4 DATA ADDRESS GENERATORS

Overview

This chapter describes the units that control the movement of data to and from the processor and from one data bus to another within the processor. These units include the following:

- Data address generators (DAGs)
- Program Memory Data (PMD) bus and Data Memory Data (DMD) bus exchange unit

Data Address Generators (DAGs)

Every device in the ADSP-218x family contains two independent data address generators so that both program and data memories can be accessed simultaneously. The DAGs provide indirect addressing capabilities. Both perform automatic address modification. For circular buffers, the DAGs can perform modulo address modification.

The two DAGs differ: DAG1 generates only Data Memory (DM) addresses, but provides an optional bit-reversal capability; DAG2 can generate both Data Memory and Program Memory (PM) addresses, but has no bit-reversal capability.

While the following discussion explains the internal workings of the DAGs, bear in mind that the ADSP-218x family development software (assembler and linker) provides a direct method for declaring data buffers as circular or linear.

The software also provides a method for managing the placement of the buffer in memory. Only the initializing of DAG registers must be explicitly programmed (see "Indirect Addressing" on page 4-4 and "Modulo Addressing (Circular Buffers)" on page 4-5).

DAG Registers

Figure 4-1, shows a block diagram of a single data address generator. There are three register files: the modify (M) register file, the index (I) register file, and the length (L) register file. Each of the register files contains four 14-bit registers that can be read from and written to via the DMD bus.

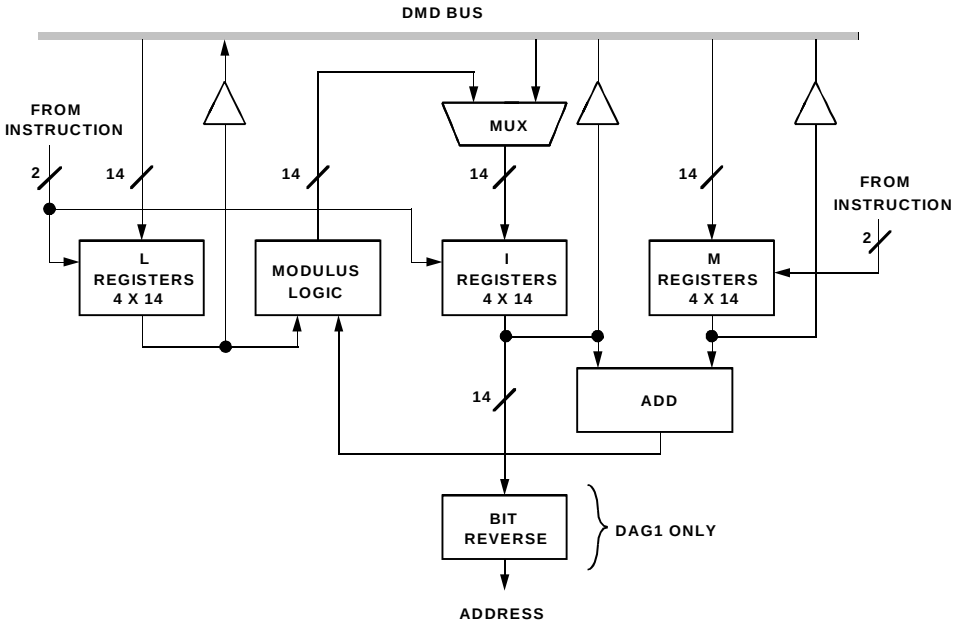

Figure 4-1. Data Address Generator Block Diagram

The I (index) registers (I0-I3 in DAG1, I4-I7 in DAG2) contain the actual addresses used to access memory. When data is accessed in indirect mode, the address stored in the selected I register becomes the memory address. With DAG1, the output address can be bit-reversed by setting the appropriate mode bit in the mode status register (MSTAT) as discussed below or by using the ENA BIT_REV instruction. Bit-reversal facilitates FFT addressing.

The data address generators employ a post-modify scheme; after an indirect data access, the specified M (modify) register (M0-M3 in DAG1, M4-M7 in DAG2) is added to the specified I register to generate the updated I value. The choice of the I and M registers are independent within each DAG. In other words, any register in the I0-I3 set may be modified by any register in the M0-M3 set in any combination, but not by those in DAG2 (M4-M7). The modification values stored in M registers are signed numbers so that the next address can be either higher or lower.

The address generators support both linear addressing and circular addressing. The value of the L (length) register corresponding to an I register (for example, L0 would correspond to I0) determines which addressing scheme is used for that I register. For circular buffer addressing, the L register is initialized with length of the buffer. For linear addressing, the modulus logic is disabled by setting the corresponding L register to zero.

Each time an I register is selected, the corresponding L register provides the modulus logic with the length information. If the sum of the M register and the I register crosses the buffer boundary, the modified I register value is calculated by the modulus logic using the L register value.

All data address generator registers (I, M, and L registers) are loadable and readable from the lower 14 bits of the DMD bus. Since I and L register contents are considered to be unsigned, the upper 2 bits of the DMD bus are padded with zeros when reading them. M register contents are signed; when reading an M register, the upper 2 bits of the DMD bus are sign-extended.

Indirect Addressing

The ADSP-218x family processors allow two addressing modes for Data Memory fetches: direct and register indirect. Indirect addressing is accomplished by loading an address into an I (index) register and specifying one of the available M (modify) registers.

The L registers are provided to facilitate wraparound addressing of circular data buffers. A circular buffer is only implemented when an L register is set to a non-zero value. For linear (i.e. non-circular) indirect addressing, the L register corresponding to the I register used must be set to zero.

Do not assume that the L registers are automatically initialized or may be ignored; the I, M, and L registers contain random values following processor reset. Your program must initialize the L registers corresponding to any I registers it uses.

Linear Indirect Addressing

Setting an L register to a non-zero value activates the processor's circular addressing modulus logic. For linear indirect addressing, you must set the appropriate L register to zero to disable the modulus logic.

Listing 4-1 provides an example of simple linear indirect addressing. Listing 4-2 provides an example of linear indirect addressing that uses a memory variable to store an address pointer.

Listing 4-1. Simple Linear Indirect Addressing

```
I3=0x3800;
M2=0;
L3=0;
AX0=DM(I3,M2);
```

Listing 4-2. Linear Indirect Addressing Using a Memory Variable

```
.VAR addr_ptr;          /* variable holds address to be */
                        /* accessed */
I3=DM(addr_ptr);        /* I3 loaded using direct addressing */
L3=0;                   /* disable circular addressing */
M1=0;                   /* no post-modify of I3 */
AX0=DM(I3,M1);          /* AX0 loaded using indirect */
                        /* addressing */
```

Modulo Addressing (Circular Buffers)

The modulus logic implements automatic modulo addressing for accessing circular data buffers. To calculate the next address, the modulus logic uses the following information:

- The current location, found in the `I` register (unsigned).
- The modify value, found in the `M` register (signed).
- The buffer length, found in the `L` register (unsigned).
- The buffer base address.

From these inputs, the next address is calculated according to the formula:

```
Next Address = (I + M - B) Modulo (L) + B
```

where:

I=current address
M=modify value (signed)
B=base address
L=buffer length
M + I=modified address

The inputs are subject to the condition:

$$|M| < L$$

This condition insures that the next address cannot wrap around the buffer more than once in one operation.

Calculating the Base Address

The base address of a circular buffer of length L is 2^n or a multiple of 2^n, where *n* satisfies the condition:

$$2^{n-1} < L \leq 2^n$$

In other words, the base address is L "rounded" upwards to the closest power of 2 (or its multiple). This rule implies that a certain number of low-order bits of the base address must be zeroes.

In practice, you do not need to calculate *n* yourself; the linker automatically places circular buffers at a proper address.

Circular Buffer Base Address Example 1

For example, let us assume that the buffer length is eight. The length of the buffer must be less than or equal to some value 2^n; *n* therefore, must be three or greater. The left side of the inequality rule specifies that the buffer length must be greater than the value 2^{n-1}; *n* therefore must be three or less. The only value of *n* that satisfies both inequalities is three. Valid base addresses are multiples of 2^n, so in this example valid base addresses are multiples of eight: 0x0008, 0x0010, 0x0018, and so on.

Circular Buffer Base Address Example 2

As a second example, assume a buffer length of seven. The inequality again yields the same value for *n*, namely, three. With a buffer length of seven, therefore, the valid base addresses are multiples of eight: 0x0008, 0x0010, 0x0018, and so on.

Circular Buffer Operation Example 1

Suppose that I0 = 5, M0 = 1, L0 = 3, and the base address = 4. The next address is calculated as:

```
(I0 + M0 - B) mod L0 + B = (5 + 1 - 4) mod 3 + 4 = 6
```

The successive address calculations using I0 for indirect addressing produce the sequence: 5, 6, 4, 5, 6, 4, 5 …. For M0 = –1 (0x3FFF), I0 would produce the sequence: 5, 4, 6, 5, 4, 6, 5, 4 ….

Circular Buffer Operation Example 2

Assume that I0 = 9, M0 = 3, L0 = 5, and the base address = 8. The five-word buffer resides at locations 8 through 12 inclusive. The next address is calculated as:

```
(I0 + M0 - B) mod L0 + B = (9 + 3 - 8) mod 5 + 8 = 12
```

The successive address calculations using I0 for indirect addressing produce the sequence: 9, 12, 10, 8, 11, 9 ... This example highlights the fact that the address sequence does not have to result in a “direct hit” of the buffer boundary.

Bit-Reverse Addressing

The bit-reverse logic is primarily intended for use in FFT computations where inputs are supplied or the outputs generated in bit-reversed order. Bit-reversing is available only on addresses generated by DAG1. The pivot point for the reversal is the midpoint of the 14-bit address, between bits 6 and 7. This is illustrated in the following chart.

	Individual address lines (ADDR$_N$)
Normal Order	13 12 11 10 09 08 07 06 05 04 03 02 01 00
Bit-reversed	00 01 02 03 04 05 06 07 08 09 10 11 12 13

Bit-reversed addressing is a mode, enabled and disabled by setting a mode bit in the mode status register (MSTAT). When enabled, all addresses generated using index registers I0-3 are bit-reversed upon output. (The modified valued stored back after post-update remains in normal order.) This mode continues until the status bit is reset.

It is possible to bit-reverse address values less than 14 bits wide. You must determine the first address and also initialize the M register to be used with a value calculated to modify the I register bit-reversed output to the desired range. This value is:

$$2^{(14 - N)}$$

where N is the number of bits you wish to output reversed. For a complete example of this, refer to Section 6.6.5.2 “Modified Butterfly” in Chapter 6, “One-Dimensional FFTs” of the applications handbook *Digital Signal Processing Applications Using the ADSP-2100 Family (Volume 1)*.

Programming Data Accesses

The ADSP-218x family development software supports the declaration and use of a simple data structure: one-dimensional arrays (or buffers). The array may contain a single value (a variable) or multiple values (an array). In addition, the array may be used as a circular buffer. Here is a brief discussion of each instance, with an example of how they are declared and used in assembly language. Complete syntax for all assembler directives is given in the *Assembler Manual for ADSP-218x & ADSP-219x Family DSPs*.

Variables and Arrays

Arrays are the basic data structure of the ADSP-218x. In our literature, the words "array," "data buffer," and "variable" are used interchangeably. Arrays are declared with assembler directives and can be:

- Referenced indirectly and by name
- Initialized from immediate values in a directive or from external data files
- Linear or circular with automatic wraparound

An array is declared and initialized with a directive such as

```
.VAR coefficients[128] = "filename.dat";
```

This directive declares an array of 128 16-bit values located in Data Memory. The following is an example of the way in which you can reference the array's address and length, respectively:

```
I0=coefficients;      /* point to address of buffer */
L0=0;                 /* set L register to zero */
MX0=DM(I0,M0);        /* load MX0 from buffer */
```

These instructions load a value into MX0 from the beginning of the *coefficients* buffer in Data Memory. With the automatic post-modify of the DAGs, you could execute the second of these instructions in a loop and continuously advance through the buffer.

Alternatively, when you only need to address the first location, you can directly use the buffer name as a label in many circumstances such as

```
MX0=DM(coefficients);
```

The linker substitutes the actual address for the label.

An array or data buffer with a length of one is a simple single-word variable, and is declared in this way:

```
.VAR coefficient;
```

Circular Buffers

A common requirement in DSPs is the circular buffer. The circular buffer is directly implemented by the processors' DAGs, using the L (length) registers. First, you must declare the buffer as circular:

```
.VAR/CIRC coefficients[128];
```

This identifies it to the linker for placement on the proper address boundary. Next, you must initialize the L register and, in the example below, the I register and M register:

```
L0=length (coefficients);   /* length of circular buffer */
I0=coefficients;            /* point to first address of */
                            /* buffer */
M0=1;                       /* increment by 1 location each */
                            /* time */
```

Now a statement like

```
MX0=DM(I0,M0);              /* load MX0 from buffer */
```

placed in a loop, cycles continuously through *coefficients* and wraps around automatically.

PMD-DMD Bus Exchange

The PMD-DMD bus exchange unit couples the Program Memory Data bus and the Data Memory Data bus, allowing them to transfer data between them in both directions. Since the Program Memory Data bus is 24 bits wide, while the Data Memory Data bus is 16 bits wide, only the upper 16 bits of PMD can be directly transferred. An internal register (PX) is loaded with (or supplies) the additional 8 bits. This register can be directly loaded or read when the full 24 bits are required.

Note that when reading data from Program Memory and Data Memory simultaneously, there is a dedicated path from the upper 16 bits of the PMD bus to the Y registers of the computational units. This read-only path does not use the bus exchange circuit; it is the path shown on the individual computational unit block diagrams.

PMD-DMD Bus Exchange Structure

Figure 4-2 shows a block diagram of the PMD-DMD bus exchange. There are two types of connections provided by this circuitry.

The first type of connection is a one-way path from each bus to the other. This is implemented with two tristate buffers connecting the DMD bus with the upper 16 bits of the PMD bus. One of these two buffers is normally used when data is exchanged between the Program Memory and one of the registers connected to the DMD bus. This is the path used to write data to Program Memory; it is not shown in the individual computational unit block diagrams.

The second connection is through the PX register. The PX register is 8-bits wide and can be loaded from either the lower 8 bits of the DMD bus or the lower 8 bits of the PMD bus. Its contents can also be read to the lower 8 bits of either bus.

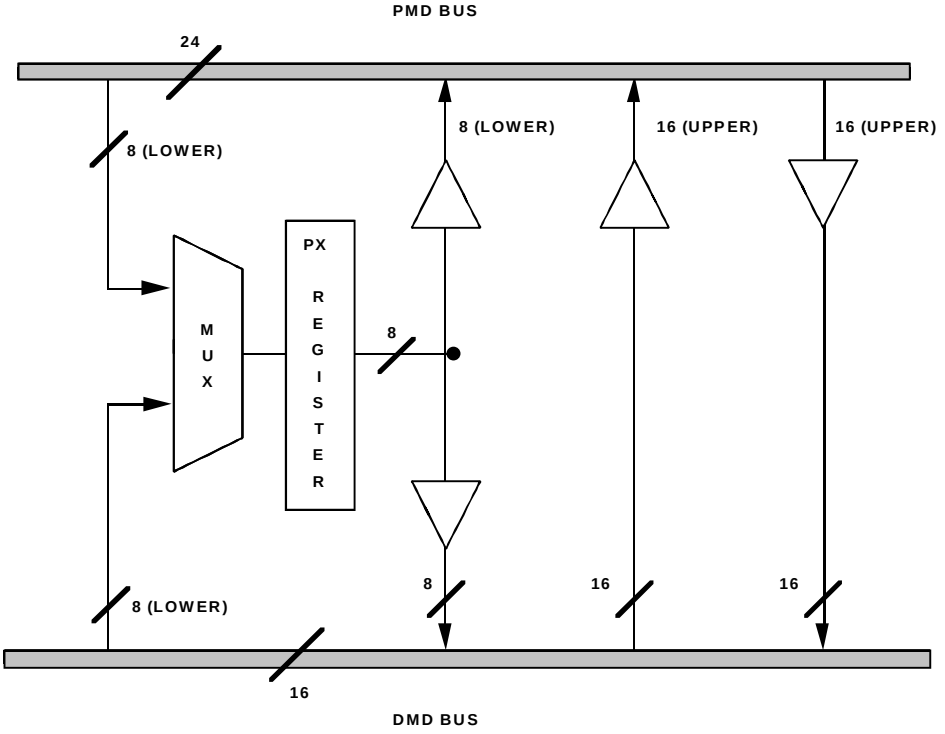

Figure 4-2. PMD-DMD Bus Exchange Block Diagram

From the PMD bus, the PX register is:

1. Loaded automatically whenever data (not an instruction) is read from Program Memory to any register. For example:

   ```
   AX0 = PM(I4,M4);
   ```

   In this example, the upper 16 bits of a 24-bit Program Memory word are loaded into AX0 and the lower 8 bits are automatically loaded into PX.

2. Read out automatically as the lower 8 bits when data is written to Program Memory. For example:

   ```
   PM(I4,M4) = AX0;
   ```

   In this example, the 16 bits of AX0 are stored into the upper 16 bits of a 24-bit Program Memory word. The 8 bits of PX are automatically stored to the 8 lower bits of the memory word.

From the DMD bus, the PX register may be:

1. Loaded with a data move instruction, explicitly specifying the PX register as the destination. The lower 8 bits of the data value are used and the upper 8 are discarded.

   ```
   PX = AX0;
   ```

2. Read with a data move instruction, explicitly specifying the PX register as a source. The upper 8 bits of the value read from the register are all zeroes.

   ```
   AX0 = PX;
   ```

Whenever any register is written out to Program Memory, the source register supplies the upper 16 bits. The contents of the PX register are automatically added as the lower 8 bits. If these lower 8 bits of data to be transferred to Program Memory (through the PMD bus) are important, you should load the PX register from the DMD bus before the Program Memory write operation.

Using DAGs with Hardware Overlays

Special care must be taken by the system programmer when using the Data address generators to access hardware overlay memory regions. The DAGs (as well as the program sequencer) work independently of the value of the PMOVLAY and DMOVLAY registers. Thus, since memory access may not be from the desired target memory overlay region, data corruption or undesired program operation could occur. The following are some examples of instances where special care is required:

- Autobuffering— Since autobuffering works with the current value of the PMOVLAY and DMOVLAY registers only, precautions must be made to ensure that memory is not overwritten by the autobuffering mechanism when performing serial port autobuffering.
- Register Indirect Jumps or Calls—Since DAG register points to the absolute address location of the active Program Memory Overlay region, switching context between Program Memory Overlays before performing a register indirect jump or call may result in undesired program behavior.
- Circular Buffers—Switching between overlay regions when using circular buffering will result in data accesses from the same physical address but from a different overlay region. However, you could use this behavior for a positive purpose: bouncing data back and forth between multiple overlay regions via the DAGs and an overlay paging scheme. (See "Serial Port Autobuffering on the ADSP-2187/2188/2189 Processors" in Chapter 5 "Serial Ports" for more information.)