

# 5 DATA ADDRESS GENERATORS

The Data Address Generators (DAGs) of the ADSP-21535 DSP generate addresses for data moves to and from memory. By generating addresses, the DAGs let programs refer to addresses indirectly, using a DAG register instead of an absolute address.

The DAG architecture, which appears in [Figure 5-1 on page 5-3](#), supports several functions that minimize overhead in data access routines. These functions include:

- **Supply address and post-modify**—provides an address during a data move and auto-increments/decrements the stored address for the next move.
- **Supply address with offset**—provides an address from a base with an offset without incrementing the original address pointer.
- **Modify address**—increments or decrements the stored address without performing a data move.
- **Bit-reversed carry address**—provides a bit-reversed carry address during a data move without reversing the stored address.

The DAG subsystem comprises two DAG Arithmetic units, eight Pointer registers, four Index registers and four complete sets of related Modify, Base and Length registers. These registers hold the values that the DAGs use to generate addresses. The types of registers are:

- **Index registers, I[3:0].** 32-bit Index registers hold an address pointer to memory. For example, the instruction  $R3 = [I0]$  loads the data value found at the memory location pointed to by the register I0. Index registers can be used for 16- and 32-bit memory accesses.
- **Modify registers, M[3:0].** 32-bit Modify registers provide the increment or step size by which an index register is post-modified during a register move.

For example, the  $R0 = [I0 += M1]$  instruction directs the DAG to:

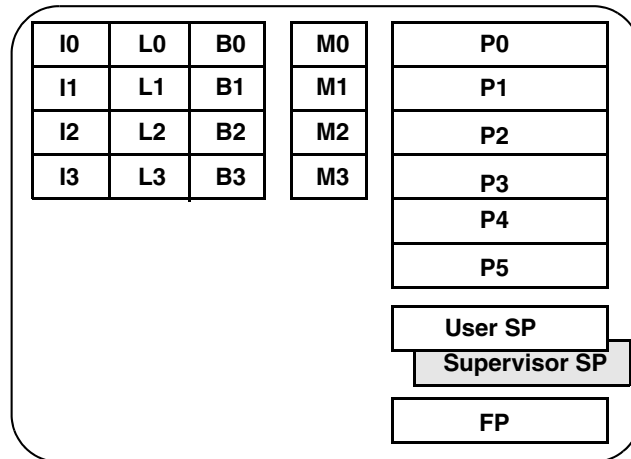
- Output the address in register I0
- Load the contents of the memory location pointed to by I0 into R0
- Then modify the contents of I0 by the value contained in the M1 register.
- **Base and Length registers, B[3:0] and L[3:0].** 32-bit Base and Length registers set up the range of addresses and the starting address of a circular buffer. For more information on circular buffers, see [“Addressing Circular Buffers” on page 5-6](#).
- **Pointer registers, P[5:0], FP, and SP.** 32-bit Pointer registers hold an address pointer to memory. The P[5:0] field, FP (Frame Pointer), and SP (Stack Pointer) can be manipulated and used in various instructions. For example, the instruction  $R3 = [P0]$  loads the register R3 with the data value found at the memory location

pointed to by the register P0. The Pointer registers have no effect on circular buffer addressing. They can be used for 8-, 16-, and 32-bit memory accesses.

- i** Do not assume the L-registers are automatically initialized to zero for linear addressing. The I-, M-, L-, and B-registers contain random values after reset. For each I-register used, programs must initialize the corresponding L-registers to zero for linear addressing or to the buffer length for circular buffer addressing.

All DAG registers must be initialized individually. Initializing a B-register does not automatically initialize the I-register.

### Data Address Generator Registers (DAGs)




 **Supervisor only register. Attempted read or write in User mode causes an exception error.**

Figure 5-1. ADSP-21535 DSP DAG Registers

# Addressing With DAGs

The DAGs can generate an address that is incremented by a value or by a register. In post-modify addressing, the DAG outputs the I-register value unchanged; then the DAG adds an M-register or immediate value to the I-register.

In Indexed addressing, the DAG adds a small offset to the value in the P-register, but does not update the P-register with this new value, thus providing an offset for that particular memory access.

The ADSP-21535 DSP addressing is always byte aligned. Depending on the type of data used, increments and decrements to the DAG registers can be by 1, 2, or 4 to match the 8-bit, 16-bit, or 32-bit accesses.

For example, consider this instruction:

```
R0 = [ P3++ ] ;
```

This instruction fetches a 32-bit word, pointed to by the value in P3, and places it in R0. It then post-increments P3 by *four*, maintaining alignment with the 32-bit access.

```
R0.L = W [ I3++ ] ;
```

This instruction fetches a 16-bit word, pointed to by the value in I3 and places it in the low half of the destination register, R0.L. It then post-increments I3 by *two*, maintaining alignment with the 16-bit access.

```
R0 = B [ P3++ ] (X) ;
```

This instruction fetches an 8-bit word, pointed to by the value in P3 and places it in the destination register, R0. It then post-increments P3 by *one*, maintaining alignment with the 8-bit access. The byte value may be zero-extended or sign-extended into the 32-bit data register.

Instructions using Index registers use an M-register or a small immediate value (+/- 2 or 4) as the modifier. Instructions using Pointer registers use a small immediate value or another P-register as the modifier. For instruction summary details, see [Table 5-3 on page 5-17](#).

## Frame and Stack Pointers

In many respects, the Frame and Stack Pointer registers perform like the other P-registers, P[5:0]. They can act as general pointers in any of the load/store instructions. For example,  $R1 = B[SP](Z)$ . However, FP and SP have additional functionality.

The Stack Pointer registers include:

- a User Stack Pointer (USP in Supervisor mode, SP in User mode)
- a Supervisor Stack Pointer (SP in Supervisor mode)

The User Stack Pointer register and the Supervisor Stack Pointer register are accessed using the register alias SP. Depending on the current processor operating mode, only one of these registers is active and accessible as SP:

- In User mode, any reference to SP (for example, stack pop  $R0 = [SP++]$ ;) implicitly uses the USP as the effective address.
- In Supervisor mode, the same reference to SP (for example,  $R0 = [SP++]$ ;) implicitly uses the Supervisor Stack Pointer as the effective address.

To manipulate the User Stack Pointer for code running in Supervisor mode, use the register alias USP. When in Supervisor mode, a register move from USP (for example,  $R0 = USP$ ;) moves the current User Stack Pointer into R0. The register alias USP can only be used in Supervisor mode.

## Addressing With DAGs

Some load/store instructions use FP and SP exclusively, for example:

- FP-indexed load/store, which extends the addressing range for 16-bit encoded load/stores
- Stack push/pop instructions

## Addressing Circular Buffers

The DAGs support addressing circular buffers—a range of addresses containing data that the DAG steps through repeatedly, wrapping around to repeat stepping through the same range of addresses in a circular pattern.

The DAGs use four types of DAG registers for addressing circular buffers. For circular buffering, the registers operate this way:

- The Index (I) register contains the value that the DAG outputs on the address bus.
- The Modify (M) register contains the post-modify amount (positive or negative) that the DAG adds to the I-register at the end of each memory access.

Any M-register can be used with any I-register. The modify value can also be an immediate value instead of an M-register. The size of the modify value must be less than or equal to the length (L-register) of the circular buffer.

- The Length (L) register sets the size of the circular buffer and the address range through which the DAG circulates the I-register.

L is positive and cannot have a value greater than  $2^{31} - 1$ . If an L-register's value is zero, its circular buffer operation is disabled.

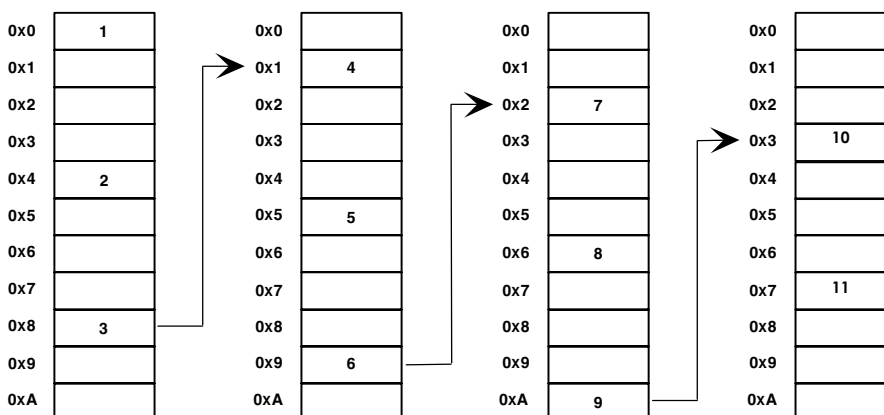
- The Base (B) register or the B-register plus the L-register is the value with which the DAG compares the modified I-register value after each access.

To address a circular buffer, the DAG steps the index pointer (I-register) through the buffer values, post-modifying and updating the index on each access with a positive or negative modify value from the M-register.

If the index pointer falls outside the buffer range, the DAG subtracts the length of the buffer (L-register) from the value or adds the length of the buffer to the value, wrapping the index pointer back to a point inside the buffer.

The starting address that the DAG wraps around is called the buffer's base address (B-register). There are no restrictions on the value of the base address for circular buffers that contains 8-bit data. Circular buffers that contain 16- and 32-bit data must be 16-bit aligned and 32-bit aligned, respectively. Circular buffering uses post-modify addressing.

LENGTH = 11  
BASE ADDRESS = 0x0  
MODIFIER = 4



The columns above show the sequence in order of locations accessed in one pass.  
The sequence repeats on subsequent passes.

Figure 5-2. Circular Data Buffers

## Addressing With DAGs

As seen in [Figure 5-2](#), on the first post-modify access to the buffer, the DAG outputs the I-register value on the address bus, then modifies the address by adding the modify value:

- If the updated index value is within the buffer length, the DAG writes the value to the I-register.
- If the updated index value exceeds the buffer length, the DAG subtracts (for a positive modify value) or adds (for a negative modify value) the L-register value before writing the updated index value to the I-register.

In equation form, these post-modify and wraparound operations work as follows.

- If M is positive:

$$I_{\text{new}} = I_{\text{old}} + M$$

if  $I_{\text{old}} + M < \text{buffer base} + \text{length}$  (end of buffer)

$$I_{\text{new}} = I_{\text{old}} + M - L$$

if  $I_{\text{old}} + M \geq \text{buffer base} + \text{length}$  (end of buffer)

- If M is negative:

$$I_{\text{new}} = I_{\text{old}} + M$$

if  $I_{\text{old}} + M \geq \text{buffer base}$  (start of buffer)

$$I_{\text{new}} = I_{\text{old}} + M + L$$

if  $I_{\text{old}} + M < \text{buffer base}$  (start of buffer)

## Addressing With Bit-reversed Addresses

To obtain results in sequential order, programs need bit-reversed carry addressing for some algorithms, particularly Fast Fourier Transform (FFT) calculations. To satisfy the requirements of these algorithms, the



DAG's bit-reversed addressing feature permits repeatedly subdividing data sequences and storing this data in bit-reversed order. For detailed information about bit-reversed addressing, see the Modify-Increment instruction in the *Blackfin DSP Instruction Set Reference*.

## Indexed Addressing With Index and Pointer Registers

Indexed addressing uses the value in the Index or Pointer register as an effective address. This instruction can load or store 16-bit or 32-bit values. The default is a 32-bit transfer. If a 16-bit transfer is required, then the *W* designator is used to preface the load or store.

For example:

```
R0 = [ I2 ] ;
```

loads a 32-bit value from an address pointed to by I2 and stores it in the destination register R0.

```
R0.H = W [ I2 ] ;
```

loads a 16-bit value from an address pointed to by I2 and stores it in the 16-bit destination register R0.H.

```
[ P1 ] = R0 ;
```

is an example of a 32-bit store operation.

Pointer registers can be used for 8-bit loads and stores.

For example:

```
B [ P1++ ] = R0 ;
```

stores the 8-bit value from the R0 register in the address pointed to by the P1 register, then increments the P1 register.

### Auto-Increment and Auto-Decrement Addressing

Auto-increment addressing updates the Pointer and Index registers after the access. The amount of increment depends on the word size. An access of 32-bit words results in an update of the pointer by 4. A 16-bit word access updates the pointer by 2, and an access of an 8-bit word updates the pointer by 1. Both 8-bit and 16-bit read operations may specify either to sign-extend or zero-extend the contents into the destination register. Pointer registers may be used for 8-, 16-, and 32-bit accesses while Index registers may be used only for 16- and 32-bit accesses.

For example:

```
R0 = W [ P1++ ] (Z) ;
```

loads a 16-bit word into a 32-bit destination register from an address pointed to by the P1 Pointer register. The Pointer is then incremented by 2 and the word is zero-extended to fill the 32-bit destination register.

Auto-decrement works the same way by decrementing the address after the access.

For example:

```
R0 = [ I2-- ] ;
```

loads a 32-bit value into the destination register and decrements the Index register by 4.

### Pre-Modify Stack Pointer Addressing

The only pre-modify instruction in the ADSP-21535 processor uses the Stack Pointer register, SP. The address in SP is decremented by four and then used as an effective address for the store. The instruction `[ --SP ] = R0 ;` is used for stack push operations and can support only a 32-bit word transfer.

## Indexed Addressing with Immediate Offset

Indexed addressing allows programs to obtain values from data tables, with reference to the base of that table. The Pointer register is modified by the immediate field and then used as the effective address. The value of the Pointer register is not updated.

For example:

```
P5 = [ P1 + 0x10 ] ;
```

is an acceptable offset, but

```
P5 = [ P1 + 0x11 ] ;
```

causes an alignment exception.



Be sure the offset is divisible by the word-transfer size, measured in bytes; for example, for a 32-bit transfer, the offset should be a multiple of 4; for a 16-bit transfer, the offset should be a multiple of 2. Correct offsets ensure memory alignment.

## Post-Modify Addressing

Post-modify addressing uses the value in the Index or Pointer registers as the effective address and then modifies it by the contents of another register. Pointer registers are modified by another Pointer register. Index registers are modified by a Modify register. This instruction does not support the Pointer registers as a destination register, nor does it support byte-addressing.

For example:

```
R5 = [ P1++P2 ] ;
```

loads a 32-bit value into the R5 register, found in the memory location pointed to by the P1 register.

## Modifying DAG and Pointer Registers

The value in the P2 register is then added to the value in the P1 register.

For example:

```
R2 = W [ P4++P5 ] (Z) ;
```

loads a 16-bit word into the low half of the destination register R2 and zero-extends it to 32-bits. It adds the value the pointer P4 by the value of the pointer P5.

For example:

```
R2 = [ I2++M1 ] ;
```

loads a 32-bit word into the destination register R2. It updates the value in the Index register I2 by the value in the Modify register M1.

## Modifying DAG and Pointer Registers

The DAGs support an operation that modifies an address value in an index register without outputting an address. The operation, address-modify, is useful for maintaining pointers.

The instruction modifies addresses in any DAG Index and Pointer register (I[3:0], P[5:0], FP, SP) without accessing memory. If the Index register's corresponding B- and L-registers are set up for circular buffering, the instruction performs the specified buffer wraparound (if needed).

The syntax is similar to post-modify addressing (index += modifier). For Index registers, an M-register is used as the modifier. For Pointer registers, another P-register is used as the modifier.

Consider the example, I1 += M2 ;

This instruction adds M2 to I1 and updates I1 with the new value.

## Memory Address Alignment

The ADSP-21535 processor requires proper memory alignment to be maintained for the data size being accessed. Unless exceptions are disabled, violations of memory alignment cause an alignment exception. Some instructions—for example, many of the Video ALU instructions—automatically disable alignment exceptions because the data may not be properly aligned when stored in memory. Alignment exceptions may be disabled by issuing the `DISALGNEXPT` instruction in parallel with a load/store operation.

Normally, the memory system requires two address alignments:

- 32-bit word load/stores are accessed on four-byte boundaries, meaning the two least significant bits of the address are `b#00`.
- 16-bit word load/stores are accessed on two-byte boundaries, meaning the least significant bit of the address must be `b#0`.

[Table 5-1 on page 5-14](#) summarizes the types of transfers and transfer sizes that the addressing modes support.



Be careful when using the `DISALGNEXPT` instruction, because it disables automatic detection of memory alignment errors.

## Memory Address Alignment

Table 5-1. Types of Transfers Supported and Transfer Sizes

Addressing Mode	Types of Transfers Supported	Transfer Sizes
Auto-increment Auto-decrement Indirect Indexed	To and from Data Registers	LOADS: 32-bit word 16-bit, zero-extended half word 16-bit, sign-extended half word 8-bit, zero-extended byte 8-bit, sign-extended byte STORES: 32-bit word 16-bit half word 8-bit byte
	To and from Pointer Registers	LOAD: 32-bit word STORE: 32-bit word
Post-increment	To and from Data Registers	LOADS: 32-bit word 16-bit half word to Data Register high half 16-bit half word to Data Register low half 16-bit, zero-extended half word 16-bit, sign-extended half word STORES: 32-bit word 16-bit half word from Data Register high half 16-bit half word from Data Register low half

Table 5-2 summarizes the addressing modes. In the table, an asterisk (\*) indicates that the ADSP-21535 DSP supports the addressing mode.

Table 5-2. Addressing Modes

	P Auto- inc	P Auto- dec	P Indirect	P Indexed	FP- indexed	P Post- inc	I Auto- inc	I Auto- dec	I Indi- rect	I Post-inc
	[P0++]	[P0--]	[P0]	[P0+im]	[FP+im]	[P0++P1]	[I0++]	[I0--]	[I0]	[I0++M0]
32-bit Word	*	*	*	*	*	*	*	*	*	*
16-bit Half Word	*	*	*	*		*	*	*	*	
8-bit Byte	*	*	*	*						
Sign/ Zero Extend	*	*	*	*		*				
Data Register	*	*	*	*	*	*	*	*	*	*
Pointer Register	*	*	*	*	*					
Data Register Half			*			*	*	*	*	

# DAG Instruction Summary

Table 5-3 lists the DAG instructions. For more information on assembly language syntax, see the *Blackfin DSP Instruction Set Reference*. In the table, note the meaning of these symbols:

- **Dreg** denotes any Data Register File register.
- **Dreg\_lo** denotes the lower 16 bits of any Data Register File register.
- **Dreg\_hi** denotes the upper 16 bits of any Data Register File register.
- **Preg** denotes any Pointer register, FP or SP register.
- **Ireg** denotes any DAG Index register.
- **Mreg** denotes any DAG Modify register.
- **W** denotes a 16-bit wide value.
- **B** denotes an 8-bit wide value.
- **immA** denotes a signed, A-bits wide, immediate value.
- **uimmAmB** denotes an unsigned, A-bits wide, immediate value that is an even multiple of B.
- **Z** denotes the zero-extension qualifier.
- **X** denotes the sign-extension qualifier.
- **BREV** denotes the bit-reversal qualifier.

The *Blackfin DSP Instruction Set Reference* more fully describes the options that may be applied to these instructions and the sizes of immediate fields.



DAG instructions do not affect the `ASTAT` Status flags.

Table 5-3. DAG Instruction Summary

Instruction
<code>Preg = [ Preg ] ;</code>
<code>Preg = [ Preg ++ ] ;</code>
<code>Preg = [ Preg -- ] ;</code>
<code>Preg = [ Preg + uimm6m4 ] ;</code>
<code>Preg = [ Preg + uimm17m4 ] ;</code>
<code>Preg = [ Preg – uimm17m4 ] ;</code>
<code>Preg = [ FP – uimm7m4 ] ;</code>
<code>Dreg = [ Preg ] ;</code>
<code>Dreg = [ Preg ++ ] ;</code>
<code>Dreg = [ Preg -- ] ;</code>
<code>Dreg = [ Preg + uimm6m4 ] ;</code>
<code>Dreg = [ Preg + uimm17m4 ] ;</code>
<code>Dreg = [ Preg – uimm17m4 ] ;</code>
<code>Dreg = [ Preg ++ Preg ] ;</code>
<code>Dreg = [ FP – uimm7m4 ] ;</code>
<code>Dreg = [ Ireg ] ;</code>
<code>Dreg = [ Ireg ++ ] ;</code>
<code>Dreg = [ Ireg -- ] ;</code>
<code>Dreg = [ Ireg ++ Mreg ] ;</code>
<code>Dreg =W [ Preg ] (Z) ;</code>
<code>Dreg =W [ Preg ++ ] (Z) ;</code>
<code>Dreg =W [ Preg -- ] (Z) ;</code>
<code>Dreg =W [ Preg + uimm5m2 ] (Z) ;</code>

## DAG Instruction Summary

Table 5-3. DAG Instruction Summary (Cont'd)

Instruction
$\text{Dreg} = \text{W} [ \text{Preg} + \text{uimm16m2} ] (Z) ;$
$\text{Dreg} = \text{W} [ \text{Preg} - \text{uimm16m2} ] (Z) ;$
$\text{Dreg} = \text{W} [ \text{Preg} ++ \text{Preg} ] (Z) ;$
$\text{Dreg} = \text{W} [ \text{Preg} ] (X) ;$
$\text{Dreg} = \text{W} [ \text{Preg} ++ ] (X) ;$
$\text{Dreg} = \text{W} [ \text{Preg} -- ] (X) ;$
$\text{Dreg} = \text{W} [ \text{Preg} + \text{uimm5m2} ] (X) ;$
$\text{Dreg} = \text{W} [ \text{Preg} + \text{uimm16m2} ] (X) ;$
$\text{Dreg} = \text{W} [ \text{Preg} - \text{uimm16m2} ] (X) ;$
$\text{Dreg} = \text{W} [ \text{Preg} ++ \text{Preg} ] (X) ;$
$\text{Dreg\_hi} = \text{W} [ \text{Ireg} ] ;$
$\text{Dreg\_hi} = \text{W} [ \text{Ireg} ++ ] ;$
$\text{Dreg\_hi} = \text{W} [ \text{Ireg} -- ] ;$
$\text{Dreg\_hi} = \text{W} [ \text{Preg} ] ;$
$\text{Dreg\_hi} = \text{W} [ \text{Preg} ++ \text{Preg} ] ;$
$\text{Dreg\_lo} = \text{W} [ \text{Ireg} ] ;$
$\text{Dreg\_lo} = \text{W} [ \text{Ireg} ++ ] ;$
$\text{Dreg\_lo} = \text{W} [ \text{Ireg} -- ] ;$
$\text{Dreg\_lo} = \text{W} [ \text{Preg} ] ;$
$\text{Dreg\_lo} = \text{W} [ \text{Preg} ++ \text{Preg} ] ;$
$\text{Dreg} = \text{B} [ \text{Preg} ] (Z) ;$
$\text{Dreg} = \text{B} [ \text{Preg} ++ ] (Z) ;$
$\text{Dreg} = \text{B} [ \text{Preg} -- ] (Z) ;$
$\text{Dreg} = \text{B} [ \text{Preg} + \text{uimm15} ] (Z) ;$

Table 5-3. DAG Instruction Summary (Cont'd)

Instruction
$Dreg = B [ Preg - uimm15 ] (Z) ;$
$Dreg = B [ Preg ] (X) ;$
$Dreg = B [ Preg ++ ] (X) ;$
$Dreg = B [ Preg -- ] (X) ;$
$Dreg = B [ Preg + uimm15 ] (X) ;$
$Dreg = B [ Preg - uimm15 ] (X) ;$
$[ Preg ] = Preg ;$
$[ Preg ++ ] = Preg ;$
$[ Preg -- ] = Preg ;$
$[ Preg + uimm6m4 ] = Preg ;$
$[ Preg + uimm17m4 ] = Preg ;$
$[ Preg - uimm17m4 ] = Preg ;$
$[ FP - uimm7m4 ] = Preg ;$
$[ Preg ] = Dreg ;$
$[ Preg ++ ] = Dreg ;$
$[ Preg -- ] = Dreg ;$
$[ Preg + uimm6m4 ] = Dreg ;$
$[ Preg + uimm17m4 ] = Dreg ;$
$[ Preg - uimm17m4 ] = Dreg ;$
$[ Preg ++ Preg ] = Dreg ;$
$[ FP - uimm7m4 ] = Dreg ;$
$[ Ireg ] = Dreg ;$
$[ Ireg ++ ] = Dreg ;$
$[ Ireg -- ] = Dreg ;$

## DAG Instruction Summary

Table 5-3. DAG Instruction Summary (Cont'd)

Instruction
[ Ireg ++ Mreg ] = Dreg ;
W [ Ireg ] = Dreg_hi ;
W [ Ireg ++ ] = Dreg_hi ;
W [ Ireg -- ] = Dreg_hi ;
W [ Preg ] = Dreg_hi ;
W [ Preg ++ Preg ] = Dreg_hi ;
W [ Ireg ] = Dreg_lo ;
W [ Ireg ++ ] = Dreg_lo ;
W [ Ireg -- ] = Dreg_lo ;
W [ Preg ] = Dreg_lo ;
W [ Preg ] = Dreg ;
W [ Preg ++ ] = Dreg ;
W [ Preg -- ] = Dreg ;
W [ Preg + uimm5m2 ] = Dreg ;
W [ Preg + uimm16m2 ] = Dreg ;
W [ Preg – uimm16m2 ] = Dreg ;
W [ Preg ++ Preg ] = Dreg_lo ;
B [ Preg ] = Dreg ;
B [ Preg ++ ] = Dreg ;
B [ Preg -- ] = Dreg ;
B [ Preg + uimm15 ] = Dreg ;
B [ Preg – uimm15 ] = Dreg ;
Preg = imm7 (X) ;
Preg = imm16 (X) ;

Table 5-3. DAG Instruction Summary (Cont'd)

Instruction
Preg += Preg (BREV) ;
Ireg += Mreg (BREV) ;
Preg -= Preg ;
Ireg -= Mreg ;

## DAG Instruction Summary