Data Addressing ◻ 4

4.1 OVERVIEW

The ADSP-2106x's two data address generators (DAGs) simplify the task of organizing data by maintaining pointers into memory. The DAGs allow the processor to address memory *indirectly*; that is, an instruction specifies a DAG register containing an address instead of the address value itself.

Data address generator 1 (DAG1) generates 32-bit addresses on the DM Address Bus. Data address generator 2 (DAG2) generates 24-bit addresses on the PM Address Bus. The basic architecture for both DAGs is shown in Figure 4.1 on the following page.

The DAGs also support in hardware some functions commonly used in digital signal processing algorithms. Both DAGs support circular data buffers, which require advancing a pointer repetitively through a range of memory. Both DAGs can also perform a bit-reversing operation, which outputs the bits of an address in reversed order.

4.2 DAG REGISTERS

Each DAG has four types of registers: *Index (I)*, *Modify (M)*, *Base (B)*, and *Length (L)* registers.

An I register acts as a pointer to memory, and an M register contains the increment value for advancing the pointer. By modifying an I register with different M values, you can vary the increment as needed.

B registers and L registers are used only for circular data buffers. A B register holds the base address (i.e. the first address) of a circular buffer. The same-numbered L register contains the number of locations in (i.e. the length of) the circular buffer.

Each DAG contains eight of each type of register:

DAG1 registers (32-bit)	*DAG2 registers (24-bit)*
B0-B7	B8-B15
I0-I7	I8-I15
M0-M7	M8-M15
L0-L7	L8-L15

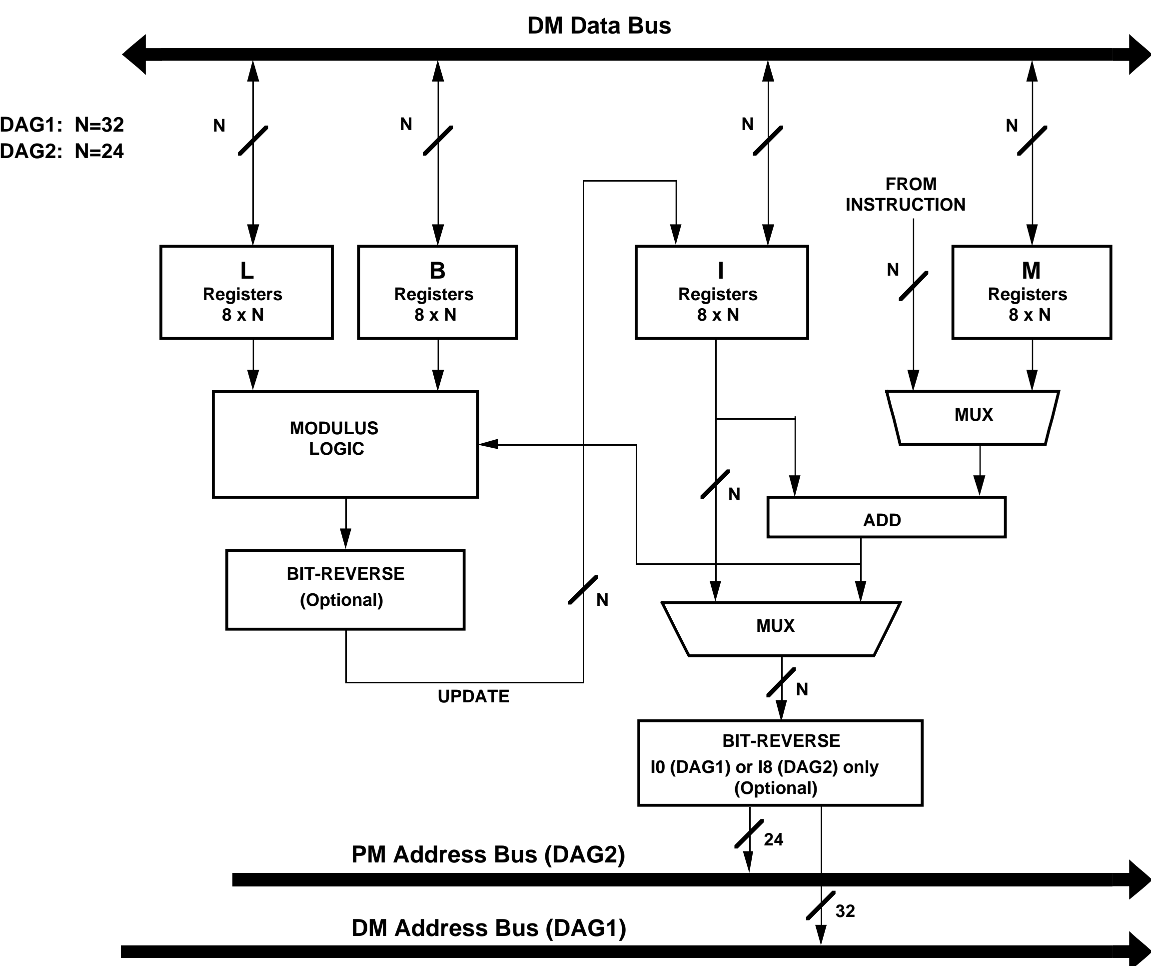

Figure 4.1 Data Address Generator Block Diagram

4.2.1 Alternate DAG Registers

Each DAG register has an alternate (secondary) register for context switching. For activating alternate registers, each DAG is organized into high and low halves, as shown in Figure 4.2. The high half of DAG1 contains the I, M, B and L registers numbered 4-7, and the low half, the registers numbered 0-3. Likewise, the high half of DAG2 consists of registers 12-15, and the low half consists of registers 8-11.

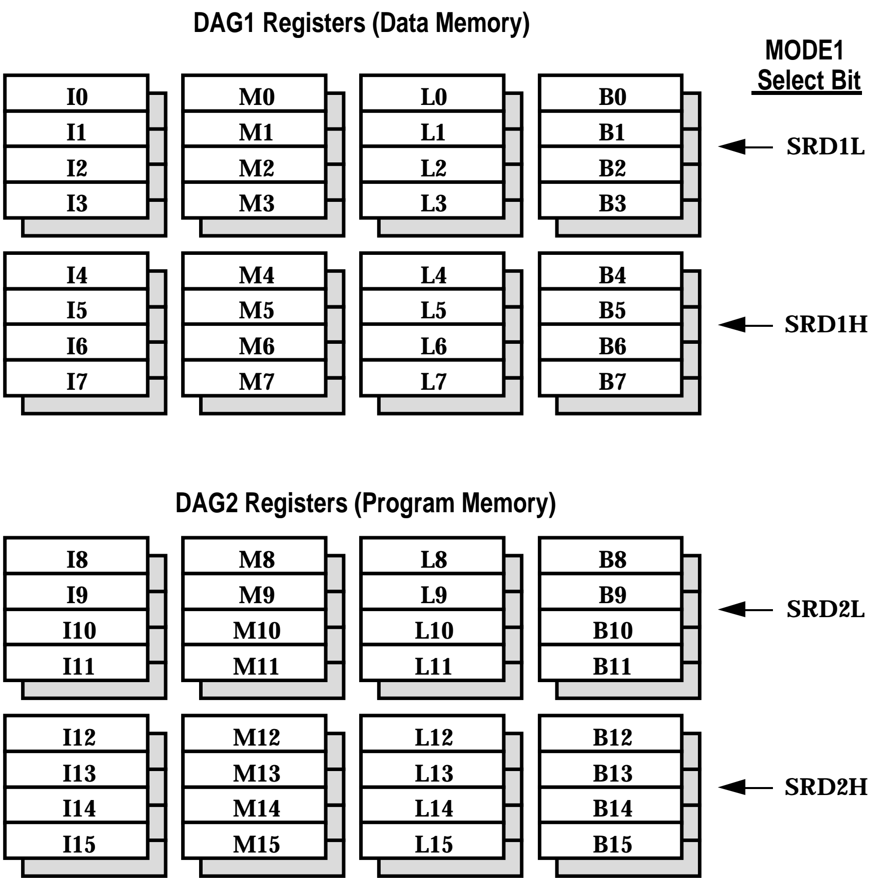

Figure 4.2 Alternate DAG Registers

Several control bits in the MODE1 register determine for each half whether primary or alternate registers are active (0=primary registers, 1=alternate registers):

MODE1 *Bit*	*Name*	*Definition*
3	SRD1H	DAG1 alternate register select (4-7)
4	SRD1L	DAG1 alternate register select (0-3)
5	SRD2H	DAG2 alternate register select (12-15)
6	SRD2L	DAG2 alternate register select (8-11)

This grouping of alternate registers lets you pass pointers between contexts in each DAG.

4.3 DAG OPERATION

DAG operations include:

- address output with pre-modify or post-modify,
- modulo addressing (for circular buffers), and
- bit-reversed addressing

Short word addresses (for 16-bit data) are right-shifted by one bit before being output onto the DM Address Bus. This allows internal memory to use the address directly. (See "16-Bit Short Words" in the *Memory* chapter of this manual for details on short word addresses.)

4.3.1 Address Output & Modification

The processor can add an offset (modifier), either an M register or an immediate value, to an I register and output the resulting address; this is called a *pre-modify without update* operation. Or it can output the I register value as it is, and then add an M register or immediate value to form a new I register value. This is a *post-modify* operation. These operations are compared in Figure 4.3. The pre-modify operation does not change the value of the I register. The width of an immediate modifier depends on the instruction; it can be as large as the width of the I register. The L register and modulo logic do not affect a pre-modified address—pre-modify addressing is always linear, not circular.

Pre-modify addressing operations must not change the memory space of the address; for example, pre-modification of an address in ADSP-2106x Internal Memory Space should not generate an address in External Memory Space. Refer to the *Memory* chapter for information on the ADSP-2106x memory map.

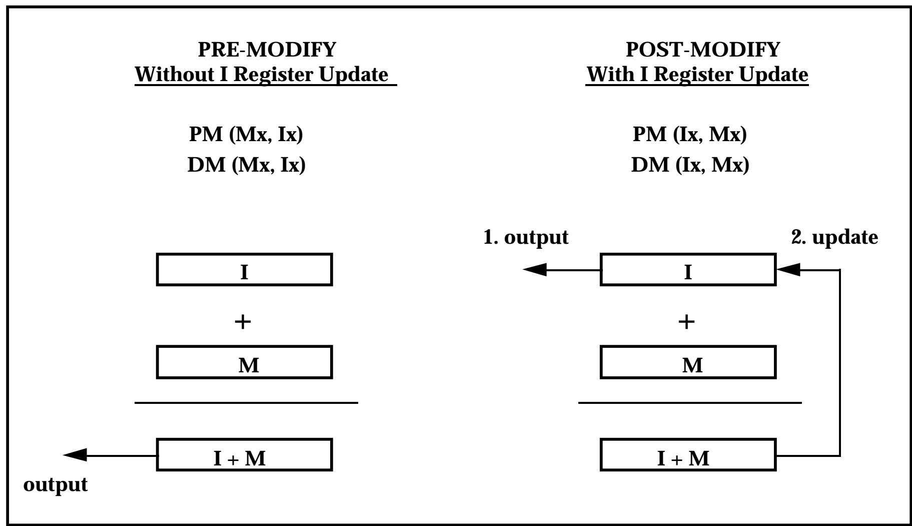

Figure 4.3 Pre-Modify & Post-Modify Operations

4.3.1.1 DAG Modify Instructions

In ADSP-2106x assembly language, pre-modify and post-modify operations are distinguished by the positions of the index and modifier (M register or immediate value) in the instruction. The I register before the modifier indicates a post-modify operation. If the modifier comes first, a pre-modify without update operation is indicated. The following instruction, for example, accesses the program memory location with an address equal to the value stored in I15, and the value I15 + M12 is written back to the I15 register:

```
R6 = PM(I15,M12);      Indirect addressing with post-modify
```

If the order of the I and M registers is switched, however,

```
R6 = PM(M12,I15);      Indirect addressing with pre-modify
```

the instruction accesses the location in program memory with an address equal to I15 + M12, but does not change the value of I15.

Any M register can modify any I register within the same DAG (DAG1 or DAG2). Thus,

```
DM(M0,I2) = TPERIOD;
```

is a legal instruction that accesses the data memory location M0 + I2; however,

```
DM(M0,I14) = TPERIOD;
```

is *not* a legal instruction because the I and M registers belong to different DAGs.

4.3.1.2 Immediate Modifiers

The magnitude of an immediate value that can modify an I register depends on the instruction type and whether the I register is in DAG1 or DAG2. DAG1 modify values can be up to 32 bits wide; DAG2 modify values can be up to 24 bits wide. Some instructions with parallel operations only allow modify values up to 6 bits wide. Here are two examples:

32-bit modifier:

```
R1=DM(0x40000000,I1);            DM address = I1 + 0x4000 0000
```

6-bit modifier:

```
F6=F1+F2,PM(I8,0x0B)=ASTAT;      PM address = I8,  I8 = I8 + 0x0B
```

4.3.2 Circular Buffer Addressing

The DAGs provide for addressing of locations within a circular data buffer. A circular buffer is a set of memory locations that stores data. An index pointer steps through the buffer, being post-modified and updated by the addition of a specified value (positive or negative) for each step. If the modified address pointer falls outside the buffer, the length of the buffer is subtracted from or added to the value, as required to wrap the index pointer back to the start of the buffer (see Figure 4.4). There are no restrictions on the value of the base address for a circular buffer.

Circular buffer addressing must use M registers for post-modify of I registers, not pre-modify; for example:

```
F1=DM(I0,M0);     Use post-modify addressing for circular buffers,
F1=DM(M0,I0);      not pre-modify.
```

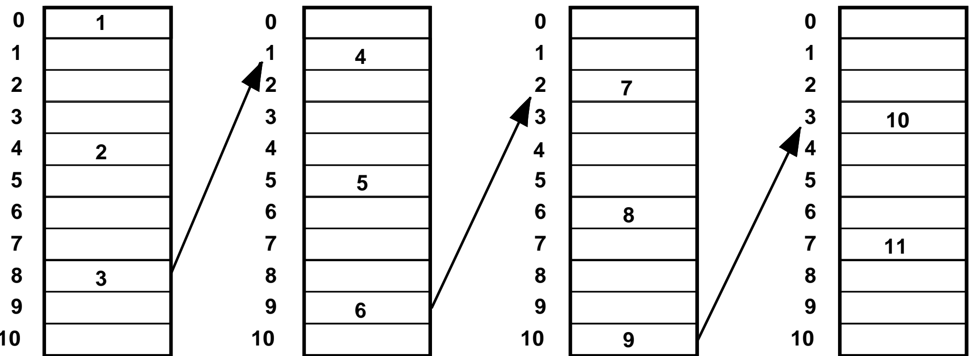

Figure 4.4 Circular Data Buffers

4.3.2.1 Circular Buffer Operation

You set up a circular buffer in assembly language by initializing an L register with a positive, nonzero value and loading the corresponding (same-numbered) B register with the base (starting) address of the buffer. The corresponding I register is automatically loaded with this same starting address.

On the first post-modify access using the I register, the DAG outputs the I register value on the address bus and then modifies it by adding the specified M register or immediate value to it. If the modified value is within the buffer range, it is written back to the I register. If the value is outside the buffer range, the L register value is subtracted (or, if the modify value is negative, added) first.

If M is positive,

$I_{new} = I_{old} + M$ *if I_{old} + M < Buffer base + length (end of buffer)*
$I_{new} = I_{old} + M - L$ *if I_{old} + M ≥ Buffer base + length (end of buffer)*

If M is negative,

$I_{new} = I_{old} + M$ *if I_{old} + M ≥ Buffer base (start of buffer)*
$I_{new} = I_{old} + M + L$ *if I_{old} + M < Buffer base (start of buffer)*

4.3.2.2 Circular Buffer Registers

All four types of DAG registers are involved in the operation of a circular buffer:

- The I register contains the value which is output on the address bus.

- The M register contains the post-modify amount (positive or negative) which is added to the I register at the end of each memory access. The M register can be any M register in the same DAG as the I register and does not have to have the same number. The modify value can also be an immediate number instead of an M register. The magnitude of the modify value, whether from an M register or immediate, must be less than the length (L register) of the circular buffer.

- The L register sets the size of the circular buffer and thus the address range that the I register is allowed to circulate through. L must be positive and cannot have a value greater than $2^{31} - 1$ (for L0-L7) or $2^{23} - 1$ (for L8-L15). If an L register's value is zero, its circular buffer operation is disabled.

- The B register, or the B register plus the L register, is the value that the modified I value is compared to after each access. When the B register is loaded, the corresponding I register is simultaneously loaded with the same value. When I is loaded, B is not changed. B and I can be read independently.

4.3.2.3 Circular Buffer Overflow Interrupts

There is one set of registers in each DAG that can generate an interrupt upon circular buffer overflow (i.e. address wraparound). In DAG1, the registers are B7, I7, L7, and in DAG2 they are B15, I15, L15. Circular buffer overflow interrupts can be used to implement a ping-pong (i.e. swap I/O buffer pointers) routine, for example.

Whenever a circular buffer addressing operation using these registers causes the address in the I register to be incremented (or decremented) past the end (or start) of the circular buffer, an interrupt is generated. Depending on which register set was used, the interrupt is either:

Interrupt	*DAG Registers To Use*	*Vector Address*	*Symbolic Name**
DAG1 circular buffer 7 overflow	B7, I7, L7	0x54	CB7I
DAG2 circular buffer 15 overflow	B15, I15, L15	0x58	CB15I

* These symbols are defined in the #include file `def21060.h.` See "Symbol Definitions File (def21060.h)" at the end of Appendix E, *Control/Status Registers*.

Specifically, an interrupt is generated during an instruction's address post-modify when:

(for M<0) $I + M < B$
(for M≥0) $I + M \geq B + L$

The interrupts can be masked by clearing the appropriate bit in IMASK.

There may be situations where you want to use I7 or I15 without circular buffering but with the circular buffer overflow interrupts unmasked. To disable the generation of these interrupts, set the B7/B15 and L7/L15 registers to values that assure the conditions that generate interrupts (as specified above) never occur. For example, when accessing the address range 0x1000–0x2000, your program could set B=0x0000 and L=0xFFFF. Note that setting the L register to zero will not achieve the desired results.

If you are using either of the circular buffer overflow interrupts, you should avoid using the corresponding I register(s) (I7, I15) in the rest of your program, or be careful to set the B and L registers as described above to prevent spurious interrupt branching.

The STKY status register includes two bits that also indicate the occurrence of a circular buffer overflow, bit 17 (DAG1 circular buffer 7 overflow) and bit 18 (DAG2 circular buffer 15 overflow). These bits are "sticky"—they remain set until explicitly cleared.

4.3.3 Bit-Reversal

Bit-reversal of memory addresses can be performed in two ways: by enabling the bit-reverse mode on DAG1 or DAG2 and using a specific I register (I0 or I8), or by using the explicit bit-reverse instruction (BITREV).

4.3.3.1 Bit-Reverse Mode

In bit-reverse mode, DAG1 bit-reverses 32-bit address values output from I0 and DAG2 bit-reverses 24-bit address values output from I8. These modes are enabled by the BR0 and BR8 bits in the MODE1 register. Only address values from I0 or I8 are bit-reversed. This mode affects both pre-modify and post-modify operations.

MODE1

Bit	*Name*	*Definition*
0	BR8	Bit-reverse mode for I8 (DAG2)
1	BR0	Bit-reverse mode for I0 (DAG1)

Bit-reversal occurs at the output of the DAG and does not affect the value in I0 or I8. In the case of a post-modify operation, the update value is not bit-reversed.

Example:

```
I0=0x80400000;
R1=DM(I0,3);          DM address=0x201,  I0=0x80400003
```

4.3.3.2 Bit-Reverse Instruction

The BITREV instruction modifies and bit-reverses addresses in any DAG index register (I0-I15) without actually accessing memory. This instruction is independent of the bit-reverse mode. The BITREV instruction adds a 32-bit immediate value to a DAG1 index register (or a 24-bit immediate value to a DAG2 index register), bit-reverses the result and writes the result back to the same index register.

Example:

```
BITREV(I1,4);         I1 = Bit-reverse of  (I1+4)
```

4.4 DAG REGISTER TRANSFERS

DAG registers are part of the universal register set and may be written to from memory, from another universal register, or from an immediate field in an instruction. DAG register contents may be written to memory or to a universal register.

Transfers between 32-bit DAG1 registers and the 40-bit DM Data Bus are aligned to bits 39-8 of the bus. When 24-bit DAG2 registers are read to the 40-bit DM Data Bus, M register values are sign-extended to 32 bits and I, L, and B register values are zero-filled to 32 bits. The results are aligned to bits 39-8 of the DM Data Bus. When DAG2 registers are written from the DM Data Bus, bits 31-8 are transferred and the rest are ignored. Figure 4.5 illustrates these transfers.

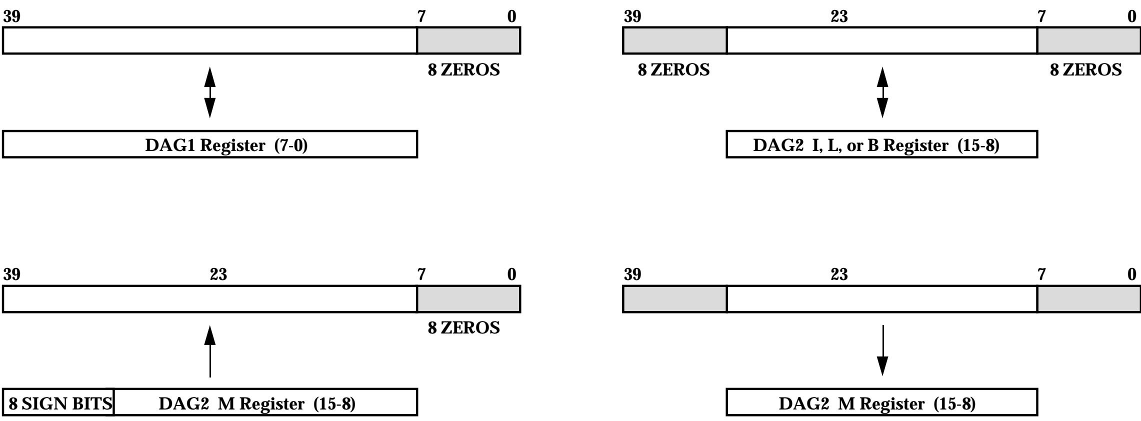

Figure 4.5 DAG Register Transfers

4.4.1 DAG Register Transfer Restrictions

For certain instruction sequences involving transfers to and from DAG registers, an extra (NOP) cycle is automatically inserted by the processor **(1)**. Certain other sequences cause incorrect results and are not allowed by the ADSP-21000 Family assembler **(2)**.

1.) When an instruction that loads a DAG register is followed by an instruction that uses any register in the same DAG for data addressing, modify instructions, or indirect jumps, the ADSP-2106x inserts an extra (NOP) cycle between the two instructions. This happens because the same bus is needed by both operations in the same cycle, therefore the second operation must be delayed.

Example:

```
L2=8;
DM(I0,M1)=R1;
```

Because L2 is in the same DAG as I0 (and M1), an extra cycle is inserted after the write to L2.

2.) The following types of instructions can execute on the processor, but cause incorrect results; these instructions are disallowed by the ADSP-21000 Family assembler:

- An instruction that stores a DAG register in memory using indirect addressing from the same DAG, with or without update of the index register. The instruction writes the wrong data to memory or updates the wrong index register.

  Examples:

  `DM(M2,I1)=I0;` *or* `DM(I1,M2)=I0;`

- An instruction that loads a DAG register from memory using indirect addressing from the same DAG, with update of the index register. The instruction will either load the DAG register or update the index register, but not both.

  Example:

  ```
  L2=DM(I1,M0);
  ```