /* read location 0x3333, put in AX0 */
DM(port1) = AY1;       /* write value in AY1 to port1 */
port1:                  /* port1 address should be in linker ldf */
NOP;
```

When you use a label, you can either specify the address that the label references or let the VisualDSP linker assign the label an address. For details on assigning label addresses, see the *VisualDSP User's Guide for ADSP-219x Family DSPs*.

Indirect Addressing

Indirect addressing uses a pointer to specify the address of a memory access. The DAG index (*Ireg*) and modify (*Mreg*) registers implement address pointers for indirect addressing. The *Ireg* supplies the address value, and the *Mreg* supplies the modify (offset) value, which, added to the address value, forms the address of the next memory location. The instruc-

tion set provides two address modification options—premodify with no update and postmodify with update.

- Premodify addressing—no update

Premodify addressing does not permanently change the value of the index register (*Ireg*). In premodify operations, the sum of the *Ireg* and *Mreg* register values provides the address of the memory access. After the access, the *Ireg* register retains its original value.

For example, setting up a DAG1 linear data buffer using the premodify option:

```
#define buffer1 0x2
DMPG1 = page(buffer1);
I0 = buffer1;
M0 = 0x0007;
L0 = 0; /* Unless L = 0 buffer is circular */
AX0 = DM(I0 + M0); /* AX0 receives data @ I0+M0 */
/* I0 retains original value */
```

or a DAG2 linear data buffer premodified with a constant:

```
#define buffer2 0x3
DMPG2 = page(buffer2);
I4 = buffer2;
L4 = 0; /* Unless L = 0 buffer is circular */
AX0 = DM(I4 + 0x0007); /* AX0 receives data @ I4+0x0007 */
/* I4 retains original value */
```

- Postmodify addressing—with update

Postmodify addressing permanently changes the value in the index (*Ireg*) register. In postmodify operations, the current value in *Ireg* is used for the memory access. After the access, the DSP adds the

Data Move Instructions

modify value in `Mreg` to the address value in `Ireg` and overwrites the contents in `Ireg` with the result.

For example, setting up a DAG1 linear data buffer using the post-`modify` option:

```
#define buffer3 0x2
DMPG1 = page(buffer3);
I0 = buffer3;
M0 = 0x0007;
L0 = 0; /* Unless L = 0 buffer is circular */
AX0 = DM(I0 += M0); /* AX0 receives data @ I0+M0 */
/* updated with sum of (I0+M0) */
```

or a DAG1 linear data buffer postmodified with a constant:

```
#define buffer4 0x3
DMPG1 = page(buffer4);
I0 = buffer4;
L0 = 0; /* Unless L = 0 buffer is circular */
AX0 = DM(I0 += 0x0003); /* AX0 receives data @ I0+0x0003 */
/* I0 updated w/sum of (I0+0x0003) */
```



Circular buffers work with `postmodify` addressing only.

To set up data buffers, you can mix and match any of the DAG index (`Ireg`) and modify (`Mreg`) registers within the same DAG group (DAG1 or DAG2), but not between DAG groups. Length (`Lreg`) and base address (`Breg`) registers, when used, must always match their corresponding `Ireg`. For example, the following code is valid, because it uses corresponding `Ireg` and `Lreg` registers:

```
DMPG1 = page(data_in);
I3 = data_in;
M1 = 0x0007;
L3 = 0; /* Unless Lreg = 0 buffer is circular */
AX0 = DM(I3 += M1);
data_in; /* data_in location could elsewhere */
NOP;
```

Circular Data Buffer Addressing

Circular data buffers enable applications to reuse the same data buffer; for example, to store the filter coefficients for a FIR or IIR filter or to act as a delay line for the convolution of an input signal.

A circular data buffer is a set of memory locations used for storing a set of data. A circular data buffer is defined by a set of DAG address registers:

- **Base (B0-B7)** Starting address.
These registers are off core, so you must use the data (Dreg) registers and this syntax to access them:
`REG(Breg) = Dreg;`
- **Index (I0-I7)** Current address.
The Ireg points to the current address within the buffer. After the access, the Ireg is postmodified with the address of next access.
- **Modify (M0-M7)** Number of locations offset from the current address. To calculate the address of the next access, the offset value in Mreg is added to the current Ireg value, and the result is written to Ireg.
- **Length (L0-L7)** Number of memory locations in the buffer.

An index pointer (Ireg) steps through the data buffer, forwards or backwards, in programmable increments as determined by a modifier (Mreg) value. The base address (Breg) and the buffer's length (Lreg) keep the pointer within the range of the buffer's memory locations. When the index pointer steps outside the buffer's address range, the logic adds or subtracts the buffer's length from the index value to wrap the pointer back to the top or bottom of the buffer, as appropriate.

For example, the following code sets up a circular data buffer:

```
.section/dm seg_data;
.VAR coeff_buffer[13] = 0,1,2,3,4,5,6,7,8,9,10,11,12;
```

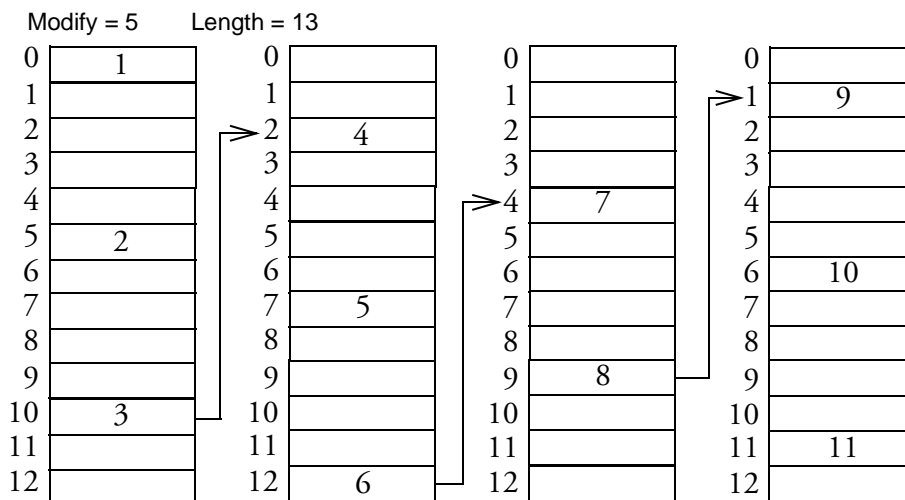
Data Move Instructions

```

.section/pm seg_code;
DMPG2 = page(coeff_buffer); /* Set the memory page */
I4 = coeff_buffer;          /* Set the current addr */
M5 = 5;                      /* Set the modify value */
L4 = LENGTH(coeff_buffer); /* If L = 0 buffer is linear */
AX0 = I4;                    /* Copy the base addr into AX0 */
REG(B4) = AX0;              /* Set the buffer's base addr */
AR = AX1 AND AY0;
AR = DM(I4 += M5);          /* Read 1st buffer location */

```

Figure 7-1, using this code example, demonstrates how the index pointer steps through the circular buffer.



- ⊘ Do not place the index pointer for a circular buffer such that it crosses a memory page boundary during post-modify addressing. All memory locations in a circular buffer must reside on the same memory page.

Bit-Reversed Addressing

Bit-reversed addressing is frequently used in FFT calculations to obtain results in sequential order. Because FFT operations repeatedly subdivide data sequences, the data or twiddle factors may be scrambled, loaded or stored in bit-reversed order.

For performing FFT operations, you can reverse the order in which DAG1 outputs its address bits. DAG2 always outputs its address bits in normal, Big Endian format. Since the two DAGs operate independently, you can use them in tandem, with one generating sequentially ordered addresses and the other generating bit-reversed addresses, to perform memory reads and writes of the same FFT data.

To use bit-reversed addressing, you set bit 1 in the `MSTAT` register (`ENA BIT_REV`). When enabled, DAG1 outputs all addresses generated by its index registers (`I0–I3`) in bit-reversed order. The reversal applies only to the address value DAG1 outputs, not to the address value stored in the index (`Ireg`) register, so the `Ireg` value is stored in Big Endian format. Bit-reversed mode remains in effect until you clear bit 1 in the `MSTAT` register (`DIS BIT_REV`).

Bit reversal operates on the binary number that represents the position of a sample within an array of samples. Using 3-bit addresses, [Table 7-3](#) shows the position of each sample within an array before and after the bit-reverse operation. For example, sample `x4` occupies position `0b100` in sequential order, but position `0b001` in bit-reversed order. Bit reversing transposes the bits of a binary number about its midpoint, so `0b001` becomes `0b100`, `0b011` becomes `0b110`, and so on. Some numbers, like

Data Move Instructions

0b000, 0b111, and 0b101, remain unchanged and retain their original position within the array.

Table 7-3. 8-point array sequence before and after bit reversal

Sequential Order		Bit Reversed Order	
Sample	Binary	Binary	Sample
x0	000	000	x0
x1	001	100	x4
x2	010	010	x2
x3	011	110	x6
x4	100	001	x1
x5	101	101	x5
x6	110	011	x3
x7	111	111	x7

Bit-reversing the samples in a sequentially ordered array scrambles their positions within the array. Bit-reversing the samples in a scrambled array restores their sequential order within the array.

In full 16-bit reversed addressing, bits 7 and 8 of the 16-bit address are the pivot points for the reversal:

Normal	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Bit-reversed	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

FFT operations often need only a few address bits reversed; for example, a 16-point sequence requires four reversed bits, and a 1024-bit sequence

requires ten reversed bits. You can bit-reverse address values less than 16-bits—which reverses a specified number of LSBs only. Bit-reversing less than the full 16-bit index register value requires that you add the correct modify value to the index pointer after each memory access to generate the correct bit-reversed addresses.

To set up bit-reversed addressing for address values < 16 bits, you need to determine:

- The number of bits to reverse (N)

You use this value to calculate the modify value.

- The starting address of the linear data buffer

The starting address of an array that the program accesses with bit-reversed addressing must be zero or an integer multiple of the number of bits to reverse (starting address = $0, N, 2N, \dots$).

- The first bit-reversed address that the DAG will output

This value is the buffer's starting address, but with the N LSBs bit-reversed.

- The initialization value for the index (I_{reg})

You initialize the index (I_{reg}) register with the bit-reversed value of the first bit-reversed address the DAG will output.

- The correct modify value (M_{reg}) with which to update the index pointer after each memory access

Use this formula to calculate the modify value: $M_{reg} = 2^{(16-N)}$.

As an example, we'll set up bit-reversed addressing that reverses the eight address LSBs ($N = 8$) of a data buffer with a starting address of $0x0020$ ($4N$).

Data Move Instructions

We need to determine the:

- First bit-reversed address that DAG1 will output

This value is the buffer's starting address (0x0020) with bits[7:0] reversed: 0x0004.

0x0020	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
0x0004	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0

- Initialization value for the index (Ireg) register

This is first bit-reversed address DAG1 will output (0x0004) with bits[15:0] reversed: 0x2000.

0x0004	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
0x2000	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0

- Correct modify value for Mreg

This is 2^{16-N} which evaluates to 2^8 or 0x0100.

[Listing 7-1](#) shows the code for this example.

Listing 7-1. Bit-reversed addressing, 8 LSBs

```
br_adds: I4=read_in;          /* DAG2 pointer to input samples */
        IO=0x0200;          /* Base address of bit_rev output */
        M4=1;              /* DAG2 increment by 1 */
        M0=0x0100;        /* DAG1 increment for 8-bit rev. */
        L4=0;             /* Linear data buffer */
        L0=0;             /* Linear data buffer */
        CNTR=8;          /* 8 samples */
        ENA BIT_REV;     /* Enable DAG1 bit reverse mode */
        DO brev UNTIL CE;
        AY1=DM(I4+=M4);  /* Read samples sequentially */
```

Data Addressing Methods

```
    brev: DM(I0+=M0)=AY1; /* Write results nonsequentially */
DIS BIT_REV;             /* Disable DAG1 bit reverse mode */
    RTS;                 /* Return to calling routine */
read_in:                 /* input buffer, could be .extern */
    NOP;
```

Data Move Instructions

Register to Register Move

Dreg1	=	Dreg2	;
G1reg1		G1reg2	
G2reg1		G2reg2	
G3reg1		G3reg2	

FUNCTION

Moves the contents in the source register to the destination register. The contents in the source register are right-justified in the destination register.

SOURCE

REG2 can be any core register, which are listed in [Table 7-1 on page 7-3](#).

DESTINATION

REG1 can be any core register, which are listed in [Table 7-1 on page 7-3](#).

DETAILS

For transfers in which the destination register is MR1 or SR1, this operation:

- Sign extends into MR2 or SR2, respectively, the value stored in MR1 or SR1 if the source is a signed value.
- Zero-fills MR2 or SR2, respectively, the value stored in MR1 or SR1 if the source is an unsigned value.

For transfers in which the destination register is smaller than the source register, this operation right-justifies the value in the destination register, such that bit 0 maps to bit 0, and truncates the extraneous high-order bits.

When you load the CCODE, ASTAT, or MSTAT register, the effect of the new value is not available immediately to subsequent instructions that are

based on it. You must insert the required number of NOP instructions before using the modified register, or your instruction may execute based the old value. For more information, see [“Register Load Latencies” on page 7-9](#)).

SSTAT is a read-only register, so `SSTAT = reg;` is invalid instruction syntax.

EXAMPLES

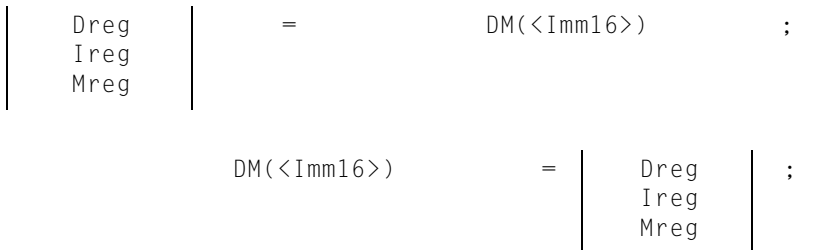
```
I0 = I4;           /* load I0 from I4 */
CCODE = AY0;      /* load CCODE from AY0 */
DMPG1 = DMPG2;    /* load DMPG1 from DMPG2 */
```

SEE ALSO

- [“Type 17: Any Reg «... Any Reg” on page 9-39](#)
- [“Core Registers” on page 7-2](#)
- [“PX Register” on page 7-4](#)
- [“DAG Registers” on page 7-6](#)
- [“Register Load Latencies” on page 7-9](#)
- [“Direct Register Load” on page 7-27](#).

Data Move Instructions

Direct Memory Read/Write—Immediate Address



FUNCTION

The memory read operation moves the contents of the memory location specified by an immediate 16-bit value or label into the destination register.

The memory write operation moves the contents of the source register into the memory location specified by an immediate 16-bit value or label.

SOURCE

Reads The data comes from the memory location specified by an immediate 16-bit value or label.

Writes The data comes from any register file data (Dreg) register or DAG Index (Ireg) or DAG Modify (Mreg) register:

Register File
AX0, AX1, AY0, AY1, AR, MX0, MX1, MY0, MY1, MR0, MR1, MR2, SR0, SR1, SR2, SI

DAG1/DAG2 Index and Modify Registers
I0, I1, I2, I3, I3, I4, I6, I7, M0, M1, M2, M3, M4, M5, M6, M7

Direct Memory Read/Write—Immediate Address

DESTINATION

Reads The data goes to any register file data (D_{reg}) register or DAG Index (I_{reg}) or DAG Modify (M_{reg}) register.

Writes The data goes to the memory location specified by an immediate 16-bit value or label.

DETAILS

This instruction is typically used by memory-intensive applications that must make highly efficient use of memory. Applications that use absolute memory locations need to configure and use them with care. For information on using absolute memory locations, see the *Linker & Utilities Manual for ADSP-219x Family DSPs* and *Assembler Manual for ADSP-219x Family DSPs*.

This instruction transfers 16-bit data only over the DM bus. It does not write or read from the PX register.

When you load 16-bit data into MR1 or SR1, it is sign-extended into MR2 or SR2, respectively.

DMPG1 provides the eight MSBs of the address. For details, see [“DAG Page Registers \(DMPGx\)” on page 7-7](#).

When you load a DAG address or page register, the new value is not available immediately to subsequent instructions that use the register for a memory access. The DAG address registers have a two-cycle latency.



Because the DAG registers are interlocked, the DSP automatically inserts up to two stall cycles, as needed, to ensure that subsequent instructions use the new address value.

For efficient programming, insert two instructions that do not use the modified register in the two instruction lines immediately following the

Data Move Instructions

load instruction. For example, separate the DMPG1 load and the memory access with two other DAG register loads:

```
I0 = buffer;      /* Ireg load, data_in defined on page 7-14 */
NOP;              /* any two non-DAG1 instructions */
NOP;              /* can execute here without latencies */
AX0 = DM(I0 + 0); /* memory access */
```

EXAMPLES

```
SI = DM(data_in); /* Dreg load, label defined on page 7-14 */
I4 = DM(coeff_buffer);
                    /* Ireg load, label defined on page 7-15 */
M5 = DM(coeff_buffer);
                    /* Mreg load, label defined on page 7-15 */

DM(coeff_buffer) = AX1;
                    /* Dreg load, label defined on page 7-15 */
DM(data_in) = I0;   /* Ireg load, label defined on page 7-14 */
DM(data_in) = M1;   /* Mreg load, label defined on page 7-14 */
```

SEE ALSO

- [“Type 3: Dreg/Ireg/Mreg «…» DM/PM” on page 9-22](#)
- [“Direct Addressing” on page 7-12](#)
- [“Core Registers” on page 7-2](#)
- [“DAG Registers” on page 7-6](#)
- [“Secondary DAG Registers” on page 7-8](#)
- [“Register Load Latencies” on page 7-9](#)

Direct Register Load

Dreg	= <Data16> ;
G1reg	
G2reg	
G3reg	= <Data12> ;

FUNCTION

Loads the destination register with immediate data supplied in the instruction. The data is right-justified in the destination register.

You use the <data16> instruction to load a value into the data registers, to initialize the DAG address registers, to enable or disable interrupts, and to load certain stack registers.

You use the <data12> instruction to load G3reg registers that are less than 16-bits wide. You load the short registers to set or clear one or more bits in the status registers, to set up flag conditions, to set the various page registers, and to load certain stack registers. For a list of the core registers, see [Table 7-1 on page 7-3](#).

SOURCE

The <data16> instruction accepts a 16-bit immediate value, a pointer to a 16-bit variable, or LENGTH(16-bit variable).

The <data12> instruction accepts only an immediate value ≤ 12 bits.

Data Move Instructions

DESTINATION

The <data16> instruction places the data in any register group 0, 1, or 2 register:

Register Group 0 (Dreg), 1 (G1reg), & 2 (G2reg) Registers

AX0, AX1, AY0, AY1, AR, MX0, MX1, MY0, MY1, MR0, MR1, MR2, SR0, SR1, SR2, SI, I0, I1, I2, I3, I3, I4, I6, I7, M0, M1, M2, M3, M4, M5, M6, M7, L0, L1, L2, L3, L4, L5, L6, L7, IMASK, IRPTL, ICNTL, STACKA, CNTR, LPCSTACKA, SB, SE
--

The <data12> instruction places the data in any register group 3 register:

Register Group 3 (G3reg) Registers (writable)

ASTAT, MSTAT, LPCSTACKP, CCODE, SE, SB, PX, DMPG1, DMPG2, IOPG, IJPG, STACKP
--



SSTAT is a read-only register.

DETAILS

When you load 16-bit data into MR1 or SR1, it is sign-extended into MR2 or SR2, respectively.

When you use the <data12> instruction to load a 16-bit register (SE, or SB), the destination register's MSBs are filled with zeros (0).

When you load certain registers (CCODE, ASTAT, MSTAT, IJPG, Ireg, or DMPGx), the new value is not available immediately to subsequent instructions. For information on register latencies, see [“Register Load Latencies” on page 7-9](#).

EXAMPLES

```
/* Loading 16-bit registers with 16-bit values: */
AR = 0x5409; /* Dreg put data */
I2 = coeff_buffer;
           /* Ireg put addr, label defined on page 7-15 */
M0 = 0x1234; /* Mreg put data */
L3 = LENGTH(coeff_buffer);
           /* put leng, label def'd on page 7-15 */

/* Loading 12-bit Reg3 registers with short constants: */
STACKP = 0;
MSTAT = 0x4; /* Enable AV_latch */
```

SEE ALSO

- [“Type 6: Dreg «... Data16” on page 9-24](#)
- [“Type 7: Reg1/2 «... Data16” on page 9-25](#)
- [“Type 33: Reg3 «... Data12” on page 9-56](#)
- [“Core Registers” on page 7-2](#)
- [“DAG Registers” on page 7-6](#)
- [“Secondary DAG Registers” on page 7-8](#)
- [“Register Load Latencies” on page 7-9](#)
- [“System Control Register Read/Write” on page 7-61](#)

Data Move Instructions

Indirect 16-bit Memory Read/Write—postmodify

Dreg	= DM(Ireg += Mreg) ;	;
G1reg		
G2reg		
G3reg		

DM(Ireg += Mreg) =	Dreg	;
	G1reg	
	G2reg	
	G3reg	

FUNCTION

Transfers 16-bit data between memory and any of the core registers (Dreg, G1reg, G2reg, or G3reg) over the DM bus. The current value in Ireg provides the address for the memory access. After the access, Ireg is updated with the sum of its current value and the value in Mreg.

SOURCE

Reads The 16-bit data comes from the memory location pointed to by the address in the Ireg, which is modified after the access by the value in the Mreg:

- DM/DAG1 I0, I1, I2, or I3 (index registers)
M0, M1, M2, or M3 (modify registers)
- PM/DAG2 I4, I5, I6, or I7 (index registers)
M4, M5, M6, or M7 (modify registers)



You can use any index register with any modify register from the same DAG. You cannot pair a DAG1 register with a DAG2 register.

Indirect 16-bit Memory Read/Write—postmodify

Writes The 16-bit data comes from any core register, except `SSTAT`, which is a read-only register. For information on core registers, see [Table 7-1 on page 7-3](#).

DESTINATION

Reads The 16-bit data goes to any core register. For information on core registers, see [Table 7-1 on page 7-3](#).

Writes The 16-bit data goes to the memory location pointed to by the address in the `Ireg`, which is modified after the access by the value in the `Mreg`.

DETAILS

On reads and writes, the data is right-justified in the destination location (bit0 of the transfer data maps to bit0 of the destination). If the width of the destination register is less than sixteen bits, the extraneous MSBs of the data are discarded. On writes from source registers less than sixteen bits, the missing high-order bits are zero-filled in the memory location.

As shown in [Figure 7-2](#), if this instruction actually references 24-bit data space at runtime, a read operation loads bits 23:8 from the memory location into bits 15:0 of a 16-bit register (or bits 23:16 into bits 7:0 of an 8-bit register, and so on). The low-order bits of the memory location are ignored. Conversely, a write operation stores bits 15:0 from a 16-bit source register into bits 23:8 of the 24-bit memory location and zero-fills the low-order bits 7:0.

To implement a linear data buffer, you must initialize the `Ireg`'s corresponding `Lreg` to 0. For details, see [“DAG Registers” on page 7-6](#).

To implement circular buffer addressing, you must initialize the `Ireg`'s corresponding `Lreg` to the length of the buffer and its corresponding `Breg` with the base address of the buffer. For details, see [“Circular Data Buffer Addressing” on page 7-15](#).

Data Move Instructions

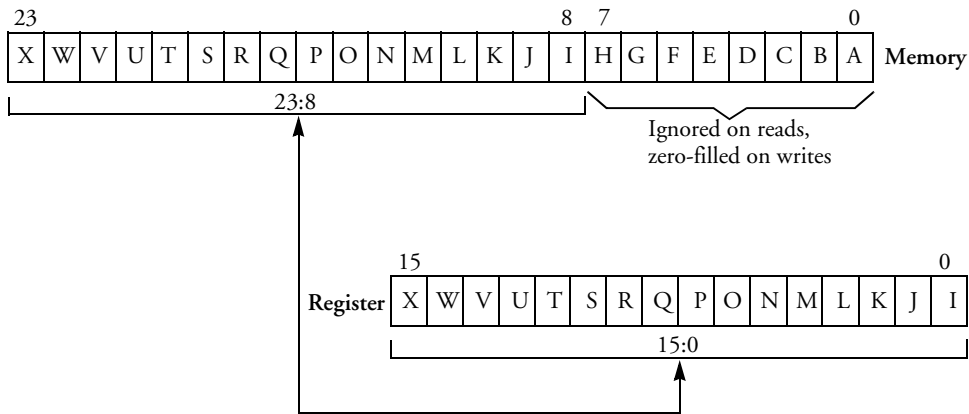


Figure 7-2. 24-bit DM bus transactions

To perform bit-reversed addressing, you must use DAG1 address registers. For details, see [“Bit-Reversed Addressing”](#) on page 7-17.

A DAG page register, DMPG1 (I3-I0) or DMPG2 (I7-I4), provides the eight MSBs of the memory address. For details, see [“DAG Page Registers \(DMPGx\)”](#) on page 7-7.

When you load certain registers (CCODE, ASTAT, MSTAT, IJPG, Ireg, or DMPGx), the new value is not available immediately to subsequent instructions. For information on register latencies, see [“Register Load Latencies”](#) on page 7-9.

EXAMPLES

```

/* This code segment demonstrates Indirect 16-bit Memory Reads
and Writes with postmodify and incurs no stall cycles: */

#define taps 10
.SECTION/DM seg_data;
.VAR signal_buffer[taps];
.VAR coeffs[taps];
.SECTION/PM seg_code;
init:
    I0 = coeff_buffer;

```

Indirect 16-bit Memory Read/Write—postmodify

```
                /* Ireg put addr, label def'd on page 7-15 */
I5 = coeffs;
M0 = 1;
M5 = 1;
L0 = LENGTH(coeff_buffer);
L5 = LENGTH(coeffs);
AX0 = I0;
AX1 = I5;
REG(B0) = AX0;
REG(B5) = AX1;
DMPG1 = 0x0;
DMPG2 = 0x0;
CNTR = taps;
DO clear UNTIL CE;
DM(I5 += M5) = 0;
clear:
DM(I0 += M0) = 0;
```

SEE ALSO

- [“Type 32: Any Reg «…» PM/DM” on page 9-54](#)
- [“Core Registers” on page 7-2](#)
- [“DAG Registers” on page 7-6](#)
- [“Secondary DAG Registers” on page 7-8](#)
- [“Register Load Latencies” on page 7-9](#)

Data Move Instructions

Indirect 16-bit Memory Read/Write—premodify

Dreg	= DM(Ireg + Mreg) ;
G1reg	
G2reg	
G3reg	

DM(Ireg + Mreg) =	Dreg ;
	G1reg
	G2reg
	G3reg

FUNCTION

Transfers 16-bit data between memory and any of the core registers (Dreg, G1reg, G2reg, or G3reg) over the DM bus. The value in Mreg added to the value in Ireg provides the address for the memory access. No update occurs after the access, so Ireg retains its original value.

SOURCE

Reads The 16-bit data comes from the memory location addressed by the Ireg plus Mreg; the Ireg retains its original value:

- **DM/DAG1** I0, I1, I2, or I3 (index registers)
M0, M1, M2, or M3 (modify registers)
- **PM/DAG2** I4, I5, I6, or I7 (index registers)
M4, M5, M6, or M7 (modify registers)



You can use any index register with any modify register from the same DAG. You cannot pair a DAG1 register with a DAG2 register.

Indirect 16-bit Memory Read/Write—premodify

Writes The 16-bit data comes from any core register, except `SSTAT`, which is a read-only register. For information on core registers, see [Table 7-1 on page 7-3](#).

DESTINATION

Reads The 16-bit data goes to any core register. For information on core registers, see [Table 7-1 on page 7-3](#).

Writes The 16-bit data goes to the memory location addressed by the `Ireg` plus `Mreg`; the `Ireg` retains its original value

DETAILS

On reads and writes, the data is right-justified in the destination location (bit0 of the transfer data maps to bit0 of the destination). If the width of the destination register is less than sixteen bits, the overflow MSBs of the data are discarded. On writes from source registers less than sixteen bits, the missing high-order bits are zero-filled in the memory location.

If this instruction actually references 24-bit data space at runtime, a read operation loads bits 23:8 from the memory location into bits 15:0 of a 16-bit register (or bits 23:16 into bits 7:0 of an 8-bit register, and so on). The low-order bits of the memory location are ignored. Conversely, a write operation stores bits 15:0 from a 16-bit source register into bits 23:8 of the 24-bit memory location and zero-fills the low-order bits 7:0. For details, see [Figure 7-2 on page 7-32](#).

A DAG page register, `DMPG1` (I3-I0) or `DMPG2` (I7-I4), provides the eight MSBs of the memory address. For details, see [“DAG Page Registers \(DMPGx\)” on page 7-7](#).

When you load certain registers (`CCODE`, `ASSTAT`, `MSTAT`, `IJPG`, `Ireg`, or `DMPGx`), the new value is not available immediately to subsequent instructions. For information on register latencies, see [“Register Load Latencies” on page 7-9](#).

Data Move Instructions

You cannot use circular buffering with this instruction, so you must initialize the `Ireg`'s corresponding `Lreg` to 0. For details, see [“Indirect Addressing” on page 7-12](#).

To perform bit-reversed addressing, you must use DAG1 address registers. For details, see [“Bit-Reversed Addressing” on page 7-17](#).

EXAMPLES

```
.SECTION/DM seg_data;
.VAR look_tbl[3] = 0x0, 0x1, 0x2;

.SECTION/PM seg_code;
cases:
    DMPG1 = 0x1;
    IO = look_tbl;
    M0 = 0;
    M1 = 1;
    M2 = 2;
    L0 = 0;
    AR = AX0 + AX1;
    IF EQ JUMP cases_end;
case1:
    AY0 = DM(IO + M0);    /* read from premodified location */
    IF GT JUMP cases_end;
case2:
    DM(IO + M1) = AY0;    /* write to premodified location */
cases_end:
    NOP;
```

SEE ALSO

- [“Type 32: Any Reg «…» PM/DM” on page 9-54](#)
- [“Core Registers” on page 7-2](#)
- [“DAG Registers” on page 7-6](#)
- [“Secondary DAG Registers” on page 7-8](#)
- [“Register Load Latencies” on page 7-9](#)

Indirect 24-bit Memory Read/Write—postmodify

Dreg	=	PM(Ireg += Mreg)	;	;
G1reg				
G2reg				
G3reg				

PM(Ireg += Mreg) =	Dreg	;
	G1reg	
	G2reg	
	G3reg	

FUNCTION

Transfers 24-bit data between memory and any of the core registers (Dreg, G1reg, G2reg, or G3reg) over the PM bus. Employs the PX register to hold the low-order bits 7:0 while it transfers the high-order bits 23:8 directly between memory and the destination register.


The current value in Ireg provides the address for the memory access. After the access, Ireg is updated with the sum of its current value and the value in Mreg.

SOURCE

Reads The 24-bit data comes from the memory location pointed to by the address in the Ireg, which is modified after the access by the value in the Mreg:

- **DM/DAG1** I0, I1, I2, or I3 (index registers)
 M0, M1, M2, or M3 (modify registers)
- **PM/DAG2** I4, I5, I6, or I7 (index registers)
 M4, M5, M6, or M7 (modify registers)

Data Move Instructions

 You can use any index register with any modify register from the same DAG. You cannot pair a DAG1 register with a DAG2 register.

Writes The 24-bit data comes from any core register, except `SSTAT`, which is a read-only register. For information on core registers, see [Table 7-1 on page 7-3](#).

DESTINATION

Reads The 24-bit data goes to any core register. For information on core registers, see [Table 7-1 on page 7-3](#).

Writes The 24-bit data goes to the memory location pointed to by the address in the `Ireg`, which is modified after the access by the value in the `Mreg`.

DETAILS

Unless this instruction is already in the instruction cache, it causes a one-cycle stall.

The 8-bit `PX` register holds the eight low-order bits of 24-bit data transferring between memory and a register. On reads, it automatically stores these bits, and on writes, it supplies them. On reads, you must explicitly move the contents of `PX` into a data register, and on writes, you must explicitly load the `PX` register with the value of the low-order bits. For details, see [“PX Register” on page 7-4](#).

On reads, the high-order bits 23:8 of the memory location are right-justified in the destination register (bit8 of the transfer data maps to bit0 of the destination register). If the width of the destination register is less than sixteen bits, the overflow MSBs of the data are discarded. For details, see [Figure 7-2 on page 7-32](#).

On writes, bits 15:0 of the source register are right-justified in the memory location (bit0 of the transfer data maps to bit8 of the memory location). If the width of the source register is less than sixteen bits, the missing high-order bits of the memory location are zero-filled.

Indirect 24-bit Memory Read/Write—postmodify

If this instruction actually references 16-bit data space at runtime, the `PX` register receives eight bits of the adjacent data. On writes, the contents of the `PX` register are ignored.

A DAG page register, `DMPG1` (I3-I0) or `DMPG2` (I7-I4), provides the eight MSBs of the memory address. For details, see [“DAG Page Registers \(DMPGx\)” on page 7-7](#).

When you load certain registers (`CCODE`, `ASTAT`, `MSTAT`, `IJPG`, `Ireg`, or `DMPGx`), the new value is not available immediately to subsequent instructions. For information on register latencies, see [“Register Load Latencies” on page 7-9](#).

To implement a linear data buffer, you must initialize the `Ireg`'s corresponding `Lreg` to 0. For details, see [“Indirect Addressing” on page 7-12](#).

To implement circular buffer addressing, you must initialize the `Ireg`'s corresponding `Lreg` to the length of the buffer and its corresponding `Breg` with the base address of the buffer. For details, see [“Circular Data Buffer Addressing” on page 7-15](#).

To perform bit-reversed addressing, you must use DAG1 address registers. For details, see [“Bit-Reversed Addressing” on page 7-17](#).

EXAMPLES

```
#define more_taps 10
.SECTION/DM seg_dmda;
.VAR dmdata_buffer[more_taps];
.SECTION/PM seg_pmda;
.VAR pmdata_coeffs[more_taps];
.SECTION/PM seg_code;
more_init:
    I0 = dmdata_buffer;      /* dmdag Ireg write/output address */
    I5 = pmdata_coeffs;     /* pmdag Ireg read/input address */
    M0 = 1;
    M5 = 1;
    L0 = LENGTH(dmdata_buffer);
    L5 = LENGTH(pmdata_coeffs);
```

Data Move Instructions

```
AX0 = I0;
AX1 = I5;
REG(B0) = AX0;
REG(B5) = AX1;
DMPG1 = 0x0;
DMPG2 = 0x0;
CNTR = taps;
SI = 0xB6A3;          /* shifter input word */
DO clear UNTIL CE;
SR1 = PM(I5 += M5); /* read upper 16-bits and post modify */
SR0 = PX;           /* read lower 8-bits from PX */
SR = SR OR ASHIFT SI BY 3 (HI); /* ashift upper word */
more_clear:
DM(I0 += M0) = SR0;
                /* 16-bit write SR0 & post modify address */
```

SEE ALSO

- [“Type 32: Any Reg «…» PM/DM” on page 9-54](#)
- [“Core Registers” on page 7-2](#)
- [“DAG Registers” on page 7-6](#)
- [“Secondary DAG Registers” on page 7-8](#)
- [“Register Load Latencies” on page 7-9](#)

Indirect 24-bit Memory Read/Write—premodify

Dreg	= PM(Ireg + Mreg) ;	;
G1reg		
G2reg		
G3reg		

PM(Ireg + Mreg) =	Dreg	;
	G1reg	
	G2reg	
	G3reg	

FUNCTION

Transfers 24-bit data between memory and any of the core registers (Dreg, G1reg, G2reg, or G3reg) over the PM bus. Employs the PX register to hold the low-order bits 7:0 while it transfers the high-order bits 23:8 directly between memory and the destination register. The value in Mreg added to the value in Ireg provides the address for the memory access. No update occurs after the access, so Ireg retains its original value.

SOURCE

Reads The 24-bit data comes from the memory location addressed by the Ireg plus Mreg; the Ireg retains its original value:

- **DM/DAG1** I0, I1, I2, or I3 (index registers)
M0, M1, M2, or M3 (modify registers)
- **PM/DAG2** I4, I5, I6, or I7 (index registers)
M4, M5, M6, or M7 (modify registers)



You can use any index register with any modify register from the same DAG. You cannot pair a DAG1 register with a DAG2 register.

Data Move Instructions

Writes The 24-bit data comes from any core register, except `SSTAT`, which is a read-only register. For information on core registers, see [Table 7-1 on page 7-3](#).

DESTINATION

Reads The 24-bit data goes to any core register. For information on core registers, see [Table 7-1 on page 7-3](#).

Writes The 24-bit data goes to the memory location addressed by the `Ireg` plus `Mreg`; the `Ireg` retains its original value

DETAILS

Unless this instruction is already in the instruction cache, it causes a one-cycle stall.

The 8-bit `PX` register holds the eight low-order bits of 24-bit data transferring between memory and a register. On reads, it automatically stores these bits, and on writes, it supplies them. On reads, you must explicitly move the contents of `PX` into a data register, and on writes, you must explicitly load the `PX` register with the value of the low-order bits. For details, see [“PX Register” on page 7-4](#).

On reads, the high-order bits 23:8 of the memory location are right-justified in the destination register (bit8 of the transfer data maps to bit0 of the destination register). If the width of the destination register is less than sixteen bits, the overflow MSBs of the data are discarded. For details, see [Figure 7-2 on page 7-32](#).

On writes, bits 15:0 of the source register are right-justified in the memory location (bit0 of the transfer data maps to bit8 of the memory location). If the width of the source register is less than sixteen bits, the missing high-order bits of the memory location are zero-filled.

A DAG page register, `DMPG1` (I3-I0) or `DMPG2` (I7-I4), provides the eight MSBs of the memory address. For details, see [“DAG Page Registers \(DMPGx\)” on page 7-7](#).

Indirect 24-bit Memory Read/Write—premodify

When you load certain registers (CCODE, ASTAT, MSTAT, IJPG, Ireg, or DMPGx), the new value is not available immediately to subsequent instructions. For information on register latencies, see [“Register Load Latencies” on page 7-9](#).

You cannot use circular buffering with this instruction, so you must initialize the Ireg’s corresponding Lreg to 0. For details, see [“Indirect Addressing” on page 7-12](#).

To perform bit-reversed addressing, you must use DAG1 address registers. For details, see [“Bit-Reversed Addressing” on page 7-17](#).

EXAMPLES

```
.SECTION/DM seg_data;
.VAR lookup_tbl[3] = 0x0, 0x1, 0x2;

.SECTION/PM seg_code;
pmrw_cases:
    DMPG1 = 0x1;
    IO = lookup_tbl;
    M0 = 0;
    M1 = 1;
    M2 = 2;
    LO = 0;
    AR = AX0 + AX1;
    IF EQ JUMP cases_end;
pmrw_case1:
    SR1 = PM(IO + M1); /* pre modify and read upper 16-bits */
    SR0 = PX;          /* read lower 8-bits from PX */
    SR = SR OR ASHIFT SI BY 3 (HI); /* ashift upper word */
    IF GT JUMP cases_end;
pmrw_case2:
    PX = SR1;          /* Load lower 8 bits into PX */
    PM(IO + M2) = SR0; /* Write all 24 bits from SR0 and PX */
pmrw_cases_end:
    NOP;
```

Data Move Instructions

SEE ALSO

- [“Type 32: Any Reg «…» PM/DM” on page 9-54](#)
- [“Core Registers” on page 7-2](#)
- [“DAG Registers” on page 7-6](#)
- [“Secondary DAG Registers” on page 7-8](#)
- [“Register Load Latencies” on page 7-9](#)

Indirect DAG Register Write (premodify or postmodify), with DAG Register Move

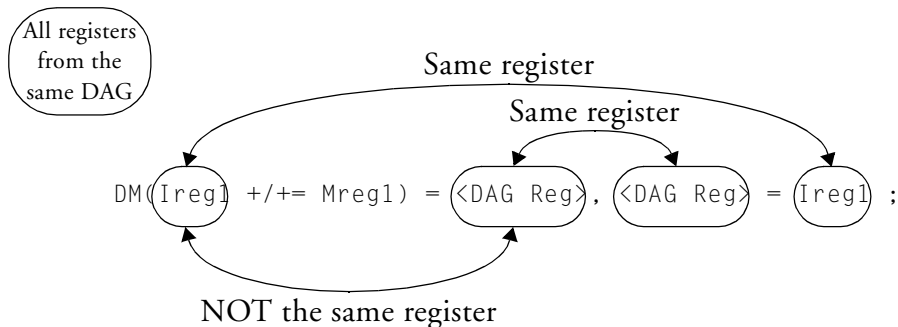
Indirect DAG Register Write (premodify or postmodify), with DAG Register Move

$$\left| \begin{array}{c} \text{DM}(\text{Ireg1}) \\ \text{+} \\ \text{+} \\ \text{=} \\ \text{Mreg1} \end{array} \right| = \left| \begin{array}{c} \text{Ireg2} \\ \text{Mreg2} \\ \text{Lreg2} \end{array} \right| , \quad \left| \begin{array}{c} \text{Ireg2} \\ \text{Mreg2} \\ \text{Lreg2} \end{array} \right| = \text{Ireg1} ;$$

FUNCTION

Writes the contents of a source address register to memory and loads the same source address register with a new value—the effective address written to memory. Register usage within this instruction has the following restrictions (shown below graphically):

- The *Ireg1* registers must be the same register.
- The *Mreg1* register must come from the same DAG as *Ireg1*.
- The *Ireg2*, *Mreg2*, or *Lreg2* registers be the same register.
- The *Ireg2*, *Mreg2*, or *Lreg2* registers must come from the same DAG as *Ireg1*, but may not be *Ireg1* ($\text{Ireg1} \neq \text{Ireg2}$).



If the premodify (+) addressing option is used, after the memory access, the instruction loads the source address register with the modified value of *Ireg* ($\text{Ireg} + \text{Mreg}$). If the postmodify (+=) addressing option is used, the

Data Move Instructions

instruction loads the source address register with the unmodified value of `Ireg`. For details on the indirect addressing options, see “[Indirect Addressing](#)” on page 7-12.

SOURCE

Memory write The 16-bit data comes from any DAG (`Ireg`, `Mreg`, or `Lreg`) register in the same DAG as `Ireg1`. The source register may not be the `Ireg1` register. For information on core registers, see [Table 7-1 on page 7-3](#).


Register load The 16-bit data comes from the `Ireg1` register.

DESTINATION

Memory write For the post-modified access (`+=`), the 16-bit data goes to the memory location pointed to by the address in the `Ireg`, which is modified after the access by the value in the `Mreg`.

For the pre-modified access (`+`), the 16-bit data goes to the memory location addressed by the `Ireg` plus `Mreg`; the `Ireg` retains its original value:

- **DM/DAG1** `I0`, `I1`, `I2`, or `I3` (index registers)
 `M0`, `M1`, `M2`, or `M3` (modify registers)
- **PM/DAG2** `I4`, `I5`, `I6`, or `I7` (index registers)
 `M4`, `M5`, `M6`, or `M7` (modify registers)

 You can use any index register with any modify register from the same DAG. You cannot pair a DAG1 register with a DAG2 register.

Register load The 16-bit data goes to any DAG (`Ireg`, `Mreg`, or `Lreg`) register in the same DAG as `Ireg1`. The destination register may not be the `Ireg1` register. For information on core registers, see [Table 7-1 on page 7-3](#).

Indirect DAG Register Write (premodify or postmodify), with DAG Register Move

DETAILS

On reads and writes, the data is right-justified in the destination location (bit0 of the transfer data maps to bit0 of the destination). If the width of the destination register is less than sixteen bits, the overflow MSBs of the data are discarded. On writes from source registers less than sixteen bits, the missing high-order bits are zero-filled in the memory location.

If this instruction actually references 24-bit data space at runtime, a read operation loads bits 23:8 from the memory location into bits 15:0 of a 16-bit register (or bits 23:16 into bits 7:0 of an 8-bit register, and so on). The low-order bits of the memory location are ignored. Conversely, a write operation stores bits 15:0 from a 16-bit source register into bits 23:8 of the 24-bit memory location and zero-fills the low-order bits 7:0. For details, see [Figure 7-2 on page 7-32](#).

A DAG page register, DMPG1 (I3-I0) or DMPG2 (I7-I4), provides the eight MSBs of the memory address. For details, see [“DAG Page Registers \(DMPGx\)” on page 7-7](#).

When you load certain registers (CCODE, ASTAT, MSTAT, IJPG, Ireg, or DMPGx), the new value is not available immediately to subsequent instructions. For information on register latencies, see [“Register Load Latencies” on page 7-9](#).

You cannot use circular buffering with the pre-modify addressing (+) form of this instruction, so you must initialize the Ireg’s corresponding Lreg to 0. For details, see [“Indirect Addressing” on page 7-12](#).

To perform bit-reversed addressing, you must use DAG1 address registers. For details, see [“Bit-Reversed Addressing” on page 7-17](#).

Data Move Instructions

EXAMPLES

```
/* This routine uses the type 32a instruction to save the
previous frame pointer and allocate a new frame pointer, before
calling C-callable subroutine. */
```

```
_memalloc:
DM(I4 += M5)=I5, I5=I4; /* save old FP and allocate new FP */
AX1 = DM(I5 + 1);      /* read a 16 bit parameter from stack */
I2=0xFFFF;           /* load preserved register I2 */
I6=0xFFFF;           /* load scratch register I6 */
DM(I4 += M5) = AX1;   /* put argument on stack for call */
I7=I6;                /* save scratch register I6 */
CALL _malloc;         /* call C function malloc */

_malloc:
NOP;                  /* _malloc code here */
```

SEE ALSO

- [“Type 32a: DM«...DAG Reg | DAG Reg«...Ireg” on page 9-55](#)
- [“Core Registers” on page 7-2](#)
- [“DAG Registers” on page 7-6](#)
- [“Secondary DAG Registers” on page 7-8](#)
- [“Register Load Latencies” on page 7-9](#)

Indirect Memory Read/Write—immediate postmodify

```
Dreg = DM(Ireg += <Imm8>) ;
```

```
DM(Ireg += <Imm8>) = Dreg ;
```

FUNCTION

Transfers 16-bit data between memory and a data register over the DM bus. The current value in `Ireg` provides the address for the memory access. After the access, `Ireg` is updated with the sum of its current value and the immediate 8-bit, two's-complement value supplied in the instruction.

SOURCE

Reads The contents of a memory location in data space pointed to by `Ireg` (I0-I7).

Writes Any of these data registers:

Register File
AX0, AX1, AY0, AY1, AR, MX0, MX1, MY0, MY1, MR0, MR1, MR2, SR0, SR1, SR2, SI

DESTINATION

Reads A data register (same as data registers—write).

Writes The contents of a memory location in data space pointed to by `Ireg` (same as read source registers).

DETAILS

The immediate value supplied in the instruction is an 8-bit two's-complement number. Valid values range from -128 through 127 .

Data Move Instructions

On reads and writes, the data is right-justified in the destination location (bit0 of the transfer data maps to bit0 of the destination).

If this instruction actually references 24-bit data space at runtime, a read operation loads bits 23:8 from the memory location into bits 15:0 of a 16-bit register (or bits 23:16 into bits 7:0 of an 8-bit register, and so on). The low-order bits of the memory location are ignored. Conversely, a write operation stores bits 15:0 from a 16-bit source register into bits 23:8 of the 24-bit memory location and zero-fills the low-order bits 7:0. For details, see [Figure 7-2 on page 7-32](#).

A DAG page register, DMPG1 (I3-I0) or DMPG2 (I7-I4), provides the eight MSBs of the memory address. For details, see [“DAG Page Registers \(DMPGx\)” on page 7-7](#).

When you load certain registers (CCODE, ASTAT, MSTAT, IJPG, Ireg, or DMPGx), the new value is not available immediately to subsequent instructions. For information on register latencies, see [“Register Load Latencies” on page 7-9](#).

To implement a linear data buffer, you must initialize the Ireg’s corresponding Lreg to 0. For details, see [“Indirect Addressing” on page 7-12](#).

To implement circular buffer addressing, you must initialize the Ireg’s corresponding Lreg with the length of the buffer and its corresponding Breg with the base address of the buffer. For details, see [“Circular Data Buffer Addressing” on page 7-15](#).

To perform bit-reversed addressing, you must use DAG1 address registers. For details, see [“Bit-Reversed Addressing” on page 7-17](#).

EXAMPLES

```
AX0 = DM(I0 += 0x11);  
DM(I6 += 0x08) = MR1;  
DM(I2 += -3) = SI;
```

Indirect Memory Read/Write—immediate postmodify

SEE ALSO

- “Type 29: Dreg «…» DM” on page 9-51
- “Core Registers” on page 7-2
- “DAG Registers” on page 7-6
- “Secondary DAG Registers” on page 7-8
- “Register Load Latencies” on page 7-9

Data Move Instructions

Indirect Memory Read/Write—immediate premodify

$$Dreg = DM(Ireg + \langle Imm8 \rangle) ;$$
$$DM(Ireg + \langle Imm8 \rangle) = Dreg ;$$

FUNCTION

Transfers 16-bit data between memory and a data register over the DM bus. The immediate 8-bit, two's-complement value supplied in the instruction added to the current value in *Ireg* provides the address for the memory access. No update occurs after the access, so *Ireg* retains its original value.

SOURCE

Reads The contents of a memory location in data space, accessed indirectly using an *Ireg* (I0-I7) and an immediate 8-bit, two's-complement value supplied in the instruction.

Writes Any of these data registers:

Register File
AX0, AX1, AY0, AY1, AR, MX0, MX1, MY0, MY1, MR0, MR1, MR2, SR0, SR1, SR2, SI

DESTINATION

Reads A data register (same as source data registers).

Writes The contents of a memory location in data space, accessed indirectly with an *Ireg* (same as source address registers) and immediate 8-bit, two's-complement value supplied in the instruction.

Indirect Memory Read/Write—immediate premodify

DETAILS

The immediate value supplied in the instruction is an 8-bit twos-complement number. Valid values range from -128 through 127 .

On reads and writes, the data is right-justified in the destination location (bit0 of the transfer data maps to bit0 of the destination).

If this instruction actually references 24-bit data space at runtime, a read operation loads bits 23:8 from the memory location into bits 15:0 of a 16-bit register (or bits 23:16 into bits 7:0 of an 8-bit register, and so on). The low-order bits of the memory location are ignored. Conversely, a write operation stores bits 15:0 from a 16-bit source register into bits 23:8 of the 24-bit memory location and zero-fills the low-order bits 7:0. For details, see [Figure 7-2 on page 7-32](#).

A DAG page register, DMPG1 (I3-I0) or DMPG2 (I7-I4), provides the eight MSBs of the memory address. For details, see [“DAG Page Registers \(DMPGx\)” on page 7-7](#).

When you load certain registers (CCODE, ASTAT, MSTAT, IJPG, Ireg, or DMPGx), the new value is not available immediately to subsequent instructions. For information on register latencies, see [“Register Load Latencies” on page 7-9](#).

You cannot use circular buffering with this instruction, so you must initialize the Ireg’s corresponding Lreg to 0. For details, see [“Indirect Addressing” on page 7-12](#).

To perform bit-reversed addressing, you must use DAG1 address registers. For details, see [“Bit-Reversed Addressing” on page 7-17](#).

EXAMPLES

```
AX0 = DM(I0 + 0x11);  
DM(I6 + 0x08) = MR1;  
DM(I2 + -3) = SI;
```

Data Move Instructions

SEE ALSO


- [“Type 29: Dreg «…» DM” on page 9-51](#)
- [“Core Registers” on page 7-2](#)
- [“DAG Registers” on page 7-6](#)
- [“Secondary DAG Registers” on page 7-8](#)
- [“Register Load Latencies” on page 7-9](#)

Indirect 16-bit Memory Write—immediate data

$DM(Ireg += Mreg) = \langle Data16 \rangle ;$

FUNCTION

Writes a 16-bit data value supplied in the instruction to a memory location over the DM bus. The current value in `Ireg` provides the address for the memory access. After the memory access, `Ireg` is updated with the sum of its current value and the value in `Mreg`.

 This instruction is a two-word instruction and requires (at minimum) two cycles to execute. [For more information, see “Type 22: DM/PM «… Data16” on page 9-45.](#)


SOURCE

$\langle data16 \rangle$ The data comes from a 16-bit number supplied in the instruction.

DESTINATION

Memory write The 16-bit data goes to the memory location pointed to by the address in the `Ireg`, which is modified after the access by the value in the `Mreg`:

- DM/DAG1 I0, I1, I2, or I3 (index registers)
 M0, M1, M2, or M3 (modify registers)
- PM/DAG2 I4, I5, I6, or I7 (index registers)
 M4, M5, M6, or M7 (modify registers)

 You can use any index register with any modify register from the same DAG. You cannot pair a DAG1 register with a DAG2 register.

Data Move Instructions

DETAILS

The data transferred is right-justified in the memory location (bit0 of the data maps to bit0 of the location). If this instruction actually accesses 24-bit data space at runtime, the write operation stores bits 15:0 in bits 15:0 of the 24-bit memory location and zero-fills the high-order bits 23:16.

A DAG page register, DMPG1 (I3-I0) or DMPG2 (I7-I4), provides the eight MSBs of the memory address. For details, see [“DAG Page Registers \(DMPGx\)” on page 7-7](#).

When you load certain registers (CCODE, ASTAT, MSTAT, IJPG, Ireg, or DMPGx), the new value is not available immediately to subsequent instructions. For information on register latencies, see [“Register Load Latencies” on page 7-9](#).

Setting up a data buffer requires initializing one or more additional address registers. For details, see [“Indirect Addressing” on page 7-12](#).

To perform bit-reversed addressing, you must use DAG1 address registers. For details, see [“Bit-Reversed Addressing” on page 7-17](#).

EXAMPLES

```
DMPG1 = page(0x0);      /* selects internal memory */
I3 = 0x8100;           /* selects 16-bit, block 1 */
M2 = 1;
L3 = 0;
DM(I3 += M2) = 0x7743; /* write data16 to memory */
```

SEE ALSO


- [“Type 22: DM/PM «... Data16” on page 9-45](#)
- [“Core Registers” on page 7-2](#)
- [“DAG Registers” on page 7-6](#)

Indirect 24-bit Memory Write—immediate data

```
PM(Ireg += Mreg) = <Data24>:24 ;
```

FUNCTION

Writes a 24-bit data value supplied in the instruction to a memory location over the PM bus. The current value in `Ireg` provides the address for the memory access. After the memory access, `Ireg` is updated with the sum of its current value and the value in `Mreg`.

 This instruction is a two-word instruction and requires (at minimum) two cycles to execute. [For more information, see “Type 22: DM/PM «… Data16” on page 9-45.](#)


SOURCE

`<data24>` The 24-bit data comes from a 24-bit number supplied in the instruction. The `:24` after the data directs the assembler to handle 24-bit data.

DESTINATION

Memory write The 24-bit data goes to the memory location pointed to by the address in the `Ireg`, which is modified after the access by the value in the `Mreg`:

- DM/DAG1 I0, I1, I2, or I3 (index registers)
 M0, M1, M2, or M3 (modify registers)
- PM/DAG2 I4, I5, I6, or I7 (index registers)
 M4, M5, M6, or M7 (modify registers)

 You can use any index register with any modify register from the same DAG. You cannot pair a DAG1 register with a DAG2 register.

Data Move Instructions

DETAILS

The data transferred is right-justified in the memory location (bit0 of the data maps to bit0 of the location).

If this instruction actually accesses 16-bit data space at runtime, the write operation stores bits 23:8 in bits 15:0 of the 16-bit memory location and discards the data's low-order bits 7:0. For details, see [Figure 7-2 on page 7-32](#).

A DAG page register, DMPG1 (I3-I0) or DMPG2 (I7-I4), provides the eight MSBs of the memory address. For details, see [“DAG Page Registers \(DMPGx\)” on page 7-7](#).

When you load certain registers (CCODE, ASTAT, MSTAT, IJPG, Ireg, or DMPGx), the new value is not available immediately to subsequent instructions. For information on register latencies, see [“Register Load Latencies” on page 7-9](#).

Setting up a data buffer requires initializing one or more additional address registers. For details, see [“Indirect Addressing” on page 7-12](#).

To perform bit-reversed addressing, you must use DAG1 address registers. For details, see [“Bit-Reversed Addressing” on page 7-17](#).

EXAMPLES

```
DMPG2 = page(0x0); /* selects internal memory */
I4 = 0x1000; /* selects 24-bit, block 0 */
M5 = 1;
L4 = 0;
PM(I4 += M5) = 0x3F4512:24; /* write 24-bit data to memory */
```

SEE ALSO

- [“Type 22: DM/PM «... Data16” on page 9-45](#)
- [“Core Registers” on page 7-2](#)
- [“DAG Registers” on page 7-6](#)

External IO Port Read/Write

```
Dreg = IO(<Imm10>) ;
```

```
IO(<Imm10>) = Dreg ;
```

FUNCTION

Transfers data between I/O memory space and a data register.

SOURCE

Reads The contents of a location in I/O memory space specified by the 10-bit immediate value (or memory-mapped register name) supplied in the instruction.

Writes Any of these data registers:

Register File
AX0, AX1, AY0, AY1, AR, MX0, MX1, MY0, MY1, MR0, MR1, MR2, SR0, SR1, SR2, SI

DESTINATION

Reads Any of the data registers (same as write source registers).

Writes The contents of a location in I/O memory space specified by the 10-bit immediate value (or memory-mapped register name) supplied in the instruction.

DETAILS

The I/O page register, `IOPG`, with the 10-bit immediate value supplied in the instruction generate the 18-bit address required for accessing I/O memory space. Valid page values (`IOPG`) are 0-255. Valid location values

Data Move Instructions

(10-bit immediate) are 0-1023. The arrangement of these values to make an address appears in [Figure 7-3](#).



Figure 7-3. Addressing I/O memory space and peripherals

At power up, the DSP initializes the IOPG register to 0x0. Although initializing the IOPG register is unnecessary unless the data to access is located on a different page, for good programming practice, we recommend that you do so whenever you access an external I/O port. To do so, you use the direct register load instruction (for details, see [“Direct Register Load” on page 7-27](#)). Then you can read or write the external I/O port.

When you load certain registers (CCODE, ASTAT, MSTAT, IJPG, Ireg, DMPGx, or IOPG), the new value is not available immediately to subsequent instructions. For information on register latencies, see [“Register Load Latencies” on page 7-9](#).

EXAMPLES

```
#define io_address1 0x1FF /* define 10-bit IO address */
IOPG = 0x01;           /* set IOPG to page 1 */
AX0 = 0xaaFF;          /* load AX0 with 16-bit data */
IO(io_address1) = AX0; /* write data to io_address1 */
```

SEE ALSO

- [“Type 34: Dreg «...» IOreg” on page 9-57](#)
- The I/O registers are specific to each ADSP-219x DSP. For register list, see the *ADSP-219x/2191 DSP Hardware Reference*.

System Control Register Read/Write

```
Dreg = REG(<Imm8>) ;
```

```
REG(<Imm8>) = Dreg ;
```

FUNCTION

Transfers data between an internal system control register and a data register.

SOURCE

Reads The contents of a location in system control memory space specified by the 8-bit immediate value, or register name, supplied in the instruction:

- **DAG** *Breg* B0, B1, B2, or B3 (DAG1 base registers)
 B4, B5, B6, or B7 (DAG2 base registers)
- **Sequencer** SYSCTL (system control register)
- **Cache control** CACTL (cache control register)



Except for the DAG base address registers (B0–B7), SYSCTL, and CACTL, the system control registers are specific to each ADSP-219x product. See your *ADSP-219x/2191 DSP Hardware Reference* for a complete list of these registers and their addresses.

Writes Any of these data registers:

Register File
AX0, AX1, AY0, AY1, AR, MX0, MX1, MY0, MY1, MR0, MR1, MR2, SR0, SR1, SR2, SI

Data Move Instructions

DESTINATION

Reads Any of the data registers (same as write source registers).

Writes The contents of a location in system control memory space specified by the 8-bit immediate value, or register mnemonic, supplied in the instruction (same as source registers).

DETAILS

System control memory space consists of 256 locations. These locations are reserved for core-based controls or for peripherals, such as DMA or serial ports, that interface with the core. For more information, see the *ADSP-219x/2191 DSP Hardware Reference*.

You cannot write the system control registers directly. Instead, you must load a data register, then load the system register from the data register.

When you access a DAG base address register (B0-B7), whether you access the primary or secondary set depends on bit 6 (SEC_DAG) in the MSTAT register. For details, see [“Secondary DAG Registers” on page 7-8](#).

When you load certain registers (CCODE, ASTAT, MSTAT, IJPG, Ireg, or DMPGx), the new value is not available immediately to subsequent instructions. For information on register latencies, see [“Register Load Latencies” on page 7-9](#).

EXAMPLES

```
AX0 = 0x0800;           /* load data into AX0 */
REG(B0) = AX0;         /* load AX0 into B0   */
REG(0x0) = AX0;       /* same as above     */
```

SEE ALSO

- [“Type 35: Dreg «...»Sreg” on page 9-58](#)
- The system registers are specific to each ADSP-219x DSP. For register list, see the *ADSP-219x/2191 DSP Hardware Reference*.

Modify Address Register—indirect

```
MODIFY(Ireg += Mreg) ;
```

FUNCTION


Updates the value of an index register without performing a memory access.

Sums the value in `Ireg` with the value in `Mreg` and writes the result to `Ireg`. If you set up circular buffering, this instruction also performs that logic operation.

SOURCE

UpdateThe DAG `Ireg` and `Mreg` registers specified in the instruction:

- **DM/DAG1** `I0, I1, I2, or I3` (index registers)
 `M0, M1, M2, or M3` (modify registers)
- **PM/DAG2** `I4, I5, I6, or I7` (index registers)
 `M4, M5, M6, or M7` (modify registers)

 You can use any index register with any modify register from the same DAG. You cannot pair a DAG1 register with a DAG2 register.

DESTINATION

UpdateThe DAG `Ireg` specified in the instruction.

DETAILS

For linear data buffers, you must initialize the `Ireg`'s corresponding `Lreg` to 0. For details, see [“Indirect Addressing” on page 7-12](#).

For circular buffers, you must initialize the `Ireg`'s corresponding `Lreg` with the length of the buffer and its corresponding `Breg` with the base

Data Move Instructions

address of the buffer. For details, see [“Circular Data Buffer Addressing”](#) on page 7-15.

When you load certain registers (CCODE, ASTAT, MSTAT, IJPG, Ireg, DMPGX, or IOPG), the new value is not available immediately to subsequent instructions. For information on register latencies, see [“Register Load Latencies”](#) on page 7-9.

EXAMPLES

```
I2 = 0x2000;
M1 = 1;
L2 = 0;
MODIFY(I2 += M1); /* updates I2 with value from M1 */
                  /* I2 = 0x2001 */
```

SEE ALSO

- [“Type 21: Modify DagI”](#) on page 9-43
- [“Core Registers”](#) on page 7-2
- [“DAG Registers”](#) on page 7-6

Modify Address Register—direct

```
MODIFY(Ireg += <Imm8>) ;
```

FUNCTION

Updates the value of an index register without performing a memory access.

Sums the value in `Ireg` with the 8-bit, two's-complement immediate value supplied in the instruction and writes the result to `Ireg`. If you set up circular buffering, this instruction also performs that logic operation.

SOURCE

The DAG `Ireg` register and the `Imm8` data specified in the instruction:

- DM/DAG1 I0, I1, I2, or I3 (index registers)
- PM/DAG2 I4, I5, I6, or I7 (index registers)

DESTINATION

The DAG `Ireg` specified in the instruction.

DETAILS

For linear data buffers, you must initialize the `Ireg`'s corresponding `Lreg` to 0. For details, see [“Indirect Addressing” on page 7-12](#).

For circular buffers, you must initialize the `Ireg`'s corresponding `Lreg` to the length of the buffer and its corresponding `Breg` with the base address of the buffer. For details, see [“Circular Data Buffer Addressing” on page 7-15](#).

When you load certain registers (CCODE, ASTAT, MSTAT, IJPG, `Ireg`, DMPGx, or IOPG), the new value is not available immediately to subsequent instruc-

Data Move Instructions

tions. For information on register latencies, see [“Register Load Latencies” on page 7-9](#).

EXAMPLES

```
I2 = 0x2000;
#define mod_val 0x1
Nop;
L2 = 0;
MODIFY(I2 += mod_val); /* updates I2 with mod_val */
                        /* I2 = 0x2001 */
```

SEE ALSO

- [“Type 21a: Modify DagI” on page 9-44](#)
- [“Core Registers” on page 7-2](#)
- [“DAG Registers” on page 7-6](#)